CONTAINERS

The Building Blocks of Cloud-Native Applications

About Me

- Senior Staff Software Engineer @ AMD AI Group
- Master's in Computer Science
  NYU Tandon School of Engineering (2023)
- Teaching Assistant
  CS:9223 Cloud Computing & Big Data, NYU Tandon (4 semesters)
- **LinkedIn**: linkedin.com/in/prateek1709

The Problem: "It Works on My Machine" 🤔

The Challenge:

Developer: "The app works perfectly on my laptop!"
Operations: "It crashes in production... again."

Common Issues:

- Dependency Conflicts
- Environment Inconsistencies: Different OS versions (Windows dev, Linux production), Missing system libraries, Different configuration files, Network settings vary etc.
- The Traditional Solution: Documentation

Analogy: Imagine a chef creating a recipe at home, but the restaurant kitchen has different ovens, different ingredients, and different tools. The dish never tastes the same!

The Solution? → Containers! Package everything together.

What are Containers?

Containers are ***lightweight, standalone***, and ***executable*** software packages that encapsulate an application and all its dependencies.

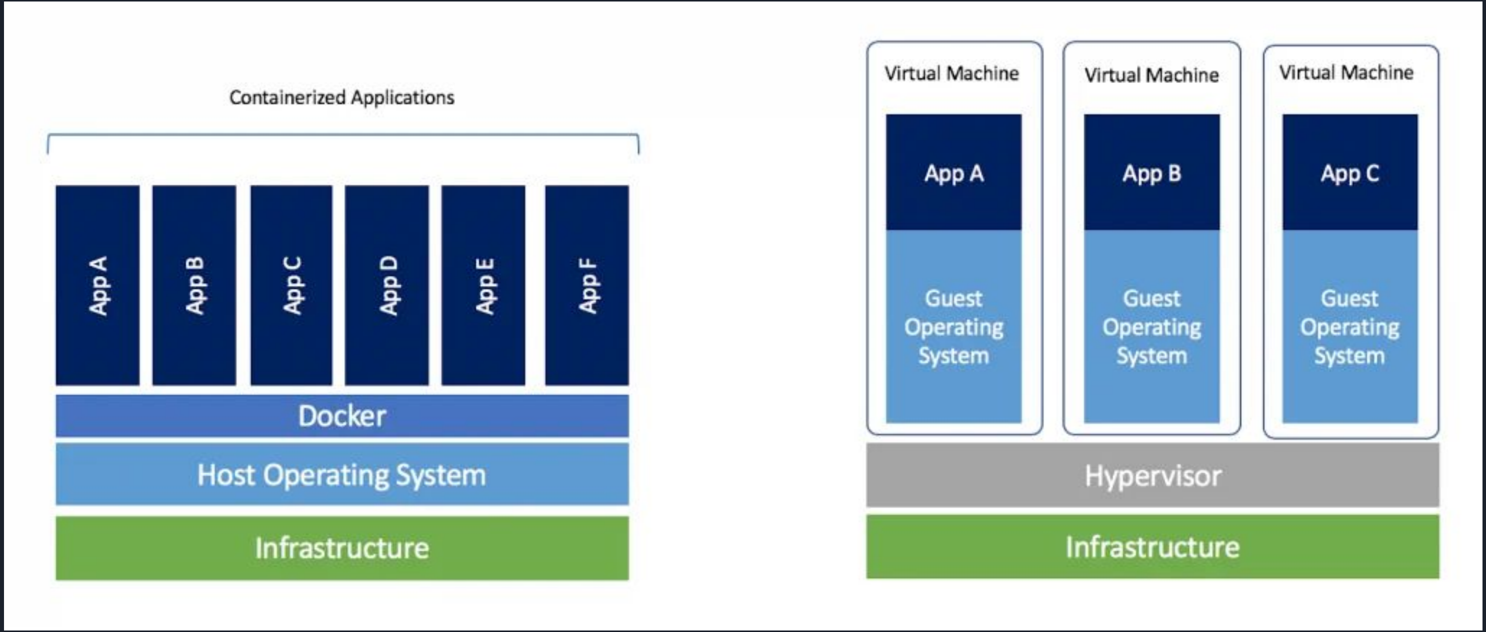
They provide a consistent environment for applications to run, regardless of the host system.

Real-life Analogy: Food Trucks

Imagine software containers as food trucks in a bustling city.

Just as food trucks package everything needed to prepare and serve specific cuisines in a mobile unit, software containers package applications and their dependencies for easy deployment and execution.

Containers vs Virtual Machines

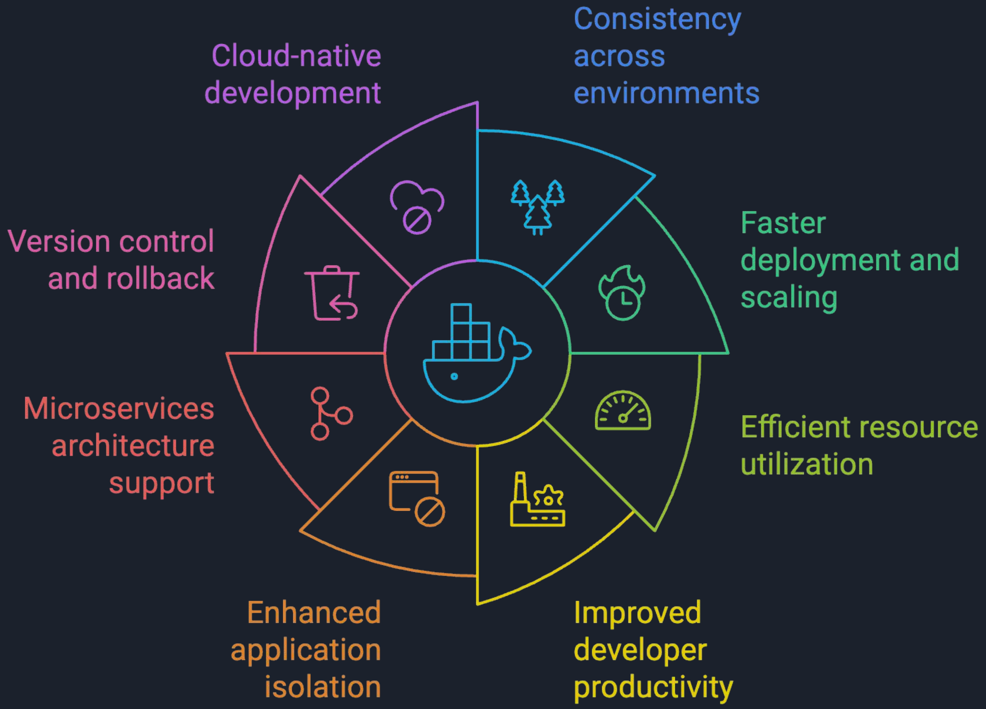


Containers vs. Virtual Machines

Think of **containers** as food trucks and **virtual machines** like traditional brick-and-mortar restaurants.

Feature	**Containers (Food Trucks)**	**Virtual Machines (Restaurants)**
Infrastructure	Share the street (host OS)	Own building and utilities
Resource Usage	Efficient use of limited space	More spacious, potentially underutilize
Startup Time	Quick setup and ready to serve (seconds)	Takes time to open and prepare (minutes)
Isolation	Separate trucks, shared environment	Complete isolation from other establishments
Customization	Limited to internal equipment	Can modify entire structure
Management	Focuses on food prep and service	Responsible for entire property management

Benefits of Using Containers


Cloud-native development
Consistency across environments
Faster deployment and scaling
Efficient resource utilization
Improved developer productivity
Enhanced application isolation
Microservices architecture support
Version control and rollback

Linux Kernel Features Enabling Containerization

Namespaces

Namespaces provide isolation for system resources, making processes believe they have their own isolated instance of the resource.

Food Truck Analogy: Namespaces are like giving each food truck its own unique street address and phone number, even though they're all in the same festival.

Control Groups (cgroups)

Cgroups limit and allocate resources to collection of processes.

Food Truck Analogy: Cgroups are like allocating specific amounts of electricity, water, and gas to each food truck, ensuring fair resource distribution.

```
# On host machine

$ ps

PID   COMMAND

1     systemd

42    chrome

137   spotify

5234  nginx   ← Your web server

5235  python
```

Every process sees ALL other processes. Your nginx knows it's process #5234.

```
# Inside Container A

$ ps

PID   COMMAND

1     nginx   ← Same nginx, thinks it's #1!


# Inside Container B

$ ps

PID   COMMAND

1     postgres ← Also thinks it's #1!


# On host machine

$ ps

PID   COMMAND

5234  nginx   ← Container A's nginx

5678  postgres ← Container B's postgres
```

Docker: Containerization Made Simple

Docker is an open-source platform for developing, shipping, and running applications in containers.

Key components of a Docker container:

Docker Component	**Food Truck Analogy**
Docker Engine	Food Truck Kitchen
Docker Images	Recipe book
Docker Containers	Operational truck
Dockerfile	Recipe instructions
Docker Hub	Recipe library

Docker Engine

Docker Engine is the core technology that runs and manages containers. It's a client-server application consisting of:

A server (daemon process)

REST API for interacting with the daemon

Command-line interface (CLI) client

The Docker Engine creates and manages Docker objects such as images, containers, networks, and volumes. It runs as a background service on the host operating system, handling container lifecycle operations.

Dockerfile

A Dockerfile is a text document containing a series of instructions for building a Docker image. It specifies:

The base image to use

Commands to update the base OS and install additional software

Source code to add to the image

The command to run when launching containers from the image

Dockerfiles automate the image creation process, ensuring consistency and reproducibility.

```dockerfile
# Start with a base image (the food truck chassis)
FROM python:3.9-slim

# Set working directory (set
 up the kitchen layout)
WORKDIR /app
# Copy dependency list (ingredients list)
COPY requirements.txt .

# Install dependencies (stock the kitchen)
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code (add your recipes)
COPY . .

# Expose the port (open the service window)
EXPOSE 5000

# Define the startup command (start cooking!)
CMD ["python", "app.py"]
```

Docker Images

Docker images are read-only templates used to create containers. They are:

Composed of layered filesystems

Built using instructions from a Dockerfile

Immutable once created

Shareable across different environments

Each image consists of a series of layers representing filesystem changes. Images leverage a Union File System to combine these layers into a single coherent filesystem.

Docker Containers

Containers are runnable instances of Docker images. They are:

Isolated environments with their own processes, networks, and mounts

Created from images and can be started, stopped, moved, and deleted

Defined by the image used to create them and any configuration options provided

Containers share the host system's OS kernel but are isolated from each other and the host using namespaces and cgroups.

Docker Hub

Docker Hub is a cloud-based registry service for:

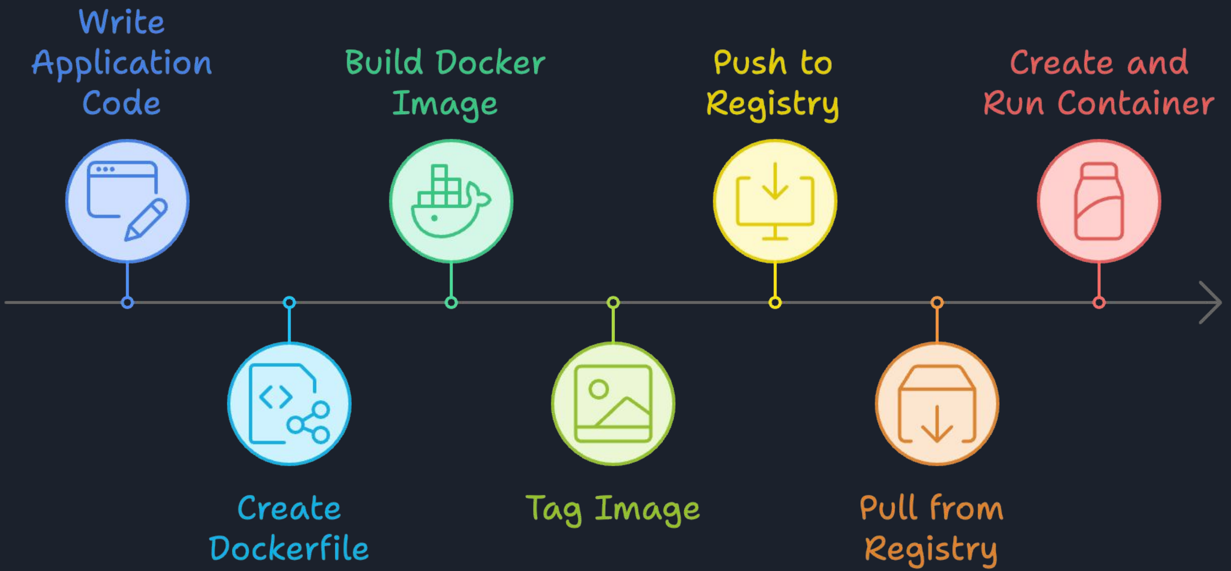
Storing and distributing Docker images

Integrating with CI/CD pipelines

Automating builds from GitHub and Bitbucket

It offers both public repositories for open-source projects and private repositories for proprietary code. Docker Hub serves as the default public registry for Docker, though private registries can also be used.

Docker Container Lifecycle



Setup Docker Desktop

Docker Desktop is a one-click-install application for your Mac, Linux, or Windows environment that lets you build, share, and run containerized applications and microservices.

It provides a straightforward GUI (Graphical User Interface) that lets you manage your containers, applications, and images directly from your machine.

Download: *https://docs.docker.com/desktop/*

Basic Docker Commands

- Pull an image: *docker pull hello-world*
- List images: *docker images*
- Run a container: *docker run hello-world*
- List running containers: *docker ps*
- List all containers (including stopped ones): *docker ps -a*
- Stop a container: *docker stop <container_id>*
- Start a stopped container: *docker start <container_id>*
- Remove a container: *docker rm <container_id>*
- View container logs: *docker logs <container_id>*
- Inspect a container: *docker inspect container_id*
- Execute a command in a running container: *docker exec -it <container_id> /bin/bash*

Demo:

https://github.com/PrateekKumar1709/Docker-Demo

Docker Volumes

Docker volumes are a mechanism for persisting data generated and used by Docker containers.

- Directories managed by Docker
- Stored on the host filesystem, outside the container's writable layer
- Can be shared and reused among multiple containers
- Persist data independently of the container's lifecycle

Docker Compose

- Tool for defining and running multi-container Docker applications
- Uses YAML files to configure application services
- Simplifies the process of managing multiple containers as a single application

Basic Commands

- docker-compose up: Start services
- docker-compose down: Stop and remove containers
- docker-compose ps: List running services
- docker-compose logs: View output from containers

Thank You!