Science and Technology Facilities Council

Technical Report
DL-TR-2020-002

Container technologies and their attributes

C Dearden

October 2020

Enquiries concerning this report should be addressed to:

Chadwick Library
STFC Daresbury Laboratory
Sci-Tech Daresbury
Keckwick Lane
Warrington
WA4 4AD

Tel: +44(0)1925 603397
Fax: +44(0)1925 603779
email: librarydl@stfc.ac.uk

Science and Technology Facilities Council reports are available online at:
https://epubs.stfc.ac.uk

ISSN 1362-0207

Container technologies and their attributes

Dr. Chris Dearden
Research Software Engineering Group
STFC Hartree Centre

June 2020

This work made use of computational support by CoSeC, the Computational Science Centre for Research Communities, through its Software Outlook Activity.

Table of Contents

1 Overview

In recent years, the use of container technologies has risen rapidly within the field of software engineering as a means of deploying applications quickly and efficiently across different platforms. Containers offer several advantages including portability, scalability and reproducibility of computational environments. However, to those unfamiliar with the technology the learning curve can be steep. There are now a number of containerisation options available and it is important that the correct choices are made at the outset of a research project to avoid potential lock-in. With this in mind, the aims of this report are to:

- Provide an overview of what containers are and raise awareness of their benefits for managing and deploying research software;
- Provide a review of current containerisation technologies, including their strengths and weaknesses;
- Identify and promote the most appropriate container options for specific situations;
- Increase awareness of container alternatives to those that may already be in use;
- Provide top tips and best practices for working with containers

Prior to providing an assessment of available container technologies, it is prudent to first explain what containers are and why they are useful for deploying application software.

2 What is a container?

A container can be thought of as a unit of software consisting of a single application plus all the dependencies and libraries required for that application to run as a standalone entity. Crucially, containers are packaged in such a way that they are completely isolated from other software and the host operating system within which they run. This means that they are particularly suited to the creation of complex research software environments where portability and reproducibility are important.

When people refer to a container in a computer science context, they are usually referring to a specific instance of a *container image*. A container image is simply a binary file that holds a containerised software bundle. The image is typically built from a text file comprising of a set of instructions that can be thought of as a recipe, defining how the container image is to be assembled and the specifics of the software stack within it. A *container engine,* such as Docker, takes this set of instructions and uses it to build the container image. The engine can also be used to invoke an instance of the image, to run the container on a host system. Multiple containers can be initiated from a single container image. This may be desirable, for example, when multiple instances of a containerised scientific application are required, each with different input parameters.

2.1 Containers and Virtual Machines

The concept of isolating software to run within its own specific environment may sound familiar, particularly to those already accustomed with *virtual machines*. However, there are some crucial differences between containers and virtual machines.

Virtual machines (VMs) are ideal for enabling multiple operating systems (OSs) such as Windows and Linux to run on a single piece of hardware. Virtual machines require something called a *hypervisor* to run. A hypervisor is a software layer that is used to manage VMs on CPUs that support *hardware*

virtualisation. Most mid-range CPUs and above support virtualisation these days; on a personal PC or laptop it is often enabled through the BIOS settings. There are two main types of hypervisor - Type 1, also known as "bare-metal" hypervisors, which are installed directly on top of hardware (e.g. *VMware* ESXi) and Type 2, known as "hosted hypervisors", which run on top of a host operating system. *Oracle Virtual Box* is an example of a Type 2 hypervisor, and versions are available for Windows, Linux and MacOS. Whatever the type of hypervisor used, a virtual machine will have its own guest OS, which 'connects' to the underlying hardware through the hypervisor layer.

Containers on the other hand, do not require a hypervisor to run an application. Instead containers perform the necessary virtualisation at the operating system level. This means that containers share the *kernel*[1] of the host OS in order to communicate with the underlying hardware, removing the need to support an entire guest OS through hardware virtualisation (see Figure). Thus, from the perspective of the host OS, a container is just another application running within it.

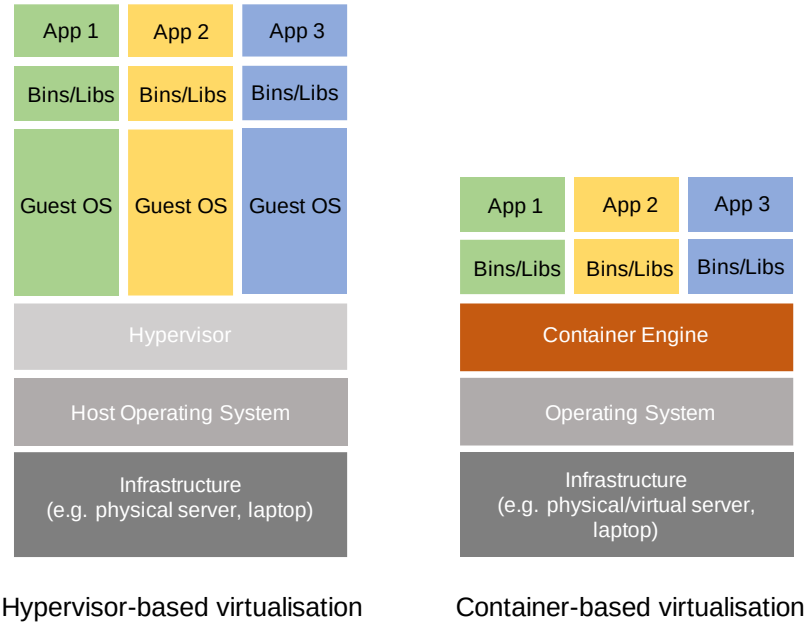


Figure 1 : Schematic showing the difference between Type 2 hypervisor-based virtualisation (left) and container-based virtualisation (right).

2.2 The benefits of using containers

Scalability. When setting up Virtual Machines, it is necessary to reserve a proportion of the system resources (e.g. memory and hard disk space) in order to run a VM with its own guest OS. By removing the need to support a copy of an entire separate operating system, containers have vastly reduced overheads compared to VMs. Consequently, containers have more headroom to allow them to handle additional load when necessary, or conserve resources where appropriate. This also means that it is much more feasible to have multiple container instances running on a single server than it is for VMs.

[1] The kernel is an important part of an operating system that acts as a communication bridge between the software and the hardware infrastructure upon which it is running

Portability. Containers make it easy to deploy applications across different systems by separating code from the underlying infrastructure. By packaging an application into a standalone container image, it is possible to run that container on any system that has an appropriate container engine installed. The lightweight nature of containers makes them more readily portable than VMs.

Reproducibility. Container technologies facilitate reproducible computational research by making it possible to script and share complete research environments as a single package.

Q: I make my open-source code available to the wider community via a centralised hosting service (e.g. GitHub). Isn't this enough? Why do I need to use containers as well?

Version control is certainly an essential requirement for traceability, however on its own this does not necessarily ensure reproducibility. For instance, if the supporting software environment changes (e.g. due to library version updates, security policies, storage), then an application may not produce consistent behaviour. This issue can be solved using containers; indeed, the deployment of version-controlled software inside a container is a powerful combination. By including a container build recipe as part of a code repository, the building of the source code can be automated using specific tagged versions of software libraries and dependencies. Not only does this make deployment of an application much quicker and easier, it also ensures that collaborators are able to replicate the *exact* computational environment. This makes it so much easier for other members of the scientific community to replicate and verify scientific conclusions. This can be particularly useful when publishing a paper, where many scientific journals are now encouraging authors to make their open source code and data available as supplementary material alongside their manuscripts.

3 Docker

Since its emergence in 2013, Docker has grown to become the most widely used container technology. For many, Docker is the entry point into the world of containers; indeed, the two have become somewhat synonymous with each other over recent years. Docker was by no means the first container engine to be developed; however, it quickly gained traction over its predecessors by offering its own ecosystem for container management, making it safer and easier to use. As its popularity grew, Docker cemented its dominance of the container market by taking the lead on standardising the container format, becoming open source in 2015. Because of its popularity there is an extensive Docker user community, and lots of pre-built Docker images are available from on-line container registries that can be used as the starting point for building custom images.

3.1 Docker basics

The Docker engine was originally developed for Linux OSs, but versions for Windows and MacOS are also available via 'Docker Desktop', available for download from https://www.docker.com. The docker engine is a client-server application, consisting of three main components:

i) *The Docker daemon* - a service which runs in the background of the host operating system, and is responsible for creating and managing Docker containers and images;
ii) *The Docker CLI* (command line interface) – a collection of command line tools that are used to interact with the daemon and control the tasks it performs
iii) *The Docker API*, which acts as the bridge between the CLI and the daemon.

Traditionally, the docker daemon requires system administrator privileges to run, which has made it ideal for use on personal PCs or within cloud-based VMs, but not on shared machines where users may not have root access permissions.

3.2 The Docker ecosystem

Figure presents a schematic showing the full Docker ecosystem. On the client side is the Docker CLI, where commands are issued with the prefix `docker`. These commands are managed and processed by the Docker daemon running on a host server. The CLI does not need to be running on the same box as the host server; it is possible to connect a client running the Docker CLI to a remote host for instance.

When containerising an application, it is common practice (and strongly recommended) to start by building on top of an existing container image. This is because Docker images are immutable - once built they cannot be modified at runtime. Thus, new container images must be constructed by adding new 'layers' on top of pre-existing images. This is where container registries become very useful. Registries are essentially cloud-based libraries of container images. The Docker daemon can access external container registries to download or 'pull' Docker images to the host server.

3.3 DockerHub

The default Docker registry is *DockerHub*. There are thousands of pre-built images on *DockerHub*, that can either be re-used as they are, or used as 'base layers' for building custom container images. Before attempting to containerise an application from scratch, it is always worth checking first on *DockerHub* to make sure it hasn't already been done by someone else and the image made publicly available.

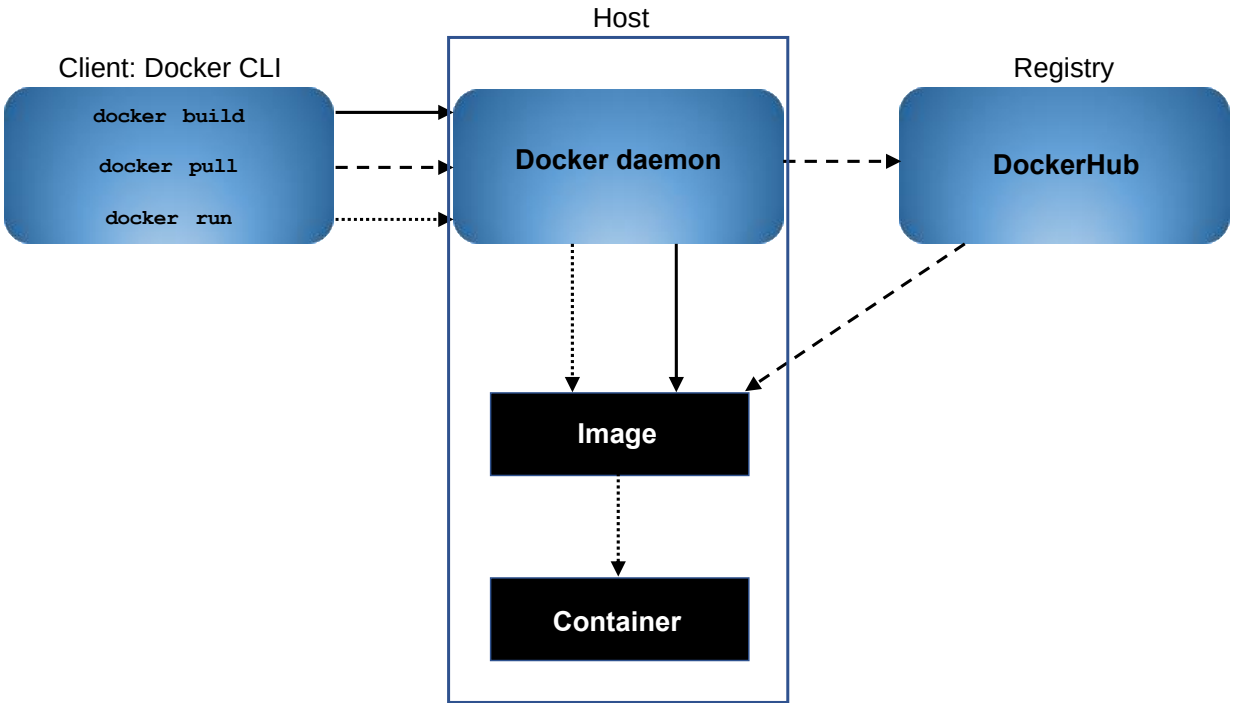


Figure 2 : Diagram showing the Docker ecosystem, with the client on the left running the Docker CLI, the host server in the centre running the docker daemon, and an external container registry on the right, accessed via the host server.

3.4 Building Docker images

To facilitate the creation of new images, Docker uses something called a *Dockerfile*, essentially a text file comprising of a set of instructions used to script and automate the image build process. Appendix A contains a walkthrough example showing the steps involved in building and running a simple Docker image from the command line using a basic Dockerfile.

DockerHub can be configured to automate the building of images from source code held within an external code repository. In this scenario, the Dockerfile along with the application source code would be hosted on a remote service such as GitHub, and each a time a change is pushed to the source code repository, the Dockerfile would be used to re-build the Docker image and push it to DockerHub automatically (subject to certain user-defined test criteria being met during the build). For more details on setting up automated builds with Docker, see https://docs.docker.com/docker-hub/builds/

3.5 Disadvantages of Docker

Docker has grown in complexity over the years, with subsequent versions offering new and additional functionality, but the trade-off is that this can seem daunting and off-putting to new users. Perhaps the biggest issue with Docker is that because the daemon traditionally needs to run as a root process[2], it is not suitable for use on HPC systems or other shared environments. Root escalation problems also mean that it is possible for a user to gain root access to the host server, which is an obvious security issue.

4 Containers for HPC: Singularity

Singularity is an alternative open-source container engine developed by Sylabs, designed specifically to run complex applications on HPC systems. By design, Singularity does not make use of a persistent control daemon, and it allows users without system administrator privileges to run containers. These features make it ideal for use on HPC clusters, overcoming a long-standing issue inherent with Docker.

The fact that Singularity runs as a user process without a daemon means there are fewer security issues relative to Docker, although it is still possible to be root inside a container if required. Singularity has its own container format, SIF (Singularity Image Format) but it can also read and import Docker images. It is also possible to convert Docker images to the native SIF format if necessary.

Although Singularity permits the running of containers at the user level, root access is still required to build Singularity containers. In practice, this means building images on a personal PC or laptop, then copying the image across to the target HPC system and running from there.

[2] Note that v19.03 of the Docker Engine introduced the ability to run the docker daemon 'rootless', i.e. as a user without admin privileges. However at the time of writing, this is still considered to be at an experimental stage.

Singularity is designed for installation on Linux systems. Singularity cannot run on Windows and MacOS natively, but installation is possible using virtual machines. The Sylabs website provides full installation instructions.

Despite the differences in the way Singularity runs compared to Docker, there remain some similarities between the two. For instance, like Docker, Singularity also has its own default registry (called *SingularityHub*). It also has its own text file format analogous to the Dockerfile for scripting and building images, called the Singularity definition file (often given the file name *Singularity*).

Appendix B contains an example of a Singularity definition file, along with some basic commands for building and running Singularity containers.

4.1 Containerised MPI with Singularity

Typically, large scientific applications are often designed to run in parallel across multiple compute nodes using the distributed memory model. Thankfully, Singularity supports the running of distributed memory applications using the Message Passing Interface (MPI) standard, and is compatible with both OpenMPI and MPICH open-source implementations of MPI.

The most common method of executing an MPI application within a Singularity container is to rely on the MPI implementation available on the host HPC system. In this approach, known as the *Hybrid model*, it is necessary to install a version of MPI within the container that is compatible with the version of MPI available on the host system. The MPI launcher (e.g. *mpiexec* or *mpirun*) is then called on the singularity command itself, such that the MPI process running on the host system can work in conjunction with the MPI within the container. For example:

```
  mpirun -n <NUMBER_OF_RANKS> singularity exec <PATH/TO/IMAGE> \
</PATH/TO/BINARY/WITHIN/CONTAINER>
```

The Sylabs website provides examples of Singularity definition files that can be used as templates to build custom MPI containers, using both MPICH and OpenMPI libraries. These examples are also replicated in Appendix C for posterity.

It must be noted that the hybrid method does present a barrier to application portability, since the containerised MPI version must be matched to that of the target system in question for it to work. However, because the MPI launcher is initiated from the host side, it does allow for integration with batch scheduling systems such as slurm and PBS.

5 Other container solutions

5.1 Rkt

Pronounced ‘rocket’, Rkt was created in 2014 by CoreOS as a direct open source alternative to Docker. One of the main aims of Rkt was to offer improved security features over its more established rival, particularly concerning the issue of root escalation privileges. Rkt uses its own native container image format, but is also compatible with Docker images, and does not rely on a daemon to manage containers. Versions of Rkt have been released for all common linux

distributions, including Fedora, Ubuntu, CentOS and Debian; installation on a Mac or Windows machine is only possible via a VM.

However, following the acquisition of CoreOS by Red Hat in 2018, the future of Rkt has been cast into doubt. Indeed, the Rkt github repository has been placed into an archived state, with all development and maintenance marked as halted.

5.2 LXC

LXC is another alternative to Docker, providing a set of low-level tools for the creation and management of full system containers. It is a product of the umbrella project linuxcontainers.org, whose goal is to provide an isolated environment as close as possible to a standard Linux installation but without the overhead of running a separate kernel. Like Rkt, there is no central daemon involved in LXC. Another distinction from Docker is that it is possible to run more than one process inside an LXC container. However, Docker containers tend to be more readily portable, due to the stronger abstraction of applications from the underlying host.

LXC is included in most Linux distributions, but there is no native implementation available for other operating systems.

5.3 LXD

Also forming part of the linuxcontainers.org project, LXD builds on LXC by extending its functionality to provide a new improved system for the management of full system containers. LXD provides a daemon process running on the host side, and uses LXC as the command line interface on the client side. The LXD daemon requires a linux kernel, but the LXC client can be configured for use on Windows and MacOS, such that the daemon can be accessed and controlled remotely. Since the focus of LXD is to provide full system containers, it is less suited to the deployment of single software applications.

5.4 Windows containers

In 2016 Microsoft introduced native support for building and running Windows containers within Windows Server edition, providing users with the ability to create containerised guest environments based on different flavours of the Windows kernel. Windows containers can be managed via Docker, and are offered in two types that differ only in terms of the degree of isolation they provide from the host system. The first type is the standard Windows Server container, which are similar to linux containers in the sense that they share the kernel of the host OS. The second type is the Windows Hyper-V container, which carry their own kernel and are more akin to a Virtual Machine in this regard. Whilst this has some benefits in terms of increased security, the trade-off is that Hyper-V containers have a slightly higher computational overhead when deployed.

The Microsoft website contains full details of how to set up and run Windows containers in both Windows Server Edition and Windows 10, and also includes a list of Windows base container images that are currently available.

5.5 Podman

Developed by RedHat, Podman is a tool for running linux containers and is designed primarily as a drop-in replacement for the Docker CLI. Indeed, Podman offers all the same sub-commands for

building and running containers that Docker does, but because it does not use the client-server model, no daemon is required. Another benefit of Podman over Docker is that it does not require root access to build or run containers. These features of Podman make it a contender for use on HPC systems, and a rival to Singularity in this regard. However, it currently lacks support for parallel file systems, which is a limiting factor. Further information on how to get started with Podman is available at https://podman.io/getting-started/

6 Container management and orchestration

Distinct from container technologies themselves, container orchestration tools exist for the purpose of managing, controlling and scheduling the execution of one or more containers, either on a single host server or across multiple hosts.

For scientific applications, which often run in non-production single user environments, the ability to orchestrate multiple containers on a single host server can be very useful. For example, in the case of application workflows that involve a number of steps (e.g. a numerical simulation that requires both pre and post processing of data), it is preferable to containerise each component separately and use a container management tool to launch them via a single command. This approach retains the ability to run a specific component by itself if so desired, instead of the whole workflow each time. Tools such as docker compose and Singularity compose are ideal for this purpose.

Container orchestration really becomes invaluable in the context of live multi-tenancy production environments, where multiple containers are typically managed across multiple host servers. In such environments, there needs to be an orchestration system in place to manage and control the containers to keep the service running as intended. For instance, should a host server fail, the orchestration tool would detect the point of failure and automatically restart the container on a different host in its place, such that there is no break in service. Some of the most popular multi-host orchestration tools are now introduced.

6.1 Kubernetes

Originally developed by Google and made open source in 2014, Kubernetes has become the market leader in container orchestration. Now maintained by the Cloud Native Computing Foundation, Kubernetes is designed to work with container engines such as Docker to automate the deployment, scaling and scheduling of application containers within a production environment.

In a Kubernetes ecosystem, a production environment would typically consist of a cluster of different host servers (which could be either physical servers or virtual), connected over a shared network. One of the servers in the cluster typically acts as the main server, providing the entry point to the cluster. The remaining servers in the cluster act as nodes[3], which receive and run specific tasks as dictated by the main server, using containers. This means that each node needs to have its own container engine installed within it, allowing nodes to create or tear down containers as necessary.

[3] Here, we refer to a node specifically in the context of a Kubernetes cluster, distinct from nodes within an HPC cluster that use MPI and interconnects.

6.2 Docker Swarm

Swarm is Docker's own rival to Kubernetes. It is similar to Docker compose, but the key difference is that it can scale containers to run across multiple hosts. Docker Swarm is easier to install and setup compared to Kubernetes, but ultimately it is not quite as powerful, and is best suited to the management of smaller clusters running relatively simple applications. Kubernetes is more adept as dealing with larger clusters running more complex applications.

6.3 OpenShift

OpenShift is Red Hat's container management product based on Kubernetes. It is available in different variants, one of which is a paid subscription that includes dedicated user support. Whereas the open source Kubernetes software can be installed on most Linux distributions, OpenShift is available only on Red Hat Enterprise Linux, although it does provide additional functionality that standard Kubernetes does not, such as integration with the Jenkins continuous integration framework, and a useful web interface that makes it easier for beginners to get to grips with.

7 Recommendations

In any given situation, the choice of container technology will largely depend on the specifics of the use case. Broadly speaking, for those new to containers and interested primarily in isolating individual applications, then Docker is probably the best place to start. This is due to its availability on multiple operating systems and its popularity, ensuring a wide support base and a huge collection of pre-existing images to work with.

For those interested specifically in deploying applications to HPC systems, Singularity is the recommended choice, although other options such as Podman are also viable contenders.

If the aim is to containerise the functionality of a whole Linux system, as opposed to an individual application, the LXC/LXD projects from linuxcontainers.org are a good option. Windows containers, in conjunction with Docker, are recommended for containerising applications that require a Windows-based kernel.

8 Links for further reading

A practical introduction to container terminology:
https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction/

Docker tutorial: https://www.docker.com/101-tutorial

Best practice guide for writing Dockerfiles:
https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

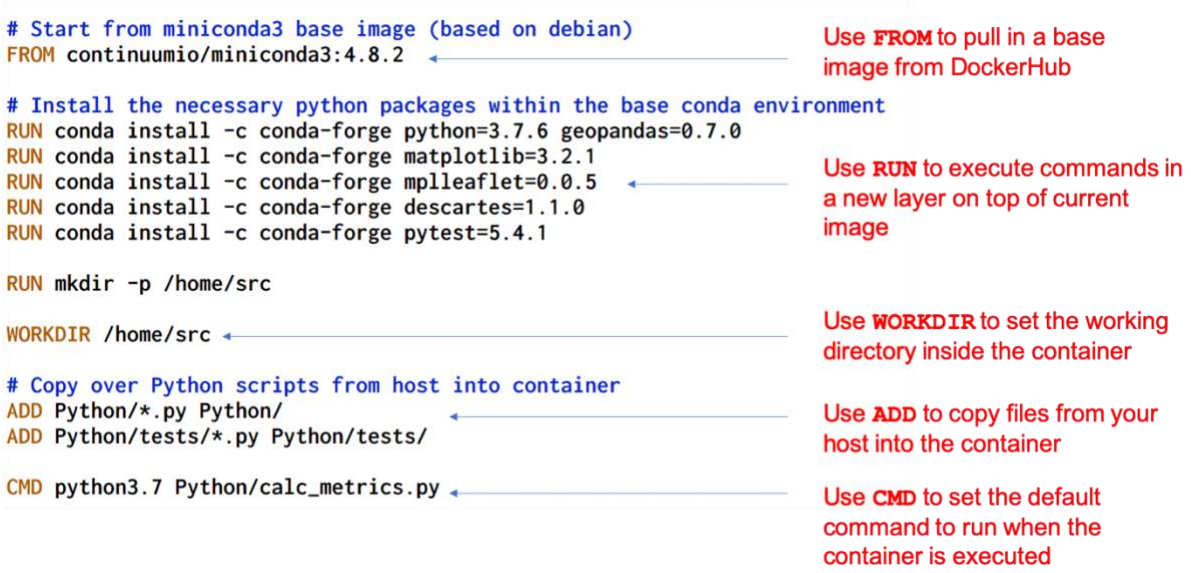
Journal article: "Ten Simple Rules for Writing DockerFiles for Reproducible Data Science"
https://osf.io/fsd7t/

Nvidia toolkit for containerising applications that run on GPUs:
https://github.com/NVIDIA/nvidia-docker

9 Appendix A: Building and running a simple Docker container image

9.1 Building a Docker image

An annotated example of a simple Dockerfile is given below in Figure 9-A. It highlights the most commonly used instructions for building Docker images.




```
# Start from miniconda3 base image (based on debian)
FROM continuumio/miniconda3:4.8.2

# Install the necessary python packages within the base conda environment
RUN conda install -c conda-forge python=3.7.6 geopandas=0.7.0
RUN conda install -c conda-forge matplotlib=3.2.1
RUN conda install -c conda-forge mplleaflet=0.0.5
RUN conda install -c conda-forge descartes=1.1.0
RUN conda install -c conda-forge pytest=5.4.1

RUN mkdir -p /home/src

WORKDIR /home/src

# Copy over Python scripts from host into container
ADD Python/*.py Python/
ADD Python/tests/*.py Python/tests/

CMD python3.7 Python/calc_metrics.py
```

Figure 9-A : A simple Dockerfile example, starting from a miniconda3 base image. As part of the custom build recipe, conda commands are used to install the necessary python packages within the container.

Other instructions are also available for use within the Dockerfile besides those used in Figure 9-A, for example to specify environment variables within the container. A comprehensive reference is available at https://docs.docker.com/engine/reference/builder/

An image is built from a Dockerfile using the following command (assuming super user privileges):

```
docker build -t <username/image_name:tag_name> .
```

The `-t` option allows images to be tagged with a user-defined alias for ease of identification. The `username/image_name:tag_name` format is not compulsory, but is a widely adopted convention. The '.' argument at the end tells docker to perform the build using the Dockerfile in the current directory.

The command `docker images` will then display a list of images currently built with their corresponding tag names.

9.2 Running a Docker image

In the simplest use case, the `docker run` command followed by the image name will execute the container and perform whatever default command was specified by the `CMD` instruction within the Dockerfile:

```
docker run <username/image_name:tag_name>
```

In the case of an image built from the Dockerfile in Figure 9-A, this would invoke the `calc_metrics.py` script within the `Python` directory of the container.

To override the default and perform some other task instead, simply append the above with the desired command to execute in its place, e.g.

```
docker run <username/image_name:tag_name> echo "hello world"
```

To start an interactive session within a container, use the `-it` option:

```
docker run -it <username/image_name:tag_name> /bin/bash
```

Here, the addition of `/bin/bash` at the end tells docker to initiate the bash shell inside the interactive container session.

It is also possible to run an application with a graphical user interface (GUI) inside a docker container. To enable this, the GUI running inside the container needs to be able to communicate with an X Window server running on the host machine. Linux distributions already come equipped with an X server; MacOS users can install XQuartz, and Windows users can download VcXsrv.

Once an X server is running on the host machine, all that remains is to map the DISPLAY environment variable inside the container, which can be done when the container is launched. For MacOS and Windows hosts, this is done as follows:

```
docker run -it -e DISPLAY=host.docker.internal:0 <username/image_name:tag_name>
```

On a Linux host, the DISPLAY environment variable is specified slightly differently:

```
docker run -it –net=host -e DISPLAY=:0 <username/image_name:tag_name>
```

9.3 Getting data into and out of a Docker container

In our current example, any data files produced by an application running inside a container will only persist for as long as the container is running. Once the container has finished its task, any files it

happens to have created will be lost. Further, scientific applications often require some input files, e.g. to specify initial conditions to perform a numerical simulation, but unless these files are somehow available during runtime the container will not be able to access them. One way to overcome this would be to add a new `RUN` instruction to the Dockerfile in Figure 9-A to ensure the required input files are copied into the container image at the build stage. However, this is often not desirable or practical, particularly if the input data files are very large in size.

Thankfully, Docker provides two options for getting data into and out of containers. The first and arguably simplest option is to use *bind mounts.* This allows a directory on the host system to be mounted inside the container during runtime as a storage volume, to which the container has read-write access.

Let's suppose that the Python script `calc_metrics.py` in our example requires two command line arguments, the first argument being the path to an input file containing some model data, and the second argument the name of the output file produced by the script. The script would be invoked as follows:

```
python3.7 Python/calc_metrics.py <input_file> <output_file>
```

Let's assume that an input file, `temp1.nc`, is available on the host system within `/home/model_data`. The example below runs the same image as before, but this time mapping this host folder to a volume called `/data` inside the container, using the `-v` option. When the python script is executed, the input file is visible within the container at `/data/temp1.nc`, and the output file is written to the same `/data` volume, which persists within the host folder `/home/model_data` after the container finishes running.

```
  docker run -v /home/model_data:/data username/image_name:tag_name \
python3.7 Python/calc_metrics.py /data/temp1.nc /data/val_out.nc
```

The second option is to use *volumes*. Volumes are created and managed by Docker, such that they are isolated from the core functionality of the host machine, acting like a disk partition available for the container. This makes them a safer, more secure option than bind mounts, which have the unwanted side effect of being able to inadvertently modify or delete files on the host system. For more information on the use of bind mounts and volumes within Docker, see https://docs.docker.com/storage/

10 Appendix B: Building and running a simple Singularity container image

Figure 10-A shows an example Singularity definition file. It performs the same tasks as the Dockerfile in Figure 9-A, to allow for a direct comparison of the two file formats.

```
# Start from miniconda3 base image (based on debian)
Bootstrap: docker
From: continuumio/miniconda3:4.8.2

%post
# Install the necessary python packages within the base conda environment
conda install -c conda-forge python=3.7.6 geopandas=0.7.0
conda install -c conda-forge matplotlib=3.2.1
conda install -c conda-forge mplleaflet=0.0.5
conda install -c conda-forge descartes=1.1.0
conda install -c conda-forge pytest=5.4.1

mkdir -p /home/src

cd /home/src


%files
# Copy over Python scripts from host into container

Python/*.py Python/
Python/tests/*.py Python/tests/

%runscript
cd /home/src
exec /bin/bash python3.7 Python/calc_metrics.py "$@"
```

Figure 10-A : Example of a Singularity definition file, equivalent to the Dockerfile shown in Figure 9-A

The first part of the Singularity definition file is the header, where the `Bootstrap:` command instructs Singularity to look for images from DockerHub. The `From:` instruction performs the same function as in the Dockerfile.

The rest of the Singularity definition file is composed of sections, each denoted by the `%` prefix. The `%post` section is used to customise a container during the build process, for example to install software and libraries inside the container, to create directories and set environment variables during build time. As such it is analogous to the RUN instruction in the Dockerfile. The `%files` section is used to copy files from the host into the container. The contents of the `%runscript` section are executed when the container image is run, much like the CMD instruction in the Dockerfile. However, when the Singularity container is invoked from the command line, any arguments that are specified after the container image name are passed to the container. More details on the different sections available within the definition file can be found at https://sylabs.io/guides/3.5/user-guide/definition_files.html

Assuming that the definition file in Figure 10-A is saved as a file called `Singularity`, it can be used to build an image as follows (remembering that root permissions are required for building Singularity containers):

```
singularity build image-name.simg Singularity
```

This will generate an image file called `image-name.simg`, which can be transferred over to a HPC system that has Singularity installed, and can be run without root privileges.

To run the container image, and execute the default task specified within the `%runscript` section of the definition file, use the `run` sub-command:

```
singularity run image-name.simg
```

Or, to override the default behaviour and execute a different script inside the image, use the `exec` sub-command as follows:

```
singularity exec image-name.simg /path/to/my_script.sh
```

To start an interactive session within the container:

```
singularity shell image-name.simg
```

11 Appendix C: Containerising MPI applications with Singularity

Below is a template Singularity recipe for containerising a simple MPI application written in C using the MPICH library. In the example, the application source code `mpitest.c` is copied from the host to the `/opt` directory within the container. Version 3.3 of the MPICH library is then installed inside the container, and used to compile the MPI application. This example is taken from the Sylabs.io website at https://sylabs.io/guides/3.5/user-guide/mpi.html

```
Bootstrap: docker
From: ubuntu:latest

%files
    mpitest.c /opt

%environment
    export MPICH_DIR=/opt/mpich-3.3
    export SINGULARITY_MPICH_DIR=$MPICH_DIR
    export SINGULARITYENV_APPEND_PATH=$MPICH_DIR/bin
    export SINGULAIRTYENV_APPEND_LD_LIBRARY_PATH=$MPICH_DIR/lib

%post
    echo “Installing required packages…”
    apt-get update && apt-get install -y wget git bash gcc gfortran g++ make

    # Information about the version of MPICH to use
    export MPICH_VERSION=3.3
    export MPICH_URL=”http://www.mpich.org/static/downloads/$MPICH_VERSION/mpich-
$MPICH_VERSION.tar.gz”
    export MPICH_DIR=/opt/mpich

    echo “Installing MPICH…”
    mkdir -p /tmp/mpich
    mkdir -p /opt
    # Download
    cd /tmp/mpich && wget -O mpich-$MPICH_VERSION.tar.gz $MPICH_URL && tar xzf mpich-
$MPICH_VERSION.tar.gz
    # Compile and install
    cd /tmp/mpich/mpich-$MPICH_VERSION && ./configure –prefix=$MPICH_DIR && make install
    # Set env variables so we can compile our application
```

```
    export PATH=$MPICH_DIR/bin:$PATH
    export LD_LIBRARY_PATH=$MPICH_DIR/lib:$LD_LIBRARY_PATH
    export MANPATH=$MPICH_DIR/share/man:$MANPATH

    echo “Compiling the MPI application…”
    cd /opt && mpicc -o mpitest mpitest.c
```

An equivalent Singularity recipe using the OpenMPI library instead of MPICH is given below.

```
Bootstrap: docker
From: ubuntu:latest

%files
    mpitest.c /opt

%environment
    export OMPI_DIR=/opt/ompi
    export SINGULARITY_OMPI_DIR=$OMPI_DIR
    export SINGULARITYENV_APPEND_PATH=$OMPI_DIR/bin
    export SINGULAIRTYENV_APPEND_LD_LIBRARY_PATH=$OMPI_DIR/lib

%post
    echo "Installing required packages..."
    apt-get update && apt-get install -y wget git bash gcc gfortran g++ make file

    echo "Installing Open MPI"
    export OMPI_DIR=/opt/ompi
    export OMPI_VERSION=4.0.1
    export OMPI_URL="https://download.open-mpi.org/release/open-mpi/v4.0/openmpi-$OMPI_VERSION.tar.bz2"
    mkdir -p /tmp/ompi
    mkdir -p /opt
    # Download
    cd /tmp/ompi && wget -O openmpi-$OMPI_VERSION.tar.bz2 $OMPI_URL && tar -xjf openmpi-
$OMPI_VERSION.tar.bz2
    # Compile and install
    cd /tmp/ompi/openmpi-$OMPI_VERSION && ./configure --prefix=$OMPI_DIR && make install
    # Set env variables so we can compile our application
    export PATH=$OMPI_DIR/bin:$PATH
    export LD_LIBRARY_PATH=$OMPI_DIR/lib:$LD_LIBRARY_PATH
```

```
export MANPATH=$OMPI_DIR/share/man:$MANPATH


echo "Compiling the MPI application..."
cd /opt && mpicc -o mpitest mpitest.c
```