

Student name:	Rajesh Barua	Student ID number:	2513042
Programme:	MSc Computing Portfolio		
Module:	Cloud Engineering Principles and Practice		
Module code:	ACCA7027	Contribution to Overall Module Assessment (%):	100%
Lecturer:	Mark Huntly	Internal Verifier:	Nathan Morgan
Assignment Title:	Containerisation and Data Persistence in Cloud Environments: A Practical Evaluation of Docker Volumes for Stateful Workloads	Word count (or equivalent):	5000 equivalent

Table of Contents

Introduction.....	3
1. Docker and Containerisation	4
1.1 From Hardware Virtualisation to Operating System Level Virtualisation	4
1.2 The Architecture of the Docker Platform	5
1.3 Images, Layers and the Dockerfile	6
1.4 Publishing Images to a Registry	7
1.5 Container Networking.....	8
1.6 Container Volumes and the Persistence Problem	11
1.6.1 Named Volumes.....	11
1.6.2 Bind Mounts.....	12
1.6.3 Demonstrating Persistence.....	12
1.7 Critical Evaluation	14
2. Load Balancing, Scalability and Application Gateways	15
2.1 A Cloud Provider's Approach to Load Balancing	15
2.2 Scalability	15
2.3 Application Gateways.....	16
2.4 Why Load Balancing and Scaling Are Complementary.....	17
2.5 The Benefit of Combining All Three.....	17
2.6 Relation to the Deployed System.....	18
3. Driving Forces and the Serverless Alternative.....	19
3.1 The Main Driving Forces.....	19
3.2 Serverless Functions	19
3.3 Comparing Serverless Functions with Containerisation.....	20
3.4 Application to the Deployed Workload	21
4. Conclusion	22
5. References	23

Introduction

In cloud environments, containerisation has gone from an emerging technique to the norm. According to Cloud Native Computing Foundation, 82 per cent of all container users are now running Kubernetes in production, up from 66 per cent in 2023, and 98 per cent of all organisations surveyed have implemented some cloud native practices. This report delves into the technology behind that change and assessment its shortcomings.

The report tackles three concerns. In Section 1, I will learn about Docker and containerisation, and will construct a working MySQL and phpMyAdmin deployment to work with the assignment, learning how to build it as an image, publish it, how to network and volumes it. Google Cloud and Microsoft Azure's load balancing, scalability and application gateways are analysed in section 2 using the example of a music streaming platform of Spotify. In Section 3 I explore the reasons for the growth of cloud computing and contrast cloud functions (serverless) against containerisation.

This argument is followed throughout. Containers are all the time touted as stateless and therefore not appropriate for databases. The exercises in this document seem to indicate that this characterization is mixing up the data lifecycle with the container lifecycle, and that Docker volumes are used specifically to separate the two.

1. Docker and Containerisation

1.1 From Hardware Virtualisation to Operating System Level Virtualisation

The concept of hardware virtualisation is to abstract physical resources in order to run multiple operating systems (guest OS) on a single machine managed by a hypervisor. The solution addressed server consolidation, but at an incurred (structural) cost: For each guest there is a full operating system image, its own kernel, and its own boot sequence, and a host with 10 guests will have 10 kernels to serve 10 applications.

On the operating system level virtualisation eliminates this duplication. Workloads are isolated at the process level instead in containers, which share the host kernel. Two long standing features of the Linux kernel are responsible for the mechanisms. The namespaces partition the processes' view of the world, including process ID, mount points, network interfaces, user ID, and interprocess communication structures, and the control groups partition the processes' view of what resources it can consume, such as CPU, memory, disk, and network resources between the resources of the container and the resources of the host [3]. A container is not a small virtual machine, therefore. Because it's an ordinary process running within a limited amount of system resources, a process that's running from within a container will still appear in the host system with a different process ID [3].

The impact on performance is not taken for granted. Aqasizade et al. performed a benchmarks comparison of different virtual machines, containers, and containers running as virtual machines, including a test where they compare the performance of a virtual machine with a virtual machine containing containers to a processor, memory and disk test plus a MySQL transactions per second test [2]. Their conclusions aren't necessarily positive for containers: disk bound database workloads were very successful when running in paravirtualised configurations, meaning the benefit of containers is not consistent. This is because the application being applied to this assignment is a database.

The rest of this section discusses the architecture for storage shown in Figure 1. This was created for this report and not a reproduction of the module materials [10].

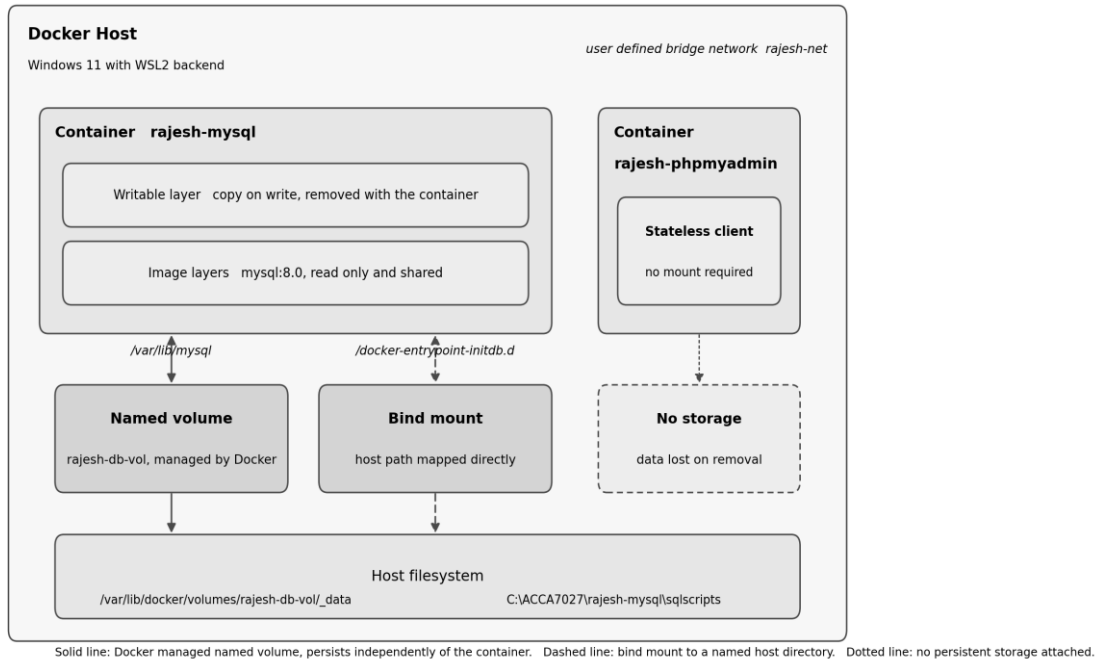


Figure 1. Relationship between the container writable layer, a Docker managed named volume, a bind mount and the host filesystem.

1.2 The Architecture of the Docker Platform

Docker has a client/server architecture. The Docker client sends commands, the Docker daemon builds images and manages the containers, networks and volumes, and a Docker registry store and distributes images. An image is a template that cannot be changed, and consists of layers. A container is a running instance of such an image that has a thin writable layer on top.

Environment where this work takes place is an important qualification. Docker relies on features provided by the Linux kernel, so Docker Desktop for Windows implements them in a virtual machine (VM) with a relatively light virtual operating system (OS), provided by Windows Subsystem for Linux, and Docker Desktop for macOS uses a similar configuration [3]. All practical's mentioned in this document were done on Windows, where an extra level of virtualisation exists between the containers and the physical device. Sobieraj and Kotynski measured this effect, and found it asymmetric, intercontainer network throughput was excellent, but random write performance was very weak, which had an effect on a containerised relational database benchmark [3]. Local timing should therefore not be considered as a production timing but as an indicative measurement of the platform.

The environment was first checked prior to creating any containers. At the beginning, the version of Docker is documented in the following figure, together with the list of containers, images and volumes.

```
PS C:\Users\User> docker --version
>> docker container ls
>> docker image ls
>> docker volume ls
Docker version 29.1.3, build f52814d
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS
2fbea35a860e   metabase/metabase:latest           "/app/run_metabase.sh"  3 weeks ago   Up About a minute   0.0.0.0:30
00->3000/tcp, [::]:3000->3000/tcp   businessos_metabase
6dfe090868e1   postgres:16                        "docker-entrypoint.s..." 4 weeks ago   Up About a minute (healthy) 0.0.0.0:54
adminer:4      1068c07446f4                       170MB          46.7MB        U
metabase/metabase:latest c37d32ea9cef                     1.71GB         736MB         U
postgres:16    287eced1f33b                       642MB          166MB         U
postgres:17    5c855ad7b85e                       645MB          167MB
v3-node:latest 7484884a2523                       197MB          47.4MB
z2m827/v3-node:latest 7484884a2523                       197MB          47.4MB

RAM 2.79 GB CPU 32.63% Disk: 6.56 GB used (limit 1006.85 GB)
```

Figure 2. Verification of the Docker environment prior to deployment.

1.3 Images, Layers and the Dockerfile

A Dockerfile is a specification of how to build a Docker image. Every instruction creates a layer, layers are cached and shared, so if the instruction is identical, it will not be re-executed on the next build, and if the base layer is identical, it will only be stored once and not re-stored multiple times for various images that depend on it. This layering is apparent when trying to retrieve an image from a registry as illustrated in Figure 3, where each layer is pulled and verified separately, before reporting the final content addressed digest.

```
Terminal
7534d1db9f8d: Pull complete
6ef6c7b50a93: Pull complete
96d30d9fbee8: Pull complete
a63160a5eda1: Pull complete
49ec2dab01d9: Pull complete
e3e5d1ac74c1: Pull complete
0d74d296605b: Pull complete
4c8a3e0d4e4b: Pull complete
ab24264a27e9: Pull complete
edf85873f64e: Pull complete
796812c73292: Download complete
af166387641d: Download complete
Digest: sha256:7cdcdc01f13bab2f15cde676d44d01f61fc9f99fe7785e86196dfc07d358ae2b
Status: Downloaded newer image for mysql:8.0
ff2e37fe79f7141a393996e57b5970b502b20ceee8fc222c9f9ae5c6dda88263
```

Figure 3. Layer by layer retrieval of the official MySQL 8.0 image, ending in a content addressed digest.

This Dockerfile builds on that official image, and sets up the initial database and credentials for the coursework environment:

```
# base image is the official MySQL 8.0 release
FROM mysql:8.0

# credentials and starting database for the coursework environment
ENV MYSQL_ROOT_PASSWORD=cepp2025
ENV MYSQL_DATABASE=clouddb

# default MySQL port, published at run time not build time
EXPOSE 3306
```

The port on which the containerised process listens, but does not publish, is documented in EXPOSE and will not be published at runtime with the -p flag to make the image portable. More important, while it's OK to include a root password in ENV when using it in a teaching exercise, it's not OK when deployed in production: all ENV variables baked into an image can be recovered from by anyone who can see its history. For a real deployment, a secrets manager, either Docker secrets or an external secrets manager, would be needed.

1.4 Publishing Images to a Registry

If the image is built locally, an artefact is created which can only be used by one machine. When published to a registry, it becomes a distributable unit: this is where containerisation comes into play for cloud deployment instead of just local deployment. The custom image was marked up and pushed into Docker Hub:

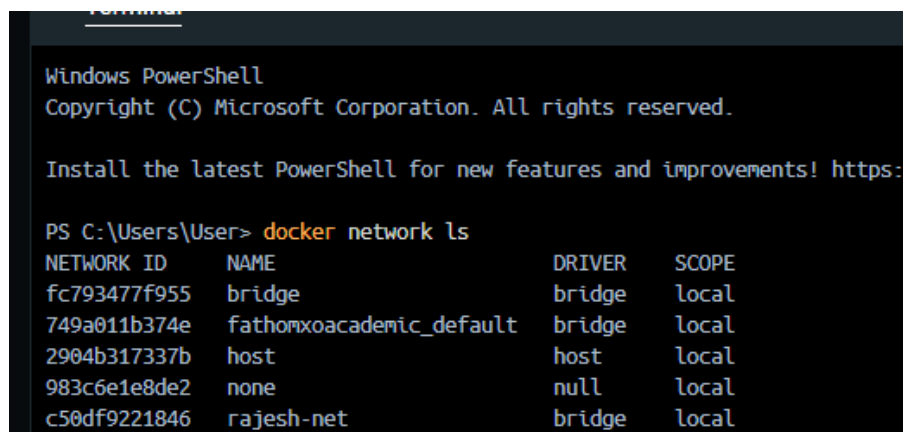
```
docker image build -t rajesh-mysql-custom .
docker image tag rajesh-mysql-custom <dockerhub-username>/rajesh-
mysql:latest
docker push <dockerhub-username>/rajesh-mysql:latest
```

The push will only push what is not already in the registry and it will report back at the end of the push a content addressed digest. It is this digest that makes a registry useful for reproducible deployment: It refers to a specific set of bytes, while a tag like latest is a variable pointer that can point to different content on different days. Deployments that are configured to a tag and not a digest are thus not truly reproducible.

1.5 Container Networking

There are a number of network drivers that Docker offers. Bridge is the default and puts the container in an isolated network with a single host; host is an option which removes the isolation between container and host; overlay is an option that connects multiple daemons, allowing containers on different hosts to communicate; and macvlan gives the container an address that appears as a physical host [7]. A user defined bridge network is the right choice for one machine's multi container application.

It's not just a difference in appearance between the default bridge and a user defined bridge. An automatically created network with the user-defined network name allows services inside attached containers to resolve names automatically, instead of by address [8]. The network and the named volume were first created before the first container was started as shown in Figures 4 and 5.

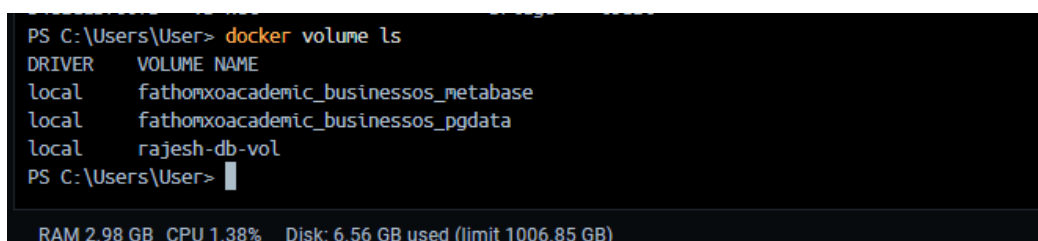


```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PowerShellLatest

PS C:\Users\User> docker network ls
NETWORK ID          NAME                                   DRIVER  SCOPE
fc793477f955        bridge                               bridge  local
749a011b374e        fathomxoacademic_default           bridge  local
2904b317337b        host                                host     local
983c6e1e8de2        none                                null     local
c50df9221846        rajesh-net                          bridge  local
```

Figure 4. The user defined bridge network rajesh-net listed alongside the default networks.



```
PS C:\Users\User> docker volume ls
DRIVER  VOLUME NAME
local   fathomxoacademic_businessos_metabase
local   fathomxoacademic_businessos_pgdata
local   rajesh-db-vol
PS C:\Users\User>

RAM 2.98 GB CPU 1.38% Disk: 6.56 GB used (limit 1006.85 GB)
```

Figure 5. The named volume rajesh-db-vol created before any container was started.

That network was then used to attach the MySQL container, and store the volume in its data directory. As can be seen in Figure 6, the container is running and the initialisation log of the container indicates that the entrypoint script for MySQL ran and started creating the database files.

```

Terminal
PS C:\Users\User> docker container ls
CONTAINER ID   IMAGE          NAMES      COMMAND                  CREATED        STATUS        PORTS
ff2e37fe79f7   mysql:8.0     rajesh-mysql  "docker-entrypoint.s..." 44 seconds ago Up 41 seconds 3306/tcp, 33060
/tcp
2fba35a860e   metabase/metabase:latest  "/app/run_metabase.sh" 3 weeks ago   Up 7 minutes  0.0.0.0:3000->3
000/tcp, [::]:3000->3000/tcp  businessos_metabase
6dfe090868e1   postgres:16   "docker-entrypoint.s..." 4 weeks ago   Up 7 minutes (healthy) 0.0.0.0:5433->5
432/tcp, [::]:5433->5432/tcp  businessos_db
f7637e2ae501   adminer:4     "entrypoint.sh docke..." 4 weeks ago   Up 7 minutes  0.0.0.0:8080->8
080/tcp, [::]:8080->8080/tcp  businessos_adminer
PS C:\Users\User> docker container logs rajesh-mysql
2026-07-27 13:21:00+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 8.0.46-1.el9 started.
2026-07-27 13:21:01+00:00 [Note] [Entrypoint]: Switching to dedicated user 'mysql'
2026-07-27 13:21:01+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 8.0.46-1.el9 started.
2026-07-27 13:21:01+00:00 [Note] [Entrypoint]: Initializing database files

```

Figure 6. The MySQL container running on rajesh-net, with entrypoint initialisation visible in the container log.

The container for phpMyAdmin was launched on the same network using the environment variable PMA_HOST pointing to the name of the container 'rajesh-mysql'. No address was found and no port was published on the database container. In the figure 7 both containers are running and in the figure 8 the network configuration that enables the use of name based addressing is confirmed.

Container CPU usage ⓘ

3.76% / 800% (8 CPUs available)

Container memory usage ⓘ

1.69GB / 5.59GB

Show charts

☐ Only show running containers

☐	Name	Container ID	Image	Port(s)	CPU (%)	Memory usage...	Actions
☐	● rajesh-mysql	ff2e37fe79f7	mysql:8.0		0.95%	377.9MB / 5.72Gi	☐ ⋮ <input type="trash"/>
☐	● rajesh-phpmyad	331f0c0e9fa7	phpmyadm		0.01%	12.38MB / 5.72Gi	☐ ⋮ <input type="trash"/>

Figure 7. Both application containers running, viewed through the Docker Desktop interface.

```
080/tcp, [::]:8080docker network inspect rajesh-net
>> docker container exec rajesh-phpmyadmin ping -c 3 rajesh-mysql
[
  {
    "Name": "rajesh-net",
    "Id": "c50df922184625817bb245fd44aa671dbf2dadf8245093c5dc56d6e58ef69133",
    "Created": "2026-07-27T13:17:04.068806458Z",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv4": true,
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": {},
      "Config": [
        {

```

RAM 5.39 GB CPU 2.62% Disk: 8.54 GB used (limit 1006.85 GB)

Figure 8. Inspection of the user defined bridge network on which both containers resolve one another by name.

The convenience has nothing on the security implication. MySQL is only accessible from the inside of rajesh-net and doesn't publish any port; only phpMyAdmin does. This is network segmentation by one flag and it is the same as the public and private subnets that are being separated in a cloud virtual network.

While deploying, a practical problem presented itself and is here rather than hidden. The service was originally bound to the host port 8080 that was already brought up by another process on the same host. The browser brought up a different app, as seen in Figure 9, when the phpMyAdmin container was functioning properly.

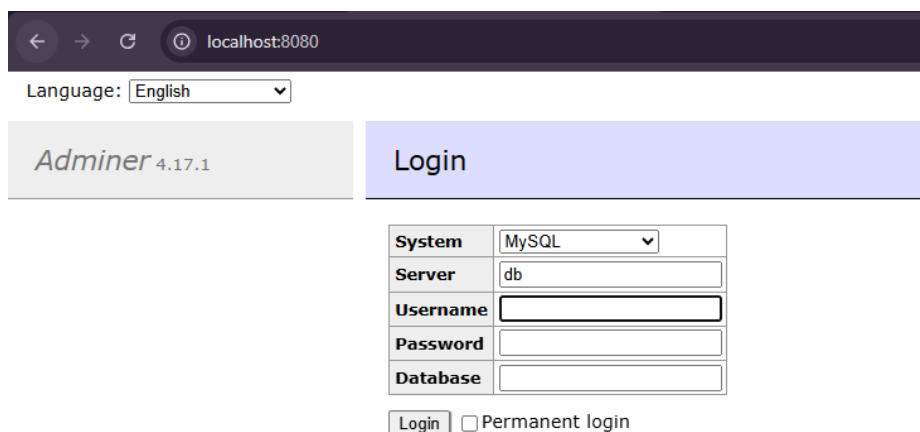


Figure 9. A host port conflict: port 8080 was already bound by an unrelated container, so the expected interface was not served.

The problem was solved by republishing the container on port 8090. This is a true example of the weaknesses of single host container deployments. The port space in the host is finite and shared and a healthy running container doesn't mean the service is accessible at the desired address. One of the advantages of the orchestration platforms is that they can eliminate this type of error by automatically assigning ports and addresses. The corrected deployment is shown in figure 10.

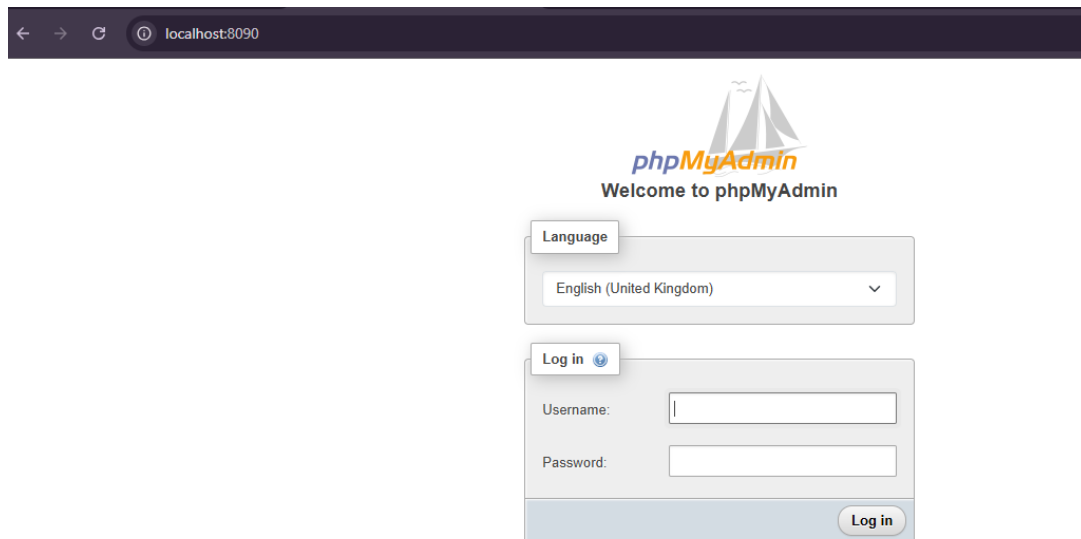


Figure 10. The phpMyAdmin interface served correctly on the reassigned host port.

1.6 Container Volumes and the Persistence Problem

1.6.1 Named Volumes

The writable layer of a container will not persist when the container is deleted, and is only meant for runtime data [6]. Secondly, it's only applicable to databases. The writable layer is created on top of a copy on write file system and the documentation of Docker itself includes that write intensive applications, like database storage, will have an overhead due to this [4] which is why Docker recommends using volumes on such applications.

A named volume is a Docker-defined space that can be created and managed without any involvement of a container and is stored in the default location of Docker's data directory on the host and remounted into containers by name [5, 10]. The volume `rajesh-db-vol` was mounted into `/var/lib/mysql` and that's where MySQL stores its data files. The correctness

argument and the performance argument thus go together and it is therefore a more valid argument than either of them would be if they were independent.

1.6.2 Bind Mounts

A bind mount is a way to designate an absolute host path to be mapped into the container rather than a Docker managed name. The main reason for using volumes over bind mounts is that volumes are portable, Docker controls them entirely, and they are not affected by the directory structure of the host machine, unlike the latter which also means that they cannot be used in orchestrated deployments using orchestration software [5], [10]. Bind mounts are still proper when it is the point that is being bound and not a particular implementation detail, like development workflows where the source code is modified on the host or where configuration is added at start-up.

Both mechanisms were implemented. The initialisation directory of a host machine was bind mounted in the MySQL container at its initialisation point, and a table was successfully queried from the container.

```
docker container run -d --name rajesh-bindtest -e
MYSQL_ROOT_PASSWORD=cepp2025 \
  -e MYSQL_DATABASE=binddb \
  -v C:\ACCA7027\rajesh-mysql\sqlscripts:/docker-entrypoint-initdb.d
mysql:8.0

docker container exec rajesh-bindtest mysql -uroot -pcepp2025 \
  -e "USE binddb; SELECT * FROM seeded_from_host;"
```

The main difference between the two mechanisms is the ownership. In this case, with a named volume, Docker has the ownership of the storage location, while the container refers to it by name, and with a bind mount the host has the ownership of the location. The first one is movable, the second one is not, and that's why there's a volume for the database, and a bind mount for the initialisation script.

1.6.3 Demonstrating Persistence

This section's central was directly tested. Using phpMyAdmin, a table was added to the clouddb database and three rows added to it as illustrated in Figure 11.

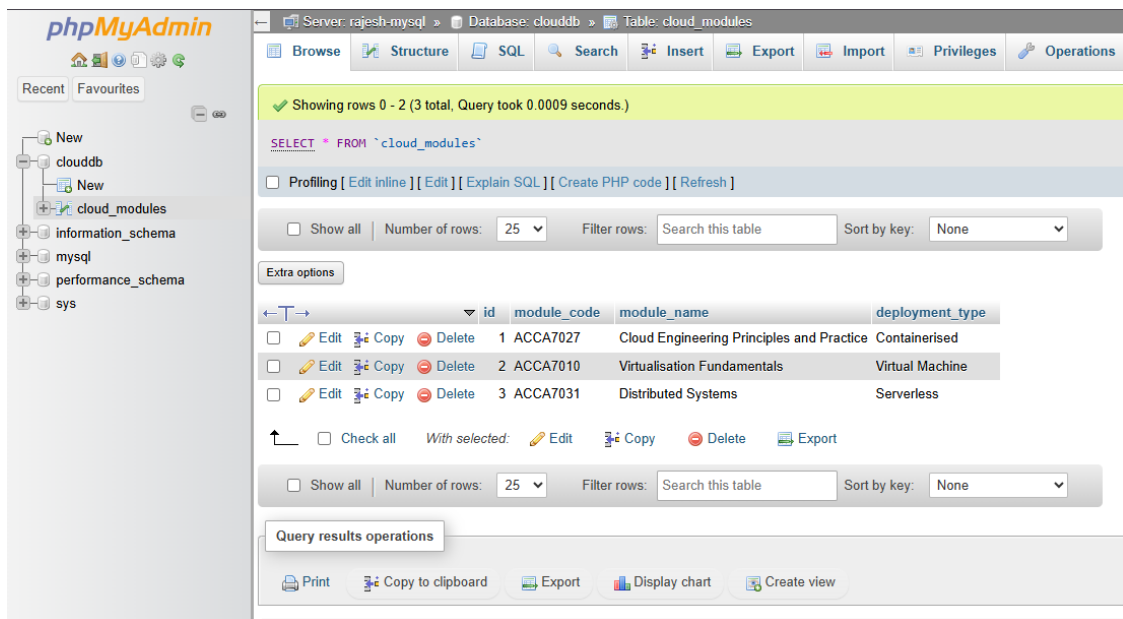


Figure 11. The cloud_modules table populated through phpMyAdmin, served from the rajesh-mysql container.

Both containers were then stopped and destroyed, and the volume listing checked:

```
docker container stop rajesh-mysql rajesh-phpmyadmin
docker container rm rajesh-mysql rajesh-phpmyadmin
docker container ls -a
docker volume ls
```

There were no containers of type rajesh left, however there was the package rajesh-db-vol. Then two completely new containers were created using the same run commands as in Section 1.5 and the database was re-inspected. All three rows were present, and retrieved using containers which were not present when the data was stored.

The outcome is a thorough statement of the argument. No longer does the container exist that wrote the data, but does the data. The property of ephemerality is not a property of the storage layer but an property of the compute layer and Docker volumes is a way in which the two are decoupled. The words “can't be deployed in containers” means that the default is that the container can't host stateful workloads; it's not a limitation, it's the result of the image being an interchangeable, disposable item.

1.7 Critical Evaluation

The demonstration must not be exaggerated. The guarantee of persistence is a much weaker than durability. A named volume is meant to keep the data safe in case the container is removed but does not guarantee that it will stay secure if the host failure, disk failure or accidentally running a volume pruning command, and does not provide any replication or point in time recovery. While Docker volumes provide an effective way to back up a production database, they do not provide the backup and/or replication strategies that are necessary for a production database. The exercise in this document illustrates the mechanism, not a production architecture.

Security is a second, and more basic, qualification. Jarkas et al. compile more than 200 recorded container vulnerabilities of 47 exploit types and 11 attack vectors, noting that a single vulnerability of the host kernel could result in compromising all containers running on that host [9]. This is not true for a hypervisor boundary. For a database that stores regulated data, it is a substantive design choice, not a theoretical one, and that is one way the public cloud providers have to go about running their containers - inside virtual machines, as opposed to bare metal.

The honest one, it's a little bit smaller, but much more useful. A decade ago, that would have been a false statement because it wasn't technically correct to containerise a stateful workload; however, that is no longer the case today, as Docker volumes allows it to be done properly. This varies depending on the durability, isolation and compliance needs of the system being deployed, and is a topic that is revisited when comparing containers with managed and serverless options - a topic that is covered in more detail in Section 3.

2. Load Balancing, Scalability and Application Gateways

These 3 ideas are taught independently and in reality are interdependent. The app discussed is a music streaming service, Spotify, selected for being a large scale consumer service with a highly variable load, and for having primary updates of the infrastructure from Spotify's engineering teams published. For consistency, this will use Google Cloud as the reference provider, but Microsoft Azure is added as a comparison, since it is the provider used in the module tutorials.

2.1 A Cloud Provider's Approach to Load Balancing

A load balancer spreads a user's traffic over a number of instances of an application, which means that any one instance is not likely to suffer from performance issues. Google's implementation is based on the same internal technologies as are found in Google's own products: Google's Maglev load balancing system, Google's Andromeda network virtualisation stack, Google's Google Front End fleet and Google's Envoy proxy [12]. Global external load balancer (ELB): A single ELB address that routes requests to ELB backends across regions by ELB front end nodes across over eighty locations globally [12].

The anycast design isn't just convenient, it's also significant to architecture. This is because one address is broadcasted from several locations, and the client's own network routes the request to the front end that is closest. This translates to a reduction in latency and load balancing as one system, instead of two. Wherever there is a requirement for jurisdictional control over data location, regional proxy load balancers are provided and this has regulatory as well as performance implications on the decisions made about load balancing.

Azure divides the problem into a layer four load balancer to distribute on address and port, and the layer seven service discussed below [11,15]. This difference in layers is the turning point for the remaining part of this section.

2.2 Scalability

It can be vertical - adding resources to an existing instance or horizontal - adding instances. Vertical scaling is limited by the largest machine available (or often a machine restart) while horizontal scaling is limited only by cost, and by the ability of the application to run more than one copy. Cloud platforms are based on the horizontal model.

Google Cloud's horizontal scaling is provided by managed instance groups. A group consists of a number of instances from an instance template and behaves as a single unit; they can be added and removed automatically from a group as necessary based on a target value specified in the autoscaling policy [14]. Signals are processor utilization, load balancing serving capacity, monitoring metrics and schedules, and even a predictive mode can scale in advance of the expected load [14]. The Azure version of this is virtual machine scale sets, which also spreads virtual machines across fault domains or availability zones to prevent the entire set from being removed due to a failure in a rack or datacentre.

It's what happened at scale in the case of Spotify. The biggest one hosted on Kubernetes processes over 10M requests/second when scaled up (substantial benefits of autoscaling) and processor utilization was improved by a factor of two to three with bin packing and multi tenancy [17]. Both of these numbers are important, but the more revealing is the utilisation number - auto-scaling can be used to serve peaks and it can be justified in terms of that, but the real use of it is to not incur a charge for the idle capacity that fixed provisioning would require in steady state.

2.3 Application Gateways

An application gateway is a layer 7 application. Unlike routing based on address and port, Azure Application Gateway routes on HTTP attributes like URL path or host header, and offers TLS termination, autoscaling, zone redundancy and integration with a web application firewall [15]. It uses path based routing to route traffic to different pools based on the path information, i.e. requests to one path prefix are sent to servers optimised for video while the other path prefix is sent to servers optimised for images, with a default pool for any requests that don't match up to any other path prefix [16].

Similarly, on Google Cloud, the equivalent is the external Application Load Balancer, which has the same capabilities as the previous one, plus the ability to share the traffic with a content delivery network and the DDoS protection service offered by Google Cloud [13]. In both, the result is a convenient decoupling: what a service exposes to clients doesn't need to be the same as what's exposed by the servers in its backend, and it is easy to split, merge or move backends without changing the addresses that are published.

2.4 Why Load Balancing and Scaling Are Complementary

Both methods are inadequate on their own and the why, is exactly the same. A load balancer in front of a fixed pool of backends distributes traffic evenly, and does not add capacity; when all of the instances are full, the load balancer distributes degradation evenly, rather than distributing work. But that's quite useless if there is no load balancer; it causes new instances to be launched, but no traffic to reach them, since the clients use the addresses of the old instances.

They are physically connected instead of conceptually. Google Cloud allows an autoscaler to scale a managed instance group on load balancing serving capacity [14] that is, the load balancer measures the utilisation of its backends to decide on scaling. The same applies to health checking; the load balancer removes traffic from unhealthy instances in the instance group and the load balancer re-creates the traffic. The two systems are a feedback loop, one measuring the demand and redistributing it, and the other adjusting supply.

That's the time factor that limits that loop. The time to provision a virtual machine and warm an application is in the order of minutes, so autoscaling can't respond to a sudden spike in real time and the load balancer must take up the time by distributing the load among the capacity available at the time. It is a significant advantage of containers as it takes only a matter of hundreds of milliseconds to start a container image, compared to the time taken to start a virtual machine, thus reducing the time that the load balancer must span.

2.5 The Benefit of Combining All Three

By combining all three technologies, a service property is achieved that is not provided by any of the technologies alone. A request comes from a customer to a single published address which is passed through the network to the closest front end where it is checked at the application level by the gateway and passed to the nearest backend pool determined by path. This is a managed instance group, whose size is controlled by an autoscaler based on the utilisation parameter taken from the load balancer, and the health checks are used for both determining the routing decision and replacing failed instances.

One hostname and one certificate are visible to the customer. The operator gets a much less tangible, but more valuable benefit: each path can be scaled separately, a promotional event that brings traffic to one part of a service will scale only the pool corresponding to that path,

and a fault in one backend pool will not take out the others. Security policy also has one time meaning at the gateway and not reimplementing it in any of the back end services.

This is not an arrangement that doesn't cost anything. The individual components are individually priced, there is a wide surface area of configuration, and there is a single logical gateway with a correspondingly wide blast radius, if it is misconfigured. When traffic is predictable, and the service is simple, a single instance with a simple load balancer can provide sufficient availability at a lower cost. The link-up is made on the basis of size and range of demand, rather than its sophistication.

2.6 Relation to the Deployed System

These mechanisms have a recognisable single host analogue that is deployed in the deployment described in Section 1. When the cloud load balancer attaches one address to many backends the equivalent service discovery is done by the user-defined bridge network, enabling phpMyAdmin to connect to the database by name without having to provide any address to be configured.

It becomes an instructive analogy when it doesn't hold true. A relational database cannot scale horizontally as the writes to independent copies of the database would be different. None of these are provided by a load balancer: horizontal database scaling can only use read replicas, sharding or consensus protocol. This is the main reason why Spotify, after moving their stateless portion onto Kubernetes, replaced the host based, or self-hosted, instances of stateful resources like Cassandra and PostgreSQL with provider managed variants, instead of moving them onto Kubernetes. A stateful workload can be deployed using containerisation, but is not horizontally scalable.

3. Driving Forces and the Serverless Alternative

3.1 The Main Driving Forces

Cost is most often mentioned as a driver, and most often misunderstood. In the study by Merlo et al., who conducted a systematic review and a survey, the benefit that was most often reported in the literature that was analysed was reduction in upfront cost, when considering cloud adoption as an operational expenditure (OE) versus capital expenditure (CapEx) [23]. The same study shows that security, privacy, accessibility, and cost continue to be a concern for organisations, putting cost on both the side of the ledger [23]. A compatible conclusion is drawn by Santos et al., who found that factors that influence decisions on adoption are security, cost effectiveness, scalability, interoperability and compliance with regulations [24].

The seeming paradox is resolved if cost is regarded in terms of the shape of the demand curve instead of the prices. The benefits of consumption billing are that when the demand is volatile, it will not cost as much because the instance will not be running all the time; the drawbacks are that when the demand is predictable, I will pay more than the amortised cost of the owned infrastructure because I will have to pay extra for the unused capacity. The driver is elasticity, with cost as its consequence.

The second and more or less autonomous force is flexibility. Previously, a purchase order needed to be requested and a delivery lead time had to be observed before the provisioning process could begin, but now a call to an API can be made and the provisioning process can begin, which is both quick and different types of experiments can be run by an organization.

3.2 Serverless Functions

Function as a service is the logical culmination of that journey, which is better known as serverless computing. AWS Lambda is a compute service that executes code in response to events without the provisioning or management of servers, allocate and free the necessary compute resources automatically [21]. Microsoft claims Azure Functions is “a managed platform service that provides event driven and event scheduled compute for the developer to own just the code” [22]. There are some properties that are common to all providers: Each invocation of the service launches a new instance of the service; The service is stateless, meaning that no knowledge of the current state of the program is maintained across invocations; The platform instantiates the service as the number of invocations increases and

shuts it down when it is idle; Billing is based on the number of times the service is invoked and the length of time it takes to execute.

That's an oversimplification, as that is definitely a part of servers. The operator assumes responsibility for these and the trade to remove them is what is compared below.

3.3 Comparing Serverless Functions with Containerisation

Table 1 summarises the principal differences between the two approaches.

Table 1. Comparison of containerisation and serverless functions across seven dimensions.

Dimension	Containerisation	Serverless functions
Unit of deployment	An image containing the application and its dependencies	A single function bound to an event trigger
State	Persistent state supported through volumes	Execution is stateless, external storage required
Scaling granularity	Per container instance, governed by the operator	Per invocation, governed by the platform
Idle cost	Charged while the container runs	No charge when no invocation occurs
Start latency	Predictable once the image is cached	Cold start can exceed several seconds [20]
Runtime control	Full control of base image, libraries and configuration	Runtime and patching controlled by the provider
Portability	Image runs on any conformant container runtime	Function is coupled to provider triggers and bindings

The differences that have been most studied and most important are the difference in cold start latency. If the platform doesn't have a warm instance, it has to spawn a new one before invoking the function, and the delay added to the request. Golec et al. did a systematic survey of the available literature and distinguished mitigations into techniques that shorten the cold start

duration and techniques that decrease the frequency of cold starts, like periodic pinging, container pre-preparation and scheduling strategies [19]. The fact that there is a body of work that exists is proof that the issue is not incidental to the Act.

Scale is confirmed by measurements taken on the scene of production. In order to investigate the impact of cold starts on a commercial serverless platform, Chen et al. took a look at a dataset containing 80 billion user requests and 11.9 million cold starts that occurred on a commercial serverless platform and concluded that cold starts took more than seven seconds in some regions, and that dependency deployment time and scheduling delay were the two main contributors [20]. A latency of that magnitude ends up being acceptable for an asynchronous background operation, but not for an interactive request; that's the edge between the two technologies.

They are split into small groups according to their portability. A container image is subject to an open specification, and is loaded on any runtime that complies with the specification, thus limiting migration costs among providers. A function is coded against provider specific triggers and bindings and is indeed tightly coupled, which is why Microsoft has published formal guidance on migrating Lambda workloads to Azure Functions, including adaptation of function signatures [22]. It's useful to have a migration path, but it's the need for it that's the point.

3.4 Application to the Deployed Workload

If we extend this comparison to the system developed for this assignment we get an easy answer. MySQL is a long lived process, which keeps client connections, loads data in memory between requests and continuously writes to an on disk dataset. Serverless functions are designed to be stateless (no local state), have a max run time, and no durable local storage (no local storage). This workload is best suited to containerisation with a named volume, as the persistence test in Section 1.6.3 shows that this is a sufficient test for a single host.

The general principle is that neither is mutually exclusive of the other and a third may supersede both. The ideal production architecture would be to use stateless request handling in containers, use functions that accept events when they are intermittent (cold start cost isn't a factor), and use the managed service that provides replication and recovery guarantees that neither containers nor functions do. Spotify did it instead [18] and this is the trend that was followed by the forces in Section 3.1, which do not choose one technology, but distribute workloads over multiple.

4. Conclusion

With this report I looked at containerisation and tested a specific claim looking at an example deployment of MySQL and phpMyAdmin. Even though each of the steps-building an image from a Dockerfile, publishing it to a registry, connecting containers over a user defined network and attaching a named volume-are simple operations, they show that the ephemerality of containers extends itself only to the compute layer. The data stored in one container was intact after it was transferred to another container after the destruction of the first container.

The assessment did not go overboard in that result. Being persistent on one host isn't being durable, sharing the kernel reduces isolation compared to the hypervisor, and a containerised database is still challenging to scale horizontally. We addressed how load balancing, autoscaling, and application gateways create a feedback loop that provides elasticity for stateless tiers in Section 2 and how cost and flexibility provide elasticity to the tiers in Section 3 without selecting one.

So the takeaway here is that containers enabled the deployability of stateful workloads, but not for the other workloads out there.

5. References

- [1] Cloud Native Computing Foundation, "The CNCF Annual Cloud Native Survey: The Infrastructure of AI's Future," CNCF, Jan. 2026. [Online]. Available: <https://www.cncf.io/reports/the-cncf-annual-cloud-native-survey/>
- [2] H. Aqasizade, E. Ataie, and M. Bastam, "Experimental assessment of containers running on top of virtual machines," IET Networks, 2025, doi: 10.1049/ntw2.12138. Open preprint available: arXiv:2401.07539.
- [3] M. Sobieraj and D. Kotynski, "Docker Performance Evaluation across Operating Systems," Applied Sciences, vol. 14, no. 15, art. 6672, Jul. 2024, doi: 10.3390/app14156672.
- [4] Docker Inc., "Storage drivers," Docker Docs. [Online]. Available: <https://docs.docker.com/engine/storage/drivers/> (accessed Jul. 2026).
- [5] Docker Inc., "Volumes," Docker Docs. [Online]. Available: <https://docs.docker.com/engine/storage/volumes/> (accessed Jul. 2026).
- [6] Docker Inc., "Persisting container data," Docker Docs. [Online]. Available: <https://docs.docker.com/get-started/docker-concepts/running-containers/persisting-container-data/> (accessed Jul. 2026).
- [7] Docker Inc., "Network drivers," Docker Docs. [Online]. Available: <https://docs.docker.com/engine/network/drivers/> (accessed Jul. 2026).
- [8] Docker Inc., "Bridge network driver," Docker Docs. [Online]. Available: <https://docs.docker.com/engine/network/drivers/bridge/> (accessed Jul. 2026).
- [9] O. Jarkas, R. Ko, N. Dong, and R. Mahmud, "A Container Security Survey: Exploits, Attacks, and Defenses," ACM Computing Surveys, vol. 57, no. 7, pp. 1 to 36, 2025, doi: 10.1145/3715001. Open version: University of Queensland eSpace, record UQ:10c44b5.
- [10] M. Huntly, "L12 Container Volumes," ACCA7027 Cloud Engineering Principles and Practice, lecture notes, University of Wales Trinity Saint David, 2026.
- [11] M. Huntly, "L6 Load Balancing," ACCA7027 Cloud Engineering Principles and Practice, lecture notes, University of Wales Trinity Saint David, 2026.

- [12] Google Cloud, "Cloud Load Balancing overview," Google Cloud Documentation. [Online]. Available: <https://docs.cloud.google.com/load-balancing/docs/load-balancing-overview> (accessed Jul. 2026).
- [13] Google Cloud, "Application Load Balancer overview," Google Cloud Documentation. [Online]. Available: <https://docs.cloud.google.com/load-balancing/docs/application-load-balancer> (accessed Jul. 2026).
- [14] Google Cloud, "Using autoscaling for highly scalable applications," Google Cloud Documentation. [Online]. Available: <https://docs.cloud.google.com/compute/docs/tutorials/high-scalability-autoscaling> (accessed Jul. 2026).
- [15] Microsoft, "What is Azure Application Gateway," Microsoft Learn. [Online]. Available: <https://learn.microsoft.com/en-us/azure/application-gateway/overview> (accessed Jul. 2026).
- [16] Microsoft, "Azure Application Gateway URL-based content routing overview," Microsoft Learn. [Online]. Available: <https://learn.microsoft.com/en-us/azure/application-gateway/url-route-overview> (accessed Jul. 2026).
- [17] Cloud Native Computing Foundation, "Spotify Case Study," Kubernetes.io. [Online]. Available: <https://kubernetes.io/case-studies/spotify/> (accessed Jul. 2026).
- [18] Spotify Engineering, "Fleet Management at Spotify (Part 2): The Path to Declarative Infrastructure," May 2023. [Online]. Available: <https://engineering.atspotify.com/2023/05/fleet-management-at-spotify-part-2-the-path-to-declarative-infrastructure>
- [19] M. Golec, G. K. Walia, M. Kumar, F. Cuadrado, S. S. Gill, and S. Uhlig, "Cold Start Latency in Serverless Computing: A Systematic Review, Taxonomy, and Future Directions," ACM Computing Surveys, vol. 57, no. 3, 2024, doi: 10.1145/3700875. Open preprint available: arXiv:2310.08437.
- [20] Q. Chen et al., "Serverless Cold Starts and Where to Find Them," in Proc. 20th European Conference on Computer Systems (EuroSys), 2025, pp. 843 to 859. Open preprint available: arXiv:2410.06145.
- [21] Amazon Web Services, "AWS Lambda Documentation." [Online]. Available: <https://docs.aws.amazon.com/lambda/> (accessed Jul. 2026).

- [22] Microsoft, "Azure Functions documentation," Microsoft Learn. [Online]. Available: <https://learn.microsoft.com/en-us/azure/azure-functions/> (accessed Jul. 2026).
- [23] T. R. Merlo, F. Fard, and S. Hawamdeh, "Cloud Computing's Impact on the Digital Transformation of the Enterprise: A Mixed-Methods Approach," *Sustainability*, vol. 17, no. 13, art. 5755, Jun. 2025, doi: 10.3390/su17135755.
- [24] A. Santos, J. Martins, P. D. Pestana, R. Goncalves, H. Sao Mamede, and F. Branco, "Factors Affecting Cloud Computing Adoption in the Education Context: Systematic Literature Review," *IEEE Access*, vol. 12, pp. 71641 to 71674, 2024, doi: 10.1109/ACCESS.2024.3400862.