International Journal of Software Engineering and Knowledge Engineering

DockerPedia: A Knowledge Graph of Software Images and their Metadata

Maximiliano Osorio, Carlos Buil-Aranda

Departamento de Informática, Universidad Técnica Federico Santa María, Avenida España 1680, Valparaiso, Chile
mosorio@inf.utfsm.cl
cbuil@inf.utfsm.cl

Idafen Santana-Perez

Departamento de Señales y Comunicaciones, Universidad de Las Palmas de Gran Canaria, España
idafen.santana@ulpgc.es

Daniel Garijo

Ontology Engineering Group, Universidad Politécnica de Madrid, España
daniel.garijo@upm.es

An increasing amount of researchers use software images to capture the requirements and code dependencies needed to carry out computational experiments. Software images preserve the computational environment required to execute a scientific experiment and have become a crucial asset for reproducibility. However, software images are usually not properly documented and described, making it challenging for scientists to find, reuse and understand them. In this paper we propose a framework for automatically describing software images in a machine-readable manner by i) creating a vocabulary to describe software images; ii) developing an annotation framework designed to automatically document the underlying environment of software images and iii) creating DockerPedia, a Knowledge Graph with over 150,000 annotated software images, automatically described using our framework. We illustrate the usefulness of our approach in finding images with specific software dependencies, comparing similar software images, addressing versioning problems when running computational experiments; and flagging problems with vulnerable software dependencies.

Keywords: software container; software image; knowledge graph; software metadata; Docker; experiment reproducibility; ontology

1. Introduction

An increasing amount of researchers use software to carry out their computational experiments in multiple domains, including Physics [1, 2, 3], Biology [4, 5] or Geosciences [6]. Software developed to this end range from simple data transformation and visualization scripts to software libraries or scientific workflows, which may stitch together complex computational pipelines.

Software has become crucial for reproducing scientific results, and therefore, it is necessary to properly document, archive and preserve it for reuse [7]. One critical aspect of a software component is its computational environment, which defines all the component dependencies as well as the required Operative System needed for the component to run in different computational infrastructures.

Nowadays *software images* have become the most commons means to capture the computational environment of software components [8]; and repositories such as DockerHub [9] allow researchers to upload and share their software images with others. These efforts have accelerated the adoption and reusability of software, but software images are often not properly described, making them black boxes that are hard to inspect, find, compare, understand or reuse by others.

Since the FAIR principles were proposed in 2016 for Finding, Accessing, Interoperating and Reusing data (FAIR) [10], the scientific community has applied them to other aspects of research, such as scientific workflows [11] or research software [12] to enable Open Science and reproducibility. However, the description of the computational environment used in scientific experiments has received little attention so far.

In this paper we address this issue by presenting a framework for automatically creating Knowledge Graphs of software images and their metadata. Our contributions include:

- A vocabulary [13] to describe software image contents and metadata.
- An annotation tool [14] that automatically describes software images in a machine-readable manner, including all its software dependencies and references to other software images.
- DockerPedia [15], a public Knowledge Graph containing over 150,000 software images annotated using our approach.

Our approach relies on Docker [16], a popular software container system that allows assembling a computational environment including all necessary dependencies, e.g., libraries, configuration, code and data needed, among others. We illustrate the benefits of our approach in four different use cases, in which we inspect the contents of an image, retrieve images with specific software dependencies or compare different computational environments. All resources derived from this paper are available online with an open license (`https://w3id.org/dgarijo/ro/dockerpedia`).

The rest of the paper is structured as follows: in Section 2 we introduce the main technologies used in this work: software virtualization, Docker and its Hub for storing software images as well as the challenges for making software images *FAIR*. In Section 3 we describe how we created a DockerPedia, including an overview of the ontology used to describe images, and the software we developed for annotating them. In Section 4 we validate our approach with four different use cases; in Section 5 we describe related work and we conclude the paper in Section 6.

2. Background

There are two main approaches for preserving computational environments. On the one hand, *virtual machines* (VMs) emulate a full computing system, including its Operating System (OS) and underlying architecture. VMs have been used by researchers to capture computational environments [17, 18], and have several advantages for reproducibility, such as their support for legacy or discontinued applications and OS [19]. However, VMs quickly become of significant size, and are less efficient than running a component directly in a physical machine.

On the other hand, *software containers* provide a lightweight encapsulation by virtualizing only the OS and software dependencies of the target application. Software containers are composed of *images*, immutable files which store the dependencies and source code needed by an application to run. Software images are only executable in containers, which share their resources with the host machine, and therefore are smaller in size than VMs while preserving a native performance when running applications. Due to the flexibility of software containers, the community has quickly adopted them in a variety of domains such as Bioinformatics [20], Genomics [21], Web Engineering [8] or Archaelogy [22]; and thus it is the approach we have used in this work. As of today, Docker has become the *de-facto* tool for representing, executing and sharing software containers.

2.1. *Software Containers and Images in Docker*

In Docker, software images are defined using Dockerfiles [23], text files that describe the OS and all the commands necessary to build an image and run it as a container. Each image is itself an extensible template that can be built on top of other images. In fact, the first line of a Dockerfile starts with the `FROM` keyword, followed by the name of the image to use as reference, i.e., its *parent image*. Most Docker images are built from a parent image, but it's also possible to build one from scratch.

The rest of the lines in a Dockerfile define the commands to be added on top of the parent image. Each of these commands are translated into an *image layer*, which represents a delta over the parent image. When building an image, all commands are executed sequentially, creating one image layer at a time. When an image is updated or rebuilt, only the modified layers are updated. Listing 1 shows a sample Dockerfile where several Python images as well as the Pegasus workflow system [24] are installed using the *debian* parent image (version 9).

```
FROM debian:9
#install dependencies
RUN apt-get install -y \
        python \
        python-astropy \
        ...
#install pegasus
```

```
RUN apt-get install -y pegasus
```

Listing 1: Sample Dockerfile. For simplicity, some instructions have been summarized with three dots (...).

Dockerfiles act as recipes with a flexible and extensible mechanism that makes it easy to build on top of existing work. However, Dockerfiles are often **not easy to follow**, and in many cases **do not clarify which specific packages are being deployed by each command**. For example, the previous listing does not specify the software versions of libraries such as `python` or `python-astropy`.

In addition, **dependencies installed in parent images are not specified in the Dockerfile itself**, which makes it even more complicated to identify the total number of software dependencies installed in a given container. In Listing 1, `debian:9` may have had another version of python installed, or depend itself on a parent image with additional installed dependencies.

2.2. *The Docker Hub Image Registry*

The Docker community has created Docker Hub, a public online registry to store public Docker images. Docker Hub contains thousands of images, and allows downloading and deploying Docker images locally, running containers in a host environment and then executing the software inside the image.

However, Docker Hub does **not control which packages are in the stored images**, nor whether an image will work as intended. In many cases, the Dockerfile used to create an image may not be available, hence making **images work as black boxes where users have no knowledge of the packages installed within**. This makes it difficult to **compare images against each other, and find images with a set of dependencies already installed**.

2.3. *Challenges in Making Software Images FAIR*

Software containers have demonstrated to be a useful tool for the community to *access* the work from others in an *interoperable* manner, proposing a flexible mechanism that makes software images extremely easy to *reuse* by others. However, the following challenges remain to be addressed:

- **Image exploration**: While Docker Hub is a registry for storing images, image contents are obfuscated to users, making it very difficult to search for images with specific software dependencies.
- **Dependency versioning**: Even when a Dockerfile is provided, it is challenging to determine the version of the software dependencies that are installed, as it is often not specified. This could have catastrophic consequences, for example, if some of the installed dependencies have vulnerabilities that have been fixed in more recent versions. This issue also raises problems when reproducing scientific results, as the same Dockerfile could

lead to different software images, depending on the moment when they were built.
- **Image reuse and comparison**: The obfuscation of the dependencies available in a software image makes it very complicated to reuse and compare them, or find commonalities among them.

In this paper we address these challenges by analyzing the packages installed in a software image; and transforming them into a machine-readable representation that describes and links different software images and their layers. With our work, software images no longer become black boxes that are difficult to explore, inspect and compare. In the following sections we describe our approach.

3. Creating a Knowledge Graph of Software Images and their Metadata

We capture the contents and context of a software image as a Knowledge Graph (KG) [25], where the nodes represent software images and their constituents; and the directed edges represent the relationship between them. For example, if a software image was created from a particular Dockerfile, both the Dockerfile and the image would become nodes in our KG, linked by the relationship *builds* (as in Dockerfile builds Docker image).

Our approach takes as input a software image generated by a Dockerfile, identified by its repository name in Docker Hub, scans it to detect all its image layers and software dependencies (including their versions); and automatically annotates them in using the Resource Description Framework (RDF) [26]. Since different images may have common image layers, our annotation process analyzes each layer of the image and is able to reuse and link previous annotations.

Our framework is composed of three main parts: the ontology we use to organize the KG (Section 3.1), the annotation tool we use to scan software images and produce RDF (Section 3.2); and DockerPedia, a KG created after analyzing thousands of images from Docker Hub (Section 3.3).

3.1. *An Ontology to Represent Software Images*

Figure 1 shows an overview of the DockerPedia ontology, designed to represent software images. A `dpv:Dockerfile` is used to build a `dpv:SoftwareImage`, which itself is decomposed in `dpv:ImageLayer`s. Since we aim to track the particular versions of all packages installed in an image, we distinguish the concept `dpv:SoftwarePackage` and its respective `dpv:PackageVersion`s. We also include the representation of `dpv:SoftwareVulnerabilities`, which are crucial to create alerts when comparing or reusing software images.

We build on the Workflow Infrastructure Conservation Using Semantics ontology

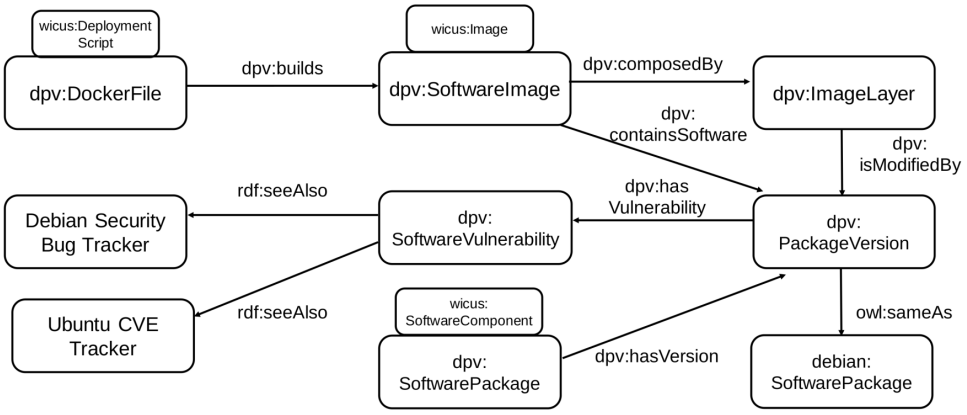


Fig. 1: Overview of the DockerPedia ontology to represent software images, their contents and metadata.

(WICUS[a]) [27], an OWL ontology network that represents the main concepts of a computational infrastructure. WICUS contains terminology for describing computational scientific experiments, including their software, hardware and computing resources; and we align our terminology to existing classes and properties. A `dpv:Dockerfile` is defined as subclass of the more generic `wicus:DeploymentScript`, as each Dockerfile contains the set of actionable steps to be carried out during the deployment of the container.

We import other classes and properties from WICUS ontology: the `wicus:DeploymentPlan`, `wicus:DeploymentStep`, `wicus:ConfigurationInfo` and `wicus:ConfigurationParameter` classes describe the steps to deploy and configure a software component; and the classes `wicus:SoftwareStack` and `wicus:SoftwareComponent` represent software elements that must be installed and their dependencies. These classes allow describing how a software component is deployed (instance of `wicus:DeploymentPlan`), which configuration it needs and its dependencies (which extend `wicus:ConfigurationInfo` and a list of `wicus:ConfigurationParameter`).

We define `dpv:SoftwarePackage` as a subclass of `wicus:SoftwareComponent` in order to define the software packages installed by the underlying Operative System.

Every `wicus:SoftwareComponent` has a `dpv:hasVersion` relation to denote the package version which was installed within the container. We annotate every line from the Dockerfile listed each repository as a `wicus:DeploymentStep` and the Dockerfile as a `wicus:DeploymentPlan`. In summary, we annotate all installed software packages in the container file system, not only those packages in the Dockerfile.

Each Dockerfile is related to their image by means of the `dpv:builds` property.

[a]with prefix `wicus` and namespace URI `http://purl.org/net/wicus`

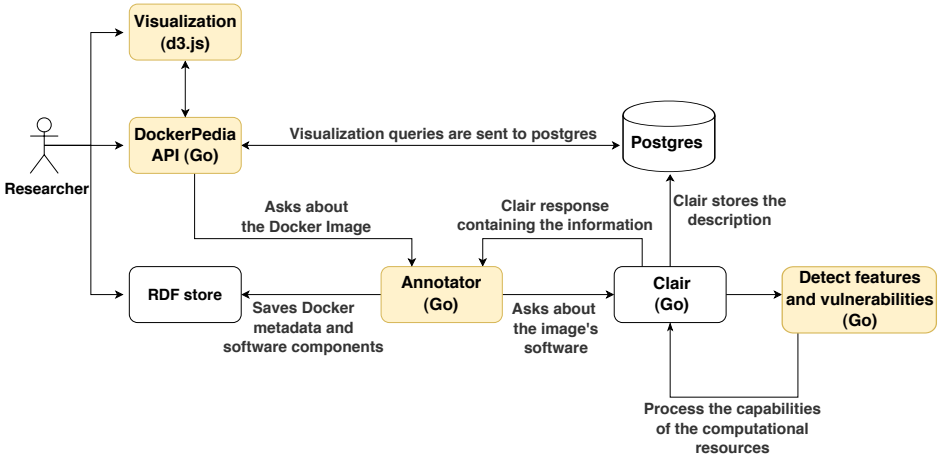


Fig. 2: General architecture of DockerPedia, with our main contributions highlighted in yellow. The annotation service provides an API which receives as input a Docker image URL from Docker Hub. The annotation service describes that image using the ontology depicted in Figure 1 and searches for software vulnerabilities. The system also provides a visualization tool for the components found within a Docker image.

Images are composed by layers, being related by the `dpv:composedBy` property. This way we can link each Dockerfile to the resulting layers, which in turn are linked to the package versions used by them via the and `dpv:isModifiedBy` property.

3.2. *Software Image Annotator*

Figure 2 shows an overview of the architecture used for populating and browsing the contents of DockerPedia. The software annotation service (in the bottom center of the figure) implements a REST interface which receives as input a Docker image, and produces an annotated RDF entry as a result. The annotation service first downloads the target Docker image and mounts it (without running it) to extract basic metadata about it (explained in Section 3.2.1). Next, the service scans the image searching for all the software packages installed, capturing their versions. The results are finally converted to RDF, populating DockerPedia and linking the packages and dependencies found to external RDF resources like the Debian package repository and the Common Vulnerabilities and Exposures database. The code used for annotating all images is available online [14].

3.2.1. *Annotating Docker files*

Docker builds an image by either reading a set of instructions from a Dockerfile or by deploying that image on a host (in case the Dockerfile is not present). Unfortunately, many images lack a Dockerfile, and in those cases, the only means to retrieve the

basic metadata from the image is by reading its manifest file (available inside the image itself). The manifest file contains the image's internal configuration and the set of layers from which the image is built.

Following the official image manifest schema documentation [28], we extract the most important attributes of the manifest file, which are:

Name: name of the image's repository

History: the list of the layers composing the current image layer. This field contains its ID and its parent layers ID's, containing the history of how the current image is constructed. More in detail, for each layer the history field contains:

Id: the layer's ID;
Parent: *string* of the parent's ID;
ContainerConfig: the layer's build command;
Author: the author's name and email.

Thanks to the information provided by either the Dockerfile or the manifest file, we are able to describe a Docker image just by deploying it, without modifying any parameter in the image.

3.2.2. *Software image analysis*

Each Docker image layer installs software packages that we need to capture to describe images accurately. Software packages are usually installed through package managers, which automate the process of installing, upgrading, configuring, and removing software components on an image layer. Package managers can operate at a system level (e.g., `apt` in Debian distribution, `yum` in RedHat distributions); or be customized for a given language (e.g., `pip` for Python, `npm` for Javascript, etc.)

A common approach for finding which components are installed is to search for the lines that use the package manager in a Dockerfile. For example, Listing 2 shows the commands to install some of the packages needed by the TensorFlow package.

```
apt install -y --no-install-recommends \
        build-essential curl libfreetype6-dev \
        libhdf5-serial-dev libpng12-dev \
        libzmq3-dev pkg-config \
        python python-dev rsync \
        software-properties-common unzip
pip install --upgrade tensorflow
```

Listing 2: Command to install packages needed by TensorFlow in Ubuntu

The main problem of this approach is that there may not be information about the version of the software packages or software dependencies installed. For example, Listing 2 installs 184 additional packages which we may not be aware of.

In order to address this issue we use Clair [29], an open-source tool designed for analyzing vulnerabilities in Docker containers, to identify which packages are

installed within an image. Clair has been primarily used to scan images private container registries such as Quay.io [30], but it can also be used to analyze images from Docker Hub.

Given a Docker image id as input, Clair downloads, mounts and analyzes all of its layers, determining the Operating System of each layer automatically. As a result of the analysis, Clair provides a list of the installed packages in the image and its associated layers, along with the relationships between different layers. Clair is compatible with the most common system package managers in distributions such as Ubuntu, Debian, Alpine, RedHat, CentOS, and Oracle, allowing our implementation to analyze images running on any of them. In addition, we extended Clair to detect Conda [31] packages and dependencies for two reasons: 1) Conda's popularity among the scientific community and 2) to showcase the extensibility of our approach.

3.3. *Generating DockerPedia*

We developed DockerPedia following the best practices described in [10]. Hence, all URIs share the following structure:

{*Service_URI*}/*resource*/{*class*}/{*instance_id*}

Where *Service_URI* is the URI corresponding to the server where we are hosting DockerPedia (in this case, `http://dockerpedia.inf.utfsm.cl/`); and *class* represents the main concept to which the instance with id *instance_id* belongs. For example, the URI corresponding to the software package *apt* is `https://dockerpedia.inf.utfsm.cl/resource/SoftwarePackage/apt`.

We populated the KG by analyzing images from the Docker Hub online repository. We performed a search over the Docker Hub free text box to obtain all available images, using a permutation of two-character keywords (1296 different permutations in total). In February 2018, this search returned 1,363,510 Docker repositories and 4,608,443 images composed of 4,593,602 community images and 14,841 official images. Since the total size of these images was 53.47 PB, we selected the 150,000 most downloaded Docker images from Docker Hub. For our analysis we used 7 virtual machines with following specifications: 2 CPU (2.20GHz) and 3 GB of memory from Digital Ocean [32]. The process took around two months (the final time may vary due to network conditions and the settings of the processed images).

Once we obtained the name of the images to analyze, we retrieved basic metadata from Docker Hub, including username, repository name, image description, whether the image is an automated build or not, when the image was updated for the last time, number of pulls and the number of stars of the image. We also obtained all versions of each image (also known as "tags" in Docker Hub). For instance, the Docker repository "google/cadvisor" has 59 different tags of the original software, and each image has different packages. For each tag we obtained its name, date of last update and its size. Then, we used the DockerPedia annotator to create the ontology instances, as described in Section 3.2.

We also took advantage of Clair's ability to identify vulnerabilities from the

Category	Number
Repositories	1,363,628
Software images	157,632
Packages	44,194
Links to Debian	13,136
Triples	142,873,563

Table 1: Number of Docker images and repositories analyzed in DockerPedia

Prefix	URI
dpv	http://dockerpedia.inf.utfsm.cl/vocab#
rdfs	http://www.w3.org/2000/01/rdf-schema#
dpsi	http://dockerpedia.inf.utfsm.cl/resource/SoftwareImage/
dppv	http://dockerpedia.inf.utfsm.cl/resource/PackageVersion/
dpos	http://dockerpedia.inf.utfsm.cl/resource/OperatingSystem/

Table 2: Prefixes used in the paper

analyzed packages and annotated them in the Dockerpedia KG. We linked these vulnerabilities by searching in the Common Vulnerabilities and Exposures (CVE) vulnerability database [33], the Ubuntu CVE Tracker [34], the Debian Security Bug Tracker [35], the Red Hat Security Data [36] and the Alpine Sec DB [37].

In total, we analyzed 1,363,628 Docker repositories, annotating 157,632 images as presented in Table 1. The annotation process took around a month, and the resultant dataset is available online [38].

4. DockerPedia: Validation and Exploitation

We demonstrate the usability and usefulness of DockerPedia through four common use cases, which range from the ability to find images with specific requirements, to comparing image execution environments to understand their similarities and differences. We showcase these use cases with real software images that are part of the KG. All illustrative examples are available online,[b] as well as the result of all the queries listed in this paper [39]. All prefixes used throughout the rest of the paper can be seen in Table 2.

4.1. *Exploring the contents of a software image*

The simplest use case starts by digging into the contents of an image, provided we know its id and tag (to identify its version). For example, Listing 3 describes one of the available images for Pegasus [24], a scientific workflow system. The selected

[b]Due to the size of the graph, only some examples are currently loaded in the SPARQL endpoint. Examples are available at `https://dockerpedia.inf.utfsm.cl/examples`

image has the identifier *dockerpedia/pegasus_workflow_images*; and the tag *pegasus-4.8.5*.

The results of this query return basic metadata of the image like its size and the list of layers and packages installed in it.

```
SELECT ?i ?p ?o WHERE {
 ?i a dpv:SoftwareImage;
    rdfs:label "dockerpedia/pegasus_workflow_images";
    dpv:tag "pegasus-4.8.5";
    ?p ?o .
}
```

Listing 3: Retrieving all metadata of the image with the id *dockerpedia/pegasus_workflow_images* and tag *pegasus-4.8.5*

4.2. *Finding images containing specific software dependencies*

With our approach, we can easily retrieve all the images containing a given software package. This is useful when looking for existing images of specific tools and when defining an experimental infrastructure (for which dependency conflicts are key).

For our use case, we focus on retrieving all available images containing the *dash* package [40], a visualization software commonly used in Python data science applications. Listing 4 specifies the required query, retrieving also the version and Operating System of the images where *dash* is installed.

```
SELECT ?image ?version ?os
WHERE {
 ?package a dpv:SoftwarePackage ;
    rdfs:label "dash";
    dpv:hasVersion ?package_version;
    dpv:isPackageOf ?os .
 ?package_version dpv:isInstalledOn ?image;
    rdfs:label ?version
}
```

Listing 4: Retrieving all images using the *dash* package, its version and Operating System

Table 3 shows a subset of the results retrieved when executing the query in Listing 4. Among this subset, *dash* is available for two different Operating Systems (Ubuntu 16.04 and Debian 9), with two different versions (0.5.8-2.4 and 0.5.8-2.2). This shows how retrieving annotated structured data with our approach is easy if the name of the package is known. In contrast, accessing this information in DockerHub is complicated, as it is not available in the UI. This task becomes even more challenging when a Dockerfile is not available for an image, as it would require users to inspect each of the installed dependencies by hand.

Image	Package version	OS
dpsi:dockerpedia-pegasus_workflow_images_latest	dppv:dash-0.5.8-2.4	dpos:debian-9
dpsi:pegasus_workflow_images-%3Alatest	dppv:dash-0.5.8-2.4	dpos:debian-9
dpsi:mintproject-pihm2cycles_1.1	dppv:dash-0.5.8-2.4	dpos:debian-9
dpsi:mintproject-pihm2cycles_1.2	dppv:dash-0.5.8-2.4	dpos:debian-9
dpsi:library-mysql_latest	dppv:dash-0.5.8-2.4	dpos:debian-9
dpsi:dockerpedia-pegasus_workflow_images_latest	dppv:dash-0.5.8-2.4	dpos:ubuntu-16.04
dpsi:pegasus_workflow_images-%3Alatest	dppv:dash-0.5.8-2.4	dpos:ubuntu-16.04
dpsi:mintproject-pihm2cycles_1.1	dppv:dash-0.5.8-2.4	dpos:ubuntu-16.04
dpsi:mintproject-pihm2cycles_1.2	dppv:dash-0.5.8-2.4	dpos:ubuntu-16.04
dpsi:library-mysql_latest	dppv:dash-0.5.8-2.4	dpos:ubuntu-16.04
dpsi:dockerpedia-pegasus_workflow_images_pegasus-4.8.5	dppv:dash-0.5.8-2.1ubuntu2	dpos:ubuntu-16.04
dpsi:pegasus_workflow_images-%3Apegasus-4.8.5	dppv:dash-0.5.8-2.1ubuntu2	dpos:ubuntu-16.04

Table 3: Results of the query listed in Listing 4

4.3. *Comparing different execution environments*

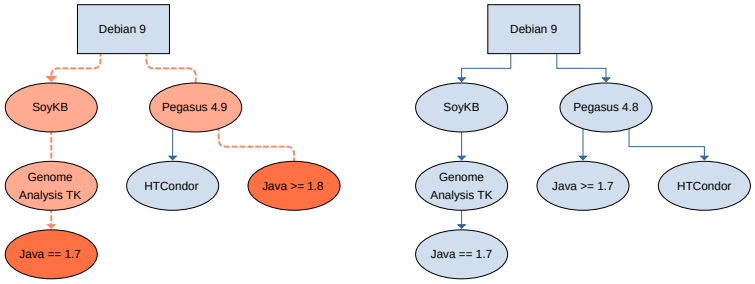
DockerPedia can be leveraged for comparing the differences and commonalities between two or more Docker images. This is a common need when evaluating alternatives to a given image, when considering installing new tools on top of an existing image (as some dependencies may already be installed); or when debugging errors of existing images (by comparing them against previous successful versions).

To illustrate these use cases, we retrieve the common packages between two versions of the Pegasus image, i.e., version 4.8.5 against the one with the *latest* tag. The query required to retrieve common packages can be seen in Listing 5. This capability provided by Dockerpedia is not currently available in Dockerhub.

```
SELECT ?soft WHERE {
 dpsi:dockerpedia-pegasus_workflow_images_latest
    dpv:containsSoftware ?soft .
 dpsi:pegasus_workflow_images%3Apegasus-4.8.5
    dpv:containsSoftware ?soft
}
```

Listing 5: Common components between two images

An analogous query can be issued to retrieve the differences between the software packages in two images, as shown in Listing 6, by using the *MINUS* operator from SPARQL.



(a) Dependency graph Pegasus 4.9 and SoyKB

(b) Dependency graph Pegasus 4.8 and SoyKB

Fig. 3: The orange nodes are the differences between the images

```
SELECT ?soft WHERE {
 dpsi:dockerpedia-pegasus_workflow_images_latest
    dpv:containsSoftware ?soft .
 MINUS{
  dpsi:pegasus_workflow_images%3Apegasus-4.8.5
     dpv:containsSoftware ?soft .
 }
}
```

Listing 6: What are the different components between two images?

Calculating the differences between images is a useful debug tool for reproducibility purposes, e.g., when we follow the steps to recreate an execution environment but there are errors. For example, we built the SoyKB workflow image [41] on August 2018, being able to execute it successfully at that time. However, after rebuilding the same image in November 2019, we were not able to run the workflow anymore.

In order to identify the issue in the reproduced infrastructure, we compared the software components inside both images with a SPARQL query similar to the one shown in Listing 6. Since the packages were different, we analyzed the SoyKB code and documentation and the Pegasus dependencies obtaining the dependency graphs shown in Figures 3a and 3b. These graphs highlight how Pegasus 4.9 and SoyKB need different Java versions (Java 1.8 and 1.7 respectively), failing thus the execution of the experiment if one of these Java versions is used incorrectly. Without the semantic descriptions in DockerPedia, it would have been difficult to spot that problem.

4.4. *Tracing vulnerabilities*

When reusing a software image, it is important to know whether it may have security vulnerabilities in any of its installed dependencies or imported layers. These vulnerabilities may range from low level security issues at the Operating System level, to those derived from third party projects hosted in Git environments, and which may be solved by a simple update.

DockerPedia can be used to annotate and propagate vulnerabilities in images. Once a vulnerability has been detected, e.g., through a Github alert, all the images that depend on that package can be retrieved and listed, by exploiting the dependency relations instantiated in the Knowledge Graph. Figure 4 shows a simple application [42] we built for browsing vulnerabilities in existing images across different tags, linking them with the Common Vulnerabilities and Exposures database [33]. The left side of the figure shows a timeline with blue dots representing different images on the X axis; and their overall risk in the Y axis. This information can be combined with a query similar to Listing 4, in order to obtain a list of the compromised images for a given software dependency. Image vulnerability detection has been recently made available in DockerHub, as a paid feature.

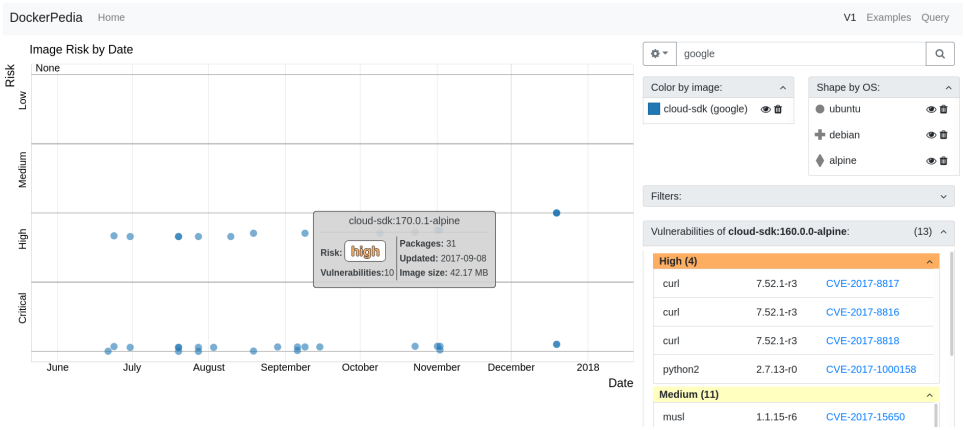


Fig. 4: Visualization of software images with their OS and their vulnerabilities, as well as the link to the CVE Database

4.5. *Discussion*

Our work addresses some of the FAIR principles (applied to computational environments) by allowing researchers to describe their computational infrastructures in a machine readable manner. In particular, as shown in case 4.2, DockerPedia eases *finding* images with certain dependency requirements, all while enabling transparent *access* and inspection to the underlying infrastructure (as shown in Section

4.1). Use cases 4.3 and 4.4 shed more light in the commonalities, differences and vulnerabilities between images, which are critical for their *reuse* and understanding. All metadata is represented using RDF, which ensures the *interoperability* of our dataset with other systems and applications.

Our approach assumes a stable versioning format, which is an established good practice when releasing software, in order to avoid processing images multiple times. Therefore, images with the *latest* tag are bound to the date when we last processed Docker Hub. New releases under the same tag would need to invoke the DockerPedia annotator to update the data.

As for limitations, our approach was designed for Docker Hub, but it would not require much effort to adapt it to other container-based platforms (our annotator can be used to work with any Linux container).

5. Related Work

We divide prior work into techniques for describing computational infrastructure and vocabularies for representing software images and their metadata.

5.1. *Describing computational infrastructure*

Scientific workflows (e.g., [3, 24, 6, 43]) have been traditionally used to capture the steps and data dependencies needed to carry out computational experiments; as they record the methods and provenance associated with a particular execution [44]. However, scientific workflows do not always succeed at documenting and recording the infrastructure needed to re-execute them. As pointed out by a study measuring the reproducibility over a corpus of public scientific workflows, 12% of the problems associated with the reproducibility of experiments were due to the lack of information about the execution environment [45]. Other studies have exposed the necessity of publishing adequate descriptions of the run-time environment of experiments to avoid replication hindering [46]. Our approach could be very beneficial in combination with scientific workflows, in order to preserve the images used by their software components.

ReproZip [47, 48] allows researchers to automatically create a VM or a companion Docker container along with a computational experiment or software. Reprozip builds a Docker image from the source code of a specific experiment, storing an internal description of the scientific experiment and its environment, which can be recreated by on another machine having installed Reprozip. However, Reprozip still requires to install all software components manually. Reprozip uses an internal description recording data and some of the software dependencies used in a component, but its focus is on encapsulation and reproducibility rather than help inspect, compare and find software images as it is our case.

Finally, container vulnerability inspectors such as Clair and Snyk [49] have gained popularity to ensure the deployment of secure applications. In DockerPedia we have incorporated Clair due to its open source nature, although during the

course of our work Docker introduced the *scan* command [50], which allows for extracting vulnerabilities from containers by using Snyk. This is an alternative to Clair, which can be used to complement the information extracted and linked by our approach.

5.2. *Vocabularies for representing software images*

Previous efforts have attempted to describe software images at different levels of granularity. For example, the community has developed parsers [51] to extract basic metadata from Dockerfiles according to schema.org [52], a popular community-based vocabulary used by search engines to describe structured information in web pages. However, the parsers do not link data between different images or expose installed packages, as we do in our case.

The Smart Container ontology [53] extends the W3C Provenance Ontology PROV-O [54] and models the execution traces and the role of Docker when for deploying images. This is orthogonal to our work, as we focus on the contents of the image rather than describing how the image was deployed on different infrastructures.

DockerOnto [55] describes how to use RDF to represent Dockerfiles. However, DockerOnto's goal is to represent the instructions and commands used in an Dockerfile, rather than exporting all the packages and layers contained within the resultant image.

Finally, [27] aims to represent computational reproducibility by describing the contents of Virtual Machine images. The authors present a set of ontologies that model software and hardware components to allow the execution environment reproducibility. We have extended this work into our own, extending it with an automated approach to annotate images (before, computational infrastructures had to be manually annotated).

6. Conclusions and Future Work

In this work, we propose a framework to describe and automatically annotate all the software components of a computational environment, captured in a software image. We combine Docker images with an annotation process in a method to obtain the software components and building steps related to the environment. Furthermore, we followed our approach to create DockerPedia, a Knowledge Graph with over 150,000 annotated software images.

Our work is a unique contribution towards adopting the FAIR principles for computational infrastructure. Thanks to DockerPedia, it is easy to find, explore, compare and inspect images to understand them better. Our annotations can even be used to detect issues in software components (e.g., with incompatibilities or incorrect versions). In addition, our approach can be extended to include external package managers (besides system packages) such as Conda.

As for future work, there are two main ways in which we would like to extend DockerPedia. First, by supporting other container-based virtualization solutions, such as Singularity [56], Linux (LXC) Containers [57] or Kubernetes [58]. For this, the parsing and annotations capabilities of DockerPedia should be updated, so as to be able to capture the particularities of these solutions, and incorporate them to the DockerPedia Knowledge Graph. Aligning the DockerPedia tool-set with the Open Containers Initiative [59] will also increase the scope of applicability our current system. Beyond including new containers, covering new package managers (e.g., CRAN [60]) would be also a relevant addition to explore in the near future. Secondly, we could improve DockerPedia by capturing better context of the Dockerfile itself (e.g., by combining our approach with DockerOnto to capture the commands required for setup, and schema.org representations for complementing our images with additional metadata), and by including a visualization of the dependency graph and installed files within an image, similar to those depicted in Figure 3. Generating these kind of images is really helpful, but not straightforward given the large amount of dependencies that are usually required to create an image. The semantic annotations from the DockerPedia KG may allow depicting the corresponding dependency graphs, spotting and highlighting the main differences between images.

Acknowledgements

Maximiliano Osorio and Carlos Buil-Aranda have been funded by Fondecyt Iniciación project 11170714 and by Insitituto Milenio de Fundamentos de los Datos, IMFD. Maximiliano Osorio has also been funded by DGIIP from Universidad Técnica Federico Santa María.

Daniel Garijo has been supported by the Madrid Government (Comunidad de Madrid-Spain) under the Multiannual Agreement with Universidad Politécnica de Madrid in the line Support for R&D projects for Beatriz Galindo researchers, in the context of the V PRICIT (Regional Programme of Research and Technological Innovation).

References

[1] G. Aad, J. Butterworth, J. Thion, U. Bratzler, P. Ratoff, R. Nickerson, J. Seixas, I. Grabowska-Bold, F. Meisel, S. Lokwitz *et al.*, The ATLAS experiment at the CERN large hadron collider, *Jinst* **3** (2008) p. S08003.

[2] R. Filguiera, A. Krause, M. Atkinson, I. Klampanos and A. Moreno, dispel4py: a Python framework for data-intensive scientific computing, *The International Journal of High Performance Computing Applications* **31**(4) (2017) 316–334.

[3] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li and T. Oinn, Taverna: a tool for building and running workflows of services, *Nucleic Acids Research* **34**(suppl_2) (2006) 729–732.

[4] T. Joshi, B. Valliyodan, S. M. Khan, Y. Liu, J. M. dos Santos, Y. Jiao, D. Xu, H. T. Nguyen, N. Hopkins, M. Rynge *et al.*, Next generation resequencing of soybean germplasm for trait discovery on xsede using Pegasus workflows and iplant infrastructure (2014).

[5] T. Joshi, M. R. Fitzpatrick, S. Chen, Y. Liu, H. Zhang, R. Z. Endacott, E. C. Gaudiello, G. Stacey, H. T. Nguyen and D. Xu, Soybean knowledge base (SoyKB): a web resource for integration of soybean translational genomics and molecular breeding, *Nucleic Acids Research* **42**(D1) (2014) D1245–D1252.

[6] Y. Gil, P. A. Gonzalez-Calero, J. Kim, J. Moody and V. Ratnakar, A semantic framework for automatic generation of computational workflows using distributed data and component catalogues, *Journal of Experimental & Theoretical Artificial Intelligence* **23**(4) (2011) 389–467.

[7] J. M. Perkel, Challenge to scientists: does your ten-year-old code still run?, *Nature* **584** (August 2020) 656–658.

[8] J. Cito, V. Ferme and H. C. Gall, Using Docker containers to improve reproducibility in software and web engineering research, in *International Conference on Web Engineering*, Springer2016, pp. 609–612.

[9] D. Inc., Docker hub. (2021), Accessed from `https://hub.docker.com/`.

[10] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne *et al.*, The FAIR guiding principles for scientific data management and stewardship, *Scientific data* **3**(1) (2016) 1–9.

[11] C. Goble, S. Cohen-Boulakia, S. Soiland-Reyes, D. Garijo, Y. Gil, M. R. Crusoe, K. Peters and D. Schober, FAIR computational workflows, *Data Intelligence* **2**(1-2) (2020) 108–121.

[12] A.-L. Lamprecht, L. Garcia, M. Kuzak, C. Martinez, R. Arcila, E. Martin Del Pico, V. Dominguez Del Angel, S. van de Sandt, J. Ison, P. A. Martinez *et al.*, Towards FAIR principles for research software, *Data Science* **3**(1) (2020) 37–59.

[13] M. Osorio, H. Vargas, I. Santana-Perez, D. Garijo and C. Buil-Aranda, The dockerpedia ontology. Accessed from `http://dockerpedia.inf.utfsm.cl/vocab`.

[14] M. Osorio, DockerPedia annotator source code. Accessed from `https://github.com/dockerpedia/annotator`.

[15] M. Osorio, C. Buil-Aranda and H. Vargas, Dockerpedia. Accessed from `https://dockerpedia.inf.utfsm.cl/`.

[16] Docker homepage.

[17] G. R. Brammer, R. W. Crosby, S. J. Matthews and T. L. Williams, Paper mâché: Creating dynamic reproducible science, *Procedia Computer Science* **4** (2011) 658–667.

[18] P. Van Gorp and S. Mazanek, SHARE: a web portal for creating and sharing executable research papers, *Procedia Computer Science* **4** (2011) 589–597.

[19] B. Howe, Virtual appliances, cloud computing, and reproducible research, *Computing in Science & Engineering* **14**(4) (2012) 36–41.

[20] F. da Veiga Leprevost, B. A. Grüning, S. Alves Aflitos, H. L. Röst, J. Uszkoreit, H. Barsnes, M. Vaudel, P. Moreno, L. Gatto, J. Weber *et al.*, BioContainers: an open-source and community-driven framework for software standardization, *Bioinformatics* **33**(16) (2017) 2580–2582.

[21] P. Di Tommaso, E. Palumbo, M. Chatzou, P. Prieto, M. L. Heuer and C. Notredame, The impact of Docker containers on the performance of genomic pipelines, *PeerJ* **3** (2015) p. e1273.

[22] B. Marwick, Computational reproducibility in archaeological research: basic principles and a case study of their implementation, *Journal of Archaeological Method and Theory* **24**(2) (2017) 424–450.

[23] Dockerfile documentation page. Accessed from `https://docs.docker.com/engine/reference/builder/`.

[24] E. Deelman, K. Vahi, M. Rynge, G. Juve, R. Mayani and R. Ferreira da Silva, Pegasus

in the cloud: Science automation through workflow technologies, *IEEE Internet Computing* **20**(1) (2016) 70–76, Funding Acknowledgements: NSF ACI SI2-SSI 1148515, NSF ACI 1245926, NSF FutureGrid 0910812, NHGRI 1U01 HG006531-01.

[25] Q. Wang, Z. Mao, B. Wang and L. Guo, Knowledge graph embedding: A survey of approaches and applications, *IEEE Transactions on Knowledge and Data Engineering* **29**(12) (2017) 2724–2743.

[26] R. Cyganiak, D. Wood, M. Lanthaler, G. Klyne, J. J. Carroll and B. McBride, RDF 1.1 concepts and abstract syntax, *W3C recommendation* **25**(02) (2014) 1–22.

[27] I. Santana-Perez, R. F. da Silva, M. Rynge, E. Deelman, M. S. Pérez-Hernández and O. Corcho, Reproducibility of execution environments in computational science using semantics and clouds, *Future Generation Computer Systems* **67** (2017) 354–367.

[28] Docker manifest specification v2.2. Accessed from `https://docs.docker.com/registry/spec/manifest-v2-2/`.

[29] Clair GitHub repository. Accessed from `https://github.com/quay/clair`.

[30] Quai main page. Accessed from `http://status.quay.io`.

[31] Conda documentation page. Accessed from `https://conda.io/docs`.

[32] Digital Ocean main page. Accessed from `https://www.digitalocean.com/`.

[33] Common vulnerabilities and exposures database home page. Accessed from `https://cve.mitre.org/`.

[34] CVE reports. Accessed from `https://ubuntu.com/security/cve`.

[35] Debian security bug tracker home page. Accessed from `https://security-tracker.debian.org/tracker/`.

[36] Red Hat Product Security Data Homepage. Accessed from `https://access.redhat.com/security/data`.

[37] Alpine Security Database of Backported fixes. Accessed from `https://github.com/alpinelinux/alpine-secdb`.

[38] M. Osorio and C. Buil-Aranda, Dockerpedia dataset (December 2018), doi: `https://doi.org/10.5281/zenodo.1897809`.

[39] M. Osorio, C. Buil-Aranda, I. Santana-Pérez and D. Garijo, Queries used to illustrate the DockerPedia Knowledge Graph (Jun 2021), doi: 10.6084/m9.figshare.14718450.v1.

[40] Dash package documentation page. Accessed from `http://manpages.ubuntu.com/manpages/xenial/man1/dash.1.html`.

[41] M. Osorio, dockerpedia/soykb: thesis (December 2018), doi: `https://doi.org/10.5281/zenodo.1889356`.

[42] Dockerpedia package vulnerability visualization page. Accessed from `https://dockerpedia.inf.utfsm.cl/visualization`.

[43] P. Amstutz, M. R. Crusoe, N. Tijanić, B. Chapman, J. Chilton, M. Heuer, A. Kartashov, D. Leehr, H. Ménager, M. Nedeljkovich, M. Scales, S. Soiland-Reyes and L. Stojanovic, Common Workflow Language, v1.0 (7 2016), doi: 10.6084/m9.figshare.3115156.v2.

[44] D. Garijo, Y. Gil and O. Corcho, Abstract, link, publish, exploit: An end to end framework for workflow sharing, *Future Generation Computer Systems* **75** (2017).

[45] K. Belhajjame, M. Roos, E. Garcia-Cuesta, G. Klyne, J. Zhao, D. De Roure, C. Goble, J. M. Gomez-Perez, K. Hettne and A. Garrido, Why workflows break — understanding and combating decay in taverna workflows, in *Proceedings of the 2012 IEEE 8th International Conference on E-Science (e-Science)*, *E-SCIENCE '12*, (IEEE Computer Society, Washington, DC, USA, 2012), pp. 1–9.

[46] N. D. Rollins, C. M. Barton, S. Bergin, M. A. Janssen and A. Lee, A computational model library for publishing model documentation and code, *Environ. Model. Softw.*

61 (November 2014) 59–64.
[47] F. Chirigati, D. Shasha and J. Freire, Reprozip: Using provenance to support computational reproducibility, in *Presented as part of the 5th {USENIX} Workshop on the Theory and Practice of Provenance*, 2013.
[48] V. Steeves, R. Rampin and F. Chirigati, Using ReproZip for reproducibility and library services, *IASSIST Quarterly* **42**(1) (2018) 14–14.
[49] Snyk home page. Accessed from `https://support.snyk.io/hc/en-us`.
[50] Scan command documentation page. Accessed from `https://docs.docker.com/engine/scan/`.
[51] Extract dockerfile code repository. Accessed from `https://github.com/openbases/extract-dockerfile`.
[52] R. V. Guha, D. Brickley and S. Macbeth, Schema. org: Evolution of structured data on the web: Big data makes common schemas even more necessary., *Queue* **13**(9) (2015) 10–37.
[53] D. Huo, J. Nabrzyski and C. Vardeman, Smart container: an ontology towards conceptualizing docker, in *International Semantic Web Conference (Posters & Demos)*, 2015. Accessible from http://ceur-ws.org/Vol-1486/.
[54] T. Lebo, S. Sahoo and D. McGuinness, PROV-O: The PROV Ontology (2013).
[55] R. Tommasini, B. De Meester, P. Heyvaert, R. Verborgh, E. Mannens and E. Della Valle, Representing Dockerfiles in RDF, in *Proceedings of the 16th International Semantic Web Conference: Posters and Demos*, **1931**October 2017, pp. 1–4.
[56] G. M. Kurtzer, V. Sochat and M. W. Bauer, Singularity: Scientific containers for mobility of compute, *PloS one* **12**(5) (2017) p. e0177459.
[57] Linux LXC Conainers Project Homepage. Accessed from `https://linuxcontainers.org/`.
[58] Kunernetes Project Homepage. Accessed from `https://kubernetes.io/`.
[59] Open Containers Initiative. Accessed from `https://opencontainers.org/`.
[60] Cran project main page. Accessed from `https://cran.r-project.org/`.