UNIVERSITY OF CAPE TOWN

DEPARTMENT OF COMPUTER SCIENCE

CS/IT Honours Final Paper 2019

Title: Containers and Reproducibility in Computational Science

Author: Robert Perrott

Project Abbreviation: ConBEnv

Supervisor(s): Robert Simmonds

Category	*Min*	*Max*	Chosen
Requirement Analysis and Design	*0*	*20*	15
Theoretical Analysis	*0*	*25*	0
Experiment Design and Execution	*0*	*20*	0
System Development and Implementation	*0*	*20*	20
Results, Findings and Conclusion	*10*	*20*	10
Aim Formulation and Background Work	*10*	*15*	15
Quality of Paper Writing and Presentation	*10*		10
Quality of Deliverables	*10*		10
Overall General Project Evaluation (*this section allowed only with motivation letter from supervisor*)	*0*	*10*	
Total marks		**80**	

Containers and Reproducibility in Computational Science

Robert Perrott
Department of Computer Science
University of Cape Town
Cape Town, Western Cape, South Africa
prrrob006@myuct.ac.za

ABSTRACT

Computational science experiments are often difficult to reproduce. This is largely because the software environments they are run within have become increasingly complex and variable in recent decades. Software containers may improve reproducibility by encapsulating an experiment's code and all its dependencies into a single file that can be shared between researchers. This file can then be opened and the code run as if it is being run on the original researcher's computer. A key issue with this approach is that popular containers such as Docker allow the user to escalate to root privileges on the host server they are running on, which introduces intolerable security risks when these containers are deployed in High Performance Computing (HPC) systems. We designed and implemented a system that allows users to safely build containers on servers in HPC systems without the user being able to gain root access. We used Jenkins – a popular automation server – to remotely build the containers on the HPC system in keeping with the principles of Continuous Integration, Delivery and Deployment.

CCS CONCEPTS

• Computing Methodologies → Distributed Computing Methodologies → High-Performance Computing • Software and its Engineering → Software notation and tools → Containers • Applied Computing → Computational Science • Software and its Engineering → Software Creation and Management → Continuous Integration •

KEYWORDS

Containers, Reproducibility, High-Performance Computing (HPC), Continuous Integration, Continuous Delivery, Continuous Deployment

ACM Reference format:

Robert Perrott. 2019. Containers and Reproducibility in Computational Science. In *UCT Honors Projects 2019*.

1 INTRODUCTION

For a scientific hypothesis to be deemed valid, it must be reproducible. Reproducibility can be defined as the degree of similarity of results achieved by independent scientists following the same methodology [1].

Reproducibility in computational science is poor for several reasons. In recent decades, computational scientists have made use of a wider variety of software and increasingly complex methodologies. This has led to more and more complex dependencies in computational science software environments, making the reproduction of experiments more difficult [5]. The complete experiment environments are rarely shared by researchers [2]. If code is provided, researchers are usually expected to install and configure the software needed to run the code. This can be time-consuming and difficult, and is often outside the domain of the researcher's expertise. Documentation on how to install, build, and run code is often inadequate. Even a minor gap in the researcher's knowledge is often enough to discourage the researcher from continuing [6]. Running the original code with newer software versions can lead to compatibility issues and bugs. Even if bug fixes and updates are added to the code, this can change the behavior of the experiment. There are tools available that address the above problems, but adoption is low due to the difficulty and time involved in learning how to use them [4, 7].

Software containers offer a potential solution to the above problems. Software containers are units of software that encapsulate a software environment and all its dependencies into a single file that can be shared between users. This container file can then be opened and the code run as if it is being run on the original user's computer. No installation or configuration is needed.

A key issue with some software container solutions such as Docker is that users of the container are able to gain root access to the host machine. While this is usually not a concern in software development environments, this poses extreme security risks in High-Performance-Computing (HPC) systems. A user with root access has the same privileges as the system administrator and has unrestricted access to the hosts file systems. Even without malicious intent, a user could unknowingly wreak havoc with the system. [10]

Our project's aim is to create an environment in which software containers can be built on an HPC system without the user needing access to elevated privileges. We also aim to use Jenkins to automate the continuous integration, delivery and deployment of the containers onto server in the HPC system.

The broad importance of this project is to assist in improving reproducibility in computational science. Reproducibility is a fundamentally important feature of the scientific method as it prevents scientists from making claims that cannot be verified by other scientists [3]. Automating the building of the containers and following a CICD methodology will make the building of containers simpler and less error-prone. By creating a safe, easy-to-use and portable environment for running software containers on HPC systems we can allow researchers to share the complete

environments that they used to obtain their results. These environments can then be used by other scientists to reproduce their work.

2 POTENTIAL SOLUTIONS TO REPRODUCIBILITY ISSUES

Below we outline four potential paths to improving the reproducibility of computational science experiments.

2.1 Compilation

Traditionally researchers have been expected to compile the code themselves. All modern PCs can do this, no elevated privileges are needed and direct access to all necessary resources is readily available. However, due to the reasons mentioned in the introduction, the compile-it-yourself approach asks too much of researchers and leads to reproducibility issues. [13]

2.2 Workflow Software

Workflow software can be used to link together many software processes into one unified process. It can successfully manage software dependencies, access to data, and version changes. There are many well-developed workflow applications that are widely available and easy to use, but adoption of these applications remains low. There is little financial or professional incentive for researchers to adopt these solutions. Researchers tend to want to keep their research tools to themselves. Workflow software businesses generally do not allow users who do not own the software to use the solutions, limiting portability. Finally, an academic environment that demands regular publishing disincentivizes the addition of workflow software, which takes extra time and energy to add. [4]

2.3 Virtual Machines

A virtual machine (VM) is a program that simulates another computer. It runs its own operating system. The state of a virtual machine can be captured in a virtual machine image.

Researchers can capture their software environment in a VM image. This image will contain all the necessary software to run the code, with the correct dependencies and configurations. The user does not need to install any software or understand the dependencies. VMs have strong isolation, so it is safe to give users full privileges. [5]

This technique creates a “black box” between the user and the inner workings of the code. The user is unable to investigate, expand upon or edit the software environment.

There are also scalability issues. A user may want to use tools from multiple different studies, but if these tools are all contained in their own virtual machine images, pipelining them into a single tool would be impractical. [4]

VMs are data heavy and can degrade performance when used in HPC environments. VMs must have their own complete operating system, kernel, and system daemons. This is unnecessary for the purposes of our project and adds needless complexity and data. [5]

2.4 Containers

A container is an encapsulation of an application with all its dependencies; it includes all the code, tools, configuration files, and libraries needed to run the application. Containers have surged in popularity in the last decade and show potential in improving reproducibility in computational science. Researchers can package their software environment in a container image which users can then open, with the software already installed, tested and configured. [4]

Containers are more suited to our purposes than VMs. They are more lightweight, since they share resources with the host. For example, VMs require their own kernels while containers share a kernel with the host. Containers are portable: if software works in a container on one computer, it is guaranteed to work in a container of the same type on another computer. Their lightweight nature means that many containers can run in an HPC system without drastically hindering performance. Containers are fundamentally designed to ensure the portability of isolated software environments, which is what this project aims to achieve. Virtual machines, on the other hand, are designed to simulate an entire system, which is unnecessary this projects aims. [8]

The rest of this paper will focus on containers, since they are the most promising of these four possible solutions (2.1-2.4).

3 CONTAINER OPTIONS

Below we describe and evaluate the four container options that showed the most promise in achieving our aims.

3.1 Docker

Docker is the current industry standard for containers. It is a platform made up of Docker Engine and Docker Hub. Docker Engine is a client-server application that runs on the users machine. It creates and runs the containers. It has three components: the Docker daemon, an API that provides an interface between the daemon and the programs that use it, and a command line interface (CLI) client that is used to manage the Docker containers. Docker Hub is an online repository where users can upload, share and manage Docker images. [15]

A Docker container is a running instance of a template called a Docker image. A container is an instance of an image. Small text files called Dockerfiles specify how to build each image. This allows for smooth portability of Docker images. [8]

Applications that use multiple Docker containers can be orchestrated by a tool called Docker Compose. [15]

Docker is mainly used by software developers, whose primary goal is to package their software and its dependencies for quick and easy transfer between collaborating developers, ensuring that the code works consistently on each developer’s computer. For this use case, Docker is a good solution. [9]

However, currently Docker is not a viable container solution in HPC environments. The Docker Engine – and its Docker daemon component – is root owned. Each Docker container is a child of the Docker daemon. Since the user is able to interact with the daemon

directly from inside the Docker container, it is possible for the user to gain root access to the host system by manipulating the Docker daemon. A malicious or careless user with root access then has the full privileges required to alter the HPC system as they wish. These security concerns have prevented the adoption of Docker by computational scientists. [9]

3.2 Singularity

Singularity was developed in response to the shortcomings of container solutions such as Docker for use in HPC systems. Singularity images are built using Singularity recipe files, which are analogous to the Dockerfiles mentioned in section 3.1. The Singularity recipe file determines the operating system of the Singularity container, the software to be installed, files to be loaded from the host system and the general setup of the environment. [16]

All container data, including metadata lies within a single Singularity image file, which makes running many containers at scale in an HPC system more efficient. [10]

The Singularity Registry Sever is a web application designed for the management and sharing of Singularity images and containers. It tracks and logs Singularity image usage and provides visualizations of these metrics. It runs locally on a server.

The Singularity Registry Client is installed on the local server and allows commands to be run from the Command Line Interface (CLI) that handle the pushing and pulling of Singularity images from the Singularity Registry Server.

Singularity containers do not come with the same security risks as Docker containers when used in HPC systems. Singularity containers do not connect to a root owned daemon process, so there is no way that an unprivileged user of a Singularity container can gain root access. For this reason, Singularity containers have seen wide adoption for use in HPC systems. [10]

3.3 Shifter

Shifter is a container solution for HPC systems. Researchers have found that its flexibility allows for the use of complex and atypical HPC patterns. Shifter is compatible with Docker. It can process Docker images without requiring additional configuration by the user. This is advantageous due to the already large and still growing popularity of Docker. [9]

Getting shifter to work at scale can present challenges. A study [9] identified that the functions used to build the Shifter containers on the compute nodes of the HPC caused efficiency bottlenecks. This was caused by a large number of requests on the systems Lightweight Directory Access Protocol (LDAP) entries. The researchers resolved this by caching the LDAP entries, allowing faster lookups and removing the bottleneck.

Shifter containers have a rapid start up time. Testing has even indicated that the size of the container image is independent of start-up-time. This means that large container images can be built and run in HPC systems without drastically lowering efficiency. [9]

Setting up Shifter can involve complex configurations involving its resource manager and its daemon. It also does not provide full hardware abstraction. [10]

3.4 CharlieCloud

CharlieCloud is a software application that makes use of Linux namespaces to run Docker containers without the use of elevated privileges. It is developed specifically for use in HPC systems. It is based on the concept of User-Defined-Software-Stacks (UDSS). These are essentially the software environments that the users of CharlieCloud would encapsulate into a container. CharlieCloud aims to provide a standardized, portable and reproducible workflow that is as simple as possible. It should run on standard HPC systems with minimal configuration. [5]

Linux kernels have six namespaces that partition the kernel resources. A process linked to a namespace can only see the kernel resources associated with that namespace. Five of the six Linux namespaces are privileged namespaces. Root access is needed to access these namespaces. The *user* namespace is unprivileged. Users are free to start a process in the *user* namespace. This process, as well as its children, will have full access to the kernel resources associated with the *user* namespace, but no access to the kernel resources of the privileged namespaces. Essentially, the *user* namespace provides an isolated environment to run UDSS without root access and without affecting the host. [5]

A problem with this approach is that currently the *user* namespace is not supported by some Linux distributions such as Red Hat Enterprise Linux. HPC systems using these distributions would not be able to use CharlieCloud.

CharlieCloud is a small program with less than 900 lines of code altogether. This simplicity would desirable to researchers wishing to extend the program, but suggests that the program may lack the diversity of tools and features that other container solutions such as Docker offer.

Before they can run, CharlieCloud containers need to be extracted from Docker containers and configured by an external C script. This means that they are not truly portable. [10]

These problems with CharlieCloud must be addressed before it can be considered a viable solution to ensuring reproducibility in HPC environments.

4 CONTINUOUS INTEGRATION, DELIVERY, AND DEPLOYMENT

4.1 Terminology

4.1.1. Continuous Integration. Continuous integration is a software development methodology where developers commit to the mainline branch on a daily basis, and frequent builds ensure that these commits do not cause problems in the code. Automation can be used to build and test software immediately after each commit. This avoids code discrepancies and complex merging operations. It also means that unless recent commit has caused an error, the software is bug free. Continuous Integration

generally allows for a shorter release cycle and improved code quality. [11]

4.1.2. Continuous Delivery. Continuous Delivery is aimed at ensuring that software can be deployed to the production environment at any given time. The main goal of Continuous Delivery is to reduce the risks and costs of deployment to the production environment. [17]

4.1.3. Continuous Deployment. Continuous Deployment goes a step further than Continuous Delivery and actually deploys the software on the production environment. [17]

4.1.4. The CICD Pipeline. It is important to note that Continuous Delivery requires Continuous Integration, and Continuous Deployment requires Continuous Delivery. The sequence of Continuous Integration, Delivery and Deployment form what we shall refer to as the *CICD pipeline*. Here the CD stands for both Delivery and Deployment. The word Continuous implies that the entire pipeline – the process that takes code from a commit in version control to production – runs automatically. [12]

4.2 Jenkins

Jenkins is an open source automation server. In the context of CICD, it can be used to automate the building and testing of software after each commit, and the Jenkins SSH plugin can then be used to deliver the containers on remote hosts [12]. It comes with an easy-to-use GUI and can be extended with a variety of plugins.

Jenkins has been successfully used by several research teams to automatically monitor computational science software and is currently in use in many HPC systems. [12]

4.3 CICD with Singularity and Jenkins

Researchers at the University of Colorado Boulder have successfully used Singularity and Jenkins to adopt CICD practices in managing and updating the RMACC Summit supercomputer [12]. The researchers used Jenkins to continuously poll the code repository for new commits. When a new commit is made, Jenkins builds a Singularity container image. A Singularity test block for the image is then run. If it passes, Jenkins completes the build and pushes the container to the appropriate server. If it fails, Jenkins aborts the build and notifies the researchers via email that the build was unsuccessful.

Singularity was used for the user-facing side of the system rather than Docker due to the security concerns around users being able to gain root access to the system via the Docker Engine. The researchers did use Docker containers for the non-user-facing infrastructure though. The multitude of readily available Docker tools allowed for more nuanced and varied Jenkins jobs to be performed.

These researchers concluded that tools such as Singularity and Jenkins have great potential to increase the reliability and quality of software in HPC environments. [12]

5 HIGH-LEVEL SYSTEM DESIGN

Based on the literature reviewed, using Singularity as our main platform together with an automated build environment supplied by Jenkins seemed like the most desirable path forward. Users can safely build Singularity containers without root access. Singularity containers are scalable and well suited to use in HPC environments. Using Jenkins to automate the CICD pipeline will help our software be maintained and updated in response to frequently changing HPC environments. Singularity container images are highly portable and ensure the reproducibility of code.

We decided that it would be useful to containerize our entire system using Docker. This would make it more portable and modular, and would make its setup far easier. Since these Docker containers would not be directly accessible to our users, the security concerns around the use of Docker containers discussed in Section 3.1 do not apply here.

After reviewing the literature, we had a basic idea of the high-level design of the solution we would need to create. A user would make a commit to their version control repository (say, GitHub). This commit would be pulled by the Jenkins master. The Jenkins master would then allocate a job to one of the Jenkins Slaves. The slave would build a Singularity container and push the container to Singularity Registry. Finally, the container would be pulled from the Singularity Registry onto the production server.

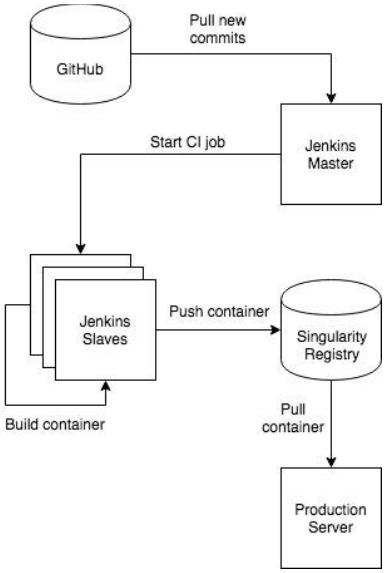

Figure 1: A high-level conceptualization of the system design

6 ETHICAL, LEGAL AND PROFFESIONAL ISSUES

All of the software we used – Singularity, Singularity Registry, Jenkins, Docker and Docker Compose – is open source. This means that we can use and modify the code without any Intellectual Property (IP) infringements.

We have a professional and ethical obligation to make sure that the information of our users is kept secure. Satisfying our aim to prevent any user of our infrastructure being able to escalate to root privileges will help to ensure this. We will also need to ensure that the links between the modules of our system (for example, the link between version control and Jenkins) are secure.

7 SYSTEM REQUIREMENTS

7.1 Security

7.1.1. No root access. As stated in our aim, users must not be able to escalate to root access from inside a container.

7.1.2. Secure links. The links between modules in our system will need to be secure. The Jenkins Master will need to access the GitHub repository in a secure manner. Containers must also be deployed on the production server in a way that does not compromise security.

7.1.3. Isolation from host. Singularity recipe files include commands which are run on the host system. These commands generally perform tasks such as copying files from the host system to the container, but the recipe file may contain commands such as "rm" (remove file) that pose risks to the security of the host system. We therefore need to make sure that the building of containers is sufficiently isolated from the host system.

7.2 CICD

The entire CICD pipeline should run automatically. This is the essence of *continuous* integration, delivery, and deployment. The time between commits to version control and the deployment of containers on target servers should be as short as possible and should involve no user interaction apart from the initial commit.

7.3 Ease of setup

The setup of our infrastructure by a system administrator should be as simple and easy as possible, with almost all configuration abstracted away from the administrator.

7.4 Portability

Our infrastructure should be portable. Its performance should be the same regardless of the server it is hosted on. Ideally it should be able to maintain its state when transferred from one server to another. This would allow maintenance to be performed on the original server without affecting our infrastructure, and would provide fault tolerance should the host server need to be taken down.

7.5 Usability

Our solution should be easy to use by both system administrators and users building containers. Any repeated effort should be automated away and as much of the system as possible should be hidden from the user. Almost all user interaction should simply involve feeding the necessary parameters to the system.

7.6 Reproducibility

In light of our broader aim of helping our users make their work reproducible, the containers we build will have to accurately replicate the user's software environment. The broad aim of this project is to assist in the accurate reproduction of computational science experiments. When a user encapsulates a software environment in a container and deploys it on a remote server using our system, the container should show the exact same behaviour as the original container. Superficial differences such as parameter names may change, but the functionality of the original container must be exactly replicated in order to provide true reproducibility.

8 DEVELOPMENT METHODOLOGY

At the onset of our project, we had no experience with any of the aforementioned technologies, and did not know how they would link together. We soon learnt that the creation of our solution would require learning how to use many ancillary tools such as Virtual Network Computers (VNCs) and new languages such as Groovy (for Jenkinsfiles) and YAML (for Docker-compose configurations). Figuring out the specifics of each step in our high-level design would require much experimentation, trying out different approaches, and learning as we went along.

8.1 Agile

These circumstances led to us adopting and Agile software development methodology, with particular emphasis on "working software over comprehensive documentation" and "responding to change over following a plan". As we learnt more about the tools we were using we would adapt quickly and change course. We would make sure that we had working software at frequent intervals, even if this software was a basic or even trivial prototype. We valued frequent face-to-face conversation over written communication or documentation. We strived to find and use the simplest solutions possible. At regular intervals we would discuss our roles in the project and make changes as the details of our project came into focus. [18]

8.2 Everything-as-Code

"Everything-as-code" is the practice of treating all parts of a system as code and storing the entire system in version control. In our case, this meant treating the entire CICD pipeline, including containers, the Jenkins master and slaves, and their surrounding Docker containers, as code. Docker images were determined solely by Dockerfiles. Singularity images were determined by Singularity recipe files. All Pipeline jobs were determined by Jenkinsfiles and Groovy scripts. The Jenkins master and slave nodes were themselves run in Docker containers, and so were the result of Dockerfiles.

Storing our entire solution as code has several benefits. It is highly portable; transferring our whole project only involves transferring text files. Changes to the infrastructure can be easily checked in

version control. There is also a single source of truth for each part of the infrastructure, so if there are errors in the system, they are always traceable to files in version control rather than external applications. [19]

9 SYSTEM DESIGN

9.1 Pipeline

Jenkins supports many job configurations. We decided that the Pipeline configuration was the option that best suited our needs. A CICD pipeline is an abstraction of the process that takes software from version control through to production and deployment. A Pipeline job in Jenkins is an automated expression of the CICD pipeline. Pipeline jobs are defined by Jenkinsfiles and Groovy scripts in keeping with the principle of "everything-as-code". A typical Pipeline job involves three stages – build, test, and deploy – although their configuration is highly flexible.

We extended Jenkins with the Blue Ocean plugin. This plugin provides an elegant UI specifically designed for the creation and analysis of Pipelines. When a Pipeline runs the UI displays a visualization of each stage, marking each as successful or unsuccessful. This feature makes it easier to trace the source of errors in the CICD pipeline. Use of the Blue Ocean plugin also made it easier to link our Pipeline jobs to version control.

9.2 Jenkins Architecture

We decided to use a master-slave architecture for Jenkins. The Jenkins master hosts a Web User Interface (UI) at port 8080 on the host VM and the Jenkins Application Programming Interface (API) at port 50000. The master schedules build jobs, dispatches jobs to the slaves, monitors the health of the slaves, gathers results of slave builds and presents these results to the Jenkins user. The Jenkins slaves are effectively Java executables running on remote servers in the network. Slaves are responsible for executing Pipeline jobs. Slaves can run on any popular operating system. The Jenkins master communicates with slaves via the TCP/IP protocol.

We chose this architecture because our solution will be deployed on an HPC system. A master-slave architecture will allow Pipeline jobs to be run at scale. Different slaves can also be configured differently and make use of different software according to the needs of the Pipeline. Having multiple slaves also means that Pipelines can build containers on nodes with different operating systems.

9.3 Version Control

We chose to use GitHub as our version control service. This is due to our familiarity with the service and due to its widespread adoption, meaning that most users will be familiar with it as well. It also lends itself well to integration with Jenkins Pipeline jobs.

9.4 GitHub-Jenkins Link

The Jenkins master container runs locally, and is not accessible to the external internet. This presents a challenge in linking GitHub to the Jenkins master so that the master can pick up new commits to the repository. We explored several different ways around this problem.

9.4.1. Webhook Relay. A webhook is a user-defined HTTP call-back. An event on one site (e.g. a commit to a GitHub repository) causes the site to make an HTTP request to another site, the URL of which is defined in the webhooks configuration. The target site (e.g. our Jenkins master at localhost:8080) can then accept the HTTP request, receive the relevant information and take the appropriate action (e.g. triggering a Pipeline build job). Webhook Relay is a service that allows the user to set up webhooks without requiring that the target site has a public IP address. It does this by generating a Webhook Relay URL that forwards HTTP requests to the site on the internal network. This Webhook Relay URL can be entered into a GitHub Webhook, which can then be configured so that each time a commit is made to GitHub, this commit is pushed to the target site, which is the Jenkins master in our case. This method was simple to implement and worked well for some Jenkins job configurations, however, it was not compatible with Pipeline, our configuration of choice. [20]

9.4.2. Nginx Reverse Proxy. Our next approach was to set up a reverse proxy using Nginx. A reverse proxy receives requests from the external internet and forwards them to servers in an internal network. Nginx is an application that facilitates the configuration of reverse proxies. This method required the use of a publicly accessible domain name to act as the reverse proxy between GitHub and the Jenkins master. A commit to GitHub would be picked up by the site at this public domain name. This site would then send the commit over the internal network to the Jenkins master. Configuring this reverse proxy was time consuming, and required the installation and configuration of ancillary software, such as Lets Encrypt to obtain an SSL certificate for the reverse proxy domain name. It also requires the purchase of a domain name. Once it was set up, the reverse proxy could trigger certain Jenkins jobs, but again, it was not able to trigger a Pipeline build. [21]

9.4.3. Blue Ocean. Finally, we discovered the Jenkins Blue Ocean plugin. Linking a Pipeline to a GitHub repository via Blue Ocean is extremely fast and simple. The user of our solution need only generate a personal access token on GitHub. This personal access token can then be given to the Pipeline when it is first created. The Pipeline will then have access to the GitHub repository. A time period can then be set specifying how often the Jenkins master must poll the GitHub repository to check for new commits. Now, whenever a new commit is detected, the Jenkins master will notice it and trigger a Pipeline job on a slave. Due to the simplicity of this method and the fact that Pipeline was our job configuration of choice, this is the method we used. One disadvantage to this method is that unlike the Webhook Relay and reverse proxy options, commits will not be automatically noticed immediately, only at the specified polling intervals. This should not be a problem for most users, who would not need the entire CICD pipeline to run after every commit. Should they wish to run the CICD pipeline, they can do so manually via the Jenkins master UI within a matter of seconds, by clicking on the Pipeline Job and hitting "Scan Repository Now" in the Pipeline menu.

9.5 Jenkins Shared Library

Usually a Pipeline job is completely defined by a Jenkinsfile in version control. Since our Pipeline process is the same for each user, this approach would mean that each user's Jenkinsfile would be exactly the same, besides superficial differences such as parameter names. Our users would probably not know how to write Jenkinsfiles, and having to learn how to do so may be a barrier to the adoption of our software.

We therefore decided to make use of a Shared Library. A Shared Library is a repository in a version control system that contains pieces of code that can be reused in carrying out Jenkins jobs. Instead of users defining their own jobs, they can use code from the Shared Library. Our Shared Library contains a Groovy script that defines all Pipeline jobs. Users still need a Jenkinsfile in their version control (see Figure 3), but this Jenkinsfile does not specify the entire Pipeline, it simply provides the parameters that need to be fed to the Groovy script in the Shared Library.

```
Pipeline(
  client: 'registry',
  username: 'robbieperrott',
  token: '4f7c7746494e340cdb31abef2a398276c6b1a771',
  image: 'test.simg',
  recipe: 'Singularity.recipe',
  collection: 'test-collection',
  container: 'test-container')
```

Figure 2: A sample Jenkinsfile. The user need only provide the parameters to be passed to the Groovy script in the Shared Library.

9.6 The Singularity Registry

Singularity Registry Server is a locally deployed web application that is used to manage Singularity containers. In the registry containers can be grouped in collections, and each collection can be made available to a set of users. User-privilege hierarchies can be defined that determine which users may access which collections.

9.7 Singularity Registry Client

The Singularity Registry Client is installed on the local server and allows commands to be run from the Command Line Interface (CLI) that handle the pushing and pulling of Singularity images from the Singularity Registry Server. It is installed on the Jenkins slaves, since these need to push the Singularity containers to the Singularity Registry Server.

9.8 Jenkins SSH Plugin

The Jenkins SSH plugin allows Jenkins to run shell commands on a remote server via the Secure Shell (SSH) protocol. The Jenkins master uses the SSH plugin to call a Singularity Registry Client pull command on the target VM. This VM then pulls the appropriate containers from our Singularity Registry Server.

9.9 Docker Compose

We decided to containerize our Jenkins master, Jenkins slaves and the Singularity Registry Server using Docker containers. This design choice was made to make our solution more portable and modular, and to simplify its setup.

We used Docker Compose – an orchestration tool for multi-container Docker applications – to integrate the above containers and allow them to communicate with each other.

10 INFRASTRUCTURE SETUP

Our project's infrastructure can be set up as follows. A Docker Compose YAML configuration file is run using the "docker-compose up" command. This creates Docker containers that contain the Jenkins master and slaves, and a Docker container that contains the Singularity Registry Server. The YAML file allocates ports 8080 and 50000 to be used by the Jenkins master User UI and API respectively. Each Docker container is created using a Dockerfile.

The Jenkins master Dockerfile specifies the plugins to be automatically installed upon creation (such as the Blue Ocean plugin). It automatically sets both username and the password of the Jenkins master UI to "admin" and causes the Jenkins master UI to skip the initial start-up wizard. This is all done to streamline the process of setting up our infrastructure. The slave Dockerfile instructs the newly created slave Docker container to install the Docker CLI and Docker compose upon creation. It then runs a python script that attaches the slave container to the Jenkins master container. This attachment happens via the Java Network Launch Protocol (JNLP).

The Jenkins master UI can then be accessed at localhost:8080 in the VMs web browser. After entering the default credentials (username and password both "admin") the Jenkins master UI is now ready to dispatch Pipeline jobs to slaves without any further configuration needed. The Docker Compose YAML file also defines a TCP/IP network between the Jenkins master and slave containers and the Singularity Registry Server container. These Docker containers can then reference each other using substitute IP addresses.

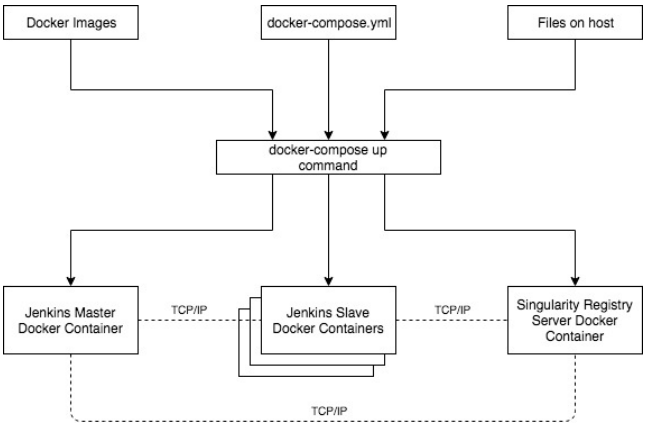

Figure 3: A "docker-compose up" command takes in Docker images, a YAML config file, and the necessary files from the host. It uses these to set up our system inside multiple Docker containers. TCP/IP links are created between the containers to allow for communication.

11 PIPELINE SETUP

A Pipeline can be created by opening the Blue Ocean plugin and clicking "Create Pipeline". In order to link the Jenkins master to the necessary GitHub repository, the user must provide a personal access token. This token is then given to the Jenkins Pipeline, allowing it to access the GitHub repository. Then the time period in between each poll from the Jenkins master to the GitHub repository to check for new commits must be set. Now the Pipeline is fully configured and ready to run according to the Jenkinsfile in the GitHub repository when a new commit is discovered. Additional Pipeline jobs can then be added for multiple users by following these same steps.

12 SYSTEM BEHAVIOUR

The final behaviour of the system we developed can be outlined as follows. Appendix A illustrates the following sequence and shows where in the system each action occurs.

A user creates an account on the Singularity Registry Server and is automatically assigned an authentication token. This token can be viewed on their Singularity Registry Server profile.

The user commits a change to a GitHub repository. This repository contains the Singularity recipe file that specifies how the Singularity container should be built, the code that the user wishes to use on the Singularity container, and a Jenkinsfile containing parameters to be fed to a Groovy script in the Shared Library. Included in these parameters are the users Singularity Registry Server username and authentication token.

At predetermined intervals, the Jenkins master polls the GitHub repository and checks for new commits. When the commit is found, the Jenkins master reads the users Jenkinsfile and passes the parameters of the Jenkinsfile to the Shared Library Groovy script.

The Jenkins master pushes the code, Singularity file and the Groovy script to an available slave and triggers a Pipeline job.

The slave builds a Singularity container from the Singularity recipe file and then pushes the container to Singularity Registry Server via the Singularity Registry Client.

The Jenkins slave then sends the results of the Pipeline job back to the Jenkins master.

Finally, the Jenkins master issues a Singularity Registry Client pull command on the target VM via the Jenkins SSH plugin to pull the container onto the appropriate server.

13 SYSTEM EVALUATION

Here we evaluate our system according to how it meets the requirements specified in Section 7.

13.1 Security

One of our key aims was to make sure that the users of containers cannot escalate to root access from inside a container. We achieved this by making use of Singularity containers. Unlike Docker containers, Singularity containers are not connected to a root owned daemon process. There is therefore no way of the user being able to manipulate the container into granting root access to the host server.

We also needed to make sure that the links between modules in our system were secure. The GitHub Jenkins link was secured by requiring a GitHub personal access token to be passed to the Jenkins master before the master can pull commits from GitHub. The link between the production server and the Jenkins slaves was secured through use of the Secure Shell (SSH) protocol.

Our system needed to be isolated form the host system. This is to solve the problem posed by the fact that Singularity recipe files can run commands on the host system. Without isolation from the host, a malicious or careless user could run unsafe commands such as rm on the host system. We provided isolation from the host by containerizing our entire solution in Docker containers. The Singularity recipe file commands then run on the Docker container rather than the host server, meaning that the hosts filesystem cannot be altered by a user of our system.

13.2 CICD

Through use of Jenkins (and the SSH plugin) we were able to fully automate the CICD pipeline. When a Jenkins poll to GitHub picks up a commit, our Singularity Registry shortly displays that the container has been rebuilt successfully. One improvement that could make our pipeline more *continuous* would be to make commits to version control trigger Jenkins builds immediately rather than after a predetermined polling interval. We were able to achieve immediate commit-to-build behaviour using the Jenkins Freestyle Project job configuration, but the advantages of using the Pipeline configuration outlined in section outweighed the benefits of perfect continuity. The Jenkins SSH plugin allowed us to automate the pulling of containers from our Singularity Registry Server onto the target machine.

13.3 Ease of Setup

The containerization of the master and slaves also meets our requirement of quick and easy setup. The infrastructure can be set up in a single docker-compose up command. Each Pipeline job can be set up by the system administrator in the Blue Ocean interface in less than one minute by simply choosing the appropriate user's version control option of choice, entering an authentication token provided by the user, and selecting the repository containing the container code.

13.4 Portability

We ensured the portability of our infrastructure by containerizing our system in Docker containers. This means that setting up the infrastructure on a different server is as easy as calling a docker-compose up command with the appropriate YAML file as a parameter. After this single command, the system administrator can start creating Pipelines using the Blue Ocean plugin.

When our system is taken down and recreated on another server, all of our Singularity Registry Server information remains as it was on the original server. Our Pipeline jobs, however, do not persist when moved to a different server and need to be recreated by the system admin.

13.5 Usability

Almost the entire system is hidden from the user – whether the user is a system administrator or a user wanting to build Singularity containers. This was achieved through automating many intermediate steps such as creating a Jenkins account and installing software. The use of a Shared Library also meant that the details of the Pipeline jobs are hidden from the user. All a user needs to provide in their Jenkinsfile are the parameters needed by the Groovy script in our Shared Library to build the Pipeline. The code that is stays the same between all Pipelines is abstracted away from the user. When a user or system administrator does have to interact with the system it is via simple CLI commands (e.g. docker-compose up), parameter files such as our simplified Jenkinsfile), or the elegant and intuitive Jenkins Blue Ocean UI.

13.6 Reproducibility

Our system makes use of Singularity containers to encapsulate and share a user's software environment. Singularity containers have been shown by Sochat et. al. to provide effectively perfect replication, with the 0.01% differences being due to trivial information such as different IDs and hostnames. Since we did not modify the Singularity container code we can be confident that our infrastructure provides the same level of replication.

Making our system secure, continuously integrated, delivered and deployed, easy to set up, portable and usable (see sections 11.1-11.5) serves the broader goal of overcoming the barriers to reproducibility outlined in the introduction.

14 CONCLUSIONS

Our system successfully allowed for the building and rebuilding of containers on servers in an HPC system. Through the use of Singularity containers, we were able to ensure that unprivileged users are not able to escalate to root access from inside a container. The entire process from a change to version control through to the deployment of containers on the target server was fully automated in a CICD pipeline. We have therefore achieved our aims stated in section 1.

The system is easy to set up and highly portable. Our Singularity Registry Server persists when moved from one server to another, but the Jenkins Pipeline jobs need to be recreated by the system administrator. This reduces the portability and ease of use of our system. The vast majority of the system is abstracted away from system administrators and the users of containers. The documentation required to use the system is minimal. These factors overcome the barriers to reproducibility mentioned in the introduction. The containers we used also achieve effectively perfect replication. We therefore conclude that our project is able to improve the reproducibility of computational science experiments.

15 FUTURE WORK

While we were able to streamline the setup process substantially through automating much of the Jenkins master-slave setup, more could be done. Perhaps scripts could be written to automate the creation of Pipelines. All parameters could be passed into the CLI and the process of creating a Pipeline – which is usually done through interaction with the Jenkins Blue Ocean UI – would be done automatically in the back ground.

As an additional layer of security, a parser could be developed to scan Singularity recipe files for commands that could pose a threat to the security of the surrounding Docker container or host server.

Currently our Jenkins master polls version control at a predefined interval – every half an hour, say – and triggers a Pipeline job when it finds a new commit. The continuity of our system could be improved by making commits to version control immediately trigger a Pipeline job. Tools that may help achieve this include webhooks, reverse proxies, and post-commit scripts.

We have used GitHub as our version control, but our system could be extended to allow commits from other version control sources such as the Git CLI tool and BitBucket.

While our Singularity Registry Server is able to persist when our system is ported from one server to another, our Pipeline jobs do not. The system administrator needs to rebuild each one again. This repeated effort is clearly undesirable and makes our system both less portable and more tedious to use. This problem could probably be solved by configuring the Docker containers encapsulating the Jenkins master and slave to save their information and reload it when moved to a different server.

REFERENCES

[1] Audris Mockus, Bente Anda and Dag I.K. Sjøberg. 2010. Experiences from replicating a case study to investigate reproducibility of software development. (January 2010). Retrieved from https://pdfs.semanticscholar.org/6983/38951ac6f8a6d6f50d1bbf9c047a7104256a.pdf

[2] Roger D. Peng. 2011. Reproducible Research in Computational Science. In *Science*, 334 (Dec. 2011). DOI: 10.1126/science.1213847

[3] Bente Anda and Dag I.K. Sjøberg. 2009. Variability and Reproducibility in Software Engineering: A Study of Four Companies that Developed the Same System. In *IEEE Transactions on Software Engineering,* 35, 3 (May. 2009), 470-420

[4] Carl Boettiger. 2014. An Introduction to Docker for Reproducible Research, with Examples from the R Environment. arXiv 1410.0846. Retrived from https://arxiv.org/pdf/1410.0846.pdf

[5] Reid Priedhorsky and Tim Randles. 2017. Charliecloud: Unprivileged Containers for User-Defined Software Stacks in HPC. In *Supercomputing 2017*, 3 (Aug. 2017).

[6] Collberg, C. et al. 2014. Measuring Reproducibility in Computer Systems Research.

[7] FitzJohn, R. et al. 2014. Reproducible research is still a challenge. Retrieved from http://ropensci.org/blog/2014/06/09/reproducibility

[8] Adrian Mouat. 2016. Using Docker: Developing and Deploying Software with Containers. O'Reilly Media, Sebastopol, CA

[9] Maxim Belkin, Roland Haas, Galen Wesley Arnold, Hon Wai Leong, Eliu A. Huerta, David Lesny, and Mark Neubauer. 2018. Container solutions for HPC Systems: A Case Study of Using Shifter on Blue Waters. In *PEARC '18: Practice and Experience in Advanced Research Computing, July 22–26, 2018, Pittsburgh, PA, USA*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3219104.3219145

[10] Gregory Kurtzer, Vanessa Sochat, Michael W. Bauer. 2017. Singularity: Scientific containers for mobility of compute. In *PLoS One* (May. 2017). https://doi.org/10.1371/journal.pone.0177459

[11] Mathias Meyer. 2014. Continuous Integration and its Tools. In *IEEE Software*, 31, 3 (May. 2014) 14-16. DOI: 10.1109/MS.2014.58
[12] Zebula Sampedro, Aaron Holt and Thomas Hauser. 2018. Continuous Integration and Delivery for HPC Using Singularity and Jenkins. DOI: 10.1145/3219104.3219147
[13] Reid Priedhorsky and Tim Randles. 2017. Linux Containers for Fun and Profit in HPC. In *;login*, 42, 3 (Sep. 2017). 12-16
[14] Vanessa V. Sochat, Cameron J. Prybol and Gregory M. Kurtzer. 2017. Enhancing reproducibility in scientific computing: Metrics and registry for Singularity containers. In *PLoS One* 12, 11 (Nov. 2017). https://doi.org/10.1371/journal.pone.0188511
[15] Docker Inc. 2019. Docker Documentation. Retrieved from: https://docs.docker.com/
[16] Joana Chavez. 2019. Singularity Container Documentation: Release 2.6. Retrieved from https://sylabs.io/guides/2.6/user-guide.pdf
[17] Mojtaba Shahin, Muhammad Ali Babar, Liming Zhu. 2017. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, 5 (Mar. 2017), 3909 – 3943. DOI: https://doi.org/10.1109/ACCESS.2017.2685629
[18] Kent Beck, James Grenning, Robert C. Martin, Mike Beedle, Jim Highsmith, Steve Mellor, Arie van Bennekum, Andrew Hunt, Ken Schwaber, Alistair Cockburn, Ron Jeffries, Jeff Sutherland, Ward Cunningham, Jon Kern, Dave Thomas, Martin Fowler, Brian Marick. 2001. Manifesto for Agile Software Development. Retrieved from https://agilemanifesto.org/
[19] Donal Spring. 2019. Everything-as-Code. Retrieved from: https://openpracticelibrary.com/practice/everything-as-code/
[20] Karolis Rusenas. 2017. Receive Github Webhooks on Jenkins Without Public IP. (November 2017). Retrieved from https://webhookrelay.com/blog/2017/11/23/github-jenkins-guide/
[21] Melissa Anderson and Kathleen Juell. 2018. How To Configure Jenkins with SSL Using an Nginx Reverse Proxy on Ubuntu 18.04. (July 2018). Retrieved from https://www.digitalocean.com/community/tutorials/how-to-configure-jenkins-with-ssl-using-an-nginx-reverse-proxy-on-ubuntu-18-04

APPENDIX A