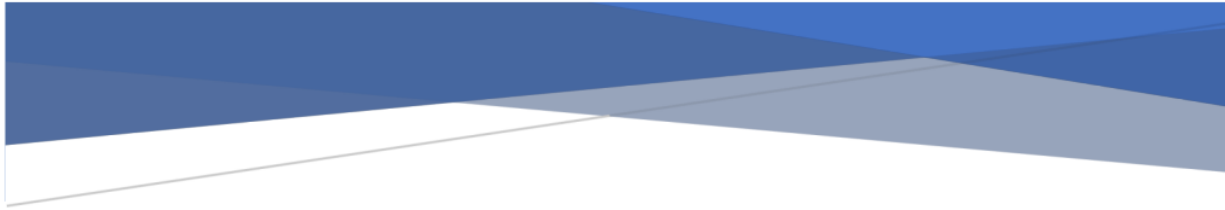


Cloud Engineering Report

by Jonathan Samuel Davies

FILE	4079_JONATHAN_SAMUEL_DAVIES_CLOUD_ENGINEERING_REPORT_34387 1_748414026.DOCX (1.06M)		
TIME SUBMITTED	08-MAY-2021 02:24PM (UTC+0100)	WORD COUNT	4573
SUBMISSION ID	152279303	CHARACTER COUNT	23584



Cloud Engineering: Principles & Practice

Name: Jonathan S. Davies

Student Number: 1707239

Lecturer: Mike Dacey & Tim Bashford

Module Code: ACCA7027

Word Count: 3570

Table of Contents

Table of Figures	3
1. Introduction.....	4
2. Containerisation.....	4
2.1. Defining Containerisation	4
2.2. Why is Containerisation Needed?	4
2.3. Containers & Images	4
2.3.1. Image Overview	4
2.3.2. Container Overview	5
2.4. Container Portability	5
2.4.1. Dockerfile Commands	6
2.4.1.1. FROM.....	6
2.4.1.2. MAINTAINER	6
2.4.1.3. COPY	6
2.4.1.4. ADD	7
2.4.1.5. ENV	7
2.4.1.6. CMD	7
2.4.1.7. ENTRYPOINT	7
2.5. Publishing and Distribution of Images	7
2.6. Persistent Data	8
2.6.1 Bind Mount.....	9
2.6.2 Volumes.....	9
2.6.3 TMPFS.....	9
2.6.4 Volumes in Action	9
2.7. Networking Containers	10
2.7.1 Container Network Model (CNM).....	11
2.7.1.1 Sandbox	11
2.7.1.2 Endpoint	11
2.7.1.3 Network	11
2.7.2 Network Drivers	11
2.7.1.1. None.....	11
2.7.1.2. Host	11
2.7.1.3. Bridge	11
2.7.1.4. Overlay	11
2.7.1.5. MACVLAN.....	11
2.7.3 Networks in Action	11
3. Kubernetes	12

3.1. Controllers	12
3.1.1. ReplicaSet	12
3.1.2. Deployments	12
3.1.3. DaemonSet	13
3.1.4. Jobs	13
3.1.5. StatefulSet	13
3.2. Services	14
4. Conclusion	14
5. References	14
6. Appendices	Error! Bookmark not defined.

Table of Figures

Figure 1 - 'Live' Images are Containers [5, pp. 52]	5
Figure 2 - Example of a Dockerfile.....	5
Figure 3 - Building an Image using a Dockerfile	6
Figure 4 - Example of Pushing an Image to Dockerhub	8
Figure 5 - Container Structure [5, pp. 190]	9
Figure 6 - Example of Building a Volume.....	10
Figure 7 - Demonstrating the Persistent Storage of Data from within a Container to a Local Volume.....	10
Figure 8 - A ReplicaSet Managing X Number of Pods [15].....	12
Figure 9 - The Architecture of a Deployment [15]	13

1. Introduction

This report serves as a collation point for research undertaken into containerisation and orchestration software, specifically using Docker and Kubernetes. Section 2 will demonstrate work carried out using different elements of Docker, namely its general functionality and a declaration of why such software is needed. Section 3 will delve into Kubernetes and how it uses Deployments and Services to monitor the healthy and operation of clusters.

2. Containerisation

2.1. Defining Containerisation

Containerisation is a solution to a long-standing problem faced by organisations wishing to deploy services remotely without fear of infrastructural incompatibilities, and generally does so at a far greater pace than that of traditional workflows [1], [2]. This is achieved by encapsulating an application in its entirety (with all dependencies and configurations), into a single instance, which can then be ported anywhere via local network or the internet. Due to the application instance containing all necessary files for its activation, the problem of software running perfectly well on one machine but failing to launch on another is resolved.

2.2. Why is Containerisation Needed?

Prior to the advent of commonplace containerisation tools such as Docker (containerisation as a concept is not new, and its early adoptions appear in 1979 when Unix V7 released the chroot functionality [3]), the mainstay method of virtualisation came from the use of Virtual Machines (VMs). Whilst a VM offered system administrators a way of segmenting servers and running different operating systems in parallel, it did so with a cumbersome frame. Spinning up a VM could take an extremely long time and doing so required the division of hardware resources. To combat this, technologies like Docker removed the need for a hypervisor, allowing the micro-environment to run straight on the hardware of the host machine, therefore boasting load times and shut down times, whilst retaining the isolation factor that was one of the most attractive qualities of a VM [4], [5], [6].

In essence, containerisation provides the means of virtualising an entire operating system so that multiple instances can be concurrently run on a single OS unit. Virtual machines on the other hand choose to virtualise the *hardware* to enable a diversion of resources to multiple OS instances. The portability, speed, and general lightness have become the favoured means of virtualisation for many industries [7].

2.3. Containers & Images

2.3.1. Image Overview

As previously discussed, docker allows applications, dependencies, and configurations to be bundled together so that once deployed, everything needed for a successful launch is present. The binding of these components forms an 'Image', which can be viewed as a blueprint that is then used to assemble containers. The structure of an image represents a bottom-up layering system, usually starting with a foundational image such as an OS construct, with consecutive layers adding various elements needed like libraries and file structuring. Once all layers are in place, the image is then ready to be distributed and is done so as a singular body [8]. Much like the relationship between classes and instantiated objects in Object Oriented Programming, the image provides the procedure of how a container will be built and is never actually considered 'live' itself. This is outlined in Figure 1.

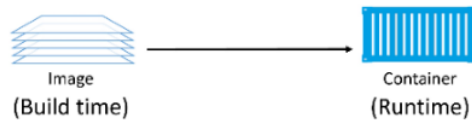


Figure 1 - 'Live' Images are Containers [5, pp. 52]

Due to the need of their existence, images are usually small in size and this is what allows their creation and deletion speeds to be so fast. Among Docker users, there is a popular trend that sees the least number of layers used within a Dockerfile (discussed in 2.4.1), with the reason being that AUFS (Advanced Multi-layered Unification Filesystem) has a maximum limit of 127 layers. This is the reason that multiple commands are usually specified in one RUN instruction [6, pp. 25-26].

2.3.2. Container Overview

Containers exist only during runtime and are the running version of their image-based cousins. They are ephemeral by their very nature due to them being so lightweight and resource friendly.

2.4. Container Portability

The containerisation paradigm has seamlessly integrated into the development structure of countless organisations due to its ability to streamline operations in a multitude of ways. Firstly, it enables the Software Development Team to build the required application focusing solely on the dependencies and code whilst simultaneously granting the Operations team the capability to concentrate on the deployment and supervision aspect without having to consider which software versions need to be shipped with different products.

Secondly, it allows the testing environment used to be an exact replication of the environment used to build the application. Developers now have the ability to containerise their project and publish it to a registry which can later be accessed by the testing team. Even if the hardware infrastructure differs from that of the developers, the application will still execute in the exact same manner.

This is achieved through the use of Images that are created by a Dockerfile. The Dockerfile stores the systematic commands and configuration data needed to build a replica of the desired work environment. At its core, a Dockerfile is a simple text file, whereby keywords are used to specify each required component. Section X below outlines some of the most popular commands used in Dockerfiles and provides a description of their uses.

Figure 2 showcases a Dockerfile that will be used to build an image for running a Python application, whilst Figure 3 demonstrates the actual building of the image using said Dockerfile.

```
FROM python:3.6
LABEL name="Jonathan Davies"
LABEL email="jonathan.davies@uwtsd.ac.uk"
COPY . .
ENTRYPOINT [ "python", "password_generator_cli.py" ]
```

Figure 2 - Example of a Dockerfile

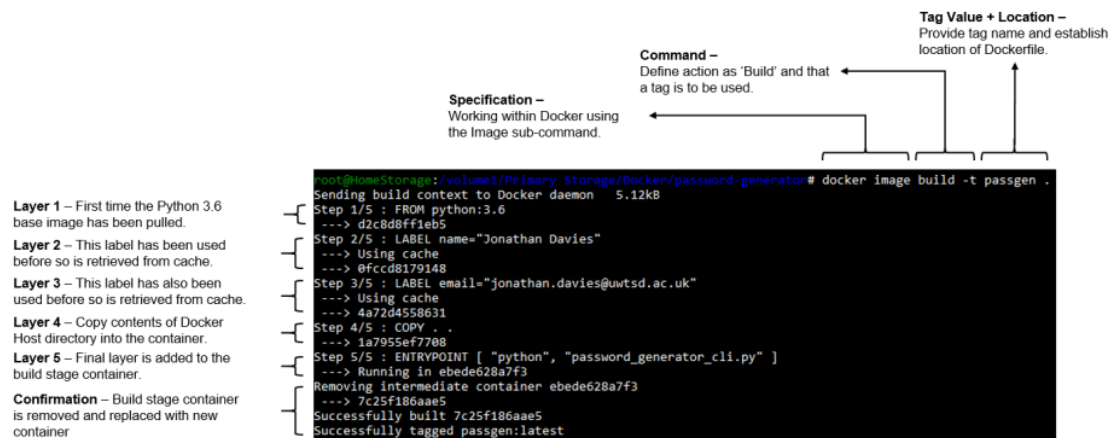


Figure 3 - Building an Image using a Dockerfile

2.4.1. Dockerfile Commands

The following sections are not an exhaustive list of all Dockerfile commands, just some of the most commonly seen. Literature pertaining to the following shall be referenced here [6, pp. 46-48], [7, pp. 55-71].

2.4.1.1. FROM

This instruction must be the first used in the file and it allows for the importing of a base image which is used as a foundation for the subsequent layers. For example, if the Ubuntu image is used in the FROM command, a lightweight version of this OS, complete with its own internal filesystem, is ported into the new image. Tags and Digests can be added to stipulate an exact version of any image that is to be used, if none are present then the latest version is used by default. For the creation of base images, the 'scratch' keyword can be used (e.g., FROM scratch).

- FROM (base image)
- FROM ubuntu
- FROM ubuntu:bionic
- FROM scratch

2.4.1.2. MAINTAINER

When creating an image, it is useful to add the name and contact details of the individual(s) that are responsible for creating and updating it. This data can be added via the 'Maintainer' command.

- MAINTAINER (author name) (<author email>)
- MAINTAINER Jonathan Davies <example.email@uwtsd.ac.uk>

2.4.1.3. COPY

This command allows files and/or directories to be copied from the host machine into the file structure of the image to be created. The Copy keyword is immediately followed by the source component (whether it be a directory or file), and then followed once more by the destination (this is the location within the new image filesystem where this element shall be copied to).

- COPY (*src*) (*dst*)
- COPY *example_script.py* /*example_script.py*

2.4.1.4. ADD

The Add command is the predecessor to Copy and is considered to be the heavier of the two, offering more functionality albeit rarely used in comparison. It carries out the same operations as the COPY command but with the added ability to handle URLs and TAR files. In the majority of cases however the Copy command is preferable, even if files are to be downloaded from URLs (this can be achieved by using the Run command and results in fewer needed layers).

- ADD *https://github.com/User/repo/archive/master.tar.gz* /*remote/*

2.4.1.5. ENV

This instruction allows for the specification of environment variables within the context of the new image. This allows any application to use the set value from anywhere within the container.

- ENV *key value*
- ENV *variable1 4041*

2.4.1.6. CMD

CMD allows for commands and applications to be executed in a similar fashion to the RUN instruction. The key difference however is that CMD will enact its directives upon the instantiation of a container that uses the image, rather than during the building of the image itself, and it is also worth noting that the entire CMD element can be overruled by the *docker run* command.

- CMD *command*
- CMD ["executable", "argument 1", "...", "argument N"]
- CMD ["my_script.py"]

2.4.1.7. ENTRYPOINT

ENTRYPOINT effectively allows the container that will use it to behave like an executable. Any arguments passed to the *docker run* command shall be passed along to the application housed within the ENTRYPOINT instruction. Unlike the CMD instruction, the *docker run* arguments do not override the entry point arguments.

- ENTRYPOINT *command*
- ENTRYPOINT ["executable", "argument 1", "...", "argument N"]
- ENTRYPOINT ["python3"]

2.5. Publishing and Distribution of Images

With one of the overarching pillars of containerisation being the ability to deploy and use them anywhere, the storage and access of images needs to be addressed. This is where the notion of *registries* come in and their use cases is bifold. Firstly, they allow an image to be pushed into a designated repository. This in itself serves as an external backup of image layer data, but also allows teams to collate the building of an application together in one centralised location. Each member of the team can now submit their deliverable and keep them organised in sub-repositories; this can be furthered by the use of *tagging*.

Secondly, persons or teams that wish to use the image/application, need only pull it down from the registry (providing that they have access of course), ensure that docker is installed on the receiving machine, and they are up and running.

Before pushing to Dockerhub, correct name conventions should be applied to the names of images. An image should be built by entering the name of the user followed by the repository it will be stored in, followed up by a specific tag if applicable (by default, users are encouraged to provide tags).

- docker push (username/repository-name:tag)
- docker push jondaviesdev/passwordgenerator:latest
- docker pull (username/repository-name:tag)
- docker pull jondaviesdev/passwordgenerator:latest

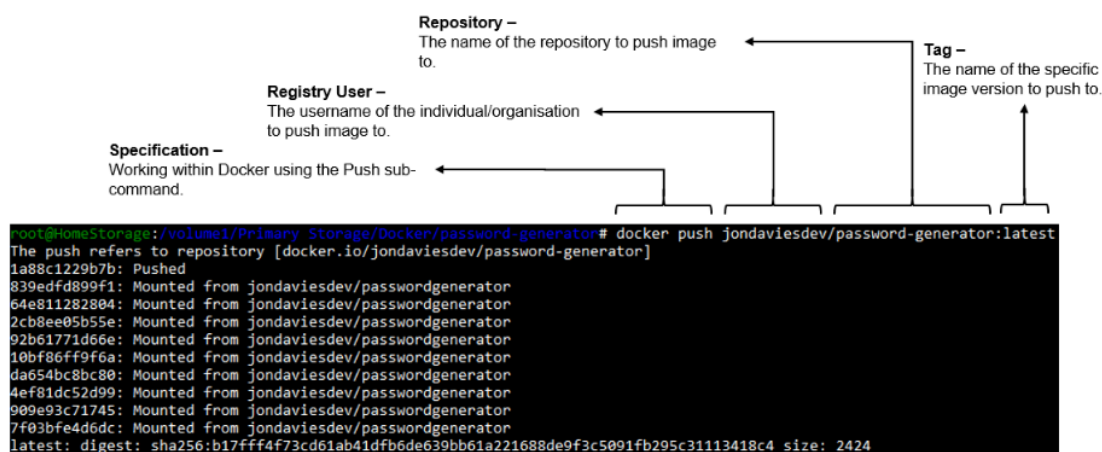


Figure 4 - Example of Pushing an Image to Dockerhub

2.6. Persistent Data

By default, all data created within a container is destroyed upon the death of the container itself. This is as much a security feature as it is a clean-up process. Sometimes however, the information generated by containers will be useful to an external system at a later date. In this case, data must be stored in a way that permits it to endure regardless of the existence of the container that created it, hence persistent data.

Nigel Poulton [5, pp. 189-198] states that containers are designed to be read only and that it best not to brute-force re-configure a container after it is deployed. Some applications however rely on the notion that they can be written to, so docker encapsulates the read-only layers and adds a small read-write layer on top. This layer is called *local storage* and is perfectly acceptable to house data that is not needed beyond the lifecycle of the container. The layering system is depicted in Figure 5 below.

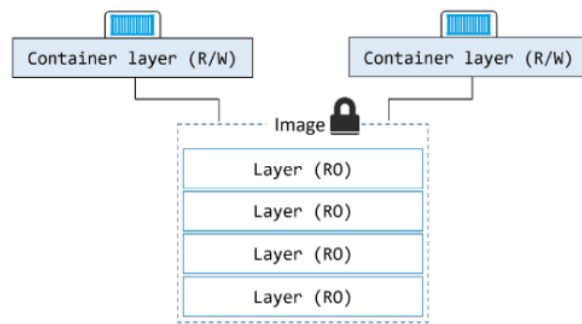


Figure 5 - Container Structure [5, pp. 190]

The *Volumes*' disassociation allows for data to be shared not only with other container on the same host machine, but also across networks, allowing containers that are external in all regards to access the shared data. There are three means of using shared volumes, with each offering different functionality [12].

2.6.1 Bind Mount

Bind Mounts are used when a particular section of the host's filesystem needs to be directly mapped into a container, essentially 'binding' the two trees together. This will allow the bound container to access files and directories on the host machine. This can be incredibly valuable if the application running inside the container requires information stored on the host.

2.6.2 Volumes

Volumes were introduced to serve as container independent storage mounts that allowed for the archiving of data completely separately from the container's Union File System. When created, a new filesystem tree is created on the host machine (this does not have to be the case, as completely external storage facilities can be used as Docker volumes), and all data generated by the container can then be passed to it. This data is saved to disk and can persist past the lifecycle of the container that created it, or a subsequent container that wrote to it.

2.6.3 TMPFS

TMPFS is a structure that when created allows for the storage of temporary files that should not exist after destruction of the container. This sort of data could be of a sensitive nature such as passwords or something more mundane like non-critical log files and buffer data. A key point to note is that all data stored in this device is saved in-memory, not to disk, which adds to its design of being volatile. When a container creates a mount point of this type, it is done so within the domain of the container, usually in */tmp*.

2.6.4 Volumes in Action

This section provides examples of both volumes being created, and data actively being added to them. Figure 6 shows the construction of a Volume, stored locally on the host machine.

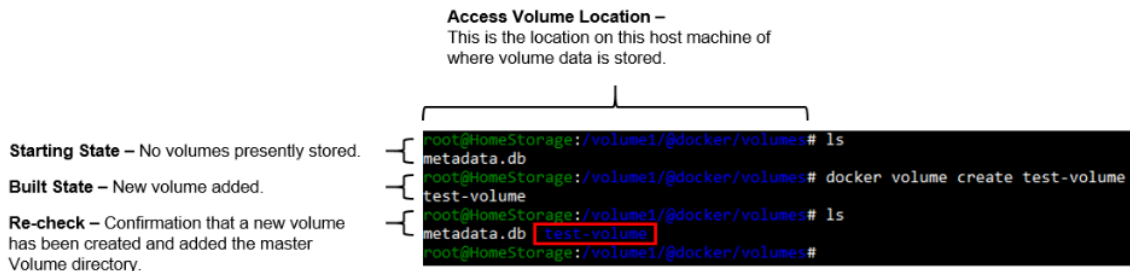


Figure 6 - Example of Building a Volume

Figure 7 takes this a step further and highlights the addition of data to this volume from within a container that has been linked to it.

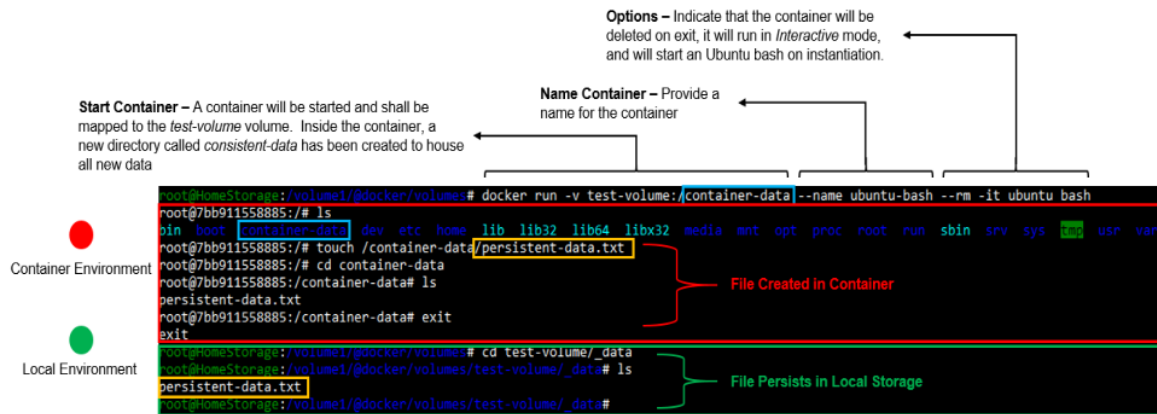


Figure 7 - Demonstrating the Persistent Storage of Data from within a Container to a Local Volume

2.7. Networking Containers

Although containers by themselves provide a powerful tool in regard to creating, testing, and deploying applications, combining containers together via networks can prove to be a much more efficient means of delivering services. A clear example would be a modular, microservice architecture. Each component of the application is split down into smaller sub-services which can be networked together to create the total package. This has many benefits including a simpler development process as different teams can all work on different parts without the need to understand (or sometimes even communicate) with other teams. It also allows for a more streamlined testing process as small, segmented sections of the application can be tested thoroughly instead of picking away at monolithic code structures.

This is all possible due to the ability of networking together many different containers to either each other, the outside world, or most commonly, both. With the notion that containers can be given their own IP and MAC addresses, they need not even be aware that they are connecting to other containers, in their view, they are connecting with just another service.

For every Docker container created, it is packaged with the three core elements of the Container Network Model (CNM), these elements are the Sandbox, Endpoint and Network.

2.7.1 Container Network Model (CNM)

2.7.1.1 Sandbox

The Sandbox is a self-contained network environment that houses routing tables, DNS configurations, ethernet interfaces and ports [5, pp. 152-153]. It provides the container with everything in needs to accept a connection to a network [13] [14].

2.7.1.2 Endpoint

Endpoints can be considered as a virtual NIC providing the same means of attachment to both the host (in this case the container's sandbox) and the network. A sandbox can hold more than one endpoint, granting the ability to connect to multiple networks simultaneously [13] [14].

2.7.1.3 Network

The network can be thought of as a virtual switch. It bunches groups of endpoints together so that they can all communicate on the same network [13], [14].

2.7.2 Network Drivers

2.7.1.1. None

In some instances, it is either crucial or preferable to completely isolate containers in terms of their network connectivity. Such examples include a container that is dealing with sensitive or private information, or a simple text editor that has no need to interact with the outside world. In these circumstances it is advised to restrict access to the network to prevent theft of data [15]. To do this, specifying the *'none'* keyword upon container creation ensures that the subject will have no access to a network, be it local or otherwise.

2.7.1.2. Host

The *'host'* keyword allows the container to be bound to the host, forfeiting its own IP address, and also shares the networking namespace of the host machine.

2.7.1.3. Bridge

If no driver is specified by the user, the *'bridge'* driver is the default setting and closely resembles a physical networking switch [16]. It constructs a virtualised bridge that allows connections between containers that exist on a private subnet, i.e., any containers that are currently running on the same Docker host machine, essentially restricting access to a local scope, unless routing options laid out by the OS dictate otherwise [17], [18].

2.7.1.4. Overlay

The *overlay* driver allows for communication between containers that are located on different host machines, therefore allowing containers that collectively form a microservice to communicate seamlessly from anywhere.

2.7.1.5. MACVLAN

In situations whereby applications or network monitoring software require a MAC address, the *macvlan* command allows for the designation of MAC addresses to containers. This will essentially make the physical network device or application think that there is a physical device connected, and not just a virtualised container.

2.7.3 Networks in Action

Figure 8 shows the creation of a network within Docker, followed by the creation of two separate containers (note the second container is formed in a different SSH instance due to the first container being run in interactive mode). Each container is then instructed to ping the other, resulting in a successful transfer of packets from both sides.

```

root@HomeStorage: /volume1/docker/volume1 # sudo docker network create network-demo
8417c9be999efad235893e90494efe01fe0d15c0ed98174c17fc55aeeb83b4a3
root@HomeStorage: /volume1/docker/volume1 # sudo docker run -it --name=container-one --network=network-demo busybox sh
/ # ping container-one
PING container-one (172.20.0.2): 56 data bytes
64 bytes from 172.20.0.2: seq=0 ttl=64 time=0.041 ms
64 bytes from 172.20.0.2: seq=1 ttl=64 time=0.068 ms
64 bytes from 172.20.0.2: seq=2 ttl=64 time=0.086 ms
64 bytes from 172.20.0.2: seq=3 ttl=64 time=0.109 ms
^C
--- container-one ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.041/0.076/0.109 ms
/ #

root@1d601febb234: /
Jon@HomeStorage: ~$ sudo docker run -it --name=container-two --network=network-demo busybox sh
Password:
/ # ping container-two
PING container-two (172.20.0.3): 56 data bytes
64 bytes from 172.20.0.3: seq=0 ttl=64 time=0.044 ms
64 bytes from 172.20.0.3: seq=1 ttl=64 time=0.112 ms
64 bytes from 172.20.0.3: seq=2 ttl=64 time=0.097 ms
64 bytes from 172.20.0.3: seq=3 ttl=64 time=0.073 ms
^C
--- container-two ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.044/0.081/0.112 ms
/ #

```

Figure 8 - Two Containers Pinging each other on the same Network

3. Kubernetes

3.1. Controllers

3.1.1. ReplicaSet

In Kubernetes, Controllers are used to manage the scale of a cluster of pods to ensure that there is always a match between the current and desired number of pods. Perhaps the most basic of these higher-level controllers is the ReplicaSet (RS). It is responsible for a group of identical containers, that when within the bounds of the RS, allows the containers to network with each other seamlessly whilst also having access to a shared volume. The ReplicaSets are favoured over the manual management of singular pods as they provide a number of benefits to the user, with the four main ones being; redundancy, ease of scaling, self-healing capability (any containers that are corrupted can be instantly killed and replaced), and sharding (multiple ReplicaSets can work together in parallel to solve problems faster) [19], [20].



Figure 9 - A ReplicaSet Managing X Number of Pods [20]

3.1.2. Deployments

A Deployment Controller is used to dispense new versions of an application and typically encapsulates a ReplicaSet (Arundel & Domingus [21] state that a user would only really interact with Deployments, and not so much with a ReplicaSet) building from the functionality that the RS offers, namely with the provision of version rollback and dispatching updates.

The main benefit of using a Deployment is that it offers the ability for multiple replicas of a pod to be made for redundancy.

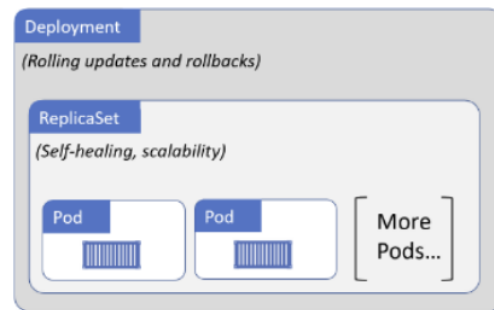


Figure 10 - The Architecture of a Deployment [20]

3.1.3. DaemonSet

In situations whereby a particular pod instance is needed across every node within a specified system, a DaemonSet is the most suitable controller. It is useful for bypassing such issues as excessive network traffic. For example, if 100 pods within a node need to report either monitoring or logging data to an external source, there is a vast amount of traffic being emitted from that node. Another option would be to have the pods send all their data to a designated pod (implanted by the DaemonSet) that could collate and disperse that information in a streamlined way. If not all nodes need such capabilities, the node labelling system allows the DaemonSet Controller to filter out the nodes that do not need its payload and just target the ones that do [22].

3.1.4. Jobs

The Job Object targets tasks that only need to be run once and exit upon successfully reaching the exit point of their program. This differs from a standard pod, which by default will continuously restart upon finishing its tasks, and it is not difficult to imagine why this may not always be favourable.

One of the features of the Job Object is that can be considered an advantage and disadvantage is the automatic re-launching of failed jobs. This can be an advantage if the job has failed due to external factors such as a scheduling hiccup, as when the job is restarted, it is likely to be able to finish successfully. However, if the job fails and restarts due to conflicts that reside within the job itself, this can lead to wasted resources.

3.1.5. StatefulSet

A StatefulSet's use cases centre mainly around the need of persistent data whether it be valuable client data that needs to be retained for future ease of access, or database information. These Controllers are very similar to Deployments in the way that they allow for self-healing, expansion of services and updates but differ in three ways:

- Facilitate the preservation of Pod names.
- Facilitate the preservation of DNS hostnames.
- Facilitate the preservation of volume bindings.

Whilst this does not necessarily make StatefulSets better and Deployments worse, it does allow the correct Controller to be selected for the task at hand.

3.2. Services

Due to the ephemeral nature of Pods within a cluster, it is highly irregular to rely on the IP of a Pod in order to connect to it over a network. If a Pod terminates and another takes its place, it will have a different IP address and therefore be unreachable by the agent that knew the IP of the previous Pod. It is for this reason that Services exist and serve as an intermediary point that sits as a barrier between the external client and the Pods that they are trying to access. By using this architecture, Pods can be added, removed, terminate successfully or unsuccessfully, restored to previous versions, or updated, and the Service will simply keep track of the healthy, available pods in a list and use that to express which Pods can be connected to and which cannot.

4. Conclusion

This report has covered two key areas of research within the topic of containerisation, the first being the general operation of a prominent containerisation tool currently used in multiple industries around the world, Docker. The next area is Kubernetes which serves as an orchestration, or managing, tool to operate clusters of containers.

5. References

- [1] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud Container Technologies: a State-of-the-Art Review," *IEEE Trans. Cloud Comput.*, May 2017.
- [2] S. P. Kane and K. Matthias, "The Docker Landscape," in *Docker Up & Running*, 2nd ed., V. Wilson, Ed. O'Reilly, 2018, pp. 11–14.
- [3] Á. Kovács, "Comparison of different linux containers," in *2017 40th International Conference on Telecommunications and Signal Processing, TSP 2017*, 2017, vol. 2017-January, pp. 47–51.
- [4] N. Aerts, E. Cailliau, L. De Groote, and L. Noterman, "A comparative study on containers and related technologies," 2016.
- [5] S. Singh and N. Singh, "Containers & Docker: Emerging roles & future of Cloud technology," in *Proceedings of the 2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology, iCATccT 2016*, 2017, pp. 804–807.
- [6] A. K. Yadav, M. L. Garg, and Ritika, "Docker containers versus virtual machine-based virtualization," in *Advances in Intelligent Systems and Computing*, 2019, vol. 814, pp. 141–150.
- [7] T. V. Doan *et al.*, "Containers vs virtual machines: Choosing the right virtualization technology for mobile edge cloud," in *IEEE 5G World Forum, 5GWF 2019 - Conference Proceedings*, 2019, pp. 46–52.
- [8] N. Zhao *et al.*, "Large-Scale Analysis of Docker Images and Performance Implications for Container Storage Systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 4, pp. 918–930, Apr. 2021.
- [9] N. Poulton, *Docker Deep Dive*, 2nd ed. Nigel Poulton, 2020.
- [10] A. Mouat, *Using Docker*, 1st ed. O'Reilly, 2016.
- [11] J. S. Chelladurai, V. Singh, and P. Raj, *Learning Docker*, 2nd ed. Packt Publishing, 2017.
- [12] N. G. Bachiega, P. S. L. De Souza, S. M. Bruschi, and S. D. R. S. De Souza, "Performance Evaluation of Container's Shared Volumes," in *Proceedings - 2020 IEEE 13th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2020*, 2020, pp. 114–123.
- [13] H. Zeng, B. Wang, W. Deng, and W. Zhang, "Measurement and evaluation for docker container networking," in *Proceedings - 2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, CyberC 2017*, 2017, vol. 2018-Janua, pp. 105–108.
- [14] G. N. Schenker, *Learn Docker - Fundamentals of Docker 19.X*, 2nd ed. Packt Publishing, 2020.
- [15] J. Nickoloff and S. Kuenzli, "When to Use Closed Containers," in *Docker in Action*, 2nd ed., Manning, 2019, p. 91.
- [16] J. Langemak, "Exploring Bridges," in *Docker Networking Cookbook*, 1st ed., Packt Publishing, 2016, pp. 16–17.
- [17] L. L. Mentz, W. J. Loch, and G. P. Koslovski, "Comparative experimental analysis of Docker container networking drivers," in *Proceedings - 2020 IEEE 9th International Conference on Cloud Networking, CloudNet 2020*, 2020.

- [18] R. Dua, V. Kohli, and S. K. Konduri, "The docker0 bridge," in *Learning Docker Networking*, 1st ed., Packt Publishing, 2016, pp. 3–6.
- [19] B. Burns, J. Beda, and K. Hightower, "ReplicaSets," in *Kubernetes Up & Running*, 2nd ed., O'Reilly, 2019, pp. 103–104.
- [20] N. Poulton, "Self-healing and Scalability," in *The Kubernetes Book*, 2nd ed., LeanPub, 2020, p. 50.
- [21] J. Arundel and J. Domingus, "Supervising and Scheduling," in *Cloud Native DevOps with Kubernetes*, 1st ed., O'Reilly, 2019, p. 54.
- [22] J. Baier, G. Sayfan, and J. White, "Sharing with DaemonSet Pods," in *The Complete Kubernetes Guide*, 1st ed., Packt Publishing, 2019, p. 202.

Cloud Engineering Report

GRADEMARK REPORT

FINAL GRADE

94/100

GENERAL COMMENTS

Instructor

A very good report with a strong implemmentation. Well written, structured and sourced. Really good choice of references including up to date journals.

Excellent work, Jonathan!

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

