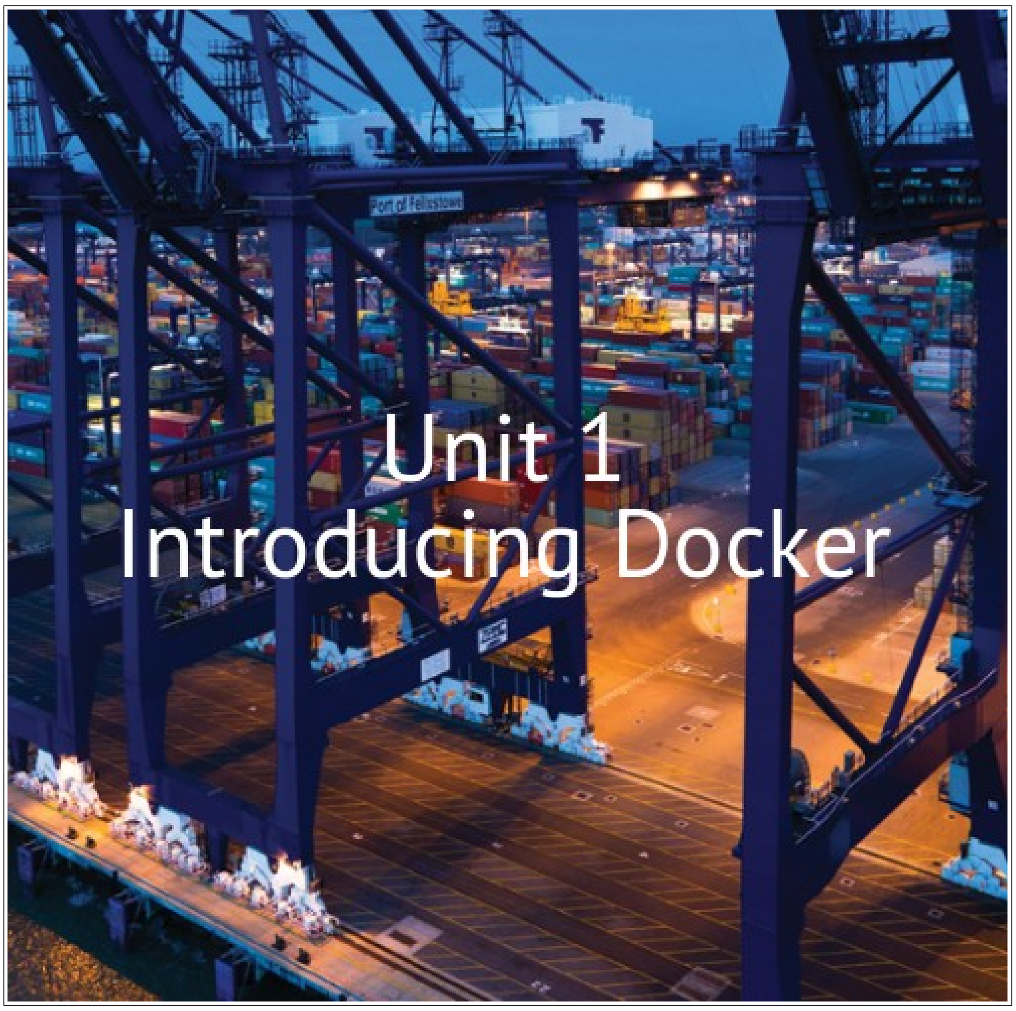


Unit 1
Introducing Docker

Objectives

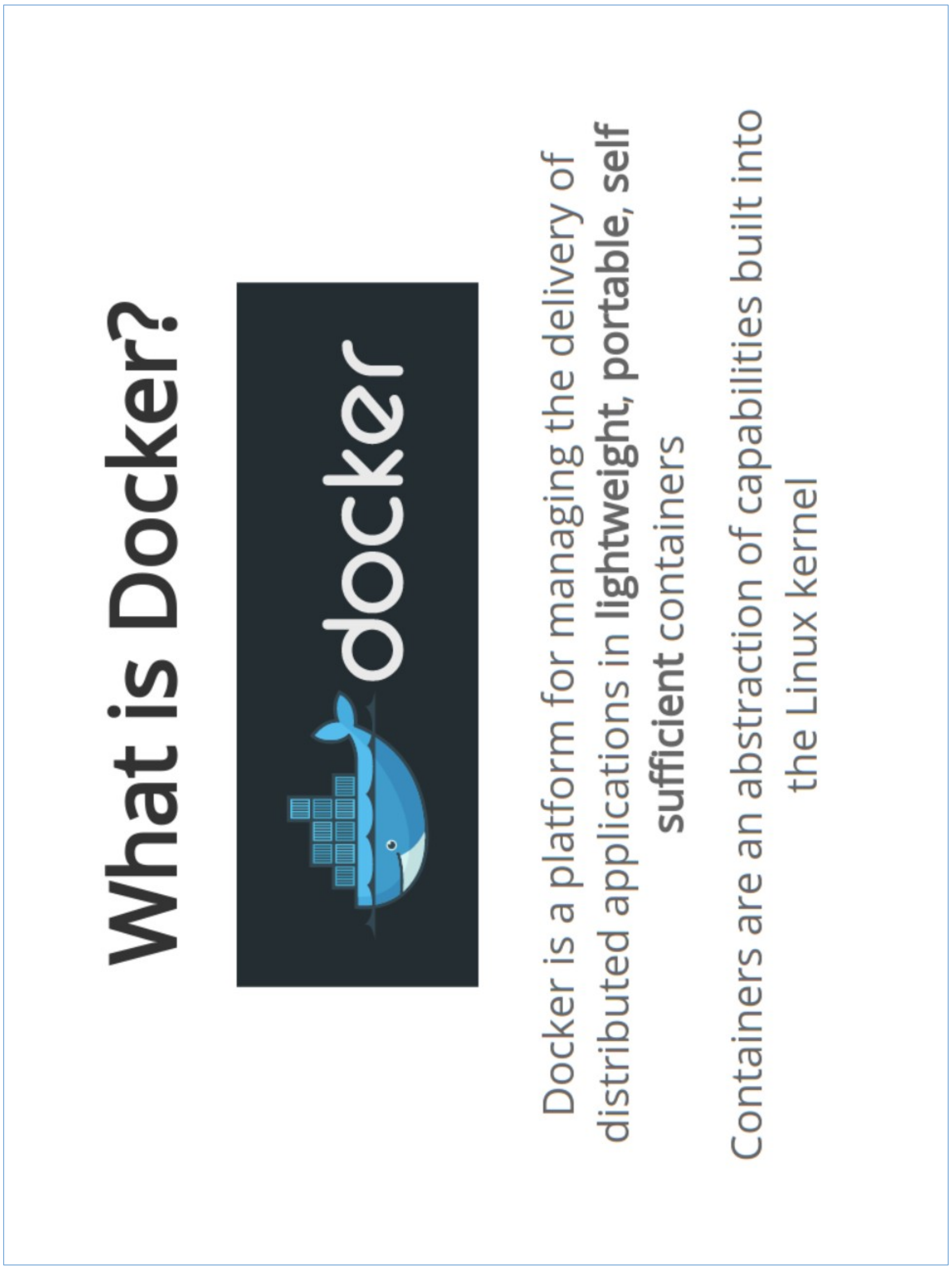
Upon completion of this unit, you will be able to:

- Describe what Docker is, its purpose, and who is responsible for it
- Differentiate between the nature and purpose of containers and virtual machines
- Pinpoint the benefits of, and use cases for, Docker
- Outline a brief history of containing processes
- Provide some detail regarding the Linux kernel capabilities that Docker utilises

Agenda

The following topics will be covered in this unit:

- Overview
- Software Delivery Idiosyncrasies
- Containers vs. Virtual Machines
- Benefits & Use Cases
- History of Container Technologies
- Linux Kernel Capabilities
- Lab exercise: Containers from First Principles



What is Docker?

Docker is a platform for managing the delivery of distributed applications in **lightweight**, **portable**, **self sufficient** containers

Containers are an abstraction of capabilities built into the Linux kernel

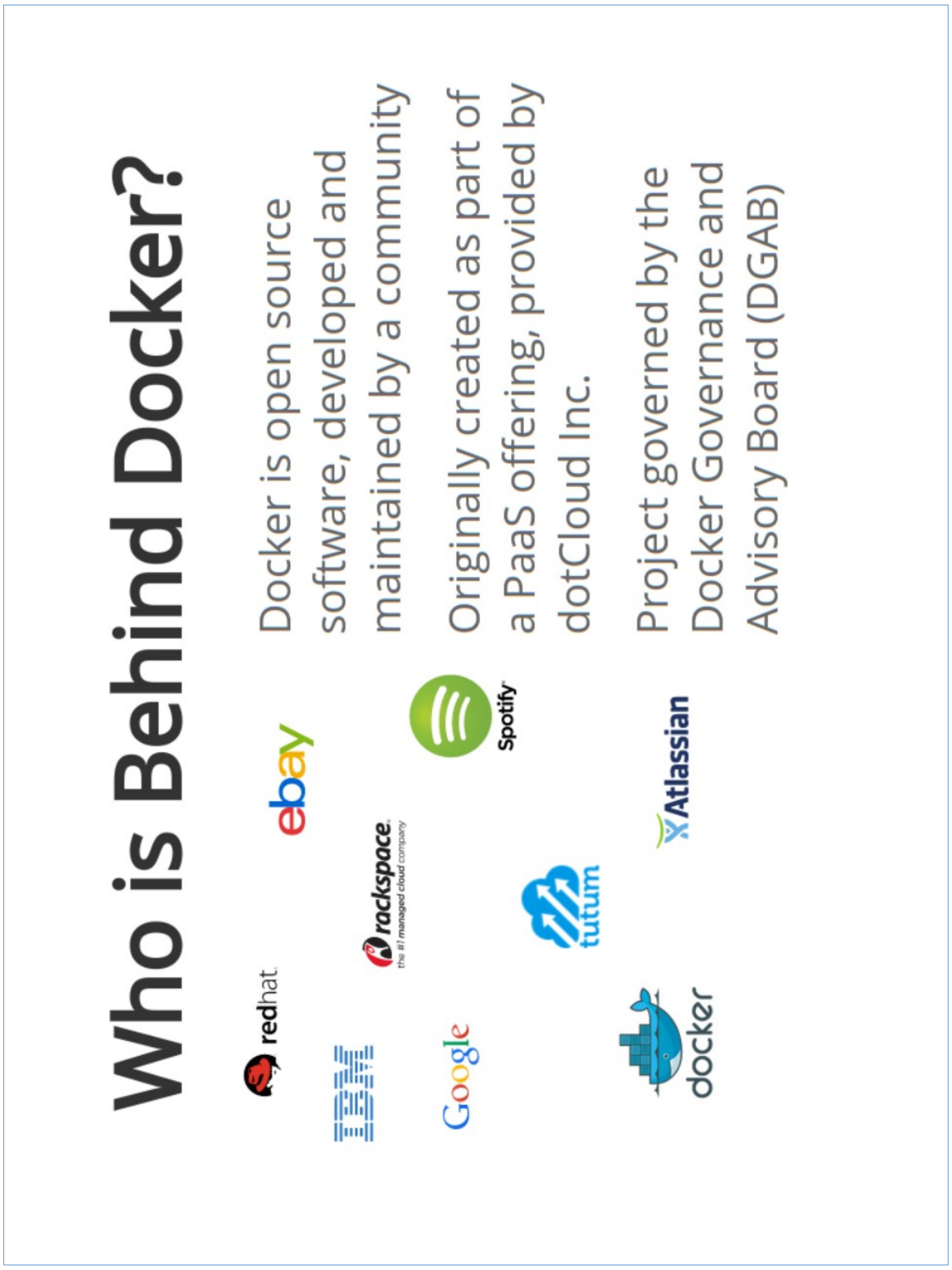
What is Docker?

- Docker is a platform for managing the delivery of distributed applications in lightweight, portable, self sufficient containers

  Platform - Docker is a means to an end rather than an end in itself, so once installed, the real work begins by way of packaging software applications using the platform
  Delivery - in this case, the term 'delivery' encompasses the development as well as the subsequent deployment of software applications
  Containers - processes are isolated within their own environment, complete with a sub-set of the host system's compute resources
  Lightweight - containers are lightweight, in that they consume relatively small amounts of computing resources
  Portable - containers are metaphorically portable, in that they can be moved from one host to another with little regard for the target environment
  Self-sufficient - an application running inside a container has all of the things its dependent on, co-located inside the container

- Containers are not exclusive to Docker, and container runtime environments are provided by numerous providers
- Containers are an abstraction of some capabilities built into the Linux kernel, and the abstraction is a form of virtualization that is implemented at the operating system layer

Notes



Who is Behind Docker?

Docker is open source software, developed and maintained by a community

Originally created as part of a PaaS offering, provided by dotCloud Inc.

Project governed by the Docker Governance and Advisory Board (DGAB)

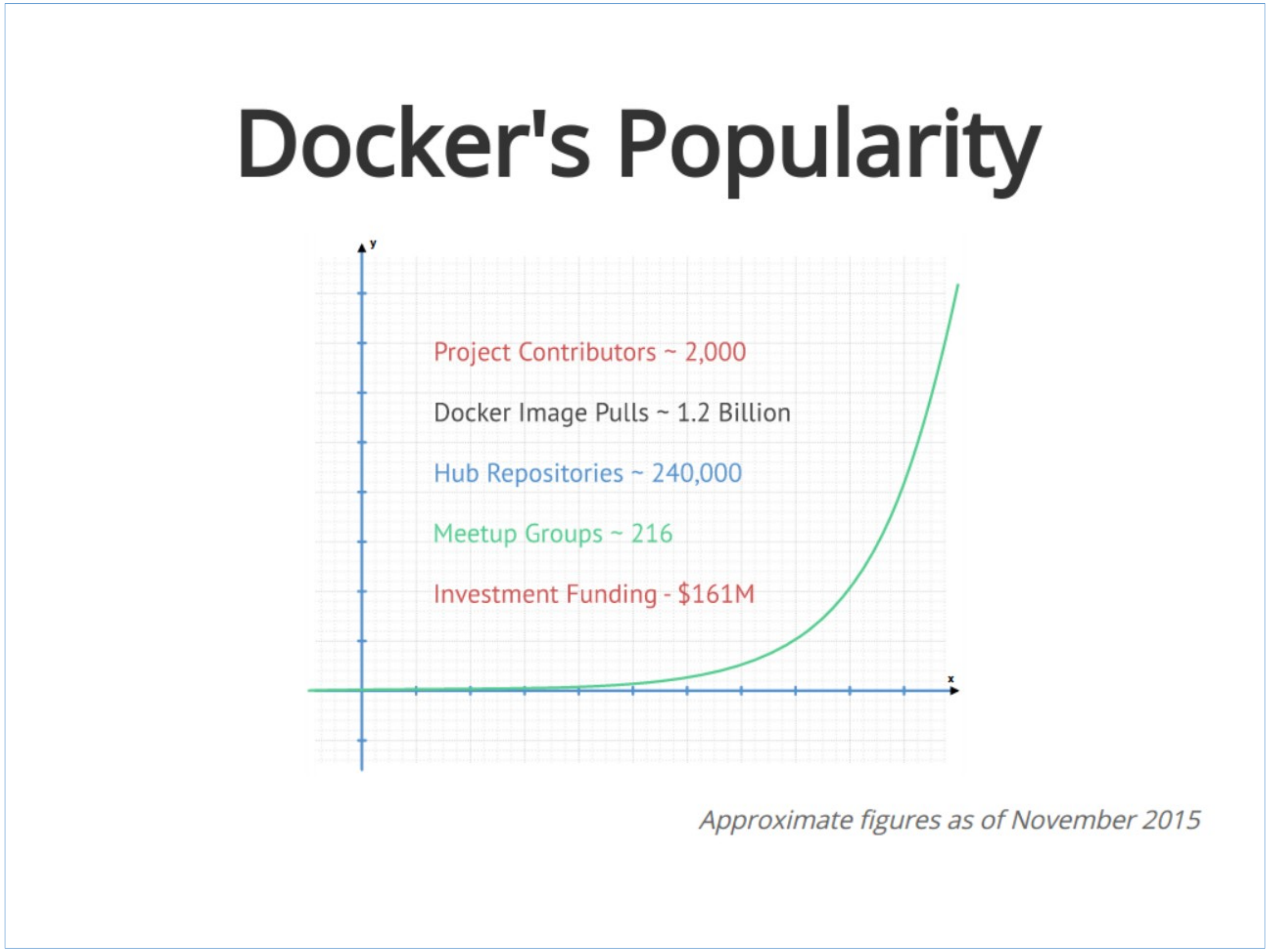
Who is Behind Docker?

- Docker is open source software, developed and maintained by a community of software developers
- Docker was originally created for the internal use of a Platform-as-a-Service (PaaS) provider called dotCloud Inc.
- Recognising the significance of what had been created, dotCloud changed their business focus exclusively to Docker, changed their name to Docker Inc., and in March 2013 made the software open source under an Apache 2.0 License
- The first stable, commercially supported version of Docker (version 1.0), was released in June 2014, and point releases arrive approximately every 2 months

Version	Release Date
1.0	09 June 2014
1.1	03 July 2014
1.2	21 August 2014
1.3	16 October 2014
1.4	12 December 2014
1.5	10 February 2015
1.6	16 April 2015
1.7	18 June 2015
1.8	11 August 2015
1.9	03 November 2015

- The Docker project is governed by the Docker Governance and Advisory Board, which comprises of representatives from Docker Inc., the corporate user community, corporate partners, and individuals who make a significant contribution to the Docker project

Notes



Docker's Popularity

Project Contributors ~ 2,000

Docker Image Pulls ~ 1.2 Billion

Hub Repositories ~ 240,000

Meetup Groups ~ 216

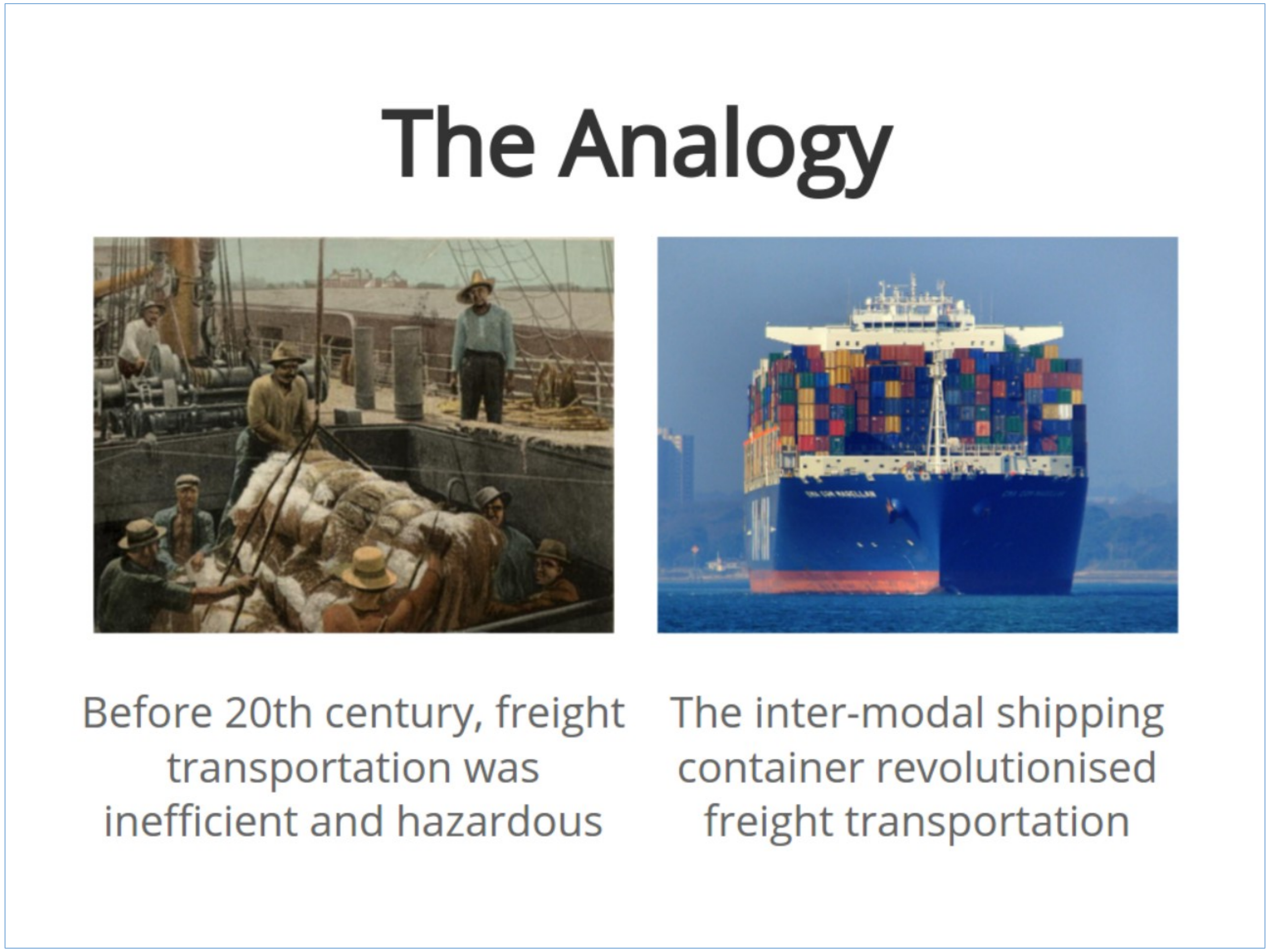
Investment Funding - $161M

Approximate figures as of November 2015

Docker's Popularity

- There is little doubt that Docker is currently one of the most popular open source, community led projects
- The total number of contributors to the Docker project reached 2,000 in November 2015, up from 900 in December 2014, and 460 in June 2014
- The total number of image pulls from the Docker Hub registry was 1.2 billion in November 2015, up from 100 million in December 2014, and 2.75 million in June 2014
- The total number of repositories on the Docker Hub registry was 240,000 in November 2015, up from 70,000 in December 2014, and 14,500 in June 2014
- The first DockerCon conference held in June 2014, and the DockerCon Europe conference held in December 2014, were oversubscribed by 400, and DockerCon 2015 hosted 2,000 people
- Despite its relative youth, Docker has been adopted into production by many organisations, and there are over 50 documented case studies
- The are in excess of 216 separate Docker meetup groups in many different countries across the world
- Docker is frequently referenced in the media, and is independently blogged about on a daily basis
- Major software and cloud providers either support, or are in the process of engineering support, for Docker on their platforms – notably, including Microsoft
- Docker has attracted almost $161 million in investment funding, with $95 million series D funding in April 2015

Notes



The Analogy

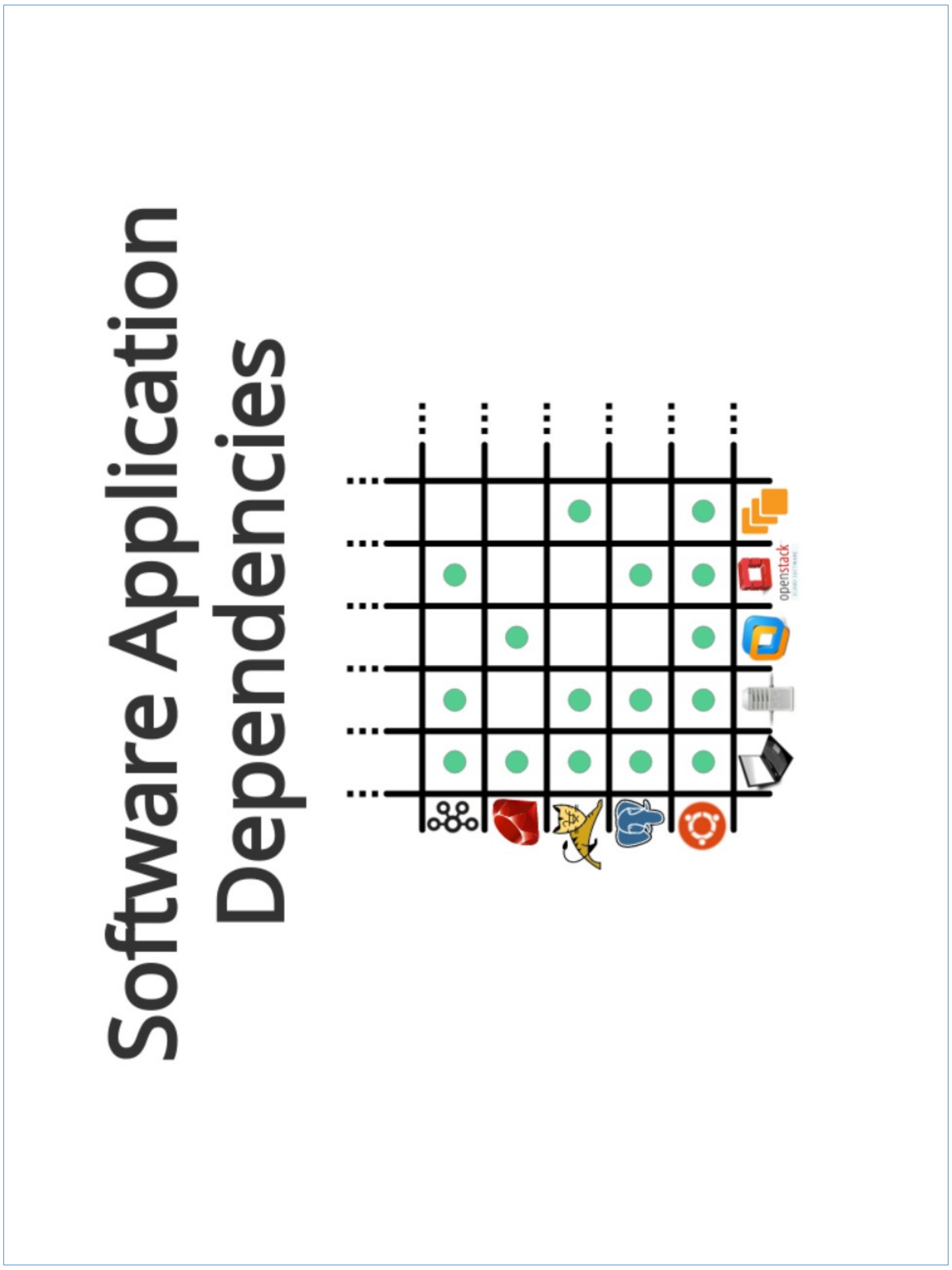
Before 20th century, freight transportation was inefficient and hazardous

The inter-modal shipping container revolutionised freight transportation

The Analogy

- Docker is often explained with reference to a shipping container analogy
- Before the introduction of a cargo transportation standard, goods were transported in all manner of types of containers in a ship's hold
- A ship could spend weeks in port being unloaded and re-loaded, often spending as much time in port as at sea, making the transportation of goods a very inefficient process
- The lack of a standard means of loading and containing goods meant that it was very hazardous work, and very expensive
- In the 1950s, Malcolm McLean developed the inter-modal shipping container, which eventually encouraged an ISO standard, and significantly reduced the cost of transporting cargo around the world ("A ship earns money only when she's at sea")
- Because of the inter-modal nature of shipping containers, they can be transported by road, rail and sea
- Nowadays, goods are transported around the world on large, specially built container ships, with the goods safely and efficiently contained within the confines of standard containers

Notes



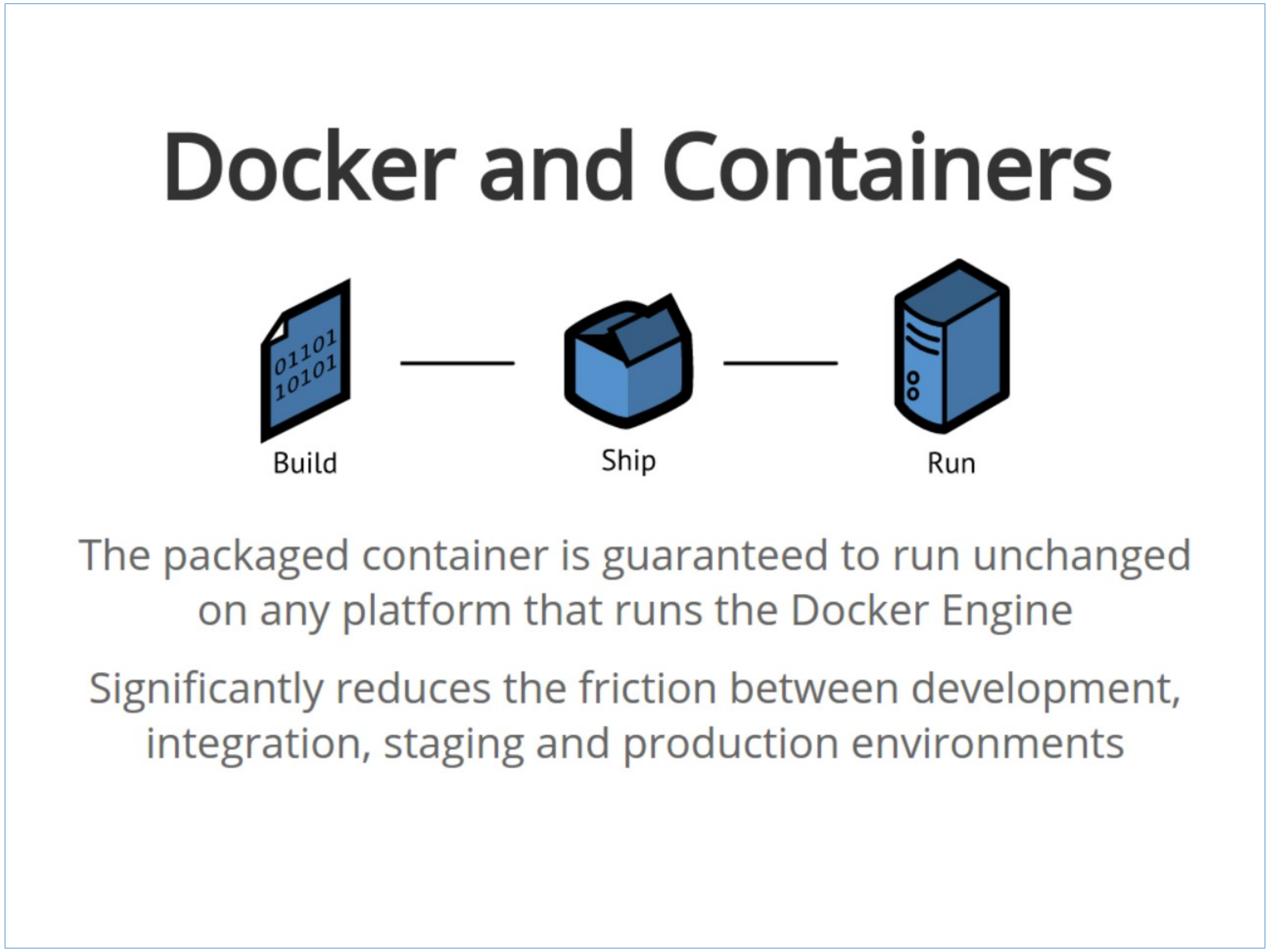
Software Application Dependencies

Software Application Dependencies

- In the IT world, we have a similar problem - software applications often consist of numerous inter-related components, which need to work together seamlessly to achieve the desired function(s)
- The components, themselves, vary according to their version and the way they are configured; libraries, packages, configuration files, binaries, RDBMS', message queues, server side code and extensions and packages, web servers, operating systems, platforms
- A software application may get deployed to many different environments during its lifecycle (laptop, physical server test environment, distributed servers in a datacentre, cloud infrastructure), and managing the dependencies in each environment is problematical
- The dependency problem introduces inefficiency and cost, and hinders;

    - expeditious time to market, thereby reducing a products' market potential, or
    - an organisation's ability to adapt and change quickly

Notes



Docker and Containers

Build
Ship
Run

The packaged container is guaranteed to run unchanged on any platform that runs the Docker Engine

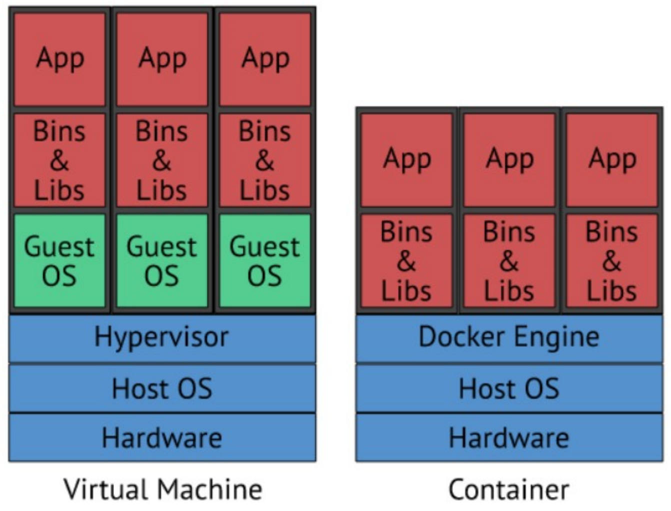
Significantly reduces the friction between development, integration, staging and production environments

Docker and Containers

- Docker seeks to resolve this conundrum, by facilitating the deployment of application components inside containers, along with the necessary dependencies that each component relies on
- It provides application components with just the required elements they need, whilst sharing the rest of the hosts' resources, with the host itself, and other containers
- Each application is guaranteed to run unchanged on any platform that runs the Docker daemon; for example, on a laptop, a virtual machine, a physical server, or even in the cloud
- Using Docker, software developers can create applications safe in the knowledge that they can be deployed across numerous environments, without the need to fret about the dependencies between components
- System Administrators can provide identical runtime environments for software applications, thereby increasing operational integrity

Notes

Containers vs VMs

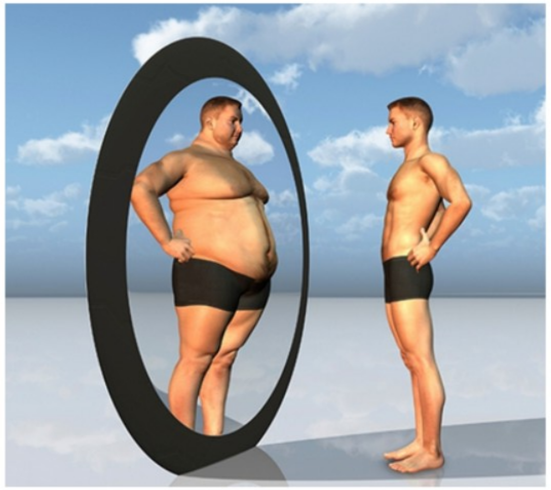


Containers vs VMs

- Virtual machines run on top of a hypervisor (either on bare metal or on a host operating system), which emulates a hardware layer on which a guest operating system runs
- Applications, and any other binaries or libraries that the application is dependent on, are then installed within the guest operating system
- By contrast, a container does not need a separate operating system, and shares the host's kernel
- Any dependent binaries or libraries are included as part of the container's filesystem, and, typically, the container will only run a single process or limited set of processes
- The absence of a guest operating system layer in the container context, makes them smaller and leaner than virtual machines, but at the cost of reduced isolation from the host

Notes

Containers vs VMs



VMs are inherently **more secure** than containers

VMs are better equipped to run **multiple processes**

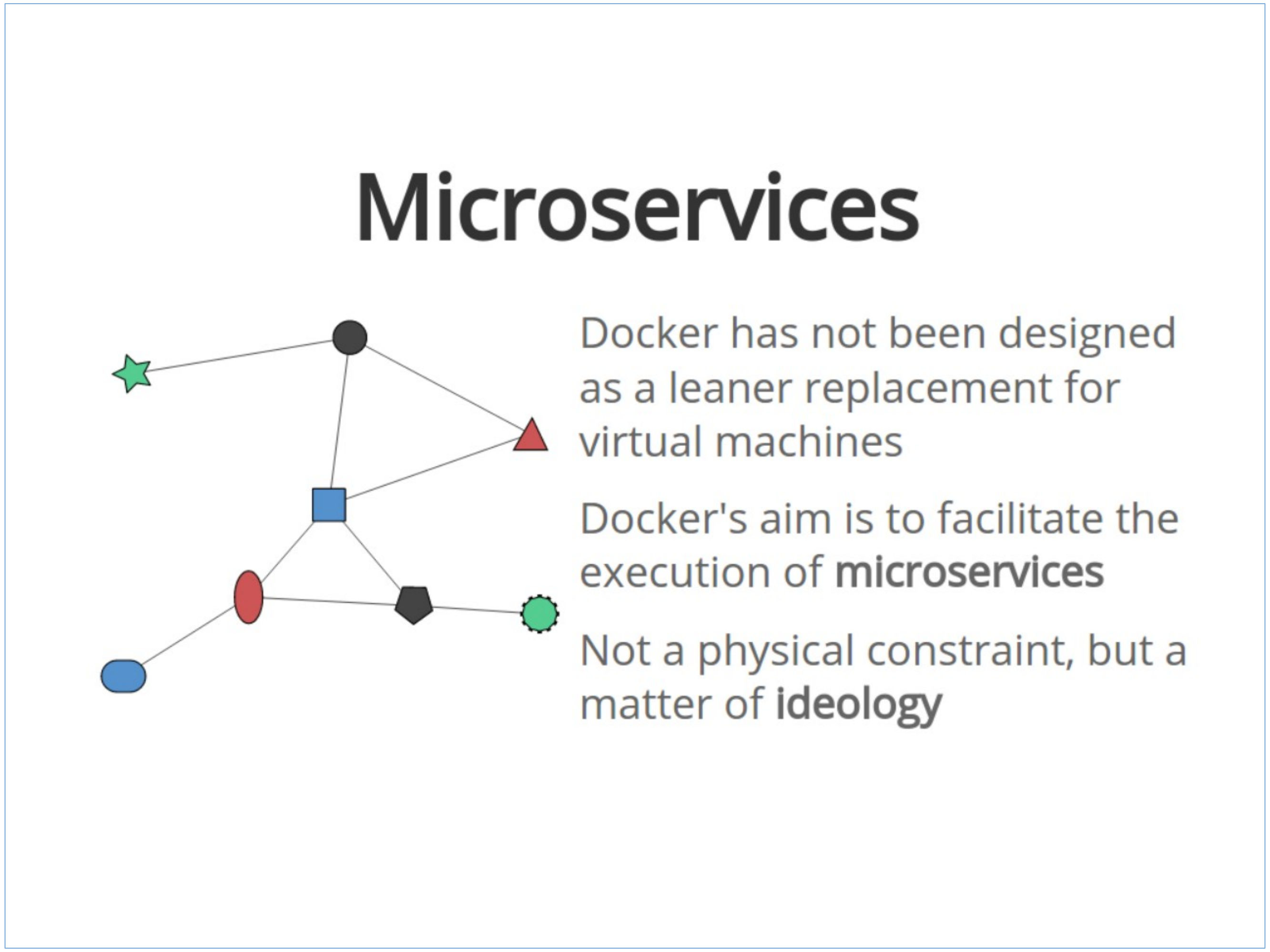
Containers are significantly **quicker** to spin up than VMs, and occupy a **smaller** footprint

The choice depends on the use case: horses for courses

Containers vs VMs

- It is a misconception that containers herald the demise of the hypervisor-based virtual machine
- Firstly, virtual machine technology is mature, and provides better workload isolation from the host, making it more difficult for miscreants to gain access to the host's kernel – a hacker would need to penetrate the kernel in the guest OS, and circumvent the hypervisor, before gaining access to the host's kernel
- Secondly, in some use cases, a full operating system is required, and in such situations it is more pertinent to use a traditional virtual machine
- If, however, the use case is to run a mircoservice, a full blown virtual machine is an extravagant choice, and a container running a single process is much more efficient
- Application components may consume megabytes of disk space and memory, whereas the supporting guest OS in a virtual machine may consume gigabytes in contrast
- The light nature of containers mean that a greater density of workloads can be achieved for a given hardware platform, and that start times can be measured in fractions of a second, rather than seconds or minutes
- Whilst there is no suggestion that containers are inherently insecure, because they are relatively novel, and are less isolated from the host than a virtual machine, many people choose to run their containers inside a virtual machine.
- Both VMware and Microsoft plan to provide container-based products wrapped inside a minimal virtual machine
- This may seem counter-intuitive, but the benefits of Docker's packaging combine with the maturity of workload isolation and tooling provided by a virtual machine
- It's even possible to run a virtual machine inside a container!
- In the case of virtual machines, the guest OS does not need to be the same flavour as the host OS – a Windows guest OS can run on a Linux host OS, and vice versa
- With containers, on the other hand, the host OS requires the process(es) in the container to share its kernel and its capabilities, and therefore its OS flavour

Notes



Microservices

Docker has not been designed as a leaner replacement for virtual machines

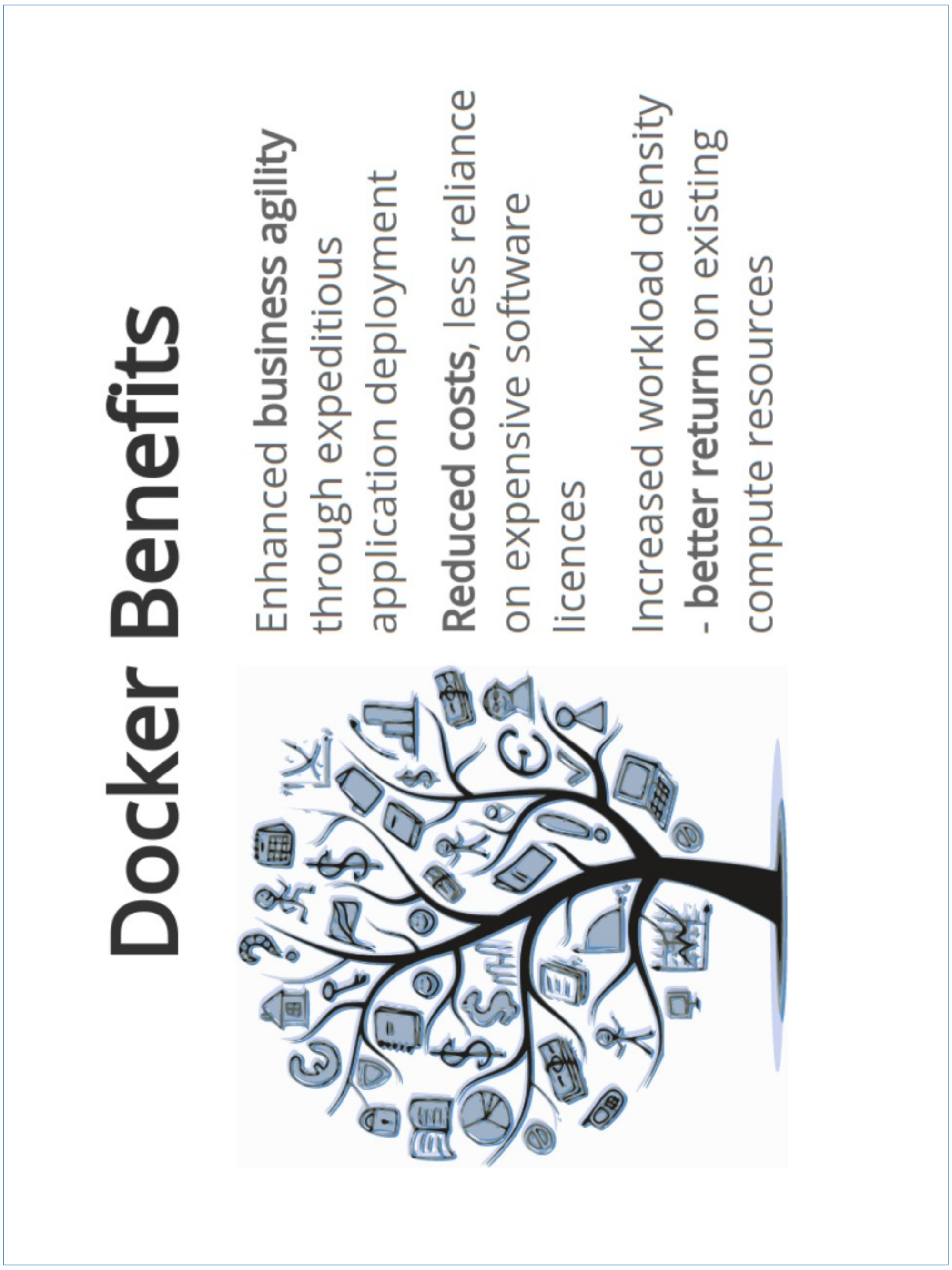
Docker's aim is to facilitate the execution of **microservices**

Not a physical constraint, but a matter of **ideology**

Microservices

- Docker has not been designed as a leaner replacement for virtual machines, although you may be forgiven for thinking so, if you read some of the hype written about Docker
- Docker's aim is to facilitate the execution of microservices, rather than monolithic applications or complete operating systems
- This is not a physical constraint, but a matter of ideology, and there is nothing stopping anyone from filling a container with numerous components - inevitably, however, shortcomings will arise as Docker is designed to run a single process inside a container
- A microservice is a small, independent, autonomous component of a larger application entity, that communicates with its peer services using lightweight communication, usually in the form of a REST API
- A microservice is commonly implemented as a single process
- The microservices architecture pattern is relatively new in concept, and presently lacks any formal definition or methodology, although there are plenty of events on the topic

Notes



Docker Benefits

Enhanced **business agility** through expeditious application deployment

Reduced costs, less reliance on expensive software licences

Increased workload density - **better return** on existing compute resources

Docker Benefits

Faster application delivery and integration

- Businesses increasingly need to be agile; at the very least they need to expeditiously respond to the market in which they operate, but ideally need to be ahead of the competition and in step with customer's expectations
- Software applications often facilitate the success or failure of a business' product or service, and therefore it is key to innovate and stay ahead of the competition
- The DevOps culture encourages faster and continuous application delivery, and Docker significantly aids this goal, with its 'run anywhere' capability
- Developers are able to mimic the end production environment more accurately, using lightweight containers, and can start and stop application components in fractions of a second

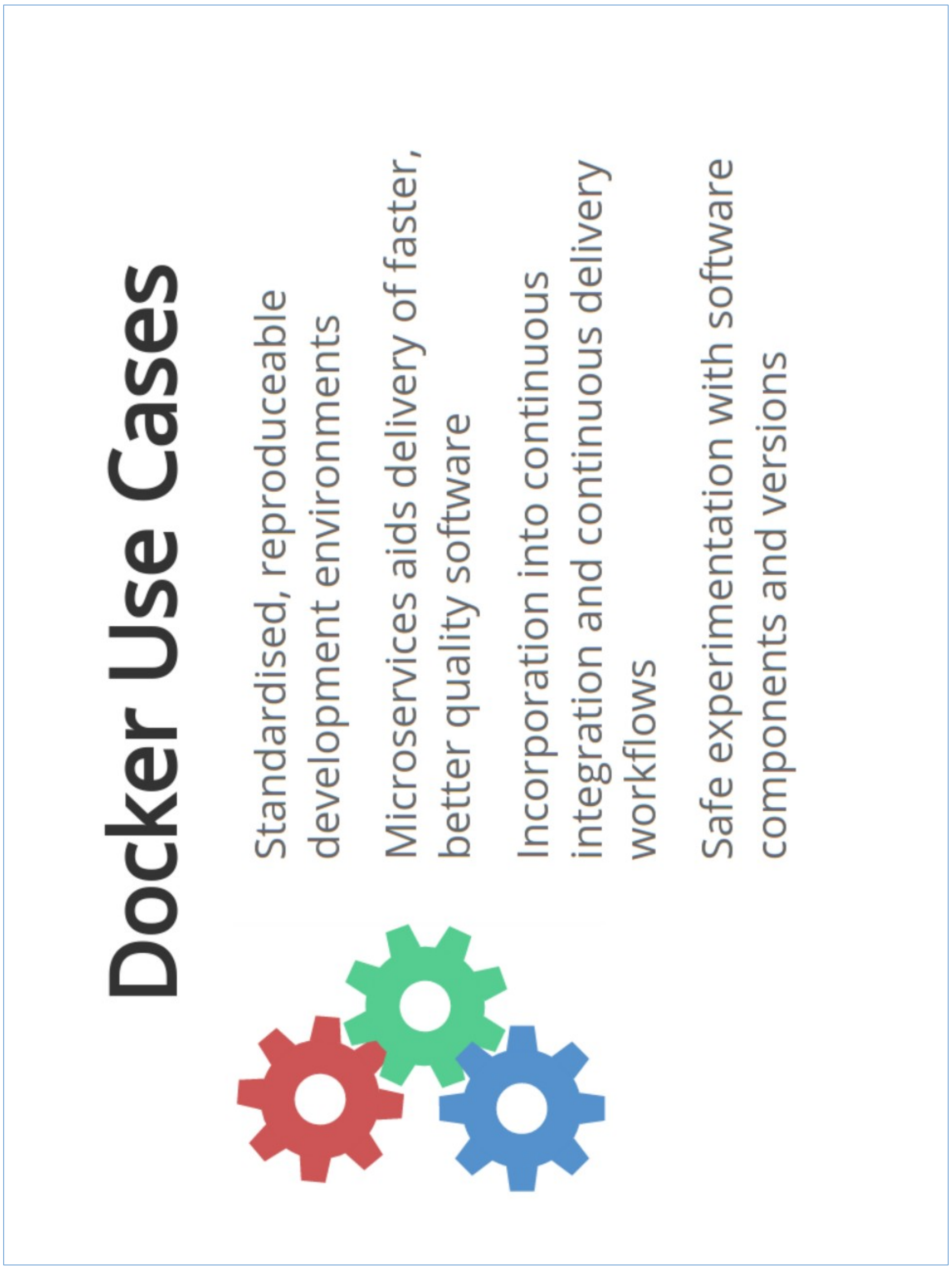
Community

- Docker is a community rather than a product, which brings substantial benefits to adopters, by providing specially crafted and curated application images, which helps to avoid re-inventing the wheel
- Docker is architected in such a way so as to allow third party 'plugins' to be overlaid onto particular parts of functionality, to best suit the requirement

Consolidation

- Because of its very light footprint compared to VMs, Docker helps to consolidate workloads by a further magnitude

Notes



Docker Use Cases

- Standardised, reproduceable development environments
- Microservices aids delivery of faster, better quality software
- Incorporation into continuous integration and continuous delivery workflows
- Safe experimentation with software components and versions

Docker Use Cases

Application Development

Docker enables software developers to more adequately replicate a production environment whilst developing. Docker containers, with their tiny footprint, enable those services to be run locally.

Developers who work in a team can work with consistent, reproducible development environments. Each team member can develop against the exact same environment, thereby reducing inconsistency and increasing productivity.

Microservices Implementation

Adopting the microservices architecture pattern provides a number of benefits, which ultimately allows for more expeditious software deployment. Docker containers are designed to run single processes, and thereby facilitate the creation and deployment of microservices.

Continuous Integration & Continuous Delivery

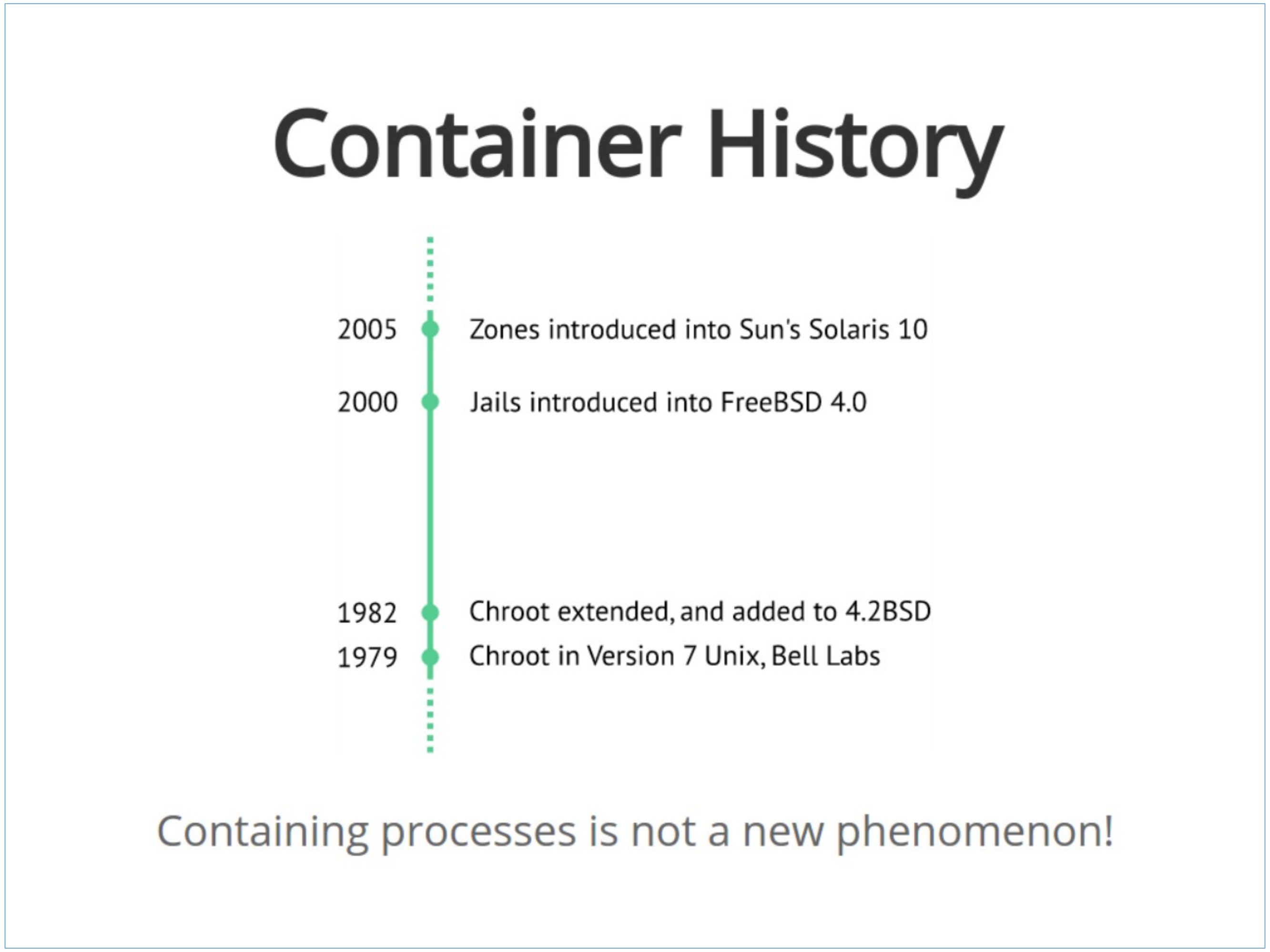
In conjunction with other tools, Docker containers can facilitate continuous integration (CI) and continuous delivery (CD) thereby allowing code changes to be delivered, ready for production in minimal time.

Docker containers are often used for running elements of CI software (i.e. Jenkins) that perform the build and test steps.

Experimentation

Docker can be used as a sandbox for experimenting with software applications without the consequential pollution of the host itself.

Notes



Container History

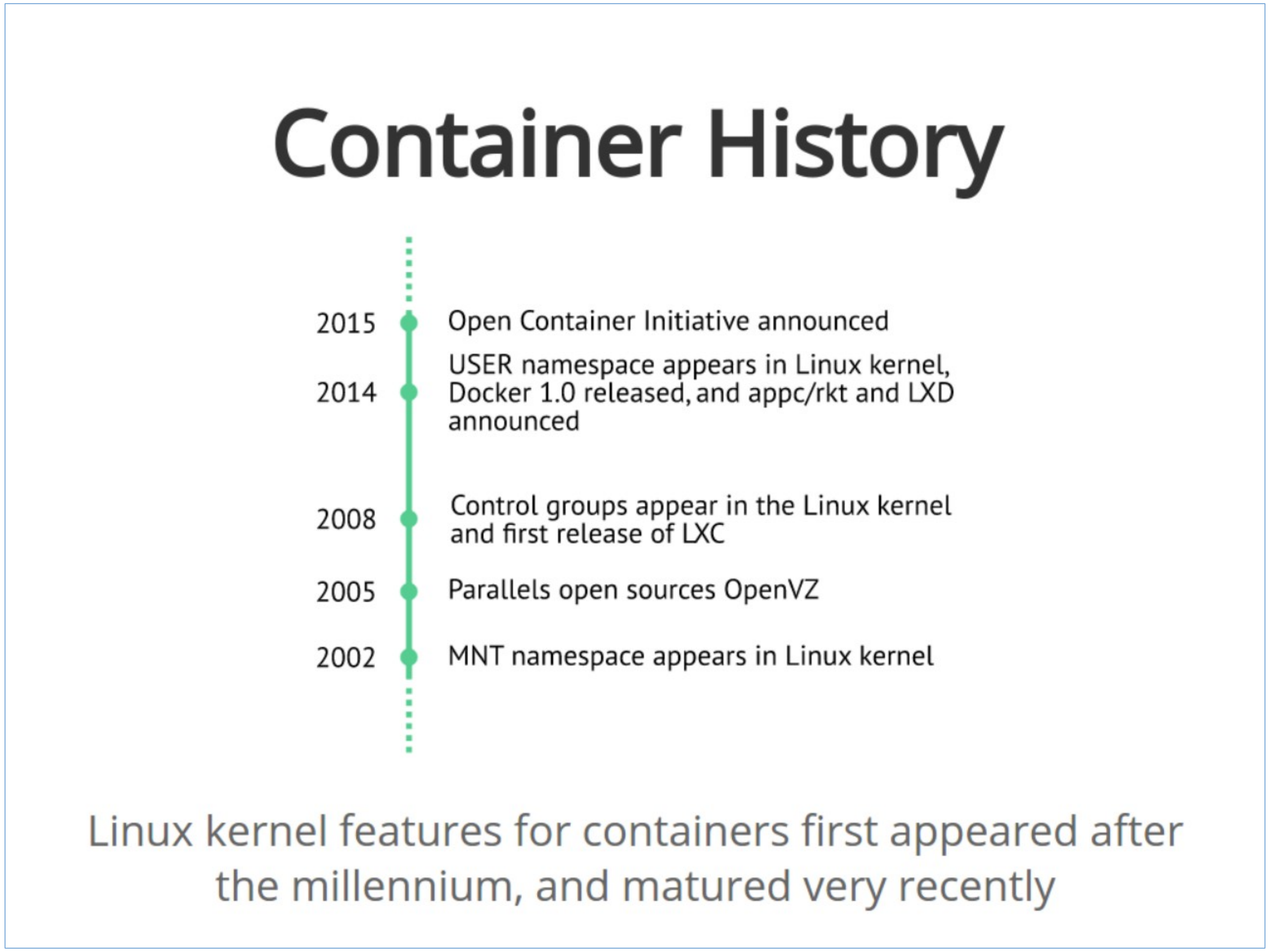
2005	Zones introduced into Sun's Solaris 10
2000	Jails introduced into FreeBSD 4.0
1982	Chroot extended, and added to 4.2BSD
1979	Chroot in Version 7 Unix, Bell Labs

Containing processes is not a new phenomenon!

Container History

- The concept of isolated environments for processes is not new, as attested to by the `chroot` capability for creating a new, contained filesytem for a process and its descendants
- The `chroot` capability dates back as far as 1979 to an early version of Unix, found its way into BSD Unix in 1982, and still exists in Linux distributions today
- However, `chroot` does not provide infallible isolation, and it is relatively easy to break out of an environment, if you have root privileges and the relevant tools
- With widespread acceptance of its vulnerabilities, an extension of the `chroot` concept was provided in the form of Jails in the FreeBSD operating system in 2000 (version 4.0), which provided for the isolation of processes, users and networks, as well as the filesystem
- Similarly, Sun Microsystem's Solaris OS provided a container capability called Solaris Zones in Solaris 10 released in 2005, and is still available today in Oracle's Solaris 11 and illumos and its distributions

Notes



Container History

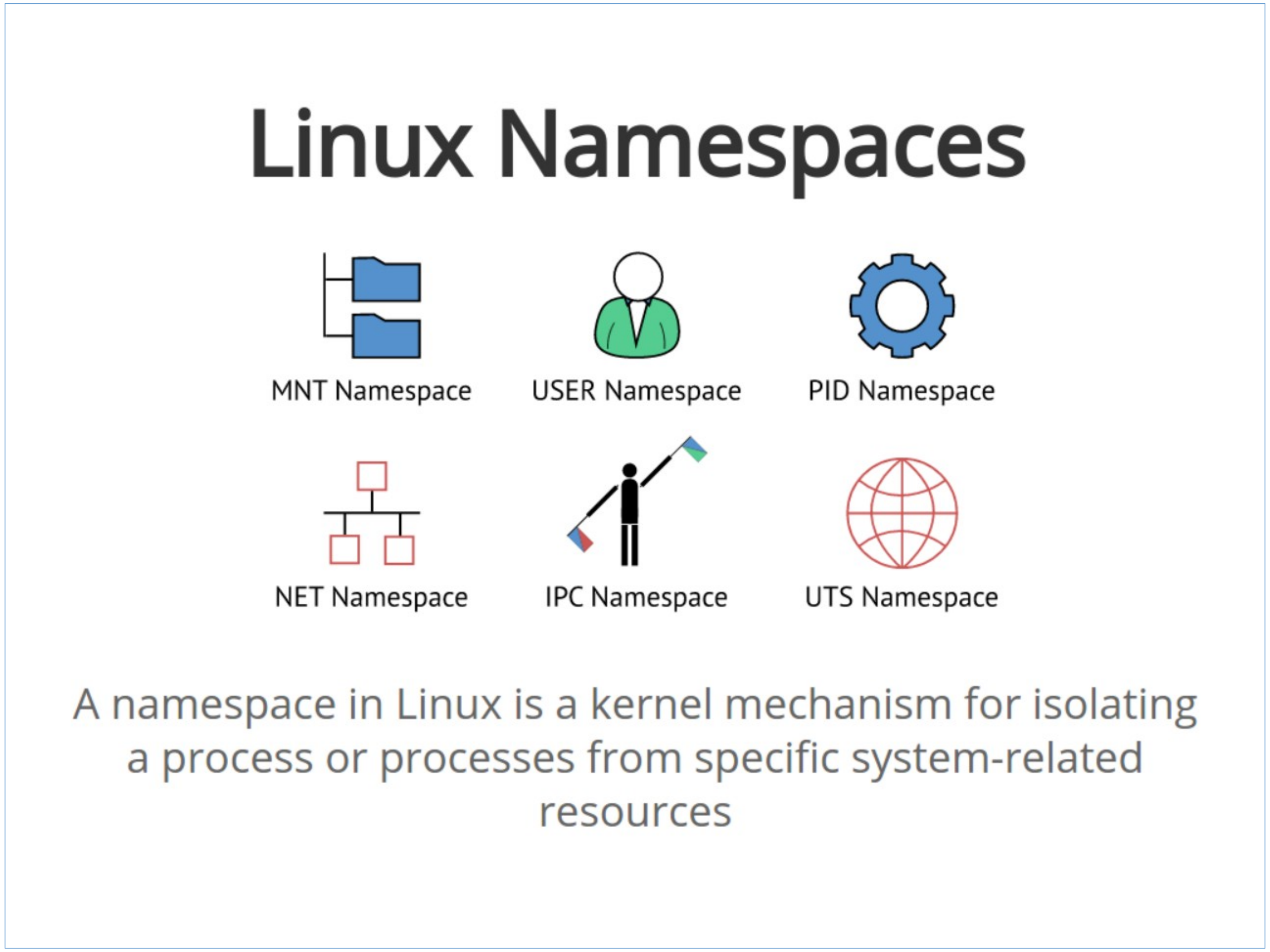
2015	Open Container Initiative announced
2014	USER namespace appears in Linux kernel, Docker 1.0 released, and appc/rkt and LXD announced
2008	Control groups appear in the Linux kernel and first release of LXC
2005	Parallels open sources OpenVZ
2002	MNT namespace appears in Linux kernel

Linux kernel features for containers first appeared after the millennium, and matured very recently

Container History

- Containers in the Linux world are based on two kernel capabilities: namespaces and control groups
- These features started appearing in the Linux kernel as long ago as 2002, with the first namespace implementation, and have been added to and refined ever since
- In 2005, Parallels open-sourced a container technology for Linux called OpenVZ, which relied on a custom built Linux kernel in order to utilise its comprehensive set of features
- Despite its rich feature set, OpenVZ has not had widespread acceptance in the Linux community, mainly because of the need for a patched or custom built kernel
- Google developed the resource management capability associated with containers, called control groups or cgroups, which first appeared in the Linux kernel in 2008
- Later in 2008, an open-source project called LinuX Containers (LXC) provided its first release of userland tools for creating and managing containers on the Linux platform
- LXC, which is heavily supported by Canonical (the company behind Ubuntu), has matured into a comprehensive toolset since 2008
- In its pre-release versions, Docker utilised LXC as the implementation means for creating and destroying containers, but replaced it in Docker 0.9 as the default "driver" with its own library of Go routines, called libcontainer
- Toward the end of 2014, Canonical announced a complimentary open source project to LXC, called LXD, which aims to be a new generation hypervisor, providing enhanced isolation, whilst maintaining the speed and lightness of containers
- Controversially, CoreOS announced a Docker-alternative container runtime called rkt (pronounced Rocket), just before Dockercon Europe in December, 2014
- Subsequently, CoreOS and Docker have joined others in the establishment of the Open Container Initiative

Notes



Linux Namespaces

MNT Namespace

USER Namespace

PID Namespace

NET Namespace

IPC Namespace

UTS Namespace

A namespace in Linux is a kernel mechanism for isolating a process or processes from specific system-related resources

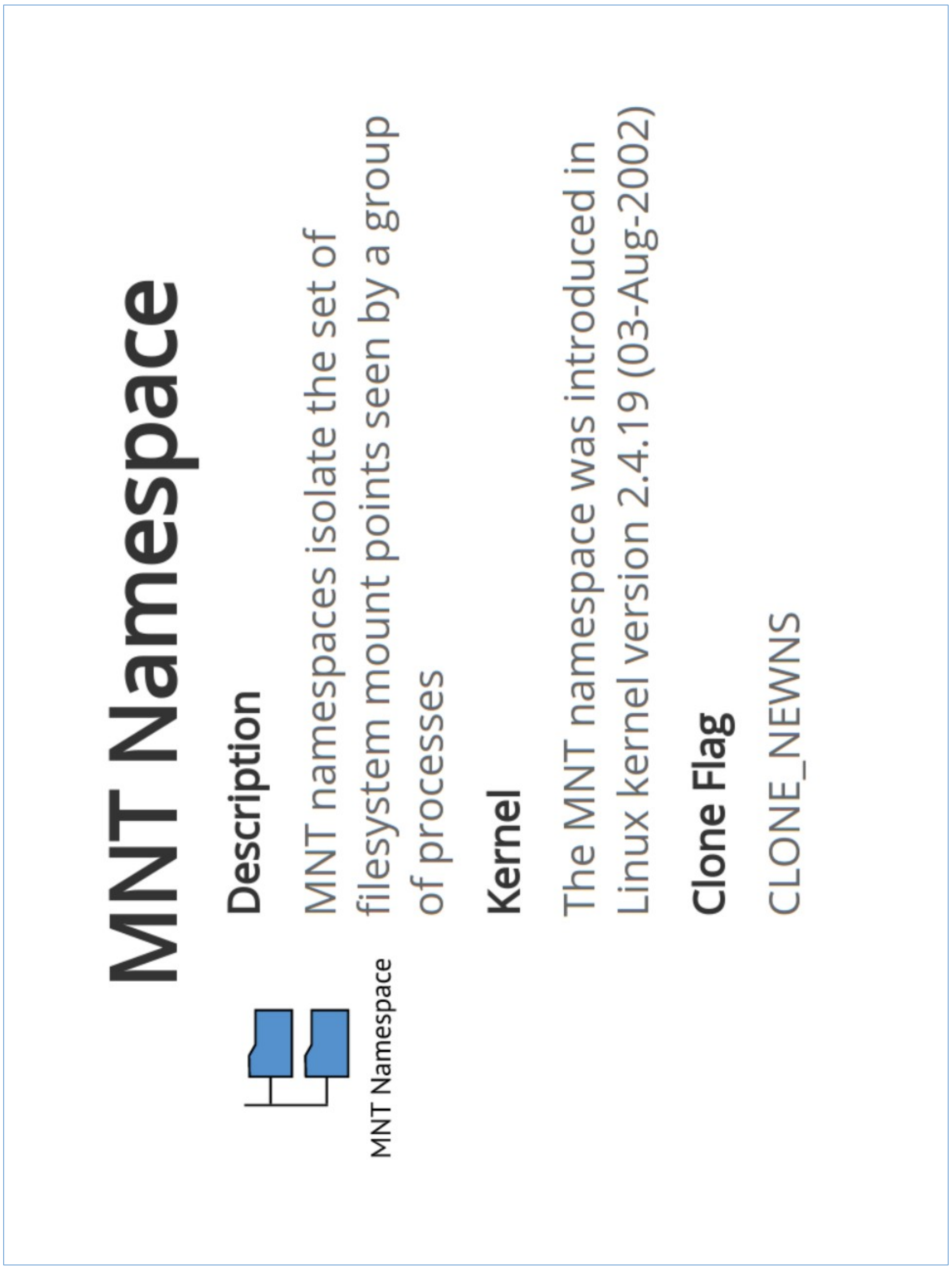
Linux Namespaces

- A namespace in Linux is a kernel mechanism for isolating a process or processes, so that the process(es) have a different viewpoint of the system to the rest of the processes on the system
- Each namespace variant abstracts a system resource, so that the processes within the namespace just see an isolated instance of that resource
- Namespaces allow isolated processes to see the same or different values for resources within the namespace to those outside the namespace, without fear of conflict or reprisal
- Whilst it was originally intended for there to be 10 namespace variants [1], only 6 have been implemented to date
- Processes are assigned to namespaces using the following system calls:

`clone`:	creates a new process, and for each namespace flag passed to the system call, a corresponding namespace of the relevant type is created, in which the process is placed
`unshare`:	the calling process is moved to a new namespace(s) for each corresponding namespace flag passed to the system call
`setns`:	the calling process is moved to an existing namespace, as identified by a file descriptor passed to the system call

[1] Multiple Instances of the Global Linux Namespaces, Eric W. Biederman, Proceedings of the Linux Symposium 2006 Vol. 1

Notes



MNT Namespace

MNT Namespace

Description

MNT namespaces isolate the set of filesystem mount points seen by a group of processes

Kernel

The MNT namespace was introduced in Linux kernel version 2.4.19 (03-Aug-2002)

Clone Flag

CLONE_NEWNS

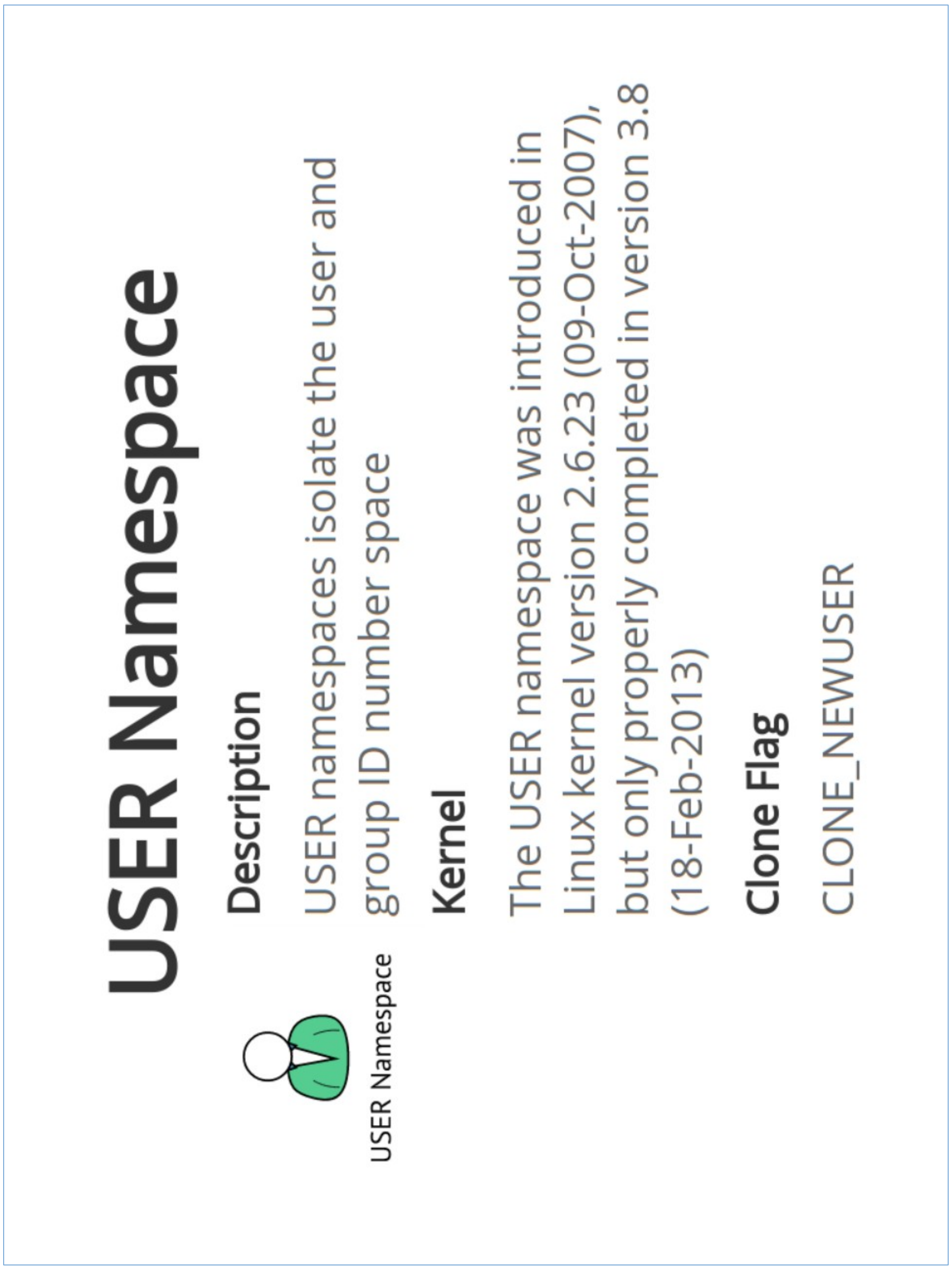
MNT Namespace

Purpose	MNT namespaces isolate the set of filesystem mount points seen by a group of processes
Kernel	The MNT namespace was introduced in Linux kernel version 2.4.19 (03-Aug-2002)
Clone Flag	CLONE_NEWNS

- Uses the flag `CLONE_NEWNS` rather than the expected `CLONE_NEWMNT` as it was the first namespace implemented, and further namespaces were not necessarily anticipated
- Upon creation, the MNT namespace receives a copy of its parent's mounted filesystem data
- Thereafter, each separate MNT namespace sees its own set of mounts only
- Mount point changes in the new MNT namespace can be seen within the namespace, but not the namespaces associated with the process' parent or its siblings, and vice versa*
- Parent or calling process requires `CAP_SYS_ADMIN` privileges

* True by default, but subject to mount propagation

Notes



USER Namespace

USER Namespace

Description

USER namespaces isolate the user and group ID number space

Kernel

The USER namespace was introduced in Linux kernel version 2.6.23 (09-Oct-2007), but only properly completed in version 3.8 (18-Feb-2013)

Clone Flag

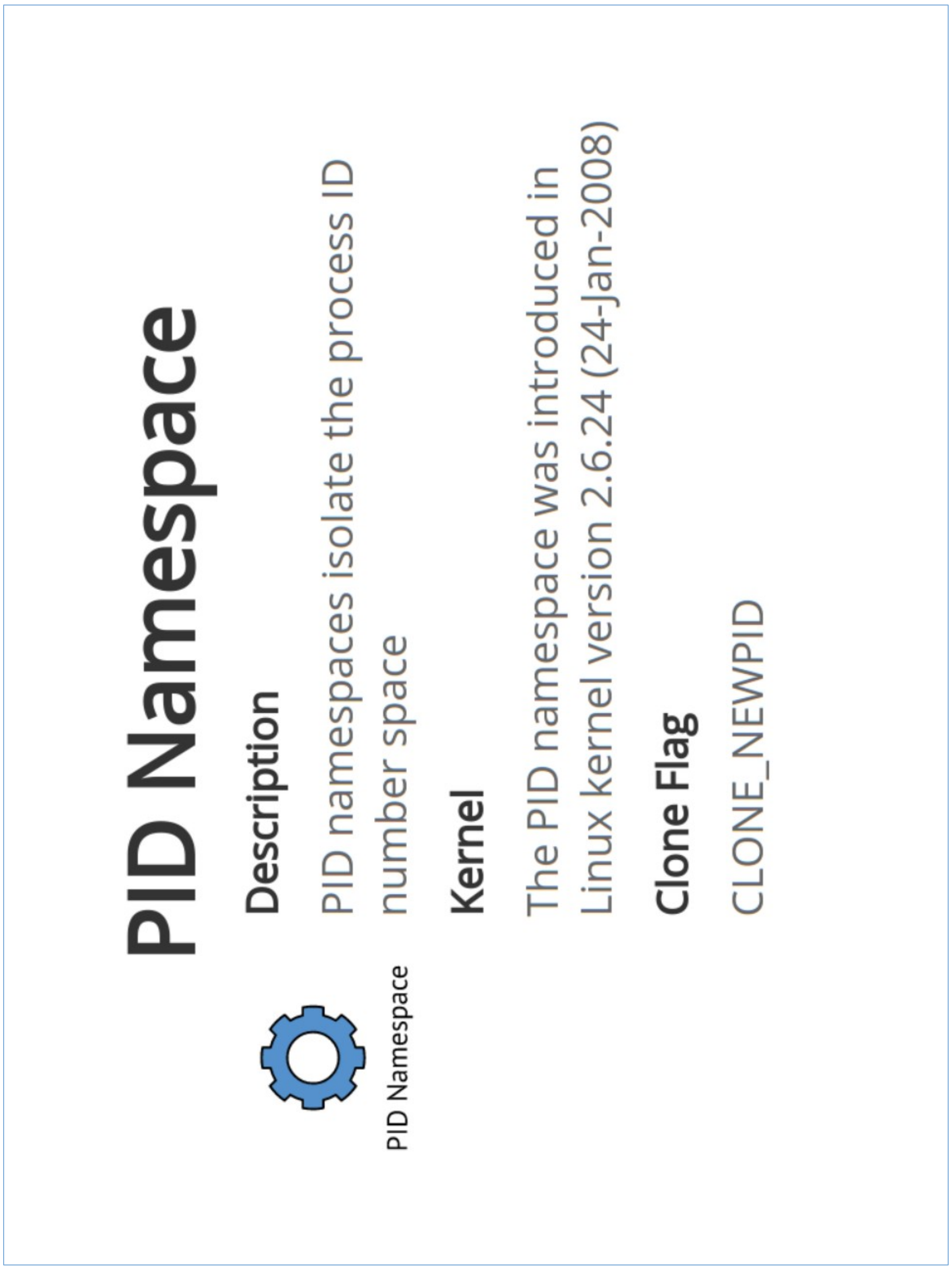
CLONE_NEWUSER

USER Namespace

Purpose	USER namespaces isolate the user and group ID number spaces
Kernel	The USER namespace was introduced in Linux kernel version 2.6.23 (09-Oct-2007), but only properly completed in version 3.8 (18-Feb-2013)
Clone Flag	CLONE_NEWUSER

- Enables a process' UIDs and GIDs to be different in different namespaces
- Enables a process to be privileged within a USER namespace (i.e. `root`), but remain an unprivileged user outside of the namespace
- Unlike invocation of the other namespace types, invoking USER namespaces does not require special privileges
- If a system call specifies the CLONE_NEWUSER flag along with additional namespace flags, the USER namespace is created first, and the calling process or child assumes sufficient privileges to establish the other specified namespaces
- UID/GID mappings for any given user are controlled using the allowed mappings defined for the user in `/etc/subuid` and `/etc/subgid`
- The per process mappings can be viewed in the `/proc/<pid>/uid_map` and `/proc/<pid>/gid_map` files

Notes



PID Namespace

PID Namespace

Description

PID namespaces isolate the process ID number space

Kernel

The PID namespace was introduced in Linux kernel version 2.6.24 (24-Jan-2008)

Clone Flag

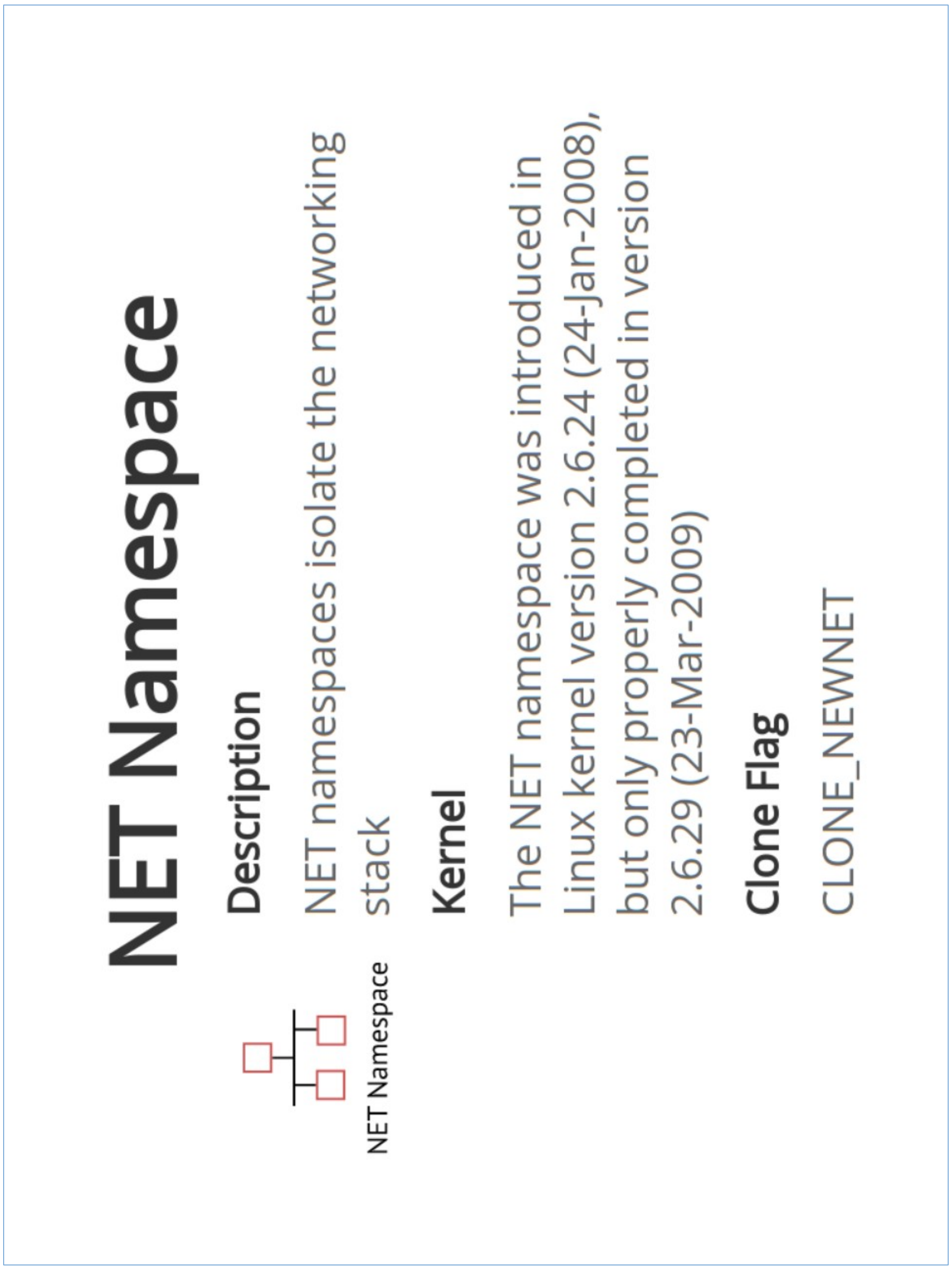
CLONE_NEWPID

PID Namespace

Purpose	PID namespaces isolate the process ID number space
Kernel	The PID namespace was introduced in Linux kernel version 2.6.24 (24-Jan-2008)
Clone Flag	CLONE_NEWPID

- Different processes in different PID namespaces can have the same PID
- A process in a given PID namespace can see (and interact with) all processes in its own namespace and all child PID namespaces
- A process in a given PID namespace cannot see (or interact with) any processes in its parent or sibling PID namespaces
- The first process created in a new namespace has PID 1, and is init-like in its behaviour (in that it will assume parenthood of orphaned processes in the same namespace)
- PID namespaces can be nested to 32 levels
- A process running in a child PID namespace will have more than one PID - a PID in its own namespace, and a PID in each of its ancestral namespaces
- Parent or calling process requires `CAP_SYS_ADMIN` privileges

Notes



NET Namespace

NET Namespace

Description

NET namespaces isolate the networking stack

Kernel

The NET namespace was introduced in Linux kernel version 2.6.24 (24-Jan-2008), but only properly completed in version 2.6.29 (23-Mar-2009)

Clone Flag

CLONE_NEWNET

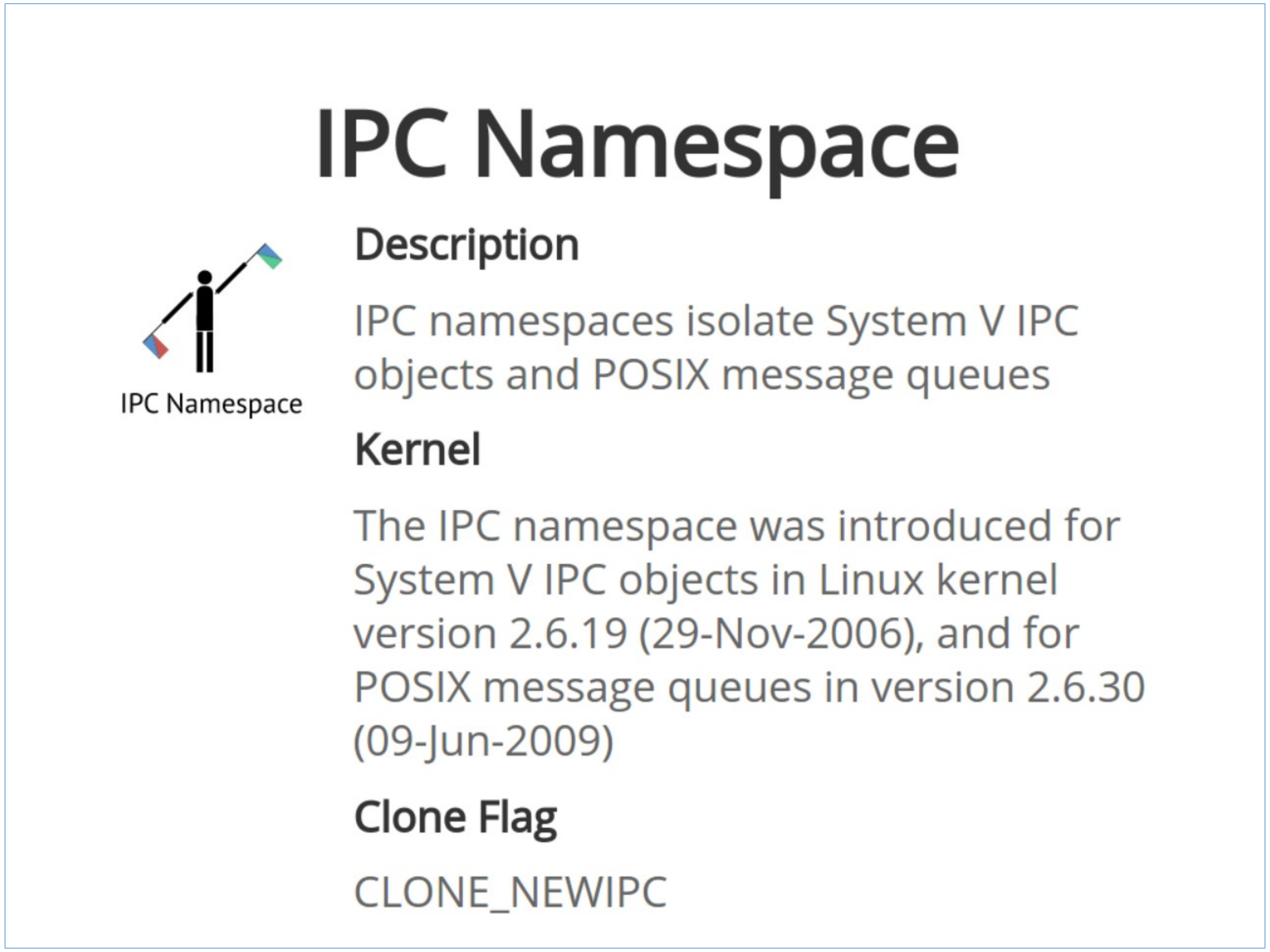
NET Namespace

Purpose	NET namespaces isolate the networking stack
Kernel	The NET namespace was introduced in Linux kernel version 2.6.24 (24-Jan-2008), but only properly completed in version 2.6.29 (23-Mar-2009)
Clone Flag	CLONE_NEWNET

- Each NET namespace has its own network devices*, routing tables, firewall rules, procfs & sysfs entries, and port number space
- NET namespaces are created with a single loopback interface, which is independent
- Inter NET namespace network communication is achieved via virtual network device pairs
- A child NET namespace can communicate with the outside world via a bridge or with NAT
- On NET namespace termination, physical network devices are moved to the default NET namespace
- Parent process requires `CAP_SYS_ADMIN` capability

* A physical network device exists in one NET namespace only

Notes



IPC Namespace

IPC Namespace

Description

IPC namespaces isolate System V IPC objects and POSIX message queues

Kernel

The IPC namespace was introduced for System V IPC objects in Linux kernel version 2.6.19 (29-Nov-2006), and for POSIX message queues in version 2.6.30 (09-Jun-2009)

Clone Flag

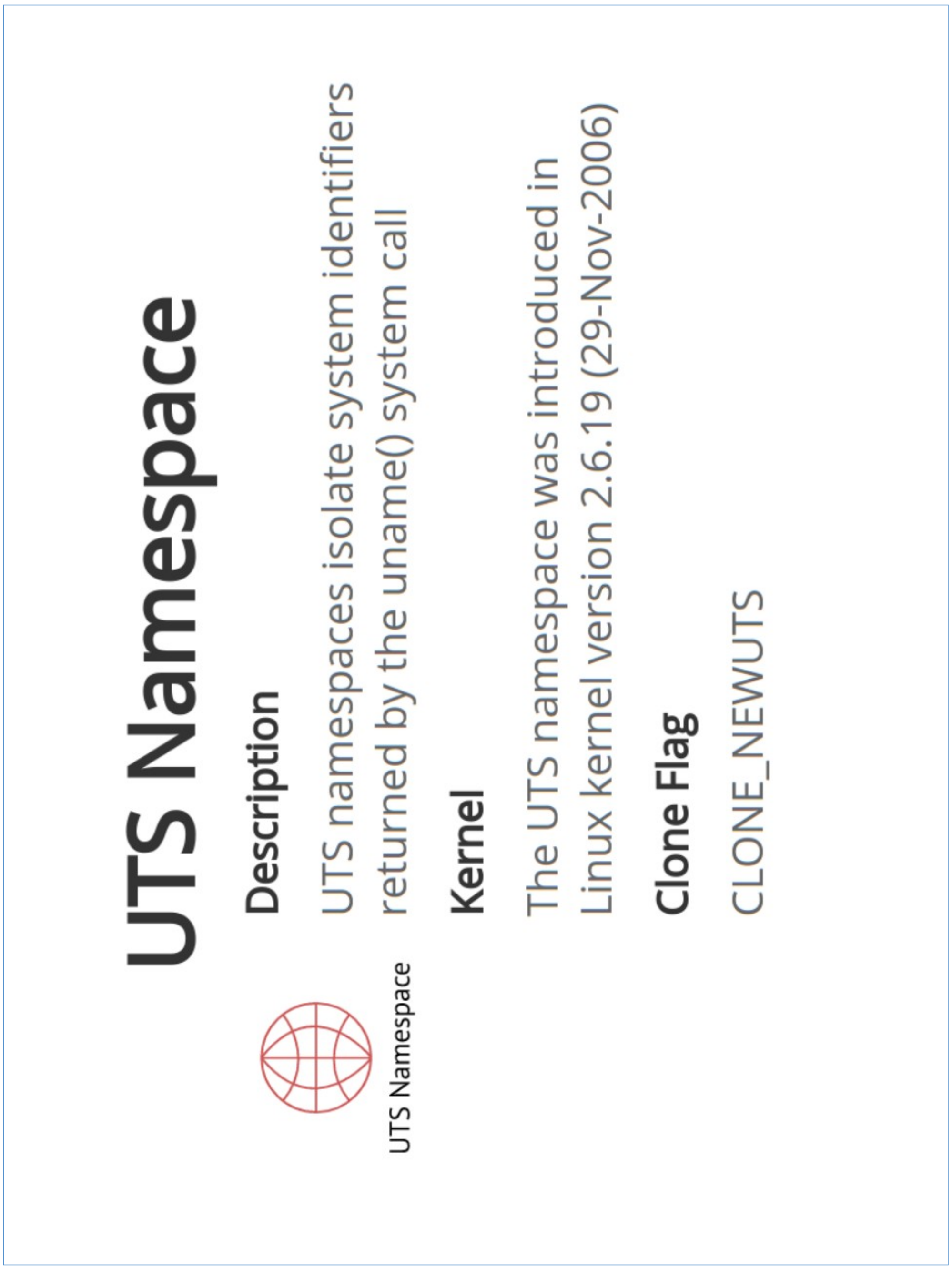
CLONE_NEWIPC

IPC Namespace

Purpose	IPC namespaces isolate System V IPC objects and POSIX message queues
Kernel	The IPC namespace was introduced for System V IPC objects in Linux kernel version 2.6.19 (29-Nov-2006), and for POSIX message queues in version 2.6.30 (09-Jun-2009)
Clone Flag	CLONE_NEWIPC

- System V IPC objects take the form of message queues, semaphore sets and shared memory segments
- System V IPC is largely obsolete nowadays, although still used by some vogue(ish) software
- IPC objects created in a given IPC namespace are only visible to processes within that namespace
- IPC objects are destroyed when the final member process of the namespace is terminated
- Parent or calling process requires `CAP_SYS_ADMIN` capability

Notes



UTS Namespace

UTS Namespace

Description

UTS namespaces isolate system identifiers returned by the uname() system call

Kernel

The UTS namespace was introduced in Linux kernel version 2.6.19 (29-Nov-2006)

Clone Flag

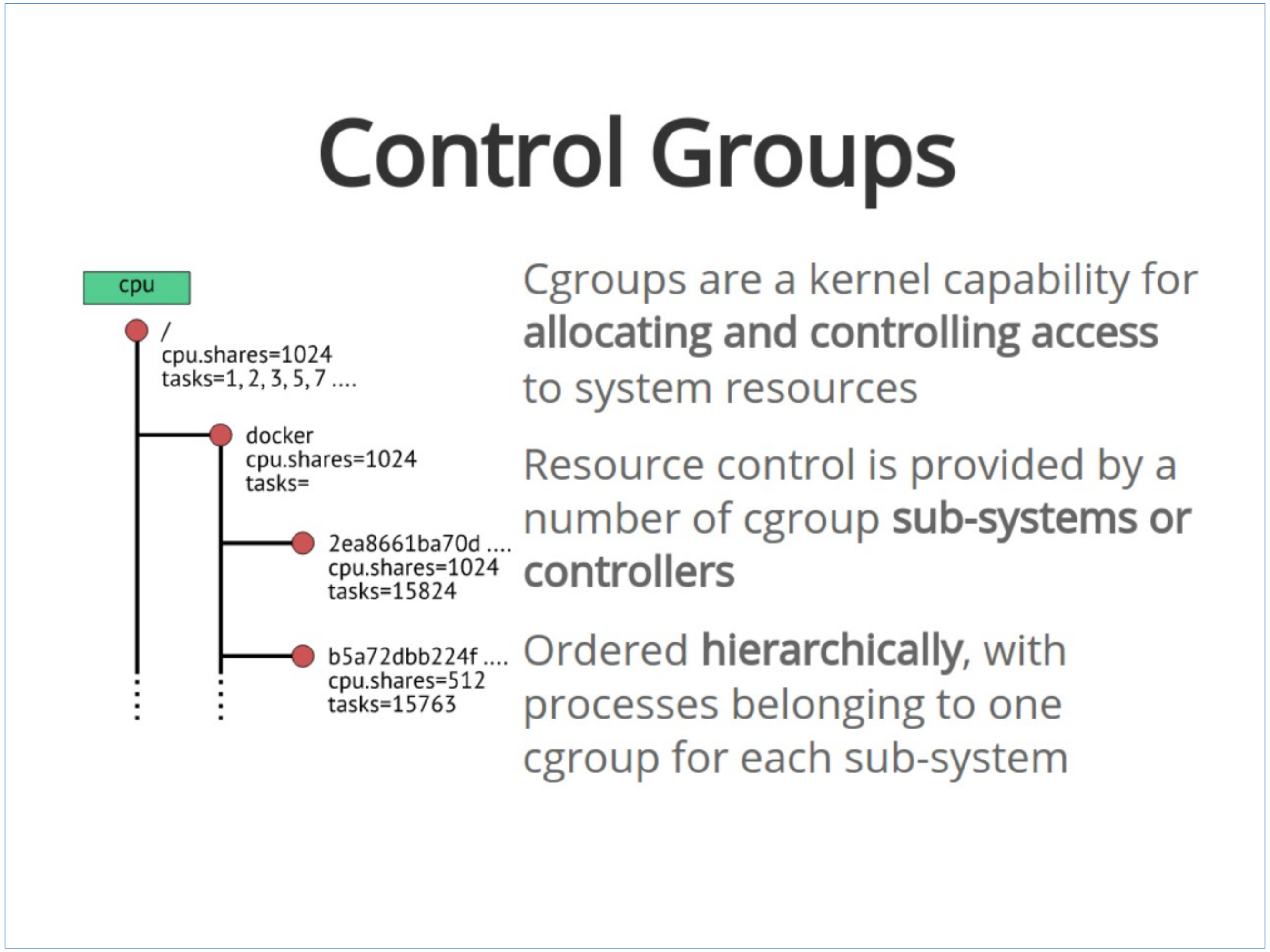
CLONE_NEWUTS

UTS Namespace

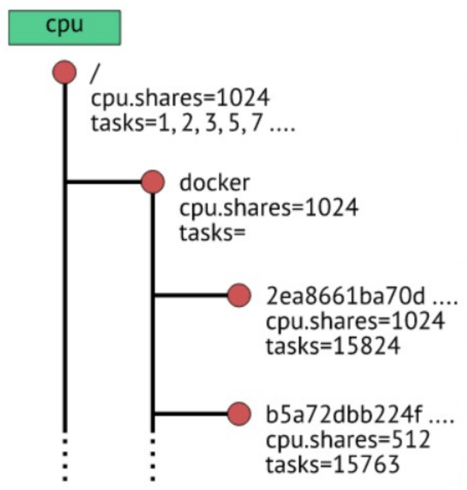
Purpose	UTS namespaces isolate system identifiers returned by the `uname()` system call
Kernel	The UTS namespace was introduced in Linux kernel version 2.6.19 (29-Nov-2006)
Clone Flag	CLONE_NEWUTS

- The term "UTS" derives from the name of the structure passed to the `uname()` system call: `struct utsname` - "UNIX Time-sharing System“
- A child UTS namespace inherits the identifiers of its parent namespace
- The UTS namespace was originally intended for changing the hostname and NIS domainname in the child namespace
- The NIS directory service is not widely used any more
- The hostname can be changed using `sethostname()` system call
- Parent or calling process requires `CAP_SYS_ADMIN` privileges

Notes



Control Groups



Cgroups are a kernel capability for **allocating and controlling access** to system resources

Resource control is provided by a number of cgroup **sub-systems or controllers**

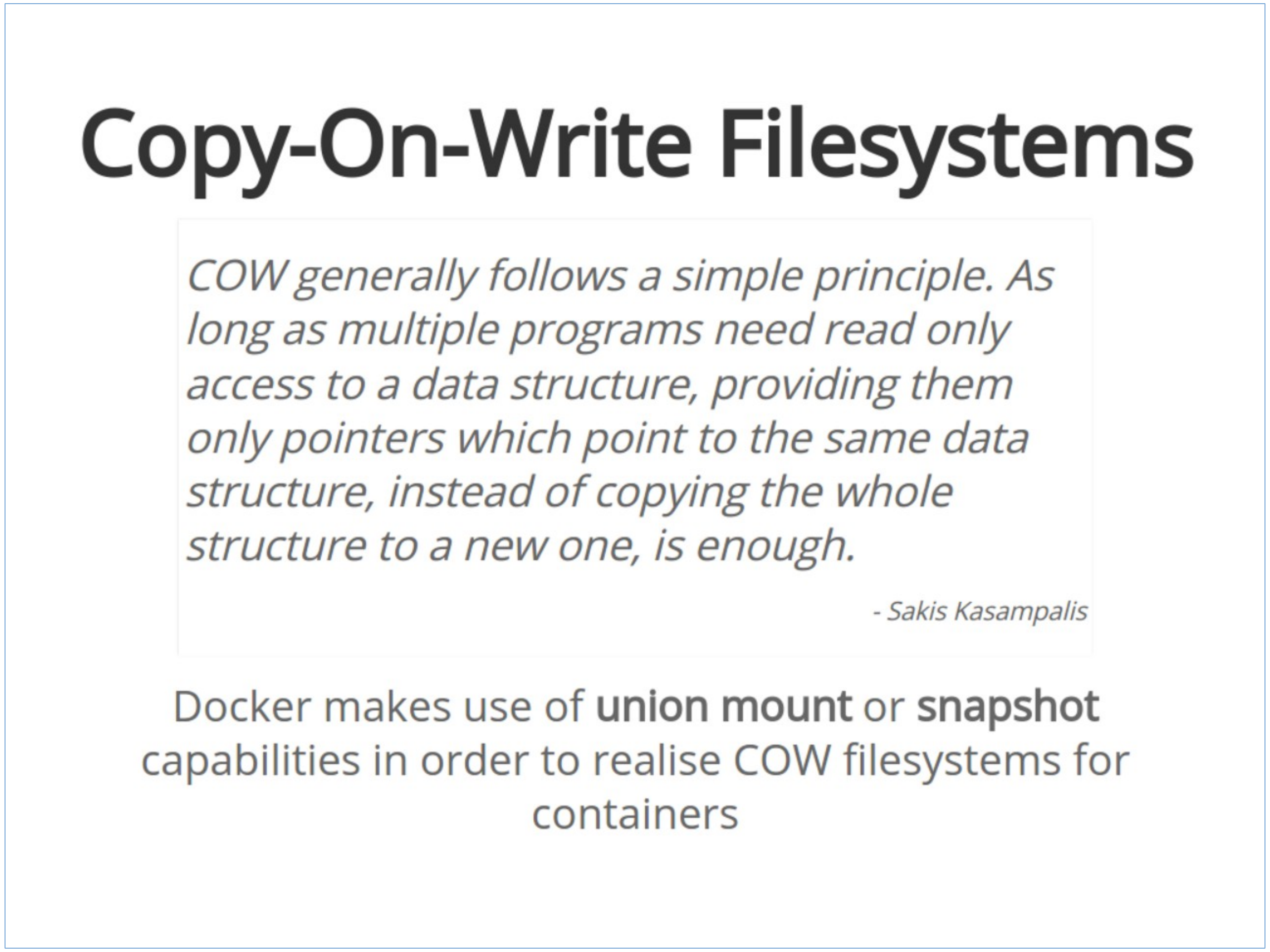
Ordered **hierarchically**, with processes belonging to one cgroup for each sub-system

Control Groups

- Control groups (or cgroups) are a kernel capability for allocating and controlling access to system resources
- Cgroups limit and track the usage of the system's resources
- They are ordered hierarchically, with child cgroups inheriting some of their parent's attributes
- Resource control is provided by a number of cgroup sub-systems or controllers
- A system's processes (or tasks in cgroup vernacular) belong in one specific cgroup within the hierarchy for each cgroup controller
- It's possible to fine tune the resources available to a set of processes, by setting the controller parameters for the cgroup they belong to
- The cgroup sub-systems are:

Sub-system	**Purpose**
cpuset	an isolation sub-system, which ties the processes which belong to a cgroup to subsets of a system's CPUs and memory nodes
cpu	a resource control sub-system, which uses the kernel's scheduler to share the CPU's bandwidth between processes according to the parameters set in each control group
cpuacct	a control sub-system, which accounts for the CPU bandwidth used by processes
memory	a resource control sub-system, which limts the memory usage for processes according to the parameters set in each control group
devices	an isolation sub-system, which limits processes' access to a system's devices
freezer	a control sub-system, which suspends or resumes the execution of processes
blkio	a resource control sub-system, which limits processes' I/O access to block devices (e.g. disk drives)

Notes



Copy-On-Write Filesystems

COW generally follows a simple principle. As long as multiple programs need read only access to a data structure, providing them only pointers which point to the same data structure, instead of copying the whole structure to a new one, is enough.

- Sakis Kasampalis

Docker makes use of **union mount** or **snapshot** capabilities in order to realise COW filesystems for containers

Copy-On-Write Filesystems

- Copy-On-Write (COW) allows processes or programs to share the same data structure, but as a read-only entity
- When one of the processes needs to write to the data structure, it makes a private copy of the data before it alters its contents
- Docker uses the COW properties of certain filesystem technologies in order to assemble a container's filesystem from the numerous layers of a Docker image, which are read-only, and provides an additional RW layer to enable copy-on-write
- The COW properties enable different containers to share much of the contents of their filesystems, thereby reducing storage requirements and enhancing container start-up speed
- The layers are assembled into a single coherent, logical filesystem representation, either through a 'union mount' of different source directories, or via 'snapshots' of the incremental changes from layer to layer
- Union mount filesystem varieties used by Docker include aufs and overlay
- Snapshot varieties used by Docker include btrfs, zfs and an implementation of the devicemapper framework using the thin provisioning target

Notes