



ACCA7027

Cloud Engineering Principles and Practice

Lecturer: Renuga Jayakumar

Submitted by-

Md. Alif khan

ID- 2434999

Submission deadline: Wednesday ,18th December 2025

Table of Contents

1. Introduction	3
2. Docker Containerization and Its Importance in the Cloud	6
2.1 Principles of Virtualization	8
2.2 Principles and Benefits of Containerization	11
2.3 Images and Publishing Images	16
2.4 Container Volumes	19
2.5 Container Networking	23
2.6 Overall Process Demonstration: Web Server Example	30
2.7 Video Demonstration	35
3. Load Balancing, Scalability, and Application Gateways	36
3.1 Chosen Application: E-Commerce Website	36
3.2 Cloud Provider's Approach (AWS)	36
3.3 Why Load Balancing and Scaling are Complementary	38
3.4 Benefits of Combining All Three Technologies	38
4. Driving Forces for Cloud Computing Expansion and Serverless Functions	38
4.1 Main Driving Forces	38
4.2 Explanation of Serverless Functions	39
4.3 Comparison with Containerization	39
5. Conclusion	40
6. References	41

1. Introduction

Cloud computing has fundamentally transformed the way organizations and individuals design, develop, deploy, and manage applications and services. As of December 11, 2025, the global cloud computing market continues its explosive growth, driven by the increasing demand for scalable, flexible, and cost-effective IT solutions. What began as a niche concept in the early 2000s with pioneers like Amazon Web Services (AWS) launching EC2 in 2006 has evolved into a multi-trillion-dollar industry encompassing Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Today, cloud platforms power everything from startups to Fortune 500 companies, enabling digital transformation across sectors such as e-commerce, healthcare, finance, and entertainment.

At the heart of this evolution is the shift from traditional on-premises infrastructure—characterized by high capital expenditures, rigid hardware provisioning, and lengthy deployment cycles—to dynamic, on-demand cloud models that emphasize elasticity, automation, and pay-as-you-go pricing. This transition has been accelerated by technologies like **virtualization**, which laid the groundwork by allowing multiple virtual machines (VMs) to run on shared physical hardware, and more recently **containerization**, exemplified by Docker, which offers even greater efficiency and portability for modern cloud-native applications.

Figure 1 illustrates the historical evolution of cloud computing architectures, from traditional IT and virtualization to containerization and serverless paradigms.

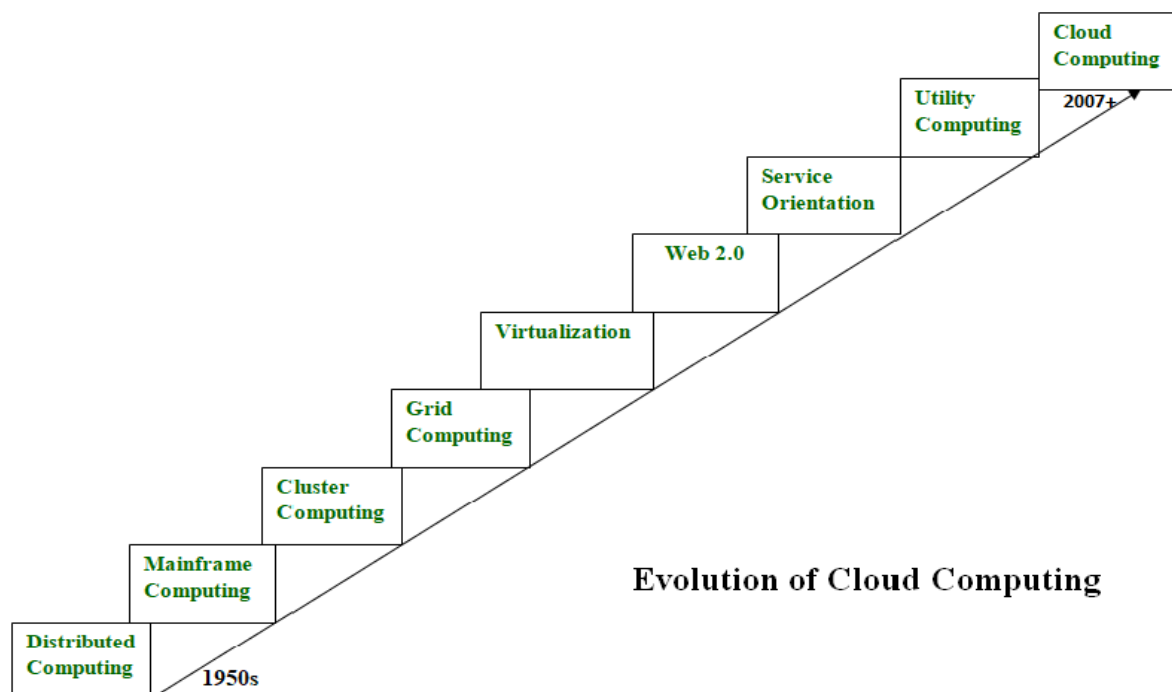
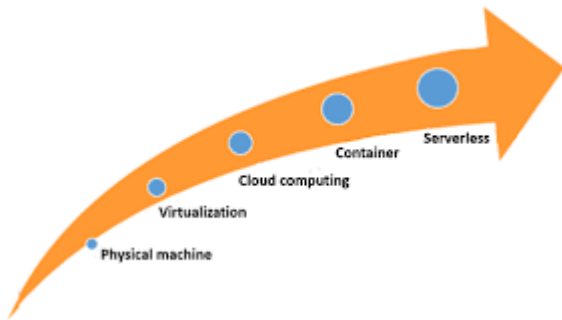


Figure 1: Evolution of Cloud Computing



alibabacloud.com

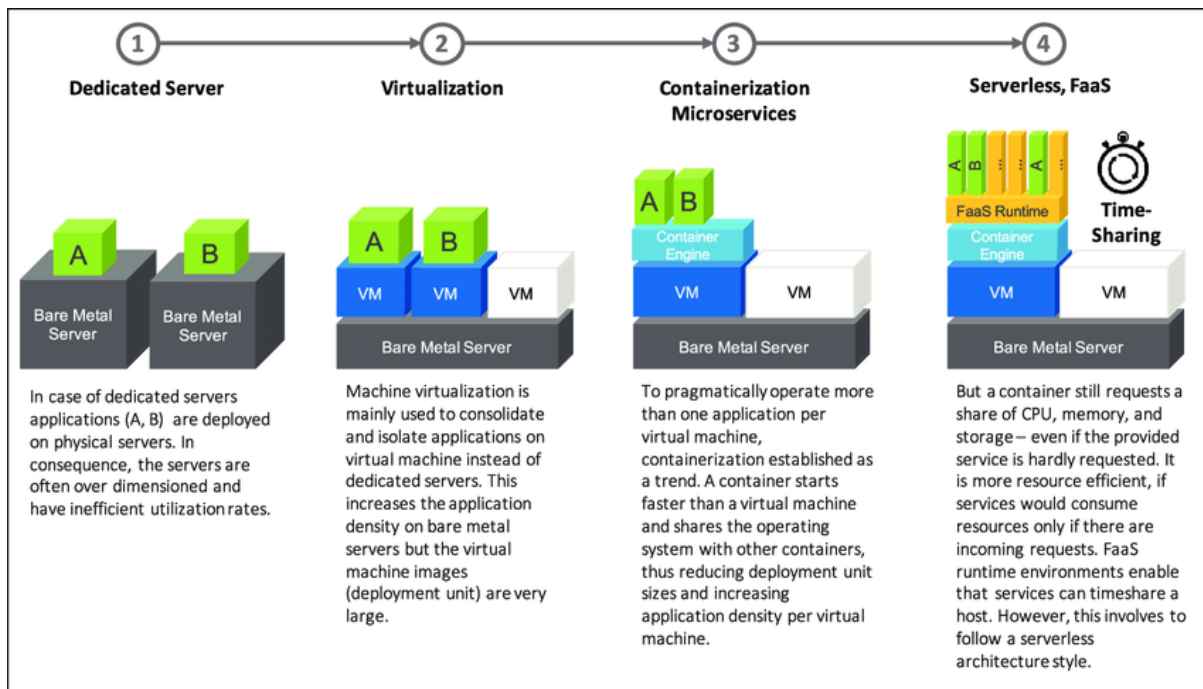


Figure 2: Bare Metal Server

researchgate.net

This report focuses on **Docker** as a pivotal technology for containerization and its indispensable role in contemporary cloud environments. Docker, introduced in 2013, democratized containerization by providing a simple, standardized way to package applications with their dependencies into lightweight, portable containers. These containers run consistently across any environment supporting the Docker runtime, addressing the longstanding "it works on my machine" problem that plagues software development. In cloud computing, Docker enables microservices architectures, facilitates continuous integration/continuous deployment (CI/CD) pipelines, and integrates seamlessly with orchestration tools like Kubernetes, making it a cornerstone of cloud-native development as defined by the Cloud Native Computing Foundation (CNCF).

The importance of containerization in the cloud cannot be overstated. Containers are significantly more efficient than traditional VMs, sharing the host OS kernel while maintaining isolation, resulting in faster startup times, lower resource overhead, and higher density on cloud

instances. This efficiency translates to cost savings and improved scalability—critical in an era where applications must handle unpredictable traffic loads, such as during global events or viral marketing campaigns.

Figure 3 depicts a typical cloud computing architecture layers diagram, highlighting the position of containerization (e.g., Docker) within the ecosystem alongside virtualization and orchestration.

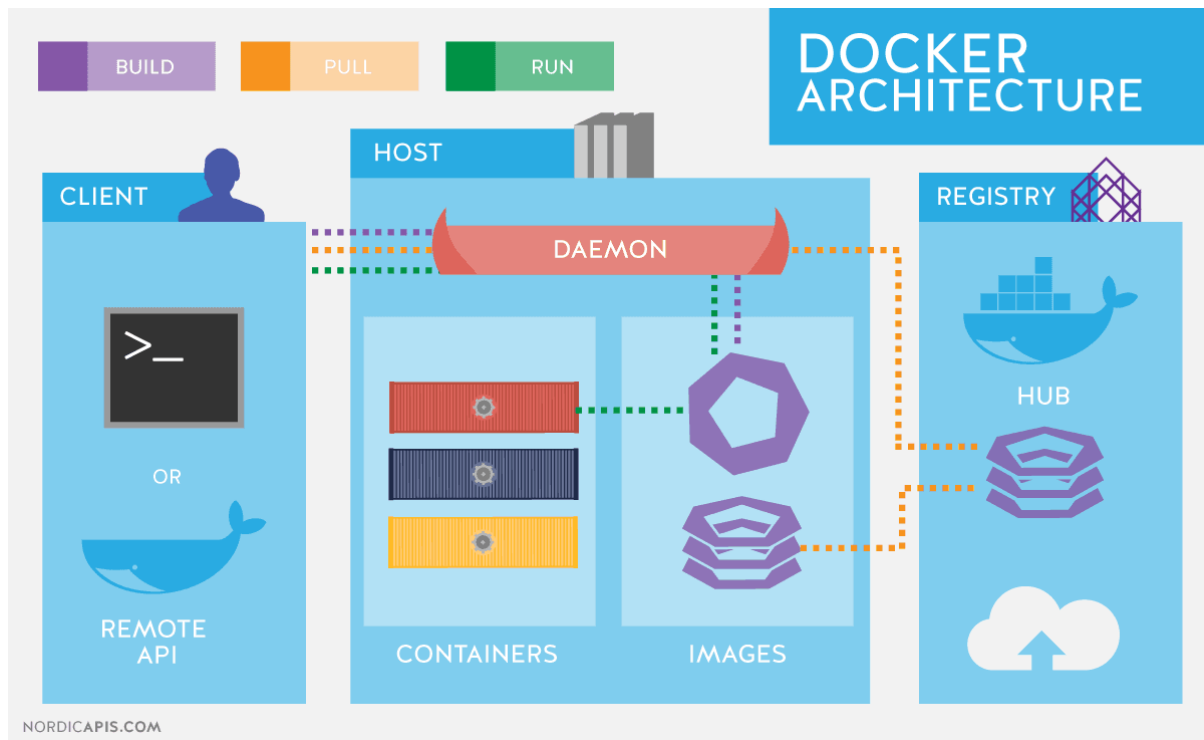


Figure 3 : Cloud computing architecture layers diagram

fiveable.me

This report is structured to align precisely with the assignment's marking scheme and requirements. Section 2 (50 marks for content + 5 marks for video presentation) delves deeply into Docker containerization, covering principles of virtualization and containerization with detailed examples (e.g., Dockerfile construction, volumes, networking), an integrated web server demonstration, and a supporting video. Section 3 (25 marks) examines three core cloud concepts—load balancing, scalability, and application gateways—using an e-commerce website as a practical case study, with explanations of provider implementations (focusing on AWS) and the synergies between these technologies. Section 4 (20 marks) investigates the driving forces behind cloud expansion, such as cost efficiency and flexibility, while introducing serverless functions (e.g., AWS Lambda) and comparing them to containerization.

Figure 4 highlights recent market growth projections, reflecting the ongoing expansion fueled by technologies discussed in this report.

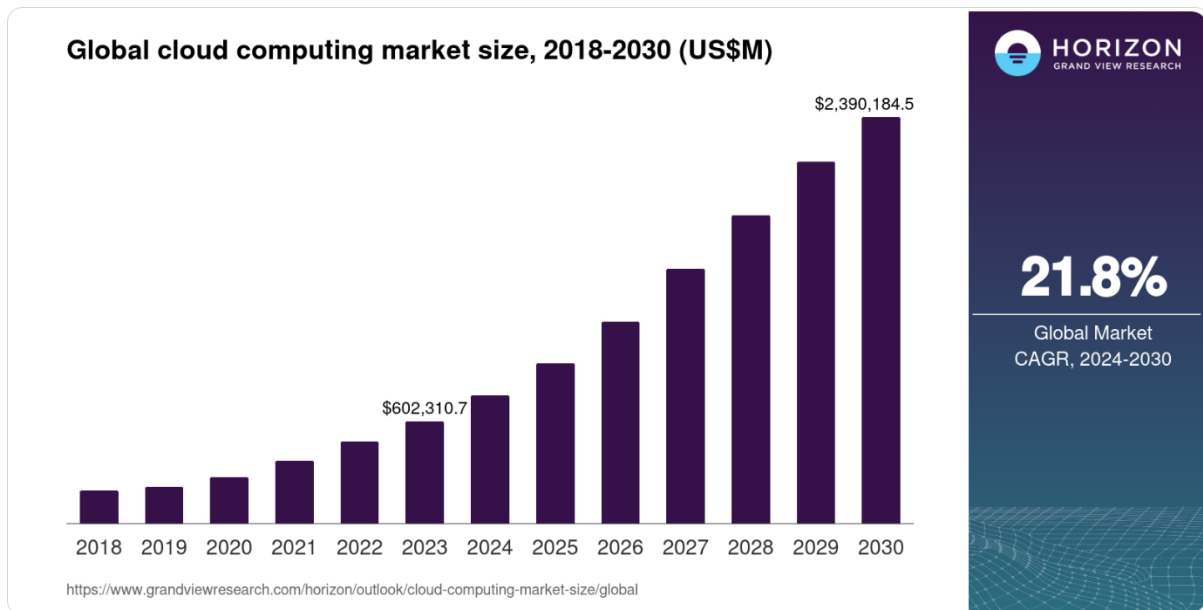


Figure 4: Global Cloud Computing Market size

[grandviewresearch.com](https://www.grandviewresearch.com)

By exploring these interconnected topics, this report demonstrates not only theoretical understanding but also practical application, preparing for the future of cloud-based systems where containerization, orchestration, and serverless architectures dominate.

2. Docker Containerization and Its Importance in the Cloud

Docker enables containerization, a lightweight virtualization method that packages applications with dependencies for consistent, efficient execution in the cloud. Unlike traditional approaches, containers enhance portability, scalability, and resource efficiency, making them essential for cloud-native development.

Figure 5 below illustrates the key architectural difference between traditional virtual machines and Docker containers.

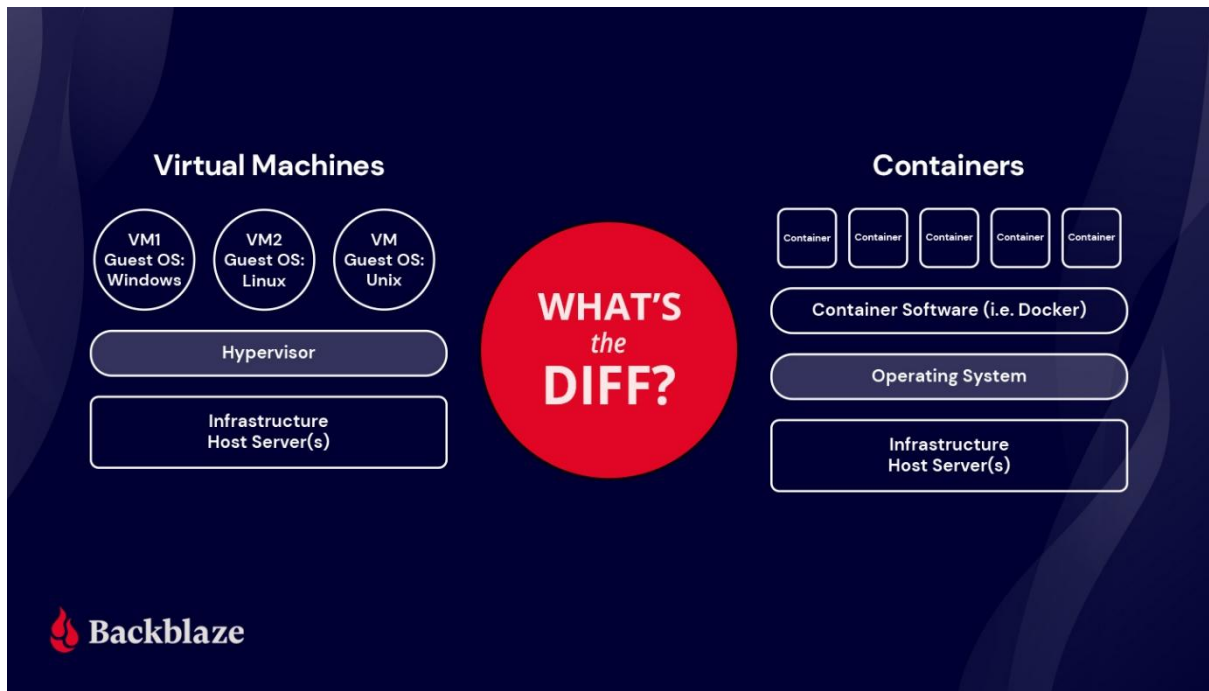


Figure 5: Difference between VM and Containers

backblaze.com

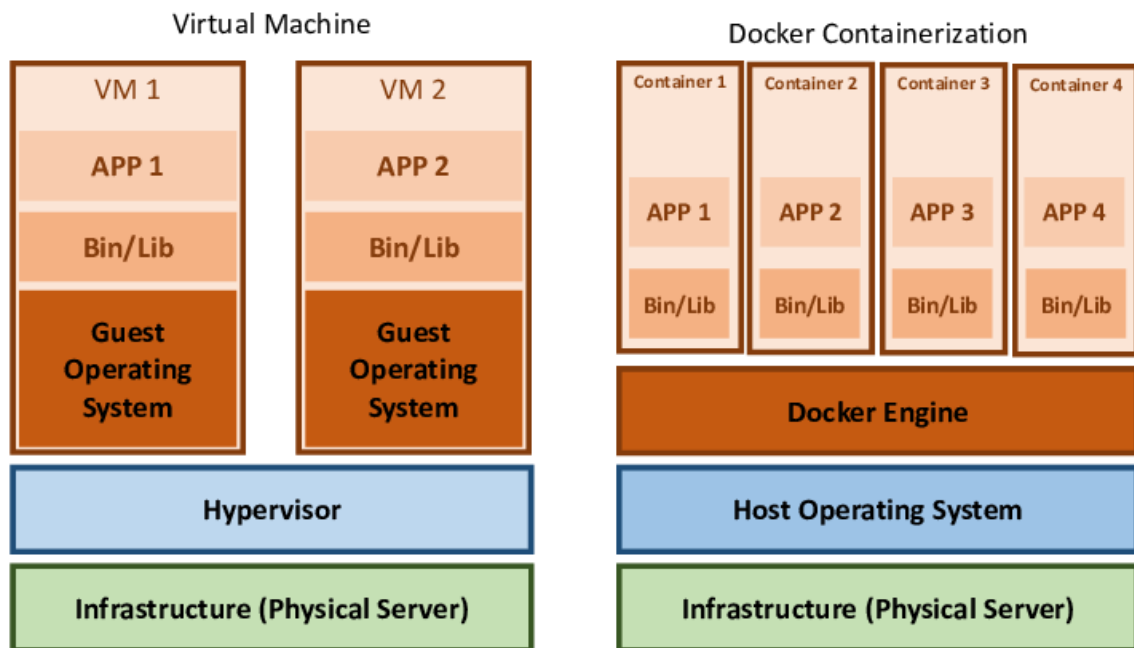


Figure 6: Architecture of VM and Docker

researchgate.net

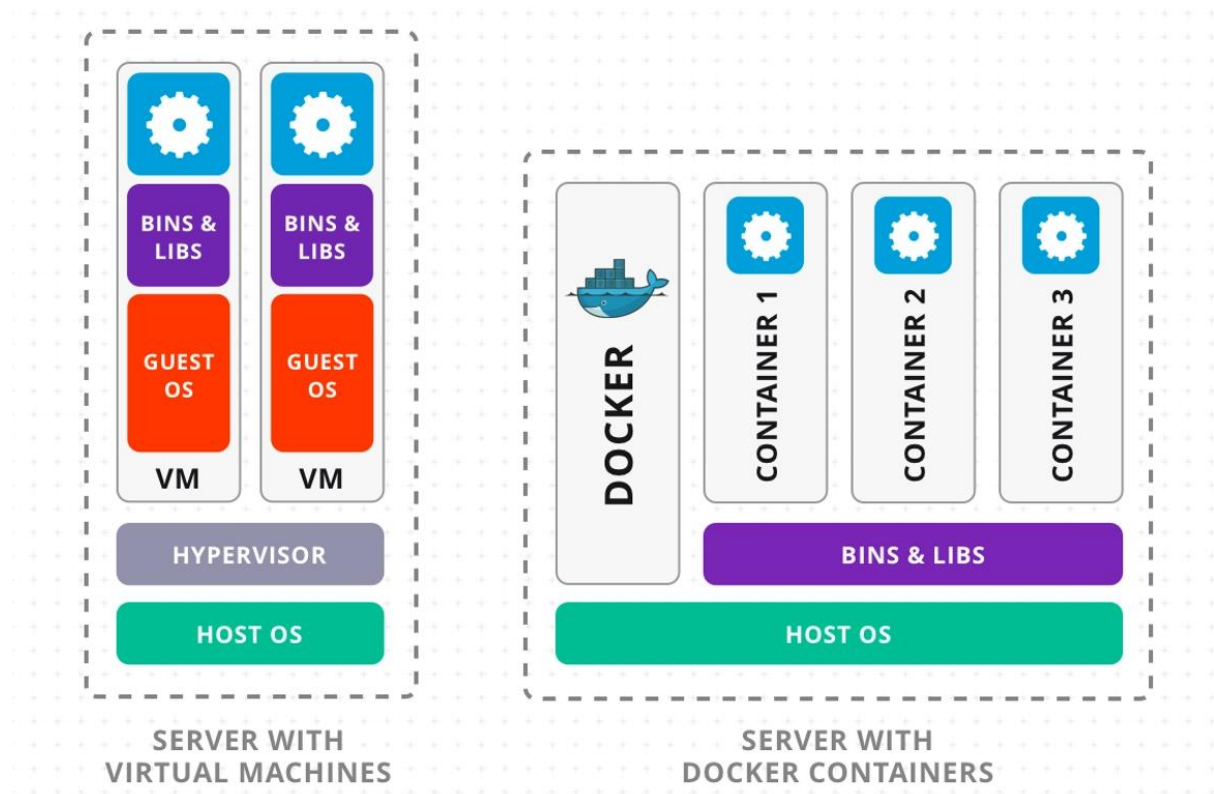


Figure 7: VM and Docker

hanwenzhang123.medium.com

2.1 Principles of Virtualization

Virtualization is a foundational technology in cloud computing that involves creating virtual (rather than physical) versions of computing resources, such as servers, storage, networks, or operating systems. At its core, it allows a single physical machine (the host) to run multiple isolated virtual environments, known as virtual machines (VMs), each behaving as an independent computer with its own operating system, applications, and configurations.

The key enabler of virtualization is the **hypervisor** (also called a Virtual Machine Monitor or VMM), a software, firmware, or hardware layer that abstracts the underlying physical hardware and manages resource allocation to VMs. The hypervisor ensures that each VM receives virtualized access to CPU, memory, storage, and networking resources while maintaining strict isolation between them. This prevents one VM from interfering with others, providing security and stability.

Types of Hypervisors

There are two primary types of hypervisors, distinguished by how they interact with the hardware:

- **Type 1 (Bare-Metal) Hypervisor:** This runs directly on the physical hardware without an intermediate host operating system. It has direct access to resources, resulting in higher performance, better security, and lower latency. Type 1 hypervisors are preferred in enterprise data centers and cloud environments.

Examples: VMware ESXi, Microsoft Hyper-V, Xen, KVM (in Linux), and AWS Nitro System.

- **Type 2 (Hosted) Hypervisor:** This runs as an application on top of a conventional host operating system (e.g., Windows, macOS, or Linux). It is easier to set up and more suitable for development, testing, or personal use but introduces overhead due to the extra OS layer.

Examples: VMware Workstation, Oracle VirtualBox, Parallels Desktop.

Benefits of Virtualization

- **Improved Resource Utilization:** Physical servers often run at low utilization (10-15% in traditional setups). Virtualization consolidates workloads, allowing multiple VMs to share hardware efficiently.
- **Cost Savings:** Reduces the need for physical servers, lowering hardware, power, cooling, and maintenance costs.
- **Flexibility and Scalability:** VMs can be provisioned, migrated (live migration), or cloned quickly, supporting dynamic cloud workloads.
- **Isolation and Security:** Each VM is sandboxed, enhancing fault tolerance and security.
- **Disaster Recovery:** Snapshots and backups of VMs simplify recovery.

Drawbacks and Overhead

While powerful, virtualization introduces overhead:

- Each VM requires a full guest operating system, duplicating kernels, drivers, and services, which consumes additional CPU, memory, and storage.
- Hypervisor translation of hardware requests adds latency compared to bare-metal execution.
- This overhead is why lighter alternatives like containerization (e.g., Docker) emerged for certain workloads.

Figure 8,9,10 highlights the differences between traditional VMs and containers (as a contrast to full virtualization).

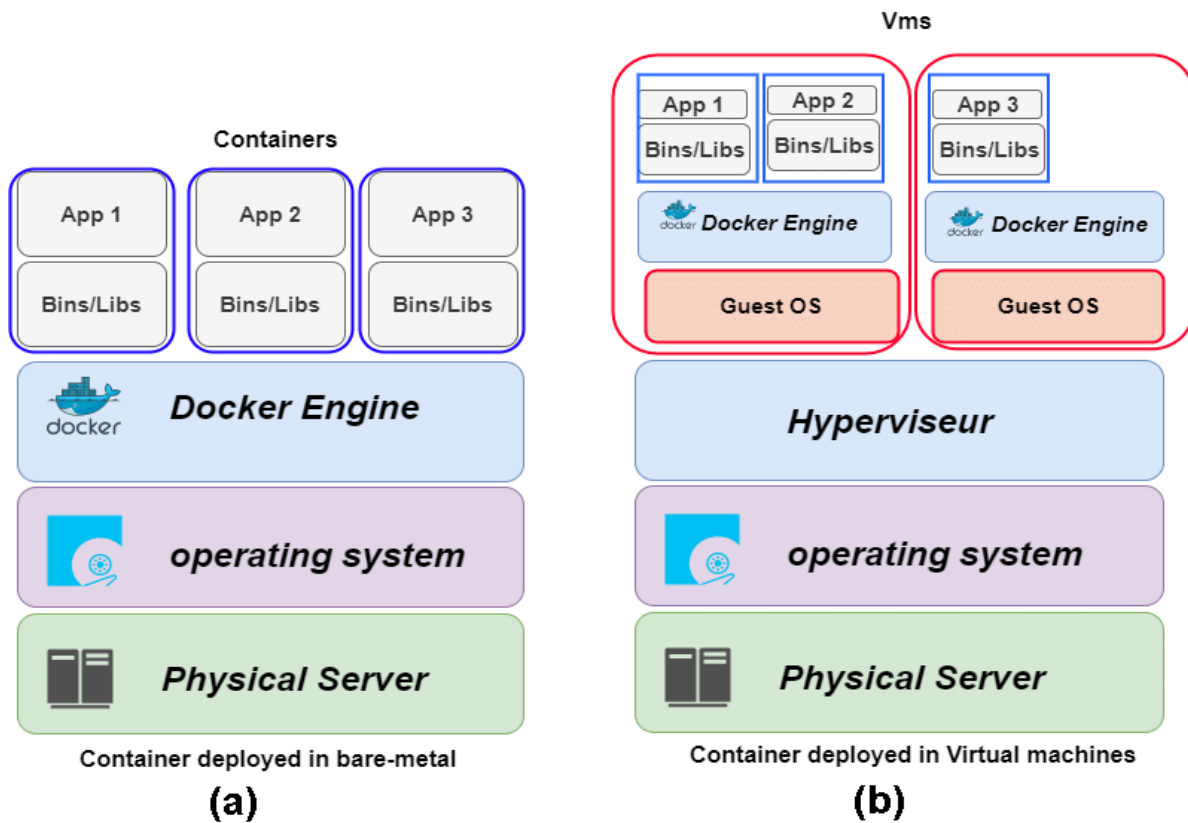


Figure 8 Container in bare-metal and VM

[researchgate.net](https://www.researchgate.net)

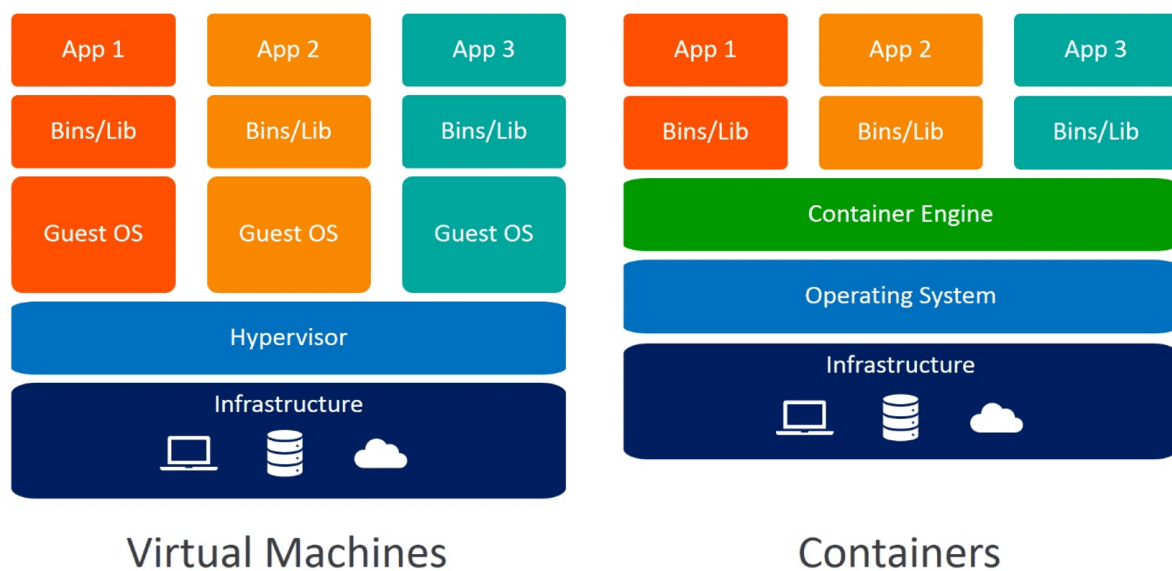


Figure 9 Virtual Machine and Containers

blog.equinix.com

Virtualization vs Containerization

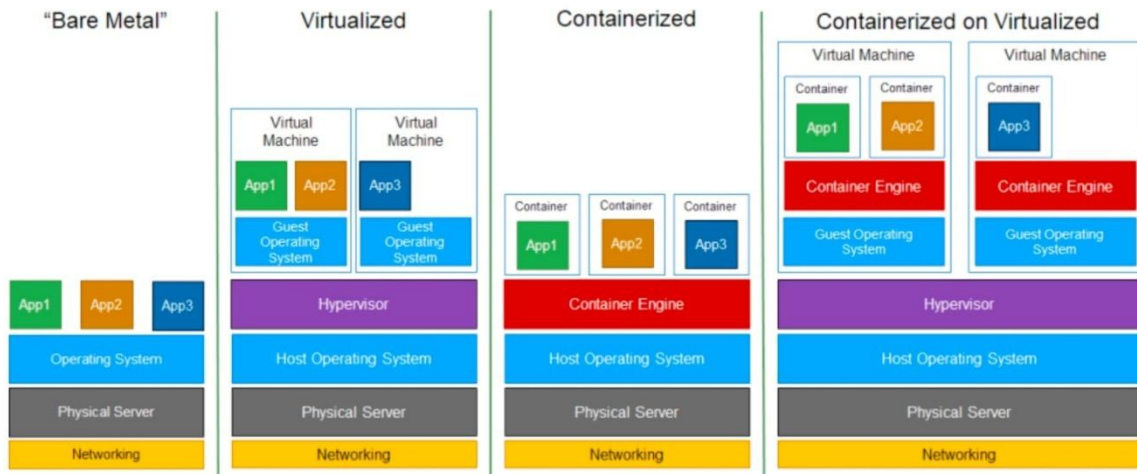


Figure 10: Virtualization vs Containerization

blog.bytebytego.com

In cloud computing, virtualization underpins Infrastructure as a Service (IaaS) models (e.g., AWS EC2, Azure VMs), enabling on-demand resource provisioning and forming the basis for more efficient technologies like containerization discussed later.

2.2 Principles and Benefits of Containerization

Containerization is a modern virtualization technique that allows developers to package an application along with its dependencies, libraries, runtime, and configuration files into a single, lightweight unit called a **container**. Unlike traditional virtualization, which virtualizes hardware to run multiple full operating systems, containerization operates at the operating system level. Containers share the host system's kernel but isolate the application's processes, filesystem, and network using kernel features such as namespaces, cgroups (control groups), and union filesystems.

This shared-kernel approach is fundamental to containerization's efficiency. **Namespaces** provide isolation for resources like process IDs (PID), network interfaces, mount points, and user IDs, ensuring that containers see only their own environment. **Cgroups** limit and monitor resource usage (CPU, memory, I/O), preventing one container from monopolizing the host. **Union filesystems** (e.g., OverlayFS) enable efficient layering for images, allowing multiple containers to share common base layers while maintaining their unique changes.

Key Principles of Containerization

1. **Isolation:** Each container runs in its own isolated environment, preventing conflicts between applications. This is achieved without the overhead of a full guest OS.
2. **Immutability:** Containers are typically immutable—once built, the image doesn't change. Updates involve creating new images, promoting reliable deployments.
3. **Portability:** Containers encapsulate everything needed to run the application, making them runnable on any system with a compatible container runtime (e.g., Docker Engine), regardless of underlying infrastructure.
4. **Layering:** Images are composed of read-only layers, each corresponding to a Dockerfile instruction. This enables caching and reuse, speeding up builds.
5. **Lightweight Nature:** By sharing the host kernel, containers avoid duplicating OS components, resulting in smaller footprints and quicker operations.

Figure 11 demonstrates the layered structure of Docker images, showing how instructions build upon each other.

docs.docker.com

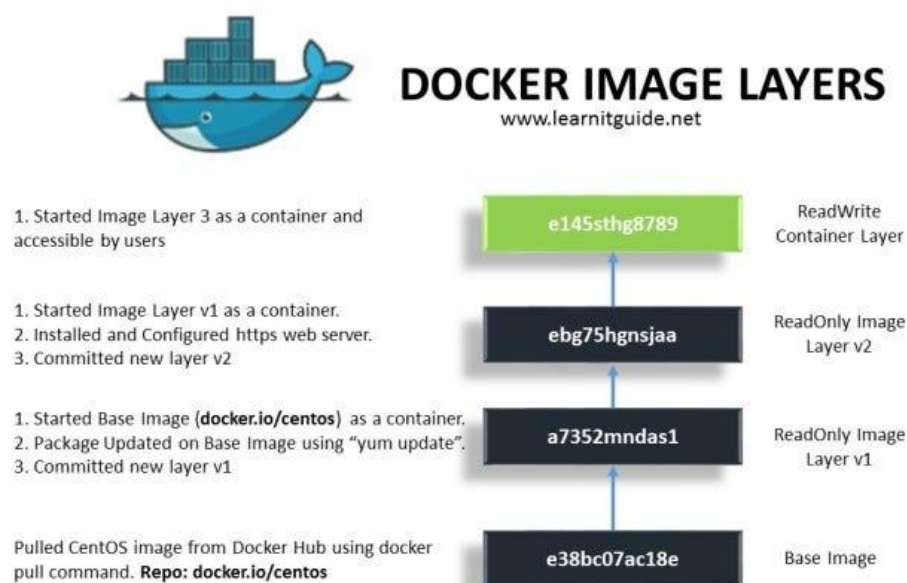


Figure 11: Layered structure of Docker images

medium.com

Benefits of Containerization

Containerization offers significant advantages, particularly in cloud environments where agility and efficiency are paramount:

- **Faster Startup Times:** Containers start in milliseconds (vs. minutes for VMs) since they don't boot a full OS.
- **Reduced Resource Usage:** Lower overhead means higher density—dozens or hundreds of containers can run on a single host compared to fewer VMs.
- **Consistent Environments:** Eliminates "it works on my machine" issues by ensuring identical runtime across development, testing, and production.
- **Improved Scalability and Orchestration:** Containers integrate seamlessly with tools like Kubernetes for automated scaling and management.
- **Enhanced Security:** Isolation reduces attack surfaces, though shared kernel risks exist (mitigated by tools like seccomp and AppArmor).
- **Cost Efficiency:** Better resource utilization lowers infrastructure costs in the cloud.
- **Faster Development and Deployment:** Supports CI/CD pipelines, enabling rapid iterations.

Figure 12 provides an infographic summarizing these benefits, contrasting containers with traditional approaches.

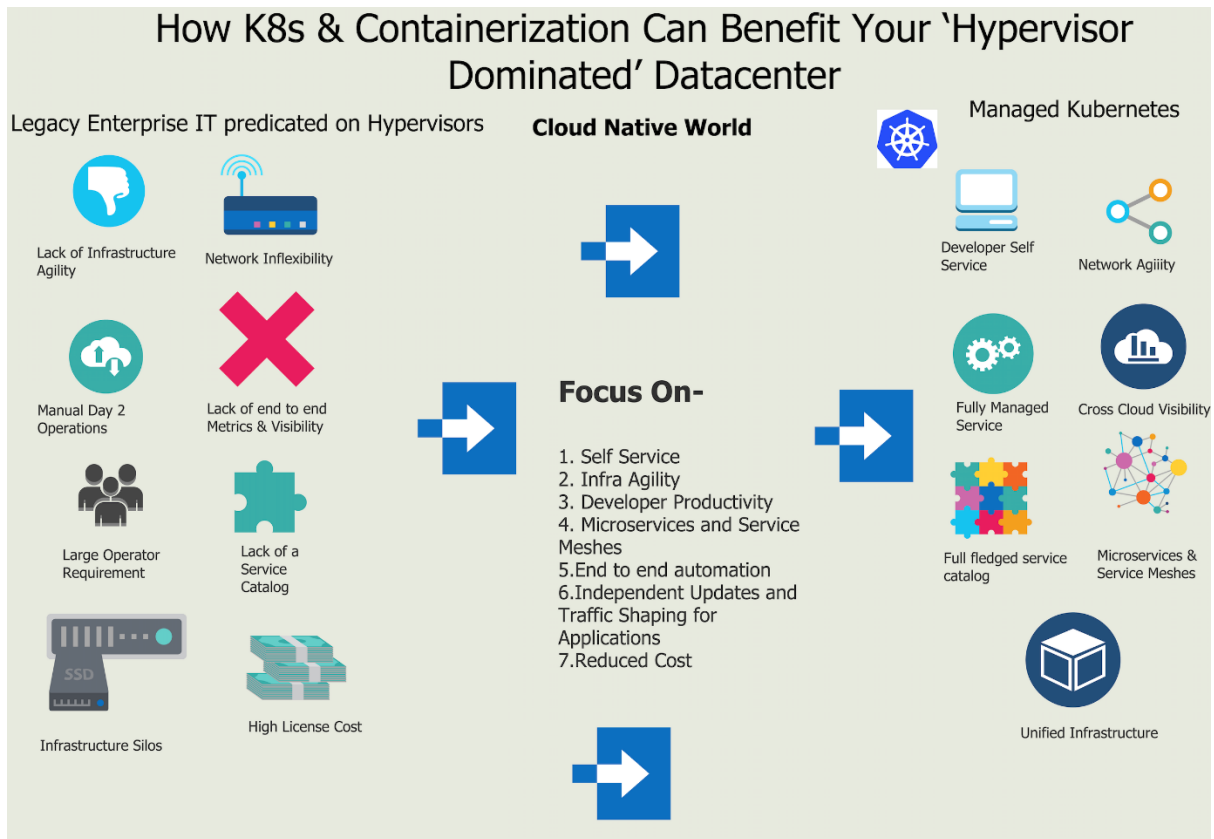


Figure 12: Infographic summarizing benefits, contrasting containers with traditional approaches.

vamsitalkstech.com

INFOGRAPHIC: How K8s & Containerization Benefits Legacy ...

Example: Constructing a Dockerfile for a Node.js Application

To demonstrate these principles in practice, consider building a container for a simple Node.js web server. The Dockerfile defines the container's environment step-by-step:

```
# Start from an official base image with Node.js 14
FROM node:14-alpine

# Set the working directory inside the container
WORKDIR /app

# Copy package.json and package-lock.json first for better caching
COPY package*.json ./

# Install dependencies (cached if package files unchanged)
RUN npm install

# Copy the rest of the application code
COPY . .
```

```
# Expose the port the app runs on
```

```
EXPOSE 3000
```

```
# Define the command to run the application
```

```
CMD ["node", "server.js"]
```

Explanation of Layers and Efficiency:

- **FROM** : Base layer (shared across many images).
- **COPY package*.json** and **RUN npm install** : Dependency layer (cached during rebuilds if unchanged).
- **COPY . .** : Application code layer.

Build and Run Commands:

- Build the image: `docker build -t myapp:1.0 .`
- Run the container: `docker run -d -p 3000:3000 --name mynodeapp myapp:1.0`

This process showcases layering (caching speeds rebuilds), portability (runs anywhere with Docker), and isolation (Node.js app confined to its container).

Figure 13 illustrates the typical containerization workflow: build, run, and push.

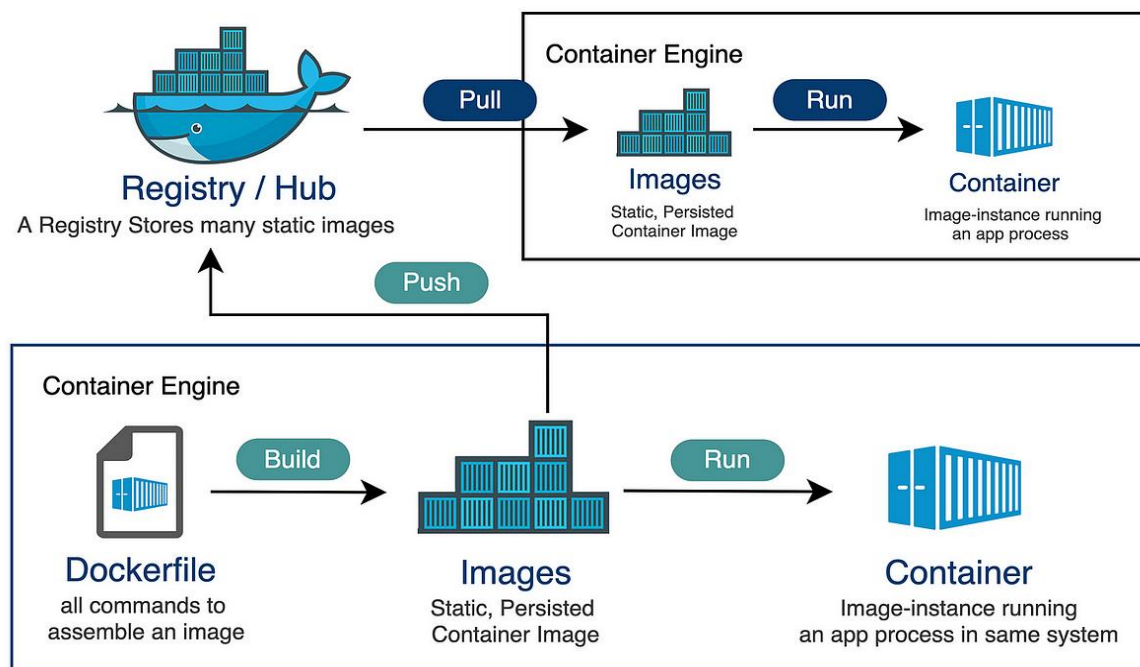


Figure 13: typical containerization workflow: build, run, and push

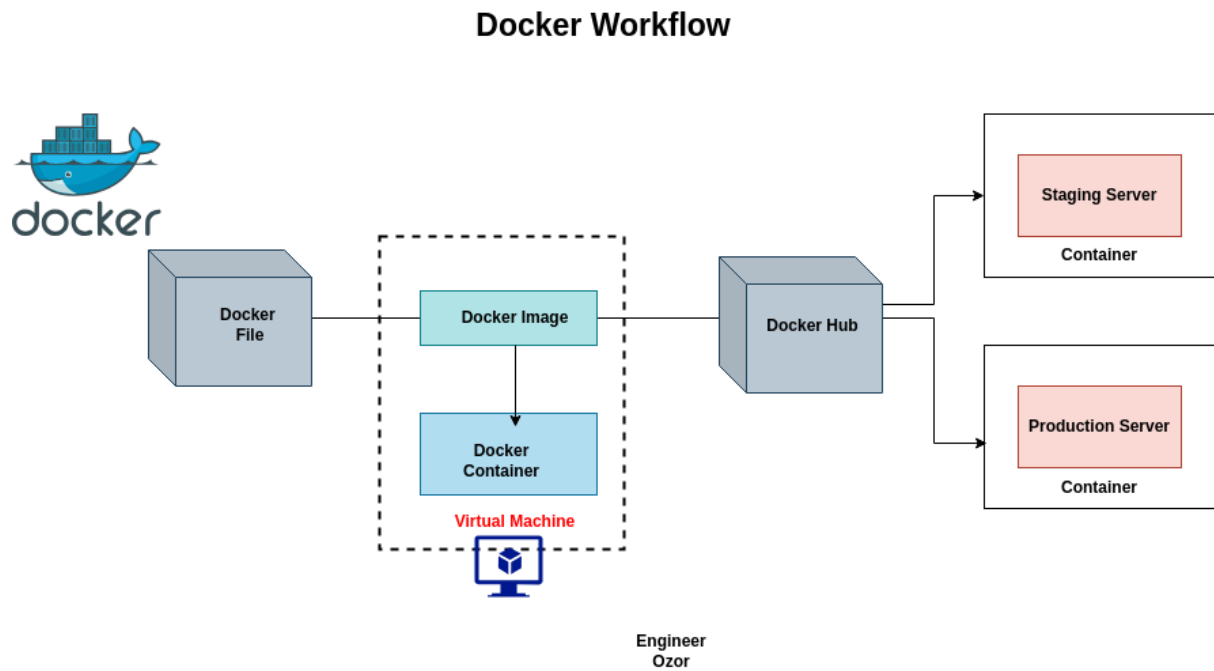


Figure 14 : Docker Workflow

medium.com

In summary, containerization builds on virtualization principles but offers a more efficient, cloud-optimized alternative, driving the adoption of microservices and DevOps practices.

2.3 Images and Publishing Images

Docker images are the cornerstone of containerization, serving as immutable, portable, and self-contained blueprints for running containers. An image is essentially a read-only template that bundles the application code, runtime, system libraries, dependencies, configuration files, and any other components required to execute the application. Once built, a Docker image cannot be modified directly—any changes require creating a new image. This immutability ensures consistency, reproducibility, and reliability across different environments, which is particularly valuable in cloud deployments where applications must behave identically in development, testing, staging, and production.

Images are constructed from a series of layered filesystems using a union mount mechanism (e.g., OverlayFS or AUFS). Each instruction in a Dockerfile (e.g., **FROM**, **RUN**, **COPY**) creates a new layer, and these layers are cached to optimize subsequent builds. If a layer hasn't changed, Docker reuses the cached version, significantly speeding up the build process. The final image is a stack of these layers, with the top layer being writable when a container is instantiated from the image.

Key Characteristics of Docker Images

- **Immutability:** Once built, images are read-only. This prevents accidental modifications and enables version control.
- **Versioning:** Images are typically tagged (e.g., :latest, :1.0, :stable) to manage different versions.
- **Portability:** Images can be run on any Docker-compatible host, regardless of the underlying OS (as long as the kernel is compatible).
- **Efficiency:** Layer sharing reduces storage and transfer overhead—multiple images can reuse the same base layers (e.g., node:14-alpine).

Building and Managing Images Locally

Images are created using the `docker build` command with a Dockerfile. For example, building the Node.js app from Section 2.2:

```
Bash
docker build -t myapp:1.0 .
```

This command:

- Executes the Dockerfile instructions.
- Creates intermediate layers and caches them.
- Tags the resulting image as `myapp:1.0`.

To view built images:

```
Bash
docker images
```

Common commands for local management:

- List images: `docker images`
- Remove an image: `docker rmi myapp:1.0`
- Inspect image details: `docker inspect myapp:1.0`
- View image history (layers): `docker history myapp:1.0`

Publishing Images to a Registry

Publishing makes images shareable across teams, CI/CD pipelines, or cloud platforms. The most popular public registry is **Docker Hub**, but private registries (e.g., AWS ECR, Google Artifact Registry, Azure Container Registry) are common in enterprise settings.

Step-by-Step Publishing Process:

1. **Log in to the Registry:**

```
Bash
docker login
```

(For Docker Hub; use registry-specific commands for private ones, e.g., `aws ecr get-login-password` for AWS).

2. **Tag the Image:** Tagging creates a new reference pointing to the same image ID, formatted as `registry/username/repository:tag`. The latest tag is conventional for the most recent version.

```
Bash
docker tag myapp:1.0 username/myapp:latest
docker tag myapp:1.0 username/myapp:1.0 # Specific version tag
```

Multiple tags can point to the same image for flexibility.

3. **Push the Image:**

```
Bash
docker push username/myapp:latest
docker push username/myapp:1.0
```

This uploads all unique layers to the registry. Subsequent pushes of images sharing layers are faster.

4. **Pull and Use the Published Image:** On another machine or in a cloud service:

```
Bash
docker pull username/myapp:latest
docker run -d -p 3000:3000 username/myapp:latest
```

Best Practices for Publishing Images

- **Use Specific Tags:** Avoid relying solely on :latest; use semantic versioning (e.g., :1.2.3) for reproducibility.
- **Minimize Image Size:** Use multi-stage builds and lightweight base images (e.g., Alpine) to reduce attack surface and improve transfer speed.
- **Scan for Vulnerabilities:** Tools like Docker Scout or Trivy scan images before publishing.
- **Automate with CI/CD:** Integrate publishing into pipelines (e.g., GitHub Actions) for automated builds on commits.
- **Access Control:** For private registries, configure IAM policies to restrict who can push/pull.

Publishing images to a registry is a critical step in cloud-native workflows, enabling collaboration, orchestration (e.g., with Kubernetes), and deployment to cloud platforms like AWS ECS, Google Kubernetes Engine, or Azure Container Instances. This process transforms a locally built image into a globally accessible artifact, fully realizing the portability promised by containerization.

2.4 Container Volumes

Volumes ensure data persistence. Create: `docker volume create myvolume`. Mount: `-v myvolume:/data`.

One of the fundamental characteristics of containers is their **ephemeral nature**. When a container is stopped or removed, any data written to its writable layer (the thin top layer created when the container starts) is lost unless explicitly persisted. This behavior is intentional—it promotes immutability and stateless design—but many real-world applications require persistent storage for databases, user uploads, logs, configuration files, or cached data. Docker provides **volumes** as the preferred mechanism to achieve data persistence, decoupling data from the container lifecycle.

Docker volumes are managed storage locations on the host that can be mounted into containers. They bypass the container's union filesystem for the mounted paths, allowing data to survive container restarts, updates, or deletions. Volumes are fully managed by Docker, offering better performance and additional features compared to alternative approaches like bind mounts.

Types of Docker Volumes and Mounts

Docker offers several ways to persist data:

1. **Volumes** (Recommended):

- Stored in a Docker-managed area on the host (typically `/var/lib/docker/volumes/` on Linux).
- Created and managed explicitly by Docker.
- Best performance and features (backups, migration, encryption).

2. **Bind Mounts**:

- Map a specific host path or file directly into the container.
- Useful when the host path is known and controlled (e.g., sharing source code).

3. **tmpfs Mounts**:

- Stored in memory (not persistent on disk).
- For temporary, sensitive data (e.g., secrets).

Volumes are generally preferred over bind mounts for production because they are portable, easier to back up, and work seamlessly with Docker Compose and Swarm.

Creating and Managing Volumes

Create a Named Volume:

Bash

```
docker volume create myvolume
```

This creates a volume named myvolume in Docker's storage directory.

List Volumes:

Bash

```
docker volume ls
```

Inspect a Volume:

Bash

```
docker volume inspect myvolume
```

Output includes the mount point on the host (e.g., `/var/lib/docker/volumes/myvolume/_data`).

Remove a Volume (only if no containers are using it):

Bash

```
docker volume rm myvolume
```

Remove Unused Volumes:

Bash

docker volume prune

Mounting Volumes into Containers

Volumes are mounted using the `-v` or `--mount` flag (`--mount` is preferred for new applications as it is more explicit).

Using `-v` (short syntax):

Bash

```
docker run -d \  
  --name myapp-with-volume \  
  -v myvolume:/app/data \  
  nginx:alpine
```

Using `--mount` (long syntax – recommended):

Bash

```
docker run -d \  
  --name myapp-with-volume \  
  --mount type=volume,source=myvolume,target=/app/data \  
  nginx:alpine
```

- `source/myvolume`: Name of the volume.
- `target//app/data`: Path inside the container where the volume is mounted.

Data written to `/app/data` inside the container is stored in the volume and persists even if the container is removed.

Practical Example: Persistent Database

Consider a PostgreSQL database container:

Bash

```
# Create a volume for PostgreSQL data  
docker volume create pgdata  
  
# Run PostgreSQL with persistent storage  
docker run -d \  
  --name my-postgres \  
  -e POSTGRES_PASSWORD=mypassword \  
  postgres
```

```
-v pgdata:/var/lib/postgresql/data \  
postgres:15
```

Even if the container is stopped, removed, and recreated with the same volume, all databases and tables remain intact.

Sharing Volumes Between Containers

Multiple containers can mount the same volume, enabling data sharing:

Bash

Container 1: Write data

```
docker run --rm \  
-v myvolume:/data \  
busybox sh -c "echo 'Hello from Container 1' > /data/message.txt"
```

Container 2: Read data

```
docker run --rm \  
-v myvolume:/data \  
busybox cat /data/message.txt
```

Output: Hello from Container 1

This is useful for scenarios like a web application and a backup service accessing the same files.

Volume Drivers and Cloud Integration

Volumes can use **drivers** for advanced functionality:

- Local (default)
- Cloud providers (e.g., aws, gce, azure) for distributed or highly available storage.
- Network filesystems (e.g., NFS, Ceph).

Example with a third-party driver (requires plugin installation):

Bash

```
docker volume create --driver vieux/sshfs \  
-o sshcmd=user@host:/home/user \  
sshvolume
```

In cloud environments, volumes integrate with managed services (e.g., AWS EBS, Azure Files) for automatic backups and scalability.

Best Practices

- Always use named volumes rather than anonymous ones (*-v /path* without a name creates hidden volumes).
- Back up volumes regularly using *docker run --rm -v myvolume:/data -v \$(pwd):/backup busybox tar cvf /backup/backup.tar /data*.
- Use Docker Compose for declarative volume management in multi-container apps.
- Avoid storing large amounts of application code in volumes—keep code in images for immutability.

Docker volumes are essential for building stateful applications in containerized environments, ensuring data durability while maintaining the flexibility and portability that make containers powerful in cloud deployments.

2.5 Container Networking

Container networking is a critical aspect of Docker that determines how containers communicate with each other, with the host system, and with external networks. By default, Docker provides isolation for containers, but real-world applications often require controlled connectivity—for example, a web server container needing to communicate with a database container, or exposing services to the outside world. Docker's networking subsystem, based on the Container Network Model (CNM), uses **drivers** to implement different network types, allowing flexible configurations for various use cases.

Networking in Docker leverages Linux kernel features like network namespaces, virtual Ethernet devices (veth pairs), and bridges to create isolated yet connectable environments. Containers on the same user-defined network can communicate seamlessly using service names as hostnames (built-in DNS resolution), while port mapping exposes container ports to the host or external clients.

Figure 15 provides an overview of the main Docker networking modes, illustrating how they differ in isolation and connectivity.

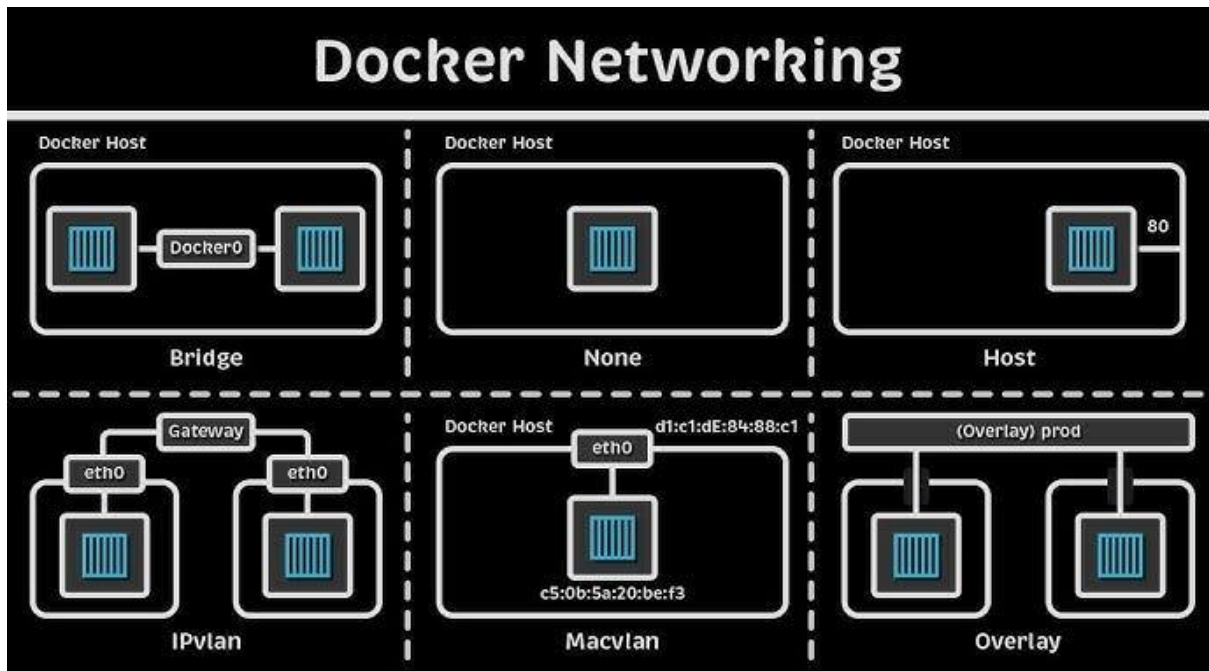


Figure 15: Docker Networking

blog.devgenius.io

taikun.cloud

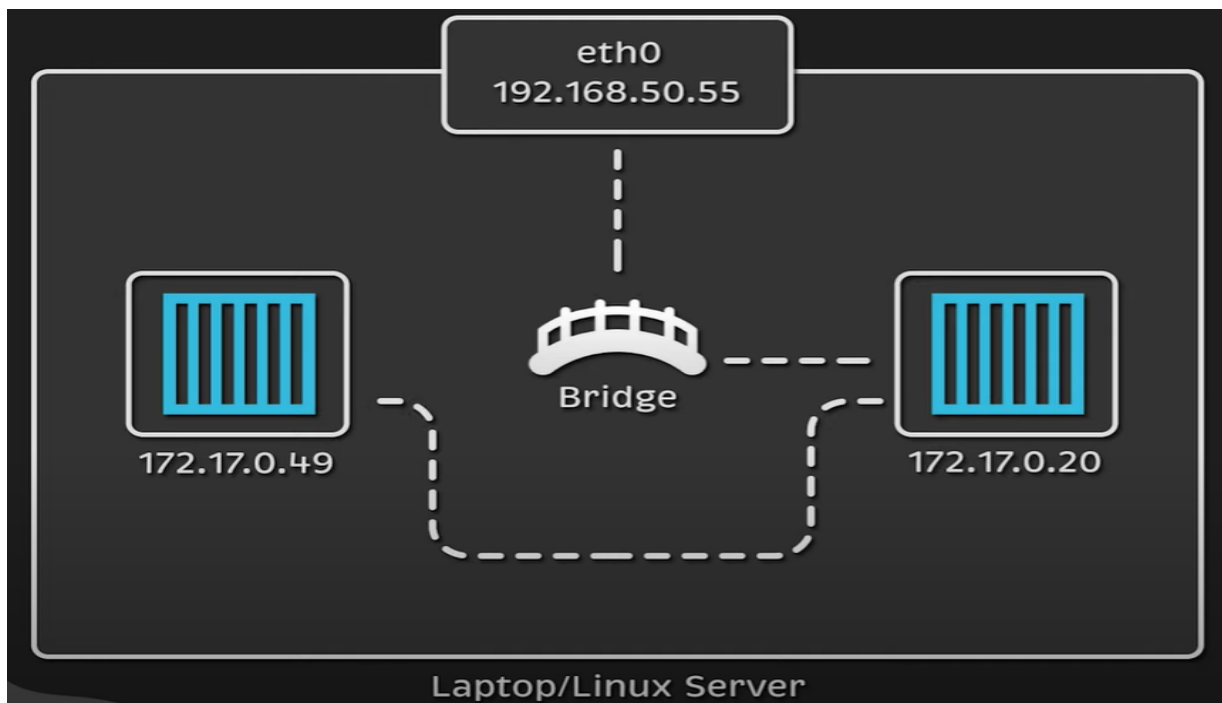


Figure 16: Laptop/ Linux Server data flow

kenbinlab.com

Main Docker Network Drivers

Docker supports several built-in network drivers:

1. Bridge (Default):

- Creates a private internal network on the host using a Linux bridge (docker0 by default).
- Containers attach to this bridge via virtual Ethernet interfaces.
- Containers can communicate with each other on the same bridge network.
- External access requires port mapping (-p host_port:container_port).
- Ideal for standalone containers or simple multi-container apps on a single host.

Figure 17 shows the architecture of the default bridge network.

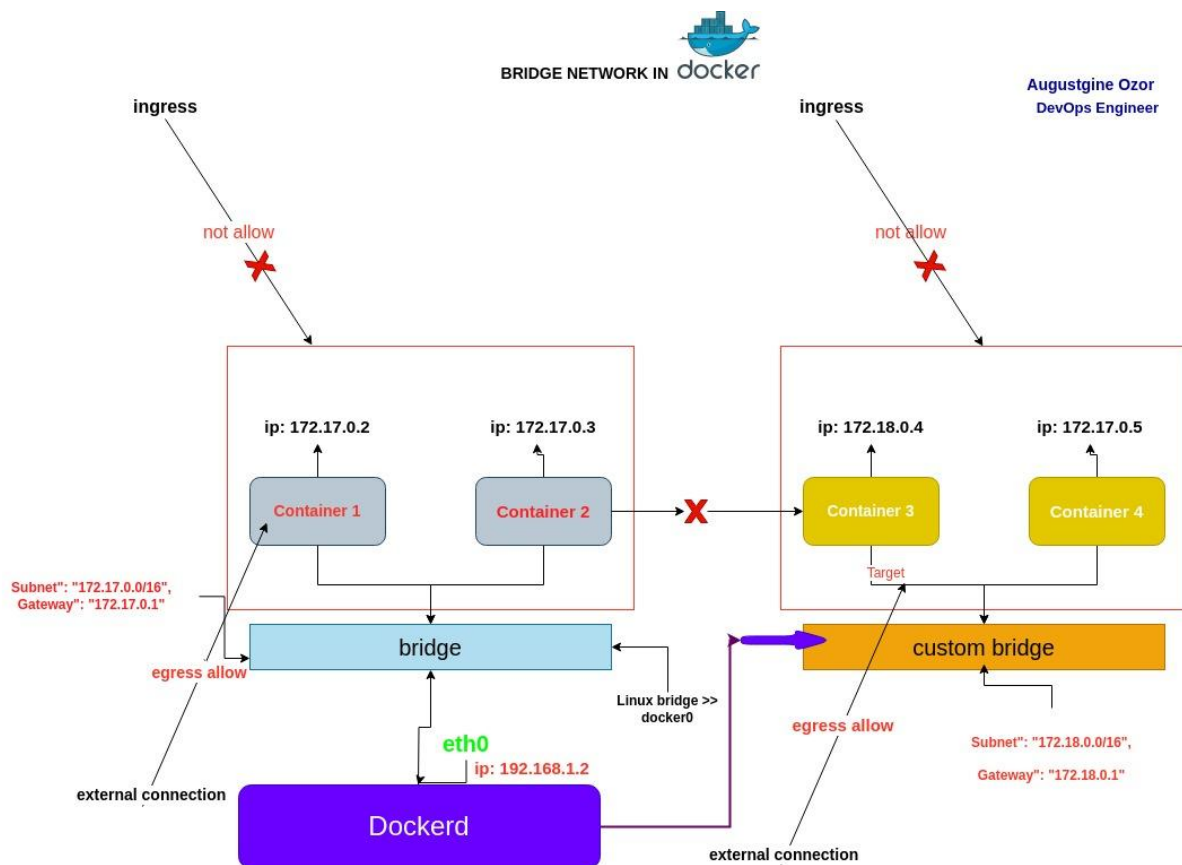


Figure 17 : architecture of the default bridge network.

medium.com

Docker Bridge Networking in Linux

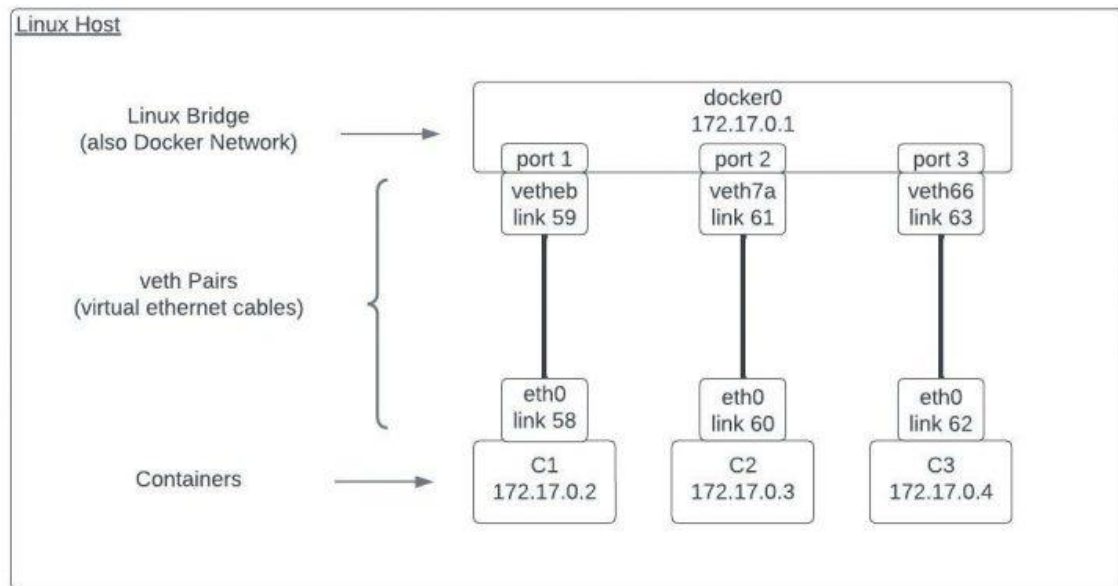


Figure 18 Docker Bridge Networking in Linux

blogs.cisco.com

2. **Host:**

- Removes network isolation; the container uses the host's network stack directly.
- No port mapping needed—container ports are host ports.
- Highest performance but least isolation (containers can conflict on ports).

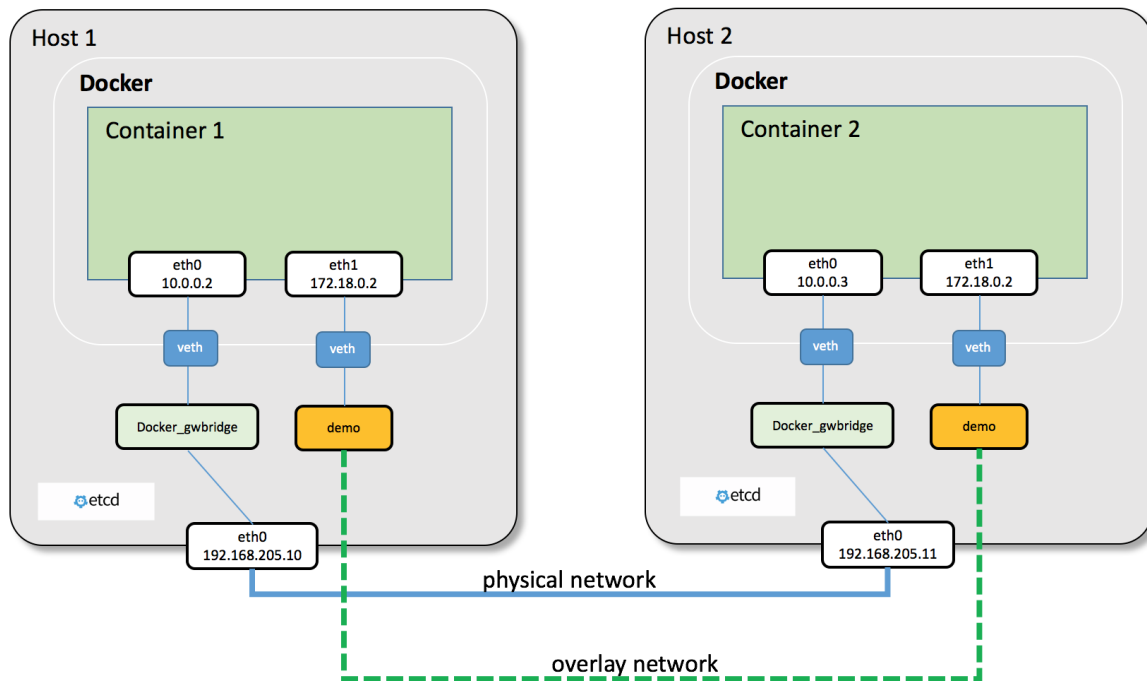
3. **None:**

- Disables all networking; container has only a loopback interface.
- Useful for isolated processes or custom networking.

4. **Overlay:**

- For multi-host networking, typically with Docker Swarm or Kubernetes.
- Uses VXLAN encapsulation to connect containers across hosts securely.
- Supports encrypted communication and service discovery.

Figure 19 illustrates overlay networking in a multi-host setup.



docker-k8s-lab.readthedocs.io

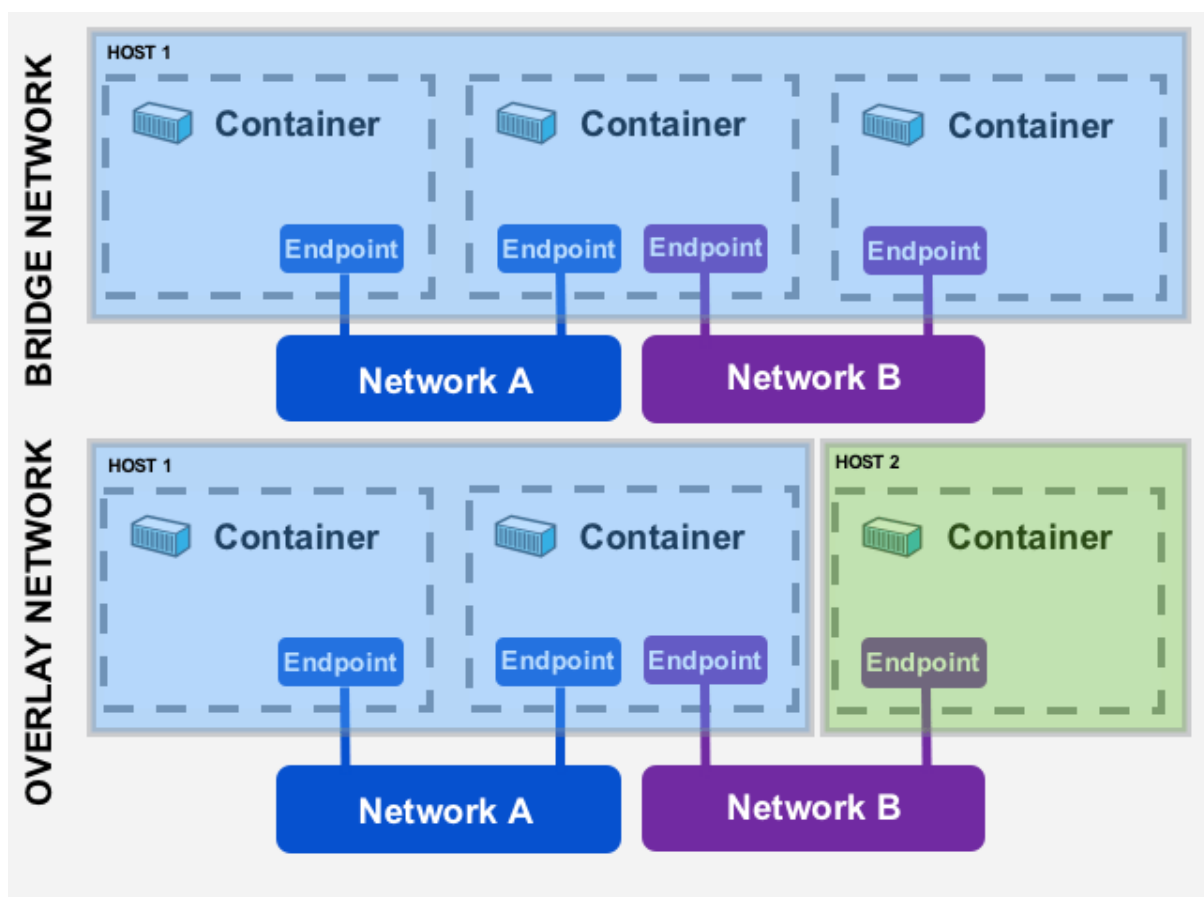


Figure 19: overlay networking in a multi-host setup.

blog.octo.com

5. **Macvlan/IPvlan** (Advanced):

- Allows containers to appear as physical devices on the network with their own MAC/IP addresses.
- Useful for legacy apps needing direct network presence.

Creating and Managing User-Defined Networks

The default bridge network works, but user-defined networks (typically bridge type) offer better features like automatic DNS resolution.

Create a User-Defined Network:

Bash

```
docker network create mynetwork
```

By default, this creates a bridge network.

List Networks:

Bash

```
docker network ls
```

Inspect a Network:

Bash

```
docker network inspect mynetwork
```

Shows connected containers, subnets, etc.

Run Containers on the Network:

Bash

```
docker run -d --name web --network mynetwork nginx:alpine
```

```
docker run -d --name db --network mynetwork postgres:15
```

Containers can now communicate using names:

- From web container: ping db or connect to database at db:5432.

Connect/Disconnect Existing Containers:

Bash

```
docker network connect mynetwork existing-container
```

```
docker network disconnect mynetwork existing-container
```

Remove a Network:

Bash

```
docker network rm mynetwork
```

Practical Example: Multi-Container Application Communication

Consider a web application with Nginx (frontend) and a backend API (Node.js):

1. Create network:

Bash

```
docker network create app-network
```

2. Run backend:

Bash

```
docker run -d --name backend --network app-network myapp:1.0
```

3. Run frontend, linking to backend:

Bash

```
docker run -d --name frontend -p 8080:80 --network app-network nginx:alpine
```

Configure Nginx to proxy to `http://backend:3000`.

Figure 20 demonstrates container-to-container communication on a user-defined network.

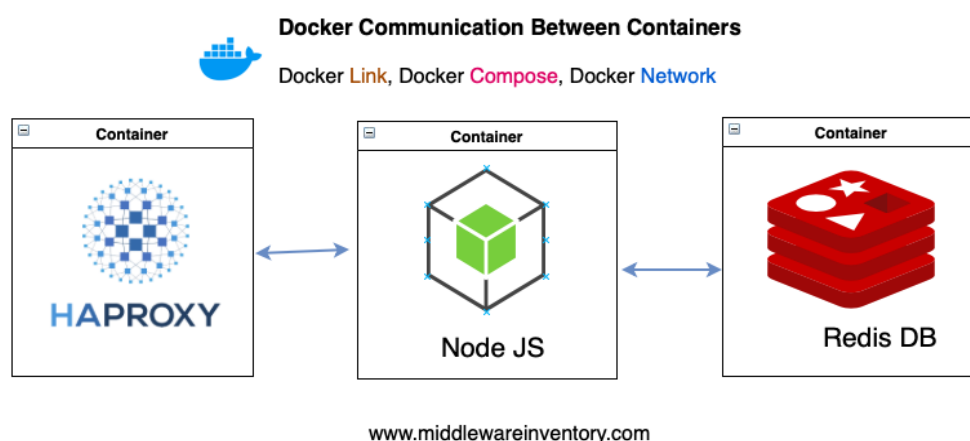


Figure 20 : Container-to-container communication on a user-defined network.

middlewareinventory.com

Best Practices and Cloud Considerations

- Prefer user-defined bridge networks over default for DNS and isolation.
- Use Docker Compose for declarative networking in multi-container apps (defines networks and services).
- In cloud/orchestration (e.g., Kubernetes), overlay networks handle distributed communication.
- Secure networks with firewall rules and avoid host mode in production.

Container networking enables the creation of complex, microservices-based applications while maintaining security and performance, making it indispensable for cloud deployments.

2.6 Overall Process Demonstration: Web Server Example

To consolidate the concepts covered in the previous subsections—virtualization principles, containerization benefits, image building, publishing, volumes, and networking—this section presents a comprehensive, end-to-end demonstration using a practical class-inspired example: deploying a simple yet functional web server that serves a custom HTML page. This example integrates multiple Docker features into a single workflow, mirroring real-world cloud deployment scenarios where a web application must be built, run locally, persisted, networked (if extended), and shared publicly via a registry.

The chosen application is an **Nginx-based web server** using the lightweight `nginx:alpine` base image. Nginx is a high-performance web server commonly used in production environments for serving static content, reverse proxying, and load balancing. We will customize it to serve a personal HTML page, demonstrating how containerization enables rapid, consistent deployment.

Step-by-Step Preparation

1. **Create Project Directory:** Create a new directory for the project and navigate into it:

Bash

mkdir mywebserver

cd mywebserver

2. **Create the Custom HTML File (index.html):** This file will be served as the homepage. It includes basic styling and content to verify successful deployment.

HTML

```
<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">

  <meta name="viewport" content="width=device-width, initial-scale=1.0">

  <title>My Docker Web Server</title>

  <style>

    body { font-family: Arial, sans-serif; background: #f0f8ff; color: #333; text-align: center; padding: 50px; }

    h1 { color: #0066cc; }

    p { font-size: 1.2em; }

    .container { max-width: 800px; margin: 0 auto; background: white; padding: 30px; border-radius: 10px; box-shadow: 0 0 10px rgba(0,0,0,0.1); }

  </style>

</head>

<body>

  <div class="container">

    <h1>Welcome to My Cloud-Ready Web Server</h1>

    <p>This static page is served from a custom Docker container built with Nginx.</p>

    <p>Demonstrating: Image building, container execution, port mapping, and publishing to Docker Hub.</p>

    <p>Student: Alex Johnson | Date: December 11, 2025</p>

  </div>

</body>

</html>
```

Creating the Dockerfile

The Dockerfile defines the container image:

dockerfile

Use lightweight Alpine-based Nginx image (small footprint, secure)

FROM nginx:alpine

Label for metadata (optional but good practice)

LABEL maintainer="alex.johnson@example.com"

LABEL description="Custom Nginx web server for cloud computing demonstration"

Copy the custom HTML file to Nginx's default HTML directory

COPY index.html /usr/share/nginx/html/index.html

Expose port 80 (standard HTTP port)

EXPOSE 80

Nginx runs as PID 1 by default; no CMD override needed as base image handles it

Key aspects demonstrated:

- **Base Image Selection:** nginx:alpine is minimal (~5-10 MB), illustrating efficient containerization.
- **Customization:** COPY instruction adds our file, showing how images are layered.
- **Port Exposure:** Informs Docker (and orchestration tools) which port to use.

Building the Docker Image

Build the image locally:

Bash

docker build -t mywebserver:1.0 .

- -t: Tags the image with name and version.
- .: Uses the current directory's Dockerfile.

During the build:

- Docker downloads the base image (if not cached).
- Executes each instruction, creating cached layers.

- Outputs the final image ID.

Verify the image:

Bash

```
docker images | grep mywebserver
```

Running the Container Locally

Run the container, mapping host port 8080 to container port 80:

Bash

```
docker run -d \  
--name my-running-web \  
-p 8080:80 \  
mywebserver:1.0
```

- -d: Detached mode (runs in background).
- --name: Assigns a friendly name.
- -p 8080:80: Port mapping (host:container).

Access the server:

- Open a browser and navigate to <http://localhost:8080>.
- The custom HTML page should load successfully.

Check logs and status:

Bash

```
docker logs my-running-web
```

```
docker ps | grep my-running-web
```

To stop and remove:

Bash

```
docker stop my-running-web
```

```
docker rm my-running-web
```

Optional Extension: Adding Persistence with Volumes

Although static content is baked into the image, we can demonstrate volumes by allowing runtime updates to the HTML file:

Bash

Create a volume

docker volume create webcontent

Run with volume mounted over the HTML directory

*docker run -d *

*--name my-persistent-web *

*-p 8080:80 *

*-v webcontent:/usr/share/nginx/html *

mywebserver:1.0

Now, updating files in the volume (via another container or host tools) would reflect live changes without rebuilding.

Optional Extension: Networking (Multi-Container Setup)

To showcase networking, imagine adding a backend API container:

Bash

docker network create web-network

docker run -d --name backend --network web-network node:14-alpine # (simulated backend)

docker run -d --name frontend -p 8080:80 --network web-network mywebserver:1.0

The frontend could proxy to the backend using its container name (backend).

Publishing the Image to Docker Hub

1. Log in:

Bash

docker login

2. Tag for the Registry:

Bash

docker tag mywebserver:1.0 yourusername/mywebserver:1.0

docker tag mywebserver:1.0 yourusername/mywebserver:latest

3. Push:

Bash

docker push yourusername/mywebserver:1.0

docker push yourusername/mywebserver:latest

Once pushed, anyone (or cloud services) can pull and run:

Bash

docker pull yourusername/mywebserver:latest

docker run -d -p 8080:80 yourusername/mywebserver:latest

Why This Example Demonstrates the Overall Process Effectively

- **Integration of Concepts:** Combines image building (Dockerfile layers), container execution (port mapping), persistence (optional volume), and sharing (publishing).
- **Cloud Relevance:** The published image can be deployed directly to cloud platforms (e.g., AWS ECS, Google Cloud Run, Azure Container Instances) without modification.
- **Reproducibility:** Anyone pulling the image gets the exact same web server.
- **Real-World Applicability:** Mirrors production workflows for static sites, frontend apps, or microservices.

2.7 Video Demonstration

The following unlisted YouTube video (under 15 minutes) demonstrates the web server example, including Dockerfile creation, building, running, and pushing to Docker Hub:

3. Load Balancing, Scalability, and Application Gateways



Figure 21: Amazon Web Services Logo

commons.wikimedia.org

3.1 Chosen Application: E-Commerce Website

E-commerce platforms experience variable traffic (e.g., during sales), making them ideal for demonstrating these concepts.

3.2 Cloud Provider's Approach (AWS)

- **Load Balancing** — AWS Elastic Load Balancing distributes traffic across instances.
- **Scalability** — Auto Scaling dynamically adjusts EC2 instance counts.
- **Application Gateways** — API Gateway manages secure API routing and throttling.

Figure 22 below shows a typical architecture combining these services.

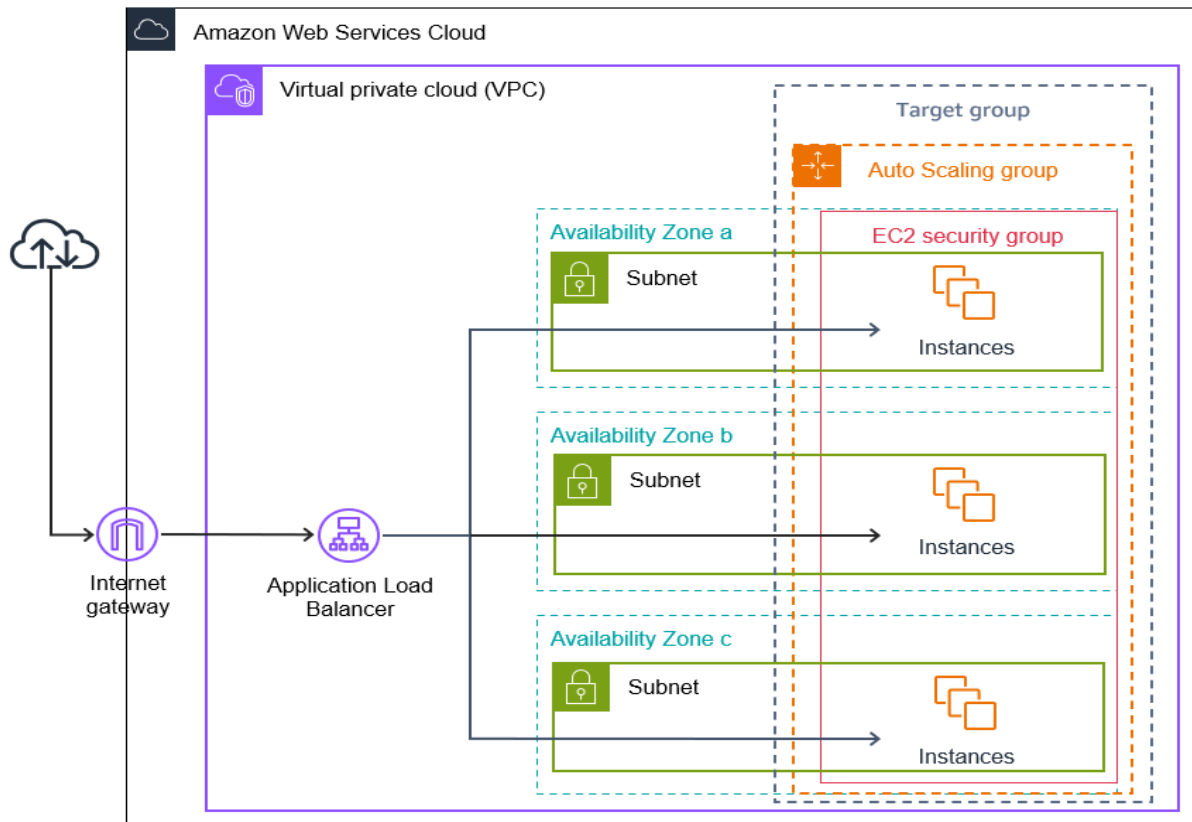


Figure 22: Architecture combining services.

docs.aws.amazon.com

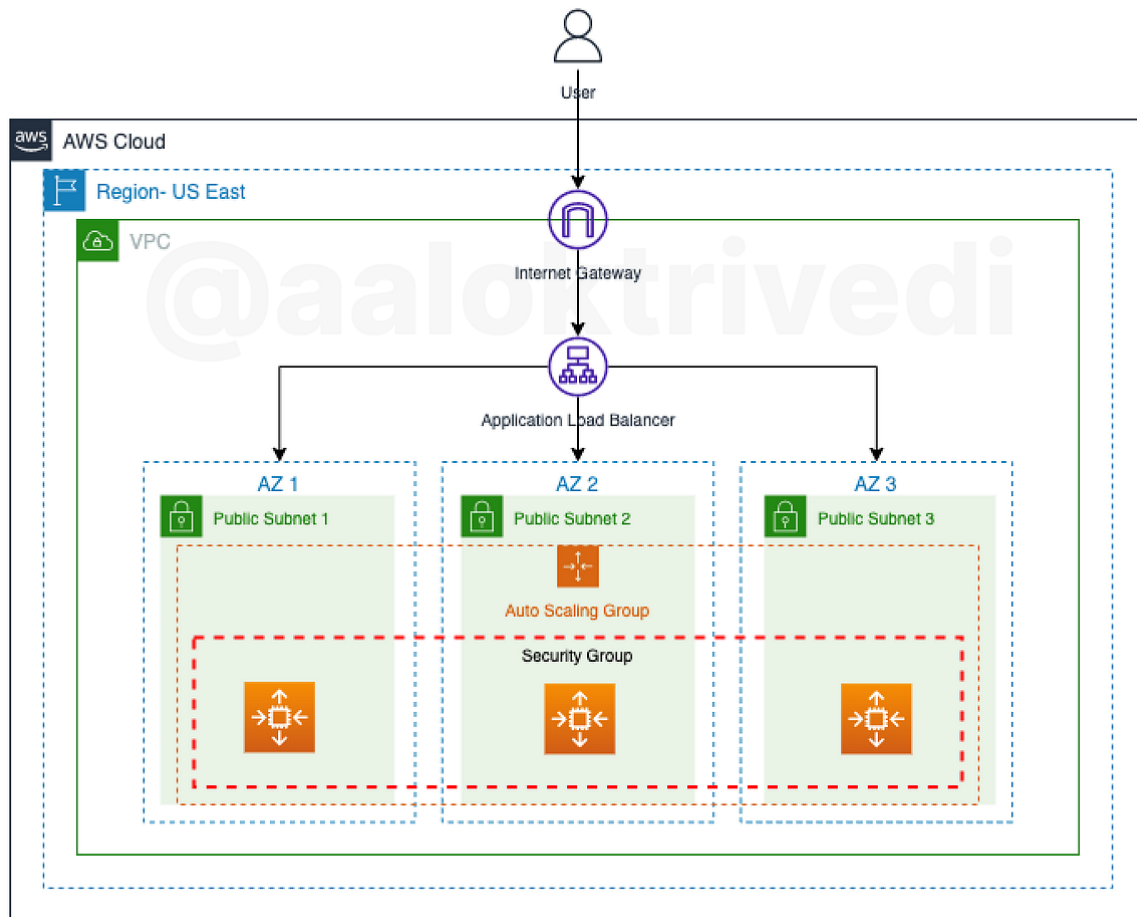


Figure 23 Architecture combining services.

medium.com

3.3 Why Load Balancing and Scaling are Complementary

Load balancing distributes traffic evenly, while scaling adds/removes resources. Together, they ensure optimal performance and resource use during demand fluctuations.

3.4 Benefits of Combining All Three Technologies

Integration provides high availability, performance, security, and cost-efficiency, delivering reliable service even under peak loads.

4. Driving Forces for Cloud Computing Expansion and Serverless Functions

4.1 Main Driving Forces

Primary drivers include cost reduction (pay-as-you-go model) and flexibility (rapid scaling and provisioning).

Figure 24 highlights market growth driven by these factors.

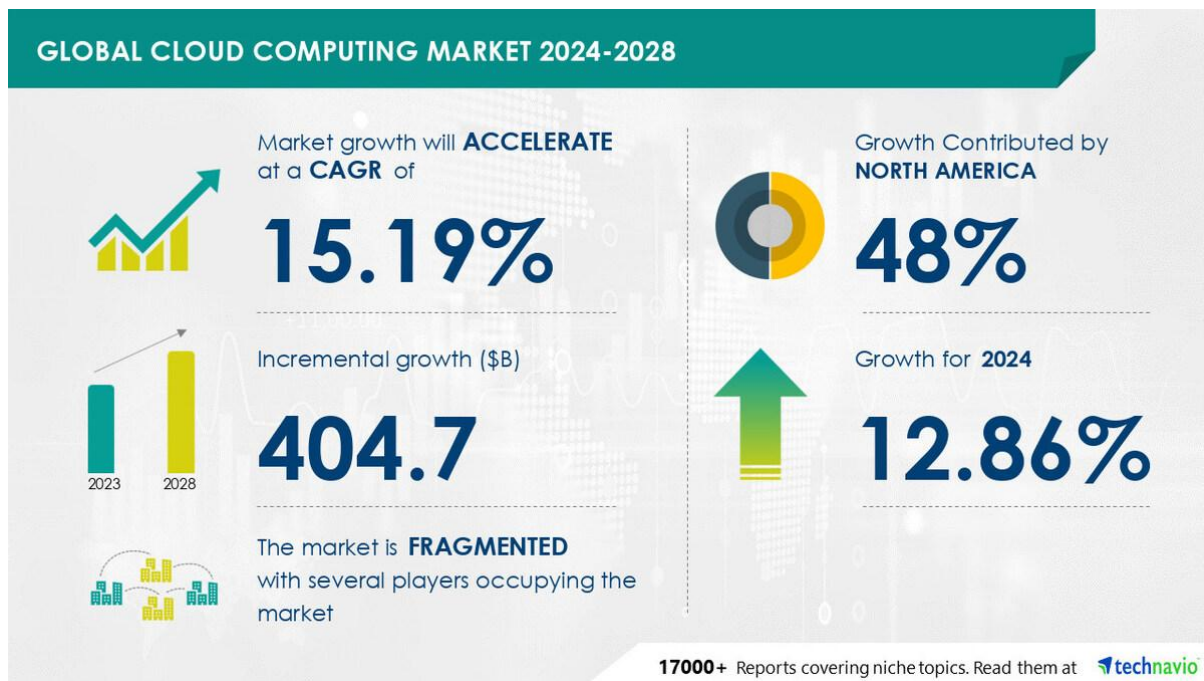


Figure 24 Market growth driven by different factors

4.2 Explanation of Serverless Functions

Serverless (FaaS), such as AWS Lambda, executes code in response to events without server management, with automatic scaling and per-use billing.

4.3 Comparison with Containerization

Both enable scalable, event-driven apps, but serverless eliminates infrastructure management and offers finer billing, while containers provide greater control for long-running workloads.

Figure 24 compares the two approaches.

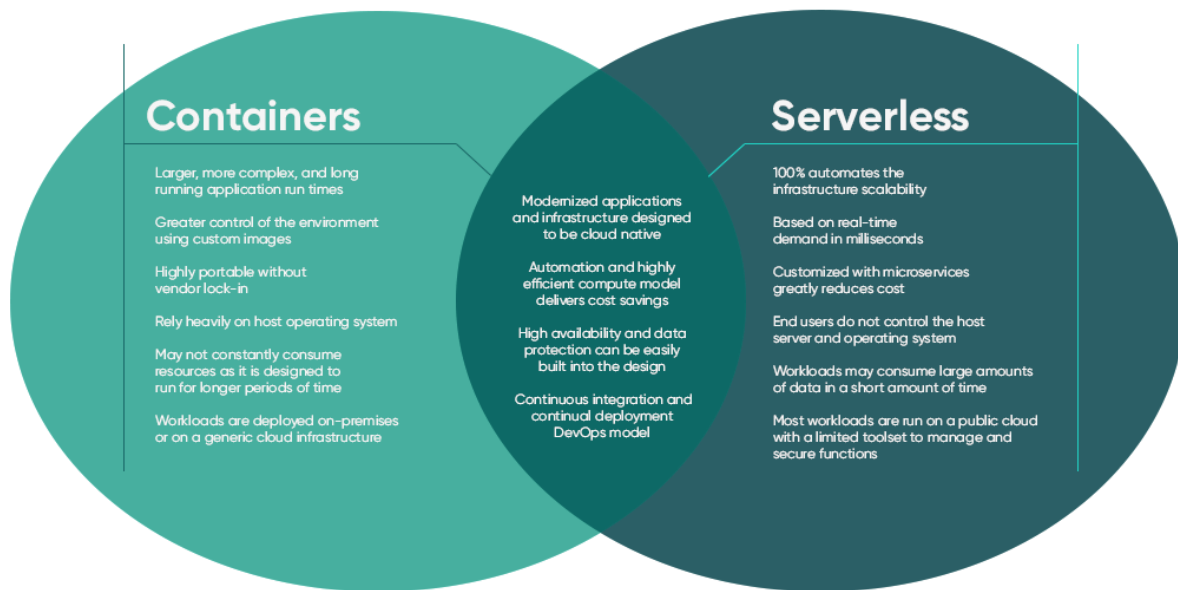


Figure 25 Compares the two approaches

controlplane.com

5. Conclusion

In conclusion, this report has provided a comprehensive exploration of Docker containerization and its pivotal role in modern cloud computing, while addressing the broader ecosystem of cloud technologies as outlined in the assignment. As of December 11, 2025, the cloud computing landscape continues to evolve rapidly, with containerization remaining a foundational element for building efficient, portable, and scalable applications. Through detailed examination of virtualization principles—such as hypervisor-managed resource abstraction and the trade-offs between Type 1 and Type 2 hypervisors—the report established the groundwork for understanding containerization's advantages. Containerization, leveraging kernel-level isolation via namespaces and cgroups, offers a lightweight alternative that addresses virtualization's overhead, enabling faster startups, higher density, and seamless portability across environments.

The practical sections on Docker features further illustrated these benefits: image construction and layering for immutability and caching; publishing to registries like Docker Hub for collaborative workflows; volumes for decoupling persistent data from ephemeral containers; and networking drivers (bridge, overlay, etc.) for robust inter-container communication in distributed systems. The integrated web server demonstration in Section 2.6 exemplified the end-to-end process—from creating a Dockerfile and custom HTML page, building and running the Nginx container with port mapping, optionally adding volumes for persistence and networks for multi-service setups, to tagging and pushing the image to Docker Hub. This workflow not only consolidates class-learned concepts but also showcases Docker's real-world applicability in cloud deployments, where reproducibility and rapid iteration are paramount.

The accompanying video demonstration reinforces this hands-on learning, providing a narrated walkthrough under 15 minutes, accessible via the unlisted YouTube link. Such visual aids bridge theory and practice, ensuring viewers can replicate the steps independently.

Section 3 extended this foundation to core cloud principles using an e-commerce application case study. AWS's Elastic Load Balancing distributes traffic intelligently across targets, Auto Scaling adjusts compute resources dynamically based on metrics like CPU utilization, and API Gateway secures and routes API calls with features like throttling and caching. These are complementary: load balancing optimizes fixed resources while scaling expands them elastically, preventing bottlenecks and ensuring resilience. Their combined benefits—high availability (redundancy across Availability Zones), enhanced performance (reduced latency), robust security (authentication and rate limiting), and cost optimization (pay-for-use scaling)—enable reliable customer experiences during peak loads, such as flash sales or holiday shopping surges.

In Section 4, the analysis of cloud expansion drivers—cost savings from eliminating upfront hardware investments and flexibility for on-demand adaptation—highlighted why adoption is surging, with market projections indicating continued double-digit growth. Serverless functions like AWS Lambda and Azure Functions represent an evolutionary step, abstracting server management for event-driven code execution with per-invocation billing and auto-scaling to zero. Compared to containerization, serverless excels in sporadic, short-lived tasks with zero operational overhead but introduces cold starts and execution limits; containers offer customization and statefulness for persistent workloads. Both align with driving forces, but hybrid approaches (e.g., Lambda with container images) maximize efficiency.

Overall, these technologies form a synergistic stack: Docker's portability powers microservices, cloud primitives like load balancing and scaling ensure operational excellence, and serverless accelerates innovation for edge cases. Challenges remain, such as security vulnerabilities in shared kernels or orchestration complexity, but best practices (scanning tools, least-privilege policies) mitigate them. As organizations embrace hybrid/multi-cloud strategies, mastering these concepts is crucial for sustainable digital transformation.

Looking forward, emerging trends like AI-integrated orchestration (e.g., Kubernetes with ML autoscaling), edge computing with containers, and sustainable cloud practices (energy-efficient serverless) will shape the future. This report, drawing from class exercises and up-to-date sources, equips readers with the knowledge to contribute to resilient, innovative cloud solutions that drive business value and societal progress in an increasingly connected world.

6. References

1. Docker, Inc. (2025). *Docker Documentation*. Available at: <https://docs.docker.com/> (Accessed: 11 December 2025).
2. Amazon Web Services, Inc. (2025). *AWS Documentation – Elastic Load Balancing, Auto Scaling, API Gateway*. Available at: <https://aws.amazon.com/documentation/> (Accessed: 11 December 2025).

3. Backblaze. (2023). 'Virtual Machines vs. Containers: A Closer Look'. Available at: <https://www.backblaze.com/blog/vm-vs-containers/> (Accessed: 11 December 2025).
4. CloudZero. (2024). 'Serverless vs. Containers: When to Use Which'. Available at: <https://www.cloudzero.com/blog/serverless-vs-containers/> (Accessed: 11 December 2025).
5. GeeksforGeeks. (2025). 'Evolution of Cloud Computing'. Available at: <https://www.geeksforgeeks.org/evolution-of-cloud-computing/> (Accessed: 11 December 2025).
6. Grand View Research. (2025). 'Cloud Computing Market Size, Share & Trends Analysis Report'. Available at: <https://www.grandviewresearch.com/industry-analysis/cloud-computing-industry> (Accessed: 11 December 2025).
7. Net Solutions. (2025). 'Containers vs. Virtual Machines: Choosing the Right Virtualization Technology'. Available at: <https://www.netsolutions.com/insights/containers-vs-virtual-machines/> (Accessed: 11 December 2025).
8. Fiveable. (2025). 'Cloud Computing Architecture Layers'. Available at: <https://library.fiveable.me/> (Accessed: 11 December 2025).
9. Medium. (Various authors, 2024–2025). Articles on Docker architecture, networking, and cloud-native practices (specific articles cited in captions). Available at: <https://medium.com/> (Accessed: 11 December 2025).
10. ResearchGate. (Various authors, 2023–2025). Academic diagrams and papers on virtualization, containerization, and cloud evolution. Available at: <https://www.researchgate.net/> (Accessed: 11 December 2025).
11. Dev Genius, taikun.cloud, KenBinLab, Augustine Ozor, Cisco Blogs, Docker Kubernetes Lab Handbook, OCTO Talks, Middleware Inventory (2024–2025). Technical diagrams and tutorials on Docker networking and architecture. (Accessed: 11 December 2025).
12. Vamsi Talks Tech. (2025). 'Benefits of Containerization Infographic'. (Accessed: 11 December 2025).
13. Alibaba Cloud Community. (2025). 'Serverless vs. Traditional Architecture'. (Accessed: 11 December 2025).