Use of Docker[☆] for deployment and testing of astronomy software

D. Morris, S. Voutsinas, N.C. Hambly and R.G. Mann

Institute for Astronomy, School of Physics and Astronomy, University of Edinburgh, Royal Observatory, Blackford Hill, EH9 3HJ, UK

Abstract

Lessons learned from using Docker for deployment and testing of astronomy software.

Keywords:

1. Introduction

In common with many sciences, survey astronomy has entered the era of "Big Data", which changes the way that sky survey data centres must operate. For more than a decade, they have been following the mantra of 'ship the results, not the data' (e.g. Quinn et al., 2004, and other contributions within the same volume) and deploying "science archives" (e.g. Hambly et al., 2008, and references therein), which provide users with functionality for filtering sky survey datasets on the server side, to reduce the volume of data to be downloaded to the users' workstations for further analysis. Typically these science archives have been implemented in relational database management systems, and astronomers have become adept in exploiting the power of their Structured Query Language (SQL) interfaces.

However, as sky survey catalogue databases have grown in size – the UKIDSS (Hambly et al., 2008) databases were 1–10 TB, VISTA (Cross et al., 2012) catalogue data releases are several 10s of TB as is the final data release from the Sloan Digital Sky Survey (DR12; Alam et al. 2015), Pan-STARRS1 is producing a ∼100 TB database (Flewelling, 2015), and LSST (Jurić et al. 2015; catalogue data volumes of up to 1 TB *per night*) will produce databases several Petabytes in size – the minimally useful subset of data for users is growing to the point where a simple filtering with an SQL query is not sufficient to generate a result set of modest enough size for a user to want to download to their workstation. This means that the data centre must provide the server–side computational infrastructure to allow users to conduct (at least the first steps in) their analysis *in the data centre* before downloading a small result set. The same requirement arises for data centres that wish to support survey teams in pro-

[☆] `https://www.docker.com`

Email address: `dmr,stv,nch,rgm@roe.ac.uk`

(D. Morris, S. Voutsinas, N.C. Hambly and R.G. Mann) cessing their imaging data (with data volumes typically 10 to 20 times larger than those quoted above for catalogue data sets). In both cases data centre staff face practical issues when supporting different sets of users running different sets of software on the same physical infrastructure (e.g. Gaudet et al. 2009).

These requirements are not, of course, peculiar to astronomy, and similar considerations have motivated the development of Grid and Cloud Computing over the past two decades. A pioneering example of the deployment of cloud computing techniques for astronomy has been the CANFAR project (Gaudet et al., 2009, 2011; Gaudet, 2015) undertaken by the Canadian Astronomy Data Centre and collaborators in the Canadian research computing community. The current CANFAR system is based on *hardware virtualization*, where the data processing software and web services are run in virtual machines, isolated from the details of the underlying physical hardware.

Following on from the development of system based on *hardware virtualization* the past few years have seen an explosion of interest within both the research computing and commercial IT sectors in *operating-system-level virtualization*, which provides an alternative method of creating and managing the virtualized systems.

A lot of the most recent activity in this field has centred on *Docker* and this paper presents lessons learned from two experiments we have conducted with Docker, a simple test of its capabilities as a deployment system and a more complicated one connecting a range of Virtual Observatory (VO; Arviset et al. 2010) services running in separate Docker containers.

Even by the standards of large open source projects, the rise of Docker has been rapid, and its development continues apace. A journal paper cannot hope, therefore, to present an up-to-date summary of Docker, nor an authoritative tutorial in its use, so we attempt neither here. Rather, we aim to describe the basic principles underlying Docker, and to contrast its capabilities with the virtual machine (VM) technologies with which astronomers may be more familiar, highlighting where operating–system–level (OS–level) virtualization provides benefit for astronomy. We illustrate these benefits through describing our two experimental Docker projects and the lessons we learned from undertaking them. Many of the issues we encountered have since been solved as the Docker engine and toolset continue to evolve, but we believe there remains virtue in recounting them, both because they illustrate basic properties of Docker containers and because they show how the Docker community operates.

For the sake of definiteness, we note the development of the systems described in this paper were based on Docker version **1.6** and that we discuss solutions to the issues we encountered that have appeared up to version **1.10**.

The plan of this paper is as follows. In Section 2 we describe hardware and OS–level virtualization, summarising the differences between the two approaches, and in Section 3 we introduce Docker as a specific implementation of OS-level virtualization. Section 4 describes our first Docker experiment, in which it was used to create a deployment system for the IVOATEX Document Preparation System (Demleitner et al., 2016), while Section 5 describes the use of Docker in the development and deployment of the *Firethorn* VO data access service (Morris, 2013). Finally, Section 7 summarises the lessons learned from these experiments and discusses the place that Docker (or similar technologies) may develop in astronomical data management. The development of Docker is taking place at a very rapid pace, with many substantial contributions made through blog posts and other web pages, rather than through formal journal papers, so we present in Appendix A a list of the online sources of information that we have found useful during our work to date with Docker.

2. Virtual machines and containers

The physical hardware of a large server may have multiple central processor units, each with multiple cores with support for multiple threads and access to several hundred giga-bytes of system memory. The physical hardware may also include multiple hard disks in different configurations, including software and hardware RAID systems, and access to Network Attached Storage (NAS). However, it is rare for a software application to require direct access to the hardware at this level of detail. In fact, it is more often the case that a software application's hardware requirements can be described in much simpler terms. Some specific cases such as database services dealing with large data sets may have specific hardware requirements for disk access but in most cases this still would represent a subset of the hardware available to the physical machine.

Virtualization allows a system administrator to create a virtual environment for a software application that provides a simplified abstract view of the the system. If a software application is able to work within this abstract environment then the same application can be moved or redeployed on any platform that is capable of providing the same virtual environment, irrespective of what features or facilities the underlying physical hardware provides. This ability to create standardized virtual systems on top of a variety of different physical hardware platforms formed the basis of the Infrastructure as a Service (IaaS) cloud computing service model as exemplified by the large scale providers like Amazon Web Services (AWS)[1]. The interface between customer and service provider is based on provision of abstract virtual machines. The details of the underlying hard-

[1]`https://aws.amazon.com/`

ware platform and the infrastructure required to provide network, power and cooling are all the service provider's problem. What happens inside the virtual machine (VM) is up to the customer, including the choice of operating system and software applications deployed in it.

With *hardware virtualization*, each VM includes a simulation of the whole computer, including the system BIOS, PCI bus, hard disks, network cards etc. The aim is to create a detailed enough simulation such that the operating system running inside the VM is not aware that it is running in a simulated environment. The key advantage of this approach is that because the guest system is isolated from the host, the guest virtual machine can run a completely different operating system to that running on the physical host. However, this isolation comes at a price. With hardware virtualization each VM uses a non–trivial share of physical system's resources just implementing the simulated hardware, resources which are no longer available for running the end user application software and services. Most of the time this cost is hidden from the end user, but it is most visible when starting up a new VM. With hardware virtualization the VM has to run through the full startup sequence from the initial BIOS boot through to the guest operating system initalization process, starting the full set of daemons and services that run the the background.

Comparing *hardware virtualization* with *OS virtualization* (Figure 1) we find a number of key differences between them, to do with what they are capable of and how they are used. A key difference is determined by the different technologies used to implement the virtual machines. As we have already described, in hardware virtualization the host hypervisor creates a full simulation of the guest system, including the system hardware, firmware and operating system. With operating-system-level virtualization the physical host operating system and everything below it, including the system firmware and hardware, is shared between the host and guest systems. This imposes a key limitation on OS virtualization in that the host and guest system must use the same operating system. So for example, while a Linux host system can use OS virtualization to support guests running different Linux distributions and versions, it cannot use OS virtualization to support a BSD or Illumos guest. However, if this limitation is not a problem, then sharing the system hardware, firmware and operating system kernel with the host system means that supporting OS VMs, or *containerisation*, represents a much lower cost in terms of system resources. This in turn leaves more of the system resources available for running the end user application software and services.

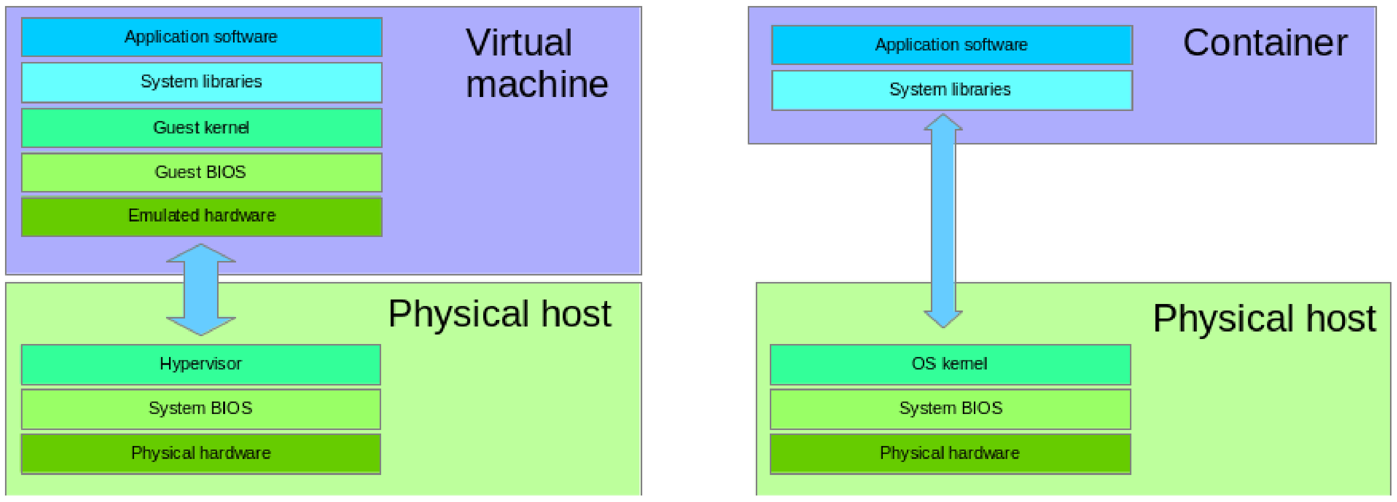

Figure 1: Comparison between (on the right) Hardware and (on the left) OS virtualization.

3. Docker

Docker☆ is emerging as the technology of choice for VM containers (Yu and Huang, 2015; Wang et al., 2015). Docker is an operating system level virtualization environment that uses software containers to provide isolation between applications. The rapid adoption and evolution of Docker from the initial open source project launched in 2013[2] by 'platform–as–a–service' (PaaS) provider dotCloud[3], to the formation of the Open Container Initiative[4] in 2015[5] suggests that Docker met a real need within the software development community which was not being addressed by the existing tools. As an aside it is interesting to note that the technologies behind OS virtualization have been available for a number of years. For example Solaris containers have been available as part of the Solaris operating system since 2005, and *cgroups* and *namespaces* have been part of the Linux kernel since 2007.

Although both the speed and simplicity of Docker containers have been factors contributing to its rapid adoption, arguably it is the development of a standardized format for describing and managing software containers that has been the 'unique selling point' differentiating Docker from its competitors[6,7], and has been the main driving force behind the rapid adoption of Docker across such a wide range of different applications:

[2]`http://www.infoq.com/news/2013/03/Docker`

[3]`https://www.dotcloud.com/`

[4]`https://www.opencontainers.org/`

[5]`http://blog.docker.com/2015/06/open-container-project-foundation/`

[6]`http://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/`

[7]`http://www.americanbanker.com/news/bank-technology/why-tech-savvy-banks-are-gung-ho-about-container-softwa html/`

- at the end user level, Docker enables users to describe, share and manage applications and services using a common interface by wrapping them in standardized containers;
- from a developer's perspective, Docker makes it easy to create standard containers for their software applications or services;
- from a system administrator's perspective, Docker makes easy to automate the deployment and management of business level services as a collection of standard containers.

3.1. Docker, DevOps and MicroServices

In a 'DevOps' (development and operations) environment, software developers and system administrators work together to develop, deploy and manage 'enterprise' level services. Describing the deployment environment using Docker containers enables the development team to treat system infrastructure as code, applying the same tools they use for managing the software source code, e.g. source control, automated testing etc. to the infrastructure configuration. Moreover Docker has emerged as one of the key technologies for automating the continuous delivery and continuous deployment of MicroService based architectures:

> "in 2014, no one mentioned Docker . . . in 2015, if they don't mention Docker, then they aren't paying attention"

[The State of the Art in Microservices by Adrian Cockcroft, Jan 2015][8]

3.2. Reproducible science

In the science and research community, Docker's ability to describe a software deployment environment has the potential to improve the reproducibility and the sharing of data analysis methods and techniques:

- Boettiger (2014) describes how the ability to publish a detailed description of a software environment alongside a research paper enables other researchers to reproduce and build on the original work;
- Nagler et al. (2015) describes work to develop containerized versions of software tools used to analyse data from particle accelerators[9];
- the Nucletid project[10] provides reproducible evaluation of genome assemblers using docker containers;
- the BioDocker[11] project provides a curated set of bioinformatics software using Docker containers.

[8] `https://www.youtube.com/watch?v=pwpxq9-uw_0&t=160`

[9] `https://github.com/radiasoft/containers`

[10] `http://nucleotid.es/`

[11] `http://biodocker.org/docs/`

3.3. Compute resource services

There are two roles in which Docker may be useful in implementing systems which enable end users to submit their own code to a compute resource for execution within a data center. Docker can be used internally to provide the virtualization layer for deploying and managing the execution environments for the submitted code. This scenario is already being evaluated by a number of groups, in particular Docker is one of the technologies being used to deliver a PaaS infrastructure for the European Space Agency's Gaia mission archive (O'Mullane, 2016; Ferreruela, 2016).

Alternatively, Docker can be used as part of the public service interface, providing the standard language for describing and packaging the software. In this scenario, the user would package their software in a container and then either submit the textual description or the binary container image to the service for execution. The advantage of this approach is that the wrapping of analysis software in a standard container enables the user to build and test their software on their own platform before submitting it to the remote service for execution. The common standard for the container runtime environment means that the user can be confident that their software will behave in the same manner when tested on a local platform or deployed on the remote service.

3.4. Reproducible deployment

It is often the case that a development team do not have direct control over the software environment where their service will be deployed. For example, the deployment platform may be configured with versions of operating system, Java runtime and Tomcat webserver which are determined by the requirements of other applications already running on the machine and by the system administrators running the system. This can present problems when attempting to update the version of these infrastructure components. Unless it is possible to isolate the different components from each other then a system component cannot be updated unless all of the other components that interact with it can be updated at the same time.

With an OS virtualization technology like Docker, each application can be wrapped in a container configured with a specific version of operating system, language runtime or webserver. The common interface with the system is set at the container level, not at the operating system, language or application server level. In theory it is possible to do something similar using hardware virtualization VMs. However, in practice the size and complexity of the virtual machine image makes it difficult to do this in a portable manner.

In a container based approach to service deployment, the development process includes a container specifically designed for the service. The same container is used during the development and testing of the software and becomes part of the final project deliverable. The final product is shipped and deployed as the container, with all of its dependencies already installed, rather than as an individual software component which requires a set of libraries and components that need to be installed along with it. This not only simplifies the deployment of the final product, it also makes it more reproducible.

4. Deploying IVOATEX with Docker

As an early experiment in using containers to deploy applications, we used Docker to wrap the IVOATEX[12] document build system to make it easier to use. The IVOATEX system uses a combination of LaTeX tools and libraries, a compiled C program to handle LaTeX to HTML conversion, and a `makefile` to manage the build process.

The IVOATEX includes a fairly clear set of install instructions. However, the instructions are specific to the Debian Linux distribution and porting them to a different Linux distribution is not straight forward. In addition, it was found that in some instances configuring a system with the libraries required by the IVOATEX system conflicted with those required by other document styles.

Installing the full IVOATEX software makes sense for someone who would be using it regularly. However, installing and configuring all of the required components is a complicated process for someone who just wants to make a small edit to an existing IVOA document. In order to address this we created a simple Docker container that incorporates all of the components needed to run the IVOATEX system configured and ready to run. (The source code for the project is available on GitHub[13] and a binary image of the container is available from the Docker registry[14]).

The source for the project consists of a build file, the `Dockerfile`, which starts with a basic Debian image and installs the required set of software libraries, including the C compiler, a set of HTML editing tools and the LaTeX tools and libraries needed to support the IVOATEX document build system. Due to the way that the IVOATEX build system is designed, the C cource code for the LaTeX to HTML translator is included as part of the document source and does not need to be included in the container.

To provide access to the document source, we use the `pwd` command to mount the current directory from the host system as `/texdata` inside the container.

[12] `http://www.ivoa.net/documents/Notes/IVOATex`

[13] `https://github.com/ivoa/ivoatex`

[14] `https://hub.docker.com/r/ivoa/ivoatex/`

```
docker run
    --volume "$(pwd):/texdata"
    "ivoa/ivoatex"
```

and once inside the container we can use the `make` commands to build the document.

```
cd /texdata
  make clean
  make biblio
  make
```

The initial idea for this project came as a result of reading about the work done by Jessie Frazelle on using Docker to wrap desktop applications [Docker Containers on the Desktop, Jessie Frazelle, Feb 2015][15].

At the time when this project was originally developed, using Docker in this way revealed a significant security issue.

When run from the command line like this, the Docker `run` command does not run the container directly, instead it uses a socket connection to send the run command to the Docker service, which runs the container on your behalf. A side effect of this is that the defult user `id` inside the container, *root*, has *root* privileges outside the container as well. This normally isn't a problem, unless you use the `volume` option to make a directory on the host platform accessible from inside the container, which is exactly what we need to do to enable the ivoatex tools to access the document text.

In our case, this doesn't prevent our program from working, but it does mean that the resulting PDF and HTML documents end up being owned by *root*, which make it difficult for the normal user to delete them.

To solve this we used a shell script to intercept the entrypoint command and use the `sudo` command to set the user `id` before running the main command. To set this to the current user `id`, we add an `/env` option to pass the current user id to the `notroot` startup script.

```
docker run
    --env "useruid=$(id -u)"
    --volume "$(pwd):/texdata"
    "ivoa/ivoatex"
```

However, this does highlight a more generic and potentially more serious problem. Consider the following commands.

If we create a standard Debian container, and mount the `/etc` directory from the host system as `/albert` inside the container.

```
docker run
    --volume "/etc:/albert"
    "debian"
        bash
```

Then, inside the container, we run the `vi` text editor and edit the file `/albert/passwd`.

[15] `https://blog.jessfraz.com/post/docker-containers-on-the-desktop/`

```
    vi /albert/passwd
```

The result of the `volume` mount means that `vi` running inside the container is editing the `passwd` file outside the container, with *root* privileges on the host system.

It is important to note that this issue is not caused by a security weakness in the container or in the Docker service. The issue occurs because the user that runs a container has direct control over what resources on the host system that container is allowed to access. Without the `/volume` mount, the container would not be able to access any files on the host system and there would be no problem. This is not normally an issue, because users would not normally have sufficient privileges to run Docker containers from the command line. In most cases users would not run containers directly on a production system, they would be given access to a container management program like Kubernetes[16] or OpenStack[17] to manage their containers. In addition, most Linux distributions now have security constraints in place which prevent containers from accessing sensitive locations on the file system. For example, on RedHat based systems the SELinux security module prevents containers from accessing a location on the file system unless it has explicitly been granted permission to do so.

Finally, it is important to note that this problem *was* an issue when we were first developing the *ivoatex* container, in March 2015. Since then, container technology has continued to evolve and there has been significant progress in a number areas that addresses this issue. In particular the work within Docker on user namespaces[18,19], but also the work in the Open Containers project[20] enabling containers to run as non-privileged users on the host system[21,22].

5. Docker in Firethorn

5.1. Firethorn overview

The goal of the Firethorn project is to enable users to run queries and store results from local astronomy archive or remote IVOA relational databases and share these results with others by publishing them via a TAP service [23]. The project has it's origins in a prototype data resource federation service (Hume et al., 2012) and is built around the Open Grid Service Architecture Data Access Infrastructure (OGSA-DAI; Holliman et al. 2011 and references therein).

[16]`http://kubernetes.io/`

[17]`https://www.openstack.org/`

[18]`https://integratedcode.us/2015/10/13/user-namespaces-have-arrived-in-docker/`

[19]`https://docs.docker.com/engine/reference/commandline/dockerd/#daemon-user-namespace-options`

[20]`https://runc.io/`

[21]`https://github.com/opencontainers/runc/issues/38`

[22]`https://blog.jessfraz.com/post/getting-towards-real-sandbox-containers/`

[23]`http://www.ivoa.net/documents/TAP/`

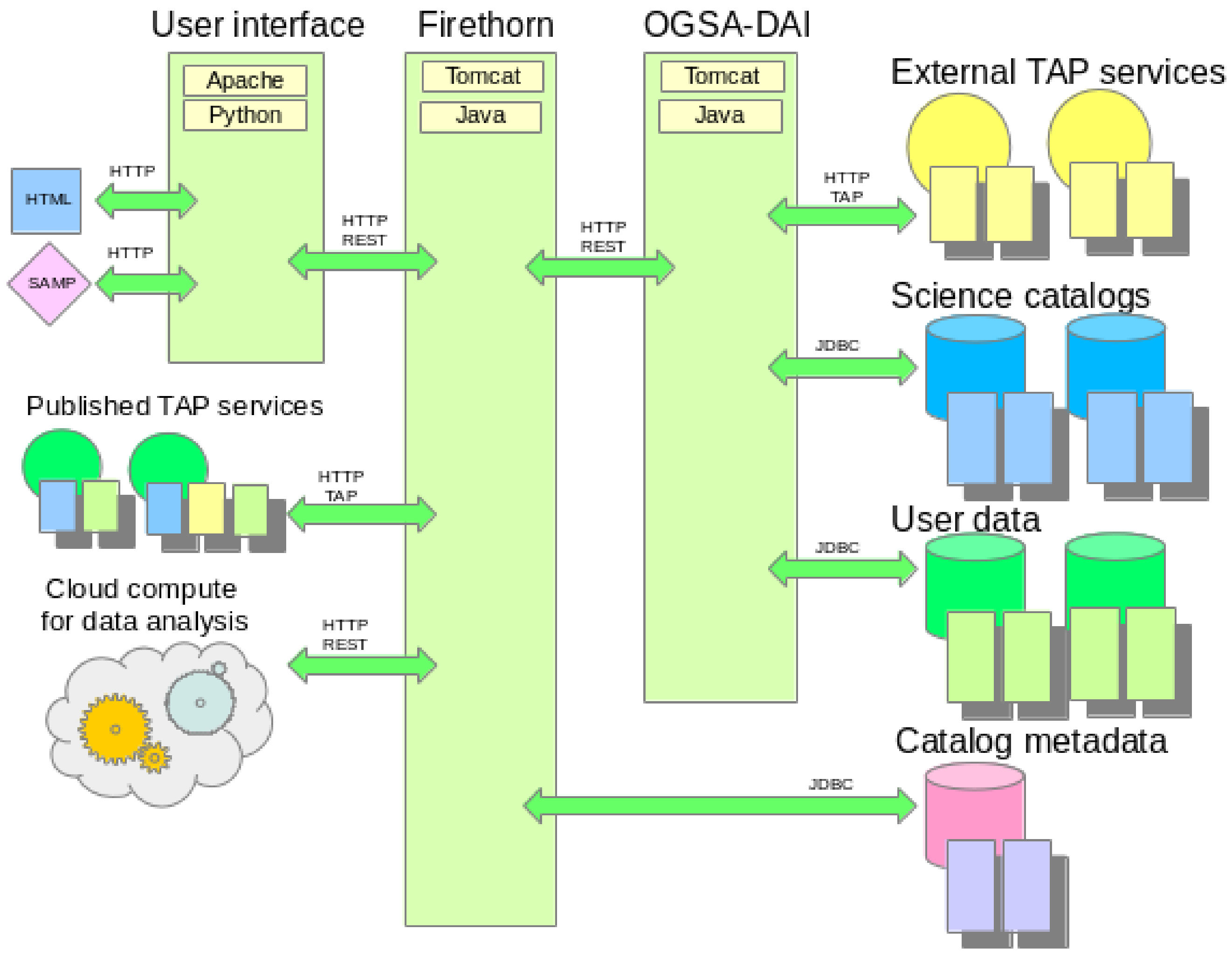

Figure 2: Firethorn architecture illustrating the components connecting local data resources and those distributed on the wide–area network.

The system architecture consists of two separate Java web services, one for handling the abstract ADQL catalog metadata, and one for handling the SQL queries and processing the results, two SQLServer databases, one for storing the catalog metadata and one for storing the user data, a Python user interface web service, and a Python testing tool. A schematic representation is shown in Figure 2.

5.2. Virtual Machine allocation and Containerization

At the beginning of the project we assigned a full KVM[24] virtual machine to each of our Java web services, connected to a Python webapp running on the physical host which provided the user interface web pages (see Figure 3; each VM was manually configured).

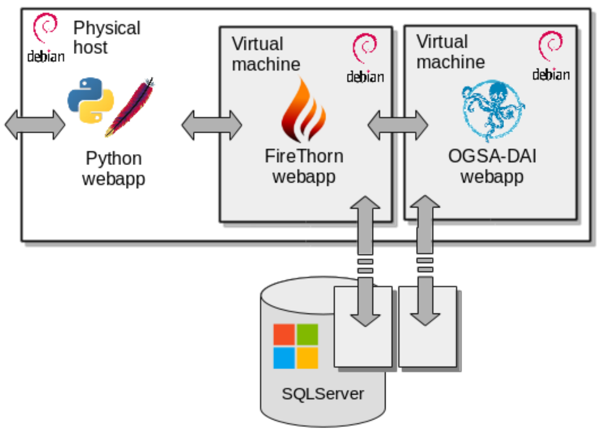

Figure 3: Manually configured VM for each web application.

[24]http://www.linux-kvm.org/page/Main_Page

Assigning a full virtual machine to each component represented a fairly heavy cost in terms of resources. However, at the time, this level of isolation was needed to support the different versions of Python, Java and Tomcat required by each of the components. Using virtual machines like this gave us an initial level of isolation from the physical host machine configuration. In theory it also allowed us to run more than one set of services on the same physical platform, while still being able to configure each set of services independently without impacting other services running on the same physical hardware.

However, in practice it was not until we moved from using manually configured virtual machines to using a set of shell scripts based on the *ischnura-kvm*[25] project to automate the provisioning of new virtual machines that we were able to run multiple sets of services in parallel. Replacing the manually configured instances with the template based instances gave us the reliable and consistent set of platforms we needed to develop our automated integration tests (see Figure 4).

[25]https://github.com/Zarquan/ischnura-kvm

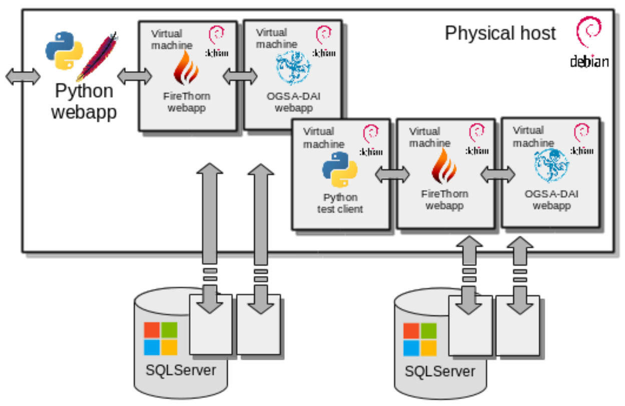

Figure 4: Multiple sets of scripted VM configurations.

The *ischnura-kvm* templates handle the basic virtual machine configuration such as cpu and memory allocation, network configuration, disk space and operating system.

Once the virtual machines were created, we used a set of shell scripts to automate the installation of the software packages needed to run each of our services. For our Java web-services, this included installing and configuring specific versions of the Java runtime[26] and Apache Tomcat[27]. The final step in the process was to deploy our web service and configure them with the user accounts and passwords needed to access the local databases.

The first stage of containerization was to create Docker containers for the two main Java/Tomcat web services, leaving the final Python webapp running in Apache web server on the physical host. The process of building the two Java/Tomcat web service containers was automated using the Maven Docker plugin[28] from Alex Collins[29]. Figure 5 illustrates this first stage.

[26]http://openjdk.java.net/

[27]http://tomcat.apache.org/

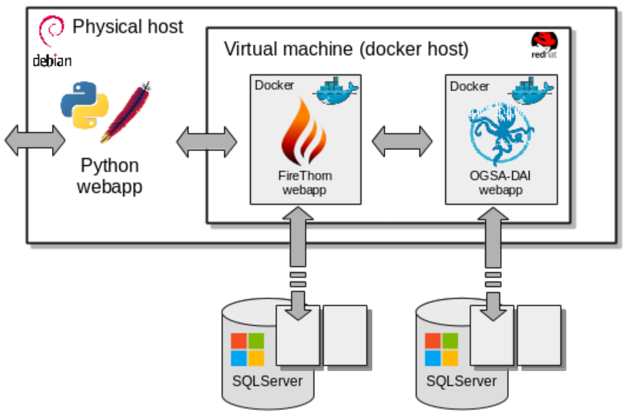

Figure 5: First stage containerization (Tomcats but not Apache).

5.3. Using pre-packaged or in–house base images

We ended up using our own containers as the base images for our Java and Tomcat web services, rather than the official Java[30] and Tomcat[31] images available on the Docker registry. The reason for this was partially as result of our early experiments with Docker where we explored different methods of creating containers from simple Linux base images but primarily because in the end this gave us more control over the contents of our containers. The flexibility of the container

[28]https://github.com/alexec/docker-maven-plugin

[29]https://github.com/alexec

[30]https://hub.docker.com/_/java/

[31]https://hub.docker.com/_/tomcat/

build system means that we were able to swap between base containers by changing one line in a Docker buildfile and re-building. This enabled us to test our containers using a variety of different base images, and work towards standardizing on a common version of Python, Java and Tomcat for all of our components.

Based on our experience we would recommend that other projects follow a similar route and define their own set of base images to build their containers, rather than using the pre-packaged images available from the Docker registry. The latter are ideal for rapid prototyping, but there are some issues that mean they may not be suitable for use in a production environment. Although the Docker project is working to improve and to verify the official images[32], there is still a lot of work to be done in this area. The main issue with using a pre-packaged base images is that the contents of containers are directly dependent on how the 3rd party image was built and what it contains. Unless full details of what the 3rd party image contains are available it can be difficult to asses the impact of a security issue in a common component such as OpenSSL[33,34] or glibc[35,36,37] has on a system that depends on an opaque 3rd party image.

[32]https://docs.docker.com/docker-hub/official_repos/

[33]http://heartbleed.com/

[34]https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160

[35]https://www.kb.cert.org/vuls/id/457759

[36]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7547

[37]http://arstechnica.co.uk/security/2016/02/extremely-severe-bug-leaves-dizzying-number-of-apps-anc

5.4. Ambassador Pattern

At this point in the project we also began to use the Docker ambassador pattern[38] for managing the connections between our webapps and databases. The idea behind the ambassador pattern is to use a small lightweight container running a simple proxy service like socat[39] to manage a connection between a Docker container and an external service.

In our case, the two socat proxies in Docker containers makes the relational database appear to be running in another container on the same Docker host, rather than on a separate physical machine. This enables our service orchestration scripts to connect our web services to our database server using Docker container links. The arrangement is shown schematically in Figure 6.

[38]http://docs.docker.com/engine/articles/ambassador_pattern_linking/

[39]http://www.dest-unreach.org/socat/

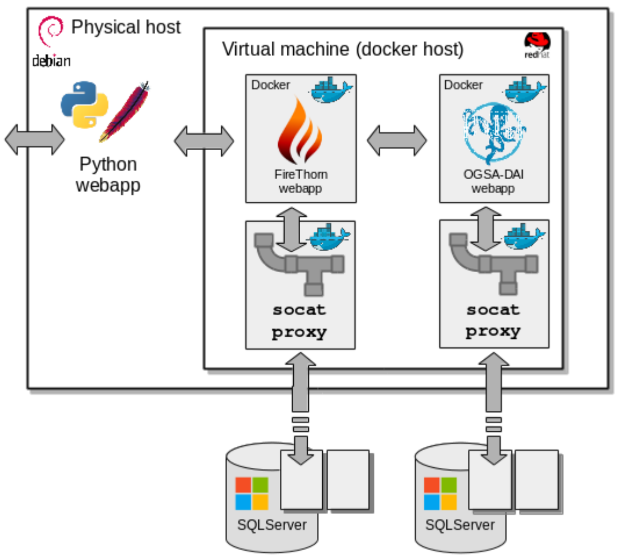

Figure 6: Socat ambassadors for connections to a relational database. .

At first glance, adding socat proxies like this may seem to be adding unnecessary complication and increasing network latency for little obvious gain. The benefit comes when we want to modify the system to support developers working remotely on platforms outside the institute network firewall who need to be able to run the set of services on their local system but still be able to connect to the relational database located inside the firewall. In this scenario (illustrated schematically in Figure 7) a small change to the Docker orchestration script replaces the socat proxy containers with proxy containers that use a tunneled ssh connection to link to the remote relational database located inside the institute network firewall.

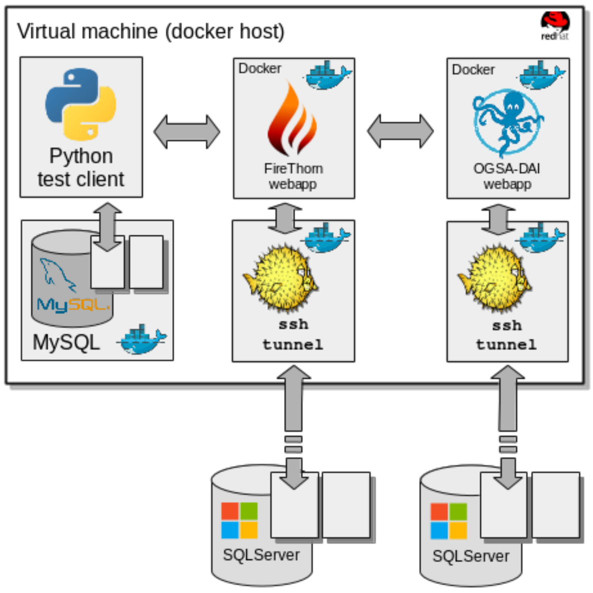

Figure 7: SSH ambassadors for connections to relational databases.

5.5. Python GUI and Python Testing

The final stage in the migration to Docker containers was to wrap the Python/Apache interface in a container and add that to our set of images. This is illustrated in Figure 8.

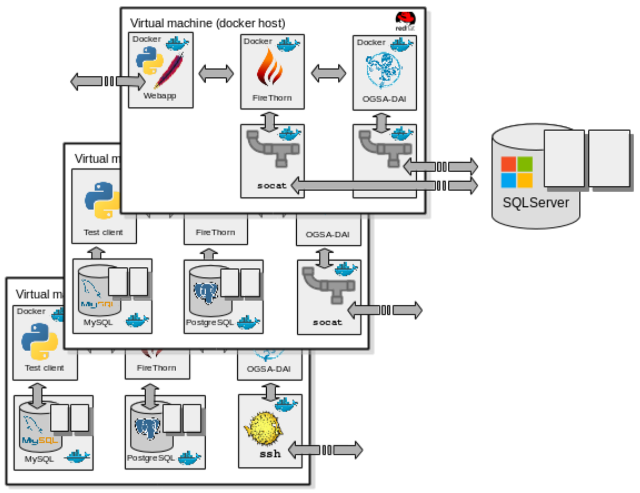

Figure 8: Multiple sets of containers, for live services and testing.

The Python-based webapp that provides the user interface consisted of an SQL proxy container that linked to our database for storing user queries, an Apache container which was built on an Ubuntu image, a base Python and a "python-libraries" container and finally a *webpy*[40] container, sitting at the top of the Apache/Python container stack.

An additional *webpy* web interface was later developed for a separate project (Gaia European Network for Improved User Services; e.g. Hypki and Brown 2016), which used the distributed querying feature of Firethorn. Because of the separation of the interfaces, Firethorn web services and databases into containers and the modular design of Docker systems, attaching this new interface container to a Firethorn docker chain was seamless. Linking a configuration file and startup script when running the webapp, a common technique when deploying webapp containers which makes the interchange of components in the system chain easier, was also used in both.

Another example of a top-level container used in our system, was the testing suite that we used to test our system for performance and accuracy, also written in Python. This consisted of a number of possible tests, which would each launch an instance of the Firethorn Docker chain, as well as a number of other required containers for each, either for databases to log results, or for loading and running the test suite code. By the end of the project we employed a set of bash scripts that allowed us to run a one–line command to start the required test, which we would run on any VM. The bash scripts were used to deploy and link the desired Docker container instances, however we have since learned of Docker Compose[41] which makes this process simpler. These were long running tests, which helped us gauge how the system run using Docker containers would behave and scale with large data volumes and long-term up-time and whether Docker as a technology was production–ready or not.

The result is a set of plug–and–play containers for each component in our system that can be be swapped and replaced with different versions or different implementations by modifying the scripts that manage the container orchestra-

[40]http://webpy.org/

[41]https://docs.docker.com/compose/

tion.

A live deployment would include the Python webapp for the user interface, and use socat proxies to connect to the local relational databases. In the test and development scenarios we replace the Python/Apache webapp with a Python test client connected to a local MySQL database running in a container, and in some cases we also replaced the connection to the SQLServer metadata database for our FireThorn webapp with a PostgreSQL database running in a container.

5.6. Orchestrating build and deployment

All of our containers are managed by a set of shell scripts which are included and maintained as part of the project source code. The Docker build scripts and the container orchestration scripts required to build and deploy a full set of services for each of our use cases are all stored in our source control repository alongside the source code for the rest of our project. Automating the service deployment, and treating the build and deployment scripts as part of the core project source code is a key step towards implementing what is referred to as *Programmable Infrastructure* or *Infrastructure as code*[42,43].

[42]http://devops.com/2014/05/05/meet-infrastructure-code/

[43]https://www.thoughtworks.com/insights/blog/infrastructure-code-reason-smile/

5.7. Portability and Reproducibility

One of the key reasons for choosing Docker to deploy our systems was the level of portability and reproducibility it provides. Within our development process our software has to be able to run on a number of different platforms, including the developers desktop computer, the integration test systems and our live deployment system. In addition, a key requirement of our project is that the software must be able to be deployed at a number of different 3rd party data centres, each of which would have a slightly different operating system and runtime environment.

If we rely on manual configuration for the target platform and runtime environment, then it is almost inevitable that they end up being slightly different. Even something as simple as the version of Java or Tomcat used to run the web application can be difficult to control fully. We could, in theory, mandate a specific version and configuration of the software stack used to develop, test and deploy our software. In reality, unless the platform is created and managed by an automated process, then some level of discrepancy will creep in, often when it is least expected.

There are a number of different ways of achieving this level of automation. A common method of managing a large set of systems is to use an automated configuration management tool, such as

Puppet[44] or Chef[45], to manage the system configuration based on information held in a centrally controlled template. Another common practice is to use a continuous integration platform such as Jenkins[46] to automate the testing and deployment. These techniques are not exclusive, and it is not unusual to use an automated configuration management tool such as Puppet to manage the (physical or virtual) hardware platform, in combination with a continuous integration platform such as Jenkins to manage the integration testing, and in some cases the live service deployment as well. However, these techniques are only really applicable when one has direct control over the deployment platform and the environment around it. In our case, we knew that although we had control over the environment for our own deployments, we would not have the same level of control over deployments at 3rd party sites.

6. Issues found

It is of course expected that issues and problems arise when using new technologies for the first time. These might be caused by mistakes made while overcoming the learning curve or by software bugs in the technology itself, which may have not been uncovered yet while adoption of the technology is still growing, and all possible usages of it have not been visited yet. We document here an example of one of the issues we encountered, including how we solved it.

6.1. Memory issue

As part of our Firethorn project we developed a testing suite written in Python as mentioned above. This suite included some long–running tests, which iterated a list of user submitted SQL queries that had been run through our systems in the past, running the same query via a direct route to the RDBMS as well as through the new Firethorn system and comparing the results. This list scaled up to several thousand queries, which meant that a single test pass for a given catalogue could take several days to complete. The issue we encountered here was that the docker process was being killed after a number of hours, with *Out of memory* error messages. An initial attempt at solving the problem was to set memory limits to all of our containers, which changed the symptoms and then caused our main Tomcat container to fail with memory error messages. After a few iterations of attempting to run the chain with different configurations, the solution was found through community forums, when we discovered that several other people were encountering the same symptoms with similar setups. Specifically, the problem was due to a memory leak, caused by the logging setup the version of Docker that we were using (1.6). Output sent to the system *stdout* was being stored in memory causing a continuous buffer growth resulting in a

[44]`https://puppetlabs.com/`
[45]`https://www.chef.io/chef/`
[46]`https://wiki.jenkins-ci.org/`

memory leak[47,48].

7. Lessons learned

More important than an analysis of the issues themselves is the understanding of the process undertaken to discover and solve them. In the example described above, the solution that we adopted was to use the `volume` option to send the system output and logs from our container processes to a directory outside the container.

```
docker run
  ...
  --volume "/var/logs/firethorn/
      :/var/local/tomcat/logs"
  ...
  "firethorn/firethorn:2.0"
```

We learned several valuable lessons through the process of researching how other developers managed these problems, for example, the approach to logging where the logs of a container are stored separately from the container itself, making it easier to debug and follow the system logs. In addition, we benefited from learning how and why limiting memory for each container was an important step when building each of our containers.

A fix for this issue was added to the Docker source code in November 2015 [Cap the amount of buffering done by BytesPipe, GitHub pull request #17877, Nov 2015] [49] and released in Docker version 1.10.

In addition, Docker added a pluggable driver based framework for handling logging [Logging drivers, GiitHub pull request #10568, Mar 2015] [50] [The State of Logging on Docker: Whats New with 1.7, Jun 2015] [51] which provides much more control over how logging output from processes running in the container is handled [Configure logging drivers, Docker user manual] [52].

7.1. Docker community

An important point to make here, is in regard to the open-source nature and culture of Docker and the Docker community. The main takeaway from this was that both finding how to go about solving issues related to containers and figuring out how the preferred method of implementing a certain feature is easy enough as doing a search of the keywords related to what you need. This can be done by either using a generic search engine or visiting the sources where the main Docker community interaction takes place [General discussion, Docker Community Forums] [53] [Questions

[47]`https://github.com/docker/docker/issues/9139`
[48]`https://github.com/coreos/bugs/issues/908`

[49]`https://github.com/docker/docker/pull/17877`
[50]`https://github.com/docker/docker/pull/10568`
[51]`https://blog.logentries.com/2015/06/the-state-of-logging-on-docker-whats-new-with-1-7/`
[52]`https://docs.docker.com/engine/reference/logging/overview/`
[53]`https://forums.docker.com/c/general-discussions/general`

tagged with 'docker', Stack Overflow] [54] [Docker project, GitHub issues] [55].

Because Docker is an open source solution, it has an active open source community behind it which enables users to find and fix issues more efficiently. An open source community means it is more likely that any issue you might find has already been encountered by someone else, and just as likely that it has been solved officially (as part of a bug fix in Docker release) or unofficially (Community member:*Here is how I solved this problem*). Contrast this with encountering issues using some proprietary technology with a more limited number of users, with a much slower pace of updating versions and bug-fixing.

While Docker's source code is open to the public, perhaps more importantly so is its issue tracking system. Apart from the fact that issues will get raised and solved quicker naturally with more eyes on them, another advantage for the users of such a platform is that they get the opportunity to contribute and help steer the direction it takes, by either raising issues or adding comments to the issue tracking system or the discussion forums. This leads to the targets for each new release being closely tied with what the majority of the community raises as important issues or requests for future enhancements.

Another key point to note is how we benefited from Dockers support team as well as the number of early adopters. We decided to take up Docker at an early stage, which can be considered its "bleeding-edge" phase (Version 1.6), at which point it was more likely to discover issues. However, with the large team and strong technological support of its developers, as well as the significant number of early adopters, new releases to solve bugs or enhance usability and performance were issued frequently. Consequently, after some research, we realized that many of the issues we found, whether they could be considered bugs or usability improvements needed, were often fixed in subsequent releases, meaning that by updating our Docker version they would be solved.

7.2. Future of Docker in Astronomy and Science

Based on our experience in development and production for the Firethorn and IVOATEXprojects, we anticipate a rapid growth of interest and usage of Docker and container-based solutions in general. We expect that this will be the case for both developing and deploying systems as a replacement or complementary to exiting *hardware virtualization* technologies, in enabling reproducible science and in system that allow scientists to submit their own code to data centers.

In terms of the future of Docker in relation to the Open Container Initiative (OCI), there is the potential for a common container standard to emerge, with the Docker project playing a leading role in the shaping of this standard. It should

[54]`http://stackoverflow.com/questions/tagged/docker`

[55]`https://github.com/docker/docker/issues`

be noted that as explicitly stated by the OCI, given the broad adoption Docker, the new standard will be as backward compatible as possible with the existing container format. Docker has already been pivotal in the OCI by donating draft specifications and code, so we expect any standard that emerges from this process will be closely tied with what exists now in Docker.

8. Conclusion

As mentioned throughout this paper, some of the main takeaways we noted from the use of Docker in development and production are the ease it provides in bundling components together, providing re-usability, maintainability, faster continuous integration environments and better collaboration between developers.

In addition, the openness of Docker and its community has contributed to its popularity in both science and business systems.

We briefly mentioned reproducible science and results of experiments, which along with authorship bias has been discussed in the scientific community for a while, but without any clear solutions. While scientists are more frequently publishing their code and data, the ability to re-run an analysis and obtain the same results is often non-trivial. Docker can potentially help with this, as it provides the tools and simplicity that scientists need to recreate the environment that was used to generate a set of test results.

Docker is not the perfect solution, and scientists or system engineers must decide when and if it is a suitable tool for their specific needs. It is most applicable in situations where reproducibility and portability are high on the list of requirements.

When deciding on whether to adopt a container technology such as Docker our experience would suggest that the benefits in terms of re-usability, maintainability and portability represent a significant benefit to the project as a whole and in most cases we would expect the benefits to outweigh the costs in terms of learning and adopting a new technology.

Acknowledgements

The research leading to these results has received funding from

- The European Community's Seventh Framework Programme (FP7-SPACE-2013-1) under grant agreement n606740,
- The European Commission Framework Programme Horizon 2020 Research and Innovation action under grant agreement n. 653477
- The UK Science and Technology Facilities Council under grant numbers ST/M001989/1, ST/M007812/1, and ST/N005813/1

References

Alam, S., Albareti, F.D., Allende Prieto, C., Anders, F., Anderson, S.F., Anderton, T., Andrews, B.H., Armengaud, E., Aubourg, É., Bailey, S., et al., 2015. The Eleventh and Twelfth Data Releases of the Sloan Digital Sky Survey: Final Data from SDSS-III. Astrophys. J. Suppl. S 219, 12. doi:10.1088/0067-0049/219/1/12, arXiv:1501.00963.

Arviset, C., Gaudet, S., IVOA Technical Coordination Group, 2010. The ivoa architecture, version 1.0. IVOA Note, 23 November 2010. URL: http://www.ivoa.net/documents/Notes/IVOAArchitecture/index.html.

Boettiger, C., 2014. An introduction to Docker for reproducible research, with examples from the R environment. ArXiv e-prints arXiv:1410.0846.

Cross, N.J.G., Collins, R.S., Mann, R.G., Read, M.A., Sutorius, E.T.W., Blake, R.P., Holliman, M., Hambly, N.C., Emerson, J.P., Lawrence, A., Noddle, K.T., 2012. The VISTA Science Archive. aap 548, A119. doi:10.1051/0004-6361/201219505, arXiv:1210.2980.

Demleitner, M., Taylor, M., Harrison, P., Molinaro, M., 2016. The ivoatex document preparation system. IVOA Note, 30 April 2016. URL: http://www.ivoa.net/documents/Notes/IVOATex/index.html.

Ferreruela, V., 2016. Gavip gaia avi portal, collaborative paas for data-intensive astronomical science, in: Lorente, N.P.F., Shortridge, K. (Eds.), ADASS XXV, ASP, San Francisco. p. TBD.

Flewelling, H., 2015. Public Release of Pan-STARRS Data. IAU General Assembly 22, 2258174.

Gaudet, S., 2015. CADC and CANFAR: Extending the role of the data centre, in: Science Operations 2015: Science Data Management - An ESO/ESA Workshop, held 24-27 November, 2015 at ESO Garching. Online at https://www.eso.org/sci/meetings/2015/SciOps2015.html, id.1, p. 1. doi:10.5281/zenodo.34641.

Gaudet, S., Armstrong, P., Ball, N., Chapin, E., Dowler, P., Gable, I., Goliath, S., Fabbro, S., Ferrarese, L., Gwyn, S., Hill, N., Jenkins, D., Kavelaars, J.J., Major, B., Ouellette, J., Paterson, M., Peddle, M., Pritchet, C., Schade, D., Sobie, R., Woods, D., Woodley, K., Yeung, A., 2011. Virtualization and Grid Utilization within the CANFAR Project, in: Evans, I.N., Accomazzi, A., Mink, D.J., Rots, A.H. (Eds.), Astronomical Data Analysis Software and Systems XX, p. 61.

Gaudet, S., Dowler, P., Goliath, S., Hill, N., Kavelaars, J.J., Peddle, M., Pritchet, C., Schade, D., 2009. The Canadian Advanced Network For Astronomical Research, in: Bohlender, D.A., Durand, D., Dowler, P. (Eds.), Astronomical Data Analysis Software and Systems XVIII, p. 185.

Hambly, N.C., Collins, R.S., Cross, N.J.G., Mann, R.G., Read, M.A., Sutorius, E.T.W., Bond, I., Bryant, J., Emerson, J.P., Lawrence, A., Rimoldini, L., Stewart, J.M., Williams, P.M., Adamson, A., Hirst, P., Dye, S., Warren, S.J., 2008. The WFCAM Science Archive. Mon. Not. R. Astron. Soc. 384, 637–662. doi:10.1111/j.1365-2966.2007.12700.x, arXiv:0711.3593.

Holliman, M., Alemu, T., Hume, A., van Hemert, J., Mann, R.G., Noddle, K., Valkonen, L., 2011. Service Infrastructure for Cross-Matching Distributed Datasets Using OGSA-DAI and TAP, in: Evans, I.N., Accomazzi, A., Mink, D.J., Rots, A.H. (Eds.), Astronomical Data Analysis Software and Systems XX, p. 579.

Hume, A.C., Krause, A., Holliman, M., Mann, R.G., Noddle, K., Voutsinas, S., 2012. TAP Service Federation Factory, in: Ballester, P., Egret, D., Lorente, N.P.F. (Eds.), Astronomical Data Analysis Software and Systems XXI, p. 359.

Hypki, A., Brown, A.G.A., 2016. Gaia archive. ArXiv e-prints arXiv:1603.07347.

Jurić, M., Kantor, J., Lim, K., Lupton, R.H., Dubois-Felsmann, G., Jenness, T., Axelrod, T.S., Aleksić, J., Allsman, R.A., AlSayyad, Y., Alt, J., Armstrong, R., Basney, J., Becker, A.C., Becla, J., Bickerton, S.J., Biswas, R., Bosch, J., Boutigny, D., Carrasco Kind, M., Ciardi, D.R., Connolly, A.J., Daniel, S.F., Daues, G.E., Economou, F., Chiang, H.F., Fausti, A., Fisher-

Levine, M., Freemon, D.M., Gee, P., Gris, P., Hernandez, F., Hoblitt, J., Ivezić, Ž., Jammes, F., Jevremović, D., Jones, R.L., Bryce Kalmbach, J., Kasliwal, V.P., Krughoff, K.S., Lang, D., Lurie, J., Lust, N.B., Mullally, F., MacArthur, L.A., Melchior, P., Moeyens, J., Nidever, D.L., Owen, R., Parejko, J.K., Peterson, J.M., Petravick, D., Pietrowicz, S.R., Price, P.A., Reiss, D.J., Shaw, R.A., Sick, J., Slater, C.T., Strauss, M.A., Sullivan, I.S., Swinbank, J.D., Van Dyk, S., Vujčić, V., Withers, A., Yoachim, P., LSST Project, f.t., 2015. The LSST Data Management System. ArXiv e-prints `arXiv:1512.07914`.

Morris, D., 2013. Wide field astronomy unit (wfau) virtual observatory data access service. URL: `http://wiki.ivoa.net/internal/ivoa/interopmay2013applications/20130508-firethorn-007.pdf`.

Nagler, R., Bruhwiler, D., Moeller, P., Webb, S., 2015. Sustainability and Reproducibility via Containerized Computing. ArXiv e-prints `arXiv:1509.08789`.

O'Mullane, W., 2016. Bringing the computing to the data, in: Lorente, N.P.F., Shortridge, K. (Eds.), ADASS XXV, ASP, San Francisco. p. TBD.

Quinn, P.J., Barnes, D.G., Csabai, I., Cui, C., Genova, F., Hanisch, B., Kembhavi, A., Kim, S.C., Lawrence, A., Malkov, O., Ohishi, M., Pasian, F., Schade, D., Voges, W., 2004. The International Virtual Observatory Alliance: recent technical developments and the road ahead, in: Quinn, P.J., Bridger, A. (Eds.), Optimizing Scientific Return for Astronomy through Information Technologies, pp. 137–145. doi:`10.1117/12.551247`.

Wang, X.Z., Zhang, H.M., Zhao, J.H., Lin, Q.H., Zhou, Y.C., Li, J.H., 2015. An Interactive Web-Based Analysis Framework for Remote Sensing Cloud Computing. ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences , 43–50doi:`10.5194/isprsannals-II-4-W2-43-2015`.

Yu, H.E., Huang, W., 2015. Building a Virtual HPC Cluster with Auto Scaling by the Docker. ArXiv e-prints `arXiv:1509.08231`.

Appendix A. Sources of information on Docker

RGM: As noted before, we need to remove almost all of the footnotes - ideally replacing them with more conventional reference sources. However, the reality is clearly that the Docker community develops through posting online, so it seems reasonable to get the unusual step for a journal paper and list some of the online resources that you have found useful during this process, especially if they are likely to continue to be sources of salient information.

Appendix A. Open Container Initiative membership

- Cloud Services
  - Amazon web services - https://aws.amazon.co
  - Google - http://www.google.com/
  - Apcera - https://www.apcera.com/
  - EMC - http://www.emc.com/
  - Joyent - https://www.joyent.com/
  - Kyup - https://kyup.com/
  - Odin - http://www.odin.com/
  - Pivotal - http://pivotal.io/
  - Apprenda - https://apprenda.com/
  - IBM - http://www.ibm.com/

- Operating systems & software
  - Microsoft - http://www.microsoft.com/
  - Oracle - http://www.oracle.com/
  - CoreOS - https://coreos.com/
  - Redhat - http://www.redhat.com/en
  - Suse - https://www.suse.com/
- Container Software
  - Docker - https://www.docker.com/
  - ClusterHQ - https://clusterhq.com/
  - Kismatic - https://kismatic.io/
  - Portworx - http://portworx.com/
  - Rancher - http://rancher.com/
  - Univa - http://www.univa.com/
- Security
  - Polyverse - https://polyverse.io/
  - Scalock - https://www.scalock.com/
  - Twistlock - https://www.twistlock.com/
- Datacenter infrastructure
  - Nutanix- http://www.nutanix.com/
  - Datera - http://www.datera.io/
  - Mesosphere - https://mesosphere.com/
  - Weave - http://www.weave.works/
- Computing hardware
  - Intel - http://www.intel.com/
  - Dell - http://www.dell.com/
  - Fujitsu - http://www.fujitsu.com/
  - Hewlett Packard Enterprise - https://www.hpe
- Telecommunications hardware
  - Cisco - http://www.cisco.com/
  - Infoblox - https://www.infoblox.com/
  - Midokura - http://www.midokura.com/
  - Huawei - http://www.huawei.com/
- Telecommunications providers
  - AT&T - http://www.att.com/
  - Verizon Labs - www.verizonwireless.com/
- System Monitoring
  - Sysdig - http://www.sysdig.org/
- Finance
  - Goldman Sachs - http://www.goldmansachs.co
- Virtualization platforms
  - VMware - http://www.vmware.com/
- IOT Embedded Systems
  - Resin.io - https://resin.io/
- Social Media Platforms
  - Twitter - https://twitter.com/