

Containers and Orchestration: a Security Perspective



Mathieu Bacou
`mathieu.bacou@telecom-sudparis.eu`
Télécom SudParis, IMT, IP Paris, Inria

Cloud applications

- Traditional applications and **monolithic**
 - Everything is tightly coupled
 - On full servers, managed from OS to deployment
 - This is a constraint on the user
- In the cloud, you don't manage real servers
 - You share servers, thanks to virtualization
 - You get new resources ("servers") on-the-fly
- We can drive this model further!

Cloud native applications

- **No OS** management by the user
- **Component-level** application scalability

Introducing: containers

- Cloud users do not want to run OSes
 - They want to run their **applications**
- How to share cloud resources closer to the applications?
 - Virtualization layer just between the OS and the application
- Virtualize the OS for multiple applications at the same time!
 - In other words, containers are OS-level virtualization
- An OS executes a **container runtime** that uses a **container engine** to run **containers**
 - Docker, LXC, OpenVZ...

Actors of OS-level virtualization

1. Container runtime
2. Container engine
3. Container

Containers runtimes

- **High-level management** of containers, artifacts and runtime configuration
 - Business-oriented container lifecycle
 - Build, download container images
 - Configure networking, volumes, security, etc.
- A **container image** packages an application and its runtime
 - Business core, dependencies, pre-configuration
- Ecosystem of reusable images stored in **registries** (DockerHub, GitLab registries, local registry...)
 - Images are built immutable for portability, reusability and composability
- Examples: Docker (containerd), Podman, Apptainer...

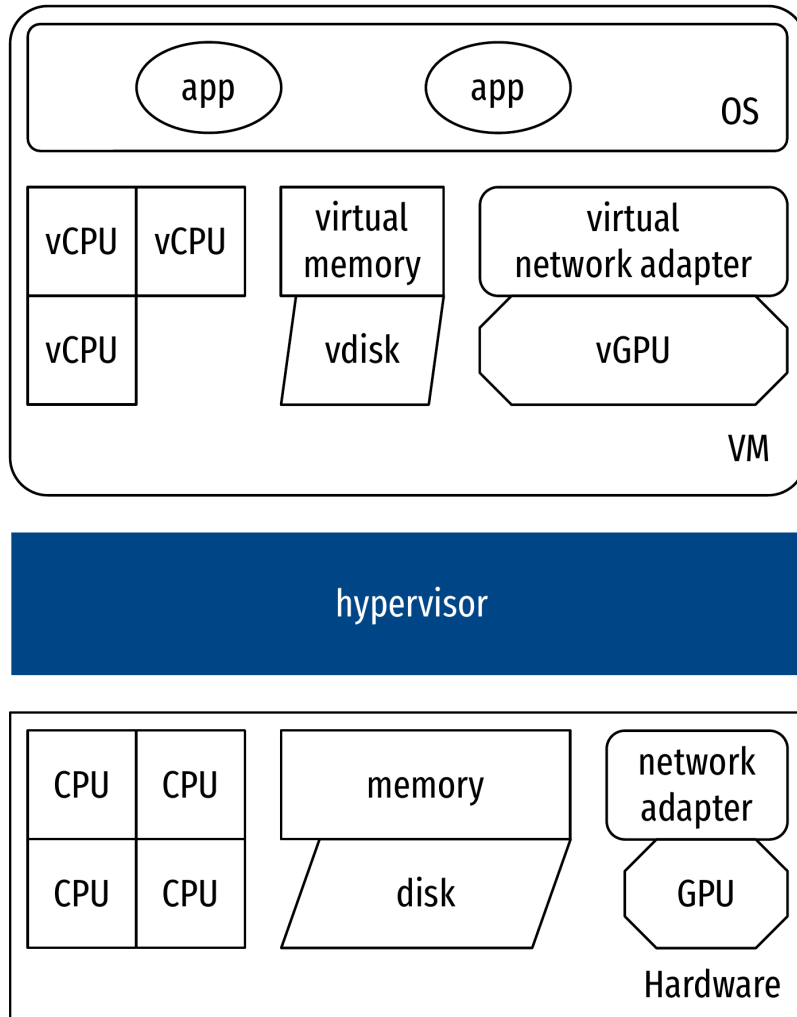
Container engines

- **Low-level management** of containers
 - Create, start, stop, destroy...
 - Prepare images for usage
 - Last-mile setup of networking, mounts, security...
- Different engines for different usages or orientations: generic, security-oriented, scientific...
- Examples: runc, Kata Container, gVisor, wasmtime...

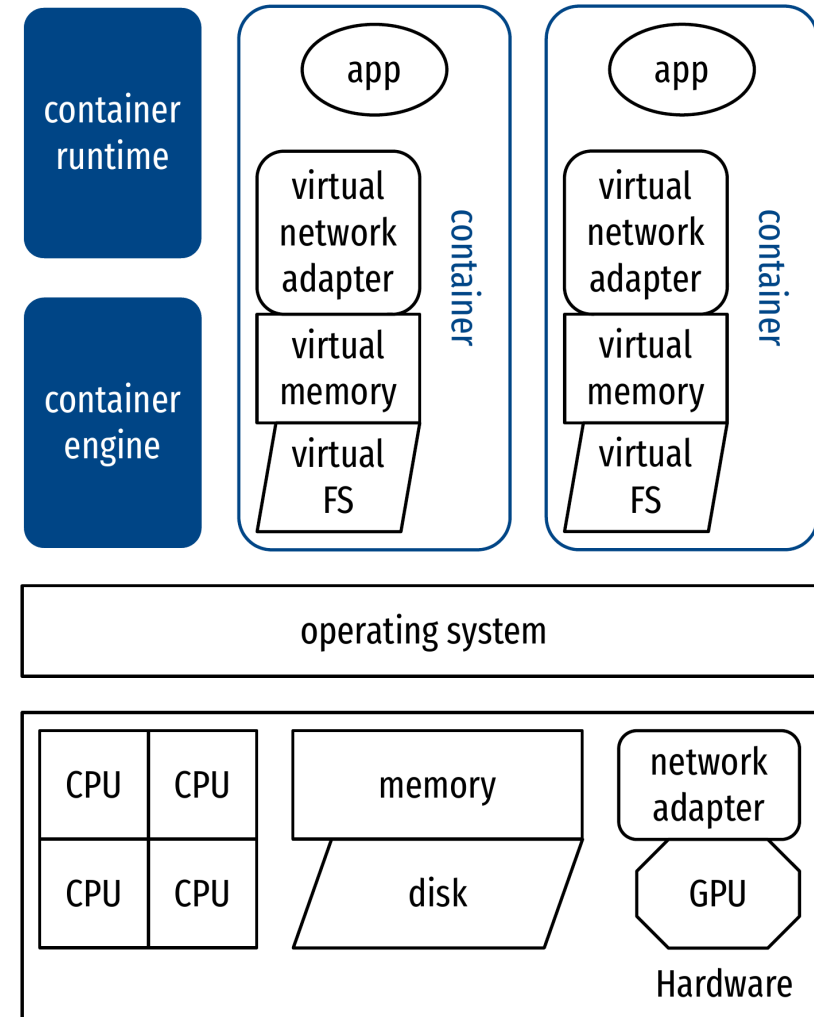
Containers

- **Container:** isolated and limited virtual copy of the host OS
 - Deploys the image to “fill in” the virtual copy
- **Isolation:** users, devices, processes...
 - Virtual filesystem: built from container image
- **Limits:** CPU, memory, I/O...
 - Also monitoring

Comparison with hardware virtualization: stack



Stack for hardware virtualization.



Stack for OS-level virtualization.

Comparison with hardware virtualization: features

Comparison of features between hardware and OS-level virtualization techniques.

	Operating system-level virtualization	Hardware virtualization
Security	-	+
Usability	++	-
Performance	0	0
Startup time	+	--
Image size	+	--
Memory overhead	+	--

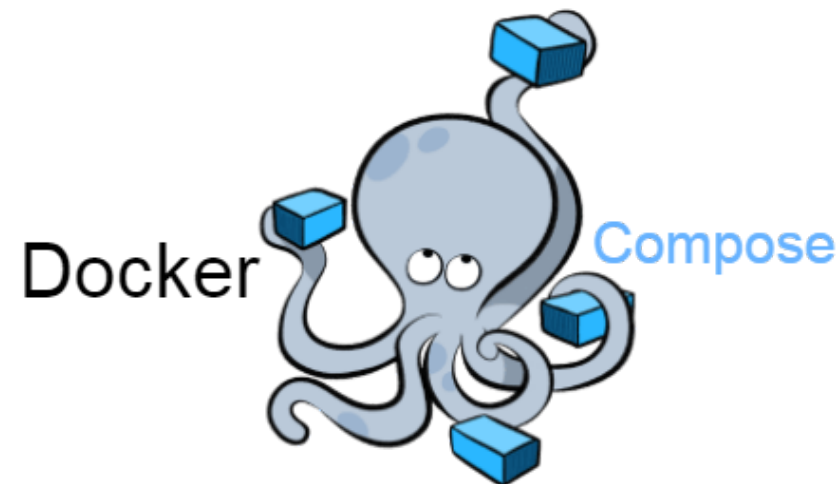
- Containers are better overall for **cloud-native applications**
 - Applications are architected to be deployed on the cloud
- Security concerns
 - Kernel shared between containers
- **VMs still have use cases:** persistent, interactive environments, robustness, first-level resource provisioning...

Demo: Docker

- Creation and usage of a Docker container
 - Run an interactive image
 - Pull and run a daemon service
 - List images, monitor containers
- Docker is rather “low-level” for applications: compose multiple components (containers) in a single application with Docker Compose



Docker logo.



Docker Compose logo.

Security of containers

1. Isolation
2. Threat models and vectors
3. Good practices

Isolation

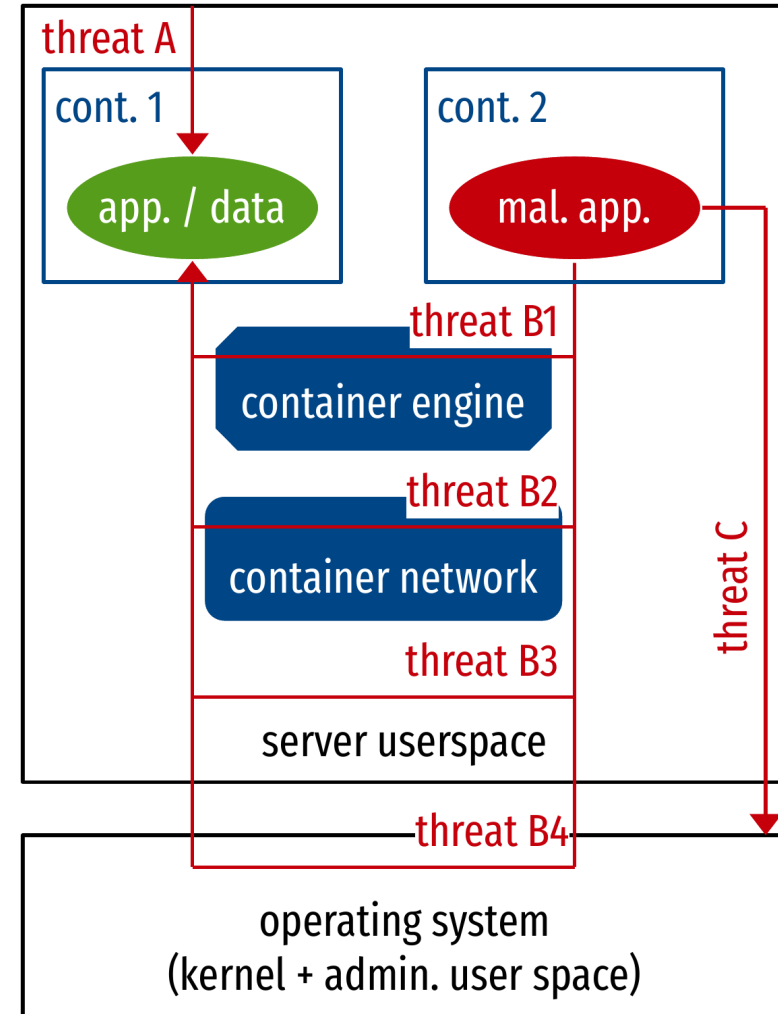
- Fundamental issue for cloud providers: execute untrusted code
- When using containers: **tenants and provider share the kernel**
 - No mitigation when the kernel is compromised
 - Incompatibility of kernel-level security policies
 - Because security measures are mostly not namespaces (AppArmor, etc.)
 - Vast attack surface and trusted code base
 - Virtual Machines (VMs) are better in this regard (hypervisor interface vs. whole host kernel)
- **Isolation** of untrusted code
 - To protect containers from each other
 - To protect the system from containers

Threat models: goals and targets

- Attack goals:
 - **Disrupt services**: bad neighbor, denial of service
 - **Subvert services**: steal identity, steal resources
 - **Steal data**
- Targets are cloud-oriented: applications are **regular services**
 - Mostly web servers or applications accessed by HTTP requests
 - But sometimes containers include system services
 - SSH, cron jobs, logs...
 - More privileged requirements => more care!

Threat models: overview

1. direct attack from outside, on the containerized application
2. indirect attack from a container, on another containerized application
3. attack from a container, on the host system



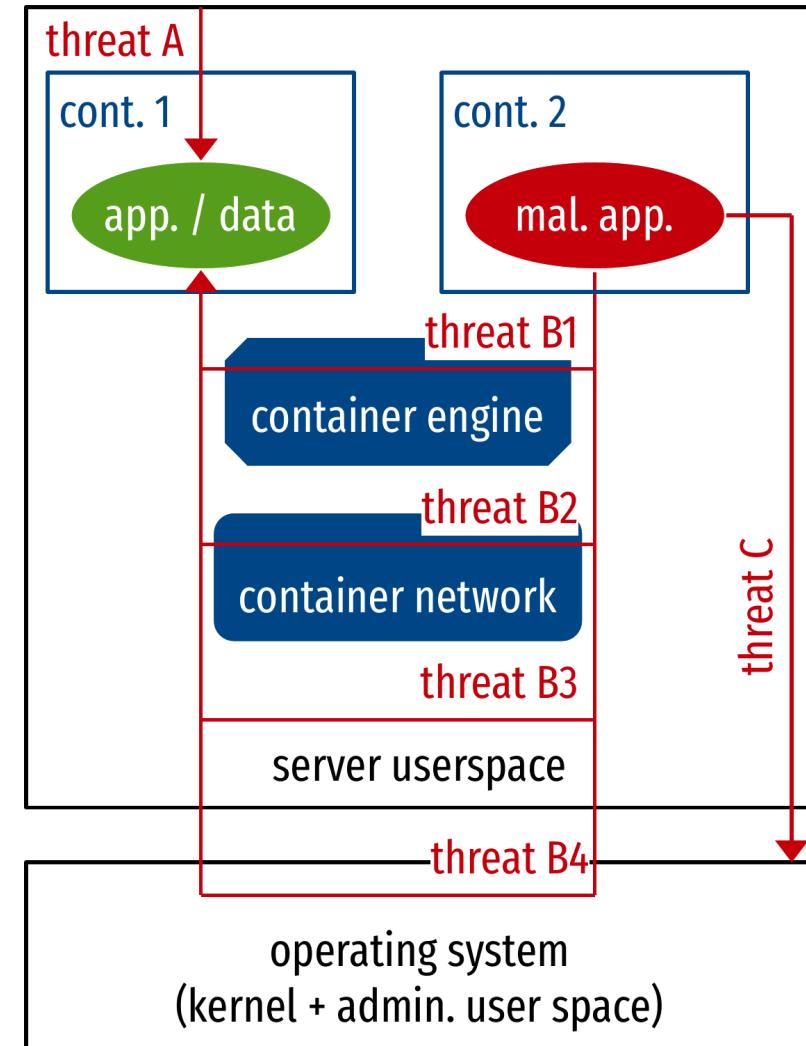
Threat models of containerization.

Threat model (A): direct attack from outside

- Containers are not security magic for Internet-facing applications!
 - A vulnerable web server remains vulnerable
- But containers help against vulnerabilities:
 - **Breach containment**
 - Importance of the micro-service model
 - **Safe configuration** is easier to achieve
 - More secure defaults
 - Fewer configuration items to tweak thank to virtualized environment
 - **Simpler audit**
 - Limited set of dependencies and software pieces
 - Fast, easy **distribution of security updates**
 - Container distribution model: generic images pulled from a centralized place
- Regarding the distribution model: it can also be a threat vector (attacks on the distribution channel)
 - Do not pull unaudited images or updates: may be freshly **vulnerable or compromised**
 - Malicious updates or owner, typosquatting...
 - Use a private repository of audited images

Threat model (B): indirect attack between containers

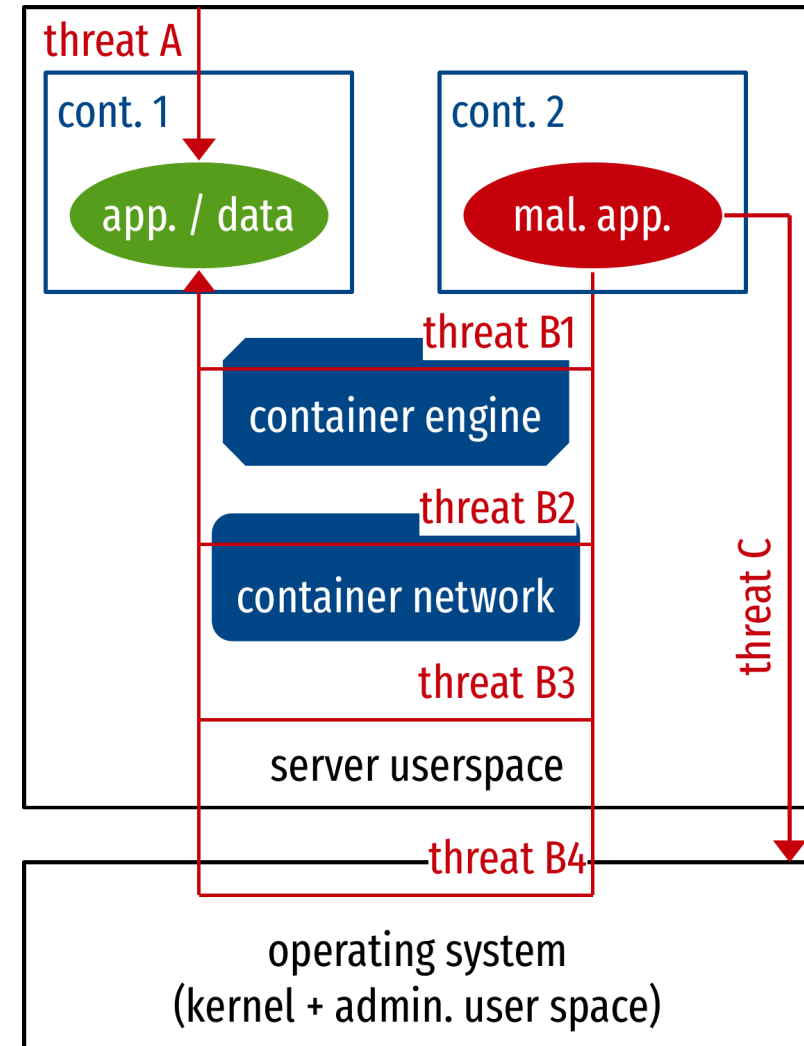
- Containers run **arbitrary code** by definition
- B1: **Escape to another container**
 - Bug in namespaces implementation
 - Leaks in the filesystem
- B2: **Abuse of the container network**
 - Packet forging
 - Layer 2 attacks
- B3: **Escalation to root**
 - Vulnerable SUID binaries
 - Vulnerable container engine implementation
- B4: **Execute arbitrary kernel code**
 - Exploitable system calls
- Not always container-specific



Threat models of containerization.

Threat model (C): attack on the host system

- Containers run **arbitrary code** by definition
- **Escape containment**
 - Namespace bug
 - Filesystem leak
- **Escalate to root**
 - Vulnerable SUID binaries
 - Vulnerable container engine implementation
- **Execute arbitrary kernel code**
 - Exploitable system calls
- Not always container-specific



Threat models of containerization.

Good practices: as a user

- **Audit public images**
 - Fix versions, but monitor for security updates
- Use the micro-services architectures
 - For intrusion detection and containment
 - Every micro-service can be augmented with its own monitor
- Mount volumes read-only when possible
 - Container images are already immutable thanks to the overlay FS
- Drop capabilities
 - Many are dropped by default, but more can usually be dropped

Good practices: as an image developer

- Use and build **immutable container images**
 - I.e., images that can be deployed identically everywhere, only configured to fit the environment
 - Example: do not build images that download binaries when starting
- **Do not run as root** in the container (non root images)
 - User namespaces allow that, but this is not an excuse
 - Do not rely on SUID binaries in general

Good practices: as a system administrator

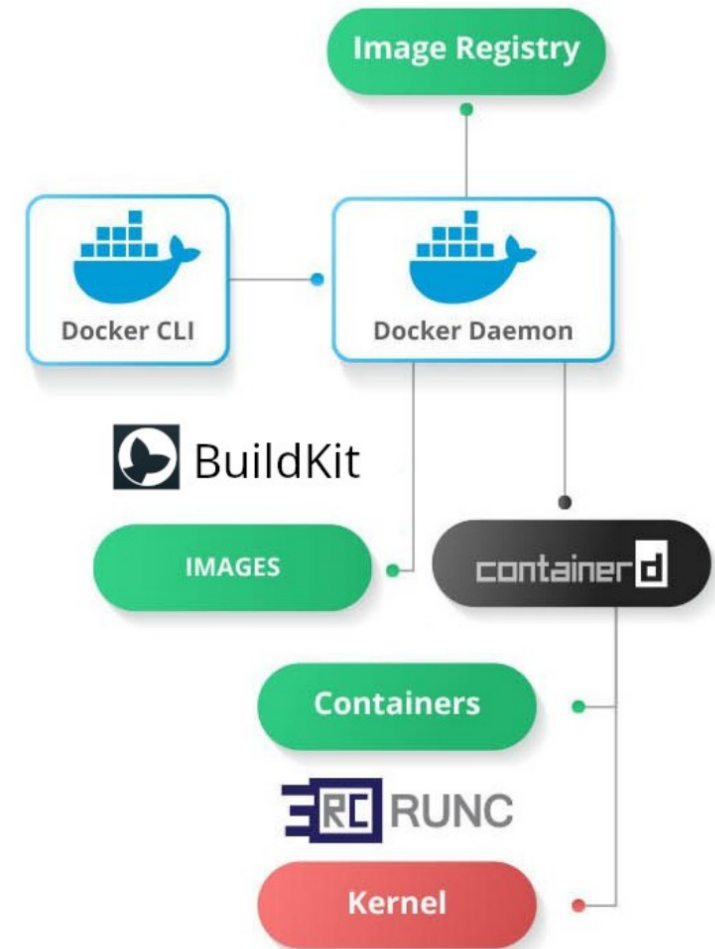
- **Harden the kernel**
 - Enable MAC: Linux Security Modules (AppArmor, SELinux, etc.), seccomp...
 - Not really containerization-aware but still very well usable
 - Use a hardened kernel (GRSEC...)
 - Update the kernel to vetted versions
 - It is a critical part, because it is shared with all containers (huge trusted codebase between tenants)
- **Configure container networking tightly**
 - Do not use host mode
 - Think about shared network namespaces, open ports, common virtual networks...
 - In practice: managed by docker-compose, Kubernetes, etc.
- Why not go one step beyond: use virtual machines!
 - An application in a Docker container in a virtual machine (*in a container ?*)
 - Kata Containers, gVisor...
 - Or just split physical servers by tenants using VMs

Internals of a container engine

- 0. Overview
- 1. Isolation
- 2. Limit
- 3. Operation control
- 4. Virtual filesystem

Overview

- user interface to all container-related services
- container management
- container monitoring
- implementation of containers



© Bilal Kalem (ios.dz)

Architecture of Docker.

Isolation: namespaces

- Provide an **isolated view** of the OS
- 8 dimensions:

1. **mnt**: mount points
 - Hierarchy of sub-filesystems
2. **pid**: hierarchy of processes
 - The first process in the container gets PID 1
3. **net**: networking facilities
 - Interfaces, ports, protocol stack...
4. **ipc**: interprocess communication
 - SysV IPC mechanisms: semaphores, message queues, shared memory segments
5. **time**: date and time
6. **user**: users, groups and privileges
 - The engine establishes a mapping between host UIDs (GIDs) and in-container UIDs (GIDs)
 - root is defined as UID 0, which is available inside the container: escape the container as root, and you are root on the host!
7. **uts**: hostname and domain name
 - For UNIX TimeSharing, from an era of remote computers and client terminals
8. **cgroup**: control groups (*see next*)

Limit: control groups (cgroups)

- **Constrain resource usage**
 - Also prioritization, accounting, control
- 8 “dimensions” (controllers):
 - `cpu`: CPU time
 - `cpuset`: task placement on memory and CPU nodes
 - `memory`: memory usage
 - `io`: block I/O
 - `pid`: number of PIDs (i.e., of processes)
 - `device`: access to device files
 - *special: only through BPF*
 - `perf_event`: performance monitoring
 - `net`: network packets priority and classes for QoS
- Other specialized controllers: `rdma`, `hugetlb`, `misc`

Operation control: capabilities and MAC

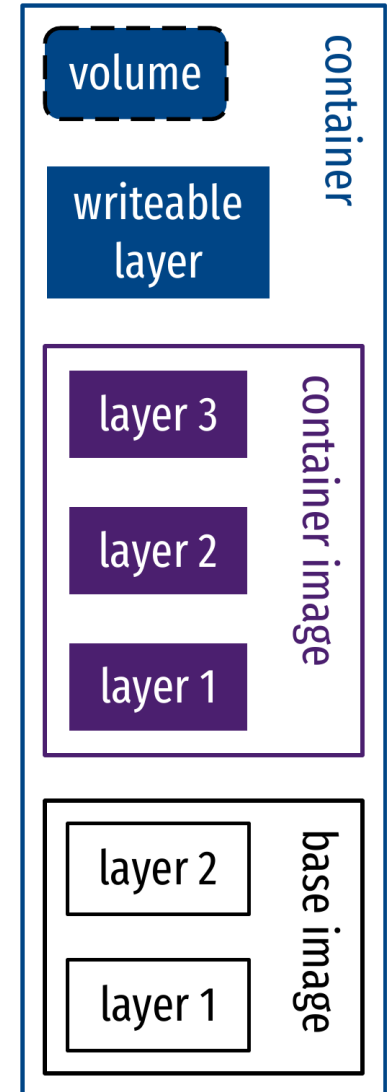
- **Capabilities:** selectively drop root privileges
- **Mandatory Access Control (MAC):** system-level operational policies with Linux Security Modules
 - SELinux, AppArmor, seccomp...
- More than 40 capabilities ([CAP_XXX](#)):
 1. [SYS_NICE](#): change process niceness
 2. [SYS_ADMIN](#): system admin (mount...)
 3. [SYS_CHROOT](#): change root path of process
 4. [SYS_MODULE](#): (un)load kernel modules
 5. [SETGID/UID](#): change process GIDs/UIDs
 6. [KILL](#): send signals
 7. [NET_ADMIN](#): network admin
 8. [NET_RAW](#): use RAW sockets
 9. [CHOWN](#): change owner

Virtual filesystem

- Isolated filesystem:
 - `mnt` namespace to isolate hierarchy
 - `chroot` to isolate the process to a subtree
- Two parts to the filesystem visible to the container:
 1. Container image: basis for virtual filesystem
 - Bundle of files and filesystem operations in layers
 2. Volumes: external data storage
 - Mounted into the virtual filesystem of the running container

Virtual filesystem: layers and volumes

- An image is made of **layers**
 - Like git commits, to represent modifications on the filesystem
 - Reusable by other images, with caching
 - Visible with `docker image history $IMAGE_NAME`
- Layers of an image, built from a Dockerfile, are **read-only**
 - The engine adds a writeable layer on top during container execution
 - Use **copy-on-write** to modify files from lower layers
- Managed by a **union filesystem**: driver of a layered virtual filesystem (Overlayfs)



Container image layers and volumes.

Virtual filesystem: OverlayFS and copy-on-write

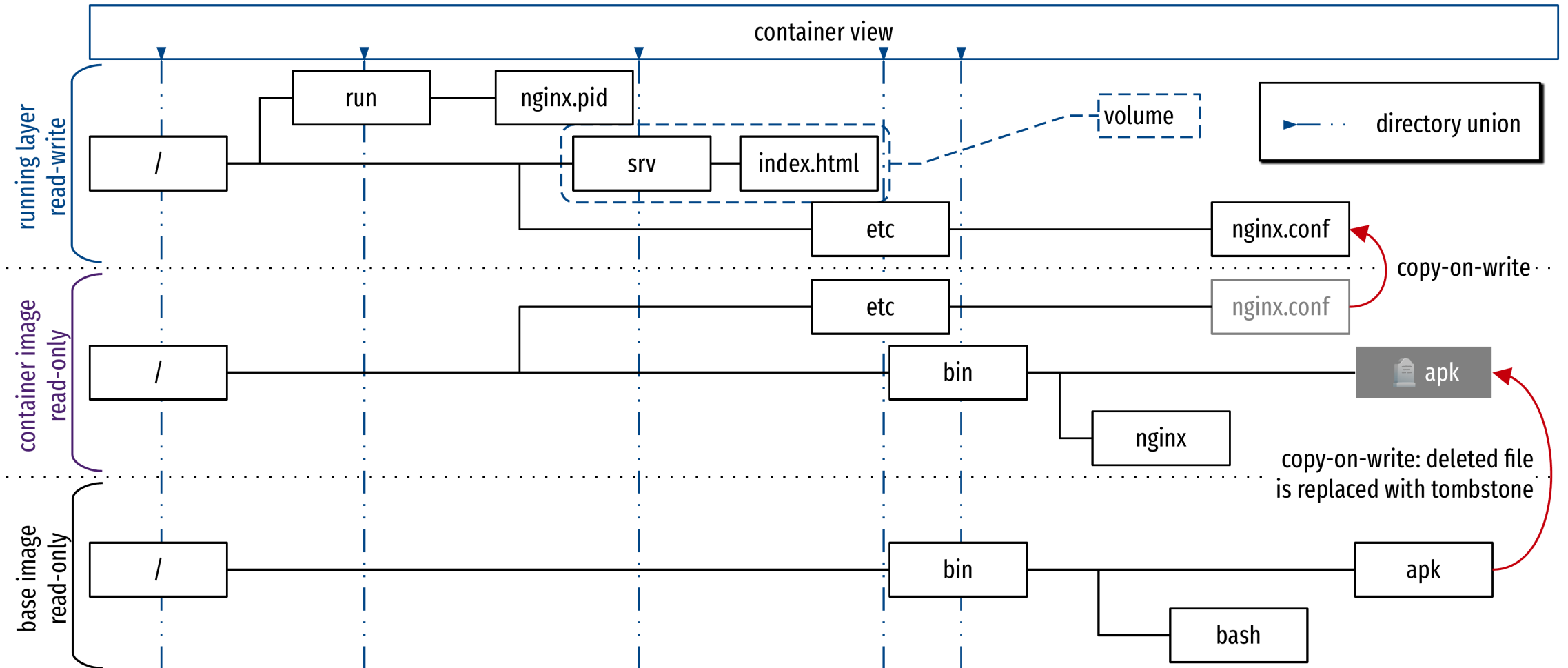
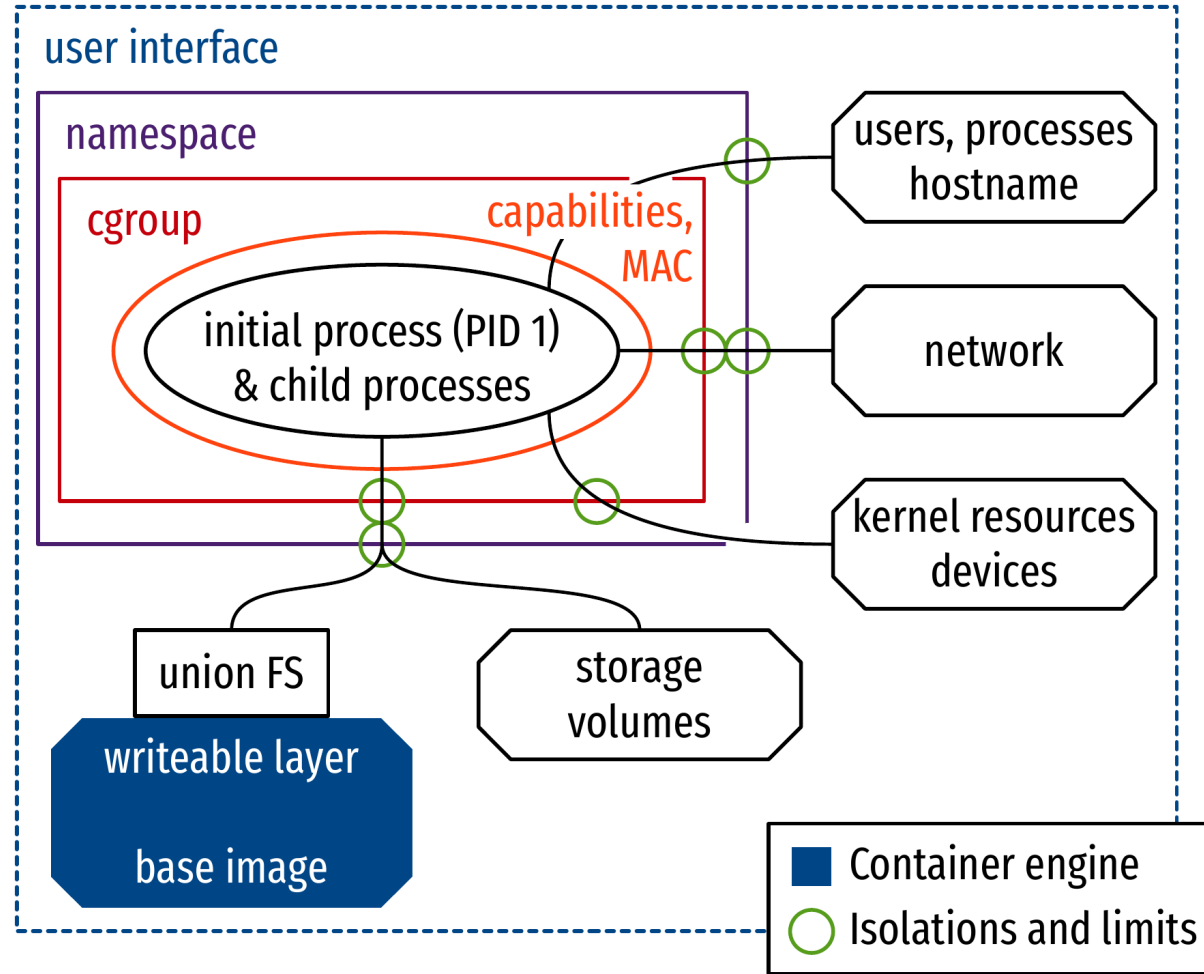


Illustration of a union filesystem and copy-on-write.

Low-level view of a container engine

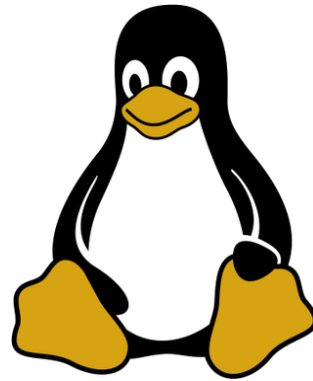


Low-level components and interface of a container engine.

- Most features that make a container, come from the Linux kernel!

Demo: namespaces and cgroups

- Spawn a new process in namespaces
- Put a process in control groups
 - Set limit and monitor resource usage
- Using the virtual filesystem interface
 - There are also syscalls



Linux logo.

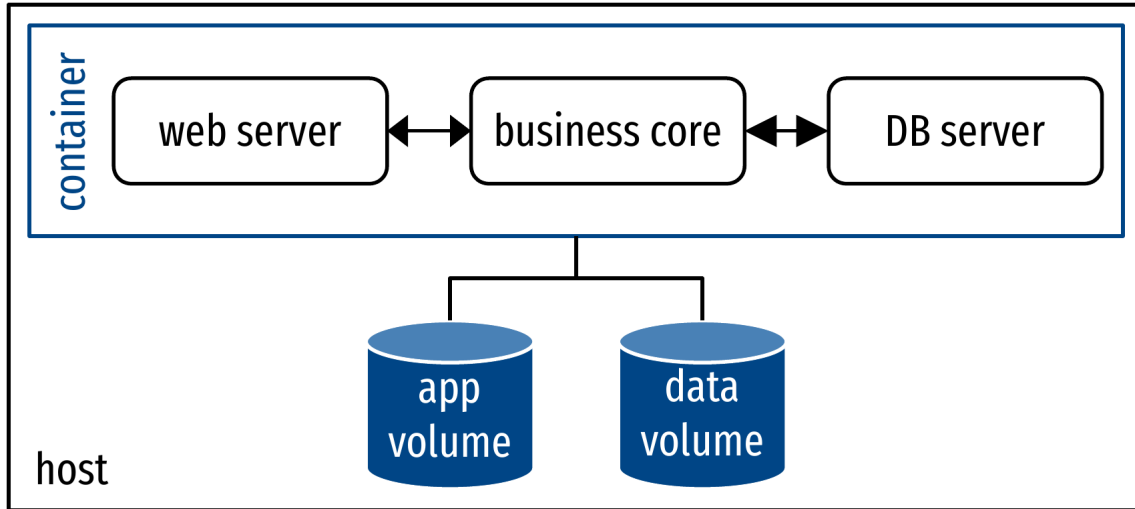
Containers for the cloud

1. Application architecture in the cloud
2. Micro-services
3. Orchestration

Cloud application architecture

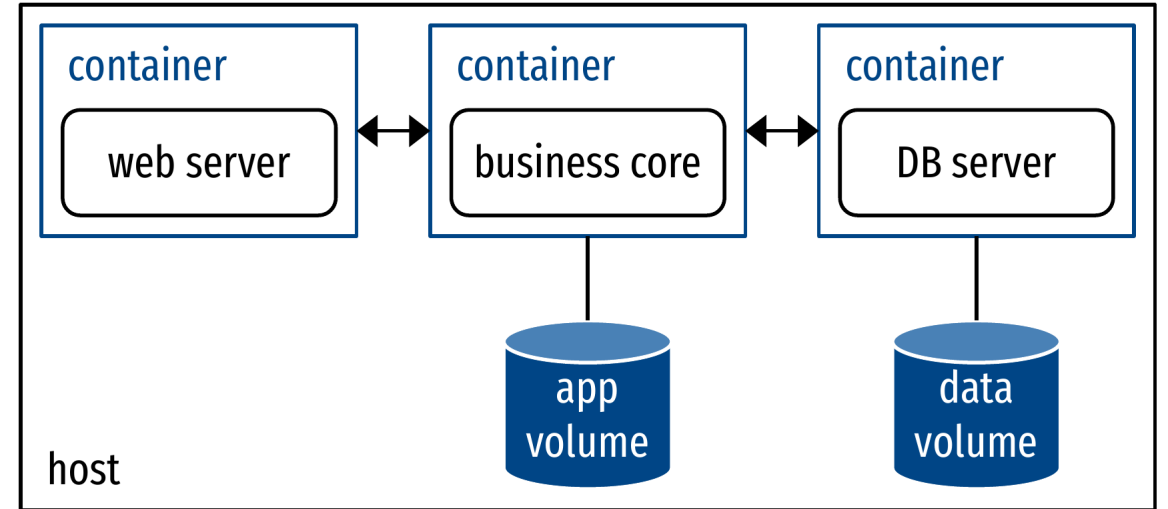
- Historic pattern: **monolithic application**
 - All components are ad-hoc, tightly coupled
- Unfit for the cloud
 - Must manage all components at once for scalability, deployment, service quality
 - Hard to reconfigure
- New paradigm enabled by containers: **micro-services**

Micro-services



✗ Monolithic container.

- Components as processes
 - Manual interfacing
 - Need in-container PID 1 (service manager) to run multiple processes
 - Cons of monolithic apps (*see previous slide*)



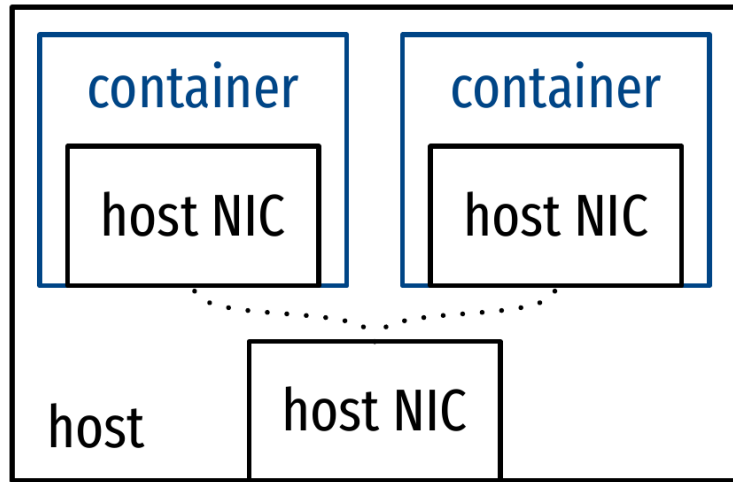
✓ Composition of containers.

- **Components as containers**
 - Maximum reuse of images
 - High flexibility, clean configuration and interfacing
 - Fine-grained scalability

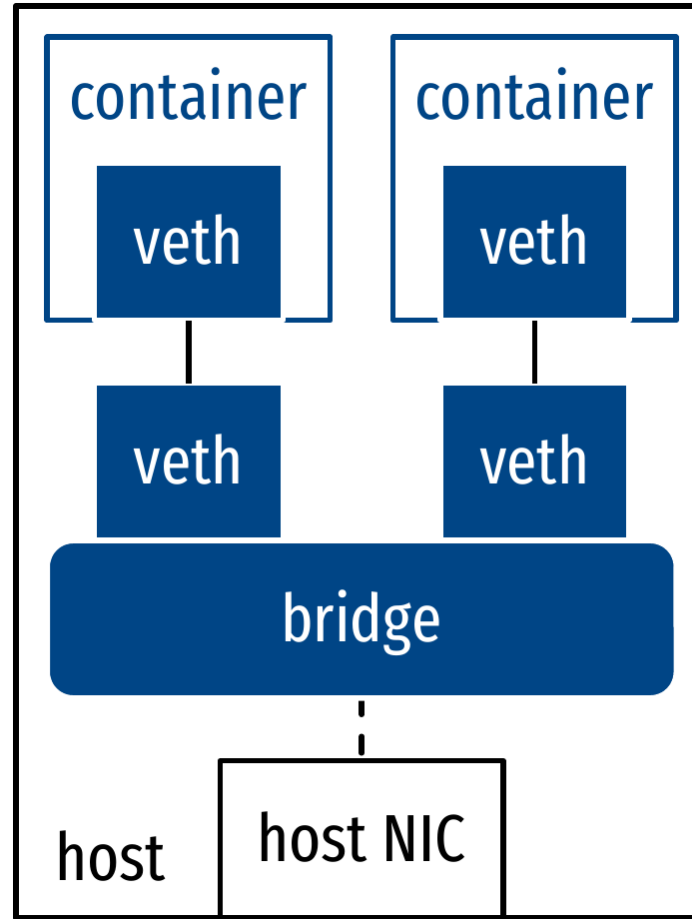
Networking for micro-services

- Configuration of networking by the container runtime
 - **Dedicated links** between component containers
 - Controlled link to the outside world
- Network drivers:
 - Host: expose host network devices to the container (no isolation)
 - Bridge: local virtual network
 - May be exposed to the outside world via virtual routing
 - Overlay: inter-host inter-container network
 - *None: no networking at all*

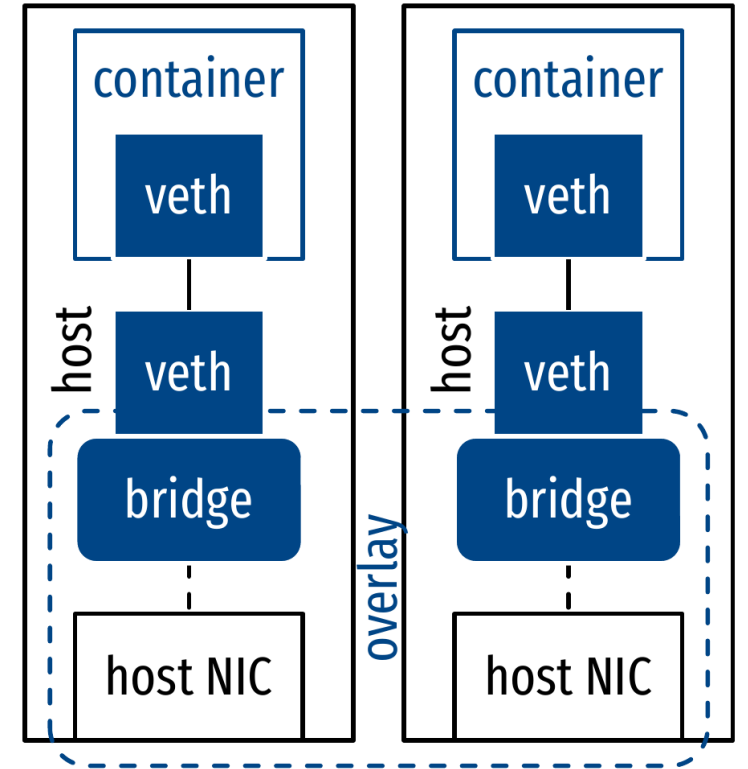
Networking for micro-services: illustrations



Host.



Bridge.



Overlay.

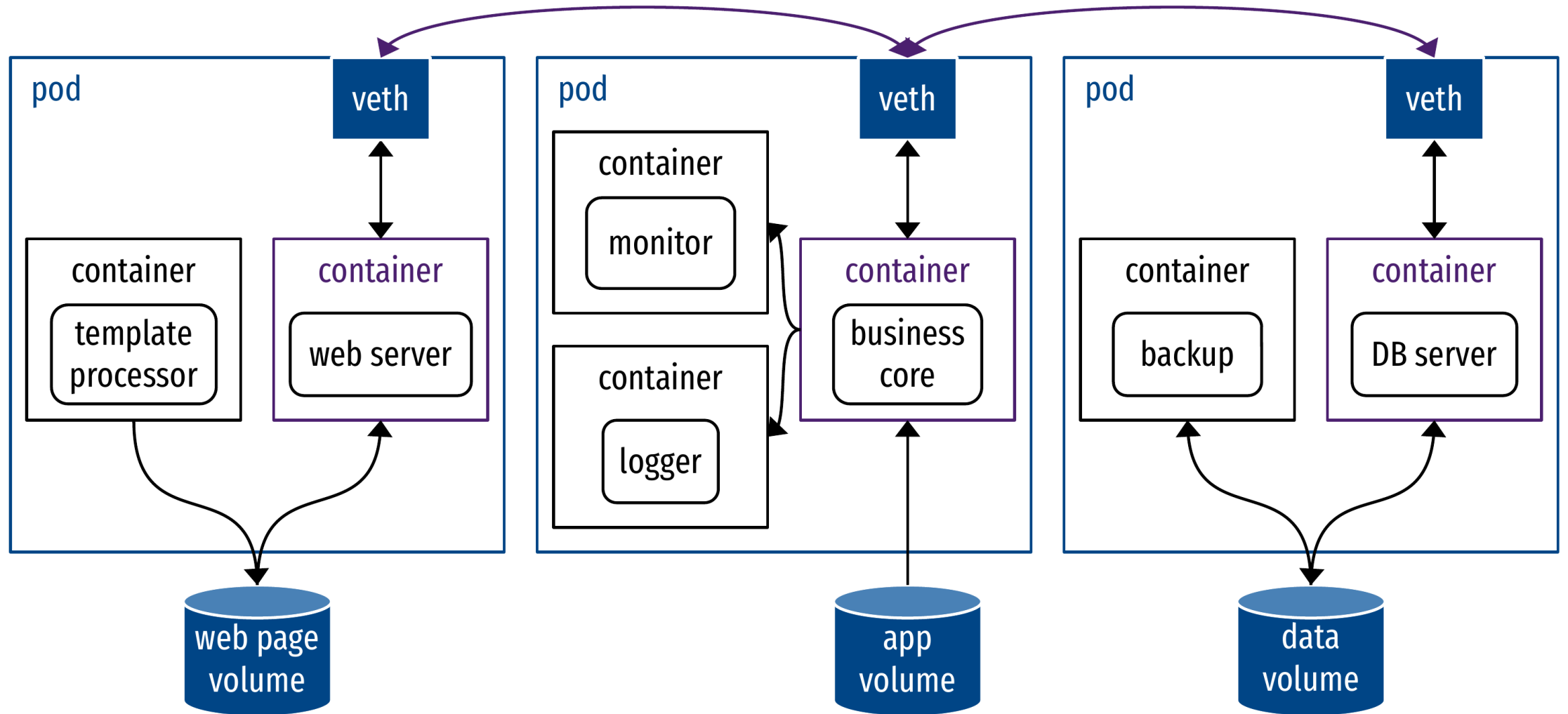
Orchestration

- Composition: build applications as micro-services
 - Roughly: manage multiple containers as one application
 - Example: Docker Compose
- **Orchestration**: manage micro-services
 - Deployment
 - Distribution
 - Replication
 - Load-balancing
 - Availability
 - Rolling updates
 - ...
- Orchestration exposes higher-level interfaces to the features of composition
 - In the end, the orchestrator is the user front-end
- Examples: Kubernetes, Docker Swarm
- Abstraction of management unit: the **pod**

Orchestration: scheduling

- Manual criteria: **filters**
 - Handle host heterogeneity
 - Settings of container runtime, host OS...
 - Container affinity: force placement for resource access
 - Image availability, volume placement, other container...
- **Strategies for deployment** on physical hosts
 - Spread: balance load over hosts
 - Binpack: colocate as much as possible
- Handle colocation of tightly-coupled containers: **pods**
 - Containers in a pod share the same network namespace and same volumes
 - Pod = service container + helper (sidecar) containers (logging, interfacing...)

Orchestration of pods



Application architecture with pods.

Operating system-level virtualization

- Virtualize the OS instead of the hardware
 - **Containers:** simpler, lighter, faster
 - Not safer!
- Based on the Linux kernel (Linux Containers, LXC): namespaces, cgroups, etc.
 - Container engines wrap those features and deliver unified specifications
 - Container runtimes bring usability, networking, development processes...
- Enabling new cloud-native application architecture: **micro-services**
 - Compositions of containers managed by orchestrators