

# Big Data Analytics

## Lecture 4: Detailed Notes

Based on Slides by Dr. Tariq Mahmood, IBA

Fall 2025

## 1 The History and Evolution of Linux Containers

This lecture provides a comprehensive overview of the history of Linux containers, tracing the technological advancements that led to modern tools like Docker and Kubernetes. We will explore the foundational concepts, key technologies, and the evolution of container orchestration.

### 1.1 Early Isolation: chroot (1979)

The concept of process isolation in UNIX-like systems began with the `chroot` system call.

- **Functionality:** `chroot` changes the root directory (/) for the current process and its children. To the process, a specified subdirectory (e.g., `/mnt/test`) appears to be the entire filesystem. This is often called a "chroot jail."
- **Isolation Level:** It provides only filesystem isolation. Processes inside the jail cannot see or access files outside of their designated root directory.
- **Limitations:** `chroot` was a very simple form of isolation and did not offer strong security. It did not isolate other system resources like networking, process space (processes inside could still see and signal processes outside), or user IDs. A privileged process within the `chroot` jail could potentially break out.

#### 1.1.1 Advantages and Disadvantages of chroot

- **Advantages:** Simplicity, lightweight filesystem isolation, useful for creating simple, contained environments for testing or running specific services.
- **Disadvantages:** Weak security, only isolates the filesystem, does not manage resources, and can be escaped by a root process.

## 1.2 Enhanced Security: FreeBSD Jails (2000)

FreeBSD Jails were a significant step forward from `chroot`, creating a much more secure and comprehensive virtualization environment.

- **Enhancements over chroot:** Jails extended the concept of `chroot` by adding restrictions on processes, users, and networking.
- **Comprehensive Isolation:** A jailed process is restricted to a specific filesystem, a set of IP addresses, and a subset of the process space. The superuser (`root`) within a jail has limited privileges and cannot affect the host system or other jails.
- **Outcome:** This created a more secure "jail" that was much closer to the modern concept of a container, as it limited what processes could do outside their environment.

## 1.3 Enterprise Virtualization: Solaris Containers (2004)

Solaris Containers, also known as Solaris Zones, introduced OS-level virtualization to the Solaris operating system, focusing on resource management and strong isolation for enterprise applications.

- **Zones:** A Zone is a virtualized OS environment created within a single instance of the Solaris OS. The main OS instance is called the "global zone," which manages all hardware and can create and control "non-global zones."
- **Isolation and Resource Management:** Zones provide a separated environment with its own user list, process space, and virtual network interface. A key feature is the integration with Solaris Resource Management, which allows administrators to dedicate specific resources (like CPU shares) to each zone.
- **Abstract Layer:** This technology separates applications from the physical attributes of the machine, making it easier to manage and migrate applications.

# 2 Core Linux Kernel Technologies for Containers

Modern containerization on Linux is built upon two fundamental kernel features: Control Groups (`cgroups`) and Namespaces.

## 2.1 Resource Management: Control Groups (`cgroups`) (2007)

Introduced by Google engineers, `cgroups` are a kernel feature for limiting, accounting for, and isolating the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.

- **Purpose:** `Cgroups` allow partitioning of system resources and allocating them among user-defined groups of processes. This prevents any single group (or container) from consuming all available resources and impacting others.

- **Key Functions:**
  - **Resource Allocation & Limitation:** Set limits on memory, CPU, etc. For example, a cgroup can be limited to 500MB of RAM.
  - **Prioritization:** Give certain groups higher priority for CPU time or disk I/O.
  - **Accounting:** Monitor and report on the resource usage of a group.
  - **Control:** Freeze (suspend) and resume processes within a group.
- **Role in Containers:** Cgroups are the building blocks that container runtimes like Docker and Kubernetes use to enforce resource limits on containers. They answer the question: **What resources can a process use?**

## 2.2 Process Isolation: Linux Namespaces (2002)

Namespaces are the kernel feature that provides isolation. They wrap a set of system resources and present them to a process to make it look like it has its own isolated instance of that resource. They answer the question: **What can a process see?**

- **Concept:** A namespace acts like a "box" for global system resources. Adding or removing a resource from one box has no effect on other boxes. It's a view of the system, not the resource itself.
- **Types of Namespaces:**
  - **PID (Process ID):** Isolates the process tree. A process inside a PID namespace can have PID 1, and it cannot see processes outside its namespace.
  - **NET (Network):** Isolates network devices, IP addresses, routing tables, etc. Each container gets its own virtual network stack.
  - **MNT (Mount):** Isolates filesystem mount points. Each container has its own root filesystem.
  - **UTS (UNIX Time-sharing System):** Isolates the hostname and domain name.
  - **IPC (Inter-Process Communication):** Isolates access to IPC resources like semaphores and shared memory.
  - **USER:** Isolates user and group IDs. A user can have root privileges (UID 0) inside a container but be a regular, unprivileged user on the host.
  - **CGROUP:** Isolates the cgroup view, preventing a process from seeing the cgroup hierarchy of the host.

## 3 The Rise of Modern Container Runtimes

### 3.1 LXC - Linux Containers (2008)

LXC was the first complete and user-friendly implementation of containerization on Linux, bringing cgroups and namespaces together.

- **Nature:** LXC provides OS-level virtualization, creating containers that feel like lightweight virtual machines with their own full Linux environment.
- **Architecture:** It is a user-space interface for the kernel's containment features. It provides tools, templates, and libraries to make creating and managing containers easier.
- **Virtualization:** Lightweight, allowing multiple isolated containers to run on a single host.

### 3.1.1 LXD: The Container Hypervisor

LXD (pronounced "lex-dee") is a management layer built on top of LXC. It acts as a container "hypervisor," providing a REST API to manage LXC containers and also virtual machines. It aims to provide a better, more cloud-like user experience for managing system containers.

## 3.2 Docker (2013)

Docker revolutionized containerization by making it incredibly accessible, portable, and developer-friendly. While built on the same kernel technologies as LXC, its focus shifted from system containers to application containers.

- **Key Innovations:**
  - **Accessibility:** Made containers easy to use with a simple command-line interface (CLI).
  - **libcontainer:** Initially used LXC, but later developed its own library, 'libcontainer' (now part of 'runc'), to interact directly with cgroups and namespaces.
  - **Container Images:** Introduced the concept of portable, layered images. A **Dockerfile** defines the steps to assemble an image, which includes an application and all its dependencies. These images can be stored in a registry (like Docker Hub) and shared easily. This solved the "it works on my machine" problem.
- **Architecture Shift:** Docker focuses on packaging and running a single application or process per container, which is ideal for microservices architectures.

### 3.2.1 Comparison: LXC vs. Docker

Aspect	LXC	Docker
<b>Focus</b>	System Containers (lightweight VMs).	Application Containers (single app).
<b>Usability</b>	More complex configuration, similar to a VM.	User-friendly interface, simple commands.
<b>Virtualization</b>	OS-Level Virtualization.	Application-Level Virtualization.
<b>Ecosystem</b>	Smaller community, fewer tools.	Extensive ecosystem with Docker Hub, vast tooling.
<b>Performance</b>	Commendable speed, but can be heavier.	Fast deployment due to lightweight nature.
<b>Security</b>	Relies on standard kernel features.	Adds layers like Seccomp and user namespaces for more reliable isolation.

## 3.3 rkt (2014)

rkt (pronounced "rocket") was developed as an alternative to Docker with a stronger focus on security, simplicity, and adherence to open standards.

- **Architecture:** Designed without a central daemon, which was seen as a security risk and a single point of failure in early Docker versions.
- **Core Unit: Pods:** Its core execution unit is the "pod," a collection of one or more applications running in a shared context, a concept that Kubernetes heavily utilizes.
- **Standards Compliance:** Implemented the App Container (appc) specification and could also run Docker images.

*Note: The rkt project is no longer actively developed, but its ideas heavily influenced the container ecosystem, especially around standards and the pod model.*

## 3.4 The Open Container Initiative (OCI) and Core Runtimes

To standardize container formats and runtimes, the Open Container Initiative (OCI) was formed. This led to the creation of modular, standard components that now form the backbone of Docker and other container platforms.

- **runc:** A low-level, CLI tool for spawning and running containers according to the OCI specification. It is the reference implementation of the OCI runtime. 'runc' handles the direct interaction with namespaces and cgroups to create the container.
- **containerd:** A higher-level container runtime that manages the entire container life-cycle. It sits above 'runc' and handles tasks like image transfer and storage, container execution and supervision, and network management. Docker donated 'containerd' to the Cloud Native Computing Foundation (CNCF).

### 3.4.1 Modern Docker Architecture Flow

When a user runs `docker run nginx`:

1. The **Docker CLI** sends a request to the **Docker Engine (dockerd)**.
2. **dockerd** asks **containerd** to create a container.
3. **containerd** pulls the image if it's not available locally.
4. **containerd** calls **runc** to create and run the container process.
5. **runc** interacts with the Linux kernel to set up namespaces and cgroups, mount the image layers, and start the container process.

## 4 Container Orchestration

As the use of containers grew, managing hundreds or thousands of them across multiple machines became a major challenge. This led to the development of container orchestration platforms.

- **Definition:** Orchestration is the automated configuration, coordination, and management of complex systems and services. For containers, it automates their entire lifecycle, including deployment, management, scaling, and networking.
- **Orchestration vs. Automation:**
  - **Automation** is about setting up a single task to run on its own (e.g., a script to deploy one container).
  - **Orchestration** is about automating many tasks as part of a larger workflow or process (e.g., deploying an application with 10 containers, ensuring they can talk to each other, scaling them based on traffic, and replacing them if they fail).

### 4.1 Key Functions of Orchestration

Orchestration platforms solve critical operational problems:

- **Provisioning and Deployment:** Deciding which container should run on which host ("scheduling").
- **High Availability:** Keeping applications running even if a container or an entire host crashes by automatically restarting or rescheduling containers.
- **Scaling:** Automatically adding or removing container replicas based on demand (load).
- **Networking & Service Discovery:** Connecting containers across different hosts and allowing them to find and communicate with each other.
- **Load Balancing:** Distributing network traffic across multiple container replicas.

- **Updates:** Managing rolling updates and rollbacks with zero downtime.
- **Resource Management:** Ensuring fair allocation of resources across containers.

## 4.2 Popular Orchestration Tools

- **Docker Swarm:** Docker's native orchestration tool, known for its simplicity and tight integration with the Docker ecosystem.
- **Apache Mesos:** A general-purpose cluster manager that can run containerized and non-containerized workloads. It's known for its scalability.
- **Nomad (HashiCorp):** A simple and flexible workload orchestrator that can manage containers and non-containerized applications.
- **Kubernetes (K8s):** The market leader and de facto standard for container orchestration.

## 4.3 Deep Dive: Kubernetes (K8s)

Originally developed at Google and now managed by the CNCF, Kubernetes is an open-source system for automating the deployment, scaling, and management of containerized applications.

### 4.3.1 Kubernetes Architecture

A Kubernetes cluster has a client-server architecture consisting of a Control Plane and multiple Worker Nodes.

- **Control Plane (The Brain):** Makes global decisions about the cluster (e.g., scheduling) and detects and responds to cluster events.
  - `kube-apiserver`: The frontend of the control plane; exposes the Kubernetes API. All components interact through it.
  - `etcd`: A consistent and highly-available key-value store used as the backing store for all cluster data (the "single source of truth").
  - `kube-scheduler`: Watches for newly created Pods with no assigned node and selects a node for them to run on.
  - `kube-controller-manager`: Runs controller processes that regulate the state of the cluster (e.g., node controller, replication controller).
- **Worker Nodes (The Muscle):** Machines (VMs or physical) where the actual application workloads (containers) run.
  - `kubelet`: An agent that runs on each node. It ensures that containers described in PodSpecs are running and healthy.

- **kube-proxy**: A network proxy that runs on each node, maintaining network rules to allow communication to your Pods from inside or outside the cluster.
- **Container Runtime**: The software responsible for running containers (e.g., containerd, CRI-O).

#### 4.3.2 Example: Kubernetes Deployment

The following YAML defines a Kubernetes Deployment object to run 3 replicas of an NGINX web server.

```

1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: webapp
5 spec:
6   replicas: 3
7   selector:
8     matchLabels:
9       app: webapp
10  template:
11    metadata:
12      labels:
13        app: webapp
14  spec:
15    containers:
16      - name: webapp
17        image: nginx
18        ports:
19          - containerPort: 80

```

## 5 Mastering Docker: Images and Storage

### 5.1 Docker Architecture Revisited

- **Client-Server Model**: The **Docker Client** (CLI) talks to the **Docker Daemon** ('dockerd') via a REST API. The daemon handles building, running, and managing containers.
- **Docker Host**: The machine where the Docker daemon and containers run.
- **Docker Registry**: A storage system for Docker images. **Docker Hub** is the default public registry.

### 5.2 Docker Images and Layers

A Docker Image is a read-only template used to create a container.

- **Layers**: Images are built up from a series of layers. Each instruction in a Dockerfile creates a new layer. Each layer is just a set of filesystem changes from the layer before it.

- **Base Image:** The starting layer of an image (e.g., ‘ubuntu:22.04’).
- **Efficiency:** The layered filesystem (using a union file system) is highly efficient. When you pull an image, you only download layers you don’t already have. When you run a container, the read-only image layers are shared, and a thin, writable layer is added on top for that specific container. This saves disk space and speeds up builds and deployments.

## 5.3 The Dockerfile

A Dockerfile is a text document that contains all the commands, in order, needed to build a given image.

### 5.3.1 Common Dockerfile Commands

Command	Purpose
FROM	Specifies the base image to build upon.
RUN	Executes a command in a new layer to install software, etc.
COPY	Copies files or directories from the local machine into the image.
ADD	Similar to COPY, but can also handle URLs and extract archives.
CMD	Provides the default command to run when the container starts. Can be overridden.
ENTRYPOINT	Configures the container to run as an executable. Arguments to ‘docker run’ are passed to it.
WORKDIR	Sets the working directory for subsequent commands.
ENV	Sets environment variables inside the container.
EXPOSE	Documents a port the application listens on (for networking purposes).
VOLUME	Creates a mount point for persistent data.
USER	Sets the user ID to run the container as.
LABEL	Adds metadata to an image (e.g., author, version).
ARG	Defines a variable that can be passed at build-time.

## 5.4 Managing Data in Docker: Storage

Data inside a container’s writable layer is ephemeral—it is lost when the container is removed. To persist data, Docker provides two main options: volumes and bind mounts.

### 5.4.1 Volumes

Volumes are the preferred mechanism for persisting data generated by and used by Docker containers.

- **How they work:** A volume is a directory on the host machine that is managed entirely by Docker. Docker creates and manages it in a specific area (e.g., `/var/lib/docker/volumes/`).
- **Advantages:**
  - Easier to back up or migrate than bind mounts.
  - Can be managed via the Docker CLI and API.
  - Can be safely shared among multiple containers.
  - Work on both Linux and Windows containers.
  - Performance is often better than bind mounts on macOS and Windows.

#### 5.4.2 Bind Mounts

A bind mount maps a file or directory from the host machine directly into a container. The path on the host is arbitrary and controlled by the user.

- **How they work:** You provide a host path and a container path, and Docker mounts the host path into the container.
- **Use Cases:** Ideal for development when you want to mount your source code directly into a container and see changes immediately. Also useful when you need to share configuration files from the host to a container.
- **Disadvantages:**
  - Tightly coupled to the host's directory structure, making it less portable.
  - Can pose security risks, as a container could modify files on the host filesystem, including system files if mounted improperly.

#### 5.4.3 Storage Plugins

For more advanced use cases, Docker's storage plugin system allows volumes to be integrated with external storage systems like cloud providers (e.g., Amazon EBS) or network file systems, enabling enterprise-grade persistence and management.