

Cloud Engineering Principles and Practice

Docker Containerization and Cloud Concepts

Module	ACCA7027 Cloud Engineering, principle and practices
Student name	Zannatul Mawya Priya
Student ID	2527766
Video demonstration link	https://youtu.be/H0aTUTLpDYI

Contents

1. Introduction	3
2. Virtualisation and containerisation principles.....	4
3. Practical implementation and analysis	6
3.1 Environment setup and runtime verification.....	6
3.2 Application files and image build process	7
3.3 Local execution and browser validation	8
3.4 Image publication and registry lifecycle	9
3.5 Persistent storage using Docker volumes.....	10
3.6 Software-defined networking.....	11
3.7 Final verification, cleanup and lessons learned.....	12
4. Cloud provider implementation of load balancing, scalability and application gateways.....	15
5. Serverless computing versus containerisation	17
6. Driving forces behind cloud expansion.....	18
7. Conclusion	19
References	20
Appendix A.....	22

1. Introduction

Cloud computing is no longer just a specialized way to provide infrastructures, today cloud computing represents the new standard (default) environment that organizations use to build and deploy their digital applications. Organizations need fast deployments, scalability when needed, low or zero upfront costs, and consistent methods of reproducing development, testing, and production environments. As organizations have come to expect all of this from cloud computing, cloud engineers are no longer able to simply rent virtual machines and call themselves "cloud engineers." Instead, cloud engineers will need to include other activities such as image based packaging of applications, automated delivery pipelines, software defined networking, persistent storage abstractions, and higher level execution models (e.g., serverless). In addition to the many higher-level concepts associated with cloud engineering, Docker is currently considered to be one of the most important practical tools available because Docker allows developers to create portable units of an application including all of its dependencies that can be built once and then deployed in multiple different environments [3], [7]. This report provides a general overview of key cloud engineering concepts using both theoretical approaches and hands-on experience. Theoretical discussions cover topics such as virtualization, containerization, Docker Images, Persistent Volumes, Container Networking, Service Models, Orchestration Challenges and how containerized applications relate to serverless computing. Practical experience is provided by completing a Docker lab session on Ubuntu Linux. This included installing and testing Docker, building a simple Web Application based on Nginx, and building and deploying a custom Docker Image. Additionally, the report describes pushing a custom image to Docker Hub. Furthermore, the report covers creating a Volume to show persistence in Docker Containers and creating a Custom Network to show how containers communicate with each other. Finally, the report shows how to display the System State at the end of the lab. In addition to describing what was done during the lab session, the purpose of this report is to explain why each task performed during the lab session is relevant to cloud engineering. Therefore, this report combines evidence collected directly from performing the lab session with critical analysis of recognized literature and industry guidelines [1], [2], [5], [11].

2. Virtualisation and containerisation principles

Virtualisation became the foundation of the majority of public clouds. A hypervisor, which presents itself as a hardware layer, creates a layer of abstraction between the underlying physical hardware and all the guest operating systems, enabling several operating systems to share a single virtualization layer. It benefits included enhanced hardware utilization, enhanced resources isolation, faster provisioning, and enabling of essential NIST cloud attributes (e.g., rapid elasticity, on demand self-service, and resource pooling) [1].

Virtual machines do have their drawbacks though. A VM requires a complete guest operating system which consumes more of the host resources in terms of disk space, memory and even startup time. This delay can hinder quick teams. Containerisation addresses this issue by providing virtualisation at the level of the operating system. Containers contain the application code, libraries, dependencies and share the kernel with the host machine [3]. They are thus able to be smaller, faster and more efficient than a full virtual machine [4].

Containers are useful for their portability. A Docker image can run on many environments (developer machine, test server, on site Linux, cloud platform) with only minimal modifications. This enables Devops, CI/CD, automation, and repeatable deployment. Pahl argues that the docker containers created a more standardised and efficient way of packaging cloud native software [5].

However, containers are still not a complete replacement for VMs. VMs generally offer much stronger isolation as they have their own guest kernels. Containers may poses some risks related to security and multi-tenancy if not properly managed [11]. Hence, many Cloud platforms use both: They offer virtualized instances for the customers, who run containers on top of them or use a managed container services.

All these concepts could be proven through my Docker practical as well. A plain webpage was built into a Docker image based on Nginx and run using port mapping from host to container. This small task illustrated the concept of packaged deployment, exposing services, immutable images, publishing image remotely, software defined storage and networking.

In practice, VMs are useful for those legacy apps, heavy workloads, full isolation and max administrative control. Containers are more suitable to develop microservices, CI/CD and high portable apps. Serverless functions allowed developers to run event-driven code together on a largely provider-managed platform [12], [13].

Firstly, containerisation is prized not simply because its lightweight but also due to the fact that it provides operational consistency. Being able to build, test, and promote a Docker image from one environment to the next and simply roll back is a major benefit in a modern cloud engineering environment.

Model	Operational unit	Strengths	Typical limitations
Virtual machine	Full guest OS plus application	Strong isolation, broad control, legacy compatibility	Heavier footprint, slower start-up, more patching overhead
Container	Application plus dependencies sharing host kernel	Portability, density, fast start, consistent packaging	Shared-kernel security concerns, orchestration needed at scale
Serverless function	Event-driven code execution on provider platform	Low ops overhead, fine-grained scaling, pay-per-use potential	Cold starts, limits and provider lock-in concerns

Table 1. Comparison of virtual machines, containers and serverless functions

3. Practical implementation and analysis

The practical work was carried out on Ubuntu Linux using Docker Engine with a basic custom web application that is served with Nginx. The workflow had a logical sequence of installation and verification, up to publication, persistence, networking and cleanup. Every step generated command-line or browser artifacts. The main screenshots are included in the body of the report, while additional supporting screenshots are included in Appendix A.

The practical tasks and the principle of cloud engineering that each task corresponds to are summarised in Table 2.

Stage	Key action	Evidence captured	Cloud relevance
1	Install and verify Docker	Package output, service status, hello-world pull	Runtime readiness and host preparation
2	Create HTML and Dockerfile	Custom files shown in terminal	Declarative packaging and reproducibility
3	Build image	Successful build and image list	Versioned artefact creation
4	Run container	docker ps and browser output	Service execution and port exposure
5	Publish to Docker Hub	Login, push and repository page	Remote registry lifecycle
6	Create volume	Volume list and proof file retrieval	Persistent storage beyond container lifecycle
7	Create custom network	Network list and inspect output	Software-defined connectivity and isolation

Table 2. Practical workflow, evidence and cloud engineering relevance

3.1 Environment setup and runtime verification

The first stage was environment preparation. Docker packages were installed on Ubuntu, after which the Docker service was started, enabled and checked. The service status output verified that the Docker daemon was running and the downloading of the hello-world image was successful. This is important since the operation of container tooling requires the operation of a running daemon or compatible runtime layer. This step in cloud engineering ensures the control plane of local container operations. A healthy runtime service is needed without which subsequent processes like image build, storage mounting and network attachment would not work.

The service check also illustrates that there is an awareness of operations readiness. Cloud workloads are based on background services, socket activation, logging and health of daemons. Status view is thus a local analogue of determining the availability of a managed service in a cloud console. The screenshot evidence substantiates the argument that the system was not only set up but it was actually prepared to do container tasks (see Figure 1).

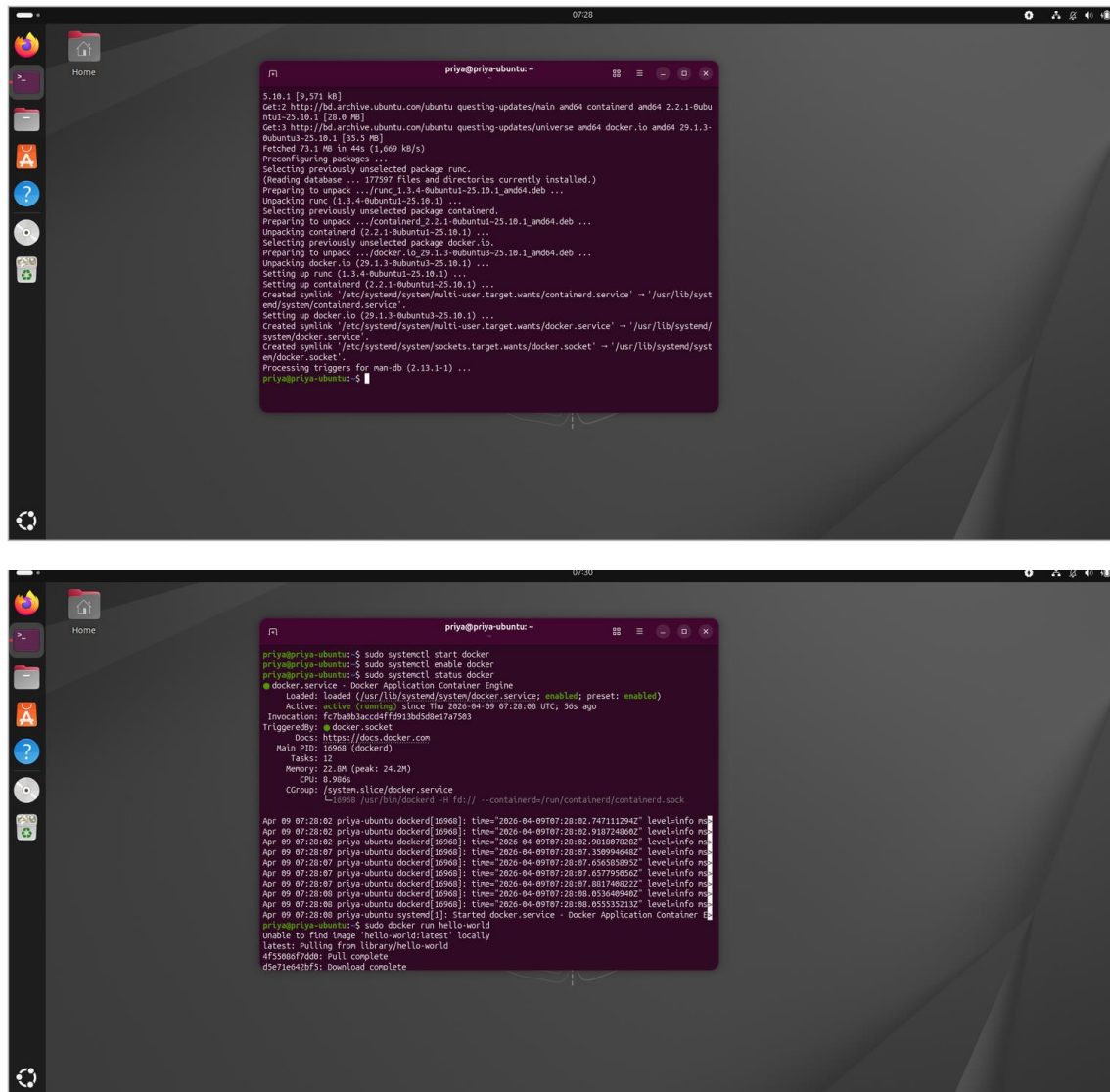


Figure 1. Installation and service verification for Docker on Ubuntu.

3.2 Application files and image build process

When the runtime has been verified, a project directory was created with a couple of files: a trivial HTML page and a Dockerfile. The HTML displayed a message confirming the requirements had been met. The Dockerfile used the official Nginx image as the base image and copied the custom-created index.html into the default web root used by Nginx, in addition to exposing port 80. This exemplifies image-based packaging - the service is packaged with a set of instructions, a portion of which can be repeated instead of providing a dynamic environment. Docker translates the instructions into layers of image components [7], [10].

The build output indicated that the nginx-based image had been built successfully and was tagged in a personal repository namespace. Image tags are useful for making builds identifiable and manageable when sharing them, or feeding a similar build into a subsequent environment. Using the same Dockerfile/files can also produce the same service behaviour with less configuration drift, which would be less convincing than starting from scratch manually.

The build stage was an opportunity for more critical thought. The commented on screenshot indicated that the builder being used was outdated, and that the use of BuildKit is recommended. For a classroom exercise

this sufficed, but in production the use of more robust steps is necessary, such as using BuildKit, pinning image base version or digest ('latest' should be avoided), use security scanning and where possible using a non-root image [10], [11]. Figure 2 illustrates the contrast between a learning artifact and a good production cloud engineering artefact.

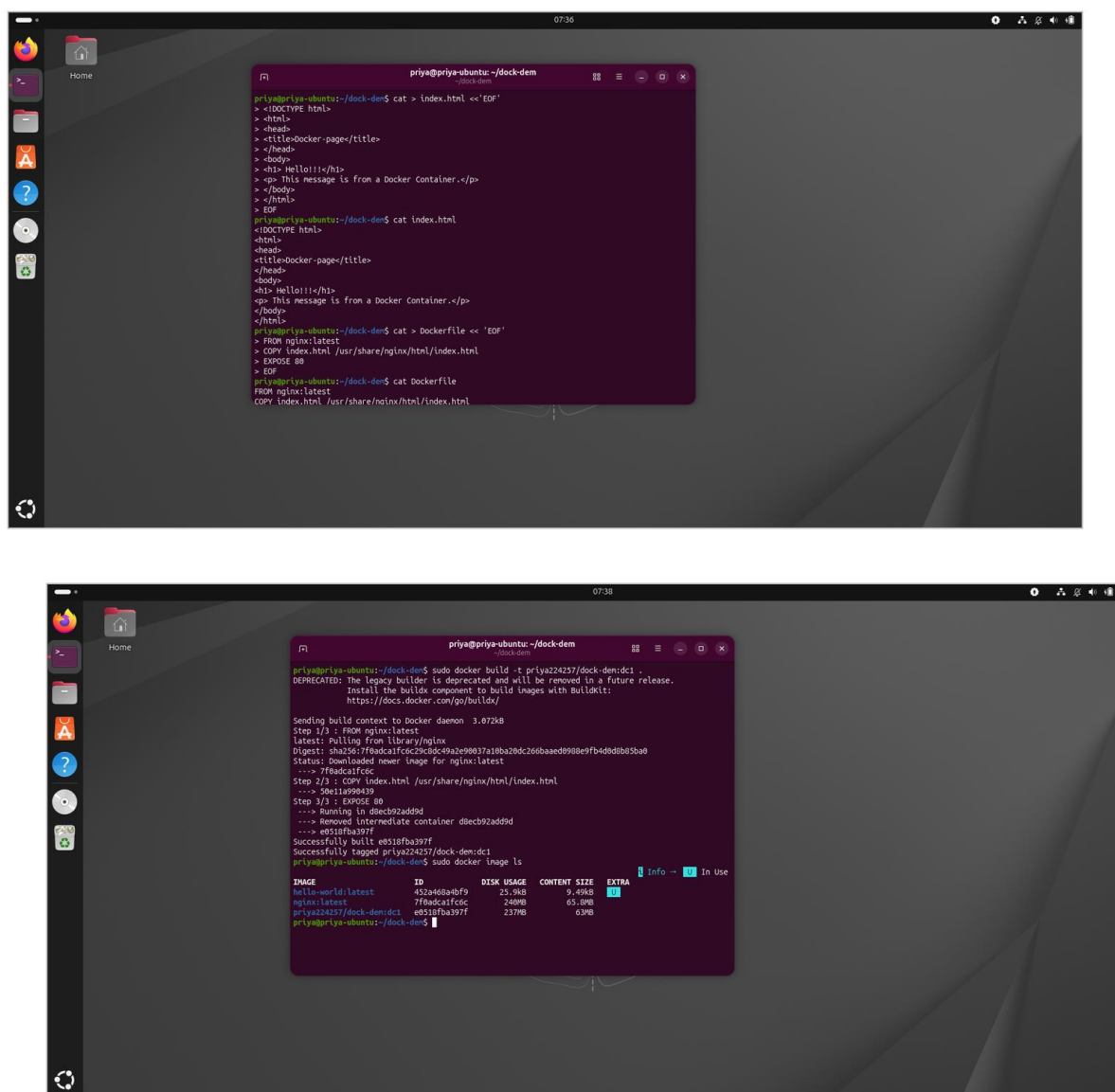


Figure 2. Source-file preparation and successful image build from the Nginx base image.

3.3 Local execution and browser validation

After the image was created, a container was started in detached mode with an explicit name and host-to-container port mapping of 8080:80. The browser screenshot, in turn, verified that the application could be accessed through localhost:8080 and provided the anticipated message. This is the most obvious stage of the exercise since it demonstrates that not only was the image constructed, but it was done as a networked service. The terminal output also indicated the running container in docker ps that confirmed the mapped port and container state was correct.

Technically, this step demonstrates several important container concepts. The former is runtime isolation: the application does not get installed on the host operating system, but is instead executed with a container context. The second one is service exposure via port binding. Containers often listen on an internal service

port, and are accessed by the users and dependent systems via host ports, proxies, gateways or service meshes. This reflects the cloud pattern of exposing an application via a controlled network path as opposed to the presumption of direct machine access, even in a small lab. The third one is disposability. At this point the container was stopped and removed without affecting the existing setup of the host which demonstrates the disposable nature of container instances.

A page loaded correctly in the browser proves more than just a port binding; it tests that the file-copy command in the Dockerfile had worked correctly and the default web server configuration of the base image was good enough to serve the new page. That is, the final product of the entire packaging chain, the output of the browser, is the ultimate validation of the entire process: source code, build command, image generation, container execution and network access. That is why the browser evidence is especially useful in a report. It embodies the functional success as opposed to command success (see Figure 3).

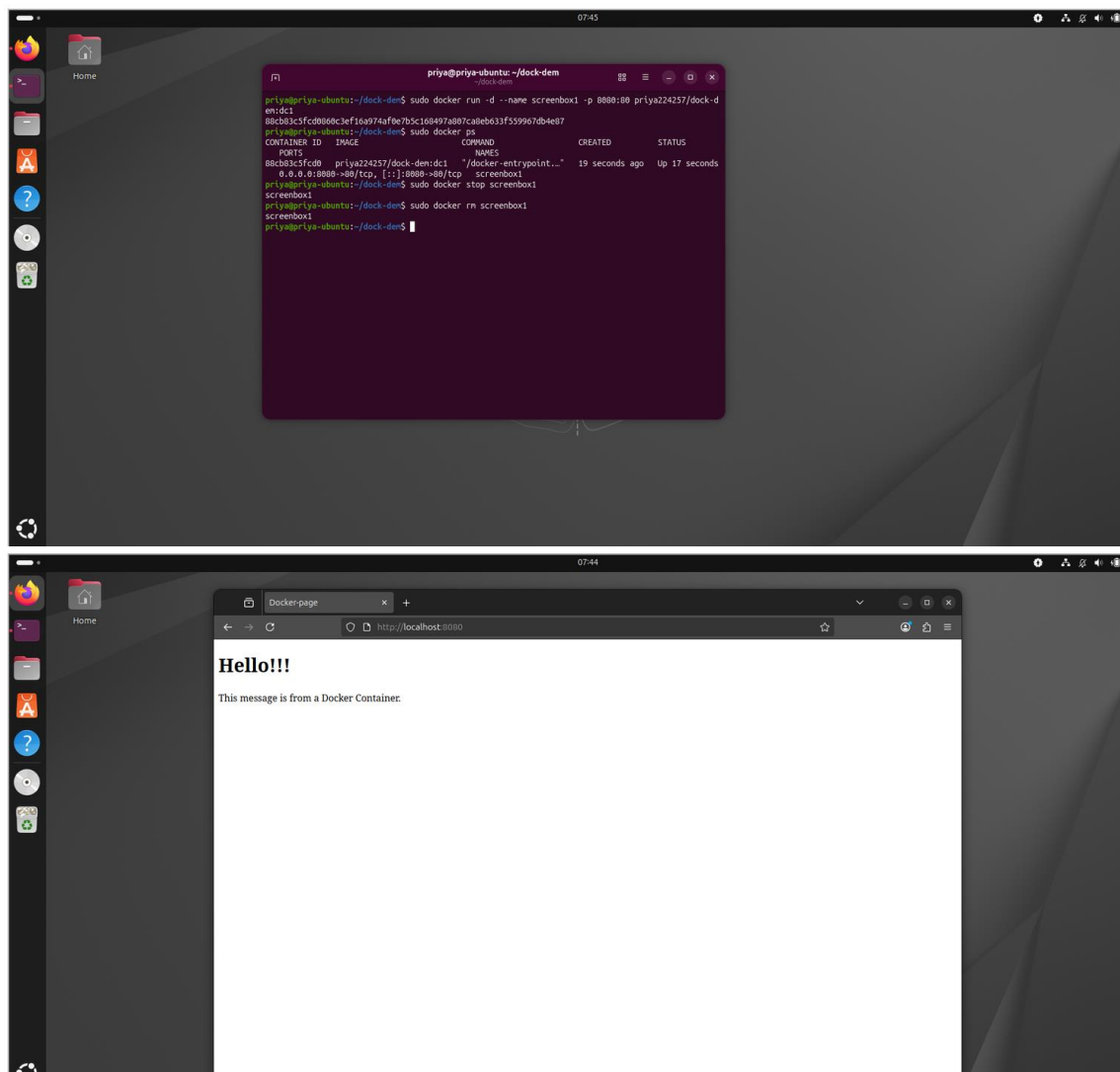


Figure 3. Container launch, port mapping and end-to-end browser validation.

3.4 Image publication and registry lifecycle

After local testing, the custom image was pushed to Docker Hub. The evidence that has been put into practice demonstrates a successful login, push operation and repository page where the image tag appears online. This step is essential since cloud-native delivery depends on registries. An image stored on a single local machine is operationally ineffective within a team, or to be deployed in the cloud. The artefact is made

accessible to reuse in CI pipelines in the future by pushing the image to a remote registry, orchestration platforms and recovery scenarios [7], [10].

Another important part of the image lifecycle is publishing. A Docker image is not a build product, but a distributable software product. Teams can pull it onto another host, it can be used by the teams as the foundation of a deployment, or it can be scanned by the teams to detect vulnerabilities before being promoted. This is just one of the reasons why image naming and tagging discipline is very essential. A poor tagging practice leads to confusion over the image to be used whereas a good tag policy enables traceability and rollback.

Another aspect that is important to analyze is the output of the login. Docker cautioned that, unless a credential helper is implemented, credentials are stored in plaintext in the local configuration file. This is a common and understandable trade-off in a learning environment. Handling of secrets in a production or an enterprise setting should however be more robust. The short lived tokens, credential helpers, secret stores and least-privilege access policies would all be desirable [11]. Recognising this warning strengthens the report because it shows awareness of operational risk (see Figure 4).

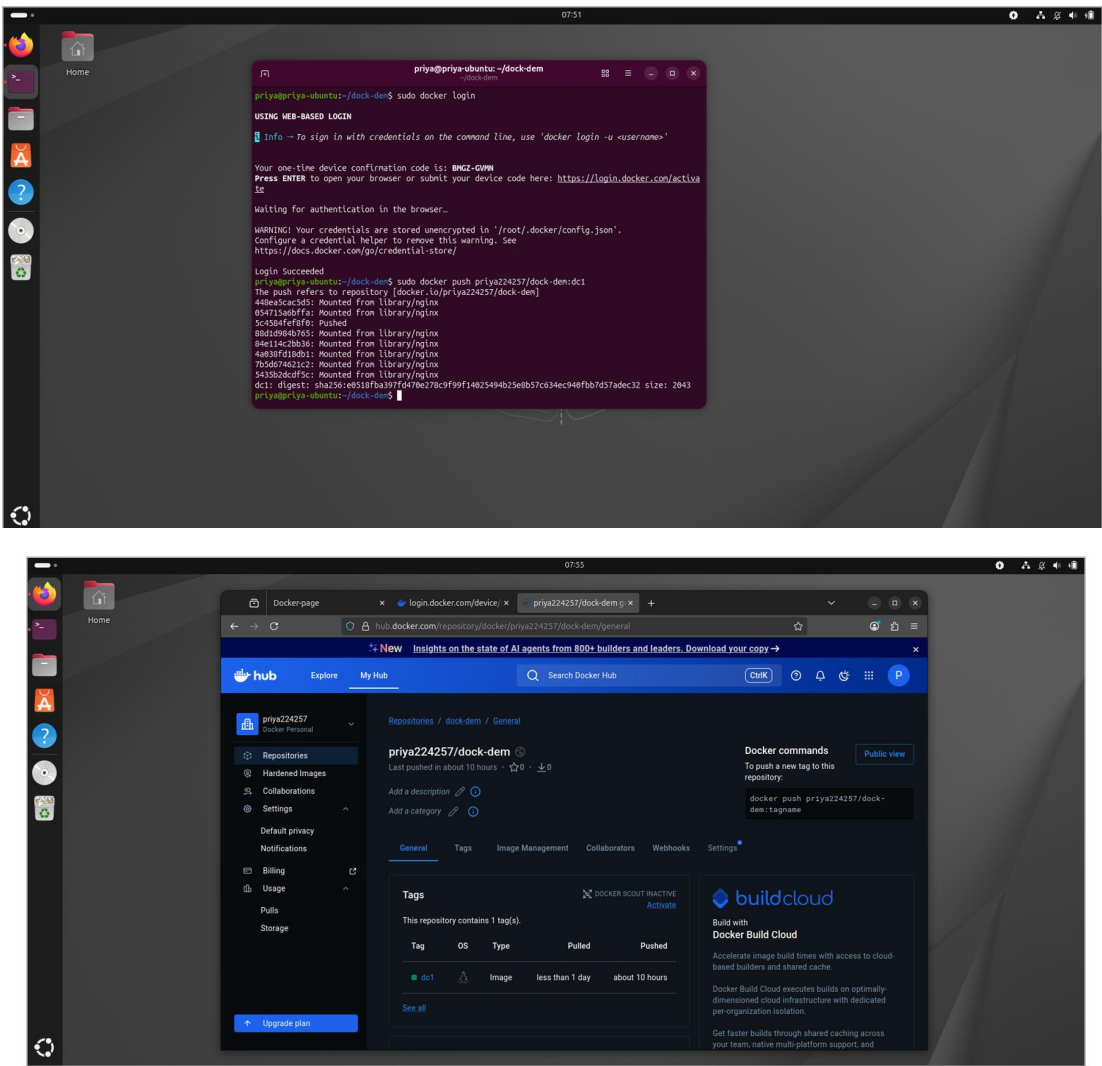


Figure 4.

Docker Hub publication evidence from command line and repository view.

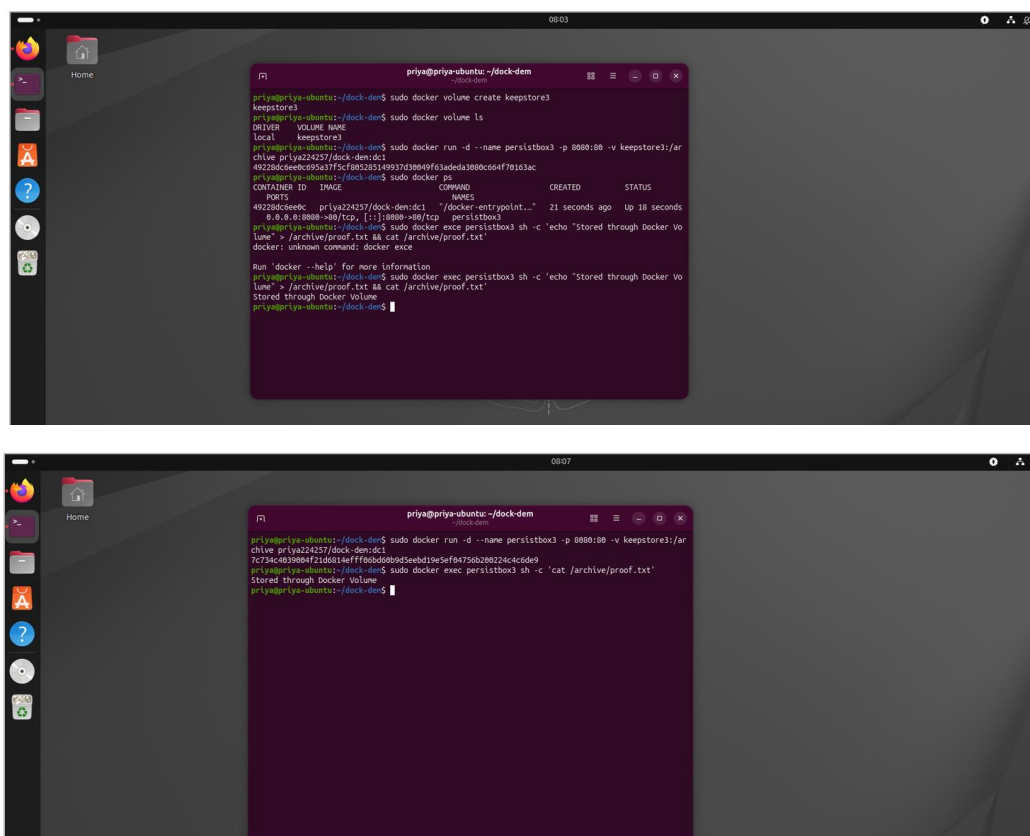
3.5 Persistent storage using Docker volumes

This led to persistent data being written to. They were enabled by creating a Docker volume, listing that volume and mounting this into a running container. A file proof was written in the mounted directory and

read successfully. The distinction between short-term container state and persistent service data was evident. By default a container's writable layer is short-term and its data can be destroyed when the container is killed. Volumes store data outside of the container's existence, removing the potential for this [8].

However, a practical example still relevant to actual cloud systems is the ability to store and retrieve data, so that databases, uploaded files, cache, user generated content and audit logs are not lost each time a container terminates. It then follows that cloud engineering should decouple application containers from data persistence. Although this activity was trivial, the same concept applies to managed block storage, network file systems and the Kubernetes persistent volume claims [6], [8].

This error (in the command) appeared again in the screenshots; being one of the reasons why it is better to explain the commands. Useful information when handling containers as the handling depends on the commands, (small errors or syntax mistakes like the ones shown in Figure 5) can delay troubleshooting, break the automation or make wrong impressions to what regards the image, container or command itself.



The figure consists of two screenshots of a terminal window. The top screenshot shows the creation of a Docker volume named 'keepstore3' and a container named 'persistbox3' that mounts the volume. The bottom screenshot shows the container running and the data being retrieved from the mounted path.

```
pritya@pritya-ubuntu:~/dock-dem$ sudo docker volume create keepstore3
keepstore3
pritya@pritya-ubuntu:~/dock-dem$ sudo docker volume ls
DRIVER    VOLUME NAME
local     keepstore3
pritya@pritya-ubuntu:~/dock-dem$ sudo docker run -d --name persistbox3 -p 8080:80 -v keepstore3:/archive pritya224257/dock-dem:dc1
49228dc6ee6c8a37f5cf8a3231498170398a9f63afeda3800c64f70163ac
pritya@pritya-ubuntu:~/dock-dem$ sudo docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS
49228dc6ee6c   pritya224257/dock-dem:dc1          "/docker-entrypoint..." 21 seconds ago Up 18 seconds
0.0.0.0:8080->80/tcp, [::]:8080->80/tcp persistbox3
pritya@pritya-ubuntu:~/dock-dem$ sudo docker exec persistbox3 sh -c 'echo "Stored through Docker Volume" > /archive/proof.txt && cat /archive/proof.txt'
Stored through Docker Volume
pritya@pritya-ubuntu:~/dock-dem$
```

```
pritya@pritya-ubuntu:~/dock-dem$ sudo docker run -d --name persistbox3 -p 8080:80 -v keepstore3:/archive pritya224257/dock-dem:dc1
7c734c4639804f21d681defff60bd08bd5eebd19e5ef0475db28024c4cdd9
pritya@pritya-ubuntu:~/dock-dem$ sudo docker exec persistbox3 sh -c 'cat /archive/proof.txt'
Stored through Docker Volume
pritya@pritya-ubuntu:~/dock-dem$
```

Figure 5. Creation of a named volume and retrieval of stored data from the mounted path.

3.6 Software-defined networking

Next the Container networking was tested by making a user defined bridge network, listing all networks, attaching a live container to the new network and inspecting the network details. The inspect output displayed the members of the network and their IP addresses. This demonstrated that docker makes an isolated logical network with controlled IP addressing like how cloud platforms control virtual network and subnets and connect to services. [9]

This is an important step as most modern applications are not executed as a single container. Instead, they rely on multiple components (like a database, API, front end, and background worker processes) that require controlled communication. A user-defined bridge network makes that concept observable on a small scale,

because engineers have the ability to choose which containers shall find and communicate with each other. This supports the concept of microservice architecture [6].

The screenshots revealed that there was a command error at the time of attaching the network, but it was fixed immediately. It is important to cite this point since this exemplifies the importance of using the right commands and confirming when testing something. The final inspection validated the information about the attached container and subnet so it was complete.

Starting point however is only local Docker bridge networks. Production clouds would require much more capabilities, namely encrypted traffic, ingress control, network policies, DNS-based service discovery, observability and overlay networking across hosts [6], [9], [11]. Nevertheless, it was very useful as the lab demonstrated that networks in cloud engineering are fully software defined, named and to be monitored, see Figure 6.

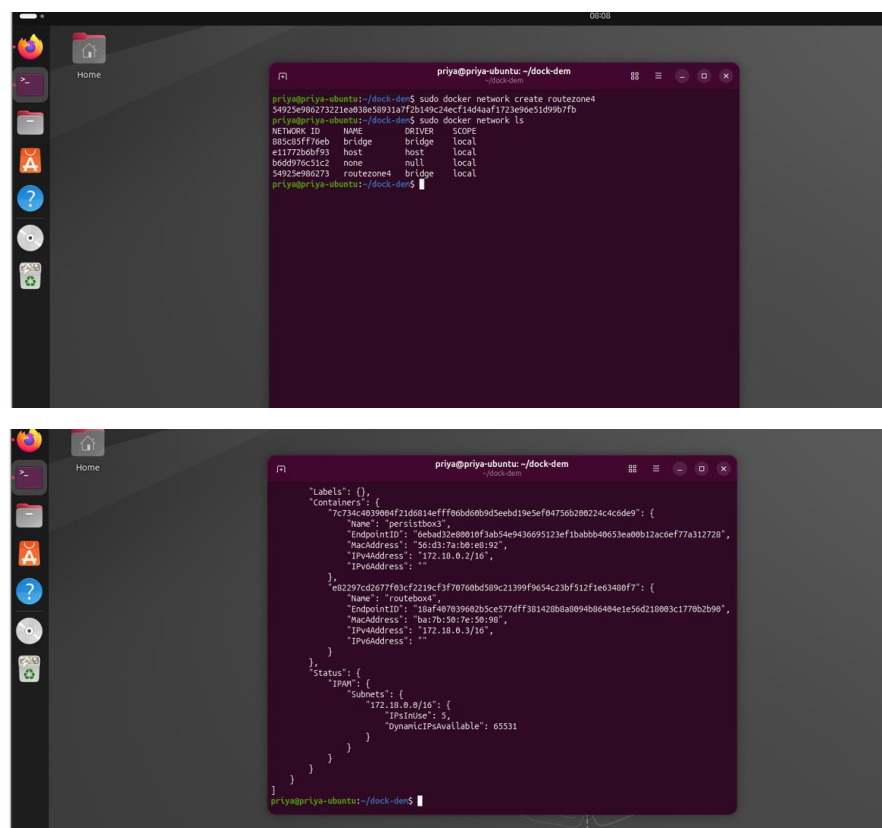


Figure 6. Custom bridge network creation and inspection of attached containers.

3.7 Final verification, cleanup and lessons learned

The last evidence mentioned the list containers, images, volumes and networks and then presented the test container being stopped and removed. This is significant because cloud engineering is not only creating resources; it is cleanup, life cycle controlling and cost limiting. Unused resources will occupy the storage space and increase troubles.

The exercise demonstrated the entire lifecycle of a small containerised service. It included installation, running, local execution, remote published and deployed, persistence and software-defined networking. This made this exercise better than a simple container demonstration.

In a production environment, more enhancements would be necessary which included fixed base-image tags, BuildKit, image vulnerability scanning, secure registry authentication, external configuration, secrets and orchestration for scaling and resilience.

Most important was the validation. Each step was validated- through the container status, image listing, browser output, volume contents and network inspection. All of this follows the cloud practice: build, test, validation, inspection and closing.

- 1) Pin the base image to a specific version or digest instead of using 'latest'.
- 2) Use BuildKit and image scanning to improve build efficiency and security assurance.
- 3) Adopt stronger registry authentication by credential helpers or managed secrets.
- 4) Move toward orchestration and health checking if the service is scaled beyond a single host. The final system state and cleanup can be seen below (see Figure 7).

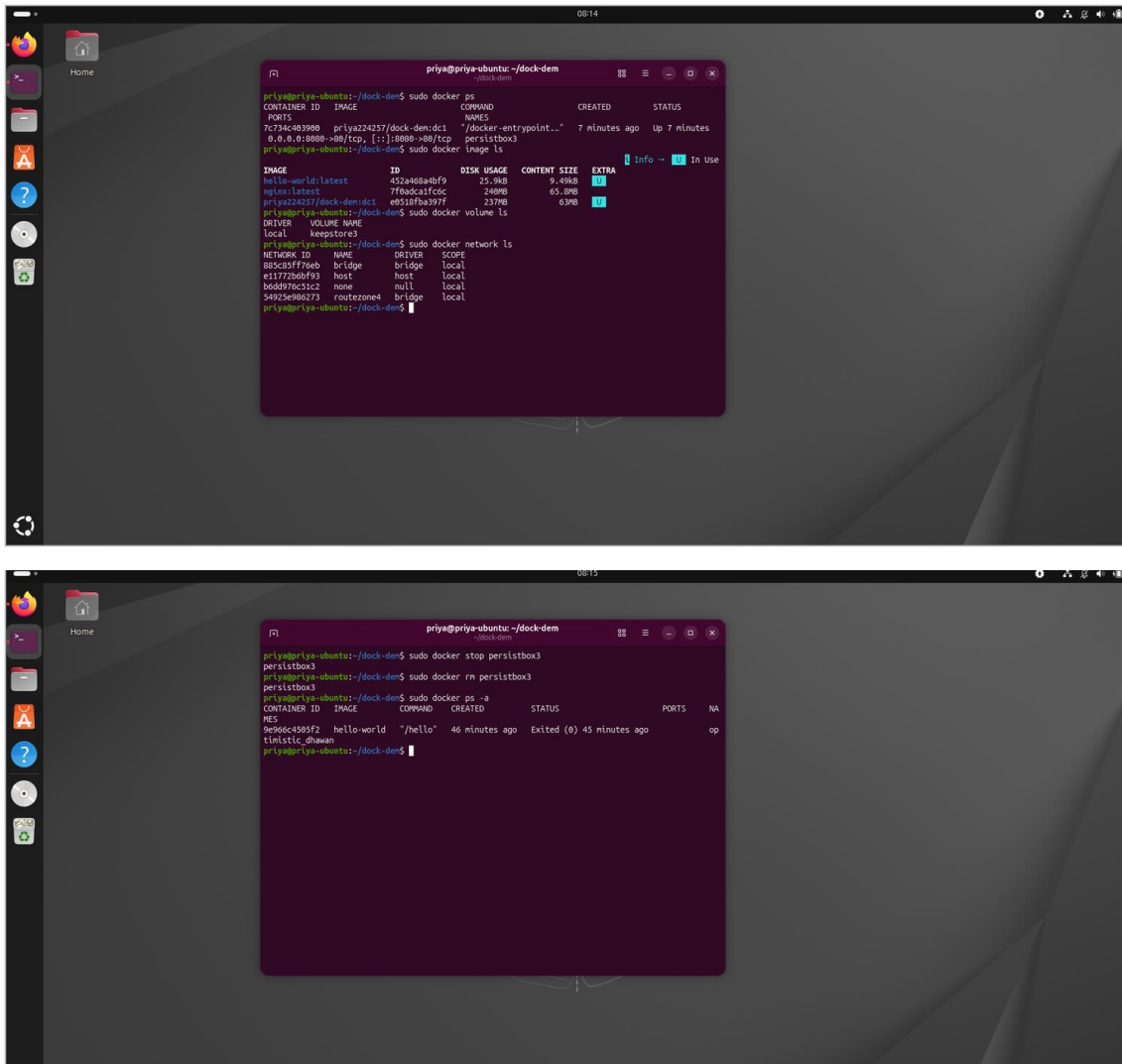


Figure 7. Final system state before and after cleanup of the practical environment.

4. Cloud provider implementation of load balancing, scalability and application gateways

4.1 Chosen application and provider

This section uses one direct example: a customer-facing web application deployed on Microsoft Azure. The web front end and API services are deployed as containers and so the containerised Docker image

becomes the repeatable deployment unit developed in the practical workflow. Azure then offers the managed services to expose, protect and scale that application to an environment that is production-like.

4.2 Load balancing

To balance the load, the Azure Application Gateway is capable of accepting public web traffic and routing the requests based on the listener rules, health probes and routing policies [14]. Behind that layer, there are the Azure Load Balancer and Kubernetes services that distribute traffic across healthy nodes and pods [15], [16]. In the case of a customer facing service, that is to say that the failure of one container or congestion of a single container does not need to disrupt the whole application.

4.3 Scalability

Scalability is concerned with the capacity and not the direction of traffic. Additional pod replicas and additional cluster nodes (when needed) can be added on Azure Kubernetes Service using autoscaling policies like Horizontal Pod Autoscaler and cluster autoscaling policies [16], [17]. This is directly connected to the real-world Docker experience, where scaling can only be reliable when all the new instances are created based on the same tested image.

4.4 Application gateway

The application gateway layer extends the capabilities beyond the mere traffic spreading. Azure Application Gateway is a web-layer-based gateway capable of terminating TLS, routing based on a host or path, and optional Web Application Firewall protection [14]. For the customer, this provides a single secure entry point in spite of the fact that a number of containerised services might be found behind the point.

4.5 Why load balancing and scaling are complementary

Load balancing and scaling complement each other as they address the same issue in different aspects. Load balancing distributes the requests over the already existing capacity, whereas scaling varies the extent of the capacity that is available. When combined, they assist the service to be responsive to spikes in demand rather than just distributing the overload among too few instances.

4.6 Benefits of combining gateway, load balancing and scaling

The best outcome for a customer service comes from combining gateway, load balancing and scaling together. A secure and policy-driven front door is provided by the gateway, load balancing directs traffic to healthy targets and scaling adds and removes replicas as the demand varies. This combination improves both system resilience and customer experience.

1. Better availability, as failed instances can be avoided and healthy replicas are still providing service to users.
2. Traffic distribution and replica growth work together to improve performance in changing demand.
3. Better security and control, as the gateway concentrates the HTTPS processing, routing policy and web protection.

5. Serverless computing versus containerisation

Offering the benefits of abstraction, both serverless and containerised computing deal with the same needs and address them on different abstraction levels. Serverless computing has providers taking care of most of the infrastructure needed for the application to run, while code is executed by means of operations on events or requests. Typical implementations are AWS Lambda or Azure Functions [12], [13], where developers are only concerned with functions, triggers and authorizations rather than hosts or long lived servers. Serverless has three major advantages over the Docker workflow we have this report; the first being reduced ops work; faster scale-up/down, including auto-scale-to-zero. It offers potential cost savings for bursty workloads because cost is linked more closely to execution time and allocated resources rather than always-on server capacity.

However containers work better for many use cases. The Docker task in this report used a continuously running Nginx service on a fixed port which integrates very naturally with container deployment. Containers offer stronger packaging control since the same image could work locally, in CI, on-premise or in the cloud.

So, it is an architectural, not an ideological choice. Serverless provides value when the main aim is to reduce infrastructure management and match cost more closely with actual usage. Containers are a more compelling option when the priority is consistent packaging, runtime control and portability. Most organizations balance both under the hood.

However, not all key operational problems are eliminated by serverless. Observability, dependency security, scheduling and tuning for concurrency continue to be relevant [12], [13]. Likewise, containers are not always completely portable when heavily dependent on provider-specific services. An experienced cloud engineer chooses the suitable execution model according to the workload.

Design question	Containers	Serverless
Runtime control	High control over image contents, process model and dependencies	Lower control because the provider manages most runtime details
Scaling model	Replica-based scaling, usually with orchestration	Event-driven scaling, often automatic and granular
Best-fit workloads	APIs, microservices, workers, long-running services	Event handlers, bursty jobs, lightweight integrations
Main caution	Operational overhead and security hardening still required	Cold starts, limits and provider lock-in concerns

Table 3. Workload-oriented comparison of containerisation and serverless execution

6. Driving forces behind cloud expansion

There are several broader reasons why cloud platforms and container-based delivery have grown so quickly. One of these is cost efficiency, since shared infrastructure and automated processes will decrease idle capacity and human effort [2]. Another is elasticity because organisations are more in demand to manage the changing demand without a protracted procurement process. Delivery speed is also important. The competition encourages the team to make quick deliveries, do quick testing in case of errors. Containerization helps here because it allows creating repeatable application artefacts.

It is also driven by standardization. Being able to claim that this particular image went through testing and approval is something that really works in the real life of distributed teams. It improves the collaboration between developers, testers, operations and security engineers. Along with registries, scanners, CI systems, and orchestration tools, containerization will be a part of an even broader concept – software supply chain.

On the other hand, technologies alone are not driving the growth of clouds. They are shaped by certain governance needs, resilience and business continuity requirements. Companies need disaster recovery options, geographic elasticity, simplified experiments, and reduced dependence on the tricky manual configuration. This set of needs was replicated at small scale in this exercise. The image has been built using reproducible methods, published into some distant registry, ran consistently and cleaned up responsibly. Exactly these kinds of things become the cornerstone of a professional approach to the cloud development.

This means that the most important conclusion from this exercise is not the fact that Docker commands worked. It's rather the cloud engineering process of packaging, verification and lifecycle management and its focus on operations.

7. Conclusion

From the above report, it is evident that practical implementation utilizing Docker can be used to illustrate many broader concepts of cloud engineering. As part of the assignment completed above, one was able to see the process involved in installing Docker, testing the execution of the image, building an image via a Docker file, running an application by using the port, pushing the image to a remote repository (Docker Hub), running a container with a mounted Docker volume, creating a connection between containers through a custom bridge network, and concluding the whole process with a final test and cleanup.

The practical work was put in the larger context of cloud computing in the theoretical discussion. Virtual machines are not yet obsolete, but containerisation is more portable, and has better deployment reliability with most cloud-native applications. Registries, volumes and networks can be used to create a small cloud engineering setup using a basic container demo. This was further demonstrated in the comparison with serverless computing, which demonstrated that modern cloud architecture needs to be judged, as opposed to a one-size-fits-all tool selection.

The most powerful conclusion is thus that effective cloud engineering is reliant on execution and assessment. The practical work must be done but it must be just as well to learn its architectural significance, security considerations and limitations of production. The evidence presented in this report indicates that the practical objectives were met whereas the analysis demonstrates the relationships between the practical objectives and the higher level cloud design decisions. It is that combination that transforms a lab exercise into a cloud engineering report that is professionally relevant.

References

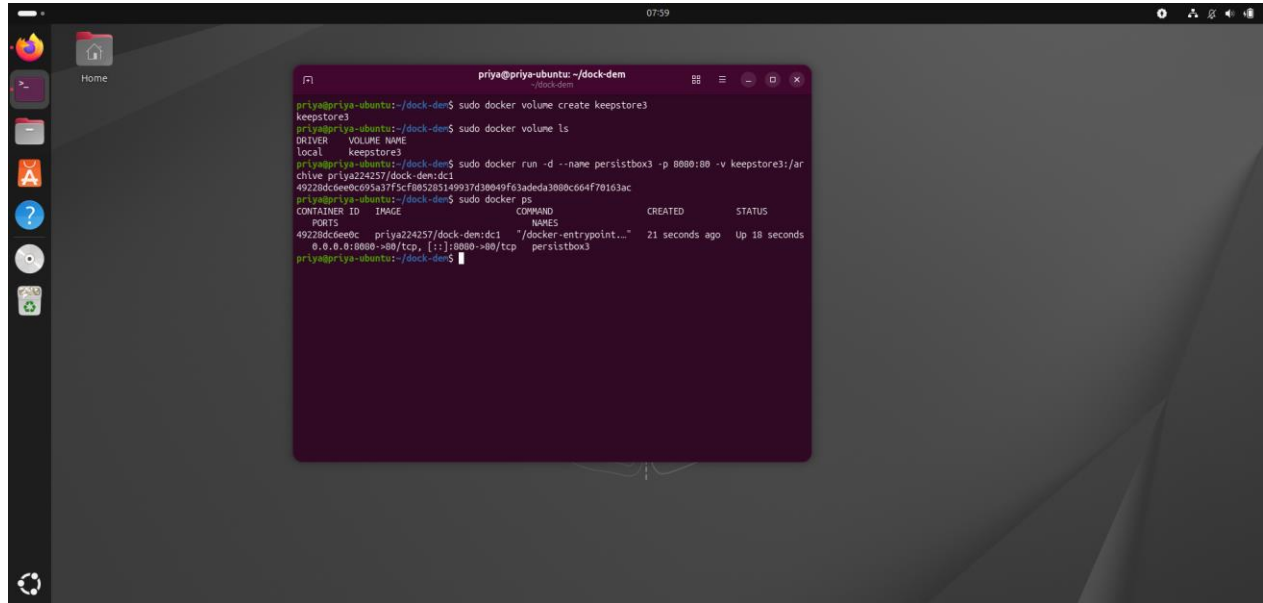
- [1] P. Mell and T. Grance, “The NIST Definition of Cloud Computing,” NIST Special Publication 800-145, 2011. [Online]. Available: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf>. Accessed: April 11, 2026.
- [2] M. Armbrust et al., “A View of Cloud Computing,” *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010. [Online]. Available: <https://dl.acm.org/doi/10.1145/1721659.1721672>. Accessed: April 11, 2026.
- [3] D. Merkel, “Docker: Lightweight Linux Containers for Consistent Development and Deployment,” *Linux Journal*, no. 239, 2014. [Online]. Available: <https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment>. Accessed: April 11, 2026.
- [4] R. Morabito, J. Kjällman, and M. Komu, “Hypervisors vs. Lightweight Virtualization: A Performance Comparison,” in *IEEE International Conference on Cloud Engineering*, 2015. [Online]. Available: <https://ieeexplore.ieee.org/document/7123014>. Accessed: April 11, 2026.
- [5] C. Pahl, “Containerization and the PaaS Cloud,” *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015. [Online]. Available: <https://ieeexplore.ieee.org/document/7131390>. Accessed: April 11, 2026.
- [6] B. Burns, J. Beda, and K. Hightower, *Kubernetes: Up and Running*, 3rd ed. Sebastopol, CA: O’Reilly Media, 2022.
- [7] Docker Inc., “Docker overview,” Docker Engine documentation, 2025. [Online]. Available: <https://docs.docker.com/get-started/docker-overview/>. Accessed: April 11, 2026.
- [8] Docker Inc., “Persisting container data with Docker volumes,” Docker documentation, 2025. [Online]. Available: <https://docs.docker.com/engine/storage/volumes/>. Accessed: April 11, 2026.
- [9] Docker Inc., “Networking overview,” Docker documentation, 2025. [Online]. Available: <https://docs.docker.com/engine/network/>. Accessed: April 11, 2026.
- [10] Docker Inc., “docker build, tag and push,” Docker documentation, 2025. [Online]. Available: <https://docs.docker.com/get-started/docker-concepts/building-images/build-tag-and-publish-an-image/>. Accessed: April 11, 2026.
- [11] NIST, “Application Container Security Guide,” NIST Special Publication 800-190, 2017. [Online]. Available: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-190.pdf>. Accessed: April 11, 2026.
- [12] Amazon Web Services, “AWS Lambda Developer Guide,” 2025. [Online]. Available: <https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>. Accessed: April 11, 2026.
- [13] Microsoft, “Azure Functions documentation,” 2025. [Online]. Available: <https://learn.microsoft.com/en-us/azure/azure-functions/>. Accessed: April 11, 2026.
- [14] Microsoft, “Azure Application Gateway documentation,” 2025. [Online]. Available: <https://learn.microsoft.com/en-us/azure/application-gateway/>. Accessed: April 11, 2026.
- [15] Microsoft, “Azure Load Balancer documentation,” 2025. [Online]. Available: <https://learn.microsoft.com/en-us/azure/load-balancer/>. Accessed: April 11, 2026.

- [16] Microsoft, "Azure Kubernetes Service (AKS) documentation," 2025. [Online]. Available: <https://learn.microsoft.com/en-us/azure/aks/>. Accessed: April 11, 2026.
- [17] Microsoft, "Scale applications in Azure Kubernetes Service (AKS)," 2025. [Online]. Available: <https://learn.microsoft.com/en-us/azure/aks/tutorial-kubernetes-scale>. Accessed: April 11, 2026.

Appendix A

These screenshots are included as supporting appendix evidence for the Docker practical workflow. They are kept separate from the main discussion to avoid interrupting the report flow while still preserving the full practical record.

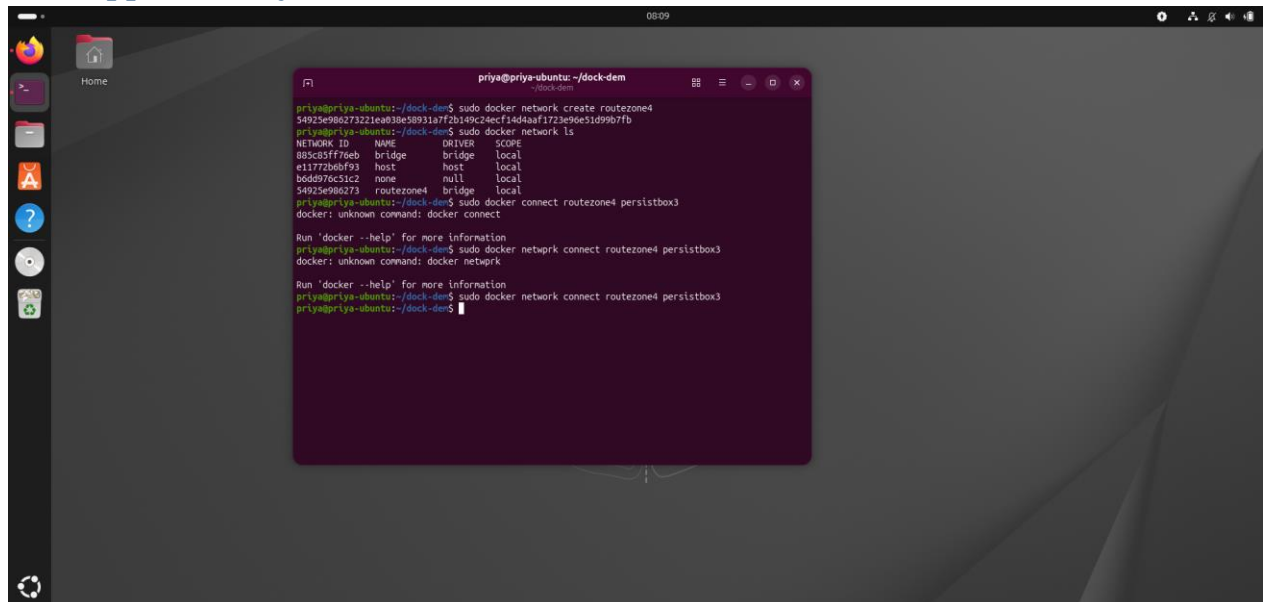
A.1 Supplementary screenshot



```
priya@priya-ubuntu: ~/dock-dem
priya@priya-ubuntu:~/dock-dem$ sudo docker volume create keepstore3
keepstore3
priya@priya-ubuntu:~/dock-dem$ sudo docker volume ls
DRIVER      VOLUME_NAME
local       keepstore3
priya@priya-ubuntu:~/dock-dem$ sudo docker run -d --name persistbox3 -p 8080:80 -v keepstore3:/ar
chive priya224257/dock-dem:dc1
49228dc6ee0c95a3775cf085283149937d38049f63aded3880c64f70163ac
priya@priya-ubuntu:~/dock-dem$ sudo docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS
49228dc6ee0c   priya224257/dock-dem:dc1   "/docker-entrypoint..."   21 seconds ago   Up 18 seconds
0.0.0.0:8080->80/tcp, [::]:8080->80/tcp   persistbox3
priya@priya-ubuntu:~/dock-dem$
```

Appendix Figure A1. Named Docker volume 'keepstore3' created, listed, and attached to the running container 'persistbox3', with the container state verified through docker ps.

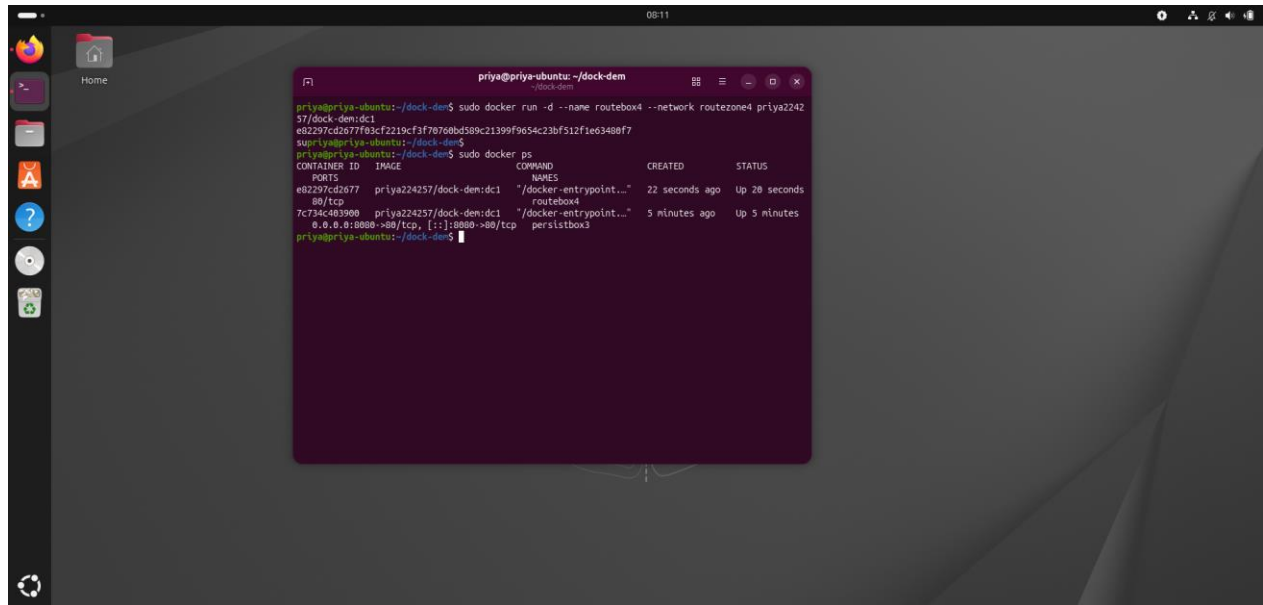
A.2 Supplementary screenshot



```
priya@priya-ubuntu: ~/dock-dem
priya@priya-ubuntu:~/dock-dem$ sudo docker network create routezone4
54925e986273221eab30e5931a728149c24ecf1444aef173299e51d99b71b
priya@priya-ubuntu:~/dock-dem$ sudo docker network ls
NETWORK ID     NAME      DRIVER  SCOPE
885c85ff79eb   bridge    bridge  local
e11772dbd693   host      host    local
b6d6976c51c2   none      null     local
54925e986273   routezone4 bridge    local
priya@priya-ubuntu:~/dock-dem$ sudo docker connect routezone4 persistbox3
docker: unknown command: docker connect
Run 'docker --help' for more information
priya@priya-ubuntu:~/dock-dem$ sudo docker netwrk connect routezone4 persistbox3
docker: unknown command: docker netwrk
Run 'docker --help' for more information
priya@priya-ubuntu:~/dock-dem$ sudo docker network connect routezone4 persistbox3
priya@priya-ubuntu:~/dock-dem$
```

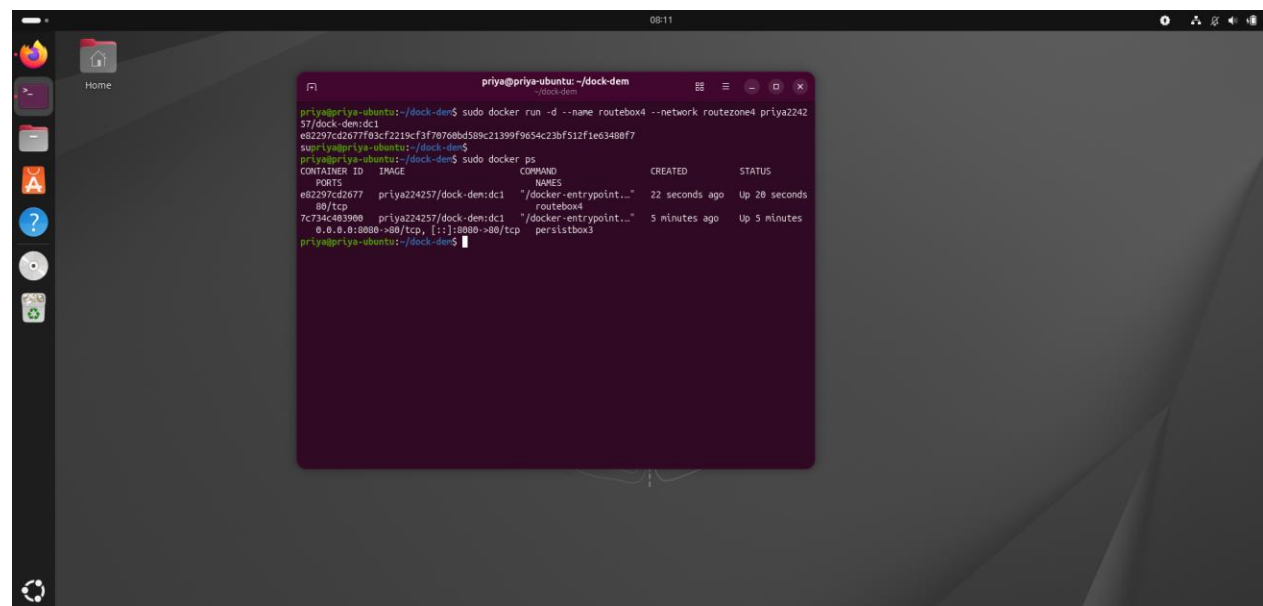
Appendix Figure A2. Intermediate networking step showing command correction during connection of the 'persistbox3' container to the custom bridge network 'routezone4'.

A.3 Supplementary screenshot



Appendix Figure A3. Additional container 'routebox4' launched directly on the custom network 'routezone4', followed by container-state verification using `docker ps`.

A.4 Supplementary screenshot (see Appendix Figure A4)



Appendix Figure A4. Follow-up verification capture confirming that both 'routebox4' and 'persistbox3' remained active after the network-based deployment step.