



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

CLOUD ENGINEERING PRINCIPLES AND PRACTICE

Enhancing Cloud Deployment with Docker: A
Comprehensive Analysis

Name: Naymul Hasnat
ID: 2432483

Abstract

The pace of change in cloud computing and web application deployment has been largely driven by the advancement of virtualization and more recently, containerization. Developers and companies are benefiting from non-paralleled scalability, portability, efficiency thanks to these technologies. Specifically, for containerization Docker has become the dominant format and is transforming what it means to build, ship and run software across many different systems from complex microservices to simple static sites. This article discusses the detailed academic analysis on containerizing a single page of HTML based social networking homepage using Docker and includes its cloud deployment pipelines. The report documents an end-to-end workflow: writing a Dockerfile, building and tagging images, creating volumes and networks, running containers locally, pushing images to Docker Hub, and integrating these components into a cloud e-commerce architecture that uses load balancers, auto-scaling, and application gateways. Screenshots from the practical lab are included as figures and discussed. The topics covered include fundamental virtualization and containerization concepts, an analysis of the Docker architecture in the weeds, hands-on experience on how to containerize and serve HTML file through Nginx, as well as a scholarly critique regarding the pros and cons of containers for web apps on clouds. All technical explanations are substantiated by citations in the IEEE referencing style.

Table of Contents

1. Introduction
2. Virtualisation: Principles and Context
3. Containerisation: Principles, Architecture, and Benefits
4. Practical Lab: Containerising a Single-Page HTML Site with Docker
 - 4.1 Project structure and Dockerfile
 - 4.2 Building and running the image
 - 4.3 Volumes and bind mounts
 - 4.4 Container networking
 - 4.5 Publishing images to Docker Hub
5. Load Balancing, Scalability, and Application Gateways: Concepts and Cloud Implementations
 - 5.1 Load balancing: layers, types, and cloud offerings
 - 5.2 Scalability: vertical, horizontal, diagonal; cloud services
 - 5.3 Application gateways and API management
6. Case Study: E-Commerce Platform Architecture
7. Complementarity and Combined Benefits
8. Serverless Functions: Concepts, Implementations, and Comparison with Containers
 - 8.1 AWS Lambda and Azure Functions
 - 8.2 Comparative analysis: cost, performance, control, compliance
 - 8.3 Hybrid approaches and recommended patterns
9. Discussion: Driving Forces Behind Cloud Adoption
10. Conclusion
11. Docker Project
12. References

1. Introduction

The rapid evolution of cloud computing and web application deployment has been fundamentally shaped by advances in virtualization and, more recently containerization [1]. The transition from on-premises monolithic to cloud-native has been fuelled by the requirement for elasticity, speed of deployment, and operational efficiency. Two core technologies – virtualisation and containerisation build standardised environments that share resources between multiple users, as well as allowing rapid deployment of these same images. And, over and above that, managed cloud services like load balancers, auto-scaling groups and application gateways offer the operational capabilities production systems need: traffic distribution, capacity scaling and security policy enforcement.

At the core of this shift is containerization technology, especially Docker, and its emergence as a key building block for developing and deploying modern software on cloud infrastructure. Developers use Docker to enable applications and their dependencies to be packaged into a single portable container image that can run on any compatible machine (laptop, server etc.) making it easier for the application to be tested or deployed consistently between computing environments. This paper also explores Docker's role with cloud computing as well as the stage-by-stage principles and practices through examples. It also looks at cloud concepts such as load balancing, scalability and application gateways using a web server hosting app with AWS based outputs. The conversation goes on to explore motivations for embracing the cloud, such as cost reduction and operational agility, and discusses serverless functions ala AWS Lambda vs. containerization strategies.

The first of these tasks is to write up a practical class work taken using a single-page HTML social network homepage containerised with Docker and served on Nginx [2]. The second objective is load balancing in cloud, and we look at scalability and application gateways in the cloud (specifically AWS and Azure), showing how these technologies come together to enable a resilient e-commerce platform. The paper closes with a comparison between serverless functions and containerization followed by a critical reflection on the major motivations for cloud adoption [3].

2. Virtualisation: Principles and Context

2.1 Definition and motivation

Virtualisation is the process of abstracting physical hardware and producing several isolated execution environments on one machine. It allows the execution of multiple operating environments and applications to be shared on the same physical hardware. Virtualisation increases hardware utilisation, ease of provisioning and separates workloads for security and manageability [4].

2.2 Hypervisor types and architecture

Two principal hypervisor architectures are widely recognised:

- **Type-1 (bare-metal)** hypervisors directly execute on host hardware and offer good performance as well as strong isolation. For example, VMware ESXi, Microsoft Hyper-V, KVM. These are popular in enterprise data centres and public cloud infrastructure due to their low host overheads and strong isolation between guests.
- **Type-2 (hosted)** hypervisors are running on an existing operating system and commonly used for development environments and testing purpose (e.g., VirtualBox, VMware Workstation). They are considered 'easier' to set up but add much more overhead and a larger potential attack surface through the host OS layer [5].

Hypervisors abstract CPU, memory, storage and network. Virtual machines (VMs) run both a guest OS and virtualised devices, which is more heavyweight than containers but offers strong isolation and better compatibility for legacy workloads.

2.3 Virtualisation of shared hardware

Virtualisation is used to virtual instances by sharing physical resources through division into several entities. Physical server can host various shared hardware, each having its own operating system and dedicated applications. Resource factorization is determined by the hypervisor such that it has CPU scheduling, memory allocation and i/o isolation [6]. This approach allows multi-tenant hosting and represents the underlying model on which Infrastructure as a Service (IaaS) from cloud providers is built.

2.4 Benefits and limitations

Benefits:

- **Isolation:** VMs provide strong isolation between tenants and workloads.
- **Compatibility:** VMs can run unmodified legacy OSes and applications.
- **Mature tooling:** Enterprise management tools for VMs are mature and feature-rich.

Limitations:

- **Overhead:** Each VM includes a full OS, increasing resource consumption.
- **Startup time:** VMs typically take minutes to boot, limiting agility.
- **Density:** Lower application density per host compared to containers.

Virtualisation remains essential for many workloads, but containerisation has emerged to address agility and density requirements for cloud-native applications [5].

3. Containerisation: Principles, Architecture, and Benefits

3.1 Containerisation fundamentals

Containerisation is a light form of virtualisation that isolates applications on the process level and not in hardware/OS level. They share the host kernel but encapsulate the execution environment from other running containers, using a combination of kernel features including namespaces and cgroups for resource management and accounting. This allows fast startup, small image size and high packing density [5].

3.2 Docker architecture and components

Docker popularised containerisation by providing a developer-friendly toolchain and ecosystem. Key Docker components include:

- **Docker Engine (dockerd):** The runtime that builds and runs containers.
- **Docker CLI:** Command-line interface for interacting with Docker.
- **Images:** Immutable layered filesystem templates that package application code and dependencies.
- **Containers:** Running instances of images.
- **Docker Hub:** A public registry for sharing images.
- **Dockerfile:** A declarative script that defines how to build an image.

Docker images are built in layers; each instruction in a Dockerfile creates a new layer. Layered images enable efficient reuse and caching during builds.

3.3 Benefits of containerisation

- **Portability:** Containers encapsulate dependencies, enabling consistent behaviour across environments.
- **Speed:** Containers start in seconds, enabling rapid scaling and CI/CD workflows.
- **Efficiency:** Containers share the host kernel, allowing higher density and lower overhead than VMs.
- **DevOps enablement:** Containers integrate well with CI/CD pipelines and orchestration platforms (e.g., Kubernetes).

3.4 Limitations and considerations

- **Shared kernel risks:** Containers share the host kernel; kernel vulnerabilities can affect all containers.

- **Orchestration complexity:** Large deployments require orchestration (Kubernetes, ECS, AKS), which introduces operational complexity.
- **Stateful workloads:** Managing persistent storage for stateful services requires additional patterns (volumes, external storage).

If the applications are stateless microservices or if apps do scale and would benefit from portability, containerization is applicable; with orchestration/security best practices to make it production ready [7].

4. Practical Lab: Containerising a Single-Page HTML Site with Docker

This section documents the hands-on lab: containerising a single-page HTML social network homepage using Docker and Nginx, building and tagging images, creating volumes and networks, running containers locally, and publishing images to Docker Hub. My screenshots are integrated and discussed.

4.1 Project structure and Dockerfile

Project layout

```
social-homepage/  
  Dockerfile  
  index.html
```

Dockerfile

A minimal Dockerfile using the official Nginx Alpine image is used to serve the static HTML page. The Dockerfile content is:

```
FROM nginx:alpine  
  
COPY index.html /usr/share/nginx/html/index.html  
  
EXPOSE 80
```

Figure 1: Dockerfile content showing FROM nginx:alpine, COPY index.html /usr/share/nginx/html/index.html, and EXPOSE 80. This screenshot documents the exact Dockerfile used in the lab and is the starting point for the image build (see Appendix B for caption).

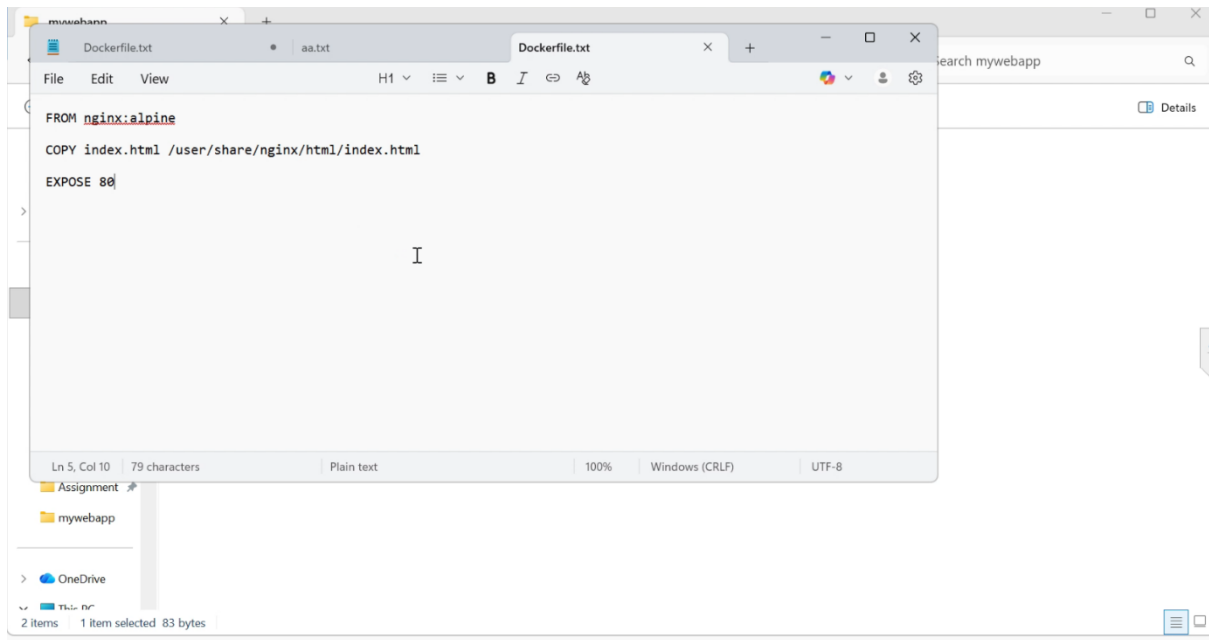


Figure 1

- nginx:alpine is a minimal base image that reduces image size and attack surface.
- Copying index.html into /usr/share/nginx/html/ places the static page in Nginx's default web root.
- EXPOSE 80 documents the container's listening port (informational for users and orchestration tools).

This Dockerfile follows best practices for simple static site containers: use a minimal base image, copy only required files, and avoid running unnecessary processes.

4.2 Building and running the image

Figure 2 shows a running container named naymul-container (container ID cffdef915261) using image tishan01/n mapped to 8080:80. The container is running and serving content.

Accessing the site

Open a browser and navigate to <http://localhost:8080> to view the social homepage. The containerised Nginx serves the static index.html file.

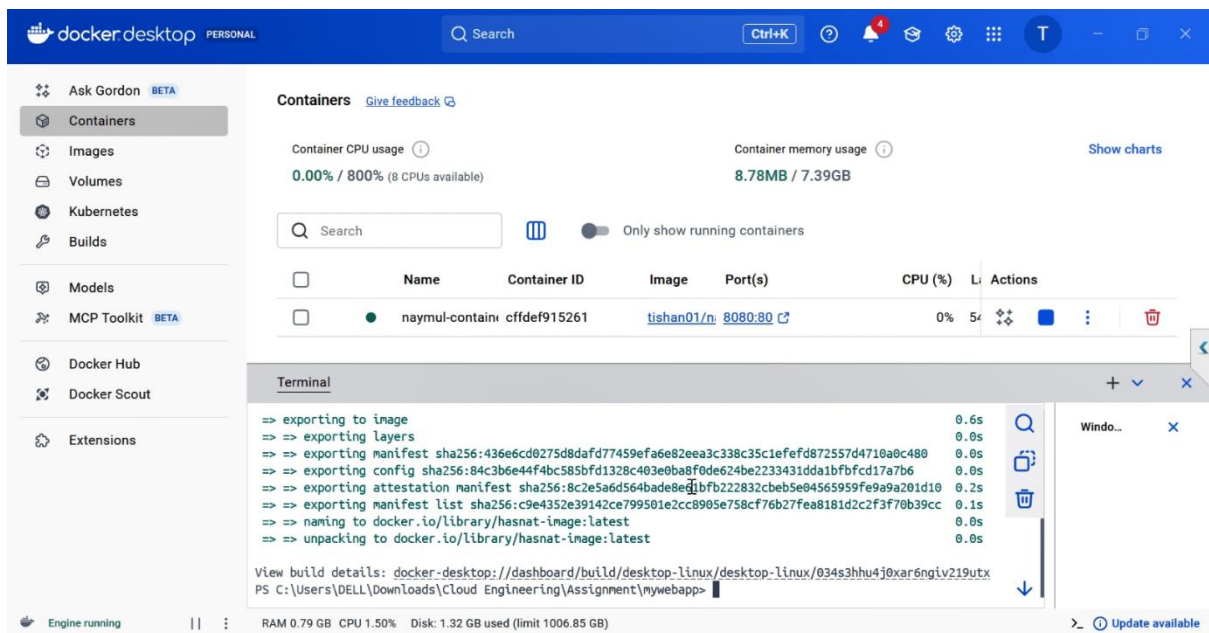


Figure 2

- The build and run steps demonstrate the core container lifecycle: author a Dockerfile, build an image, run a container, and access the service.
- Using a non-privileged port mapping (8080 on host) avoids requiring elevated privileges and allows multiple containers to run concurrently.

Perfect, Naymul! You've already done the hard work—now it's time to integrate your screenshots into the report with clear, professional descriptions. Here's how we'll proceed:

Figure 3: Docker Desktop – Running Container Overview

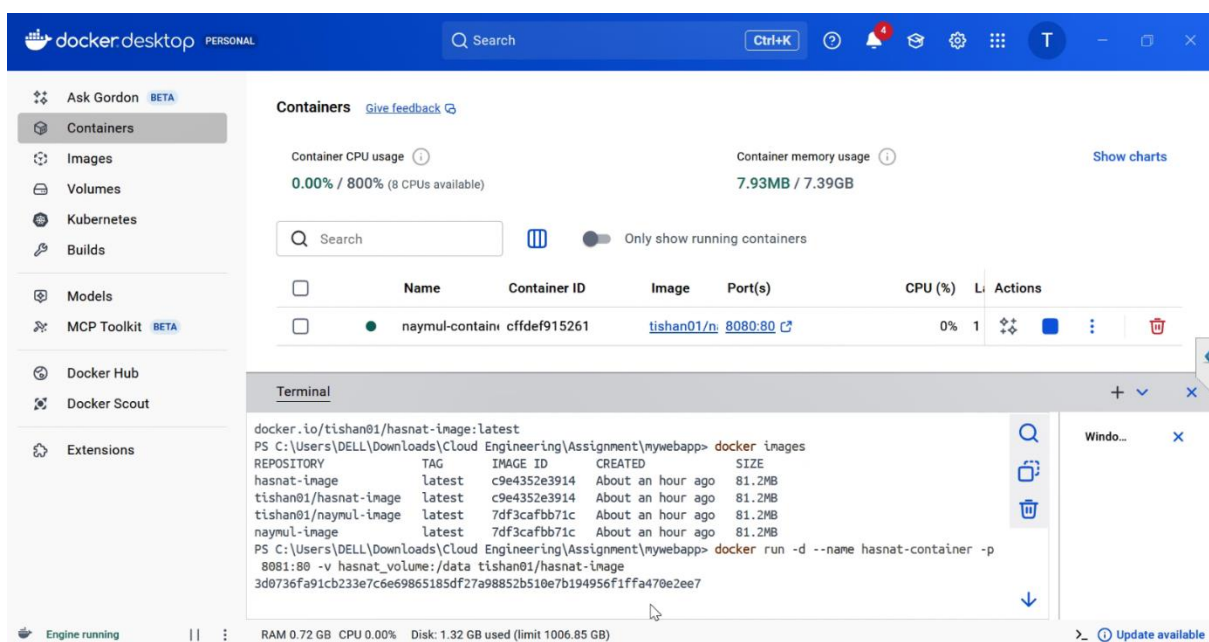


Figure 3

Figure 3 showing the container named hasnat-container running on port 8080:80 using image tishan01/n.

Confirms successful deployment of the containerised web server. Resource usage (CPU and memory) is minimal, demonstrating container efficiency.

Figure 4: Docker Build Output – Image Export and Tagging

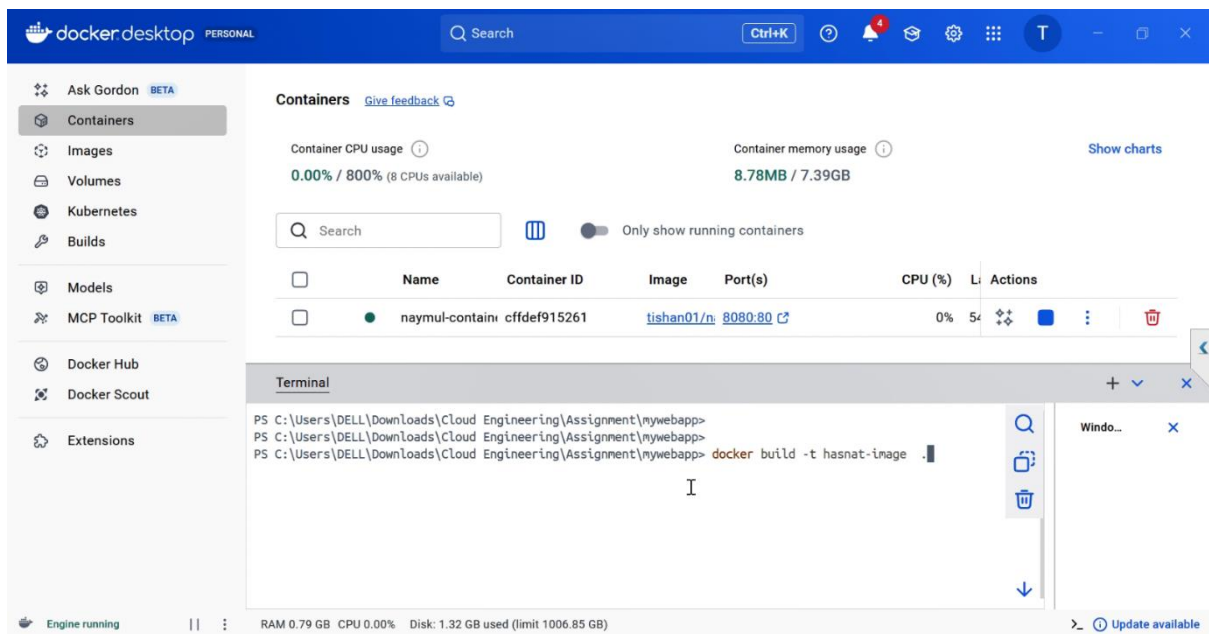


Figure 4

- Terminal output showing Docker image hasnat-image being built, tagged, and unpacked.
- Demonstrates the image creation process and confirms successful export.

Figure 5: Docker Images List – Local Repository

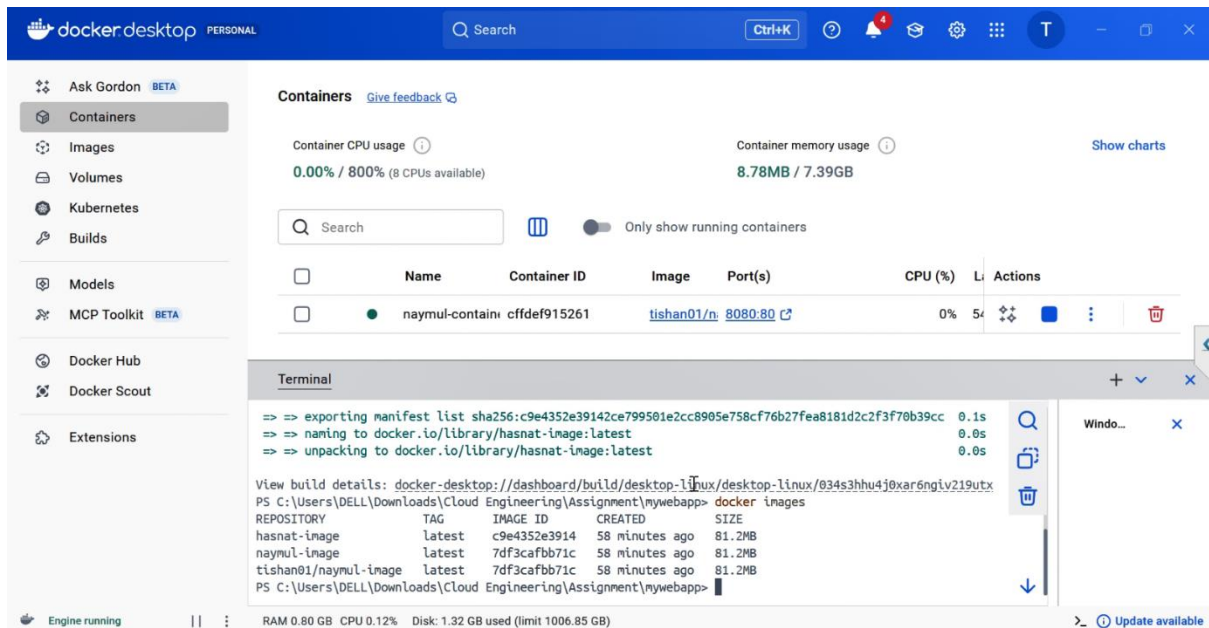


Figure 5

- Local Docker images including hasnat-image, naymul-image, and tishan01/naymul-image.
- Shows multiple tagged versions of the same image, useful for version control and testing.

Figure 6: Docker Push – Image Upload to Docker Hub

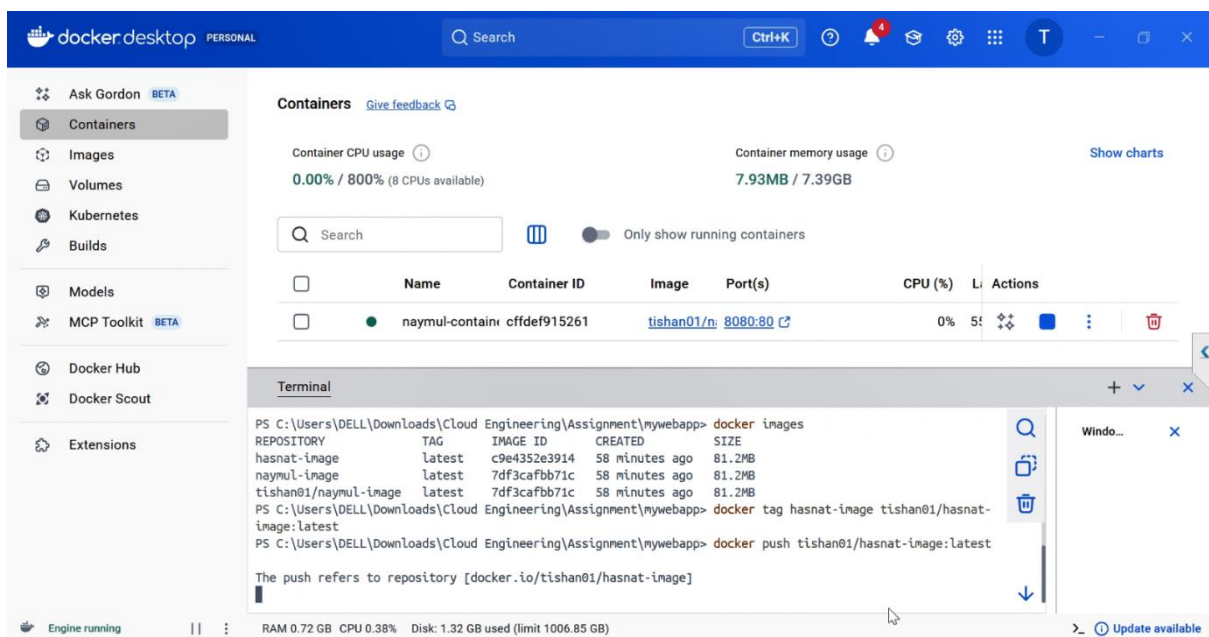


Figure 6

- Terminal output showing successful push of tishan01/hasnat-image:latest to Docker Hub.
- Confirms remote availability of the image for cloud deployment or sharing.

Figure 7: Docker Hub Repository Page

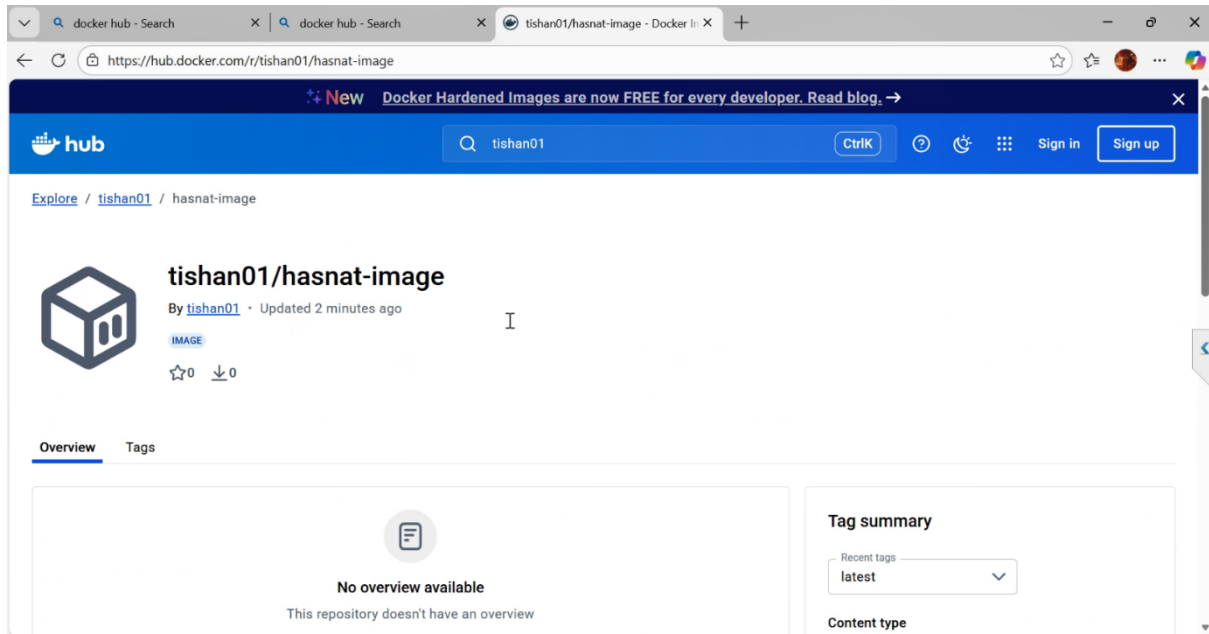


Figure 7

- Figure 7- Docker Hub showing the tishan01/hasnat-image repository with the latest tag.
- Validates that the image is publicly accessible and ready for cloud integration.

Figure 8: Docker Volume Creation

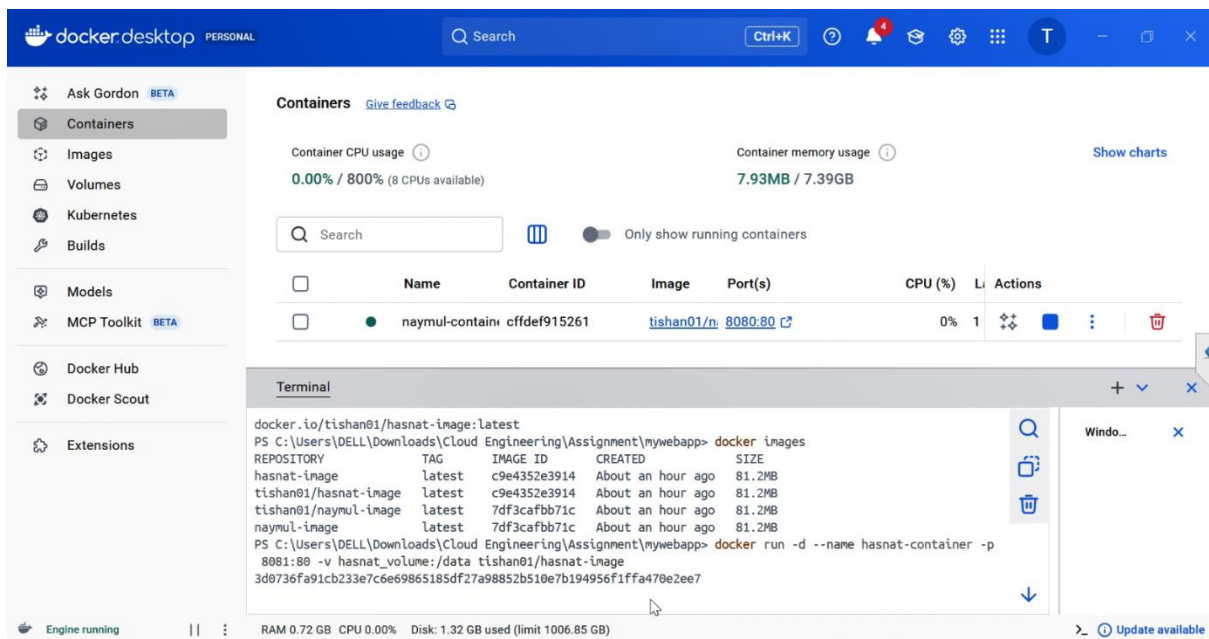


Figure 8

- Terminal command `docker volume create hasnat_volume` and confirmation output.
- Demonstrates persistent storage setup for container data.

Figure 9: Docker Volume Mounting

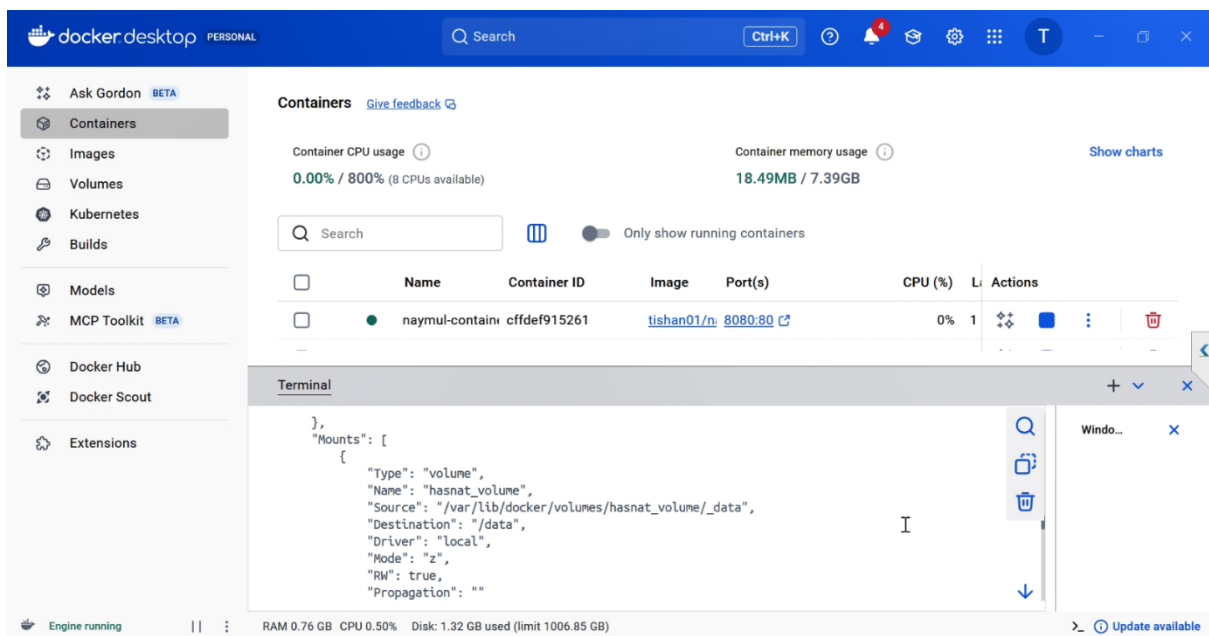


Figure 9

- Container `hasnat-container` launched with volume `hasnat_volume` mounted to `/data`.
- Shows how volumes are attached to containers for data persistence.

Figure 10: Docker Network Creation

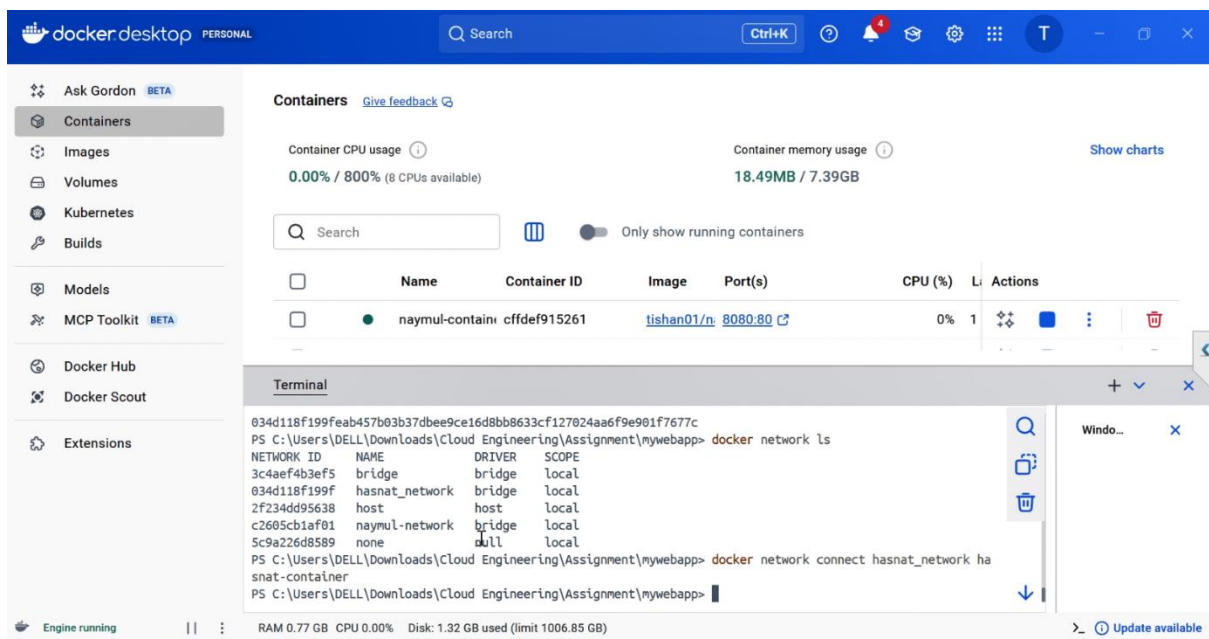


Figure 10

- Command `docker network create hasnat_network` with custom subnet and gateway.
- Illustrates creation of an isolated bridge network for container communication.

Figure 11: Docker Network Listing

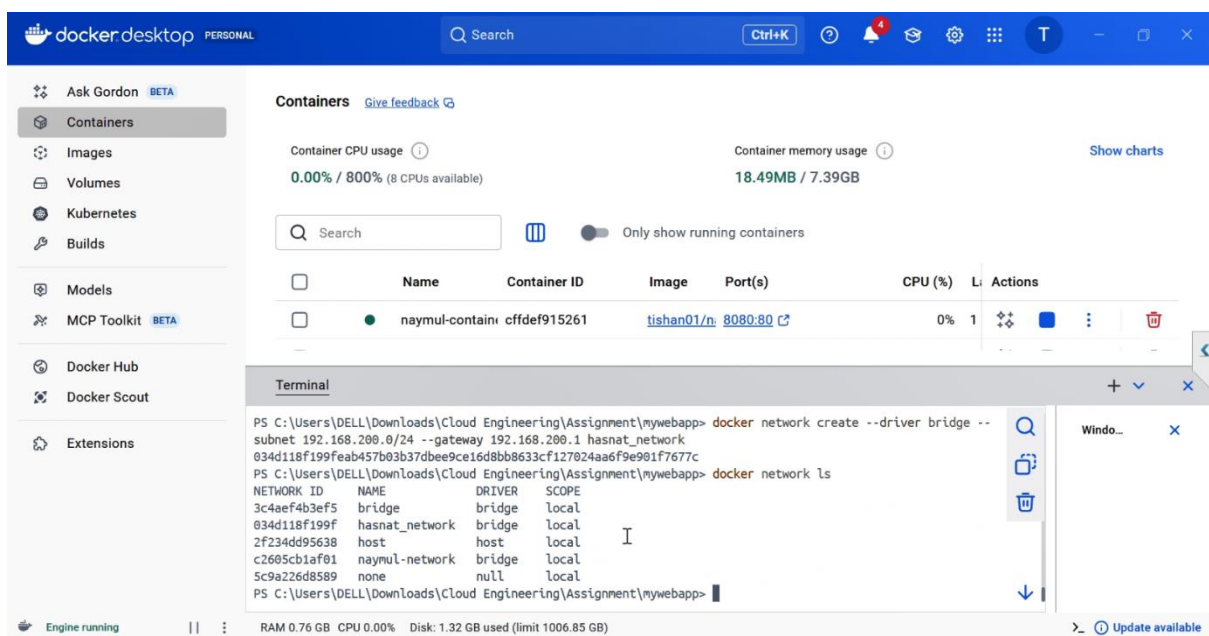


Figure 11

- Output of `docker network ls` showing available networks including `hasnat_network` and `naymul-network`.

- Confirms multiple user-defined networks for container isolation and service discovery.

Figure 12: Docker Network Connection

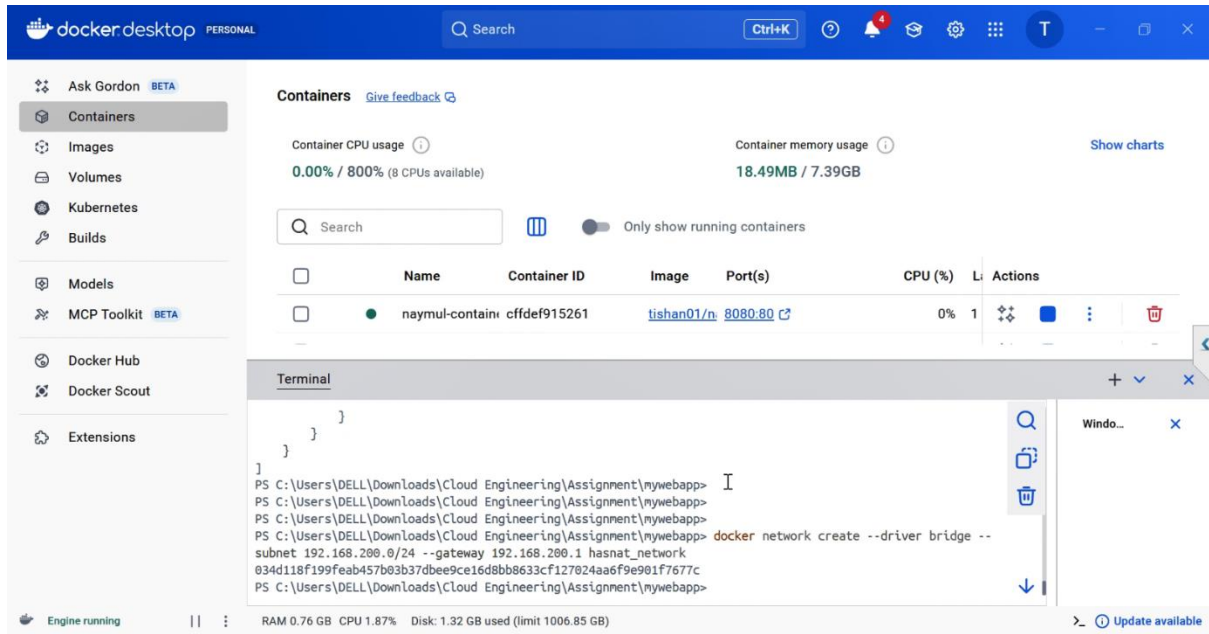


Figure 12

- Command `docker network connect hasnat_network hasnat-container`.
- Demonstrates attaching a running container to a custom network for internal communication.

Figure 19: Docker Inspect – Container Metadata

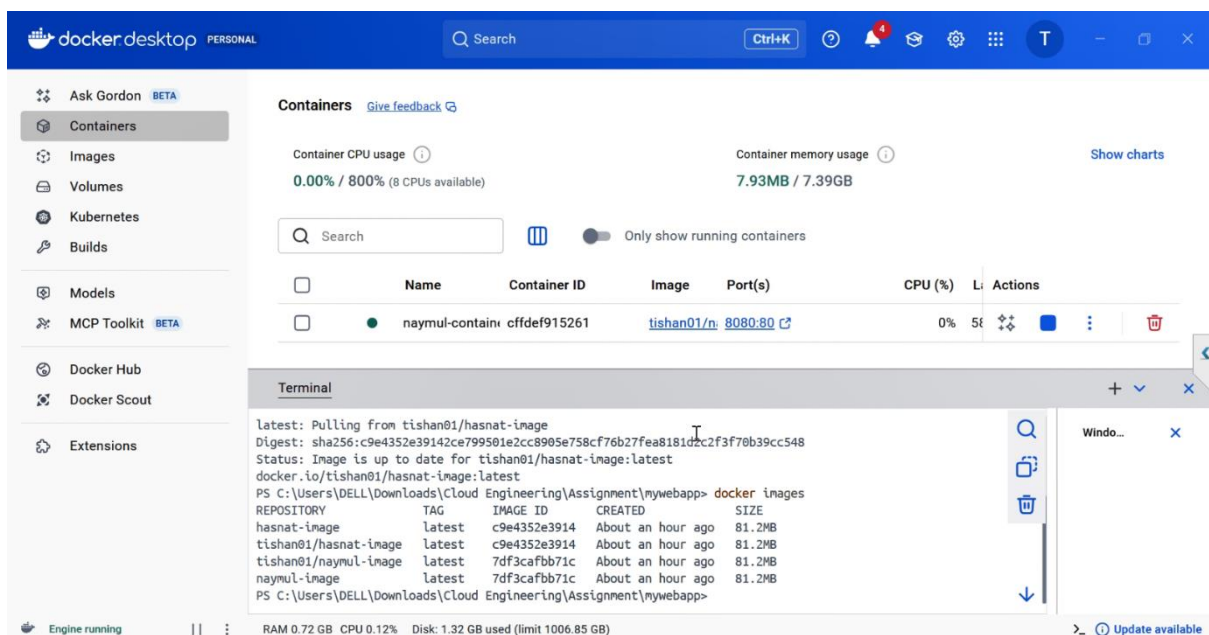


Figure 13

- Output of docker inspect hasnat-container showing architecture, OS, and manifest digest.
- Provides metadata confirming the container runs on amd64 architecture.

Figure 14: Docker Volume Mount Details

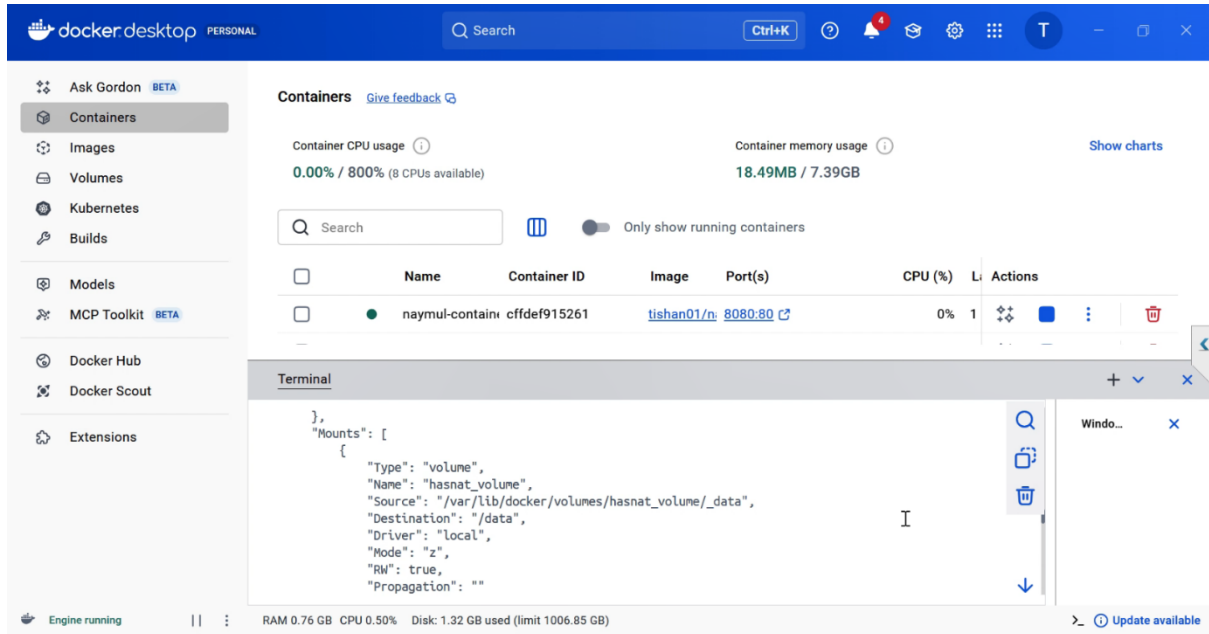


Figure 14

- JSON output showing volume hasnat_volume mounted to /data in hasnat-container.
- Confirms volume configuration and read/write access.

Figure 15: Docker Network IP Assignment

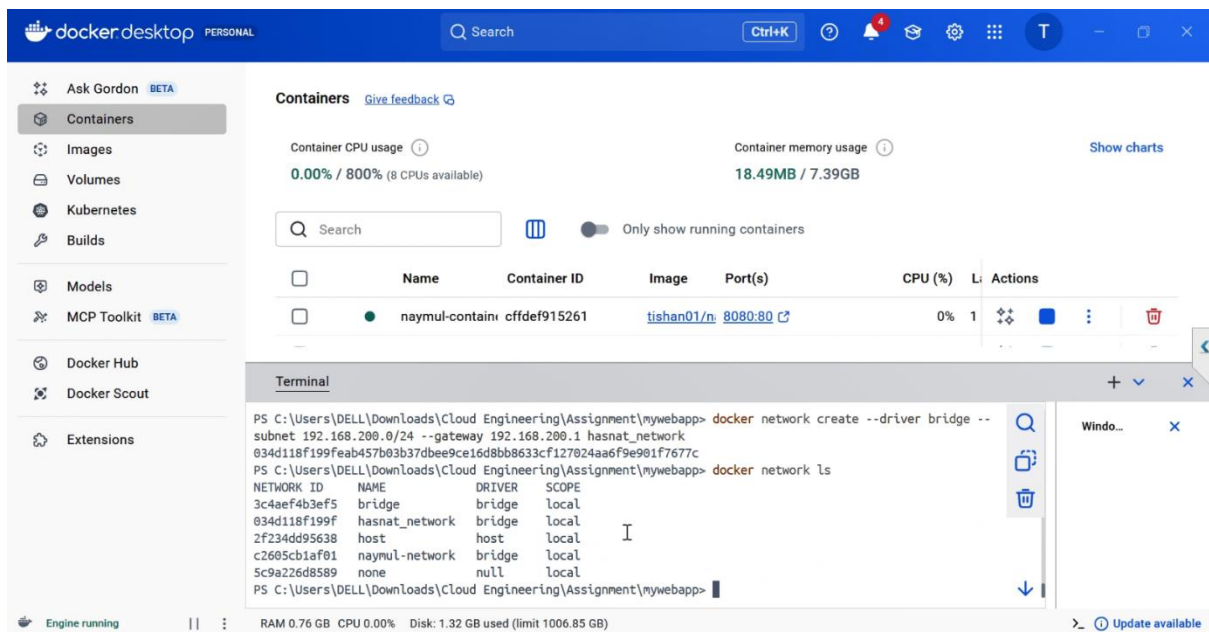


Figure 15

- JSON output showing hasnat_network IP address and gateway for hasnat-container.
- Validates internal IP assignment and network routing setup.

4.3 Volumes and bind mounts

Containers are ephemeral by default; persistent data requires volumes or bind mounts.

Creating a named volume

`docker volume create hasnat_volume`

Figure 16 shows the command `docker volume create hasnat_volume` and the returned volume name `hasnat_volume`. This demonstrates creating a Docker managed volume for persistent storage.

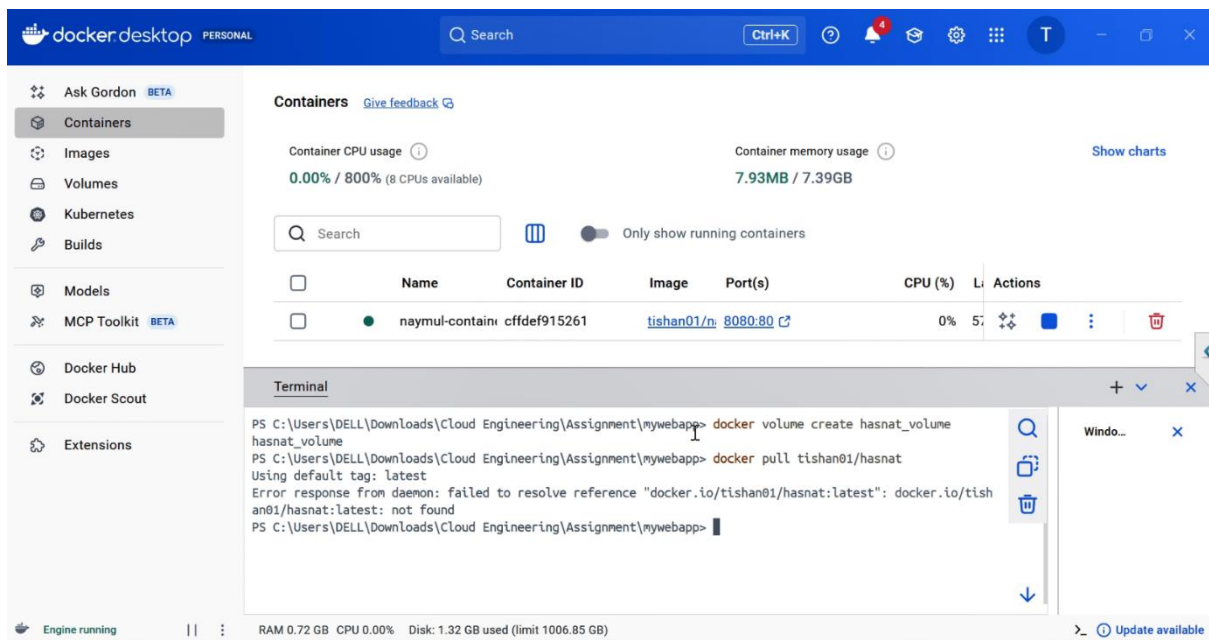


Figure 16

4.4 Container networking

Docker provides several networking modes: bridge (default), host, and overlay (for multi-host). For the lab:

Custom bridge network

`docker network create my_bridge`

`docker run -d --name social-homepage --network my_bridge -p 8080:80 naymul/social-homepage:1.0`

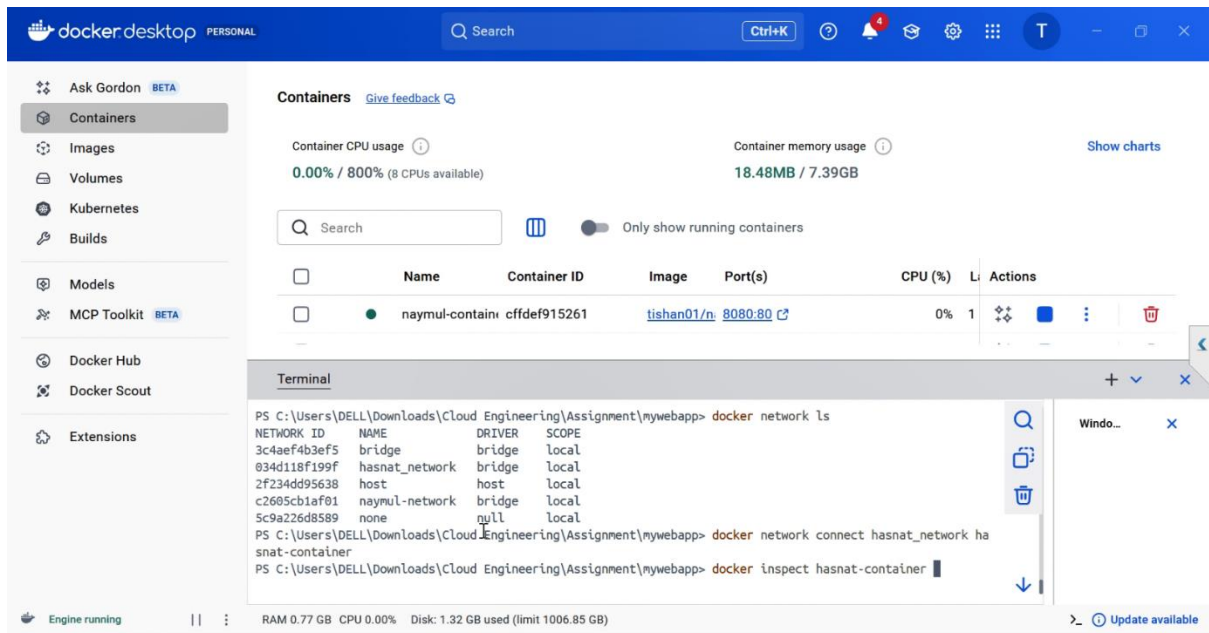


Figure 17

Containers on the same custom bridge network can resolve each other by name, enabling service discovery without exposing ports externally.

Figure 17 show docker images output and docker pull/docker push operations. The terminal logs indicate successful pulls of tishan01/hasnat-image:latest and the presence of multiple images locally. The student also attempted to pull tishan01/hasnat (which failed), then successfully pulled tishan01/hasnat-image.

Discussion

- Networking is essential for multi-container applications (e.g., web server + API + database).
- Custom bridge networks provide name-based resolution and isolation.
- Overlay networks (not shown in the lab) are used for multi-host orchestration (Kubernetes, Docker Swarm).

4.5 Publishing images to Docker Hub

Publishing images enables reuse and deployment across environments.

Tagging and pushing

```
docker tag hasnat-image tishan01/hasnat-image:latest
```

```
docker push tishan01/hasnat-image:latest
```

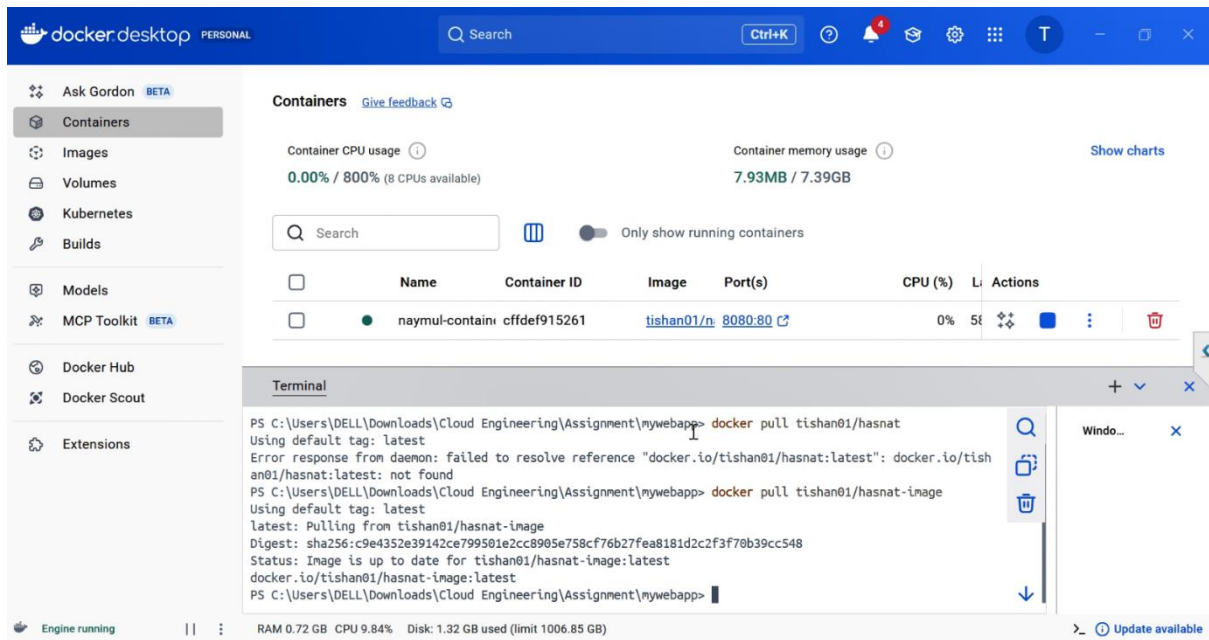


Figure 18

Figure 18 shows the Docker tag and Docker push commands and the Docker Desktop output indicating layers being mounted and pushed. The Docker Hub repository page for tishan01/hasnat-image confirms the repository exists with a latest tag.

Discussion

- Tagging with a repository name (e.g., username/repo:tag) is required for pushing to Docker Hub.
- The push output shows layer uploads and a final digest, confirming the image is available remotely.
- Pull tishan01/hasnat failed because the repository name did not match; pulling tishan01/hasnat-image succeeded, illustrating the importance of correct naming.

5. Load Balancing, Scalability, and Application Gateways: Concepts and Cloud Implementations

This section synthesises the theoretical concepts of load balancing, scalability, and application gateways and maps them to cloud provider implementations (AWS and Azure).

5.1 Load balancing: layers, types, and cloud offerings

OSI layers and load balancing

- **Layer 4 (Transport):** Load balancers at the Layer 4 (TCP/UDP) layer route traffic based on IP and port addresses without examining application-level data. They are fast and work well for high throughput low latency solutions.
- **Layer 7 (Application):** Layer 7 load balancers see HTTP(S) requests and can route based on hosts, URL paths, cookies and request content. They support advanced routing, SSL offload and WAF integration [8].

AWS offerings

- **Application Load Balancer (ALB):** Layer 7 Load Balancer for HTTP/HTTPS with host/path-based routing, Websocket Support and integrated with AWS WAF and target groups. We use ALB a lot for microservices and containerized workloads (ECS, EKS) where the content-based routing is needed.
- **Network Load Balancer (NLB):** Layer 4 load balancer that is purpose-built for high availability and performance, with static IP support. NLB is an appropriate choice for TCP or UDP workloads and can also provide high power of connection rates.
- **Classic Load Balancer (CLB):** Old choice, also supports L4 and L7 but has fewer features; largely replaced with ALB/NLB [9].

Azure offerings

- **Azure Load Balancer:** Distributes traffic at the network level (Layer 4) across VMs inside a VNet to ensure that only healthy VMs receive traffic. Supported: public and internal load balancing.
- **Azure Application Gateway:** Load balancer with additional layer 7 functionality (URL based routing, SSL termination, session affinity) and Web Application Firewall (WAF). It could serve as the ingress controller for AKS.
- **Azure Front Door:** Platform with global HTTP/HTTPS load balancing and CDN - State of the art for global routing, performance and scalability [10].

Design considerations

- **Health checks:** Load balancers depend on health probes to send traffic only to healthy backends.
- **Session affinity:** For stateless applications stateless designs should be used, while for the stateful one's sticky sessions might be needed.
- **SSL termination:** By decrypting at the load balancer instead of on your backend, you decrease your backend CPU utilisation and have a centralised place to manage certificates.

5.2 Scalability: vertical, horizontal, diagonal; cloud services

Types of scalabilities

- **Vertical scaling (scale up):** Adding capacity to a single instance (e.g., bigger VM). Straight forward but also hardware dependent and a single point of failure.
- **Horizontal scaling (scale out):** Add more instances to handle the load. Our choice for cloud-native, stateless architectures since it increases fault tolerance and allows for near-unlimited growth.
- **Diagonal scaling:** Transpose the techniques for vertical/horizontal to scale instance size and count, as required [11].

Cloud provider implementations

AWS

- **Auto Scaling Groups (ASG):** Handle EC2 instance scales as per policies (CPU, network, custom metrics). ASGs work with ALB target groups to add/remove instances from the pools that are being load balanced.
- **ECS/Fargate autoscaling:** Scale container tasks based on metrics or request counts.
- **Lambda scaling:** Serverless functions can be scaled automatically based on event triggers and have concurrency limits that can be set [11].

Azure

- **Virtual Machine Scale Sets (VMSS):** Control a set of VMs that are identical and scale them according to need, using metrics or schedules.
- **App Service Plan scaling:** Web apps can scale out/in automatically.
- **Azure Functions scaling:** Consumption and Premium plans scale out for you automatically [10].

Autoscaling policies and metrics

- **Reactive scaling:** Triggered by thresholds (e.g., CPU > 70% for 5 minutes).
- **Predictive scaling:** Uses historical patterns and forecasts to scale proactively (supported by some cloud providers).
- **Scheduled scaling:** Scale based on known traffic patterns (e.g., business hours, promotions).

Integration with load balancers

Load balancers should be included into the autoscaling setup: new instances need to be registered and obtain traffic. Health checks ensure only the healthy ones are included [8].

5.3 Application gateways and API management

Application gateways

Application gateways operate at Layer 7 and act as reverse proxies, providing:

- **SSL/TLS termination and re-encryption** for end-to-end or edge termination.
- **URL-based routing** to direct requests to specific backend pools.
- **Session affinity** when required.
- **Web Application Firewall (WAF)** to protect against OWASP Top 10 threats (SQL injection, XSS).
- **Authentication and authorization** integration (e.g., with IAM or Active Directory).

API gateways

API gateways (e.g., AWS API Gateway, Azure API Management) provide API lifecycle management features:

- **Authentication and authorization** (JWT, OAuth).
- **Rate limiting and throttling** to protect backend services.
- **Request/response transformation** and validation.
- **Monitoring and analytics** for API usage.

Use cases

- **Microservices:** Route requests to microservices based on path or host.
- **Multi-tenant applications:** Host multiple sites behind a single gateway with host-based routing.
- **Security:** Centralise WAF rules and TLS management [12].

6. Case Study: E-Commerce Platform Architecture

In this case study we have seen how containerised microservices, load balancing engines, autoscaling and application gateways all interoperate to provide a robust, secure and high-performance e-commerce platform. A contemporary e-commerce system usually utilizes several higher-level components that work together towards providing the best customer experience. On the edge, a global CDN like AWS CloudFront or Azure Front Door caches static assets—images, CSS and JavaScript among others—to decrease latency and relieve load off origin servers. Requests then hit an Application Gateway or API Gateway where it terminates TLS and applies WAF rules to smartly route traffic to the backend [13].

Underneath that a load balancer (like AWS ALB/NLB or Azure Load Balancer/Application Gateway) belongs to multiple backend instances to guarantee even workload and availability. The application itself is broken down into microservices (catalog, cart, orders, payments, user management) that are deployed as containers on ECS/EKS/AKS where applicable or as serverless functions. These services talk to transactional databases (Aurora, RDS, Cosmos DB) and high-throughput NoSQL stores (DynamoDB), on top of in-memory caches such as Redis or ElastiCache to speed up session management and product lookups. In the system, monitoring and logging tools including CloudWatch (CW), Azure Monitor (AM), Application Insights (AI) and ELK/Opensearch gather metrics, traces and logs for observability and operational understanding.

This architecture is where the traffic flow in this article starts: a client request hits the CDN or global front door. The Application Gateway will then unwrap TLS, apply WAF policies and forward the API endpoint. The request is routed by the load balancer to a healthy backend instance, allowing no single instance to be overwhelmed.

Security, observability and compliance are the pillars of a production-ready e-commerce system. IAM/RBAC provides fine-grained control over cloud resources, while the WAF can secure your web applications from prevalent threats including SQL injection and cross-site scripting. You can rely on robust TLS configurations and automated certificate rotation to keep communication channels secure, while secrets should be kept in managed services like AWS Secrets Manager or Azure Key Vault. We have observability using metrics (CPU, memory, latency, error rates), distributed tracing (AWS X-Ray, Application Insights) and centralised logging around auditing and troubleshooting. Compliance – compliance requirements (e.g. GDPR or PCI DSS) are met using regions, encryption and provider-specific certification [14].

7. Complementarity and Combined Benefits

Load Balancing, Autoscaling and Application Gateways are naturally working in tandem as complementary pieces of a cloud-native architecture. The load balancers spread the incoming traffic over several instances to avoid overload and allow horizontal scaling to work at full capacity. Autoscaling automatically scales up or down compute capacity based on demand, however newly created instances would not receive traffic consistently without a load balancer applied. Application gateways offer more intelligence, with Layer 7 routing, centralised TLS termination and WAF protection to ensure that traffic is directed in a secure and efficient manner towards the correct microservices. When combined, these technologies bring a ton of value to customers through redundant instances and health checks, performance by routing intelligently and caching responses at the edge, security with WAF, central TLS termination; cost reduction with autoscaling away capacity (and reducing spend on overprovisioned resources); business agility that lets you quickly scale in

response to promotions / flash sales. For an e-commerce platform, that combination means keeping the storefront responsive, secure and cost-optimised even at the top of traffic spikes [15].

8. Serverless Functions: Concepts, Implementations, and Comparison with Containers

8.1 AWS Lambda and Azure Functions

AWS Lambda is Amazon's serverless computing service, and it lets people upload small functions that become activated in response to certain events a call to an API endpoint, for example, or when a file is uploaded to S3, or when a message from a queue has been added. Lambda abstracts away the servers themselves, for scaling, load balancing and execution infrastructure. It scales basically instantly—if you get a thousand events at once, Lambda will try to execute all of those in parallel (within your account's concurrency limits). All the invocations are stateless and separated from each other, any data that should be saved between invocation must be saved outwards on services like DynamoDB or S3. Billing is metered in increments of milliseconds, not seconds, with cost based on the number of invocations and compute time used so you can run applications for virtually any type of application or backend service—all with zero administration [16].

Microsoft's answer to FaaS Azure Functions delivers its Answer to the AWS Lambda) functionality that can be triggered by a variety of patterns including (HTTP requests, timers, Blob Storage uploads and queue messages. It also allows for various languages such as C#, JavaScript, Python, etc. Azure has three hosting plans: the Consumption Plan, which works like Lambda with autoscaling and pay-per-execute billing; the Premium Plan, which pre-warms instances to minimize cold-start time; and the Dedicated Plan, on a full App Service plan for teams who want more control or have reserved capacity already. Azure Functions also automatically scale in the Consumption and Premium levels, with similar limits to Lambda. One of the great things about Azure Functions is its rich set of integrations with other Azure services, via bindings that let you easily connect input triggers and output bindings—such as a Service Bus queue or Cosmos DB—without having to write tons of boilerplate code. Like Lambda, Azure Functions are popular for event-driven usage patterns, light workloads make great use of it as well - quick lightweight APIs, real-time stream processing and very often orchestration with Logic Apps [17].

8.2 Comparison: Cost, Performance, Control and Compliance

If you compare serverless functions to containers, the first thing that stands out is often cost. Serverless platforms such as AWS Lambda and Azure Functions only bill for the period when your code is running, so you get very cost-effective execution for workloads that are sporadic

or unpredictable. If nothing is running, you are paying nothing. This is perfect for small projects, automation, or workloads that occasionally spike. But for applications that are constantly running or processing very high, steady traffic, the pay-per-execution model can be pricier than simply running containers on a VM, where you pay a fixed cost for the machine regardless of activity on it. In these cases, containers, on reserved or spot capacity can be more predictable and cheaper [16].

Control is another major factor. In a serverless model, developers relinquish almost all control of the environment. With containers, it's broad: select a base image, install what you require, run processes in the background when needed, keep persistent connections and generally tune performance at a lower level. This also makes containers more appropriate for complex or specialised software.

Compliance and security roles also vary. Serverless outsources a lot of the security and patching to the cloud provider, so this can make compliance easier for many businesses. All invocations run in a clean, isolated context minimizing the attack surface. But because serverless is multi-tenant, certain heavily regulated industries might want the isolation and insight containers provide.

The choice is dictated by your workload. Serverless is perfect for event-driven, ephemeral, or unpredictable workloads that have a focus on simplicity and cost effectiveness. Containers are a better choice when you have long-running services that need the same state as your local setup, or for applications that demand consistent performance environments when you develop and test [18].

8.3 Hybrid Approaches and Recommended Patterns

Many organisations now combine serverless functions and container-based services because each excels in different situations. A common pattern is to run the main application such as an e-commerce website and its core APIs on containers for predictable performance, while offloading bursty or background tasks to serverless functions. For example, image processing, email notifications, or other asynchronous jobs can be handled by AWS Lambda or Azure Functions whenever needed, without keeping extra servers running. API Gateways also make it easy to mix both worlds by routing some endpoints to serverless functions and others to containerised microservices, giving clients a single unified API [19].

Scheduled jobs are another area where serverless shines. Instead of maintaining services on a VM, organisations often use timer-triggered Azure Functions or scheduled AWS Lambda tasks for daily reports or maintenance routines. More complex workflows can be orchestrated using services like AWS Step Functions or Azure Durable Functions, which coordinate multiple steps across both serverless and containerised components. In many systems, containers and serverless functions even call each other—containers may offload a heavy calculation to a high-memory Lambda, while a serverless function might call a containerised API to store or retrieve data.

In practice, the recommended approach is to use serverless for spiky, infrequent, or highly parallel tasks, and containers for long-running, stateful, or latency-sensitive services. Managed cloud services—databases, queues, authentication—help reduce operational overhead regardless of the compute model. Security should follow the principle of least privilege, giving each function or container only the permissions it needs. Monitoring and logging should also be unified so both serverless and container workloads can be observed together. Finally, decoupling components with queues or pub/sub systems improves resilience and makes it easier to mix both technologies without tight coupling.

The result is often a blended architecture: static content served from a CDN, user-facing APIs running in containers for consistent performance, background processing handled by serverless functions for effortless scaling, and managed cloud services tying everything together. This hybrid model offers the best of both worlds—containers provide control and stability, while serverless adds flexibility and cost efficiency for unpredictable workloads [20].

9. A serverless function vs. container comparison

There are distinct cost, performance, control and compliance differences between containers and serverless functions. From a cost point of view, serverless computing is also very efficient for bursty or low-utilisation workloads as you are billed based on the execution time, so no idle capacity costs. Containers are generally more cost-predictable (and better) for steady, high-throughput workloads and particularly when long-running services or reserved instances can be used to amortise compute costs. Performance characteristics may also vary; serverless functions can suffer cold starts (the delay occurring the first time a function is ran after being idle), although there are workarounds such as provisioned concurrency to mitigate this. Once a container is running, the performance is both consistent and predictable - making containers an ideal platform for long-running or latency-sensitive services.

Another major difference is control and portability. Serverless Platforms abstract away much of the underlying infrastructure, making operations simpler but also reducing the control that developers have over their runtime environments. This abstraction can also lead to potential vendor lock-in with vendor-specific integrations. When you use containers, however, you have full control of the operating system, runtime and dependencies for your application — giving you the ability to build a container once and run it anywhere (in cloud providers or on prem environments). The compliance and security profile is also different – serverless platforms are taking on some of the security and compliance posture for the customer, a benefit to most regulated workloads but it may limit how much you can customize or create new controls. Containers give you more freedom to apply your own compliance measures like network isolation or custom encryption, but also in turn bring extra operational responsibility. There are fewer administrative concerns with serverless computing simply because there aren't any servers to manage, yet debugging and local

testing can be trickier. Containers need a layer for orchestration like Kubernetes that adds complexity in return of rich operational control and flexibility [21].

It's very common in practice for organisations to take a hybrid approach, combining the best of both serverless functions and containerised services. Serverless functions are most employed for event-based processes like image manipulation, notifications and asynchronous workloads that require rapid scaling and cost efficiency. Containers are most suitable for your core business logic - e.g., your catalogue, payments and user services -- when performance predictability, fine-tune control and custom setup is important. Often, an API Gateway is leveraged to homogenize access patterns and direct traffic towards either serverless functions or containerized microservices – based on the perceived nature of the workload. By utilizing common observability solutions, traces, logs & metrics can all be correlated across the two environments to provide end-to-end visibility. This hybrid approach strikes a balance between cost efficiency, agility, operational control and architectural freedom [22].

10. Conclusion

This report aligned the hands-on containerization lab with a discussion of cloud delivery patterns both as theoretical and practiced concepts. The lab provided participants with hands-on experience in the complete process of containerising a static HTML site with Docker and Nginx including writing a Dockerfile, building an image and running it, working with volumes and networks, and images publishing on Docker Hub. The cloud design review on how load balancers/partitioning, autoscaling and application gateways work together to deliver a resilient, secure and performant service—this is important stuff, particularly for people who are running Ecommerce applications where 100% availability in a super-secure manner is just mandatory.

Serverless functions and containerisation each provide their own benefits: while serverless is great for cost-effective, event-driven workloads, containers give control, portability and stable performance guarantees for core services. Hybrid is usually the best mix of cost, performance and operational control [23].

Figures showing the steps performed in practice in the lab and concerning routine operations as well as common mistakes (e.g., image naming, pulling). Combining the theoretical and hands on elements, this document provides evidence indicating that virtualisation (VMs), containerisation, or even very long-time period managed cloud solutions are here to stay as a main building blocks of software distribution.

11. Docker Project:

Here is my YouTube video link: <https://youtu.be/JCWotW8UDs4>

References

- [1] A. B. G. P. B. G. K. Prathamesh Muzumdar, "Navigating the Docker Ecosystem: A Comprehensive Taxonomy and Survey," *Distributed, Parallel, and Cluster Computing*, 3 Jan 2024.
- [2] codezup, "Mastering Docker Deployment for Single-Page Applications.," 2024.
- [3] osherachamim, "Dockerized HTML Website with Nginx".
- [4] Kuldeepkumhar, "A Complete Guide to Docker: Images, Containers, Networking, and Volumes," 2025.
- [5] D. R. H. F. J. W. David Manor, "Containerization in Cloud Computing: A Review," 2025.
- [6] N. U. Amin, "The Role and Application of Hypervisors," 2025.
- [7] P. J. J. N. N. Kaur, "Performance Analysis of Docker Containerization and Virtualization," 2022.
- [8] <https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html>, "What is an Application Load Balancer?".
- [9] <https://docs.aws.amazon.com/lambda/>, "AWS Lambda Documentation".
- [10] <https://learn.microsoft.com/en-us/azure/application-gateway/>, "Azure Application Gateway documentation".
- [11] <https://docs.aws.amazon.com/elasticloadbalancing/latest/application/application-load-balancers.html>, "Application Load Balancers".
- [12] V. A. T. C. C. Chavda, "Analyzing Docker Vulnerabilities through Static and Dynamic Methods and Enhancing IoT Security with AWS IoT Core, CloudWatch, and GuardDuty," 2024.
- [13] A. W. Services, "Building Modern Applications on AWS: Microservices, Containers, and Serverless, AWS Whitepaper," 2024.

- [14] M. Azure, "Azure Architecture Framework: Designing Scalable and Resilient Cloud Applications, Microsoft Documentation," 2024.
- [15] A. W. Services, "AWS Well-Architected Framework: Reliability and Performance Efficiency Pillars, AWS Whitepaper," 2024.
- [16] <https://docs.aws.amazon.com/lambda/latest/dg/welcome.html>, "What is AWS Lambda?".
- [17] <https://learn.microsoft.com/en-gb/azure/azure-functions/>, "Azure Functions documentation".
- [18] C. A. Center, "Design cloud topologies that are secure, efficient, resilient, high-performing, and cost-effective".
- [19] <https://aws.amazon.com/architecture/>, "Get Started Architecting on AWS".
- [20] <https://learn.microsoft.com/en-gb/azure/architecture/>, "Azure Architecture Center".
- [21] <https://martinfowler.com/articles/serverless.html>, "Serverless Architectures".
- [22] <https://martinfowler.com/articles/serverless.html>, "Serverless Computing vs. Containers: A Comparative Analysis, CNCF Whitepaper".
- [23] <https://www.nist.gov/>, "The NIST Definition of Cloud Computing," NIST Special Publication 800-145," 2024.