



Training Session CN001

# Containers and container orchestration

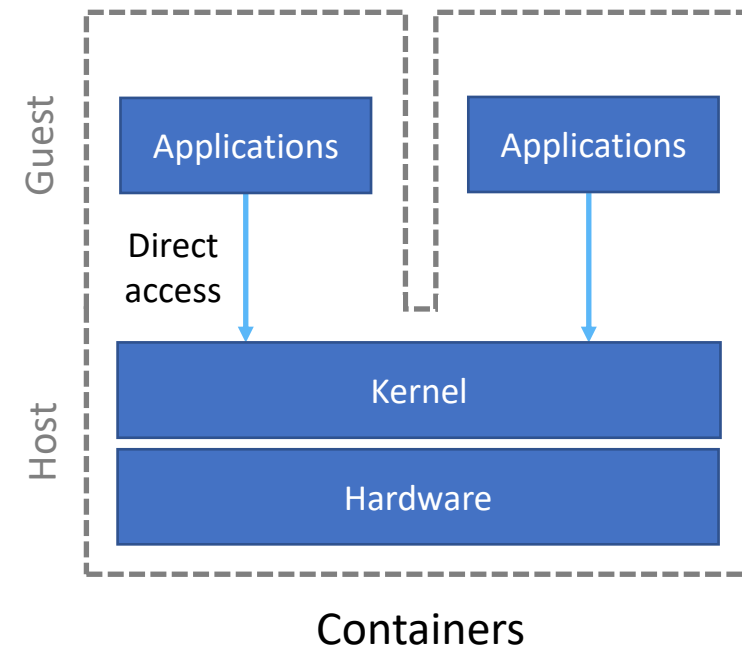
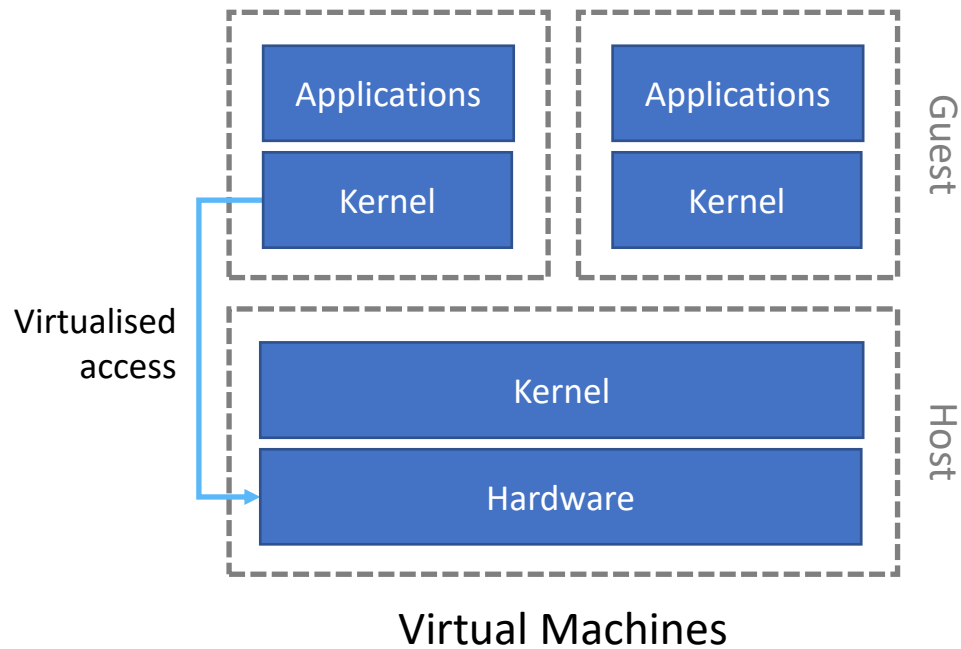
# Outline

- Introduction to containers
- Peeking under the hood: process isolation explained
- Differences between Linux containers and Windows containers
- Virtual Machine-based isolation implementations
- Hardware acceleration in containers
- Overview of container orchestration
- Orchestration with Kubernetes

# Introduction to containers

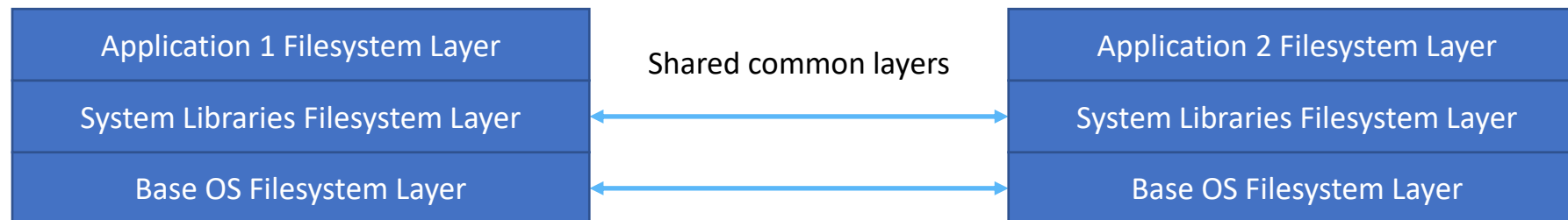
# Introduction to containers

Conceptually, **containers** can be thought of as a lightweight and flexible alternative to **Virtual Machines (VMs)**: *(although they're actually much more than that!)*



# Introduction to containers

- The defining feature of containers is that they **share the kernel** with the host system
- Direct access to the host kernel makes container creation as fast as process creation
- Containers typically utilise **union filesystems** to facilitate composing images from a set of compact, reusable filesystem layers that are combined at runtime



# Introduction to containers

Popular container frontends like [Docker](#) pair union filesystems with a [configuration-as-code](#) approach to deterministically build images from declarative specifications:

```
# Specifies the base image whose filesystem layers we will build upon  
FROM ubuntu:18.04
```

```
# Copies files from the host into the image (creates a new filesystem layer)  
COPY . /app
```

```
# Builds source code inside the image (creates a new filesystem layer)  
RUN make /app
```

# Introduction to containers



Although popular tools such as [Docker](#) and [Kubernetes](#) typically act as de-facto standards for building and running containers, the [Open Container Initiative](#) was founded in 2015 to provide standard specifications that all container implementations should adhere to:

- [OCI Runtime Specification](#) defines how runtimes create and manage containers
- [OCI Image Specification](#) defines a standard format for packaging containers

# Introduction to containers

In addition to the Open Container Initiative, the [Linux Foundation](#) maintains a number of other initiatives with a strong focus on the use of containers:



Oversees the development of key container tools, including [Kubernetes](#) and [containerd](#)



Oversees projects that focus on CI/CD tooling for building and deploying containers



# Introduction to containers

Important events in modern container history:

- **2007:** Google contributes [cgroups](#) to the Linux kernel
- **2008:** [LXC](#) is released by Google and IBM (along with other contributors)
- **2013:** [Docker](#) is released, bringing containers to widespread attention
- **2014:** Google releases [Kubernetes](#), the first major container orchestration system
- **2014:** CoreOS [rkt](#) is released, providing the first major alternative to Docker

# Introduction to containers

Important events in modern container history (**continued**):

- **2015:** The [Open Container Initiative](#) (OCI) is founded
- **2015:** The [Cloud Native Computing Foundation](#) (CNCF) is founded
- **2016:** Microsoft adds container support to Windows Server 2016 and Windows 10
- **2019:** The [Continuous Delivery Foundation](#) (CDF) is founded
- **2019:** Kubernetes 1.14 introduces production-ready support for Windows containers

# Introduction to containers

The following operating systems support containers in one form or another:

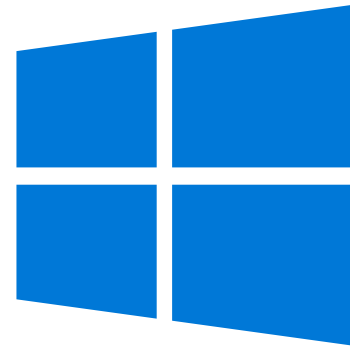
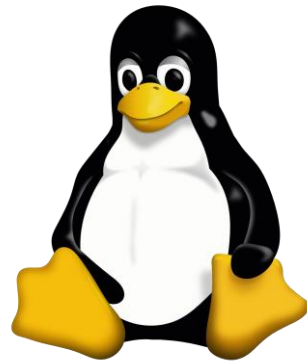
- **FreeBSD**: supported in the form of **FreeBSD jails**, introduced in 2000
- **Solaris**: supported in the form of **Solaris Zones**, introduced in 2005
- **Linux**: supported since the early 2000s, but popularised by Docker in 2013
- **Windows**: supported since 2016, with improved orchestration support since 2018
- **macOS**: **not officially supported**, but that won't stop us: <https://macOScontainers.org>

Post-Section Activity:

# 01 – Building and running Linux containers

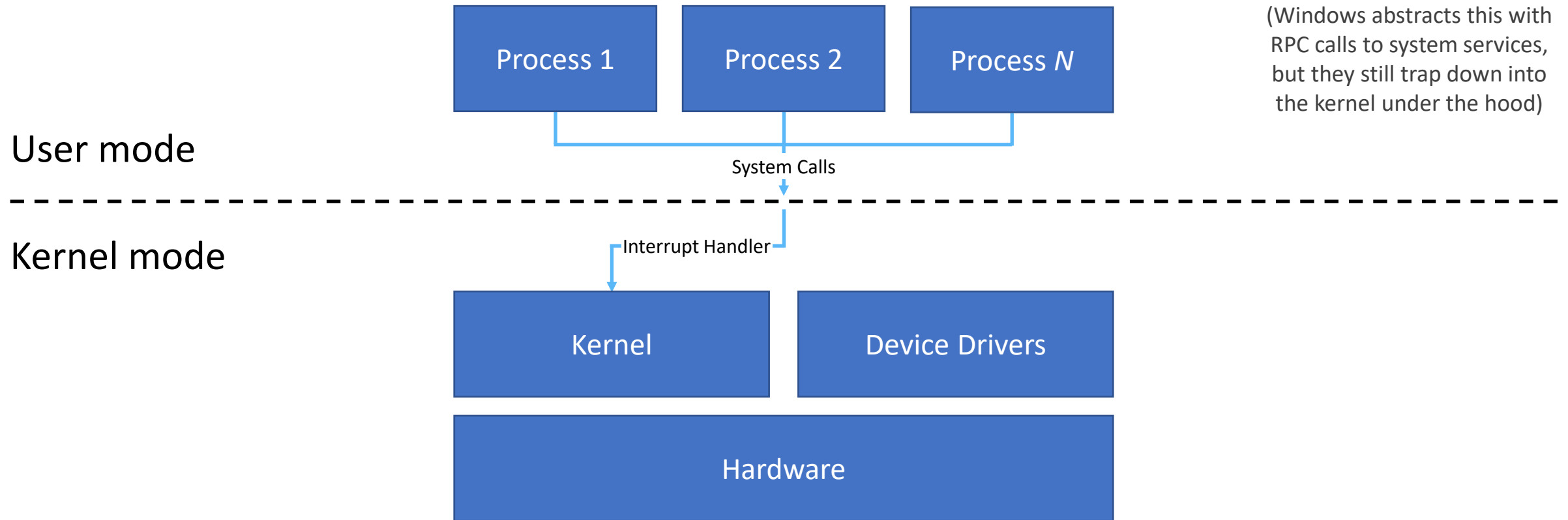
# Peeking under the hood: process isolation explained

# Peeking under the hood: process isolation explained



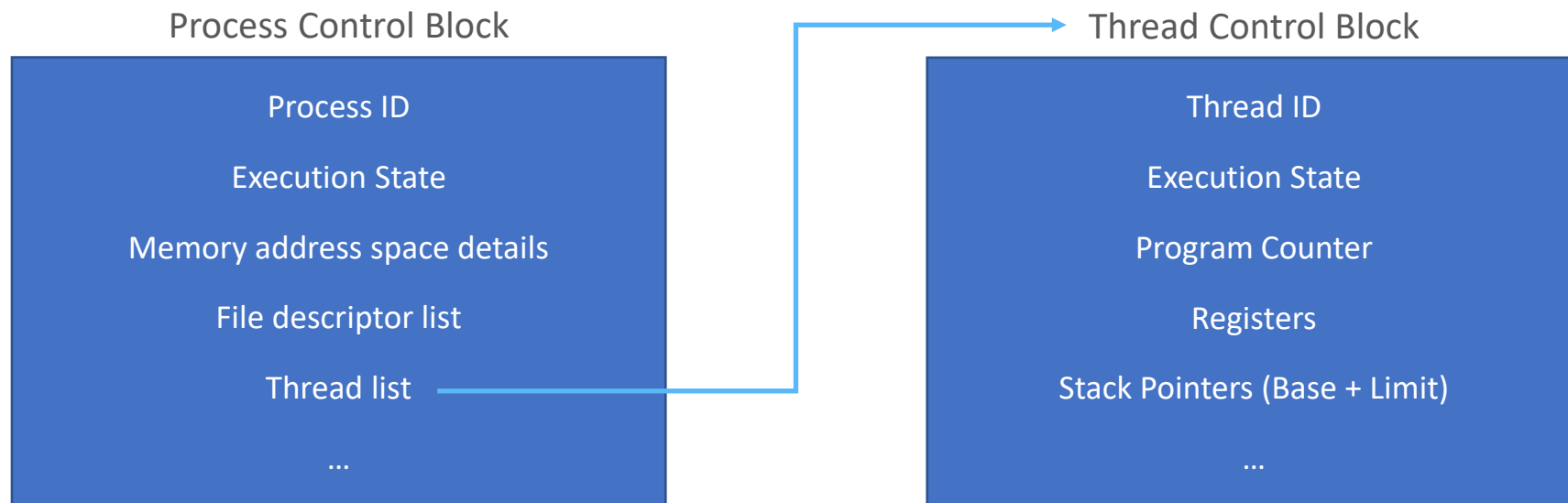
**Note:** implementation details in this section refer to the Linux kernel, but the key concepts are the same under other platforms

# Peeking under the hood: process isolation explained



# Peeking under the hood: process isolation explained

Processes are really just **Process Control Blocks** managed by the kernel:





# Peeking under the hood: process isolation explained

**Every** interaction between processes and their environment is abstracted by the kernel:

- Thread management (including CPU scheduling)
- Memory management (allocation, deallocation, etc.)
- Device access (I/O and filesystems, networking, etc.)
- IPC mechanisms (shared memory, PIDs, signals, etc.)

# Peeking under the hood: process isolation explained

This abstraction makes it easy to control what processes see and do:

- **Namespaces** partition environment visibility into distinct views
- **Control groups** apply resource accounting & access restrictions to process groups

# Peeking under the hood: process isolation explained








The first type of namespace in the Linux kernel was the [Mount Namespace](#):

- This powered an early isolation mechanism known as a [chroot jail](#)
- Set the root filesystem directory for a process & its children using the **chroot** command
- Only the required files are mounted into the new filesystem

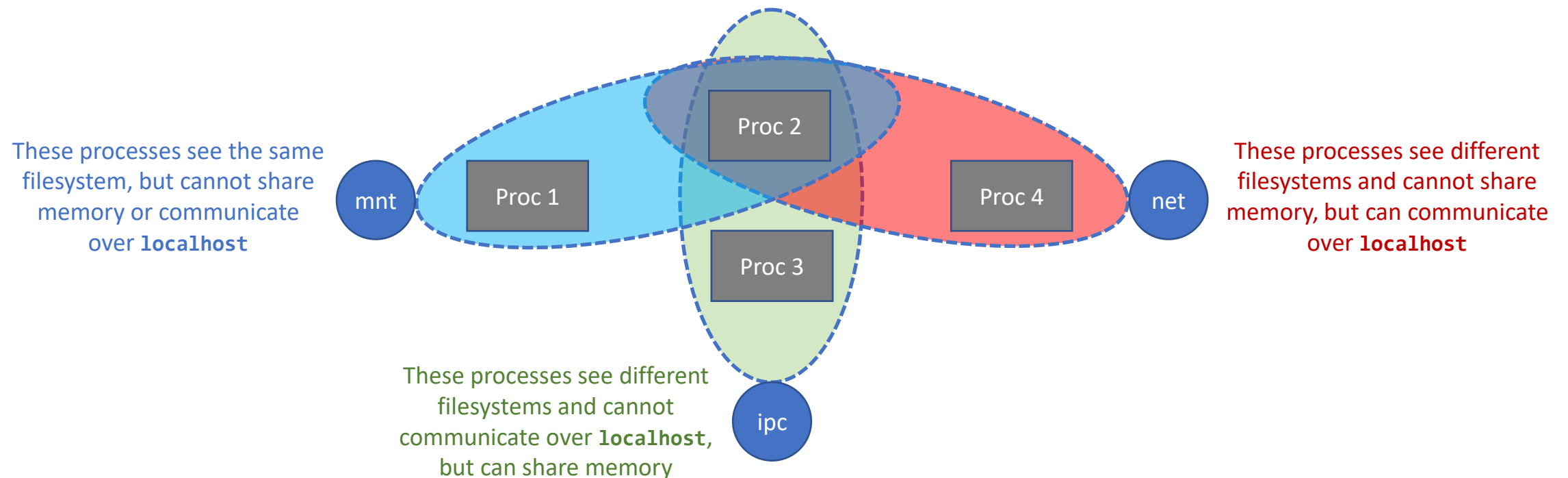
# Peeking under the hood: process isolation explained

Linux kernel developers soon realised the potential and added more namespaces:

-  The **PID Namespace** creates unique sets of PIDs for processes
-  The **Network Namespace** controls the visibility of network devices
-  The **IPC Namespace** controls the visibility of resources such as shared memory
-  The **UTS Namespace** controls the hostname that processes see
-  The **User ID Namespace** creates virtual user IDs that map to real user IDs

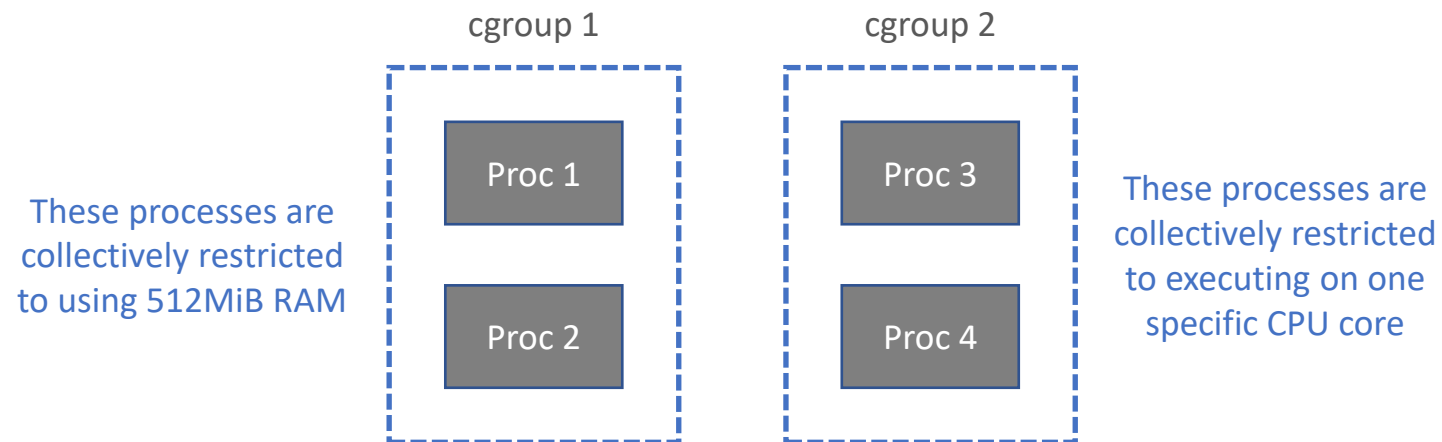
# Peeking under the hood: process isolation explained

Every process belongs to one instance of each namespace, and each namespace instance can contain an arbitrary number of processes:

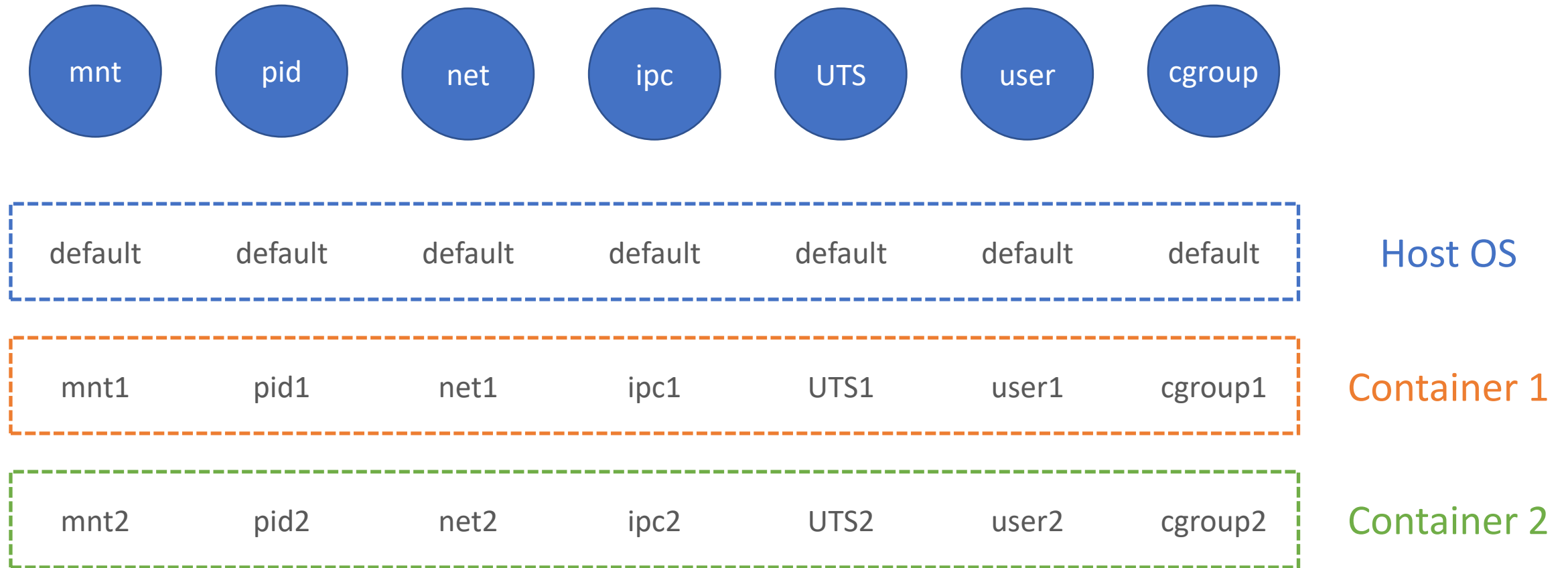


# Peeking under the hood: process isolation explained

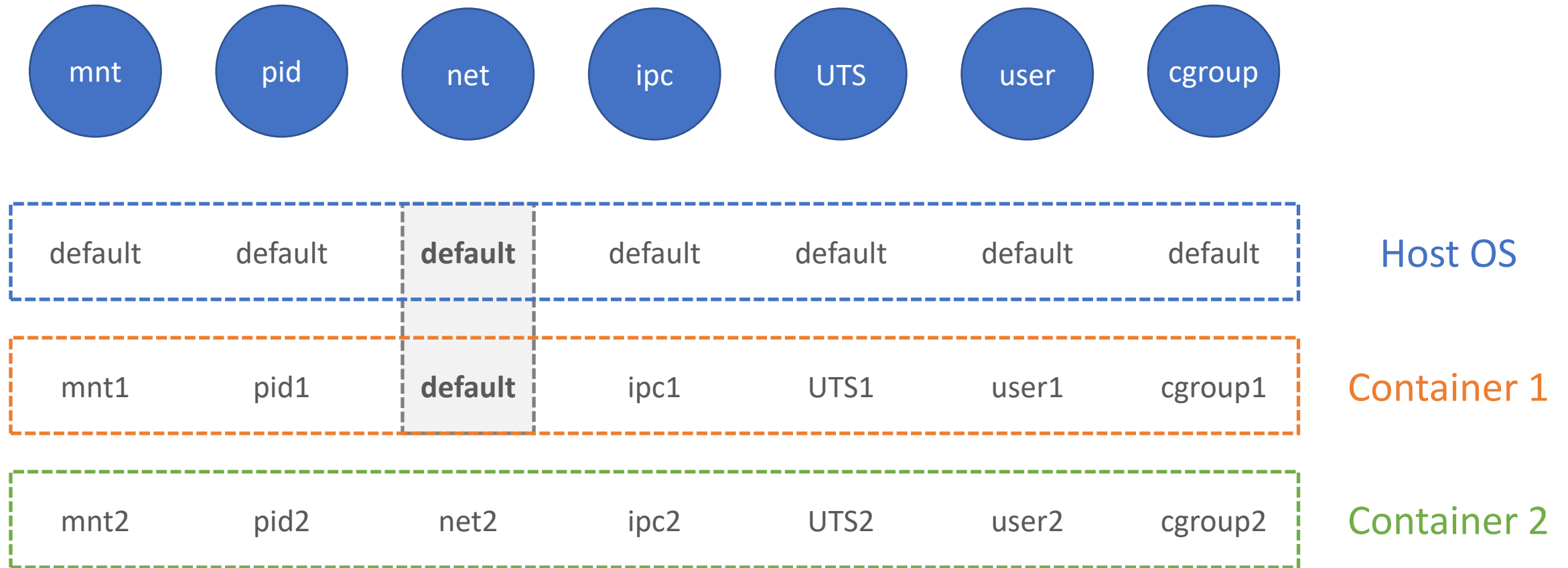
Like namespaces, every process belongs to one cgroup, and each cgroup can contain an arbitrary number of processes:



# Peeking under the hood: process isolation explained

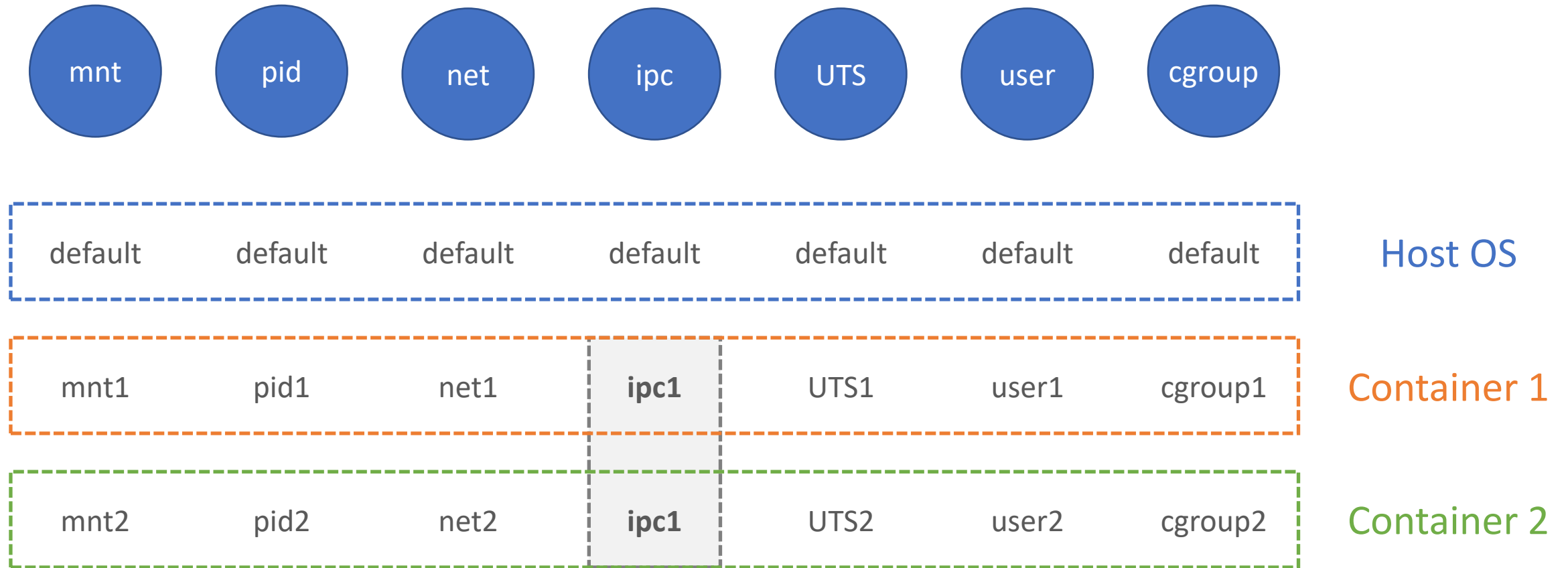


# Peeking under the hood: process isolation explained





# Peeking under the hood: process isolation explained



# Peeking under the hood: process isolation explained

Containers are just an abstraction on top of the isolation features of the kernel:

- Each container commonly has its own **unique** cgroup and set of namespaces
- Containers can **share** a cgroup or any namespace(s) with the host or other containers
- Orchestration systems often group related containers using shared namespaces

Post-Section Activity:

## 02 – Exploring namespace sharing

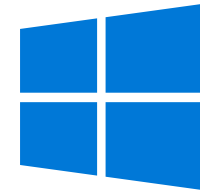
# Differences between Linux containers and Windows containers

# Differences between Linux containers and Windows containers

Container support was added to the Windows kernel in Windows Server 2016:

- Extended the existing Windows **Job Object** functionality that groups processes
- Improved existing resource control mechanisms
- Added chroot support to the system-level **Object Namespace** (\DosDevices\C:, etc.)
- Added a new system service called the **Host Compute Service (HCS)** to abstract all this

# Differences between Linux containers and Windows containers

**Interface:**

Exposed directly by the kernel

Abstracted by the Host Compute Service

**Processes:**

Only the entrypoint process & its children

Entrypoint & children + system services

**Union Filesystem:**

True Union FS for everything

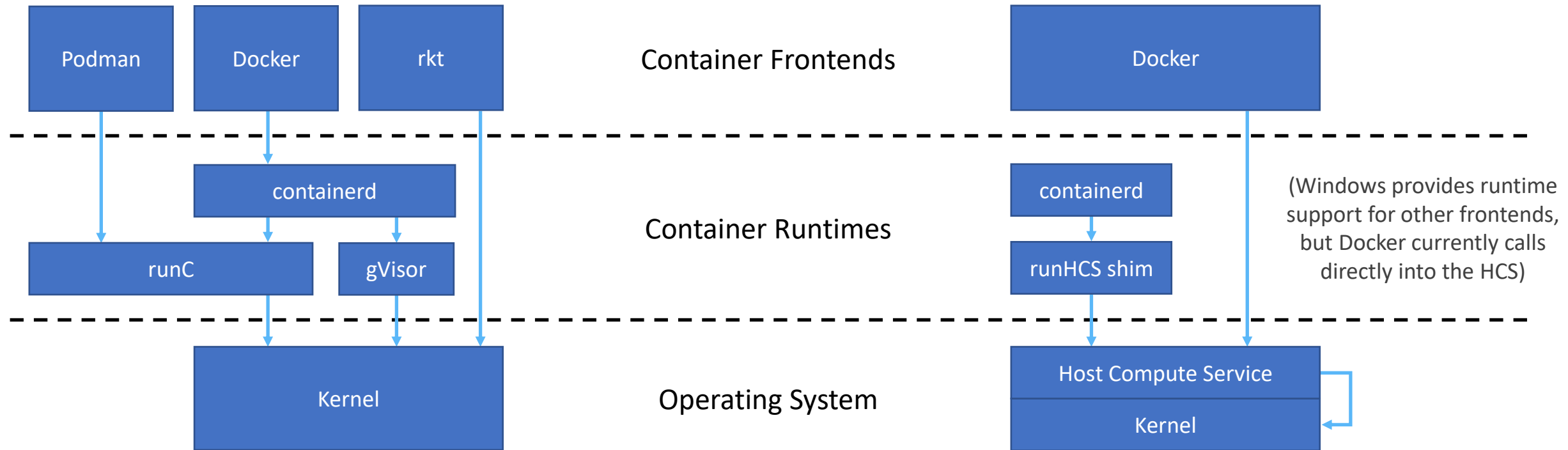
True Union FS for registry, hybrid for NTFS

**Kernel Compatibility:**

Full backwards compatibility

Containers must match compatible host kernel

# Differences between Linux containers and Windows containers



# Differences between Linux containers and Windows containers

Unlike Linux containers, Windows containers only support Microsoft-supplied base images:

- Three base image variants exist:
  - Nano Server
  - Server Core
  - Full Windows (version 1809 and newer only)
- New versions of each base image are released for each Windows kernel version
- Base filesystem layers are marked as “foreign layers” to ensure they’re always pulled directly from Microsoft’s container registry



# Differences between Linux containers and Windows containers

Windows supports three types of containers:

HostProcess

*a.k.a.*

“Not actually a  
container”

Process-isolated

*a.k.a.*

“A traditional  
container”

Hyper V-isolated

*a.k.a.*

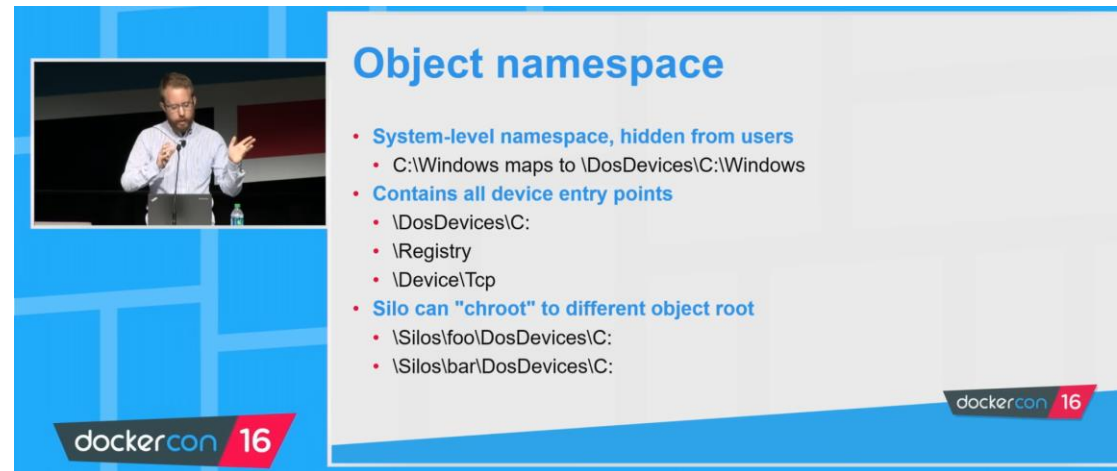
“A container  
wrapped in a VM”

*(Discussed in the next section)*

# Differences between Linux containers and Windows containers

Jon Starks' DockerCon 2016 presentation provides all the low-level details:

<https://youtu.be/85nCF5S8Qok>



The screenshot shows a presentation slide from DockerCon 2016. On the left, there is a small video inset of Jon Starks speaking. The main slide has a blue header with the title 'Object namespace'. Below the title, there are three bullet points: 'System-level namespace, hidden from users' (with a sub-bullet 'C:\Windows maps to \DosDevices\C:\Windows'), 'Contains all device entry points' (with sub-bullets '\DosDevices\C:', '\Registry', and '\Device\Tcp'), and 'Silo can "chroot" to different object root' (with sub-bullets '\Silos\foo\DosDevices\C:' and '\Silos\bar\DosDevices\C:'). The slide has a blue footer with the 'dockercon 16' logo.

**Object namespace**

- System-level namespace, hidden from users
  - C:\Windows maps to \DosDevices\C:\Windows
- Contains all device entry points
  - \DosDevices\C:
  - \Registry
  - \Device\Tcp
- Silo can "chroot" to different object root
  - \Silos\foo\DosDevices\C:
  - \Silos\bar\DosDevices\C:

Post-Section Activity:

## 03 – Building and running Windows containers

# Virtual Machine-based isolation implementations

# Virtual Machine-based isolation implementations

Process isolation is frequently associated with security holes that risk privilege escalation attacks known as [container breakouts](#):

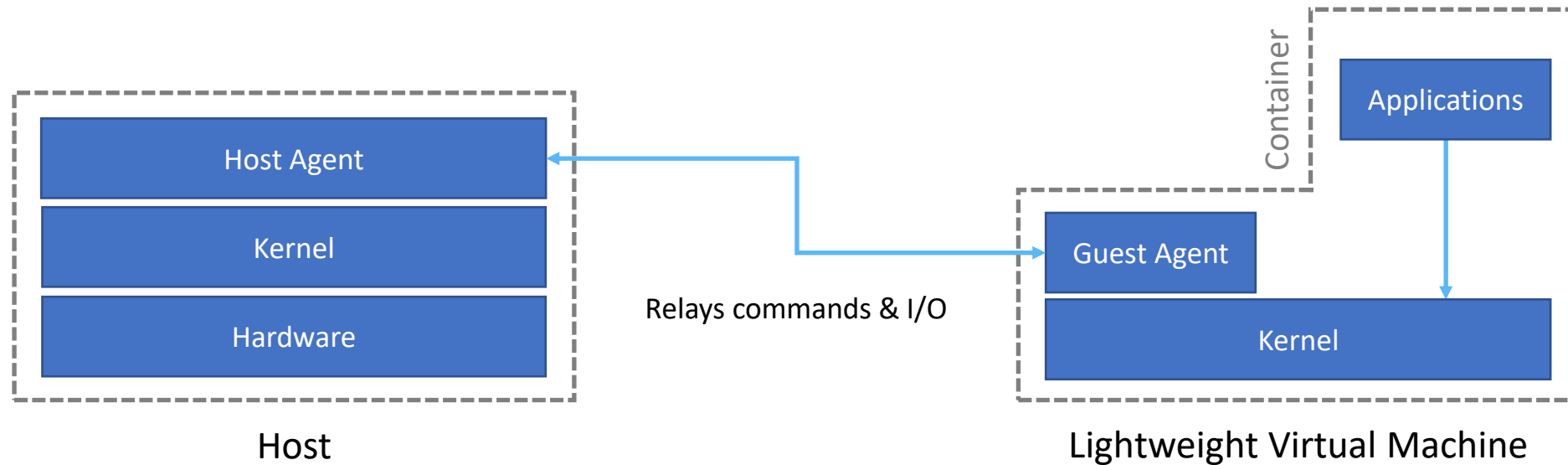
- **December 2014:** [CVE-2014-9357](#) demonstrates the first critical security bug in [Docker](#)
- **February 2019:** [CVE-2019-5736](#) demonstrates a critical security bug in [runC](#)
- **May 2019:** three major security flaws found in the now-unmaintained [rkt](#) runtime
- **July 2020:** first breakout attack documented for process-isolated Windows containers
- **September 2020:** [CVE-2020-14386](#) demonstrates a security bug in the Linux kernel that affects all process isolation-based Linux container runtimes except for [gVisor](#)

# Virtual Machine-based isolation implementations

VM-based isolation implementations address security concerns by wrapping host kernels in purpose-built **lightweight virtual machines**:

- Contain only the kernel and an agent to communicate with the host container runtime
- Optimised for faster startup and lower memory overheads than standard VMs
- Typically include platform-specific tweaks to help mitigate performance reductions
- Can run multiple containers within a single VM using process isolation

# Virtual Machine-based isolation implementations



# Virtual Machine-based isolation implementations

VM isolation provides **improved security** but introduces limitations not present when using process isolation:

- Inability to easily share hardware devices with the host system
- Reliant on nested virtualisation to function when the host is itself a VM
- Communication overheads between the host container runtime and VM guest agent
- Ability to share namespaces between related containers varies by implementation



# Virtual Machine-based isolation implementations



Windows containers support both [process isolation mode](#) and [Hyper-V isolation mode](#):

- Runs a Windows Server kernel in a lightweight “utility VM”
- Facilitates running older kernel versions under newer versions of Windows
- The only supported isolation mode under Windows 10 prior to version 1809

# Virtual Machine-based isolation implementations



**Kata Containers** is an open source project that provides VM isolation for Linux containers:

- Drop-in replacement for the standard **runC** runtime
- Uses lightweight virtual machines similar to **Hyper-V** utility VMs
- Only supports sharing namespaces between related containers when using Kubernetes

Post-Section Activity:

## 04 – Running Hyper-V isolated Windows containers

# Hardware acceleration in containers

# Hardware acceleration in containers

- Access to devices such as GPUs is useful for containerising specialised workloads
- **Shared hardware access** **only available when using process isolation**
- **Exclusive hardware access** theoretically possible when using VM isolation, but **not currently implemented** for Linux or Windows containers
- Device support varies by platform and container runtime

# Hardware acceleration in containers



Linux containers can access NVIDIA GPUs using the [NVIDIA Container Toolkit](#):

- Adds a prestart hook that works with the standard [runC](#) runtime
- Supports CUDA, OpenCL, OpenGL and Vulkan
- Base images with the required runtime libraries provided by NVIDIA on [Docker Hub](#)

# Hardware acceleration in containers



Linux containers can access AMD GPUs using the [Radeon Open Compute \(ROCm\)](#) platform:

- Kernel modules on the host communicate with runtime libraries inside containers
- Supports APIs that ROCm can run (HIP, OpenCL, CUDA, etc.)
- Base images with the required runtime libraries provided by AMD on [Docker Hub](#)

# Hardware acceleration in containers



Windows containers can access any GPU with a WDDM 2.5 or newer driver:

- Requires Docker 19.03 and Windows Server 2019 / Windows 10 version 1809 or newer
- Only works with the full-fat Windows base image (1809 or newer)
- Only supports DirectX and APIs built atop it (e.g. DirectML), **no other APIs**



## Post-Section Activity:

# 05 – Running GPU-accelerated Linux containers with the NVIDIA Container Toolkit

# Overview of container orchestration

# Overview of container orchestration

Tools like [Docker](#) provide a way of building and running containers, but we need [container orchestration frameworks](#) to effectively deploy containers at scale:

- Manage scaling and scheduling of containers across a dynamic cluster of host nodes
- Manage network configuration and [service discovery](#) (both internal and external)
- Monitor container and host node health, automatically replacing crashed instances
- Manage security and provide storage and deployment of runtime secrets

# Overview of container orchestration

## Concern #1: Scheduling and scaling

- The underlying cluster may change its size and composition dynamically
- The number of **replicas** for a service's containers may scale based on demand
- Containers need to be scheduled (and often moved) in an efficient manner
- **Bin packing** allows us to maximise deployment density and minimise wasted resources

# Overview of container orchestration

## Concern #2: Network configuration and service discovery

- **Load balancing** routes requests to an ever-changing set of ephemeral containers
- In a **microservices architecture**, many internal services need to communicate
- An additional infrastructure layer known as a **service mesh** often provides advanced networking functionality on top of the container orchestration framework itself

# Overview of container orchestration

## Concern #3: Health monitoring and self-healing

- If a container or host node crashes, it needs to be replaced automatically
- **Liveness probes** facilitate proactive culling & replacement of unresponsive instances
- Event/status metrics should be collected for improved visibility and traceability of issues
- Health checks can be used to control service migration during **canary deployments**

# Overview of container orchestration

## Concern #4: Security and secrets management

- **Security policies** should be supported to restrict the access that containers have to both internal (on-cluster) and external resources
- Sensitive information (passwords, SSH keys, API keys, etc.) should **never** be stored in plaintext in container images or configuration files
- Secrets should be managed by the orchestration framework, encrypted at rest and securely injected into containers at runtime via files or environment variables

# Overview of container orchestration

There are a number of container orchestration frameworks available, including:

- [Kubernetes](#), created by Google
- [Docker Swarm](#), created by Docker, Inc.
- [Mesosphere Marathon](#), created by the Apache Software Foundation

Despite providing competing orchestration implementations, [Docker Enterprise Edition](#) and [Apache Mesosphere](#) both **support Kubernetes** in addition to their own offerings.



# Orchestration with Kubernetes

# Orchestration with Kubernetes



**Kubernetes** is a container orchestration framework created by Google and subsequently adopted as a project of the [Cloud Native Computing Foundation](#) (CNCF):

- Kubernetes is a modern successor to Google's internal [Borg](#) cluster manager, which has been running containers in production since 2003
- Google contributed Kubernetes to the CNCF as its first project in 2015

# Orchestration with Kubernetes

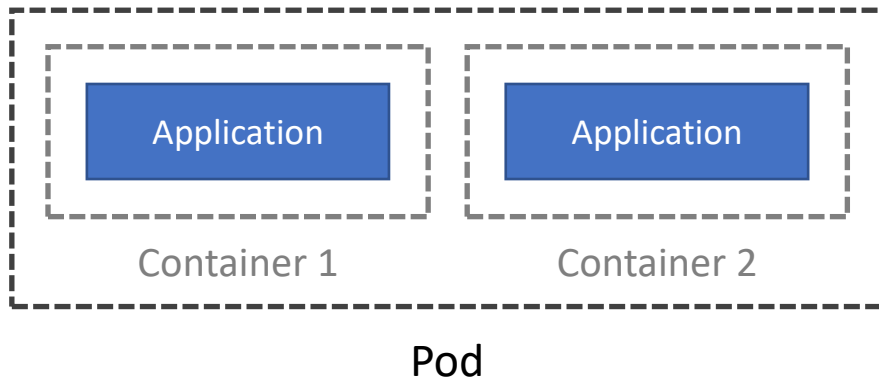
Kubernetes has become the de facto standard for container orchestration and **managed Kubernetes services** are now offered by all major cloud providers:

- Google Kubernetes Engine (GKE)
- Amazon Elastic Kubernetes Service (EKS)
- Azure Kubernetes Service (AKS)
- Red Hat OpenShift Online
- IBM Cloud Kubernetes Service

Plus dozens of others...

# Orchestration with Kubernetes

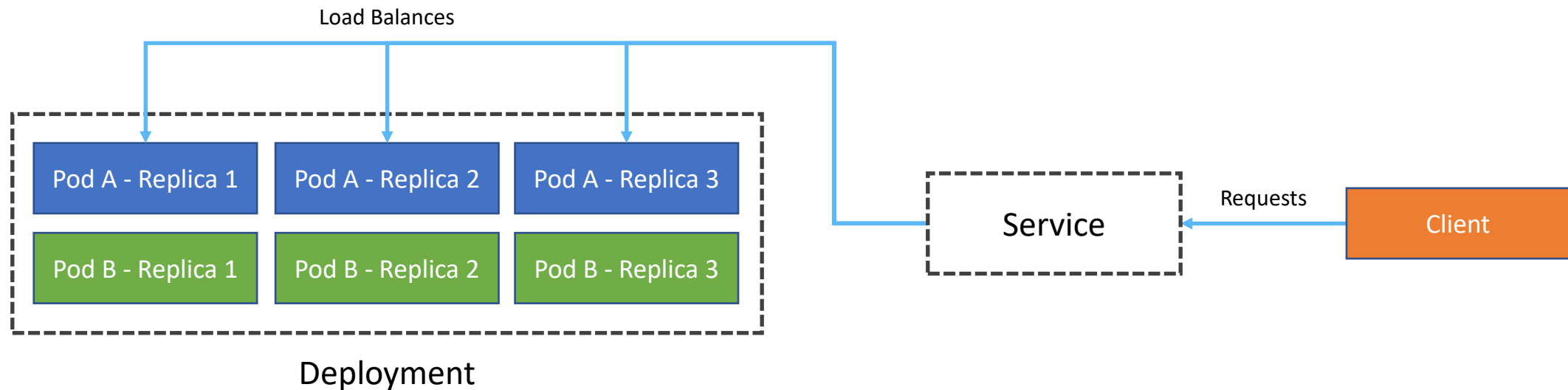
The primitive unit of computation in Kubernetes is a **Pod**, which consists of a set of one or more tightly-coupled containers that share the **net**, **ipc** and **UTS** namespaces:



The processes in the containers see different filesystems / user IDs / PIDs / etc., but can share memory and communicate over **localhost**, and can also bind-mount common shared volumes

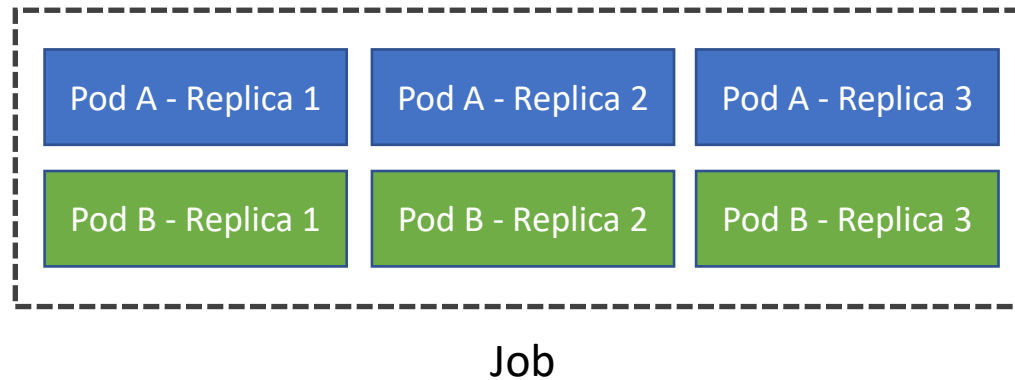
# Orchestration with Kubernetes

Pods can be deployed and exposed using **Deployments** and **Services** for always-available applications that service requests:



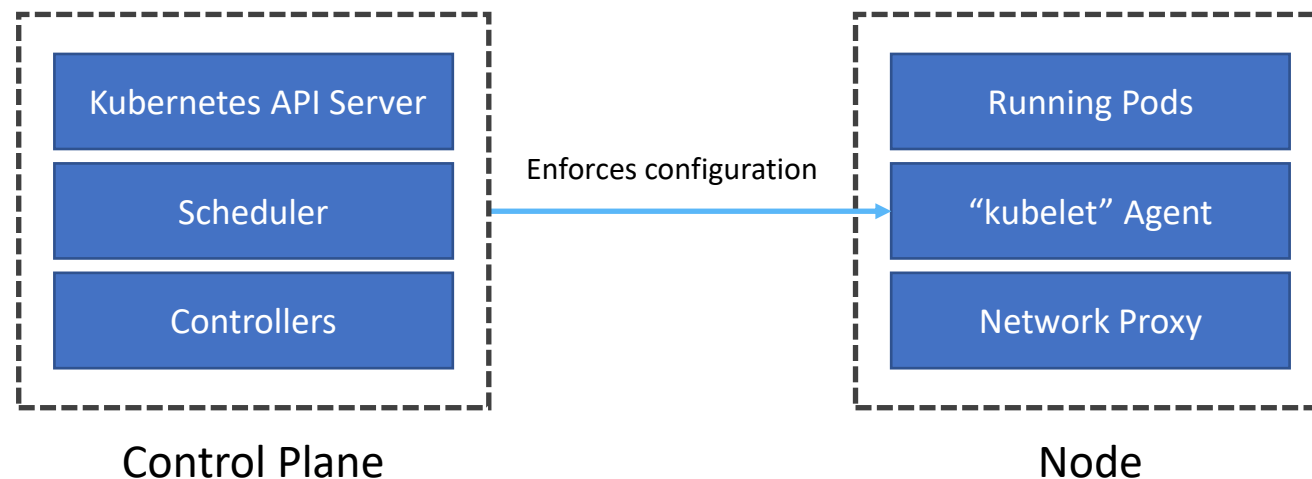
# Orchestration with Kubernetes

Pods can be also deployed using **Jobs** for batch processing tasks that run to completion and stop, which can use replicas for batching data if desired:



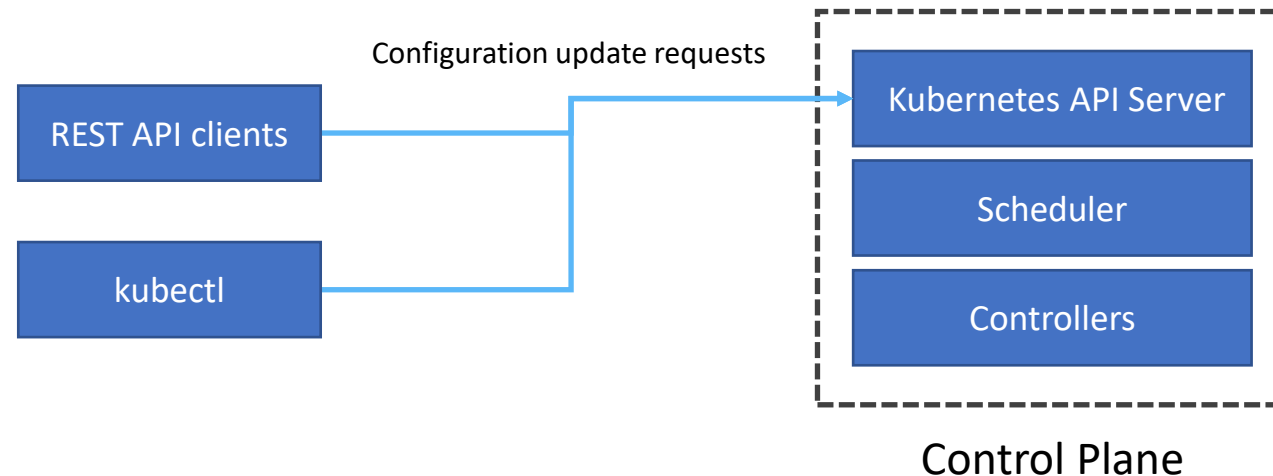
# Orchestration with Kubernetes

Kubernetes architecture is split into the **control plane** and **cluster nodes**. The control plane contains the Kubernetes **master components**, which control the cluster nodes:



# Orchestration with Kubernetes

The [Kubernetes API Server](#) services requests to modify cluster configuration via the Kubernetes REST API or frontends such as the [kubectl](#) command-line tool:





# Orchestration with Kubernetes

Desired configuration states are specified declaratively using the [JSON](#) or [YAML](#) markup languages (using **YAML is recommended** as a best practice):

```
apiVersion: apps/v1
kind: Deployment      # Creates a Deployment (for running always-available services)
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3          # Creates 3 replicas of our nginx pod
  selector:
    matchLabels:
      app: nginx        # Label matching is how the Deployment recognises the Pods it owns
  template:
    metadata:
      labels:           # These labels are used to identify the Pods that get created
        app: nginx
    spec:
      containers:      # The pod consists of a single container running nginx
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80  # nginx will be servicing HTTP requests on TCP port 80
```

Post-Section Activity:

## 06 – Deploying a simple Kubernetes service

# Related Resources

## Introduction to containers:

- Red Hat knowledgebase article discussing the mechanics and history of Linux containers:  
<https://www.redhat.com/en/topics/containers/whats-a-linux-container>
- Red Hat knowledgebase article discussing how containers fit into the context of cloud native apps:  
<https://www.redhat.com/en/topics/cloud-native-apps>
- Official definition of cloud native computing from the CNCF (which lists containers as a key component):  
<https://github.com/cncf/toc/blob/master/DEFINITION.md>
- Overview of Docker storage drivers, which discusses union filesystems:  
<https://docs.docker.com/storage/storagedriver/>

# Related Resources

## Process isolation:

- Linux manpage for namespaces:  
<http://man7.org/linux/man-pages/man7/namespaces.7.html>
- *Linux Journal* article discussing cgroups:  
<https://www.linuxjournal.com/content/everything-you-need-know-about-linux-containers-part-i-linux-control-groups-and-process>

# Related Resources

## Windows containers:

- Jon Starks' DockerCon presentation with all the low-level details about how Windows containers work:  
<https://youtu.be/85nCF5S8Qok>
- Microsoft documentation for Windows containers:  
<https://docs.microsoft.com/en-us/virtualization/windowscontainers/>

# Related Resources

## VM isolation:

- Documentation for process-isolated Windows container breakout vulnerability:  
<https://unit42.paloaltonetworks.com/windows-server-containers-vulnerabilities/>
- Microsoft documentation for Hyper-V isolation mode:  
<https://docs.microsoft.com/en-us/virtualization/windowscontainers/manage-containers/hyperv-container>
- Microsoft documentation for LCOW:  
<https://docs.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/linux-containers>
- Kata Containers architecture documentation:  
<https://github.com/kata-containers/documentation/blob/master/design/architecture.md>

# Related Resources

## Hardware acceleration in containers:

- NVIDIA blog post introducing NVIDIA Docker and discussing how it works:  
<https://devblogs.nvidia.com/nvidia-docker-gpu-server-application-deployment-made-easy/>
- AMD blog post discussing using ROCm with Docker containers:  
<https://community.amd.com/community/radeon-instinct-accelerators/blog/2018/11/13/the-amd-deep-learning-stack-using-docker>
- Microsoft documentation for hardware device support in Windows containers:  
<https://docs.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/hardware-devices-in-containers>
- Microsoft documentation for GPU acceleration in Windows containers:  
<https://docs.microsoft.com/en-us/virtualization/windowscontainers/deploy-containers/gpu-acceleration>

# Related Resources

## Container orchestration:

- Google publication describing the design of their internal scheduler, Borg (the predecessor of Kubernetes) with an empirical evaluation of its performance:  
<https://ai.google/research/pubs/pub43438>
- *Wired* article (from shortly before Kubernetes was released) discussing how Apache Mesos aimed to replicate Google's "secret weapon" (Borg):  
<https://www.wired.com/2013/03/google-borg-twitter-mesos/>
- Red Hat knowledgebase article discussing service meshes:  
<https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh>



# Related Resources

## Kubernetes:

- Google blog post discussing the origins of Kubernetes and how it evolved from Borg:  
<https://cloud.google.com/blog/products/gcp/from-google-to-the-world-the-kubernetes-origin-story>
- Official Kubernetes documentation, providing comprehensive details on every aspect of the framework:  
<https://kubernetes.io/docs/>



Prepared for TensorWorks by Dr Adam Rehn. Copyright © 2019 - 2021, TensorWorks Pty Ltd.

Licensed under a Creative Commons Attribution 4.0 International License.

All logos are the copyright, trademark or registered trademark of their respective owners.