Container-Based Application Design

From Kernel Primitives to Cloud Orchestration

Iman Salehi

Course:

Distributed Systems & Cloud Computing

Abstract

This technical report investigates the evolution of virtualization technologies, specifically focusing on the shift from Hypervisor-based Virtual Machines to Containerization. It analyzes the underlying Linux Kernel mechanisms (Namespaces and Cgroups) and details the architecture of the Docker Engine. Furthermore, it presents an engineering approach to designing a Spring Boot microservice, utilizing Multi-stage builds and Docker Compose, before concluding with the necessity of Kubernetes for production-grade orchestration.

December 6, 2025

Contents

1 Theoretical Foundations: The Paradigm Shift

The fundamental challenge in distributed systems is resource isolation and efficiency. While traditional virtualization solved isolation, it introduced significant overhead.

1.1 Virtual Machines vs. Containers

Hypervisor-based virtualization employs a software layer (Hypervisor) to emulate hardware, allowing multiple operating systems to run on a single physical host. This is categorized into two types:

1. **Type 1 (Bare Metal):** The hypervisor runs directly on the hardware without an underlying host OS.
2. **Type 2 (Hosted):** The hypervisor runs as an application on top of a conventional operating system.

In contrast, Containerization utilizes **OS-level virtualization**, where applications share the Host OS kernel but run in isolated user-spaces [1].

Real-World Examples:

- **Type 1 Hypervisors:** VMware ESXi, Microsoft Hyper-V, Xen (Used in enterprise data centers and cloud infrastructure like AWS EC2).
- **Type 2 Hypervisors:** VMware Workstation, Oracle VirtualBox (Used for local development and testing).
- **Container Engines:** Docker, Podman, LXC (Used for microservices and cloud-native applications).

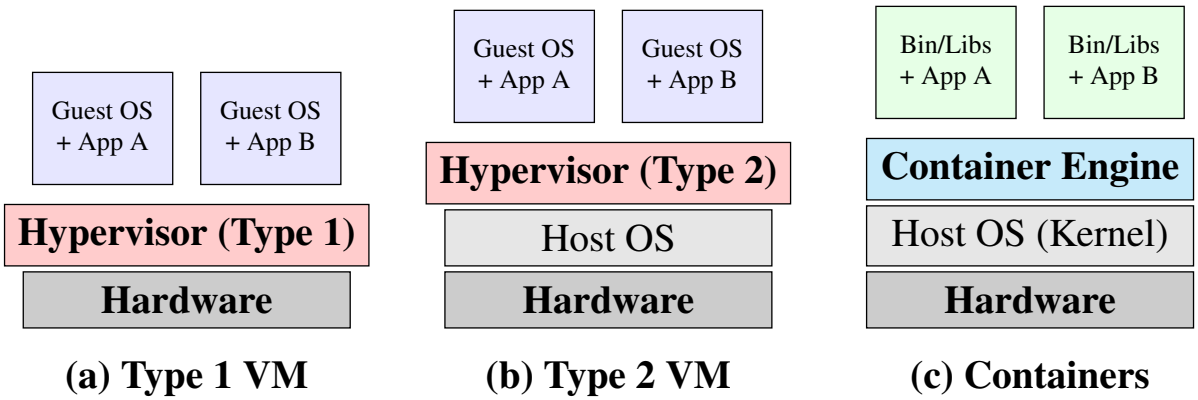

Figure 1: Comparison of Virtualization Architectures: (a) Bare Metal, (b) Hosted, and (c) Container-based.

1.2 Kernel Primitives: Namespaces & Cgroups

To understand *why* containers are efficient, we must look at the Linux Kernel's approach to resource management. Unlike VMs that virtualize hardware, containers virtualize the Operating System via two primitives.

1.2.1 Namespaces: The Logic of Isolation

Why is it needed? In a standard OS, resources like Process IDs (PIDs) and Network Ports are global. If two applications try to bind to port 80, a conflict occurs.
The Solution: Namespaces wrap global system resources in an abstraction layer. This allows processes within a namespace to have their own "view" of the system.

- **PID Namespace:** Remaps process IDs. A process can have PID 1 (root) inside the container while having PID 1534 on the host.
- **NET Namespace:** Virtualizes the network stack. Each container gets its own ‘eth0‘ interface and routing table, preventing port conflicts.

1.2.2 Control Groups (Cgroups): The Logic of Constraints

Why is it needed? Even with isolation, a rogue process could consume 100% of the Host CPU, starving other critical applications. This is known as the *"Noisy Neighbor"* problem.
The Solution: Cgroups (Control Groups) provide resource accounting and limiting. The kernel scheduler checks the Cgroup limits before allocating CPU cycles or memory pages to a process. If a container exceeds its memory limit, the kernel invokes the **OOM Killer** (Out Of Memory) to terminate it, protecting the host [2].

1.2.3 Comparative Analysis: The Windows Approach

Historically, containerization was unique to Linux/Unix due to the `fork()` system call architecture. Microsoft Windows implemented containerization differently in modern versions (Server 2016+):

1. **Process Isolation (Windows Server Containers):** Similar to Linux containers. Applications share the host Windows kernel. This provides speed but lesser security boundaries.
2. **Hyper-V Isolation:** Unlike Linux’s pure shared-kernel approach, Windows often defaults to wrapping containers in a lightweight utility VM (Hyper-V).

   > *"Why does Windows do this?"* The Windows API surface is vast and tightly coupled. Sharing the kernel presents a higher security risk compared to Linux. Therefore, Microsoft emphasizes Hyper-V isolation for multi-tenant security, trading some performance for stronger isolation.

2 Docker Architecture

Docker is not a monolith; it is a modular platform composed of several specialized components. Over the years, it has been refactored into distinct, specialized components to adhere to the Open Container Initiative (OCI) standards. This modularity ensures stability and decoupling.

2.1 The Engine Components

The Docker Engine operates on a client-server architecture, but the execution flow involves several distinct layers:

1. **Docker Daemon (dockerd):** The high-level management process. It listens for API requests, manages images, and handles networking. It does not run containers directly.
2. **containerd:** An industry-standard container runtime (graduated CNCF project). It manages the complete container lifecycle: pulling images from registries, storage management, and supervising execution.
3. **runc:** The low-level CLI tool responsible for spawning and running containers according to the OCI specification. It interacts directly with Kernel Namespaces and Cgroups. Crucially, `runc` exits after the container is created, leaving the process running.
4. **containerd-shim:** A small process that sits between ‘containerd‘ and the running container. It allows ‘runc‘ to exit (daemonless containers) and keeps the container’s STDOUT/STDERR streams open.

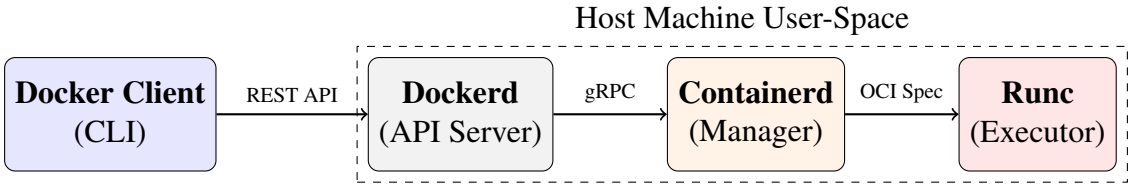

Figure 2: The Docker Execution Flow: From API to Kernel Execution

2.2 The Docker Object Model

Understanding the distinction between an Image and a Container is critical for system design.

2.2.1 Images: The Read-Only Template

A Docker image is an immutable, multi-layered file system. Each instruction in a `Dockerfile` (e.g., `COPY`, `RUN`) creates a new layer. These layers are stacked using a **Union File System** (such as OverlayFS).

- **Content Addressable:** Images are identified by a SHA256 digest of their content, ensuring security and integrity.
- **Layer Caching:** If a layer hasn’t changed, Docker reuses the cached version during the build process, significantly speeding up CI/CD pipelines.

2.2.2 Containers: The Runnable Instance

A container is essentially a Docker Image plus a thin **Read/Write Layer** on top. Any changes made by the running application (writing logs, modifying files) happen in this ephemeral layer. When the container is deleted, this layer is lost, while the underlying image remains untouched.

2.3 Data Persistence Strategies

Since the container's writable layer is ephemeral, Docker provides mechanisms for persistent storage:

- **Volumes:** Managed by Docker (`/var/lib/docker/volumes`). This is the preferred mechanism for databases as it bypasses the storage driver's overhead and is OS-agnostic.
- **Bind Mounts:** Maps a specific file or directory on the Host OS to the container. Ideal for injecting configuration files or source code during development.
- **tmpfs Mounts:** Stored only in the host's memory. Useful for storing sensitive credentials that should never be written to disk.

2.4 The Mechanics of Storage: OverlayFS

Why is Docker so fast? The secret lies in the storage driver. Docker uses Union File Systems (UnionFS) to layer image data. The specific implementation on modern Linux kernels is **OverlayFS**.

OverlayFS takes two directories:

1. **LowerDir (Read-Only):** The base image layers (OS, libraries, app code). These are immutable and shared across all containers derived from this image.
2. **UpperDir (Read-Write):** The container-specific layer.

Copy-on-Write (CoW): When a container attempts to modify a file that exists in the *LowerDir*, Docker does not modify the original. Instead, it copies the file up to the *UpperDir* and modifies the copy. The original file remains hidden "underneath." This architecture explains why spinning up 100 containers consumes almost no extra disk space for the base OS.

2.5 Container Networking Model (CNM)

By default, Docker creates a virtual bridge network named `docker0`.

- **VETH Pairs:** Docker creates a pair of virtual interfaces. One end sits inside the container (seen as `eth0`), and the other attaches to the `docker0` bridge on the host.
- **NAT (Masquerading):** To allow containers to access the internet, Docker utilizes IP Tables to perform Network Address Translation (NAT), masking the container's private IP behind the host's IP.

3 Practical Implementation: From Setup to Code

Before diving into the application development, we must establish the runtime environment.

3.1 Environment Setup & Tooling

For this demonstration, we utilize a modern cloud-native stack.

- **Java 21 (LTS):** The application is built on the latest Long-Term Support version of Java.

  > *Future Outlook:* Java 21 introduces **Virtual Threads (Project Loom)**, which revolutionize concurrency in high-throughput distributed systems. A detailed presentation on how Virtual Threads replace reactive programming models will be delivered in a future session.

- **SDKMAN!:** A CLI tool used to manage parallel versions of multiple Software Development Kits. We use it to switch between JDK versions effortlessly (e.g., `sdk install java 21.0.2-tem`).
- **IntelliJ IDEA:** The Integrated Development Environment (IDE) of choice, utilizing its built-in "Spring Boot Dashboard" and Docker integration for seamless debugging.
- **Docker Engine/Desktop:** As the container runtime environment.

3.2 Essential Docker CLI Commands

Understanding the CLI is prerequisite for any automation. The most frequently used commands include:

Command	Description
`docker pull <image>`	Downloads an image from a registry (e.g., Docker Hub).
`docker build -t <tag> .`	Builds an image from a Dockerfile in the current directory.
`docker run -d -p 80:80 <tag>`	Runs a container in detached mode, mapping host port 80 to container port 80.
`docker ps`	Lists running containers (process status).
`docker exec -it <id> sh`	Opens an interactive shell session inside a running container.
`docker logs <id>`	Fetches the STDOUT/STDERR streams (crucial for debugging).

Table 1: Core Docker Commands Reference

3.3 Developing the Spring Boot Microservice

For the practical demonstration, we develop a cloud-native CRUD application named **Nebula Core**. The application follows a clean layered architecture to ensure separation of concerns, adhering to a standard enterprise package structure.

3.3.1 Project Structure

The application is organized into distinct layers under the `com.nebula.core` namespace, ensuring modularity for future microservices expansion:

```
src/main/java/com/nebula/core
|-- config                // Configuration (Security, Swagger)
|-- util                  // Helper utilities
`-- ws                    // Web Service Module
  |-- controller          // REST Endpoints
  `-- model               // Domain Models
      |-- dto             // Data Transfer Objects
      `-- entity          // Database Entities
  |-- repository          // Data Access Layer (DAO)
  `-- service             // Business Logic
    `-- impl              // Logic Implementation
```

Package Hierarchy

3.3.2 Implementation Logic: Mission Control

We implement a `MissionService` to manage interplanetary missions. This scenario provides a perfect context for future scaling challenges in Kubernetes (e.g., scaling missions across different nodes). The architecture flows as follows:

1. **Entity Layer:** The `Mission` entity represents a record in the PostgreSQL database, containing fields like `missionName`, `status`, and `launchDate`.
2. **Repository Layer:** `MissionRepository` extends 'JpaRepository' to provide seamless data access.
3. **Service Layer:** The `MissionServiceImpl` handles business rules (e.g., ensuring a mission cannot be launched twice).
4. **Controller Layer:** The `MissionController` exposes REST endpoints (GET /missions, POST /missions) to the external world.

Note: During the live session, we will execute the full build process, demonstrating how this structure is packaged into a single executable JAR file inside the container.

Source Code
The complete implementation is available at: `https://github.com/j-imsa/Nebula`

3.3.3 Data Modeling: The Mission Entity

The core domain object is the `Mission` entity. It is designed with distributed systems best practices in mind, such as using UUIDs for primary keys to avoid collision in sharded database environments.

```java
@Entity
@Table(name = "missions")
public class Mission {

  @Id
  @GeneratedValue(strategy = GenerationType.UUID)
  private UUID id;  // Distributed-safe ID

  @Column(nullable = false, unique = true)
  private String missionName; // e.g., "Nebula One"

  @Enumerated(EnumType.STRING)
  private MissionStatus status; // PLANNED, ORBITING...

  private String commander;   // e.g., "Cpt. Salehi"
  private String destination; // e.g., "Kepler-186f"
  private LocalDateTime launchDate;
  private Double fuelPercentage;

  // Getters and Setters omitted for brevity
}
```

Mission.java

To manage the lifecycle of a mission, we define a strict state machine using a Java Enum:

```java
public enum MissionStatus {
  PLANNED,      // Initial state
  LAUNCHING,    // Container starting up
  ORBITING,     // Stable running state
  COMPLETED,    // Task finished
  ABORTED       // System failure / Crash
}
```

MissionStatus.java

3.4 Engineering Best Practice: Multi-Stage Builds

A common problem in containerization is large image sizes. A standard Java image containing Maven and the source code can exceed 800MB. To solve this, we use **Multi-Stage Builds**.

This technique divides the `Dockerfile` into two distinct phases:

1. **The Build Stage:** We use a "heavy" image (containing Maven and JDK) to compile the source code and generate the executable `.jar` file.
2. **The Runtime Stage:** We start fresh with a "lightweight" image (containing only the JRE). We then **COPY** only the final `.jar` file from the first stage.

The Result: All the heavy build tools (Maven, local repo) are discarded, reducing the final image size from ≈800MB to ≈150MB.

```
# --- Stage 1: Build ---
FROM maven:3.9-eclipse-temurin-21 AS builder
WORKDIR /app
COPY pom.xml .
COPY src ./src
# Compile the code and package it into a JAR
RUN mvn clean package -DskipTests

# --- Stage 2: Runtime ---
# Start fresh with a tiny Linux image (Alpine)
FROM eclipse-temurin:21-jre-alpine
WORKDIR /app

# COPY ONLY the JAR file from the 'builder' stage
COPY --from=builder /app/target/*.jar app.jar

EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Dockerfile (Multi-Stage)

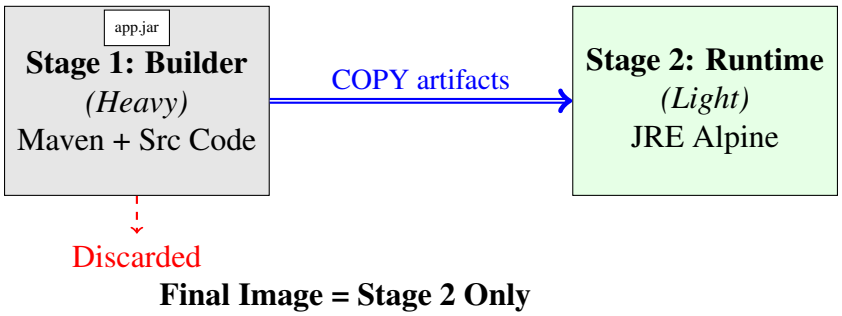

Figure 3: Visualizing the Multi-Stage Copy Process

4 Service Composition (Microservices)

Real-world microservices are rarely isolated. Our **Nebula Core** application requires a relational database (PostgreSQL) to persist Mission data. Managing these dependencies manually via CLI is error-prone.

4.1 Infrastructure as Code: Docker Compose

Docker Compose allows us to define the entire application stack (Services, Networks, Volumes) in a single declarative YAML file. This follows the "Infrastructure as Code" (IaC) paradigm.

Below is the configuration to launch the Nebula Core system:

```yaml
services:
  # 1. The Spring Boot Application
  nebula-app:
    build: .                    # Build from current Dockerfile
    ports:
      - "8080:8080"             # Map Host:8080 -> Container:8080
    environment:
      - SPRING_DATASOURCE_URL=jdbc:postgresql://nebula-db:5432/
        nebuladb
      - SPRING_DATASOURCE_USERNAME=postgres
      - SPRING_DATASOURCE_PASSWORD=secret
    depends_on:
      nebula-db:
        condition: service_healthy # Wait for DB to be ready!

  # 2. The Database Service
  nebula-db:
    image: postgres:15-alpine
    environment:
      - POSTGRES_DB=nebuladb
      - POSTGRES_PASSWORD=secret
    volumes:
      - pg_data:/var/lib/postgresql/data # Persist data on host
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 5s
      timeout: 5s
      retries: 5

volumes:
  pg_data: # Named volume for persistence
```

docker-compose.yml

4.2 Container Networking & Service Discovery

One of the most powerful features of Docker Compose is automatic service discovery.

1. **Bridge Network:** Docker creates a private internal network for these services.

2. **DNS Resolution:** Services can reach each other using their names as hostnames.

For example, the Spring Boot app connects to the database using the URL:

`jdbc:postgresql://nebula-db:5432/nebuladb`

Docker internally resolves `nebula-db` to the dynamic IP address of the database container.

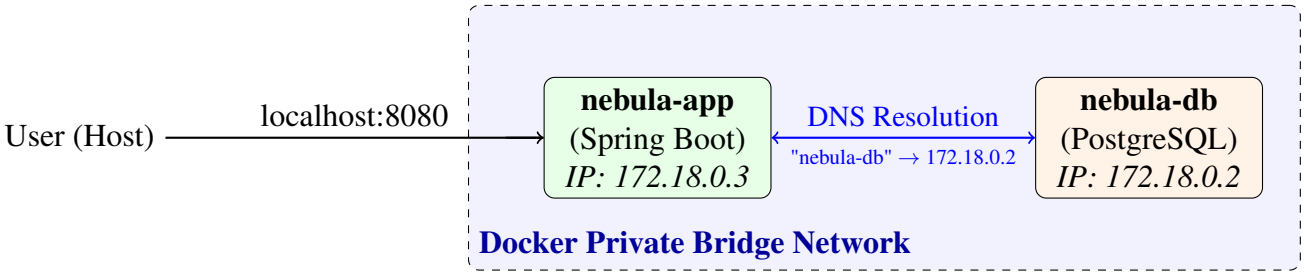

Figure 4: Service Discovery and Port Mapping in Compose

5 Challenges & Introduction to Orchestration

While Docker Compose is excellent for development, it runs on a single host. This creates a Single Point of Failure (SPOF).

5.1 The Need for Kubernetes (K8s)

To achieve High Availability (HA) and horizontal scaling, we must transition to an orchestrator like Kubernetes. K8s abstracts the hardware into a **Cluster**.

- **Control Plane:** The brain of the cluster (scheduling, API server).
- **Worker Nodes:** The machines where applications (Pods) actually run [3].

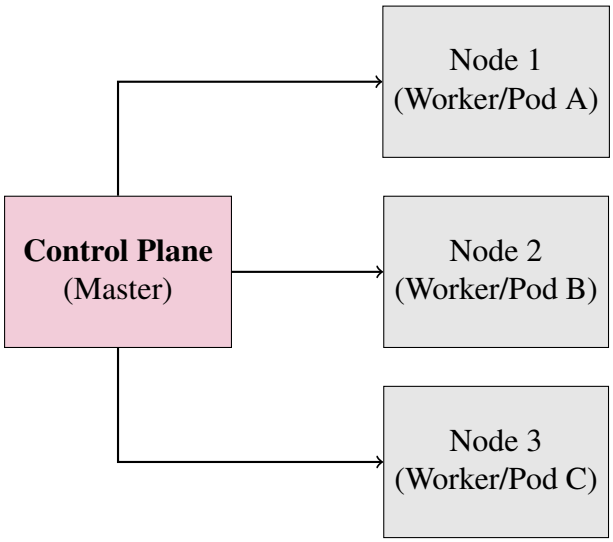

Figure 5: Kubernetes: Decoupling Workload from Hardware

5.2 Future Roadmap: Solving Nebula's Limitations

Currently, our Nebula Core deployment relies on a single Docker Compose instance. While functional, it faces critical distributed systems challenges that we aim to address in the next phase of this project using Kubernetes.

Current Challenge (Docker Compose)	**The Kubernetes Solution**
Single Point of Failure (SPOF) If the VM hosting the Docker Engine crashes, the entire Mission Control system goes offline.	**Self-Healing & ReplicaSets** K8s automatically detects node failures and reschedules the "Mission Pods" onto healthy worker nodes to maintain the desired state.
Manual Scaling If mission traffic spikes, we must manually SSH into the server and run `docker-compose up -scale nebula-app=5`.	**Horizontal Pod Autoscaler (HPA)** K8s monitors CPU/Memory metrics and automatically adds more Pods during high load, scaling down when traffic subsides.
Downtime Updates Updating the application version (‘v1‘ $\rightarrow$ ‘v2‘) requires stopping and restarting containers, causing service interruption.	**Rolling Updates** K8s incrementally updates Pods one by one with zero downtime, ensuring Mission Control is always accessible.

Table 2: Roadmap: Transitioning from Monolithic Hosting to Orchestration

Conclusion: The transition to Kubernetes is not merely an operational change but an architectural necessity to ensure the resilience and scalability required for a mission-critical system like Nebula Core.

References

[1] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," *IBM Research*, vol. 172, 2015, The foundational paper comparing VM and Container overhead.

[2] S. Sultan, I. Ahmad, and T. Dimitriou, "Containerization technologies: Taxonomies, applications, and challenges," *IEEE Access*, vol. 7, pp. 57 753–57 781, 2019, A comprehensive survey on modern container technologies.

[3] E. Casalicchio, "The state-of-the-art in container technologies: Application orchestration and elastic autoscaling," in *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, IEEE, 2020, pp. 366–370.