Automatizing experiment reproducibility using semantic models and container virtualization

Maximiliano Osorio [a,*], Idafen Santana [b] and Carlos Buil-Aranda [a]
[a] *Departamento de Informática, Universidad Técnica Federico Santa María, Avda España 1680, Valparaíso Chile*
E-mails: mosorio@inf.utfsm.cl, cbuil@inf.utfsm.cl
[b] *Ontology Engineering Group, Universidad Politécnica de Madrid, Campus de Montegancedo, Madrid, Spain*
E-mail: isantana@fi.upm.es

Abstract.
Experimental reproducibility is a major cornerstone of the Scientific Method, allowing to run an experiment to verify its validity and advance science by building on top of previous results introducing changes to it. In order to achieve this goal, in the context of current in-silico experiments, it is mandatory to address the conservation of the underlying infrastructure (i.e., computational resources and software components) in which the experiment is executed. This represents a major challenge, since the execution of the same experiment on different execution environments may lead to significant result differences, assuming the scientist manages to actually run that experiment. In this work, we propose a method that extends existing semantic models and systems to automatically describe the execution environment of scientific workflows. Our approach allows to identify issues between different execution environments, easing experimental reproducibility. We also propose the use of container virtualization to allow the distribution and dissemination of experiments. We have evaluated our approach using three different workflow management systems for a total of five different experiments, showcasing the feasibility of our approach to both reproduce the experiments as well as to identify potential execution issues.

Keywords: experiment reproducibility, semantic models

1. Introduction

Experiment reproducibility is the ability to run an experiment with the introduction of changes to it, getting results that are consistent with the original one. Introducing changes allows to evaluate different experimental features of that experiment since researchers can incrementally modify it, improving and re-purposing the experimental methods and conditions [1]. To allow experiment reproducibility it is necessary to provide enough information about that experiment, allowing to understand, evaluate and build it again. Commonly, experiments are described as scientific workflows (representations that allow managing large scale computations) which run on distributed computing systems. To allow reproducibility of these scientific workflows it is necessary first to address a workflow conservation problem and second, the conservation of the underlying infrastructure (i.e. computational resources and software components) in which the scientific workflow is executed. Experimental workflow conservation refers to the process of obtaining the same result from an experiment in a different environment [2]. Experimental workflows need to guarantee that there is enough information about the experiments so it is possible to build them again by a third party, replicating its results without any additional information from the original author [2].

To achieve that conservation, the research community has focused on conserving workflow executions by conserving data, code, and the workflow descrip-

*Corresponding author. E-mail: mosorio@inf.utfsm.cl.

tion, but little work has been done in conserving the underlying infrastructure. The work in [3] or the Timbus project[1] [4] (which focuses on business processes and the underlying software and hardware infrastructure) provided some advances in the problem of conserving the experiment infrastructure. In [3], authors identified two approaches for conserving the environment of a scientific experiment: physical conservation, where the research objects within the experiment are conserved in a virtual environment; and logical conservation, where the main capabilities of the resources in the environment are described using semantic vocabularies to allow a researcher to reproduce an equivalent setting. They defined a process for documenting the workflow application and its related management system, as well as their dependencies. However, this process is done in a non-automated manner, leaving much work left to the scientists, and thus, prone to errors. Furthermore, that work relies completely on virtualization technologies using virtual machines to guarantee physical conservation. Even when this approach provides some advantages, the high storage demand of VM images hinders the physical conversation. The Timbus project also tries to address such logical and physical conservation problems; however, they incur in several problems such as running and modifying the physical and logical image to allow their analysis (installing new software), and thus non-intrusive reproducibility cannot be provided. In summary, most of the current approaches leave out of the scope the physical conservation of the workflow computational environment (relying on the chosen infrastructure). However, logical and physical conservation are important to achieve experiment reproducibility.

One way to solve the physical conservation problem is to use operating-system-level virtualization. This technology, also known as containerization, refers to an Operating System (OS) feature in which the OS kernel allows the existence of multiple isolated user-space instances called containers. One of the most popular virtualization technologies is Docker[2], which implements software virtualization by creating minimal versions of a base operating system (a container). Docker Containers can be seen as lightweight virtual machines that allow the assembling of a computational environment, including all necessary dependencies, e.g., libraries, configuration, code and data needed, among others. Herein we propose a solution to improve the physical and logical conservation solution by using containers and automated annotations. First, we propose to use Docker images as means for preserving the physical environment of an experiment. We use containers since they are lightweight and more importantly, they are easier to automatically describe so we improve the process of documenting scientific workflows. Using Docker, the users can distribute these computational environments through software images using public repositories such as DockerHub[3]. In order to achieve logical conservation, we built an annotator system for the Docker Images that describe the workflow management system, as well as their dependencies. In this way, we aim to address the physical conservation. To validate our solution we reproduce 5 different computational experiments. These experiments span different systems, languages and configurations, showing that our approach is generic and can be applied to a wide range of representative computational experiments. We run these experiments, we describe them logically and next we reproduce them based on the logical descriptions we obtained before. To validate the approach we compare the outputs obtained from the reproduced experiments to the original ones.

2. Related work

This work aims to allow scientists to reproduce their in-silico experiments. This is not the first work in trying to enable experiment reproducibility, and thus it is mandatory to look at the work done so far before starting our journey. For computational experiments to become reproducible, one needs to develop a system for linking scientific publications with computational recipes. These recipes (scientific workflows) need to be executed by data workflow systems [5–7], which run over commodity machines and make use of several other software components during its execution (i.e. the execution environment). All these experiment components need to interact to execute it, and they also have to run similarly if the experiment is reproduced by some other scientist in another environment. However, such component coordination is prone to errors, as the work in [8] pointed out. The authors studied the reproducibility of scientific workflows, showing that almost 80% of the workflows could not be reproduced.

[1] http://www.timbusproject.net/
[2] https://www.docker.com/
[3] https://hub.docker.com/

About 12% of the problems to reproduce these experiments were due to the lack of information about the execution environment. Furthermore, 50% of them were due to the use of third-party resources such as web services and databases that were not available anymore. Other studies have exposed the necessity of publishing adequate descriptions of the run-time environment of experiments to avoid replication hindering [9].

One way to allow experiment reproducibility is to use virtualization techniques by using a virtual machine (VM). In this way, the execution environment can be packed within a Virtual Machine and distribute it along with the experiment [10, 11]. However, the high storage demand of VM images remains a problem since a virtual machine needs to install the whole Operating System before installing any software component within it (consider that the minimal version of Windows is 4GB and 1GB for a Linux distribution). Also, the cost of storing and managing data in the Cloud is still high, and the execution of high-interactivity experiments through a network connection to remote virtual machines is also challenging. A list of advantages and challenges of using VMs for achieving reproducibility was exposed in [12]. VMs also introduce problems at the time of reproducing experiments: VMs work as black boxes, and even though is not strictly required to describe the experiment execution environment since we rely on the VM, it should be possible to know what components are installed within that VM, to be able to reproduce such environment outside the initial VM. In summary, even when VMs support reproducibility, they are too large and fixed, not being possible to know what components are inside the VM and which of those are really needed to reproduce the execution of the in-silico experiment.

To solve the aforementioned problems, contributions such as [13–15], propose the use of Docker Containers to allow reviewers, interested readers, and future researchers to reproduce the experiments within the same environment. Container solutions solve the problem of storage, however, the challenge of reproducing scientific contributions due to their high dependence on developed algorithms, tools and prototypes, quantitative evaluations, and other computational analyses that are not adequately documented still persists. It persists due to the containers working as black boxes in which the scientist does not know what packages are installed. Besides, these works do not provide any means to perform it, they only express desiderata.

The work in [3] addresses the problem of describing what is inside the VM image. To do that the authors solve the problem proposing a semantic modeling approach to conserve computational environments in scientific workflow executions. However, the annotation process is manual, and the high storage problem of the virtual machines persists. In the Timbus project[4] [4] the problem of logical conservation is addressed installing new software inside each environment (in this case, a virtual machine). However, this approach increases the complexity of the environment and requires to execute the computational environment. Thus, the system can suffer scalability and security issues.

To know what is inside a container image, the work introduced in [16] analyzed over 300,000 Docker images stored at Docker Hub. Authors have found that the images at Docker Hub have more than 180 vulnerabilities on average, being the root of such amount of vulnerabilities the fact that many images had not been updated for several days; many of these vulnerabilities are propagated from parent to child images. To perform the software vulnerability analysis, the authors used the open source project Clair[5] from CoreOS[6]. Clair executes a static analysis of vulnerabilities in application containers. The authors in [17] analyzed the security of Docker images from a higher level point of view. They presented a framework which characterized several vulnerabilities and how they may be prevented. Thus, not only the experiments lack reproducibility, but they also may be introducing security vulnerabilities into their systems.

In terms of ontological engineering for describing software components, the authors in [17] present the Smart Container ontology which extends the Provenance Ontology PROV-O [18] and models Docker in terms of its interactions for deploying images. Another related work [19] describes how to use RDF to represent Docker files. Similarly, a different approach in which software ontologies are used is to allow computational reproducibility [3]. In this work, the authors present a set of ontologies that model software and hardware components to allow the execution environment reproducibility.

As described in this section, several works have addressed the experiment reproducibility problem, providing frameworks in which virtualization techniques and structured knowledge representation are used. However, they either rely on VMs as black boxes, which are large and fixed environments that make the

[4] http://www.timbusproject.net/
[5] https://github.com/coreos/clair
[6] https://coreos.com/

portability of such experiments not optimal or require a manual annotation process, which hinders the quality and trust of the annotations. In the following section, we will describe our approach, which combines semantic descriptions and container-based virtualization to tackle these two problems.

3. Scientific workflows using containers

Scientific communities are following container-based approaches more and more often for conducting their empirical studies. Even when their main goal is not necessarily experimental reproducibility, the fact that these communities, which are mainly IT-oriented [20], are embracing such paradigms shows that lightweight virtualization techniques are a suitable environment for computational science.

Taking this as an starting point, we envision the following main requirements for a system that targets the overall reproducibility problem for computational workflows:

Packaging: the system must be able to bundle the scientific workflow in a single and meaningful unit, containing the necessary operational resources for allowing the community to execute them without being experts on the required environment, which is often complex and non-trivial to set up.

Portability: the system must support the execution of the experiment in different infrastructures, being able to be deployed in several different systems, allowing to be executed in the wide range of infrastructures available for the community.

Annotation: in order to ensure its logical conservation, the system should allow to describe the characteristics of the computational infrastructure in enough detail, including the software and hardware components to be included, as well as the required steps to be performed to set them up and configure the computational environment.

Isolation: the system should be able to identify those components related to the workflow and the corresponding WMS, in order to minimize the software to be annotated and deployed.

Storage: the system should aim at reducing the size of the shareable computational units. This is intended not only for reducing the overall size of the experimental environment, but also to promote they archival and spread throughout the scientific communities.

As introduced before, container virtualization systems meet several of these requirements, specially those related to physical conservation. Thus, we include them as part of our solution. As for the logical conservation, we build on top of the container technology, adding audit and annotation features for describing them in a structured and interoperable way.

From a technological point of view, Docker is a solution that allows virtualizing a minimal version of an Operating System (OS), sharing the resources from the host machine by means of software images. On top of this virtualized layer/image, dependencies can be deployed and applications can be executed. These images, containing the Operating System and dependencies can be easily published, due to their reduced size, into Docker Hub[7], the canonical repository which also supports the maintenance and download of containers. Throughout this section, we introduce how Docker and its hub work, starting with how Docker images are created and stored in Docker Hub.

3.1. Docker repositories and files

Docker builds a software image by following the set of instructions, defined as steps, written in the Dockerfile. A Dockerfile is a text file that contains all commands to build a Docker image and run a container using this image. The Dockerfile usually have several lines, which are translated into image layers whereas Docker builds the image. In the building process, commands are executed sequentially, creating one layer at a time. When an image is updated or rebuilt, only the modified layers (i.e. modified lines) are updated. These explicit mentions to the commands to be executed, including their order and parameters, is what allows our system to extract the necessary information for annotating the computational environment, including the components and steps. Since various images can have common layers, the proposed annotation process analyses each layer of the image and not the resultant image. Thus, the system can reuse previous analysis.

3.2. Publishing and Deploying Docker images from Docker Hub

Docker Hub is an online registry that stores two types of public repositories, official and community. Official repositories contain public, verified images, from well-known companies and software providers,

[7] https://hub.docker.com/

such as Canonical, Nginx, Red Hat, and Docker itself. At the same time, community repositories can be public or private repositories that are created by individual users and organizations, which share their applications and results.

By using that registry and a command line, it is possible to download and deploy Docker images locally, running the container in a host environment and then executing the software inside the image. Users are allowed to create and store images into the Docker Hub registry, by creating a Dockerfile or extending an existent one. This descriptor file describes all steps and the software packages needed to build the Docker image, builds the image and finally uploads it to Docker Hub. However, Docker Hub and the image do not control what packages are in the images, whether the image will deploy correctly or the images might have any security problem.

These registries allow scientific communities to store and share, curating the content of the containers, as well as to check the identity of the publisher and retrieve the containers of interest. However, the information of the content of each repository is not always accessible in an clear manner. As most of the times only short descriptions of the containers is provided, it is not straightforward for a user to understand which components are installed on it. Even when Dockerfiles are available, these are not intuitive enough to follow and understand which packages are being deployed by each command. Also, some components might exist in the container that are not specified by the Dockerfile itself. To tackle this problem, we propose an automatic approach for analyzing the content of a Docker image and extract the information about the software components installed on them. This information is then converted to semantic data, codified as RDF under the set ontologies we have developed, as mentioned in section 4.1.

4. Reproducibility in scientific workflows using Docker Containers

In this paper, we mainly focus on the role that Docker plays for experimental reproducibility, which fundamentally happens once the experiment has been conducted and its results have been disseminated. The use of containers poses several benefits during the designing, development, testing and execution phases of a research process, before the publication of the results. As light-weighted and isolated environments, containers allow users, mostly researchers from different areas conducting computational simulations, to test new solutions without jeopardizing real, and costly, production infrastructures. Moreover, as their evolution can be tracked along the development process, it is possible to rollback to previous versions in case new dependencies or modifications introduce errors. Thus, containers are being more and more commonly adopted by research communities, publishing them as part of the scholarly communication process, hosted on the aforementioned repositories. However, some work is still needed to use containers as key element for experiment reproducibility.

Here, we argue that the description of computational environments is necessary for the reproduction of the experiment. Furthermore, the information must be enough to compare and detect differences between the original and the reproduced environments.

Since Docker images are an isolated and independent environment, the installed software components within the container should only be related to a single experiment. This is considered a best practice in the software engineering community and we adhere to it. By making this assumption, we consider that the description of the environment is done without any noise from other tools or experiments.

To automatically annotate the software packages within the Docker images we propose a system which receives as input the repository name, queries the container's package system which software packages are installed and stores the annotations in our RDF repository.

4.1. Semantic models

In [3], the authors proposed The Workflow Infrastructure Conservation Using Semantics ontology (WICUS). WICUS is an OWL2 (Web Ontology Language) ontology network that implements the conceptualization of the main domains of a computational infrastructure. These are: Hardware, Software, Workflow and Computing Resources domain. Scientific workflow requires a stack of software components, and the researchers must know how to deploy this software stack to achieve an equivalent environment. Thus, we import some abstract classes, and relations from WICUS ontology: (1) DeploymentPlan, DeploymentStep, ConfigurationInfo and ConfigurationParameter classes describe the steps to deploy and configure the software. (2) SoftwareStack describes the software components that must be installed and their de-

pendencies. We also extend the ontology with the specific classes and properties related to OS virtualization, generic to any virtualization system that uses deployment layers such as Docker or Singularity[8]. We define the new class `SoftwarePackage` as a subclass of `SoftwareComponent` so we are able to define the software packages installed by the underlying OS. Every `wicus:SoftwareComponent` has an object type relation to `dockerpedia:hasVersion` denoting the package version which was installed within the container.

We annotate every line from the Docker file as a `wicus:DeploymentStep` and the Dockerfile as a `wicus:DeploymentPlan`. In summary, we annotate every installed software package on the container file system. We want to highlight that this is the only approach that gets to such level of detail of describing a virtual environment (either a container or a VM). This allows us to reproduce any experiment as long as it uses a container virtualization system and imports and build their tools using their configuration files (such as with multi-stage builds[9]).

Our final ontology is depicted in Figure 1.

4.2. Annotator

The annotation service implements a REST interface which receives as input the Docker image in which the scientific workflow will run. By scanning it, the system can describe the software components that support such the filesystem of the image and the building steps needed to run it. The whole annotation process is the following: First, the system downloads the Docker image and mounts it (without running it). Next, we scan the image searching the software packages installed, and finally, the system creates the RDF data from the scan process and link these data to external RDF resources such as the Debian package repository and the Common Vulnerabilities and Exposures database[10]. We show the architecture of the annotator in Figure 2.

4.2.1. Building steps annotations

Docker builds an image by either reading a set of instructions from a Dockerfile or just deploying that image on a host in case the Dockerfile is nor present. Thus, to identify what packages are going to be installed within the docker image we either read that Dockefile and execute it or we deploy that image and run an analysis over it. In case of the latter, we have to extract the information from the Docker image by using the image's manifest file, which is always available for a given image in the image manifest file (available in every Docker image). This file contains the image's internal configuration and the set of layers from which the image is built. According to the official documentation [11], the most important attributes of the manifest file 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. It is 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* the parent's ID;
ContainerConfig: the layer's build command;
Author: the author's name and email.

With all these information provided by either the Docker file or the manifest file within the Docker image we are able to reproduce the Docker image only deploying it, and thus without modifying any parameter in the image. In the next section we describe how we annotate the software components within the Docker images.

4.2.2. Software Components annotations

Each Docker image layer installs or removes software packages. To be able to describe the execution environment of a scientific experiment we need to describe the software components that are installed within the experiment image. To do that we rely on the installation process done by the software package managers of the image's operating system. A package manager system is a collection of software tools that automate the process of installing, upgrading, configuring, and removing software components. We classify the package managers in two types: system and general. System package managers are the managers of the operating system (e.g., apt by Debian Family, yum by RedHat Family) and general package managers are custom package manager, which generally are used to

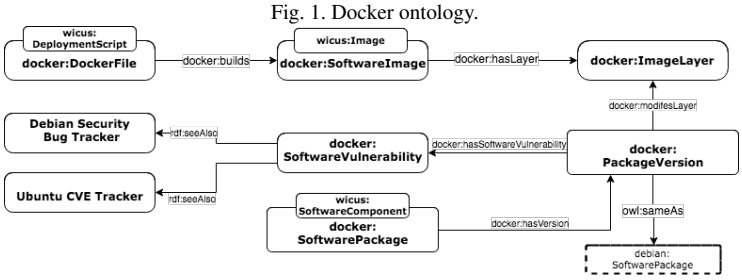
[8] https://singularity.lbl.gov/

[9] https://docs.docker.com/develop/develop-images/multistage-build/

[10] https://cve.mitre.org/

[11] https://docs.docker.com/registry/spec/manifest-v2-2/

Fig. 1. Docker ontology.



install a specified language package (e.g., pip, conda, npm).

A common approach for finding the software components is to search the lines that use the package manager. For example, Listing 1 shows the command to install the TensorFlow software package.

```
apt-get 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
```

Listing 1: Ubuntu command to install the TensorFlow software

The main problem of this approach is that there is no information about the software packages versions nor the software dependencies installed. Also, this command installs 184 packages which the scientist may not be aware of.

We use the Clair tool to identify which packages the container virtualization system installs. Clair is an open-source tool from CoreOS designed to identify known vulnerabilities in Docker images. It has been primarily used to scan images in the CoreOS private container registry, Quay.io[12], but it can also be used to analyze images from DockerHub. Clair downloads all layers of an image, mounts and analyzes them, determining the operating system of the layer and the packages added and removed from it. As a result from the analysis, Clair downloads all layers of an image, mounts and analyzes them, determining the operating system of the layer and the packages added and removed from it. Clair is compatible with several of the most common system package managers, such as Ubuntu, Debian, Alpine, RedHat, CentOS, and Oracle. Thus, our implementation supports experiments that run over those well-known Linux distributions. Due to the popularity of the Conda[13] management tool in the scientific community and as a generalization proof, we extend Clair in our system to detect the packages and dependencies installed by this package manager.

4.3. Reproduce and compare the new environment

We store the RDF descriptions of each software image packages in an Apache Jena TDB[14] storage system and provide access to it through an Apache Fuseki server [15] using the SPARQL 1.1 W3C recommendation [21]. Once we annotate the experiment software image and the installed packages within that image, we can reproduce that experiment. To do that, we create the Docker image again using the previous annotations. A simple SPARQL query as shown in Listing 2 suffices to obtain all the software packages installed in the image and just repeating the package manager install command we are able to reproduce the exact execution environment.

Listing 2: SPARQL query obtaining the software packages installed in the latest Pegasus version

```
PREFIX vocab:
<http://dockerpedia.inf.utfsm.cl/vocab#>

SELECT * WHERE {
```

[12]http://status.quay.io

[13]https://conda.io/docs/

[14]https://jena.apache.org/documentation/tdb/

[15]https://jena.apache.org/documentation/serving_data/

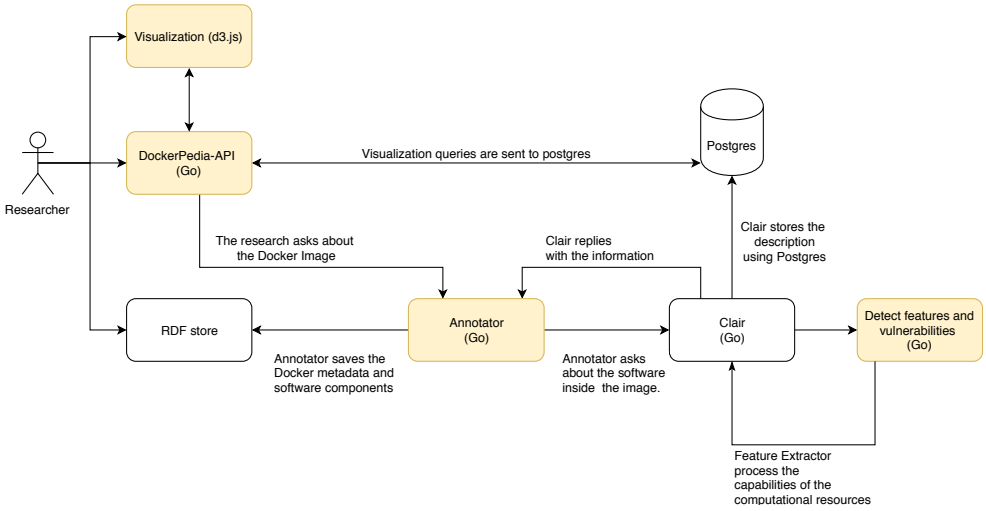


Fig. 2. This Figure shows the general architecture of the Annotator system. The Annotator provides researchers an API which receives as input a Docker image URL from DockerHub. The annotator will describe that image using the semantic model depicted in Figure 1 and it will also search for software vulnerabilities. The system also provides a visualization tool of the components within each Docker image. These components are highlighted in yellow.

```
5   pegasus_workflow_images%3Alatest
6         vocab:containsSoftware ?p .
7       }
```

In case there was any problem reproducing the execution environment of an experiment we can also compare the differences between two images. For example, the query in Listing 3 shows the query to compare two versions of the Pegasus image.

Listing 3: SPARQL query comparing two the software packages installed in two different Pegasus versions

```
1  SELECT * WHERE {
2   pegasus_workflow_images%3Alatest
3         vocab:containsSoftware ?p .
4   pegasus_workflow_images%3Apegasus-4.8.5
5         vocab:containsSoftware ?p
6       }
```

5. Experimentation Process

In this section we evaluate our approach for allowing scientific experiment reproducibility. We reproduce five different experiments (SoyKB, Montage, Internal extinction and Modflow) running on three different data workflow systems (Pegasus, Dispel4Py and WINGS). We will deploy these workflow systems on three different execution environments (Google Cloud, Digital Ocean and on a local machine) to guarantee that our approach is platform independent.

This section is organized as follows: we first describe how we execute the experiments we use to evaluate our approach (i.e. how we build the experiment images, how and where we store them, how we describe them and how we compare the results from the different experiment executions). When evaluating our approach, we first introduce the workflow system we use for next introducing the experiments that will run on that workflow system. Before continuing to the next workflow system and scientific experiments we show and compare the results from the workflow executions with the existing state of the art results for the same experiments, showing the feasibility of our approach.

5.1. Building and storing the images

We built one Docker image for each workflow system so a researcher can import that image and install on it the software components needed to run the scientific data workflow. We did that by using the FROM instruction in the Dockerfile. For example, the SoyKB experiment uses the Pegasus workflow manager, thus the SoyKb image imports the Pegasus software image at DockerHub. The SoyKB workflow needs to install other software packages besides the Pegasus system, which we also incorporated into the Dockerfile

Resource	Digital Ocean	Google Compute	Local
RAM (GB)	8	8	4
Disk (GB)	100	100	70
CPU (Arch)	64	64	64
OS	Centos 7	Debian 9	Fedora 27

Table 1
Image appliances characteristics.

describing the container image. These files are available on our repositories for each workflow [16]. We also describe the workflow images using the Container description vocabulary developed by the Open Container Initiative[17].

5.2. Physical conservation

We rely on the physical conservation to DockerHub and the Docker Images. Thus, the first experiment is to test if the Docker images are capable of packaging the software components of the selected experiments. The images that we are using in our experimentation are publicly available on DockerHub[18]. Moreover, we published the images with the corresponding Dockerfiles so that any user inspect and improve them. To evaluate that the Docker Images are lightweight and storable, we built the environment for each workflow using virtual machines (VM) and containers and compared the disk usage of both. Finally, we use two different infrastructure providers (DigitalOcean and Google Cloud) and a local machine to evaluate the reproducibility using physical conservation. Table 1 shows the hardware characteristics of these three environments.

The Docker version (37) tested for this experimentation is compatible with CentOS 7, Debian 10/9/8/7.7, Fedora 26/27/28, Ubuntu 14.04/16.06/18.04, Windows 10, macOS El Capitan 10.11 and newer macOS releases.

5.3. Describing the components of the environment

As described in Section 4.2, we annotate the aforementioned workflows using the set of semantic models we have extended. The annotations are grouped by building steps and software components. In order to get the annotations from the containers, we use and extend Clair, as depicted in Figure 2, which shows the main steps of the process. Overall, the annotation process works based on the following phases.

1. The user queries the DockerPedia annotator API, using a Docker image at DockerHub as input.
2. The DockerPedia annotator uses our extended version of Clair to analyze the image. Clair downloads all layers of an image, mounts and analyzes them, determining the operating system of the layer and the packages added and removed from it. In parallel, the annotator reads the building steps, labels, architecture and manifest from DockerHub.
3. Finally, the annotator combines all the gathered information and codifies and stores it in RDF, using the semantic models we developed.

5.4. Is the reproduced environment similar?

To evaluate if the original and reproduced environments are similar enough, we compare both annotated images using SPARQL to query the annotations. We have tested this in a real scenario, where one image was able to execute a workflow, whereas the other could not. More in detail, recently Pegasus updated to version 4.9 and required Java version 1.8 while the SoyKB workflow requires Java version 1.7 making thus incompatible the new Pegasus version with the workflow. With the SPARQL query in Listing 4 is easy to spot the differences between both execution environments.

Listing 4: What are the different components between two images?

```
PREFIX vocab:
<http://dockerpedia.inf.utfsm.cl/vocab#>

SELECT * WHERE {
  pegasus_workflow_images%3Alatest
    vocab:containsSoftware ?p .
  MINUS{
  pegasus_workflow_images%3Apegasus-4.8.5
    vocab:containsSoftware ?p
  }
}
```

In summary, for each of the experiments described in the next Section, we perform the following steps:

– Build a Docker image for each of the scientific workflows.

[16] https://github.com/dockerpedia
[17] https://www.opencontainers.org/
[18] https://hub.docker.com/u/dockerpedia/

- Annotate each of the previous Docker images.
- Reproduce the environment from the annotations obtained by our approach.
- Compare both execution environments.
- Evaluate and compare the size between virtual machine image and container image.
- Run the experiments and compare their execution results with the original workflow execution results. If the results are the same we managed to successfully reproduce the experiment.

5.5. Pegasus

Pegasus [6] is a WMS able to manage workflows comprised of millions of tasks, recording data about the execution and intermediate results. Pegasus optionally relies on HTCondor[19] as the task manager and we build two versions of the Pegasus image, one with Condor and another without it. The Pegasus package has obtained from the official repository [20]. The main pegasus requirements are Openssh-server, Condor, and Java (the Java version depends on the Pegasus version). The Pegasus images used in this work are available on DockerHub [21].

5.5.1. Soybean Knowledge Base

The SoyKB workflow [22, 23] is a genomics pipeline that re-sequences soybean germplasm lines selected for desirable traits such as oil, protein, soybean cyst nematode resistance, stress resistance, and root system architecture. The workflow implements a Single Nucleotide Polymorphism (SNP) and injection/deletion (indel) identification and analysis pipeline using the GATK haplotype caller15 and a soybean reference genome. The workflow analyzes samples in parallel to align them to the reference genome, to de-duplicate the data, to identify indels and SNPs, and to merge and filter the results. The results are then used for genome-wide association studies (GWAS) and genotype to phenotype analysis. The workflow instance used in this paper is based on a sample dataset that requires less memory than a full-scale production workflow, however it carries out the same process and requires the same software components.

The SoyKB main software dependencies for running the workflow on Pegasus are classified in self-dependencies (in yellow) and third-party dependencies (in green). The main components in these dependencies are `bwa`, `gatk` and `picard` while the main third-party dependency is an unknown version of Java.

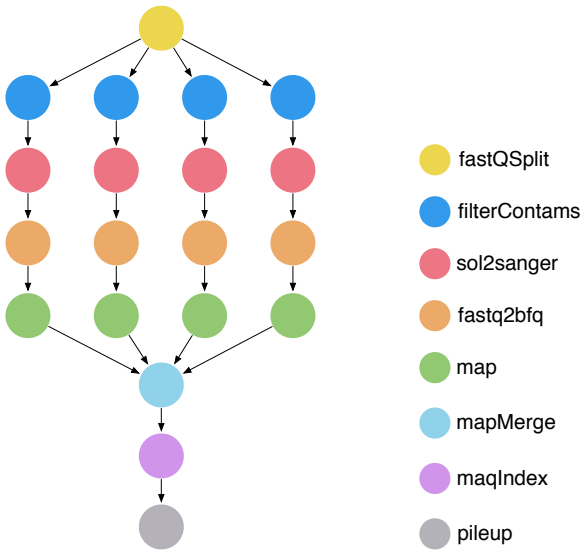


Fig. 3. Pegaqsus's SoyKB Workflow representation.

We evaluated the results obtained manually, as in [3], since the scientific workflow execution is non-deterministic, due to some of its steps being probabilistic. We compared the structure of the resulting data, file sizes, number of lines and the absence of errors. According to Pegasus, the workflow outputs a VCF file containing genomic data, which is the same data available in the official Pegasus repository in GitHub [22], thus we can conclude that the outputs are similar and that we reproduced the workflow successfully. Both the SoyKB workflow image and the workflow execution results are in DockerHub [23].

5.5.2. Montage

The Montage workflow [24] was created by the NASA Infrared Processing and Analysis Center (IPAC) as an open source toolkit that can be used to generate custom mosaics of astronomical images in the Flexible Image Transport System (FITS) format. In a Montage workflow, the geometry of the output mosaic is calculated from the input images. The inputs are then

[19] https://research.cs.wisc.edu/htcondor/
[20] http://download.pegasus.isi.edu/wms/download/debian
[21] https://hub.docker.com/r/dockerpedia/pegasus_workflow_images/
[22] https://github.com/pegasus-isi/PGen-GenomicVariations-Workflow
[23] https://doi.org/10.5281/zenodo.1889356,https://doi.org/10.5281/zenodo.1897809

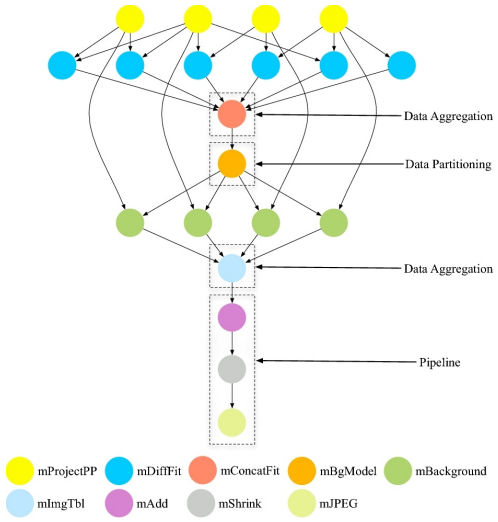


Fig. 4. Workflow representation provided by Pegasus. Each of the nodes is an operation executed by different software components. This shows the complexity of the execution environment needed to run the experiment.

re-projected to have the same spatial scale and rotation, the background emissions in the images are corrected to have a uniform level, and the re-projected, corrected images are co-added to form the output mosaic. Figure 4 illustrates a small (20 node) Montage workflow. The size of the workflow depends on the number of images required to construct the desired mosaic. Each of the nodes in the workflow is a binary software that contributes to the final image generation. Since the software is only provided in its binary format (not packaged), we downloaded it and added it as a dependency in our DockerFile [25].

We use a perceptual hashing tool [24] to compare the outputs from the original Montage workflow execution provided by Pegasus and our reproduced Pegasus environment. We obtain as result a similarity factor of 1.0 (out of 1.0). Figures 5a and 5b show both resulting images and the output files are available in DockerHub [25].

5.6. dispel4py

dispel4py [26] is a Python library for describing workflows. It describes abstract workflows for data-intensive applications, which are later translated and enacted in distributed platforms (e.g. Apache Storm, MPI clusters, etc.). The dispel4py images are available at DockerHub [26]. To install the packages needed to run the workflows we use Conda, a package, dependency and environment manager for Python-based execution environments. To freeze the version of the packages that will be installed, we include the complete list of installed packages on the GitHub repository.

5.6.1. Internal extinction

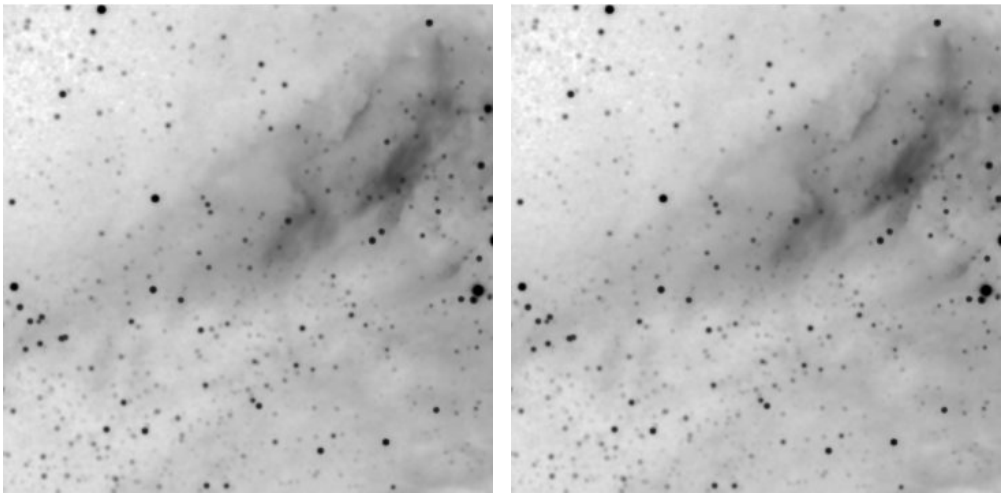
Internal Extinction of Galaxies The Virtual Observatory (VO) is a network of tools and services implementing the standards published by the International Virtual Observatory Alliance (IVOA) [27] to provide transparent access to multiple archives of astronomical data. VO services are used in Astronomy for data sharing and serve as the main data access point for astronomical workflows in many cases. This is the case of the workflow presented here, which calculates the Internal Extinction of the Galaxies from the AMIGA catalogue [28]. This property represents the dust extinction within the galaxies and is a correction coefficient needed to calculate the optical luminosity of a galaxy.

The scientific workflow first reads the file containing the inclination and ascension rate of 1051 galaxies. Next, the workflow use these data values to to query the Virtual Observatory and obtains the results selecting only those values corresponding to the morphological type (Mtype) and the apparent flattening (logr25) features of the galaxies. Finally, the workflow calculates the internal extinction of the galaxy. Figure 6 shows the execution steps fo the workflow.

The main software dependencies needed to run such workflow are `requests`, `Python 2.7`, `numpy` and `astropy`.

Since the Internal Extinction workflow uses an online service, the input data may vary among workflow executions. Thus, we validated the different workflow execution outputs by verifying the results data structures, file sizes, number of lines and nonexistence of errors during the execution. Both executions (from the original and reproduced execution environments) were identical and we conclude that we were able to reproduce the experimental workflow. The Docker images and their results from these experiments are available at DockerHub [29].

[24] http://phash.org
[25] https://doi.org/10.5281/zenodo.1889328,https://doi.org/10.5281/zenodo.1889360
[26] https://hub.docker.com/r/dispel4py/dispel4py
[27] http://www.ivoa.es
[28] http://amiga.iaa.es
[29] https://doi.org/10.5281/zenodo.1889332,https://doi.org/10.5281/zenodo.1889328



(a) Result from the Montage workflow execution on Pegasus

(b) Result from the Montage workflow execution on our reproduced Pegasus environment.

Fig. 5. The results from the Montage workflow execution using two different execution environments are exactly the same, validating our hypothesis.

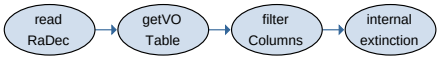


Fig. 6. Execution steps of the Internal Extinction scientific workflow

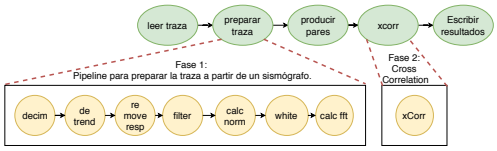


Fig. 7. Seismic Ambient Noise Cross-Correlation workflow representation. Each of the nodes has several operations executed by different software components, leading thus to multiple software versions that may cause problems when reproducing the experiment.

5.6.2. *Seismic Ambient Noise Cross-Correlation*

Seismic Ambient Noise Cross-Correlation workflow (or xcorr workflow) is part of the project *Virtual Earthquake and seismology Research Community e-science environment in Europe* (VERCE). The goal of this workflow is to prevent damages by earthquakes and volcano eruptions. These natural events are preceded by changes in the Earth's geophysical properties such as wave speed.

The `xcorr` workflow has two stages, being the first a time series pre-processing of a seismic station data (which is done in parallel) and next each station correlates each pair. Figure 7 shows the workflow steps.

The main software dependencies are `Python 2.7`, `obspy` and `numpy`. Regarding the workflow execution, both executions on the original and replicated environment obtained the same results, thus concluding that we successfully reproduced the experiment. The workflow images are at DockerHub [30] while the results are at GitHub [31].

5.7. *WINGS*

WINGS is a semantic workflow system that assists scientists with the design of computational experiments. A feature of WINGS is that its workflow representations incorporate semantic constraints about datasets and workflow components, and are used to create and validate workflows and to generate metadata for new data products. WINGS submits workflows to execution frameworks such as Pegasus and OODT to run workflows at large scale in distributed resources. Similarly, its main dependencies are `Git`, `Python 2.7`, `Java 1.8`, `Tomcat 8.5` and `Docker`. We also include the complete list of installed packages on the Dockerpedia GitHub repository.

[30] https://doi.org/10.5281/zenodo.1889342
[31] https://doi.org/10.5281/zenodo.1889336

5.7.1. MODFLOW-NWT

The USGS MODFLOW-NWT is a Newton-Raphson formulation for MODFLOW-2005 to improve solution of unconfined groundwater-flow problems. MODFLOW-NWT is a standalone program that is intended for solving problems involving drying and rewetting nonlinearities of the unconfined groundwater-flow equation. MODFLOW-NWT imports the Upstream-Weighting (UPW) Package for calculating intercell conductances, needs the Flow-property input for the UPW Package. The NWT linearization approach generates an asymmetric matrix. Figure 8 shows the three-step experiment workflow. The process starts by reading the MODFLOW-NWT model, next the workflow specifies in wich geographical zone the model is executed and finally the results are shown. Figure 9a and Figure 9b show the results generated by the original and reproduced execution evironments respectively. Both images indicate that the results are identical. Docker images for both workflows are in DockerHub [32] and the results are available in the GitHub repository of this paper [33].

5.8. Results and discussion

We executed the images for the workflows over their corresponding platforms, and each one of them used a different Docker version. Nonetheless, Docker guarantees that software will always run the same, regardless of underlying infrastructure and the Docker version.

All the executions were compared to their original one in a predefined VM image, where the execution environment was already in place. Results show that the container execution environments are able to execute their related workflows fully. To check that not only the workflows are successfully executed but also that the results are correct and equivalent, we checked their produced output data.

In the case of Montage, which produces an image as output, we used a perceptual hash tool. The resulting image (0.1 degree image of the sky) against the one generated by the baseline execution, obtaining a similarity factor of 1.0 (over 1.0) with a threshold of 0.85.

In SoyKB and internal extinction workflows, the output data is non-deterministic due to the existence of probabilistic steps. In this case, the use of a hash method is unfeasible. Hence, we validated the correct execution of the workflow by checking that correct

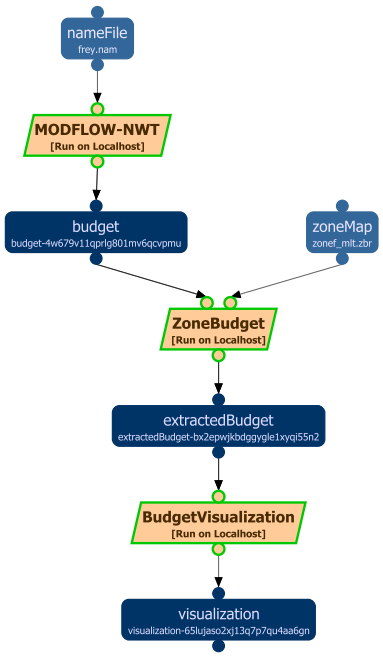


Fig. 8. MODFLOW-NWT workflow representation which we run locally. The workflow we run is identical to the existing in [27] and each workflow node executes an action for which different software components are needed. Descriptions are needed to ensure correct workflow reproducibility.

output files were actually produced, and that the standard errors produced by the applications did not contain any error message. In both cases the results obtained in each infrastructure were equivalent in terms of their size (e.g., number of lines) and content.

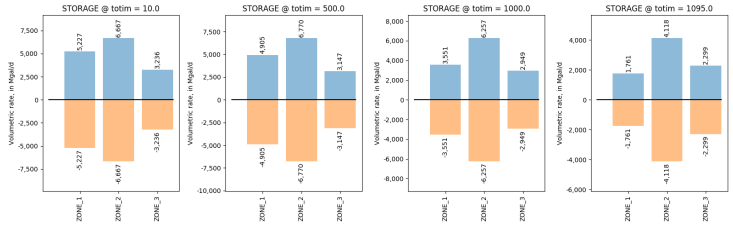
In the case of MODFLOW-NWT, which produces a histogram by zone, so we can compare it easily. The resulting histograms are the same between the original and reproduced environment.

Experimental results show that our proposal can automatically detect the software components, related vulnerabilities, building steps, and specific metadata of scientific experiments in the form of Docker images. Also, the results show that it is possible to extend Clair to annotate other package managers. In particular, we extended for Conda Package Manager.

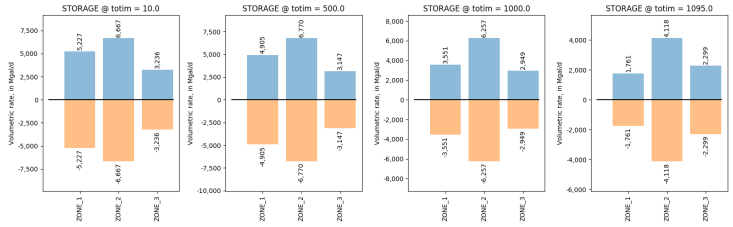
The annotations generated by our approach allow comparing the software components between two or more environments. This feature can be used as a debug tool when a reproduced environment did not work.

For example, On August 2018, we built the SoyKB workflow image, and we could execute the workflow successfully. However, we rebuilt a new image

[32] https://doi.org/10.5281/zenodo.1889348
[33] https://doi.org/10.5281/zenodo.1889328



(a) This image shows the results from the MODFLOW workflow execution by the Information Sciences Institute (ISI)



(b) This image shows the results from the MODFLOW workflow execution in our local system.

Fig. 9. The results from executing the MODFLOW workflow at the ISI and on our local machine after using are identical. We managed to reproduce the original workflow by using our semantic model and container virtualization using Docker images.

in November with the same DeploymentPlan and we were not able to run the workflow successfully.

We compared the software components inside both images and found the following differences:

- August image: Pegasus 4.8 and Java 1.7
- November image: Pegasus 4.9 and Java 1.8

Thus, we analyzed the code and documentation of SoyKB and the dependencies of Pegasus 4.9. As a result, we got the dependency graphs showed in the figure 10a. The dependency graphs show that the Pegasus 4.9 and SoyKB are not compatible due to their java version requirements

Thus, we built a new image installing Pegasus 4.8, and we got the dependency graphs showed in the figure 10a. Here, the graph does not have a conflict and we could run the workflow successfully. This new image was named: `pegasus_workflow_images:4.8.5`

The main reason for previous works to avoid physical conservation was the high demand for storage of virtual machines. As Docker images are lightweight we tackle this problem. The results on the disk usage comparison show that there is a 64.2% decrease for the Pegasus image and there is a 41.5% decrease for the dispel4py. Table 2 shows the difference for the workflow system Pegasus and dispel4py.

In summary, the results of our experiments are:

Approach	Pegasus (MB)	dispel4py (MB)
Virtual machine	1929	3509
Container	690	2050

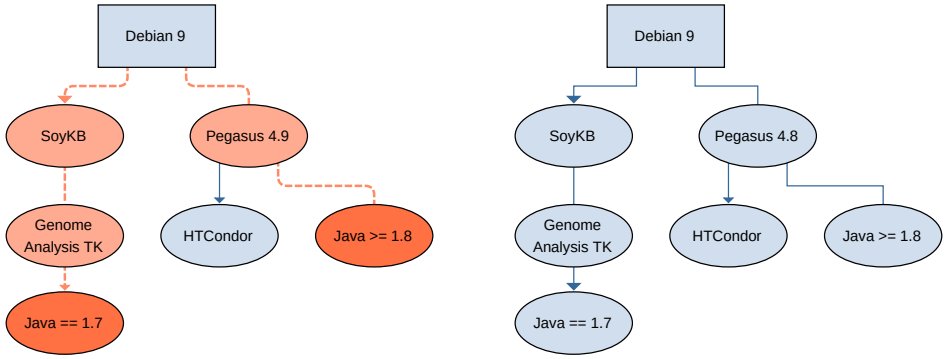
Table 2

Disk usage pegasus and dispel4py images.

- Docker Images allow the execution the selected scientific workflows.
- Our annotator correctly obtained the software components installed by the supported package managers without install new components inside the image or execute the experiment. Thus, our approach is secure and noiseless.
- We could reproduce the environment for the five workflows using the annotations obtained by our approach.
- We can detect the similarities and differences between two versions of a image. Furthermore, we used the feature to detect and solve a issue.
- The Docker images takes less disk space compare to Virtual Machine images

6. Conclusion and future work

In this work, we proposed an automatic tool to describe the software components using container-based environments and an approach that allows to conserve and reproduce computational environments of scien-



(a) Dependencies graph Pegasus 4.9 and SoyKB (b) Dependencies graph Pegasus 4.8 and SoyKB

Fig. 10. The orange nodes are the differences between the images

tific workflow executions using semantic models and virtualization techniques.

We conducted experiments to show the Docker Images are a lightweight unit and the images can be stored using public or private repositories such as DockerHub. Thus, we can ensure the physical conservation using these repositories. The utilization of Docker Images combined with our annotation process is a powerful approach to obtain the software components and building steps related to the environment. Also, these tools do not need to install new software components and without human intervention. Moreover, we prove that the approach can be extended to other package managers such as Conda. In conclusion, this is a unique contribution which successfully addresses both types of conservation of computational environments of scientific experiments. In regard to the reproducibility, our proposal can use these annotations, rebuild the computational environment and execute the related workflow. Also, the results of our experimentation showed that these annotations can detect software components issues (e.g., incorrect versions).

In summary, this work adopts, modifies and implements new tools building a framework to automate the process of obtaining the requirements of the computational environment, to then store these annotations according to semantic models. Finally, this proposal concludes that the use of these annotations allows the reproduction and detection of problems in the computational environment of a scientific experiment.

As for the future work, this work could be extended to support other container-based virtualization solutions, such as Singularity [28]. Also, it would be possible to improve the software components descriptions by providing a visualization of the dependency graph and the installed files within the experiment. Finally, a much more challenging contribution would be to detect workflow execution events using *perf events*[34] from the Linux Kernel, allowing to identify which workflow steps are being executed and what files are being accessed so a more fine grained analysis could be done.

References

[1] V.C. Stodden, Reproducible research: Addressing the need for data and code sharing in computational science, *Computing in science & engineering* **12**(5) (2010), 8–12.

[2] D. Garijo, S. Kinnings, L. Xie, L. Xie, Y. Zhang, P.E. Bourne and Y. Gil, Quantifying reproducibility in computational biology: the case of the tuberculosis drugome, *PloS one* **8**(11) (2013), 80278.

[3] 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.

[4] A. Dappert, S. Peyrard, C.C. Chou and J. Delve, Describing and preserving digital object environments, *New Review of Information Networking* **18**(2) (2013), 106–173.

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

[6] E. Deelman, K. Vahi, M. Rynge, G. Juve, R. Mayani and R. Ferreira da Silva, Pegasus in the Cloud: Science Automation through Workflow Technolo-

[34] https://perf.wiki.kernel.org/index.php/Main_Page

gies, *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. doi:10.1109/MIC.2016.15. http://dx.doi.org/10.1109/MIC.2016.15.

[7] 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.

[8] 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. ISBN 978-1-4673-4467-8. doi:10.1109/eScience.2012.6404482. http://dx.doi.org/10.1109/eScience.2012.6404482.

[9] 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**(C) (2014), 59–64, ISSN 1364-8152. doi:10.1016/j.envsoft.2014.06.022. http://dx.doi.org/10.1016/j.envsoft.2014.06.022.

[10] 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.

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

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

[13] 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), 1273.

[14] 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*, Springer, 2016, pp. 609–612.

[15] 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.

[16] R. Shu, X. Gu and W. Enck, A Study of Security Vulnerabilities on Docker Hub, in: *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, ACM, New York, NY, USA, 2017, pp. 269–280. ISBN 978-1-4503-4523-1. doi:10.1145/3029806.3029832. http://doi.acm.org/10.1145/3029806.3029832.

[17] D. Huo, J. Nabrzyski and C. Vardeman, Smart Container: an Ontology Towards Conceptualizing Docker., in: *International Semantic Web Conference (Posters & Demos)*, 2015.

[18] T. Lebo, S. Sahoo and D. McGuinness, PROV-O: The PROV Ontology, W3C Recommendation, https://www.w3.org/TR/prov-o/, 2013.

[19] R. Tommasini, B. De Meester, P. Heyvaert, R. Verborgh, E. Mannens and E. Della Valle, Representing dockerfiles in RDF, in: *ISWC2017, the 16e International Semantic Web Conference*, Vol. 1931, 2017, pp. 1–4.

[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] S. Harris and A. Seaborne, SPARQL 1.1 Query Language, W3C Working Draft 14 October 2010, http://www.w3.org/TR/2010/WD-sparql11-query-20101014/, 2010.

[22] 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), 1245–1252. doi:10.1093/nar/gkt905. http://dx.doi.org/10.1093/nar/gkt905.

[23] 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, XSEDE, 2014.

[24] G.B. Berriman, E. Deelman, J.C. Good, J.C. Jacob, D.S. Katz, C. Kesselman, A.C. Laity, T.A. Prince, G. Singh and M.-H. Su, Montage: a grid-enabled engine for delivering custom science-grade mosaics on demand, in: *Optimizing Scientific Return for Astronomy through Information Technologies*, Vol. 5493, International Society for Optics and Photonics, 2004, pp. 221–233.

[25] M. Osorio, dockerpedia/montage: thesis, 2018, doi: https://doi.org/10.5281/zenodo.1889360. doi:10.5281/zenodo.1889360. https://doi.org/10.5281/zenodo.1889360.

[26] 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.

[27] U.S.G. Survey, R.G. Niswonger, S. Panday and M. Ibaraki, MODFLOW-NWT, A Newton formulation for MODFLOW-2005, Technical Report, Reston, VA, 2011. doi:10.3133/tm6A51. https://pubs.usgs.gov/tm/tm6a37/.

[28] G.M. Kurtzer, V. Sochat and M.W. Bauer, Singularity: Scientific containers for mobility of compute, *PloS one* **12**(5) (2017), 0177459.