TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA
Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

OS-Level and Lightweight Virtualization

Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2025/26

Valeria Cardellini

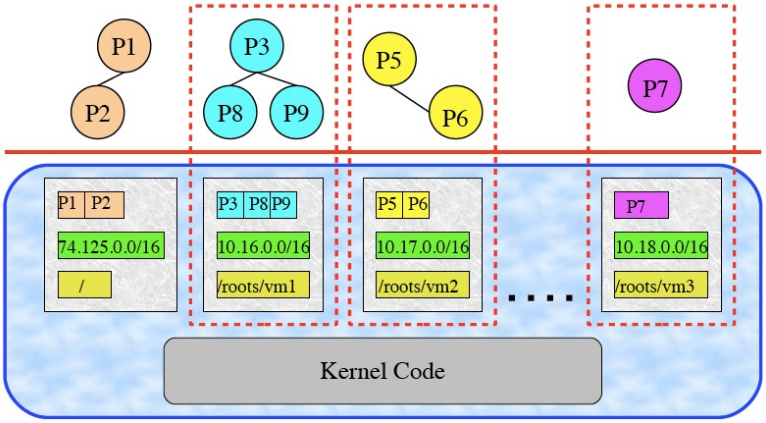
Laurea Magistrale in Ingegneria Informatica

OS-level virtualization

- Let's consider **operating system (OS) level virtualization** (or ***container-based virtualization***)
- It allows running multiple isolated (*sandboxed*) user-space instances on top of a single OS
  - Such instances are called:
    - **containers**
    - jails
    - zones

OS-level virtualization

- OS kernel allows the existence of multiple isolated user-space instances, called containers
- Each container has:
  - Its own set of processes, file systems, users, network interfaces with IP addresses, routing tables, firewall rules, …
- Containers share the same OS kernel (e.g., Linux)

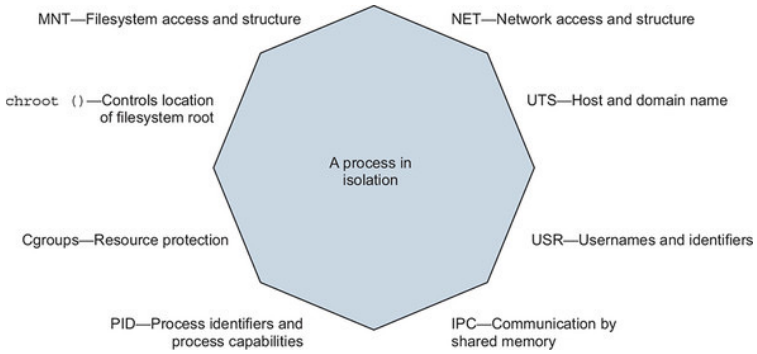


OS-level virtualization: mechanisms

- Which OS kernel mechanisms are used to manage containers?
  - Containers need to isolate processes from each other in terms of sw and hw (CPU, memory, …) resources
- Main mechanisms offered by Unix-like OS kernels
  - chroot (change root directory)
    - Allows changing the apparent root folder for the current running process and its children
  - cgroups (Linux-specific)
    - Manage resources for groups of processes, such as CPU and memory allocation
  - namespaces (Linux-specific)
    - Per-process resource isolation, ensuring that each container has its own isolated environment

Mechanisms: namespaces

- Feature of Linux kernel that allows to isolate what a set of processes can see in the operating environment
  - Includes resources such as processes, ports, files, …
- Kernel partitions resources so that one set of processes can see a specific set of resources, while another set of processes sees a different set of resources
- 6 different types of namespaces



Mechanisms: namespaces

- mnt: isolates mount points seen by a container
  - Virtually partitions the file system, so processes running in separate mount namespaces cannot access files outside of their mount point
- pid: isolates PID space, ensuring that each process only sees itself and its children (PID 1, 2, 3, …)
- network: allows each container to have its dedicated network stack
  - Its own private routing table, set of IP addresses, socket listings, firewall rules, and other network-related resources
- user: isolates user and group IDs
  - E.g., allows a non-root user on host to be mapped to root user within the container, without granting actual root access to host

Mechanisms: namespaces

- uts (Unix timesharing): provides dedicated host and domain names
  - Allows processes to think they are running on servers with different names, even though they share the same host
- ipc: provides dedicated shared memory for IPC
  - E.g., separate Posix message queues for different containers

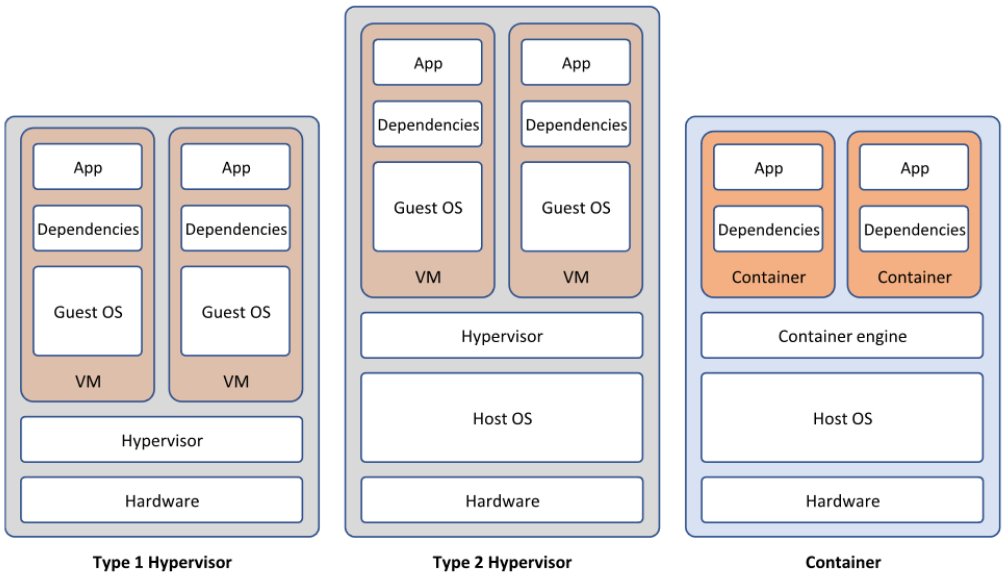
Mechanisms: cgroups

- cgroups = control groups
- Limit, measure and isolate the use of hw resources (CPU, memory, I/O, network) for a group of processes
- Exposed via a filesystem interface (similar to `sysfs` and `procfs`)
  - Default mount point: `/sys/fs/cgroup/`

- In a nutshell:
  - namespaces implement *information isolation*: what a container can see
  - cgroups implement *resource isolation*: how many resources a container can use

OS-level virtualization: pros

- VMM-based vs container-based virtualization



In a nutshell: lightweight vs. heavyweight

OS-level virtualization: pros

vs. VMM-based virtualization (type-1)

✓ Near-native performance
  – No VMM indirection for system calls

✓ Fast startup and shutdown
  – Seconds (even msec) per container vs. minutes per VM

✓ High density
  – Hundreds of containers per physical machine (PM)

✓ Small footprint
  – Container images are smaller since they exclude the OS kernel

✓ Memory efficiency
  – Containers can share memory pages on the same PM

✓ Portability and interoperability
  – Apps run across environments

OS-level virtualization: cons

vs. VMM-based virtualization (type-1)

✗ Less flexible
- Only supports native apps for the OS kernel (e.g., no Windows container on Linux host)
- Cannot run different OS kernels on same PM; however, can run multiple Linux distributions (e.g., Ubuntu, CentOS)

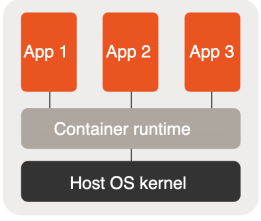
✗ Weaker isolation
- Process-level isolation leads to higher performance interference on shared resources

✗ Higher security risks
- A kernel vulnerability affects the entire system
- Since containers share the kernel, one compromised container can impact other containers and the host

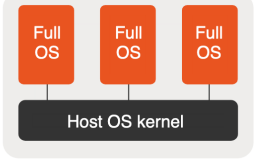
✗ Reduced hardware/device isolation
- Device passthrough is more difficult and less secure

OS-level virtualization: products

- Docker
  - Most popular container engine
  - Provides *application containers*
    - Package and run *a single application* with its dependencies
  - Supports Open Container Initiative (OCI) standards https://opencontainers.org
- LXC (LinuX Containers) https://linuxcontainers.org/lxc/
  - Supported by mainline Linux kernel
  - Provides *system containers*
    - Run a *whole OS user space with multiple processes*, but share the host kernel



Application containers



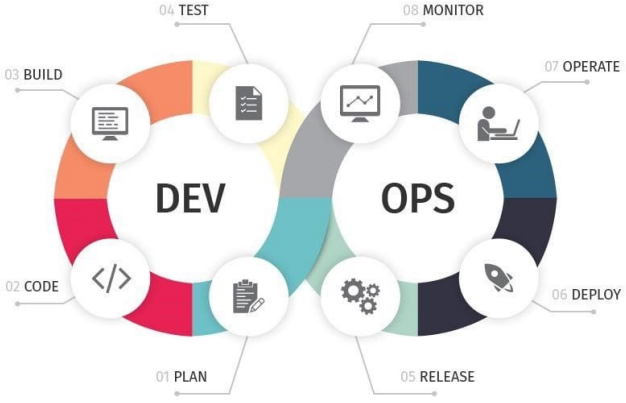
System containers

OS-level virtualization: products

- Podman https://podman.io
  - Supports OCI; Docker compatible CLI
  - Daemonless, rootless operation for improved security
- FreeBSD Jail
  - Strong process and filesystem isolation
- OpenVZ / Virtuozzo https://openvz.org
  - For Linux; primarily for system containers
- Non-Linux platforms
  - Windows and macOS support containers (e.g., Docker Desktop)
- Alternative approach
  - Install a Linux VM as a guest OS
  - Run container engines (e.g., Docker) inside the VM
  - Performance Impact: nested virtualization leads to reduced performance

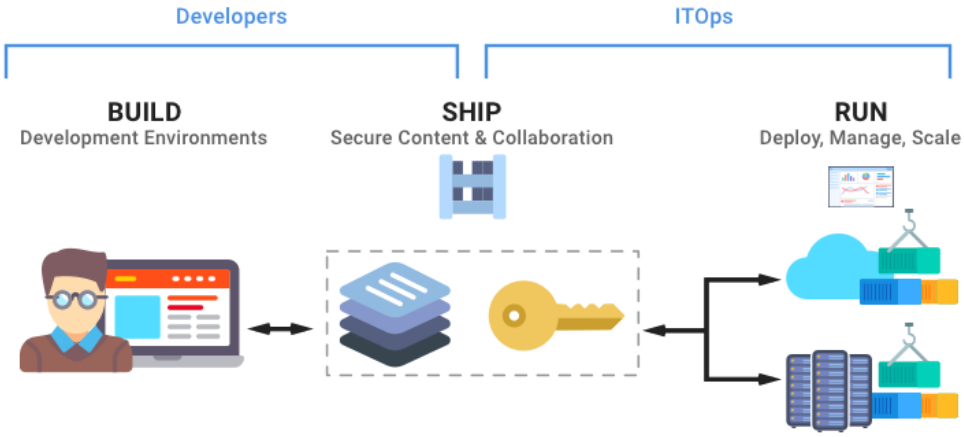
Containers, DevOps and CI/CD

- DevOps: development methodology that bridges the gap between Development and Operations, focusing on collaboration, continuous integration, and automated delivery
- CI/CD:
  - Continuous Integration (CI): merges developers' work into a shared codebase
  - Continuous Delivery (CD): ensures frequent and reliable releases



Containers, DevOps and CI/CD

- Containers: simplify building, packaging, sharing, and deploying apps with all dependencies
- Enable collaboration by sharing images, and streamline deployment across environments without extra configuration

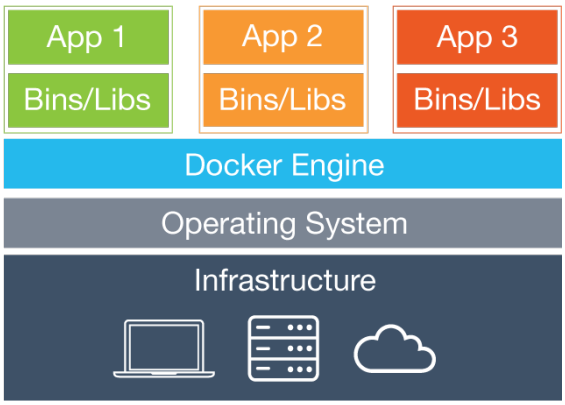


Containers, microservices, and serverless

- Using containers
  - Package apps and all dependencies into a single unit that runs almost anywhere
  - Use fewer resources than traditional VMs
- Containers enable
  - Microservices: break down apps into small, independent services
  - Serverless: package functions into containers for efficient execution

Case study: Docker

- Lightweight and secure container-based virtualization
  - Application containers: contain the application and its dependencies, but share OS kernel with other containers
  - Isolation: containers run as isolated processes in user space on host OS
  - Portability: containers are infrastructure-agnostic and can run anywhere



Docker: features

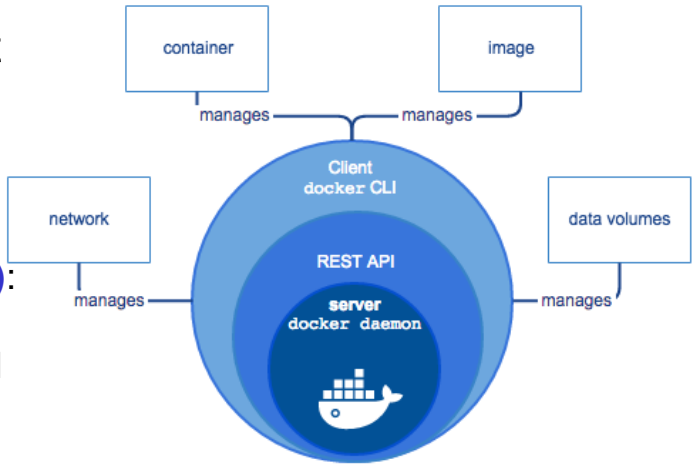
- Portable deployment
  - Easily deploy across different machines and environments
- Versioning
  - Git-like version control for container images
- Component reuse
  - Reuse components (e.g., libraries, services) via Docker images
- Shared libraries
  - Access to pre-built images on Docker Hub https://hub.docker.com
- OCI support
- Scalability
  - Works seamlessly with Kubernetes for scaling applications

Docker: internals

- Written in Go
- Exploits Linux kernel mechanisms for resource management and isolation (cgroups and namespaces)
  - Early versions were based on Linux Containers
  - Then transitioned to libcontainer, a Go-based container runtime: provides tools for managing containers with namespaces, cgroups, capabilities, and filesystem access controls
    https://pkg.go.dev/github.com/opencontainers/runc/libcontainer
  - runc: libcontainer is now part of runc, the CLI tool for spawning and running containers based on OCI specification
    https://github.com/opencontainers/runc

Docker Engine: architecture

- Docker Engine: core component of Docker that enables containerization
- Client-server application composed by
  - **Docker daemon (dockerd)**: server component that listens for API requests and manages Docker objects like images, containers, networks, and volumes

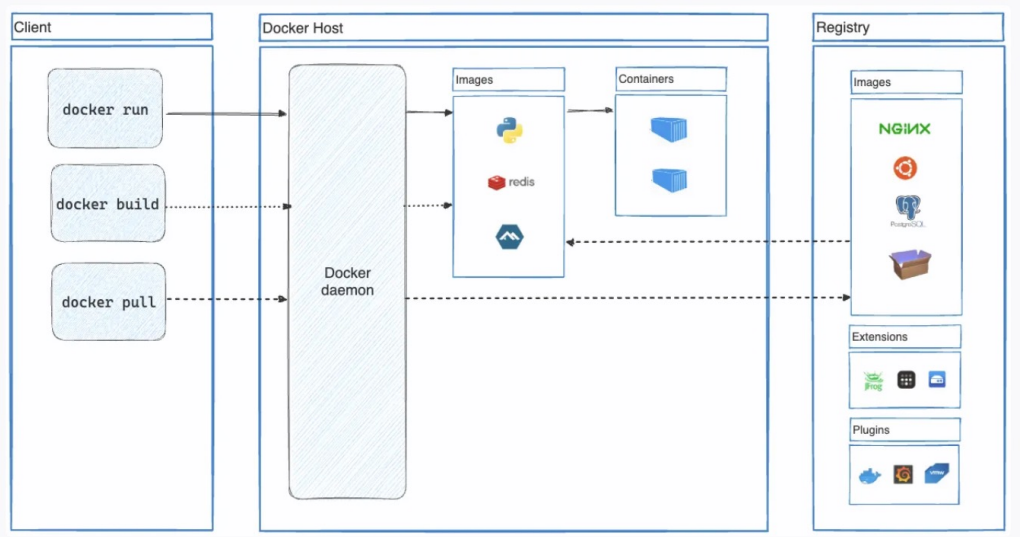


  - **REST API**: specifies the interfaces that programs can use to interact with Docker daemon for operations like creating, managing, and querying Docker objects
  - **CLI client**: allows users to send commands to Docker daemon via REST API

https://docs.docker.com/get-started/docker-overview/#docker-architecture

Docker: client-server architecture

- Docker client
  - Interface through which users interact with Docker
  - Sends commands to Docker Daemon to build, run, and distribute Docker containers
  - Client and daemon communicate through sockets or the REST API



Docker: images

- Read-only template
  - Used to create Docker containers, containing everything needed to run app (code, dependencies, configurations)
- Build component
  - Docker images enable distribution of apps with their runtime environments, removing the need to manually install packages
  - Target machine must be Docker-enabled
- **Dockerfile**
  - Text file with simple instructions that Docker uses to build images automatically
- **Image registry**
  - Images can be pulled and pushed to/from public or private registries
- Image naming
  - Format: `[registry/][user/]name[:tag]`
  - Default tag is latest

Docker image: Dockerfile

- Docker images are created from a Dockerfile and a context
  - Dockerfile: text file containing instructions to assemble the image
  - Context: set of files (e.g., app code, libraries) used during the image build process
  - Images often build on parent images (e.g., Alpine, Ubuntu)
- Dockerfile syntax

  `# Comment` (for comments)

  `INSTRUCTION arguments` (commands like RUN, COPY, etc.)
- Instructions in the Dockerfile run sequentially

Docker image: Dockerfile

- Common Dockerfile instructions
  - **`FROM <image>`**: specifies the parent image (mandatory unless you want to start from **`scratch`**)
  - **`WORKDIR <path>`**: sets the working directory inside the container
  - **`COPY <host-path> <image-path>`**: copies files from host to container image
  - **`RUN <command>`**: executes the specified command during image build
  - **`ENV <name> <value>`**: sets an environment variable
  - **`EXPOSE <port>`**: exposes a network port for the container
  - **`CMD ["<command>", "<arg1>"]`**: defines the default command to run when the container starts

https://docs.docker.com/get-started/docker-concepts/building-images/writing-a-dockerfile/

Docker image: Dockerfile

- Example: Dockerfile to build the image of a container that will run as application a simple todo list manager written in Node.js

```
# syntax=docker/dockerfile:1

FROM node:lts-alpine
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
EXPOSE 3000
```

Directory with app code

```
├── getting-started-app/
│   ├── package.json
│   ├── README.md
│   ├── spec/
│   ├── src/
│   └── yarn.lock
```

https://docs.docker.com/get-started/workshop/02_our_app/

Docker image: build

- Building Docker image from Dockerfile and context
  - Context: set of files located in the specified PATH or URL
- Build command

```
$ docker build [OPTIONS] PATH | URL | -
```

- Example: to build an image for Node.js app (slide 24)

```
$ docker build -t getting-started .
```

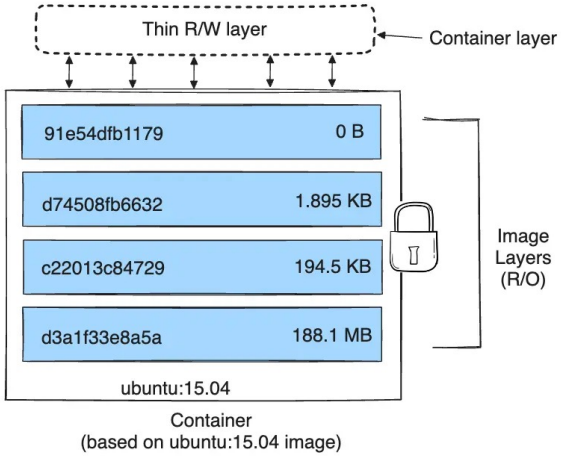
  - The `-t` flag is used to tag the image with a name (and optionally a version)
  - If Dockerfile is named something different from Dockerfile, use the `-f` flag:

```
$ docker build -t getting-started -f myDockerfile .
```

https://docs.docker.com/reference/cli/docker/build-legacy/

Docker image: layers

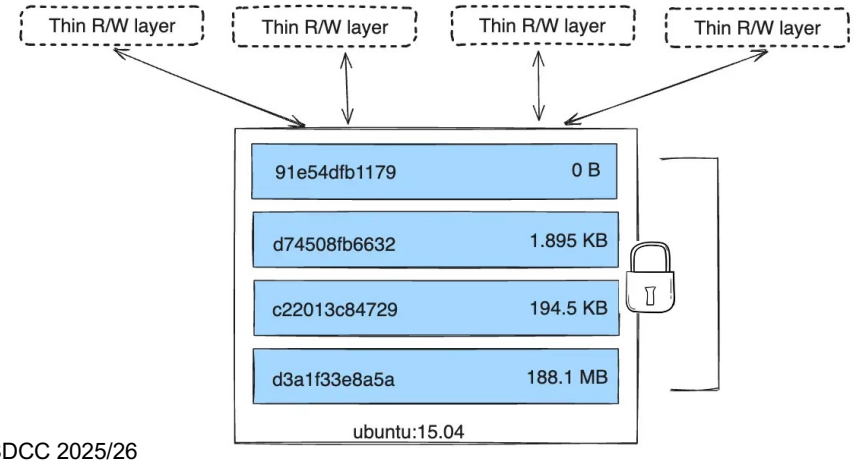
- Each Docker image consists of a series of layers
- Docker uses *union file systems* to combine these layers into a single unified view
  - These layers are stacked to form a base the base of a container's root file system
  - Docker leverages copy-on-write (CoW) strategy to optimize performance



Docker image: layers

- Layering pros
  - ✓ Efficient layer sharing and reuse: common layers are installed only once, saving bandwidth and storage space
  - ✓ Separation of concerns: allows better management of dependencies
  - ✓ Facilitates software specialization

https://docs.docker.com/storage/storagedriver



Docker image: layers

- Dockerfile instructions that modify the filesystem create new layers
  - Examples: `FROM`, `RUN`, `COPY`, `ADD`
- Instructions that only modify the image's metadata do not create new layers
  - Examples: `CMD`, `LABEL`
- All image layers (except the top one) are read-only
  - To enable efficient layer sharing across images
- When a container is started, a writable layer (aka *container layer*) is added on top
  - Changes made by a running container (e.g., creating or modifying a file) are written to the writable layer
  - Unique to each container
  - Ephemeral: deleted when the container is removed
  - Use it only for temporary/runtime data, not persistent data

Docker container storage

- Containers are usually stateless
- Why? Easier to:
  - Scale: start quickly new replicas
  - Restart: from failure
  - Migrate: move between hosts
- Very little data is written to container's writable layer
- Data is typically stored in **Docker volumes**
- However, some workloads require writing data to container's writable layer

Docker: storage backends

- How Docker daemon stores image layers and container writable layers on disk
  - Storage drivers
  - containerd image store
- Storage drivers (legacy): common options
  - Overlay2: file-level, preferred for all Linux distros
  - Btrfs: supports snapshotting
  - Zfs: block-level
- Storage driver considerations
  - Driver choice affects container performance
  - Drivers are optimized for space efficiency
  - Write performance speeds may be lower than native file system performance due to CoW

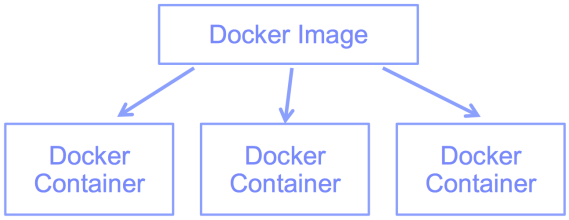
https://docs.docker.com/storage/storagedriver/select-storage-driver

Docker: storage backends

- New default backend: containerd image store (Docker Engine 29.0+)
- Uses content-addressable blob store
- Pros and cons:
  - ✓ Cleaner architecture
  - ✓ Unifies Docker and containerd ecosystem
  - ✗ Uses more disk space than storage drivers (no deduplication at driver level)

Docker: containers and registry

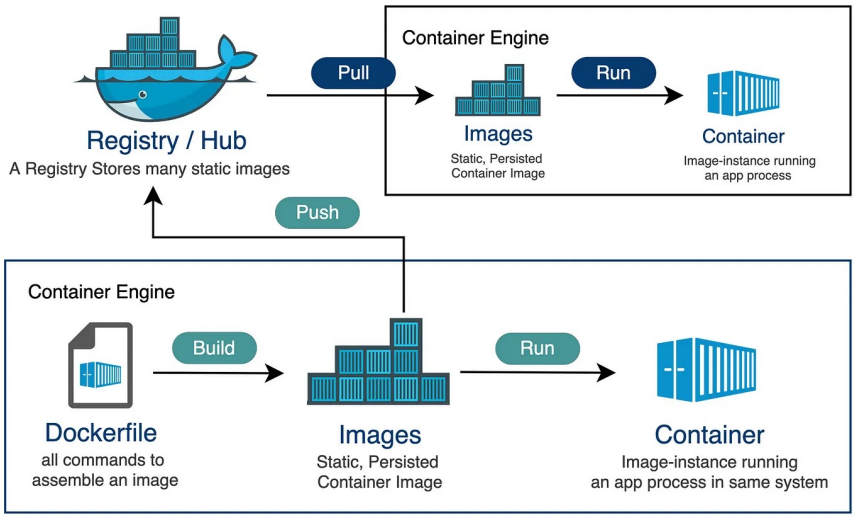
- Docker **container**: runnable instance of a Docker image
  - Containers are the ***run*** component of Docker
  - Run, start, stop, move, or delete a container using Docker API or CLI commands
  - Stateless nature of containers: when a container is deleted, any data written outside of data volumes is lost



- Docker **registry**: stateless server-side application that stores and distributes Docker images
  - Registry is the ***distribute*** component of Docker
  - Public and private registries
    - Docker-hosted registries: Docker Hub (official one), Docker Store (for open-source and enterprise-verified images)
  - Open library of images

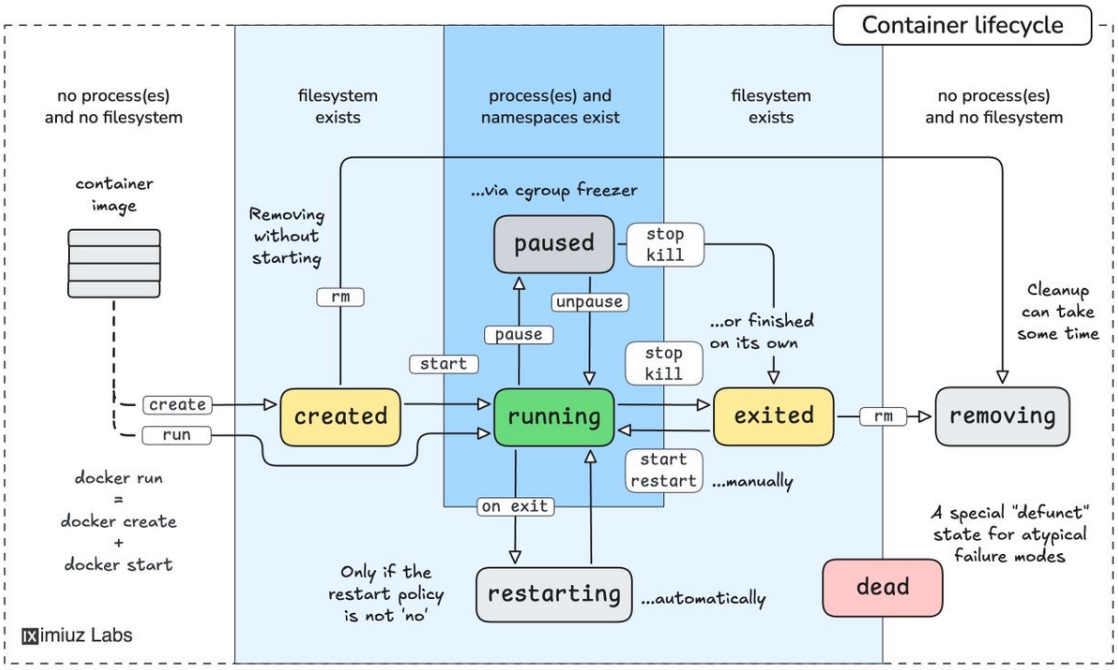
Docker: run container

- When you run a container from an image that is not yet installed locally but is available on Docker Hub, Docker will pull the image from the registry before starting the container



Docker container: states and transitions

- Different states a container can go through during its lifecycle and actions that trigger transitions



Docker commands: info

- Get system-wide info on Docker installation

  ```
  $ docker info
  ```

  including:
  - Number of images and containers, along with their status
  - Storage driver
  - Operating system, architecture, total memory
  - Docker registry information

Docker commands: image handling

- List all images on host

  `$ docker images` or `$ docker image ls`

- List all images, including intermediate image layers

  `$ docker images -a` or `$ docker image ls -a`

- You can filter and format the output to list images by name, tag, image digests (sha256) or image that meet specific conditions
  - E.g., list unused images (<none>) that are no longer associated with any tagged images but consume disk space

  `$ docker images --filter "dangling=true"`

- Remove an image

  `$ docker rmi imageid`

  alternatively, `$ docker image rm imageid`

or use *imagename* instead of *imageid*

Docker commands: image handling

- Remove dangling images

  `$ docker image prune`

- Inspect an image, including layers and image metadata

  `$ docker inspect imageid`

Docker commands: run

```
$ docker run [OPTIONS] IMAGE [COMMAND] [ARGS]
```

- Common options
  - `--name` set container name
  - `-d` detached mode (background)
  - `-i` interactive (keeps STDIN open)
  - `-t` allocate a pseudo-tty, usually with `-i`
  - `--expose` declare port(s) inside container
  - `-p` or `–publish` map container port(s) to host
  - `-v` mount volume
  - `-e` set environment variables inside container
  - `--rm` automatically remove container when it exits
  - `--restart` set restart policy (e.g., `always`, `on-failure`)

https://docs.docker.com/reference/cli/docker/container/run/

Docker commands: containers management

- List containers
  - Only running containers: `$ docker ps`
    alternatively, `$ docker container ls`
  - All containers (including stopped or killed containers):
    `$ docker ps -a`
- Manage container lifecycle
  - **Stop** a running container
    `$ docker stop containerid`
  - **Start** a stopped container
    `$ docker start containerid`
  - **Kill** a running container
    `$ docker kill containerid`
  - **Remove** a container (after stopping it)
    `$ docker rm containerid`

- Stop and remove a container

```
$ docker ps
$ docker stop containerid
$ docker ps -a
$ docker rm containerid
```

or use *containername* instead of *containerid*

Docker commands: containers management

- Stop all containers

```
$ for i in $(docker ps -q); do docker stop $i; done
```

- Run a command inside a running container

```
$ docker exec [OPTIONS] CONTAINER [COMMAND] [ARGS]
```

Example: `$ docker exec -it mycontainer /bin/bash`

- Inspect a container
  - To get the most detailed view of a container environment

```
$ docker inspect containerid
```

- Copy files between host and container

```
$ docker cp containerid:path localpath
$ docker cp localpath containerid:path
```


Docker: networking

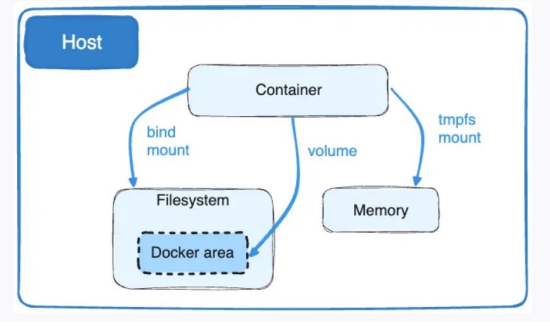
- Containers communicate with each other or with external systems
- Published ports
  - Use `-p hostPort:containerPort` (e.g., `-p 8080:80`) in `docker run` to expose a container port externally
  - Security issue: exposing container ports can be insecure
    - Restrict access to host only by binding to localhost (e.g., `-p 127.0.0.1:8080:80`)
  - Work only on host network: containers can communicate with each other via internal Docker network
- IP address and hostname
  - Containers get an IP address from Docker's dynamic subnet
  - Docker daemon manages subnetting and IP allocation
  - Default container hostname = container ID (can override with `--hostname`)

Docker: network drivers

- Docker networking is pluggable
  - Docker supports multiple network drivers that provide different networking behaviors, including:
- `bridge` (default)
  - Default network driver when no network is specified
  - Use when containers need to communicate on the same host
  - Implements a software bridge
    - Containers on the same bridge network can communicate with each other
    - Containers on different bridge networks are isolated
- `host`
  - Removes network isolation between container and host
  - Container uses host's network stack directly (no private IP assigned)

Docker: volumes

- Preferred mechanism for persisting data generated or used by containers
  - Volume content persists outside the container lifecycle: use volumes over container writable layer for persistent data
  - Does not increase container image size
- How volumes work
  - Docker creates a directory inside its storage directory to manage volume content
  - Default location on Linux: `/var/lib/docker/volumes/`
  - Volumes are created automatically if they do not already exist



Docker: volumes

- Mounting a volume: use `-v` or `--volume` in `docker run`

  `$ docker run -v source:destination:[options] imageid`
  - If the volume does not exist, Docker automatically creates it
  - `[options]`: optional flags, e.g., `ro` for read-only volume
- Managing volumes
  - Create a volume: `$ docker volume create volumename`
  - List all volumes: `$ docker volume ls`
  - Inspect a volume: `$ docker volume inspect volumename`
  - Remove a volume: `$ docker volume rm volumename`
- Can also be declared in a Dockerfile using

  `VOLUME ["/localpath"]`
- Working with volume data: pre-populate or load data using `docker cp`

  `docker cp /localpath containerid:/path`

 https://docs.docker.com/engine/reference/commandline/

Docker volume: pros

- ✓ Fully managed by Docker
- ✓ Easy to back up or migrate
- ✓ Accessible through Docker CLI or API
- ✓ Work on both Linux and Windows containers
- ✓ Shareable across multiple containers
- ✓ Can store encrypted content
- ✓ Suitable for pre-populated or write-heavy workloads (e.g., database, logging service)

Docker hands-on

- Download and install Docker
  - Available on multiple platforms

  https://docs.docker.com/get-started
- Test Docker installation

  ```
  $ docker --version
  ```
- Run the default hello-world container

  ```
  $ docker run hello-world
  ```
- Run a “Hello World” message using Alpine Linux

  ```
  $ docker run alpine /bin/echo 'Hello world'
  ```
  - alpine: lightweight Linux distro with very small image size
- Use commands to:
  - List containers and container images
  - Stop and remove containers, remove container images

Docker hands-on: networking

- Run nginx Web server inside a container
  - Bind container port 80 to host port 80

  ```
  $ docker run -dp 80:80 --name web nginx
  ```

  Flag `-p`: publish container port (80) to host port (80)

  Flag `-d`: run in detached mode

1. Send HTTP request through Web browser
   - First retrieve hostname of host machine (e.g., localhost)
2. Send HTTP requests to nginx from another interactive container using a custom bridge network

```
$ docker network create -d bridge my_net
$ docker run -dp 80:80 --name web --network=my_net nginx
$ docker run -it --network=my_net --name web_test busybox
/ # wget -O - http://web:80/
/ # exit
```

Docker hands-on: from Dockerfile

- Running Apache web server with minimal index page
  1. Define the container image using Dockerfile
     - Start from Ubuntu, install and configure Apache
     - Declare incoming port 80 using EXPOSE

```
FROM ubuntu:22.04

# Install dependencies
RUN apt-get update -y && \
    apt-get install -y apache2 && \
    apt-get clean

# Add simple web page
RUN echo "Hello World!" > /var/www/html/index.html

# Expose port and run Apache in the foreground
EXPOSE 80
CMD ["apache2ctl", "-D", "FOREGROUND"]
```


Docker hands-on: from Dockerfile

  2. Build the image

```
$ docker build -t hello-apache .
```

  3. Run the container and bind ports

```
$ docker run -dp 127.0.0.1:8080:80 hello-apache
```

  4. Execute an interactive shell in the running container

```
$ docker exec -it hello-apache /bin/bash
```

- To reduce container image size, avoid unnecessary layers
  - E.g., in Dockerfile combine `apt-get update` and package installation into a single RUN instruction (see slide 48)

Docker hands-on: volumes

- Run nginx container with a volume

```
$ docker volume create my-vol
$ docker volume ls
$ docker volume inspect my-vol
$ docker run -d \
    --name devtest \
    -v my-vol:/app \
    nginx:latest
```

  - `my-vol` is the source volume, `/app` is the target path inside container

```
$ docker inspect devtest
```

  - Check that Docker has created and mounted the volume correctly

Docker: optimize Docker images

- Fewer layers → smaller images → faster builds and deployments
- Why optimize Docker images?
  - Essential for DevOps engineers at every stage of CI/CD process
  - Reduces image size and disk usage
  - Speeds up image transfer, deployment, and startup times
  - Improves security by reducing the attack surface
  - Best practice used by Google and other major tech companies
  - Best practice employed by Google and other tech giants

https://devopscube.com/reduce-docker-image-size

Docker: optimize Docker images

- Techniques
  1. Use minimal base images (e.g., alpine, minideb) or distroless base images
     - Distroless images:
       - contain only application and its runtime dependencies
       - include package managers, shells, or other common utilities
       - More secure, smaller, harder to tamper with

       https://github.com/GoogleContainerTools/distroless
  2. Minimize the number of image layers
     - Combine related commands in a single RUN instruction
     - Avoid unnecessary COPY, ADD, or repeated RUN steps

Docker: optimize Docker images

- Techniques
  3. Multistage builds
     - Use intermediate images (build stages) to
       - compile code
       - install dependencies, and package files
     - Final image contains only the files and libraries needed to run app

```
# Build stage
FROM golang:1.21 AS builder
WORKDIR /app
COPY . .
RUN go mod init myapp
RUN go mod tidy
RUN go build -o myapp

# Final stage
FROM alpine:3.20
COPY --from=builder /app/myapp /usr/local/bin/myapp
CMD ["myapp"]
```

Docker: optimize Docker images

- Techniques
  4. Exploit layer caching
     - Place instructions that change infrequently (like installing dependencies) before COPY commands
     - Docker reuses cached layers for faster builds when source code changes.
  5. Use a `.dockerignore` file
     - Specify files and directories to exclude from the build context
     - Common exclusions: `node_modules`, `.git`, `*.log`, `__pycache__/`
  6. Keep application data in a volume
     - Avoid storing persistent data inside the container.
     - Use Docker volumes to store databases, logs, uploaded files

Docker: sizing containers

- By default, containers have no resource constraints
  - Can use as much CPU, memory, and I/O as the host's kernel scheduler allows
- Control resources by setting runtime configuration flags of `docker run`
  - Docker uses cgroups to manage resource limits

https://docs.docker.com/engine/containers/resource_constraints

Docker: sizing containers - memory

- Avoid running out of memory (OOM)
  - Containers may be killed
  - Docker daemon has lower OOM score, so less risk than containers
- Enforce hard or soft memory limits
  - Hard limit: container cannot use more than the specified limit; use `--memory` flag
  - Soft limit: container can use more memory if needed, unless certain conditions are met (e.g., kernel detects contention or low memory on host machine)
  - Example: hard limit (500 MB) and soft limit (300 MB)

```
$ docker run –it --memory-reservation="300m" \
  --memory="500m" ubuntu /bin/bash
```


Docker: sizing containers - CPU

- Options to limit CPU usage

  `--cpus=<value>`: limit container to a specific number of CPUs (hard limit)

  `--cpu-quota=<value>`: set Completely Fair Scheduler (CFS) CPU quota on container

  `--cpuset-cpus`: restrict container to specific CPUs/cores, example `--cpuset-cpus="0,1"` (use only CPU 0 and 1)

  `--cpu-shares`: set relative CPU weight (soft limit)

- Example: limit container to use at most 50% of CPU every second

```
$ docker run -it --cpus=".5" ubuntu /bin/bash
```

Alternatively,

```
$ docker run -it --cpu-period=100000 \
  --cpu-quota=50000 ubuntu /bin/bash
```

Resizing containers

- Containers can be resized and migrated, just like VMs
- Resizing allows changing CPU, memory, and I/O limits dynamically
  - Note: on Docker, dynamic resizing is not supported on Windows

```
$ docker update [OPTIONS] CONTAINER [CONTAINER...]
```

  - Examples

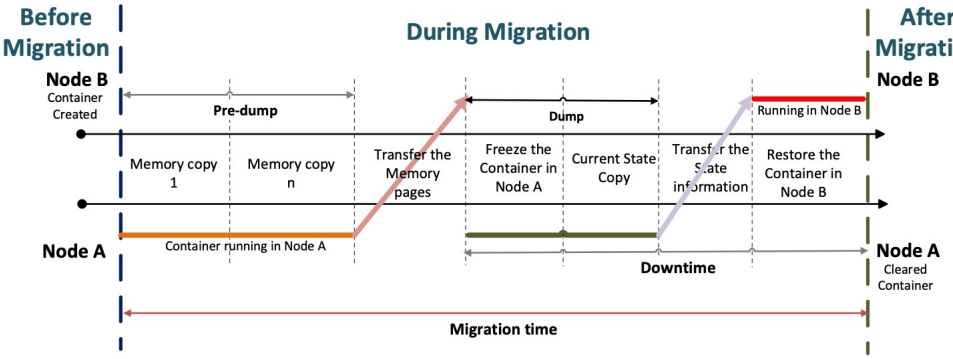
```
$ docker update --cpu-shares 512 containerID
$ docker update --cpu-shares 512 -m 300M containerID
```


Container live migration

- As for VM migration, we need to:
  - Save state
  - Transfer state
  - Restore state
- State saving, transferring and restoring happen with frozen app: migration downtime
  - Use memory pre-copy or memory post-copy
- No native support in container engines, additional tools required
- We also need to migrate container image, volumes, and network connections

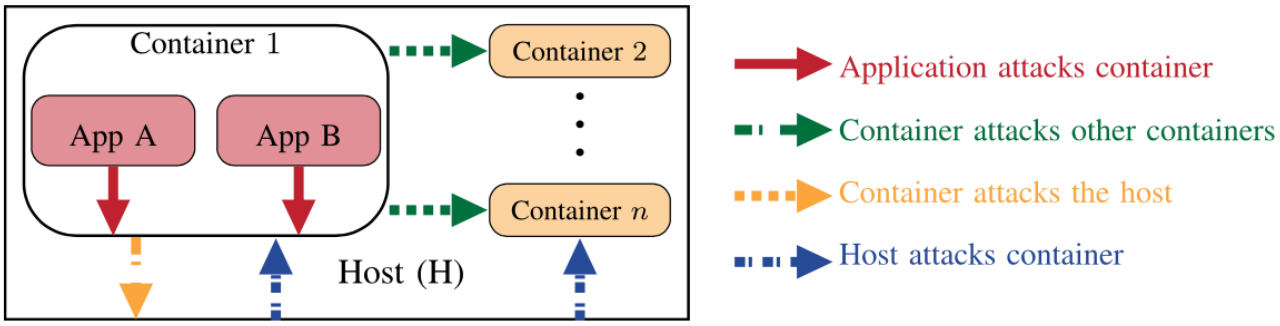
Container live migration

- CRIU (Checkpoint/Restore in Userspace) tool for live migration through checkpointing and restoration (Docker and other engines) https://criu.org
  - Checkpoint: freeze running container on source host and collects information about its CPU state, memory content, and process tree
  - Transfer and restore: transfer collected information to destination host, restore container's state and resume execution https://docs.docker.com/reference/cli/docker/checkpoint



Container security

- Where attacks come from in containerized environment?



- Attack origins
  - Vulnerabilities in containerized apps
  - Container runtime/kernel flaws

Container security

- Types of attacks
  - Container escape
    - Exploitation of container vulnerabilities to break isolation and access the host system
  - Privilege escalation
    - Once on the host, attackers can escalate privileges to:
      - Access other containers
      - Run malicious code on host system
- Consequences
  - Compromised host system
  - Cross-container attacks

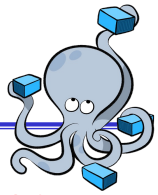
Docker: useful tools

- To manage images
  - Reduce image size: Slim https://slimtoolkit.org
  - Explore image layers: Dive https://github.com/wagoodman/dive
  - Automate image builds: Packer https://www.packer.io
- To monitor containers
  - cAdvisor https://github.com/google/cadvisor
- To check fo vulnerabilities
  - Docker Scout https://docs.docker.com/scout
  - Trivy https://trivy.dev
  - Static analysis: Clair https://github.com/quay/clair
- Many other tools: https://github.com/veggiemonk/awesome-docker

Container orchestration

- Sw platforms for managing multi-container apps
- Functionalities: configure, provision, deploy, monitor, and dynamically control containerized apps
  - Designed to integrate and manage containers in large-scale environments across multiple hosts
  - Can include autoscaling, load balancing, networking, monitoring and logging, fault tolerance and self-healing mechanisms
- Examples
  - Docker Swarm
  - **Kubernetes**
  - Nomad https://developer.hashicorp.com/nomad
- Also available as fully managed Cloud services
- For single-host deployment of small-scale apps
  - **Docker Compose**

Docker Compose



- Tool for defining and running multi-container Docker applications https://docs.docker.com/compose
  - Included with Docker Desktop https://docs.docker.com/compose/install
- How it works
  - Configuration-as-code
  - Define services: users specify the containers (services) to be instantiated, their configuration and relationships in a YAML file
- Single host, multiple containers
  - Orchestrates multiple containers on a single host (single Docker engine)
- Network setup
  - Compose automatically creates a network and attaches all containers to it, enabling easy communication between them

Docker Compose: how to use

- Define containers in a YAML file named `compose.yaml` (or `compose.yml`)
  - Specifies the containerized services
  - Defines how containers interact and their configurations
- Start Docker composition (background `-d`):

```
$ docker compose up -d
```

  - By default, Docker Compose looks for `compose.yaml` in working directory
    - Use `-f` flag to specify a different YAML file

```
$ docker compose -f composefile up -d
```

- Stop running containers:

```
$ docker compose stop
```

- Bring composition down, removing everything

```
$ docker compose down
```


Docker Compose: Compose file

- To configure Docker application's services, networks, volumes, and more
  - Different versions of Compose file format
  - Compose V2: implements format defined by Compose Specification https://compose-spec.io/ and includes support for legacy formats (2.x and 3.x)
- What is inside `compose.yaml`
  - `version`, `services`, `networks`, `volumes`, `configs`, `secrets`
  - Only `services` is required, others are optional

https://docs.docker.com/reference/compose-file
https://github.com/docker/awesome-compose

Docker Compose: Compose file

- Service: abstract representation of computing resources within app, that can be scaled, updated or replaced independently from other components
  - Defines a set of containers
  - Compose file must include the `services` top-level element
- Within each service
  - `build`: defines how to create service image (e.g., from Dockerfile)
  - `container_name`, startup and shutdown dependencies between services (`depends_on`), exposed containers `ports`, CPU and memory limits, `volumes` that are accessible to service containers
  - and other settings, see https://docs.docker.com/reference/compose-file/services/

Docker Compose: example

- Simple Python web app running on Docker Compose
  - 2 containers: Python web app and Redis
  - Use Flask framework and maintain a hit counter in Redis
  - Redis: in-memory, key-value data store

  See https://docs.docker.com/compose/gettingstarted

- Steps:
  1. Write Python app
  2. Define Python container

```
# syntax=docker/dockerfile:1
FROM python:3.10-alpine
WORKDIR /app
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
RUN apk add --no-cache gcc musl-dev linux-headers
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
EXPOSE 5000
COPY . .
CMD ["flask", "run", "--debug"]
```

Docker Compose: example

- Steps (cont'd):
  3. Define services in Compose file
     - 2 services: web (image built from Dockerfile) and redis (official image pulled from Docker Hub https://hub.docker.com/_/redis)

compose.yaml

```
services:
  web:
    build: .
    ports:
      - "8000:5000"
  redis:
    image: "redis:alpine"
```

  4. Build and run app with Compose

     ```
     $ docker compose up -d
     ```

  5. Send HTTP requests using curl or browser (counter is increased)
  6. Stop Compose and bring everything down

     ```
     $ docker compose down
     ```


Docker Compose: example

- Specify restart policy for containers
  - Options: `on-failure[:max-retries]`, `always`, `unless-stopped`
- Start multiple replicas of same service using `deploy` specification
  - Scale out or in manually the number of replicas

```
$ docker compose -f compose_v2.yaml up --scale web=4 -d
$ docker ps
$ docker compose -f compose_v2.yaml up --scale web=1 -d
$ docker ps
```

  ✗ Docker Compose only supports manual scaling
  - To experience autoscaling, learn Kubernetes

Docker Compose: full example

compose_v2.yaml

```
services:
  web:
    build: .
    ports:
      - "5000"
    restart: always
    deploy:
      replicas: 3
    environment:
      - FLASK_APP=app.py
    depends_on:
      - redis

redis:
    image: "redis:latest"
    container_name: redis
    restart: always
    volumes:
      - redis-data:/data

volumes:
  redis-data:
```


Docker Compose: example

- Drawback of v2:
  - The replicas of `web` service are visible: how can we add distribution transparency?
- Solution (see `compose_v3.yaml`):
  - Add a load balancer in front of the `web` replicas
  - Use Nginx a layer-7 proxy by adding a `nginx` service to the composition
  - In the `nginx` service, use a bind mount to mount the Nginx configuration file on the host inside the container
  - Use an internal network for all the containers

Example of Dockerized distributed system

- Kafka cluster using Docker Compose 

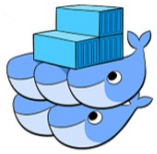
- Multiple pre-configured options are available, e.g.,
  - https://bitnami.com/stack/kafka/containers includes both single container and Docker Compose setup with Zookeeper or Kraft mode
  - https://medium.com/@darshak.kachchhi/setting-up-a-kafka-cluster-using-docker-compose-a-step-by-step-guide-a1ee5972b122 cluster of 3 brokers (Kraft mode) and UI, Docker network for inter-broker communication, and persistent volume storage
    - Added also a Kafka client

Docker Compose: pros and cons

✓ Simplified control
✓ Efficient collaboration
✓ Rapid application development
✓ Reproducible environments
✓ Portability across environments (development, staging, production)

✗ Single host limitation
✗ Lack of elasticity
✗ Not production-grade orchestration
✗ Only basic security features

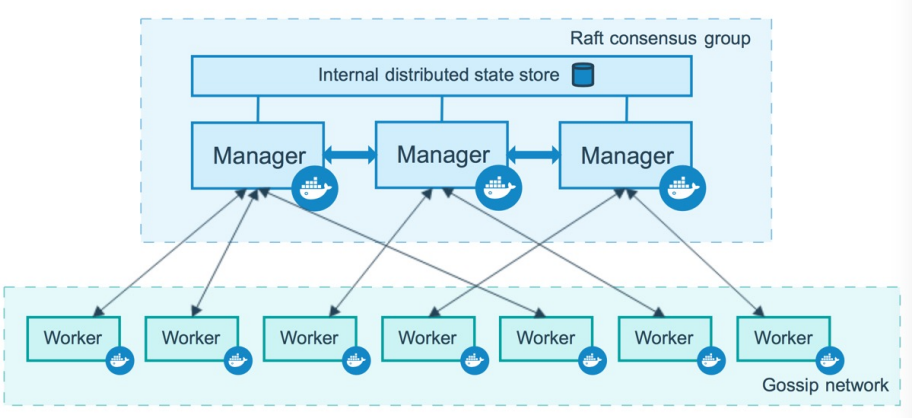
Docker Swarm



- Swarm mode: advanced feature of Docker to natively manage a cluster of Docker engines called a *swarm* https://docs.docker.com/engine/swarm/
- Swarm: multiple Docker engines running in swarm mode
  - Swarm mode helps you orchestrate containers across multiple machines
- Composed of Manager nodes and Worker nodes
  - Manager nodes: control the swarm and handle the orchestration of services
  - Worker nodes: run containers (tasks) as assigned by manager nodes
  - Tasks: containers running in a service
    - Task: smallest unit of work, typically one container
  - Services: how tasks (containers) should run on the swarm; provide an abstraction for deploying and managing tasks

Docker Swarm: architecture

- Node: single instance of Docker engine in a swarm
  - Manager nodes: handles cluster management, including task scheduling
    - Multiple managers to improve fault tolerance
    - Raft as consensus algorithm to ensure consistency
  - Worker nodes execute tasks
    - Workers use a gossip protocol to exchange information about their state



Docker Swarm: features

- Cluster management integrated with Docker
- Decentralized: distributed decision-making
- Declarative service model
- State reconciliation
  - Swarm monitors cluster state and reconciles any differences wrt desired state (e.g., a node crashes)
- Scaling
  - Easily scale services but lacks auto-scaling
- Multi-host networking
  - Use overlay networks to enable communication between services across nodes
- Load balancing
  - Can expose service ports to an external load balancer
- Secure: TLS authentication, encryption, role-based AC
- Easy to use and lightweight

Container-as-a-Service (CaaS)

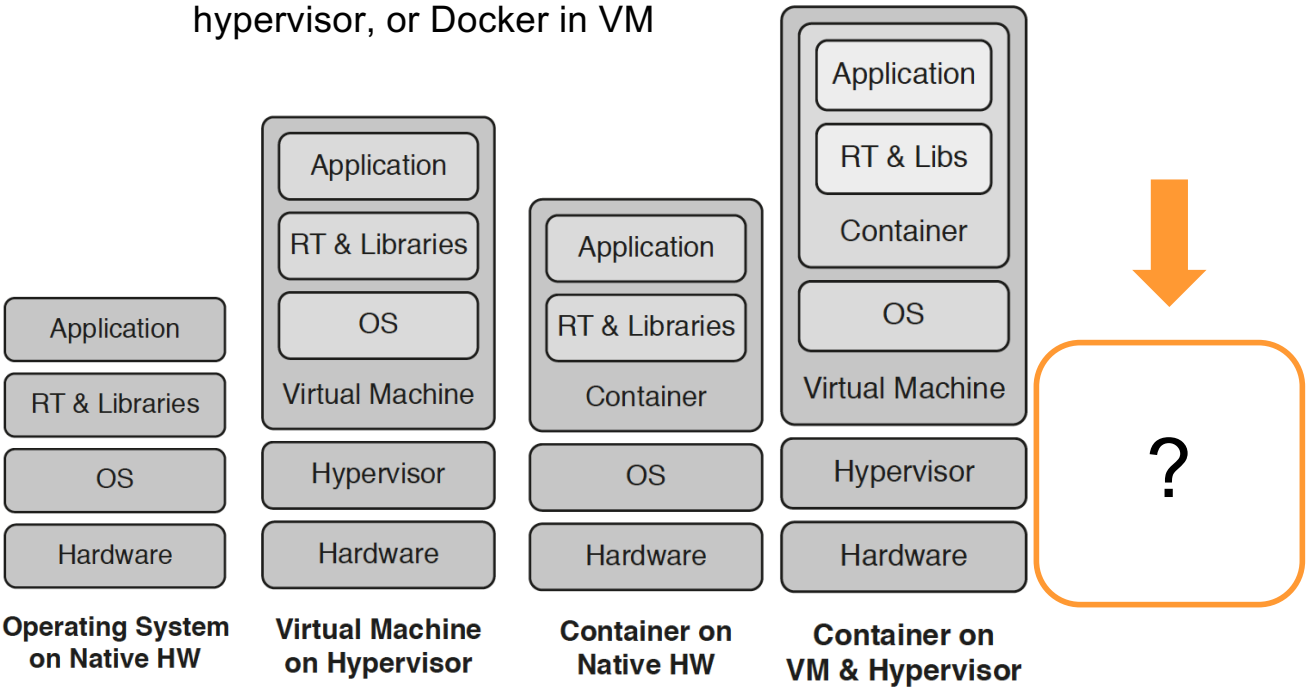
- Cloud-based platform for managing and deploying containerized applications
  - Combines containerization with cloud benefits
- Features
  - Container orchestration and management: container lifecycle, scheduling, load balancing, and fault tolerance
  - Configuration for resource optimization (e.g., auto-scaling)
  - Security and access control
  - Integration with other cloud services (e.g., monitoring)
- Examples
  - Amazon Elastic Container Service https://aws.amazon.com/ecs
  - Azure Container Instances https://azure.microsoft.com/products/container-instances
  - Google Cloud Run https://cloud.google.com/run

Virtualization and IaaS providers

- Which virtualization technology for IaaS providers?
  - ✓ Hypervisor-based virtualization: greater security and isolation, flexibility (different OSs on same PM)
  - ✓ Container-based virtualization: smaller deployment size and higher density, faster startup/shutdown
- Questions
  - Containers on top of bare metal or inside VMs?
    - Performance and density vs. isolation and security
  - Are containers replacing VMs?
    - Not entirely: lightweight vs. isolation and security

New lightweight virtualization approaches

- Deployment approaches examined
  - Plus nested virtualization: hypervisor inside VM on top of an hypervisor, or Docker in VM

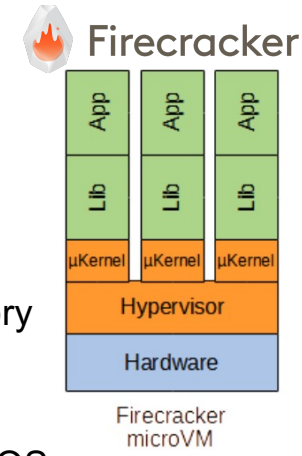


New lightweight approaches to virtualization

- Microservices, serverless computing, cloud-edge continuum demand for
  - Low-overhead (or lightweight) virtualization techniques, even lighter than containers
  - Better security
  - Portability across OSs and architectures (e.g., Arm, Intel)
- Technologies enabling lightweight virtualization: **MicroVM**, **lightweight OSs**, **unikernels** and **WebAssembly**
- MicroVMs, lightweight OSs, and unikernels: reduce OS overhead and attack surface
  - OS overhead: many common OS services (shells, editors, core utils, package managers) are unnecessary
  - Attack surface: images contain only the essential code needed to run the app, reducing potential attack vectors

MicroVM runtimes

- Tiny, specialized VMM that run lightweight VMs (called microVMs)
- Goal: reduce memory footprint and improve security of virtualization layer
- Firecracker: minimalist VMM purpose-built by Amazon for secure, efficient and multi-tenant microVMs
  https://firecracker-microvm.github.io
  - Why? To enable AWS Lambda and AWS Fargate for serverless and containerized workloads
  - Based on KVM but with minimalist design (no unnecessary devices and guest functionality)
  - Open source, written in Rust
  - microVM: <125 ms startup time and <5 MB memory footprint
  - Scales to thousands of multi-tenant microVMs
  - Supported OS guests inside microVM: Linux and OSv

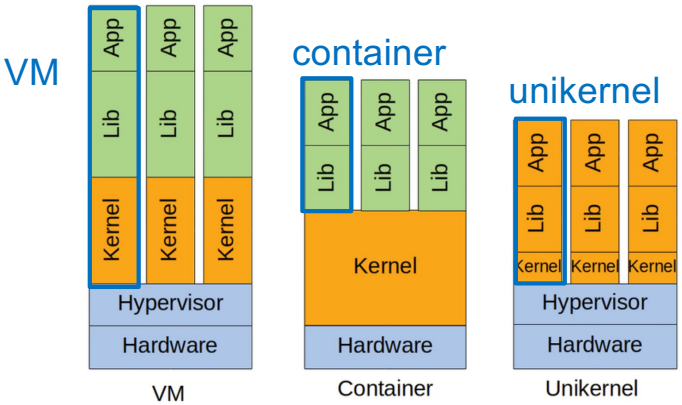


Lightweight operating systems

- Minimal, special-purpose OSs to run containerized apps
- Fedora CoreOS https://fedoraproject.org/it/coreos/
  - Minimal, monolithic, and compact Linux distribution designed for running containers
    - Only components for container deployment, together with built-in tools for service discovery, container management, and configuration sharing
  - Designed for scale and security
  - Fast bootstrap and small memory footprint
  - Can be installed directly on hardware or on hypervisor
  - Includes Docker and podman
- Other products: Ubuntu Core, balenaOS
  - Designed for edge and IoT devices

Unikernels

- Specialized, single-purpose OS designed to run a single application with minimal overhead
  - Single application + OS into a single executable (aka library OS): monolithic process that runs entirely in kernel mode
  - Single address space: app and OS share the same memory space
  - Built by compiling high-level language directly into specialized machine image that runs on hypervisor
  - Goal: isolation benefits of hypervisor without overhead of guest OS



Unikernels: pros and cons

- Pros (specialized → high performance)
  - ✓ Lightweight: less resource-intensive (memory and CPU)
  - ✓ Minimal footprint
  - ✓ Reduced attack surface
  - ✓ Fast execution (no context switching)
  - ✓ Fast boot (measured in ms)
  - ✓ Strong isolation
- Cons
  - ✗ Less flexible than VMs and containers
  - ✗ Limited ecosystem support, including debugging and monitoring tools
  - ✗ Scalability is more challenging (e.g., multi-instance deployments or load balancing)
  - ✗ Unikernel orchestration is not as easy as container orchestration

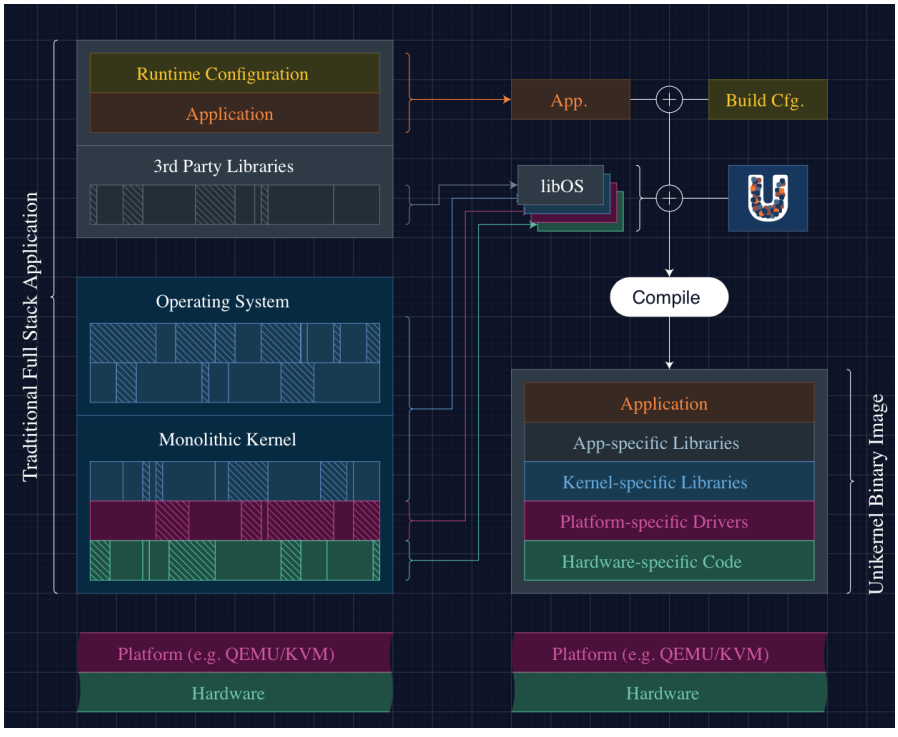
Unikernels: frameworks

- Frameworks
  - MirageOS (Ocaml language) https://mirage.io
  - OSv https://github.com/cloudius-systems/osv
  - Nanos https://nanos.org/
  - Unikraft
- OSv
  - Cloud-native unikernel optimized for high performance and low overhead in virtualized environments
  - Linux ABI compatibility allows running Linux applications with minimal changes
  - Open-source and fast
    - Can boot in ~5 ms on Firecracker using 11 MB of memory

Unikernels: Unikraft

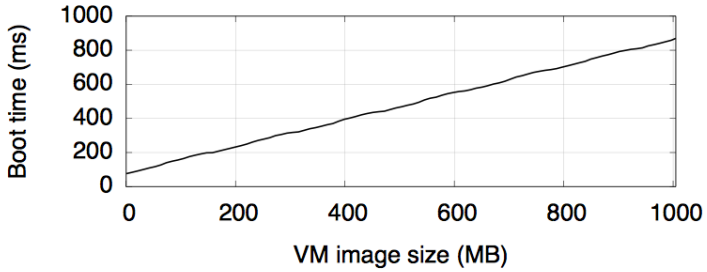
- Fast, secure, and open-source Unikernel Development Kit https://unikraft.org
  - Designed to simplify unikernel creation without requiring deep expertise
  - Build, run and package: similar to Docker
  - Modular: supports a wide range of components (e.g., networking, storage), allowing to create custom unikernels
  - Multi-language support (e.g., C, C++, Rust, Go)
  - Application compatibility
    - Can run complex apps (e.g., Redis, Nginx, Memcached)
    - POSIX compliant: compatibility with a broad range of Unix-like applications
  - Architecture compatibility
    - Works with multiple hypervisors (e.g., Xen, KVM) and supports various CPU architectures, allowing deployment on both virtualized and bare-metal environments
  - Active development

Unikernels: Unikraft

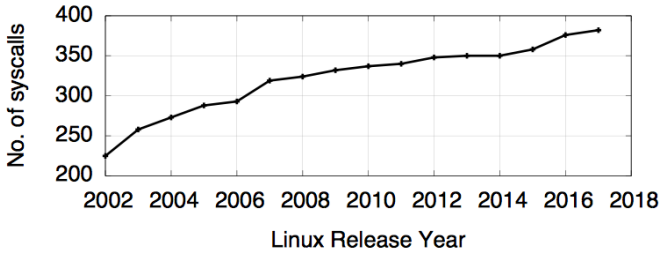


Performance of virtualization approaches

- VM boot times grow linearly with VM size



- Difficulties in securing containers due to growth of Linux syscall API



My VM is lighter (and safer) than your container, SOSP 2017
https://dl.acm.org/doi/pdf/10.1145/3132747.3132763

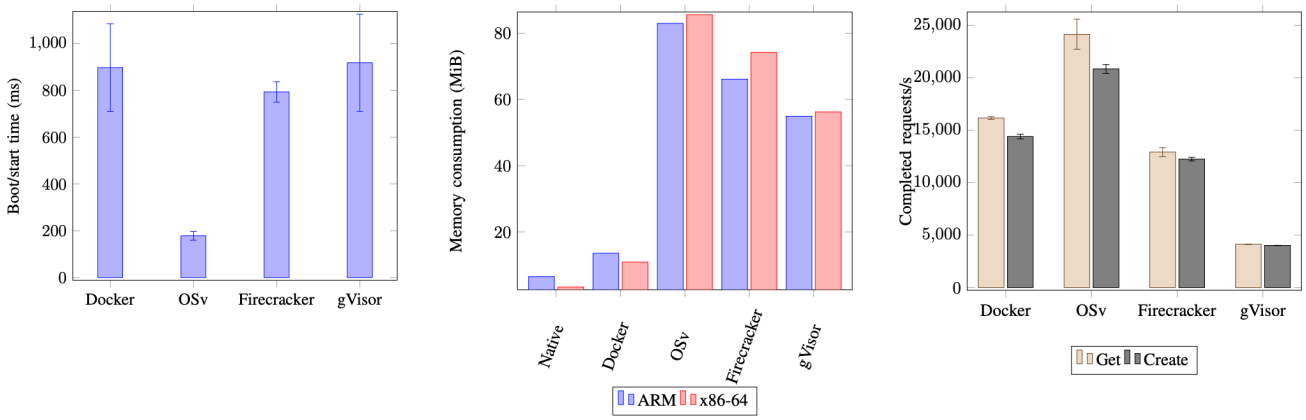
Performance of virtualization approaches

- Performance comparisons of hypervisor vs. lightweight virtualization
- Key findings:
  - Overhead introduced by containers is almost negligible
  - Fast instantiation time of containers
  - Small per-instance memory footprint
  - High density
  - But security tradeoffs: containers offer less isolation

Virtualization	Boot time	Image size	Memory footprint	Programming language dependance	Live migration
VM	~5/10 sec	~1 GB	~100 MB	No	Yes
Container	~0.8/1 sec	~50 MB	~5 MB	No	Non-native
Unikernel	<10 msec	<20 MB	~10 MB	Partially	No

Performance of virtualization approaches

- Comparing lightweight virtualization approaches
- Overall result: no clearly superior solution, each one has its own strengths and weaknesses



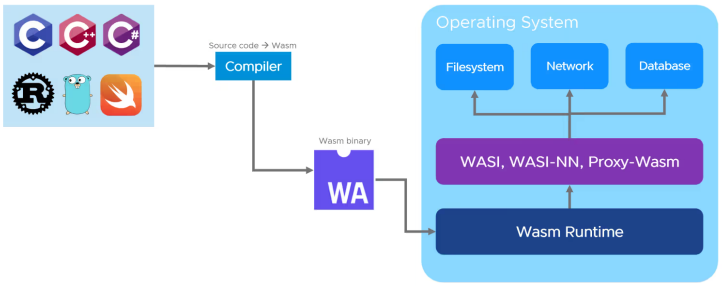
A functional and performance benchmark of lightweight virtualization platforms for edge computing, EDGE 2022 https://ieeexplore.ieee.org/document/9860335

WebAssembly (Wasm)

- Safe, portable, binary code format designed for efficient execution and compact representation https://webassembly.org
  - Safe: runs in a sandboxed environment, preventing untrusted code from harming the host system
  - Portable: the same Wasm binary can run on any platform that supports the Wasm runtime (browsers, servers, embedded)
  - Binary format: designed to be compact and fast to load and execute
- Other features
  - Open standard https://www.w3.org/TR/wasm-core-2
  - Portable compilation target for many programming languages
    - Originally built to safely execute JavaScript code in browsers
  - Memory-safe, sandboxed execution
  - Computational model based on stack VM

WebAssembly: features

- Wasm code is validated and executed in a memory-safe, sandboxed environment → strong isolation and protection from unsafe operations
  - Wasm interacts with the host system through WebAssembly System Interface (WASI), which provides a standardized set of capability-based APIs
  - A Wasm module cannot directly perform OS systems calls, instead imports host-provided WASI functions
- Development workflow: write code in one of many supported languages, compile it to Wasm, and run it inside a Wasm runtime



WebAssembly: features

- Wasm uses stack-based VM to execute instructions https://en.wikipedia.org/wiki/Stack_machine
  - Code is composed of sequences of instructions executed in order
  - The operand stack is used to store values for computation
  - Instructions:
    - Pop argument values from the stack and push back onto it
    - Each operation manipulates values on the stack
    - Example: `i64.add`
      - Takes two i64 values from the stack
      - Add them
      - Pushes the result back onto the stack
    - Control instructions alter control flow
- JVM is a famous example of stack-based VM

WebAssembly: example

- Factorial function written in C and its corresponding Wasm code after compilation
  - In `.wat` text format (human-readable textual representation of Wasm)

```c
int factorial(int n) {
  if (n == 0)
    return 1;
  else
    return n * factorial(n-1);
}
```

```
(func (param i64) (result i64)
  local.get 0             # put arg[0] on stack
  i64.eqz                 # compare top of stack to zero
  if (result i64)         # if it is zero
      i64.const 1         # put 1 on stack
  else
      local.get 0         # put arg[0] on stack
      local.get 0         # put arg[0] on stack
      i64.const 1         # put 1 on stack
      i64.sub             # subtract the top 2 values in stack
      call 0              # Call function #0 (return value is on the stack)
      i64.mul             # Multiply
  end)'
```


WebAssembly: example

- Factorial function written in C and its corresponding Wasm code after compilation
  - In `.wasm` binary format

```c
int factorial(int n) {
  if (n == 0)
    return 1;
  else
    return n * factorial(n-1);
}
```

```
00 61 73 6D 01 00 00 00
01 06 01 60 01 7E 01 7E
03 02 01 00
0A 17 01
15 00
20 00
50
04 7E
42 01
05
20 00
20 00
42 01
7D
10 00
7E
0B
0B
```

WebAssembly: pros and cons

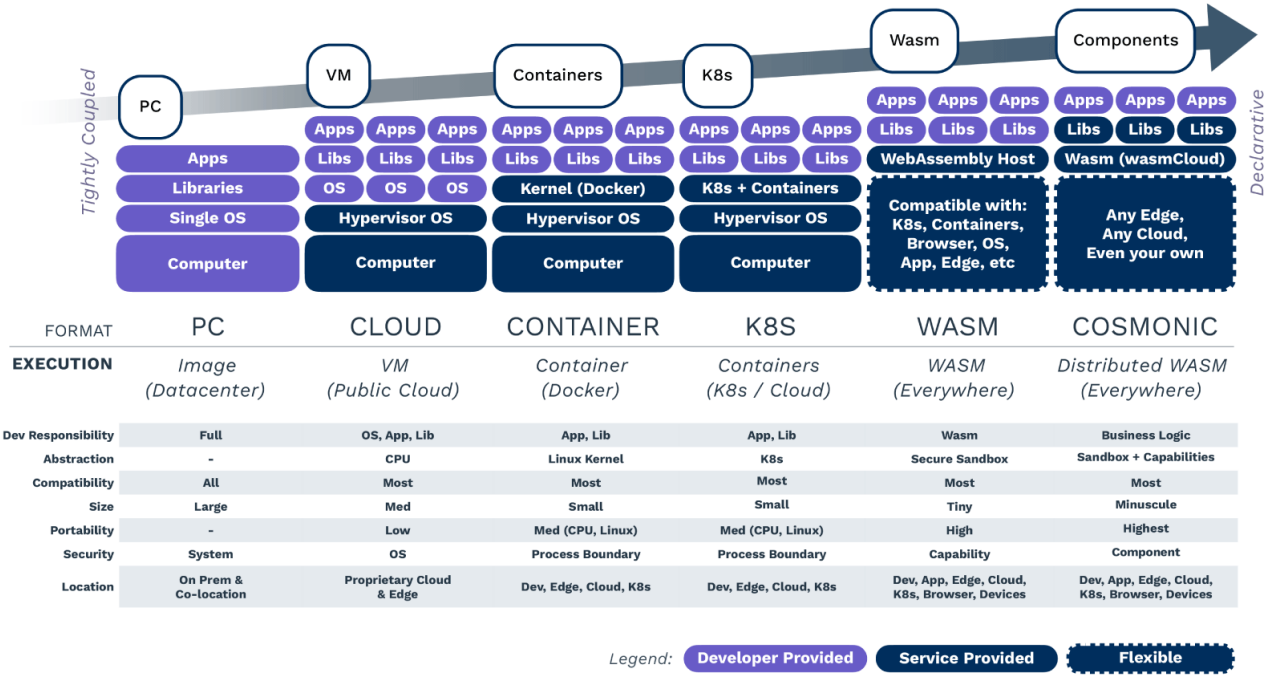
- ✓ Efficient: near-native execution speed
- ✓ Secure: memory-safe, sandboxed execution, which prevents data corruption and security breaches
- ✓ Language-, platform-, and hardware-independent
  - Does not favour any particular language
  - Can run as a standalone VM
  - Can be compiled for all modern architectures, including desktop, mobile devices, and embedded systems
- ✗ In development
- ✗ Support varies by language
- ✗ Multiple runtimes (e.g., Wasmtime, Wasmer, WasmEdge) with different features: choice is complex

WebAssembly

- How to try: Wasm applications with Linux containers in Docker (beta)
  https://docs.docker.com/desktop/features/wasm/
  - Enable it on Docker Desktop by checking *Enable Wasm* on the *Features in development* tab under *Settings* (requires containerd image store)

```
$ docker run \
  --runtime=io.containerd.wasmedge.v1 \
  --platform linux/arm64 \
  secondstate/rust-example-hello
```

The full scenario



FORMAT	PC	CLOUD	CONTAINER	K8S	WASM	COSMONIC
EXECUTION	*Image (Datacenter)*	*VM (Public Cloud)*	*Container (Docker)*	*Containers (K8s / Cloud)*	*WASM (Everywhere)*	*Distributed WASM (Everywhere)*
Dev Responsibility	Full	OS, App, Lib	App, Lib	App, Lib	Wasm	Business Logic
Abstraction	-	CPU	Linux Kernel	K8s	Secure Sandbox	Sandbox + Capabilities
Compatibility	All	Most	Most	Most	Most	Most
Size	Large	Med	Small	Small	Tiny	Minuscule
Portability	-	Low	Med (CPU, Linux)	Med (CPU, Linux)	High	Highest
Security	System	OS	Process Boundary	Process Boundary	Capability	Component
Location	On Prem & Co-location	Proprietary Cloud & Edge	Dev, Edge, Cloud, K8s	Dev, Edge, Cloud, K8s	Dev, App, Edge, Cloud, K8s, Browser, Devices	Dev, App, Edge, Cloud, K8s, Browser, Devices

Legend: Developer Provided | Service Provided | Flexible

References

- Sections 4.13 of Marinescu book
- Docker workshop https://docs.docker.com/get-started/workshop
- Docker Docs https://docs.docker.com
- Kane and Matthias, Docker up and running 3rd edition, O'Reilly, 2023
- Agache et al., Firecracker: Lightweight virtualization for serverless applications, NDSI 2020 https://www.usenix.org/conference/nsdi20/presentation/agache
- Kuenzer et al., Unikraft: fast, specialized unikernels the easy way, EuroSys 2021 https://dl.acm.org/doi/pdf/10.1145/3447786.3456248
- Menétrey et al., WebAssembly as a common layer for the cloud-edge continuum, FRAME 2022 https://dl.acm.org/doi/10.1145/3526059.3533618