Distributed Systems

Chun-Feng Liao

廖峻鋒

Department of Computer Science

National Chengchi University

Distributed Systems

Container

Chun-Feng Liao

廖峻鋒

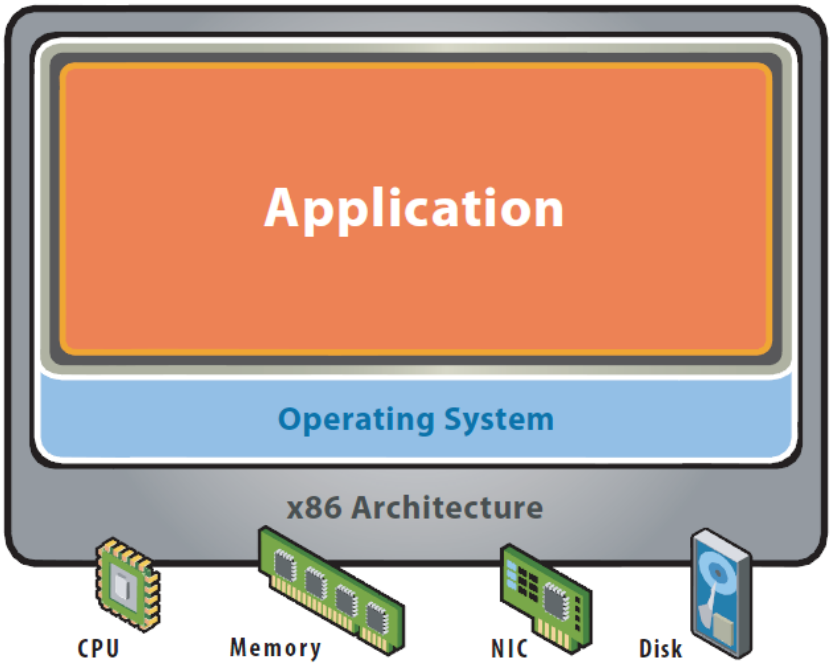
Dept. of Computer Science

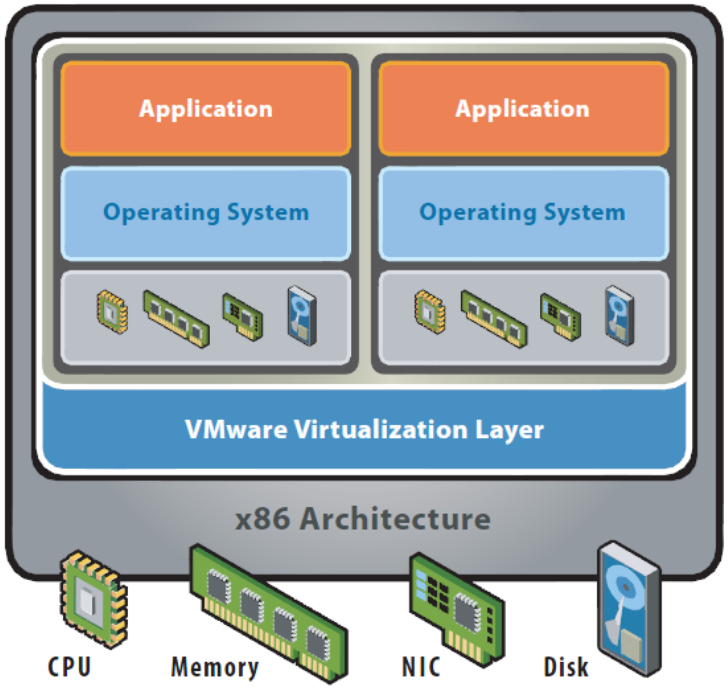
National Chengchi University

Virtualization Technology

- Virtualization
  - To create a software-based version of something
    - Something = OS, Database, Server, Storage, Network…
- Virtual Machine
  - A software-based implementation of some real (hardware-based) computer
- Virtual Machine Monitor (VMM, or called Hypervisor)
  - The software that creates and manages the execution of virtual machines
  - Essentially an operating system

Virtual Machines





Before Virtualization:

- Single OS image per machine
- Software and hardware tightly coupled
- Running multiple applications on same machine often creates conflict
- Underutilized resources
- Inflexible and costly infrastructure

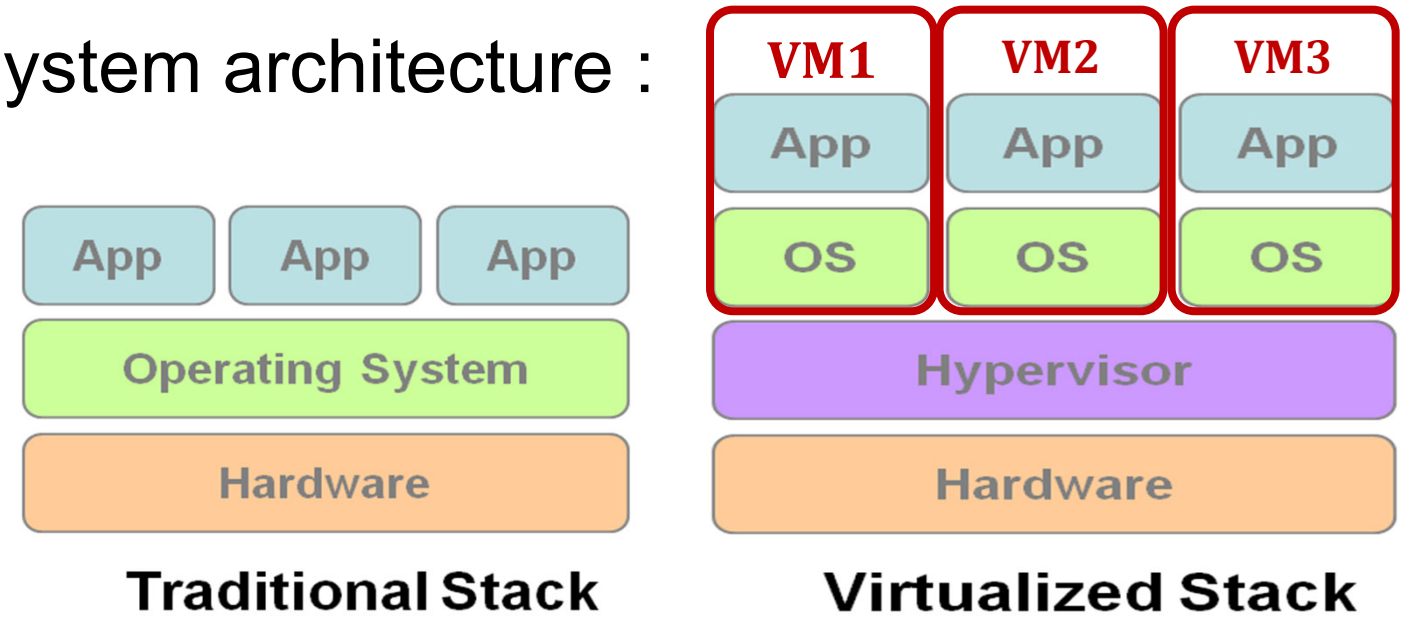
After Virtualization:

- Hardware-independence of operating system and applications
- Virtual machines can be provisioned to any system
- Can manage OS and application as a single unit by encapsulating them into virtual machines

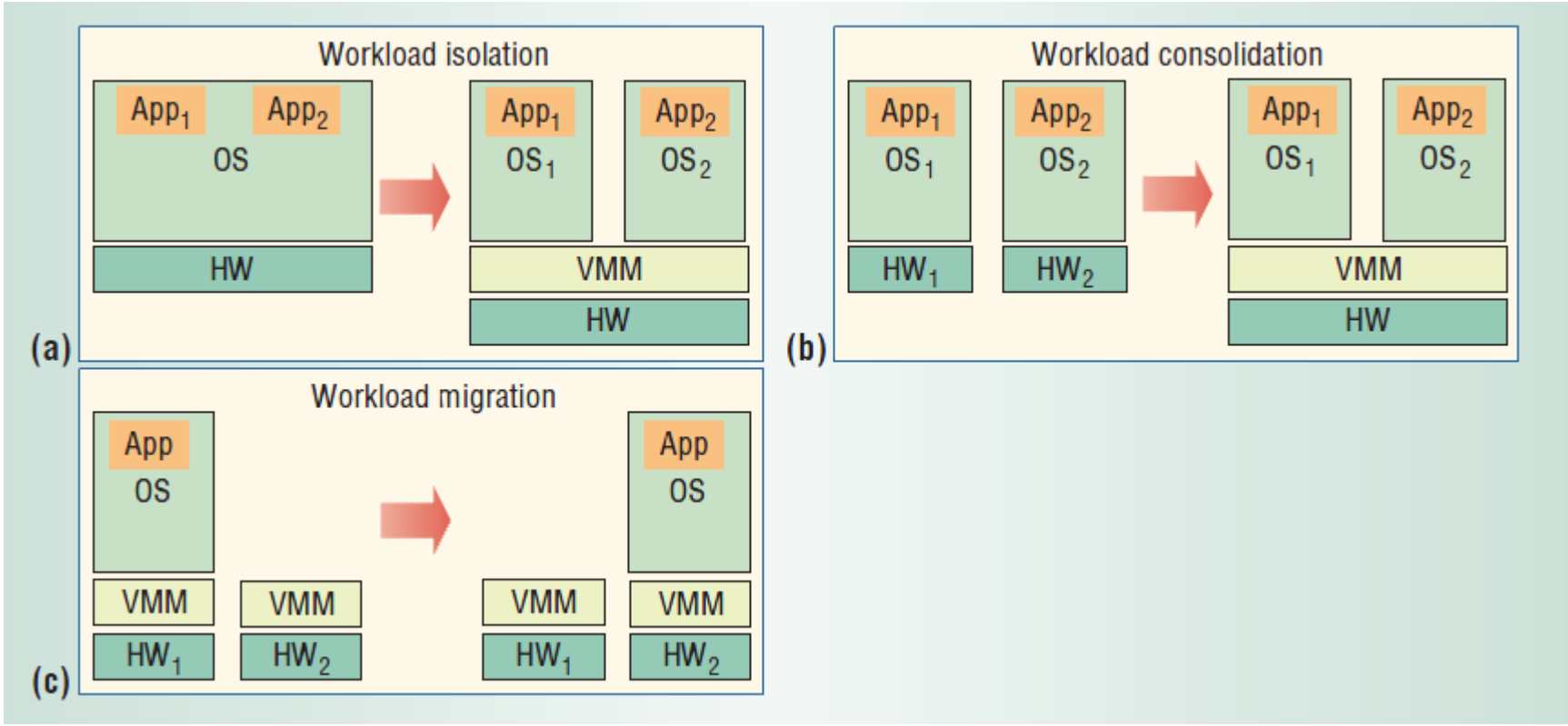
Virtual Machine Monitor

- What's Virtual Machine Monitor (VMM) ?
  - **VMM** or **Hypervisor** is the software layer providing the virtualization.

- System architecture :



為何要虛擬化?


Workload isolation
App1
App2
OS
HW
App1
OS1
App2
OS2
VMM
HW
(a)
Workload consolidation
App1
OS1
App2
OS2
HW1
HW2
App1
OS1
App2
OS2
VMM
HW
(b)
Workload migration
App
OS
VMM
HW1
VMM
HW2
App
OS
VMM
HW1
VMM
HW2
(c)

虛擬化的起源: IBM System/360

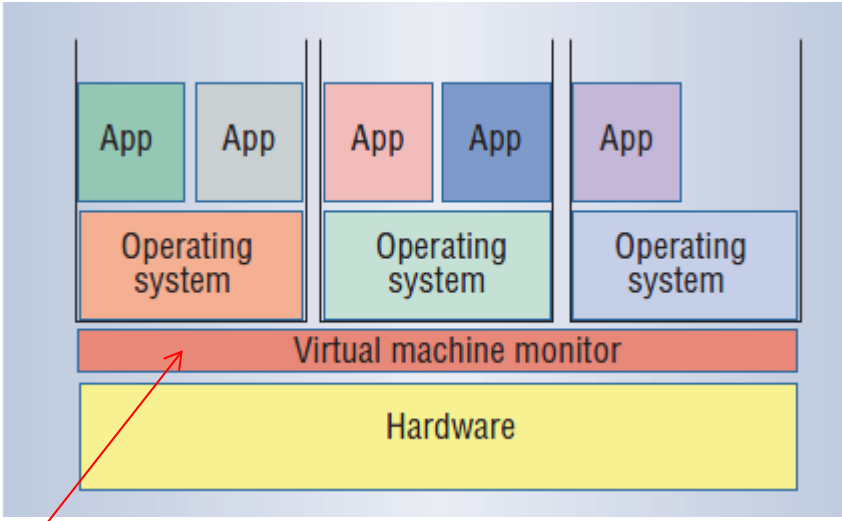
- IBM公司史上最大的豪賭
  - (當時) 人類史上最複雜的軟體系統
  - 開發過程徵召60,000員工、建立五座廠區
  - 1964/4/7公開後，IBM從此在大型主機奠定獨大地位

UNIVAC Honeywell RCA IBM CONTROL DATA NCR Burroughs

Snow White and The Seven Dwarfs

虛擬化的起源: IBM System/360

- 配合CP/CMS作業系統，成為史上第一個可虛擬化(Virtualization)的電腦
- CP (Control Program)
  - 相當於VMM
- CMS (Cambridge Monitor System)
  - 可在System/360上「同時」跑多種相容的作業系統
  - 也允許使用者自行創造作業系統



概念: 提供一個(虛擬的)共同硬體介面

VM系統的分類

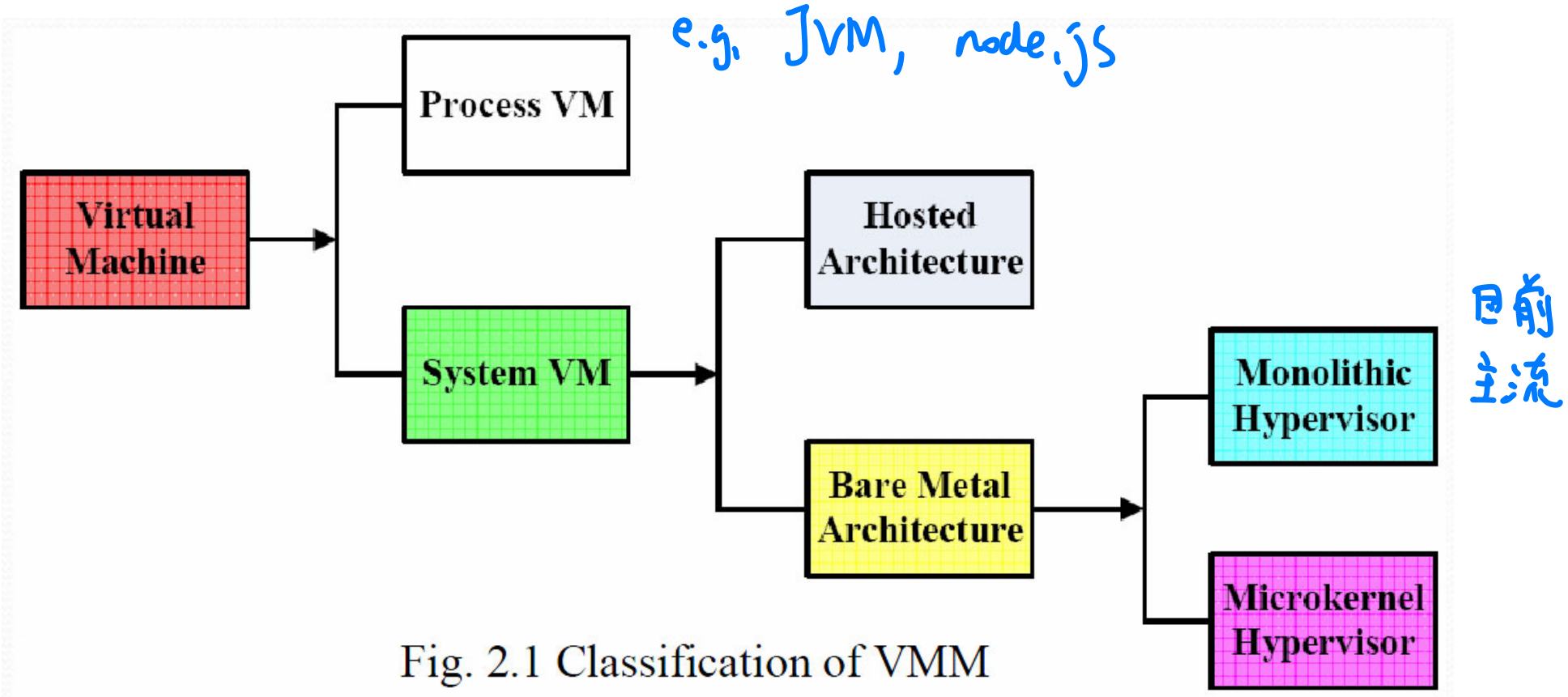
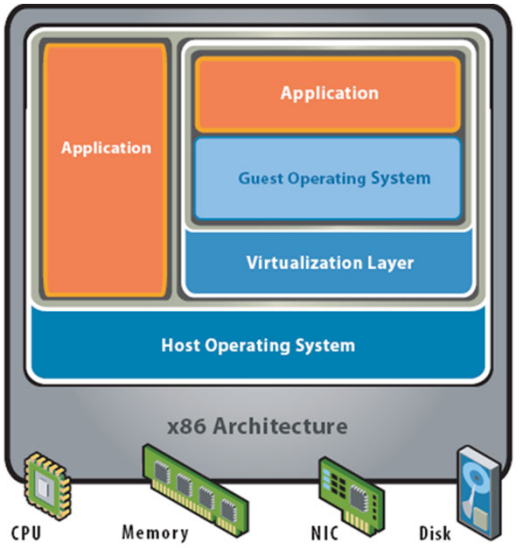


Fig. 2.1 Classification of VMM

System VM Virtualization Types

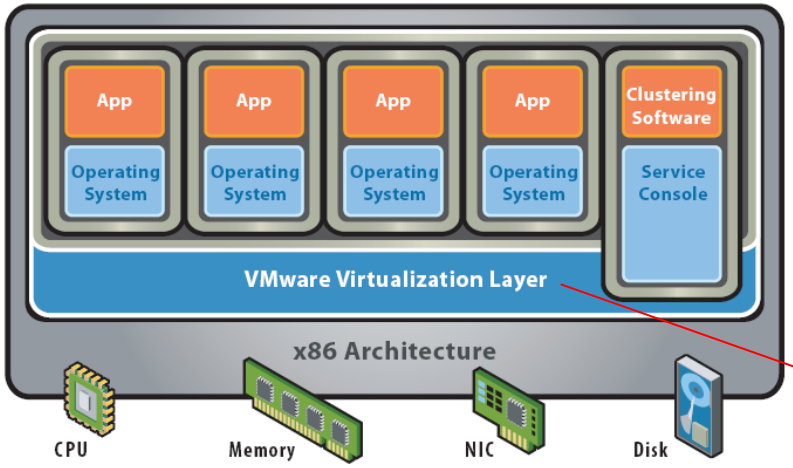
- Hosted
  - 硬體已安裝了主作業系統，虛擬層被當做「應用程式」被安裝在主作業系統上。
  - 虛擬層必需取得主作業系統所給予硬體資源，才能再分配給虛擬層上的寄居作業系統
  - Ex: VirtualBox



Hosted Architecture

System VM Virtualization Types

- Bare metal
  - 在硬體之上先建一個虛擬層(類似小的作業系統)，在虛擬層上再建作業系統，虛擬層完全控制硬體和資源分配，並直接分配給虛擬層上的作業系統



效能較高，但Virtualization Layer要implements所有driver!

Bare-Metal (Hypervisor) Architecture

VM系統的分類

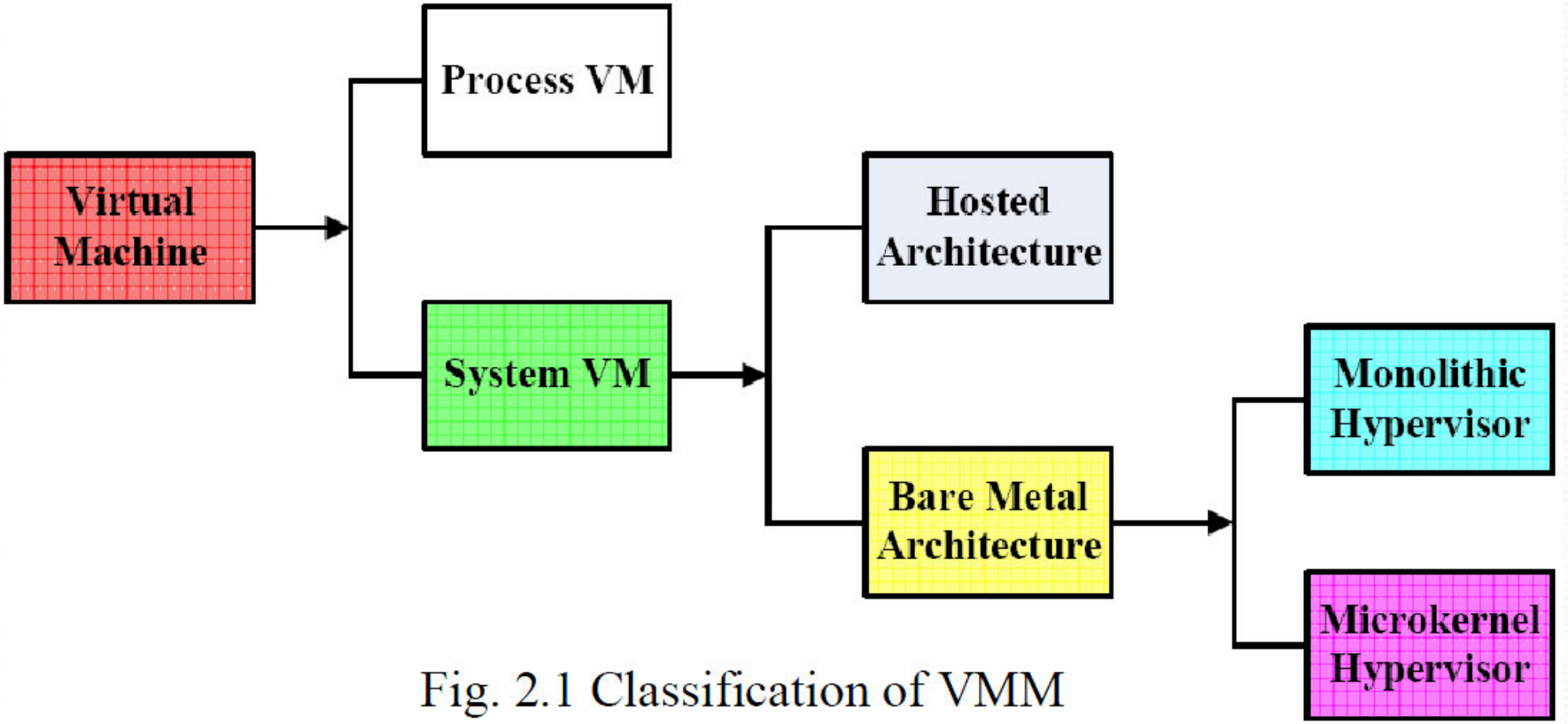
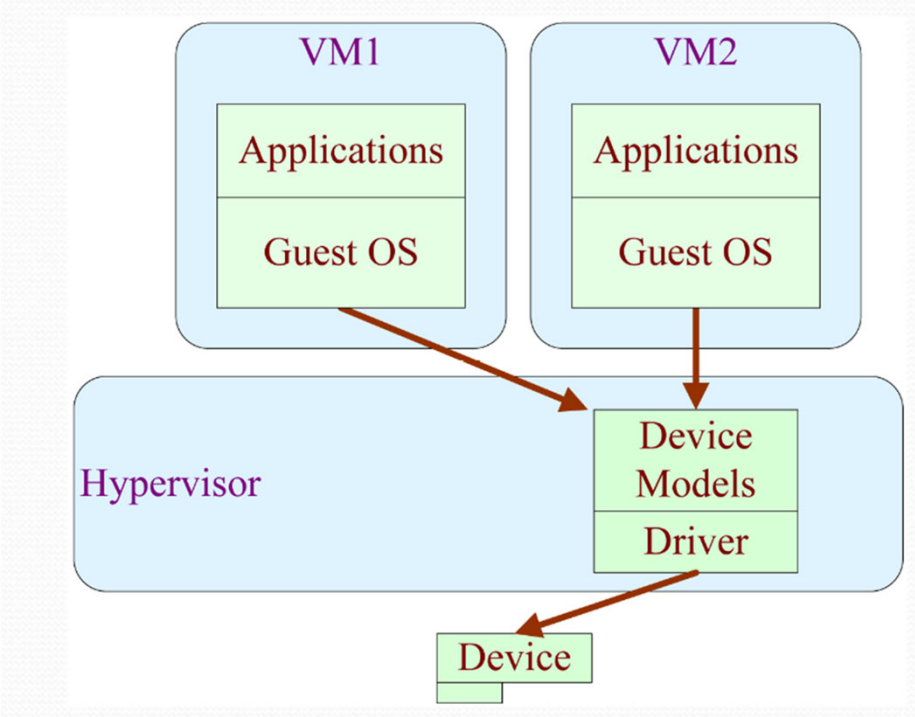


Fig. 2.1 Classification of VMM

Monolithic Hypervisor

- VMM可以看作為了虛擬化而設計出來的一個完整OS，它掌控並管理所有硬體資源
- VMM尚需負責VM的建立與管理

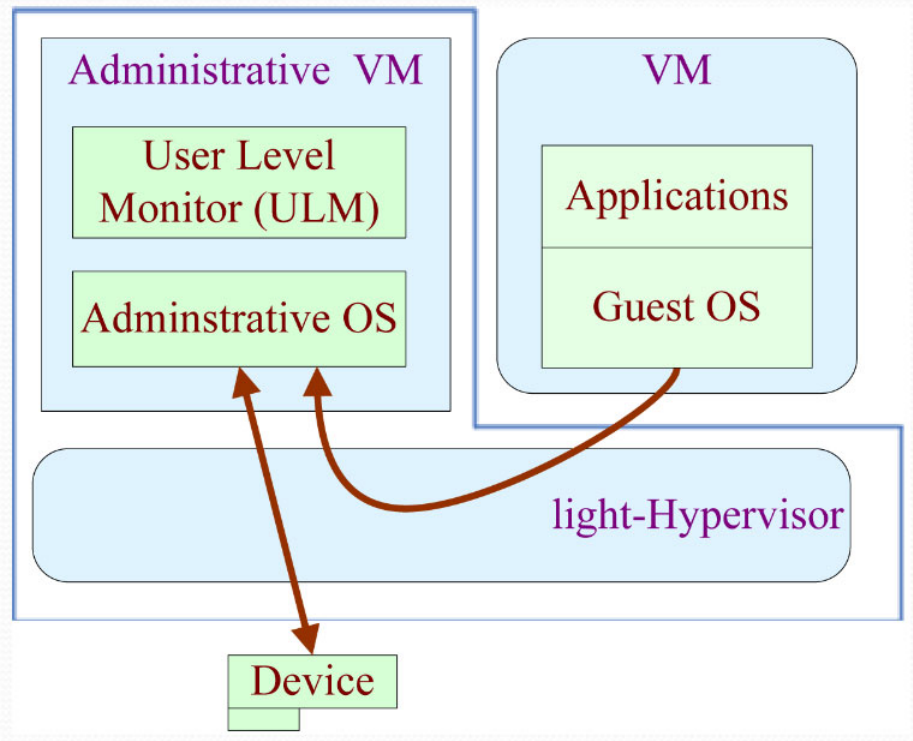


Monolithic Hypervisor

- 優點：因VMM同時具有硬體資源的管理功能和虛擬化功能，故效能較高
- 缺點：VMM開發商需提供所有IO設備驅動程式
- 產品：採用該結構的VMM有VMWare ESX Server、Wind River Hypervisor、KVM（後期）
- 對象：企業層級虛擬化、大型伺服器虛擬化

Microkernel Hypervisor

- VMM是一個輕量型Hypervisor，讓出大部分I/O設備的控制權，給Administrative VM中的 Administrative OS來控制
- VMM只負責CPU和Memory的虛擬化，I/O設備的虛擬化由VMM和Administrative OS共同完成



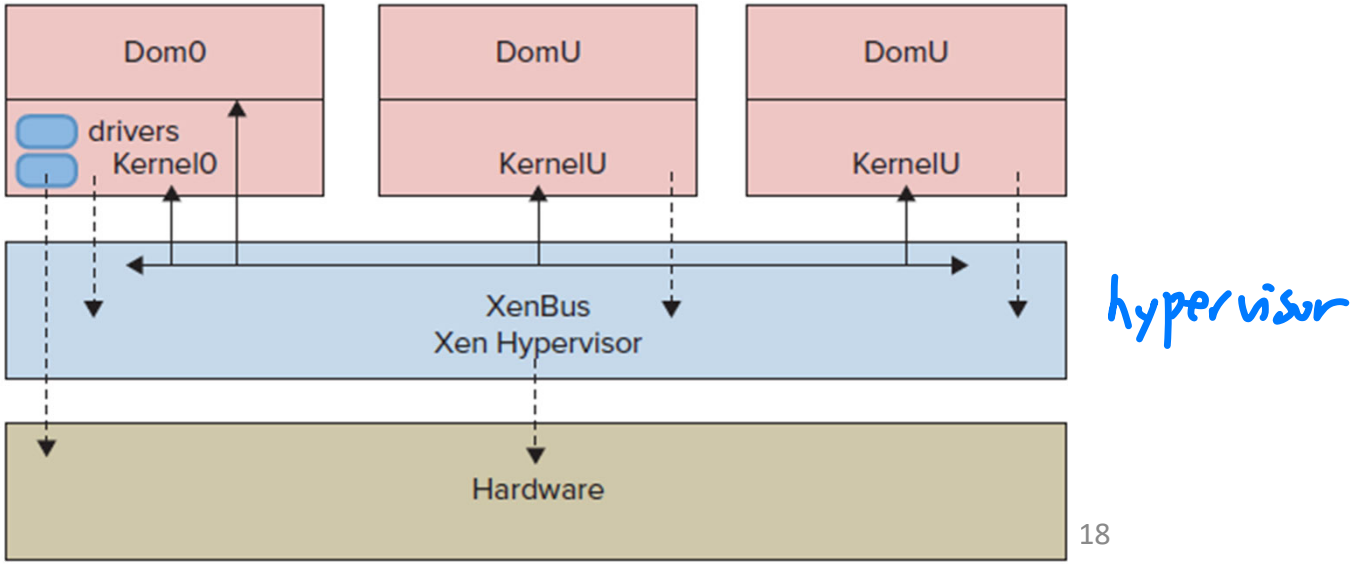
Microkernel Hypervisor

- 優點：可利用現有OS的I/O設備驅動程式，避免在VMM上開發I/O設備驅動程式
- 缺點：低效能
  - Admin OS運行於VMM上，當需要OS提供服務時，VMM需要將工作委派到Admin OS，產生時間浪費
  - 對象：研發或小型辦公室虛擬化、一般伺服器或PC虛擬化

Case: Xen

- 2002: OSS project by University of Cambridge
- 2013: Become Linux foundation project
- Microkernel hypervisor architecture
  - Dom0有效能與availability議題

Dom: Domain



Case: VMware

- 1998：VMware成立; 2001：推出產品
  - 市佔率: 70%
- 架構
  - ESX: Bare Metal; GSX: Hosted (已終止)
  - ESX為Linux-based Admin+Hypervisor架構
    - 面臨資源耗用過多與安全性問題
  - ESXi將兩者整合
    - Monolithic架構
    - 2011後只保留ESXi產品線

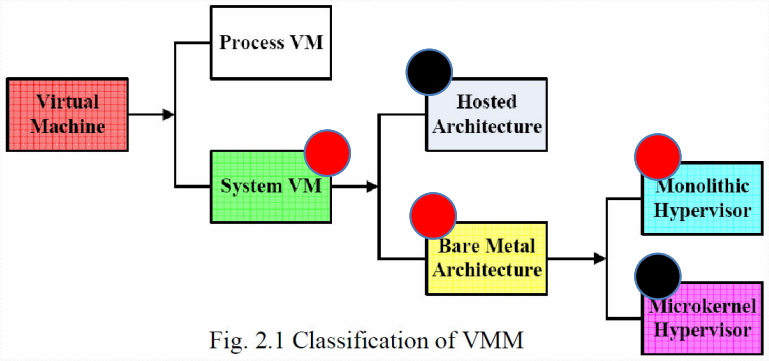
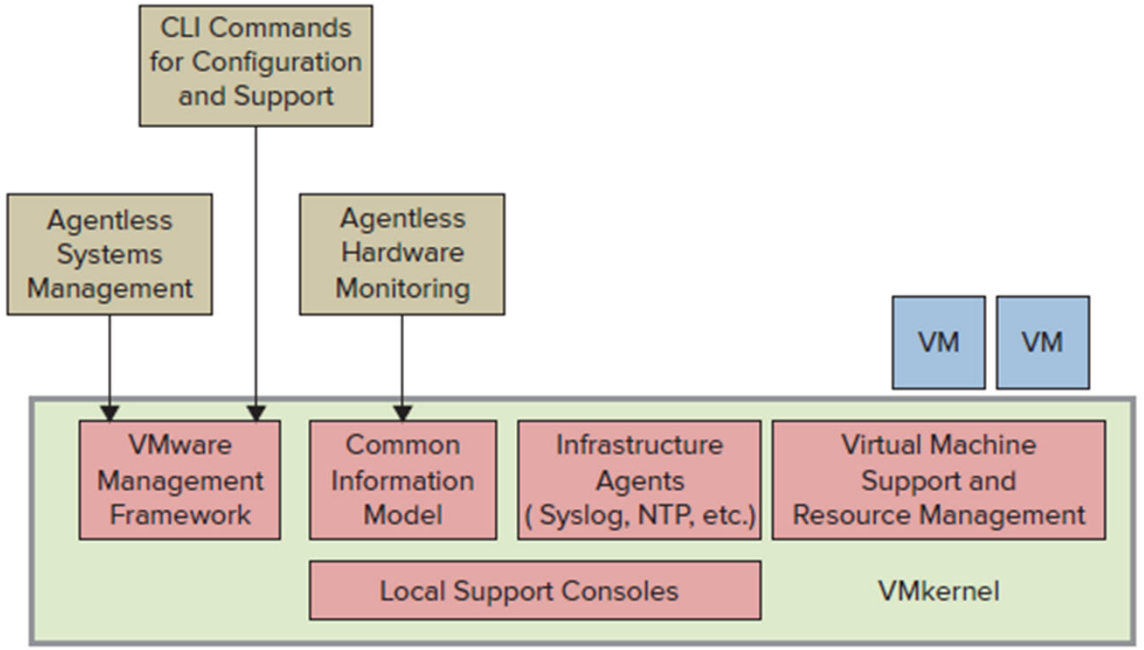


Fig. 2.1 Classification of VMM

VMWare ESXi Hypervisor



重要服務全部整合進單體的大kernel中

虛擬化的優勢

- 充分利用現有資源的程度
- 透過縮減實體基礎架構和提升伺服器/管理員比率以降低運轉成本
- 提高硬體和應用程式的可用性，進而提高事務連貫性
- 呈現運營方面靈活性
- 提升桌面的可管理性和安全性

虛擬化的顧慮

- 以下兩點是虛擬化技術未來繼續發展過程中必需要解決的問題：
  - 硬體使用效率: 如何在多虛擬機模式下，充分發揮硬體的極緻能力
  - 安全及可靠性: 安全性是最重要的，然而可靠性也是同等重要

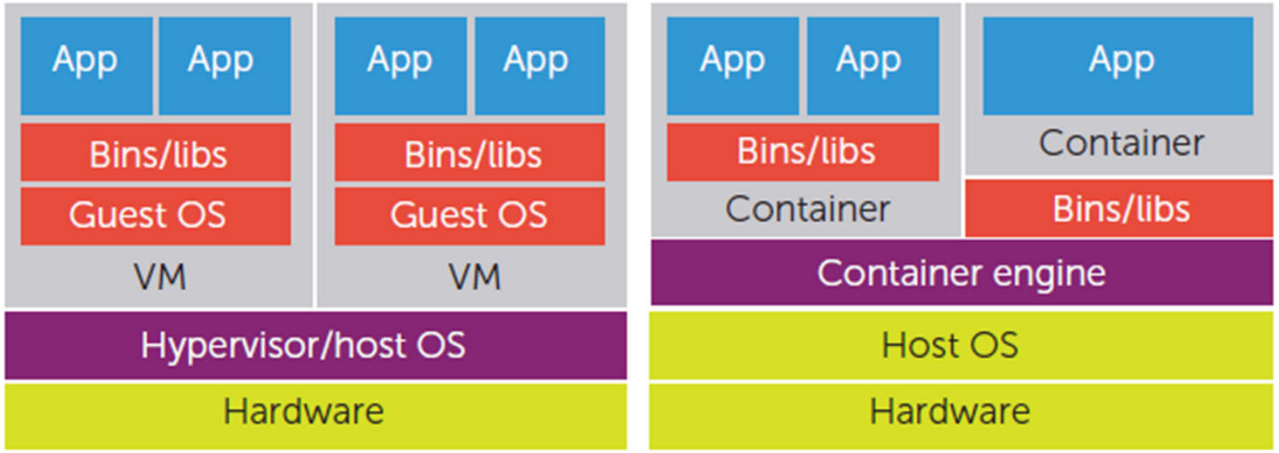
OS-Level Virtualization (Container)

- 定義
  - A set of processes running on the same OS instance
  - It appears that they have the entire OS to themselves
    - Resources are isolated from each other; Ex: PID/檔案系統/網路…
- 目的
  - 安全性
    - 雖然是同OS instance，彼此無法看到，要相互存取必須明確設定
  - 經濟性
    - 資源有效分享、利用
  - 模組化
    - Low or no dependencies with each other; low interference
    - 下一頁有更詳細說明

VM vs. Container

VM讓人以為獨佔整個電腦，其實是和別人共享電腦
Container讓人以為獨佔整個OS，其實是和別人共享OS

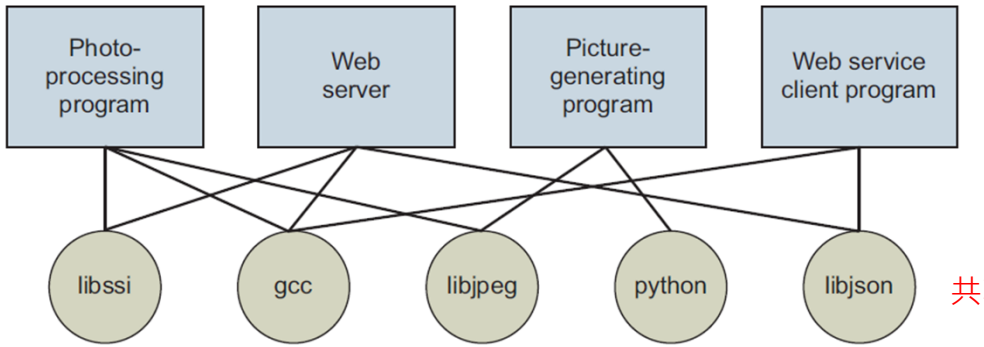
- VM
  - Full guest OS images are required for each VM
- Container 注意: 這裡的Container和Java EE中的container 意義不同
  - Holds packaged, self-contained, ready-to-deploy parts of applications
  - All containers share the same Host OS



Why Container?

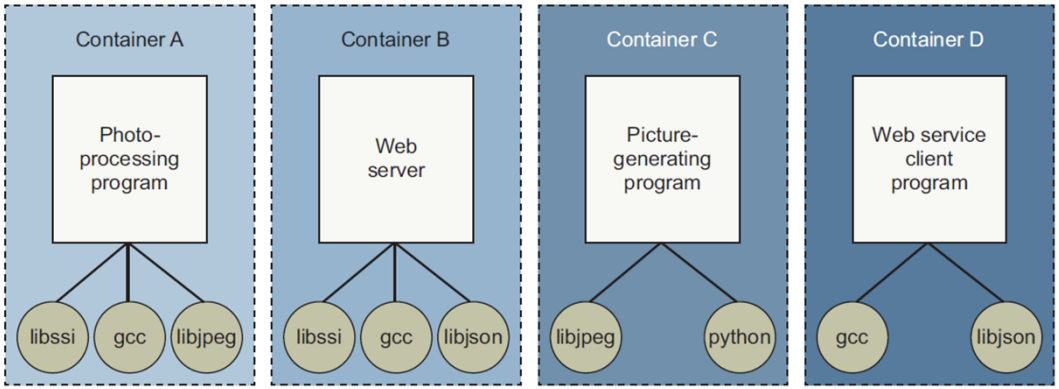
- 自我包含的軟體部署
  - 易於隨時安裝、移除，不會互相影響

Without container



共享OS/ File systems

With container



(以為) 各自有獨立的OS/ File systems

Portability Consideration

- Container instances should be OS-dependent
  - 本質上是直接在同一個OS上跑
- The core reason that containers are portable
  - Containers were assumed to run on Linux
  - Linux has great binary portability among variants (POSIX)

Container: 歷史觀點

- Using containers has been a best practice for a long time
  - UNIX chroot: 1979 Unix ver. 7 (Bell Labs)
    - change the root directory of a running process
    - enhance security by limiting the scope of fs
  - Jail: 1998 FreeBSD
    - Used for web hosting
    - Extends chroot; independent fs, users, and network;
  - Zones/Container: 2004 Solaris
    - “virtualized operating system”
  - LXC: 2008 Linux
    - Began as a project to utilize the ns and cgroups feature
  - Docker: 2013
  - Kubernetes: 2017

Bernstein, D. (2014). Containers and cloud: From LXC to docker to kubernetes. IEEE Cloud Computing, 1(3), 81-84.

Core technologies of LXC

- Linux namespace
  - PID namespace—Process identifiers and capabilities
  - UTS namespace—Host and domain name
  - MNT namespace—File system access and structure
  - IPC namespace—Process communication over shared memory
  - NET namespace—Network access and structure
  - USR namespace—User names and identifiers
- cgroups: resource isolation
- chroot: controls the location of the file system root

LXC Implementation

- Namespace
  - Limits the view scope of various OS resources
- Cgroups
  - Limit and isolate the resource usage (CPU, memory, disk I/O, network, etc.) of collections of processes

```
struct uts_namespace {
        struct new_utsname name;
        struct user_namespace *user_ns;
        struct ucounts *ucounts;
        struct ns_common ns;
} __randomize_layout;
extern struct uts_namespace init_uts_ns;
```

```
struct nsproxy {
        refcount_t count;
        struct uts_namespace *uts_ns;
        struct ipc_namespace *ipc_ns;
        struct mnt_namespace *mnt_ns;
        struct pid_namespace *pid_ns_for_children;
        struct net           *net_ns;
        struct time_namespace *time_ns;
        struct time_namespace *time_ns_for_children;
        struct cgroup_namespace *cgroup_ns;
};
```

```
struct task_struct {
        /* Namespaces: */
        struct nsproxy                  *nsproxy;
```

Why Docker?

- Problem

  LXC=Linux Container

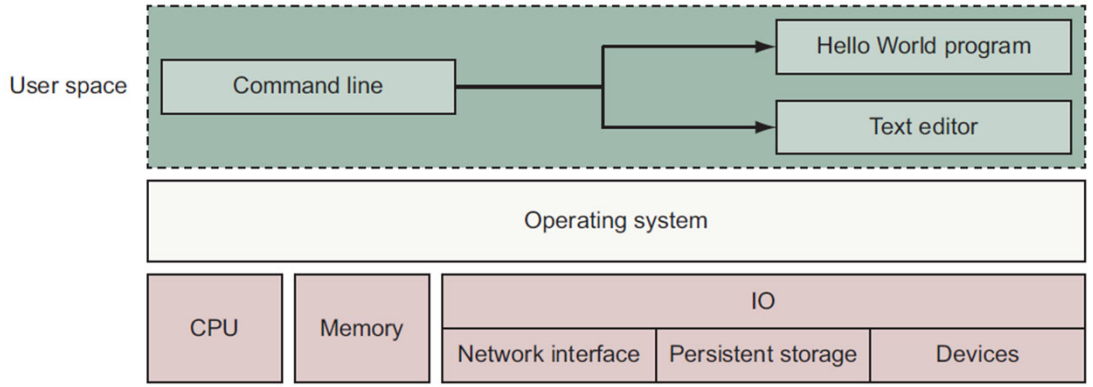
  – Manually building LXC can be challenging and error prone
- Solution
  – Provide a systematic way to automate package and deployment of LXC
  – Docker uses existing LXC engines to provide consistent containers built according to best practices
- Benefits
  – Using LXC is easier and at lower cost
  – Provide a consistent way of using LXC

What Docker Does?

- Docker在LXC之外做了些什麼
  - Provides kernel and application-level API
    - Takes care of Isolation
    - PID, File system, process tree, user space, CPU, network …
  - Image: the shipping container instance
    - Composed of a layered file system
      - Each action taken forms a new layer
    - Dockerfile: the script of constructing a new image
  - Ecosystem: DockerHub
    - Images becomes reusable and stackable

Container, Docker and OS

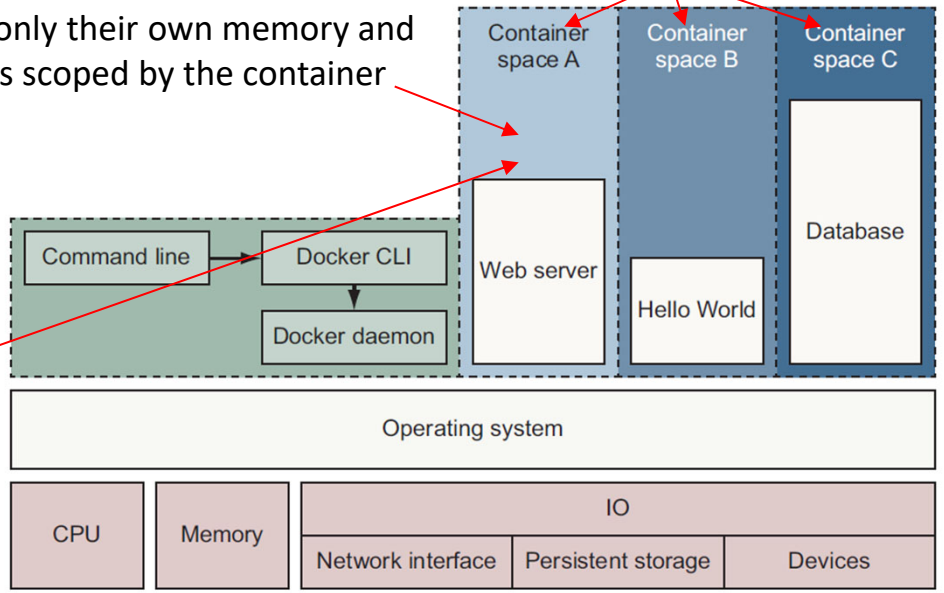
Without container



With container

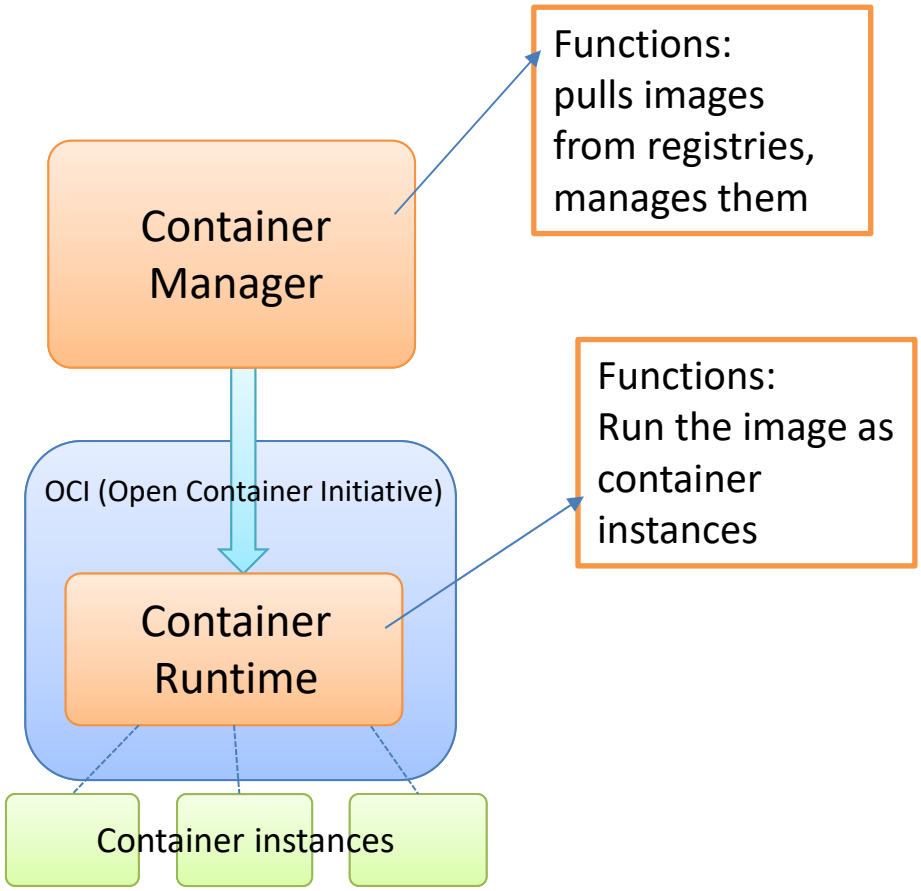
running in its own memory space

can access only their own memory and resources as scoped by the container

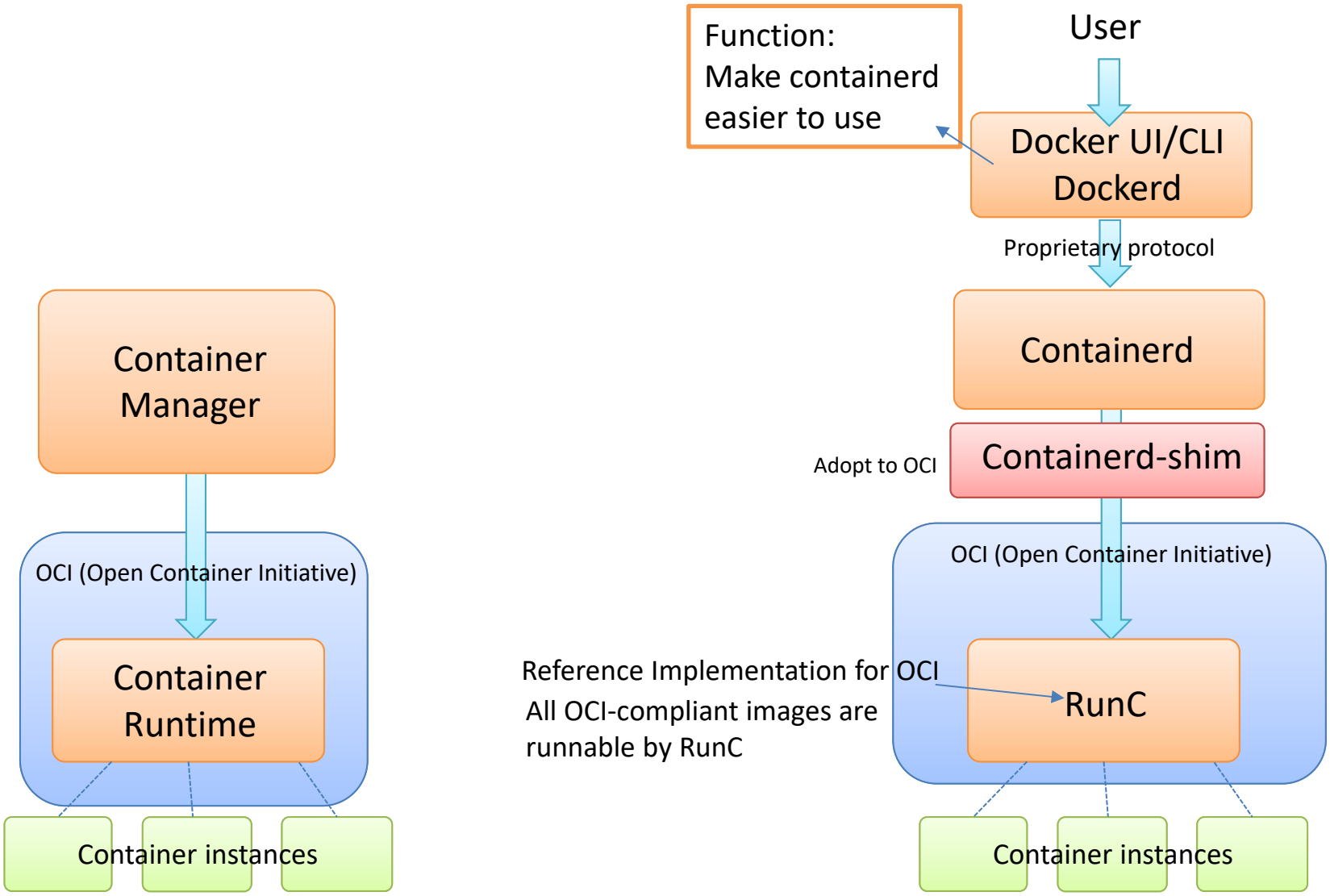


Each container is a child process of the Docker engine

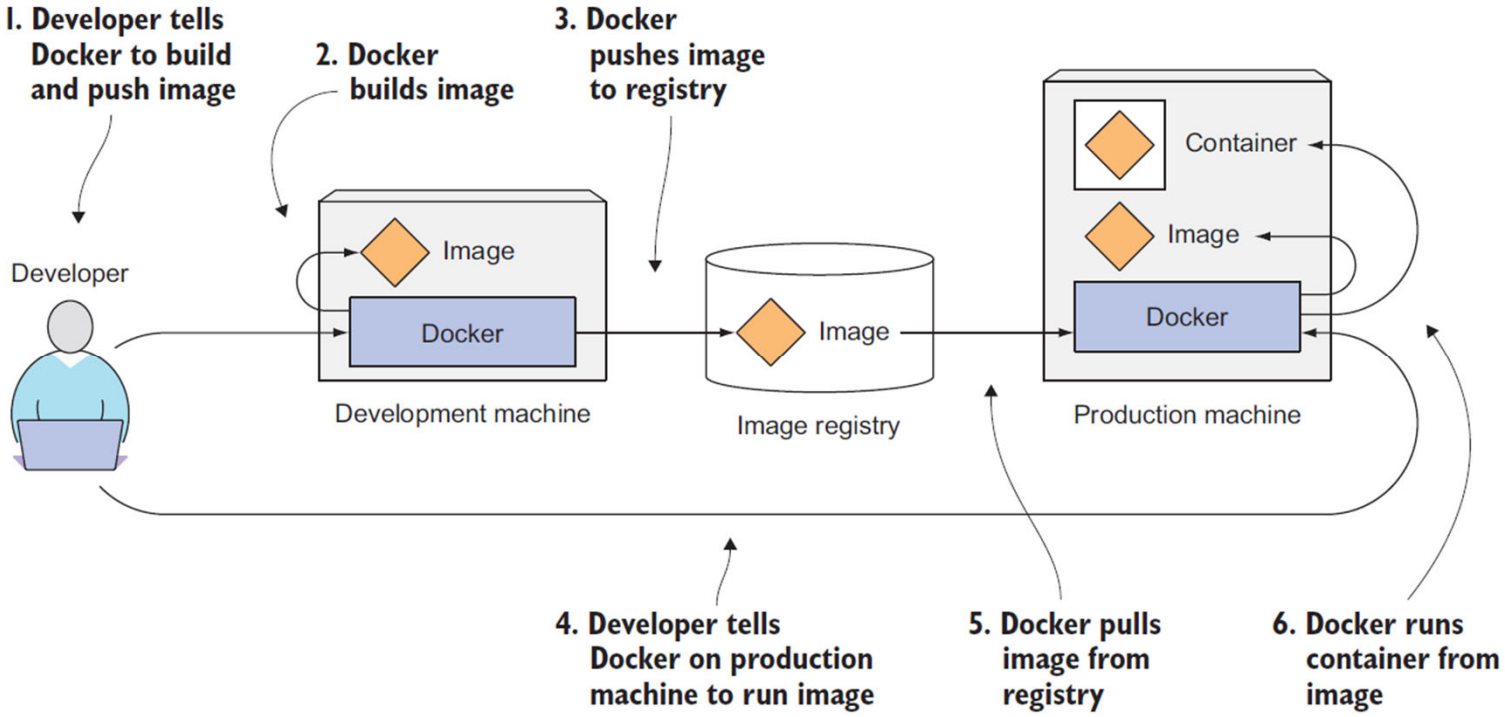
General Container Engine Structure



Docker

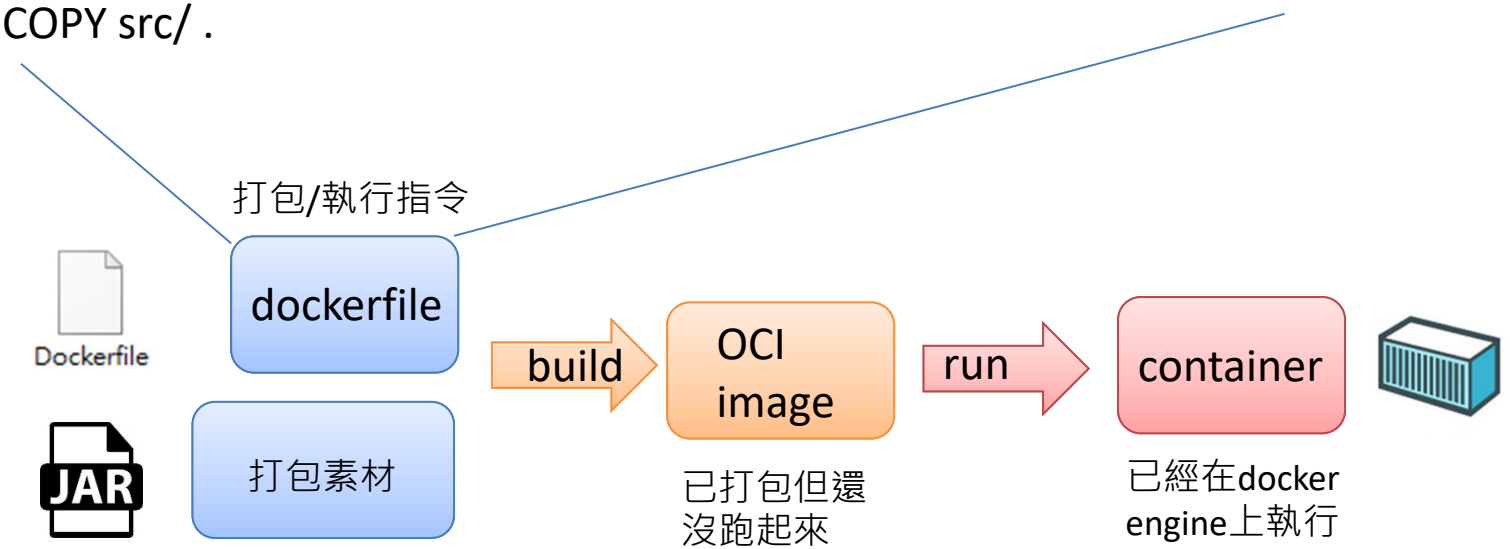

Function:
Make containerd
easier to use
User
Docker UI/CLI
Dockerd
Proprietary protocol
Containerd
Adopt to OCI
Containerd-shim
OCI (Open Container Initiative)
Reference Implementation for OCI
All OCI-compliant images are
runnable by RunC
RunC
Container instances
Container
Manager
OCI (Open Container Initiative)
Container
Runtime
Container instances

Container應用程式的開發與佈署

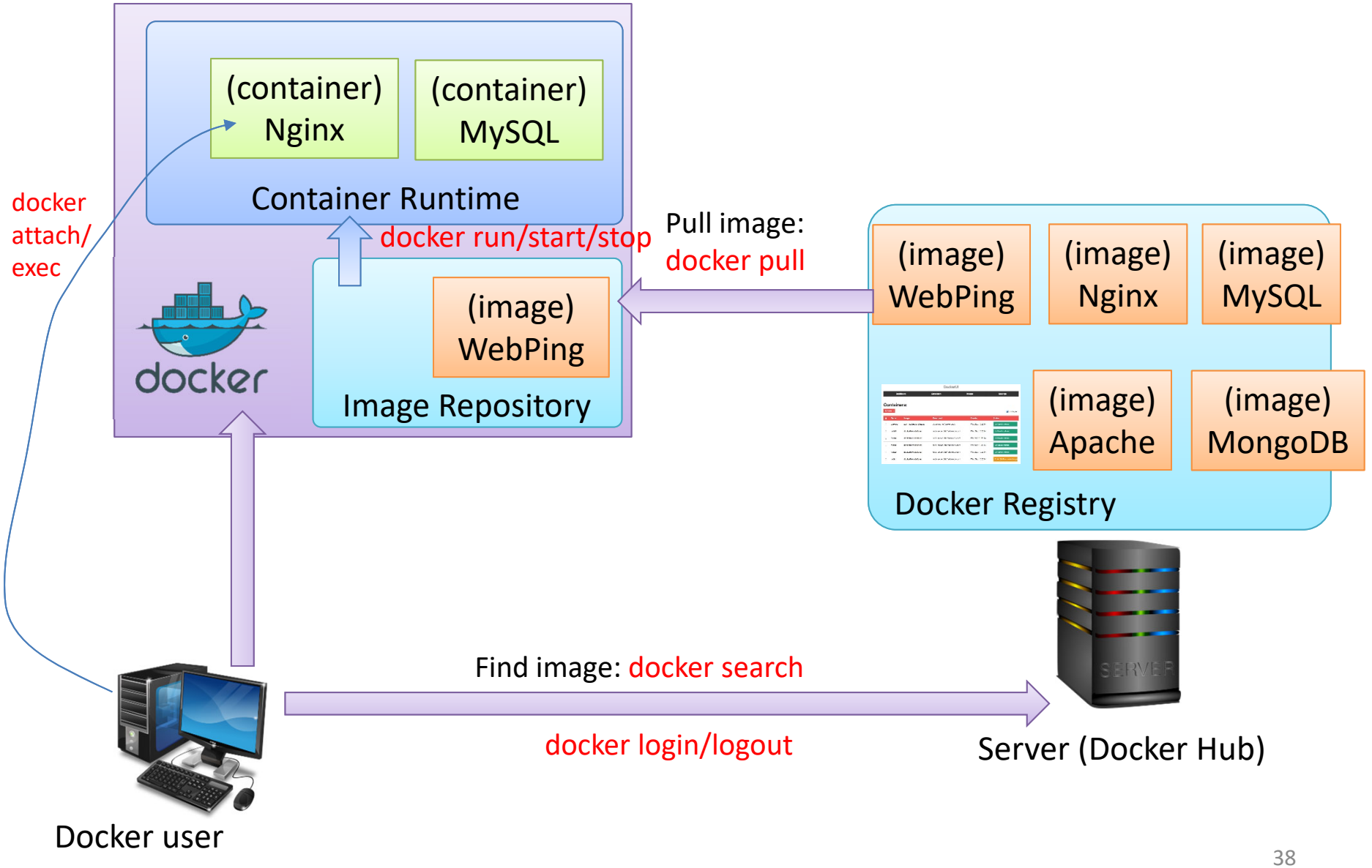


開發與佈署

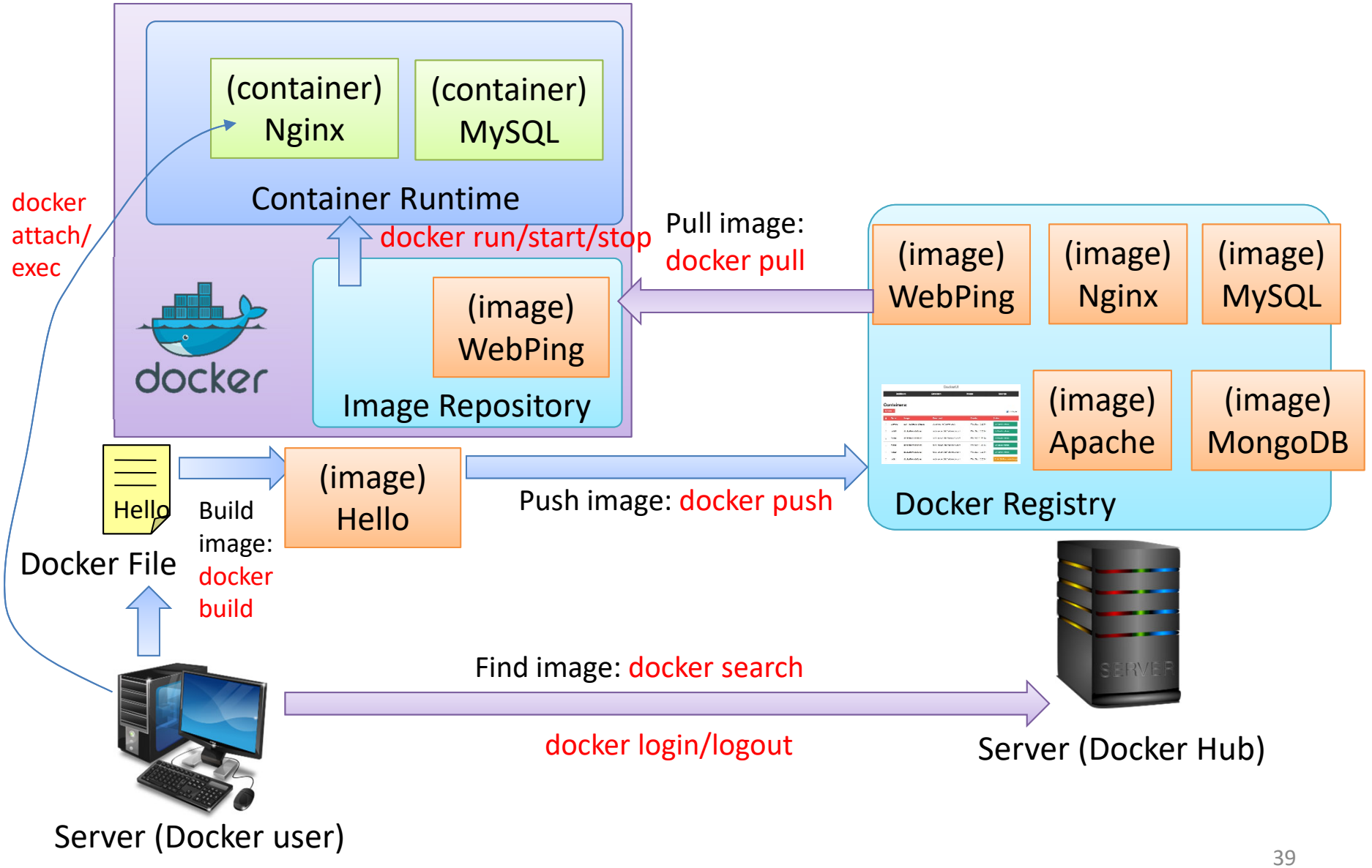
```
FROM node:14-alpine
EXPOSE 80
cmd ["node", "server.js"]
WORKDIR /app
COPY --from=builder /src/node_modules/ /app/node_modules/
COPY src/ .
```



Container開發與佈署 (details)



Container開發與佈署 (details)



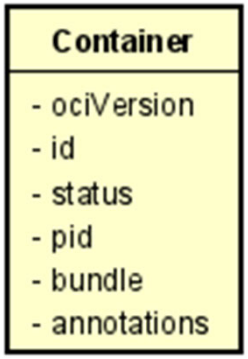
Open Container Initiative (OCI)

- Initiated by the Linux Foundation in 2015
- Runtime Specification
  - Configuration file format
  - Execution environment
  - Lifecycles
  - Standard operations (API)
- Image Specification
  - a manifest
  - an image index (optional)
  - a set of file system layers
  - a configuration

OCI Runtime Specification

- Container states
  - id
    - MUST be unique across all containers on this host
  - pid
    - ID of the container process
  - status
    - creating, created, running, stopped
  - annotations
    - 附加說明
    - Key-value pairs

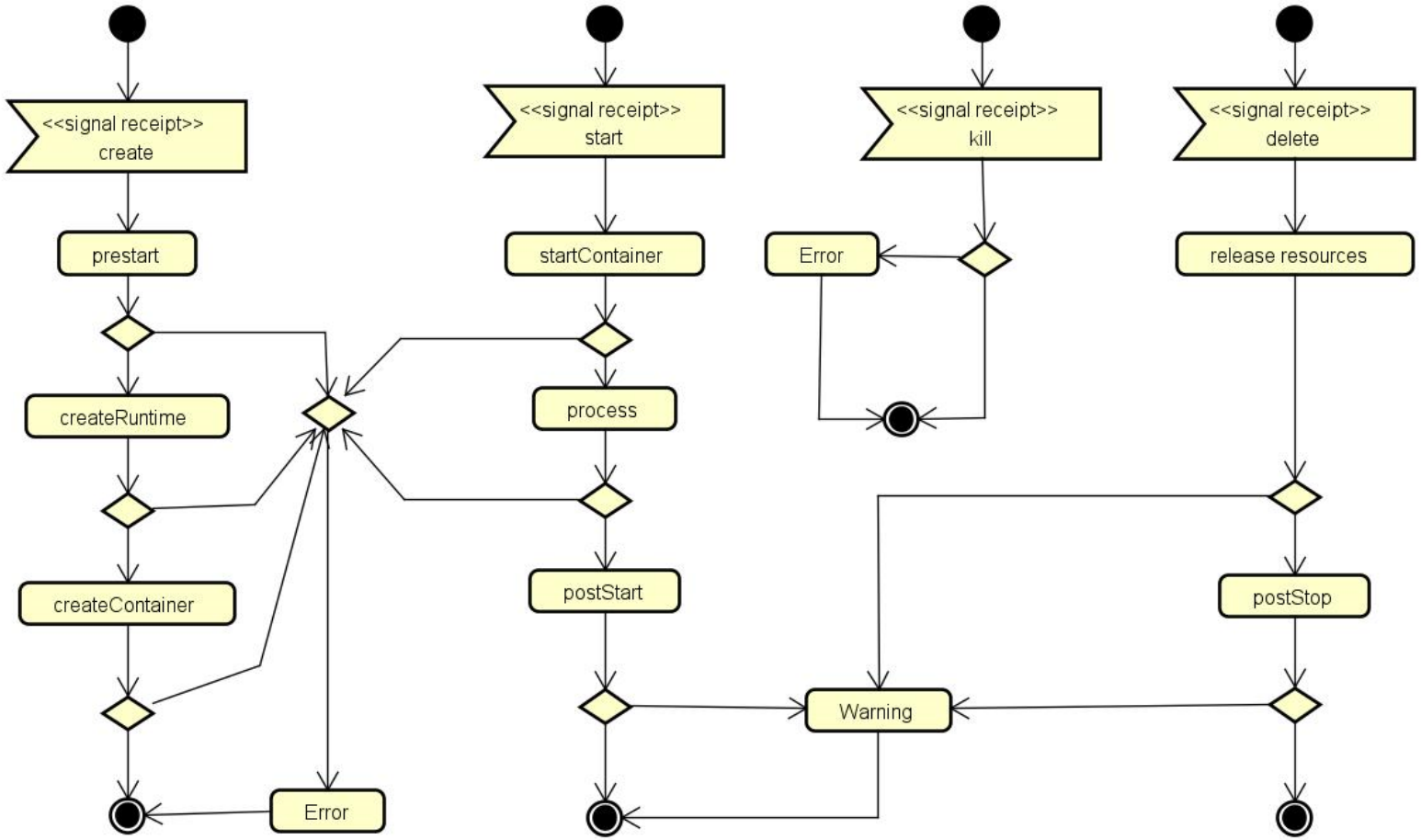
```
{
    "ociVersion": "0.2.0",
    "id": "oci-container1",
    "status": "running",
    "pid": 4422,
    "bundle": "/containers/redis",
    "annotations": {
        "myKey": "myValue"
    }
}
```



Container Format

- Defined as a file-system bundle
  - how a container, and its configuration data, are stored on a local file system
- Artifacts
  - config.json
    - configuration data of a container
  - Root file system of the container
    - Defined by root.path in config.json
    - 可在config中掛上hook: 指定特定生命週期時要執行什麼script

Container Lifecycle

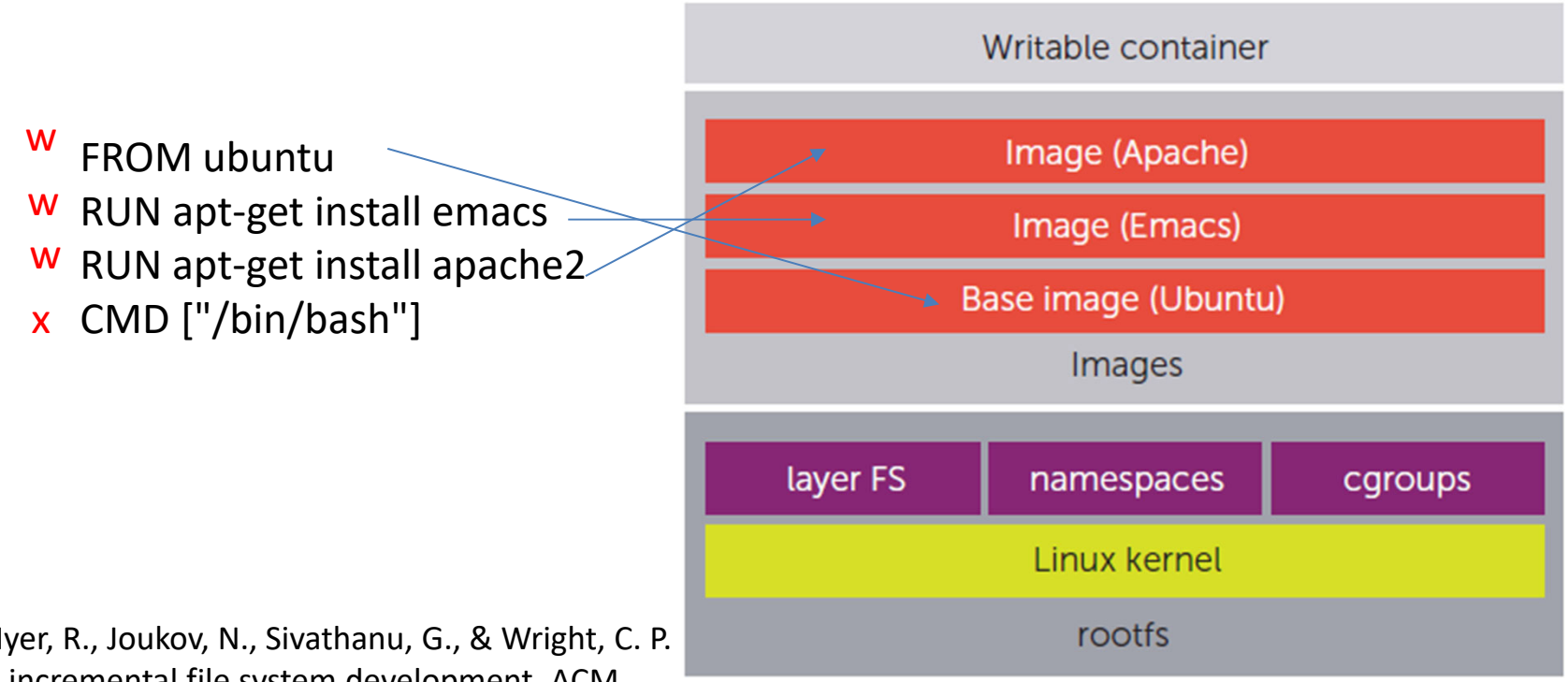


Union FS

- UnionFS was originally developed by Prof. Erez Zadok and his team at Stony Brook University

  https://unionfs.filesystems.org/

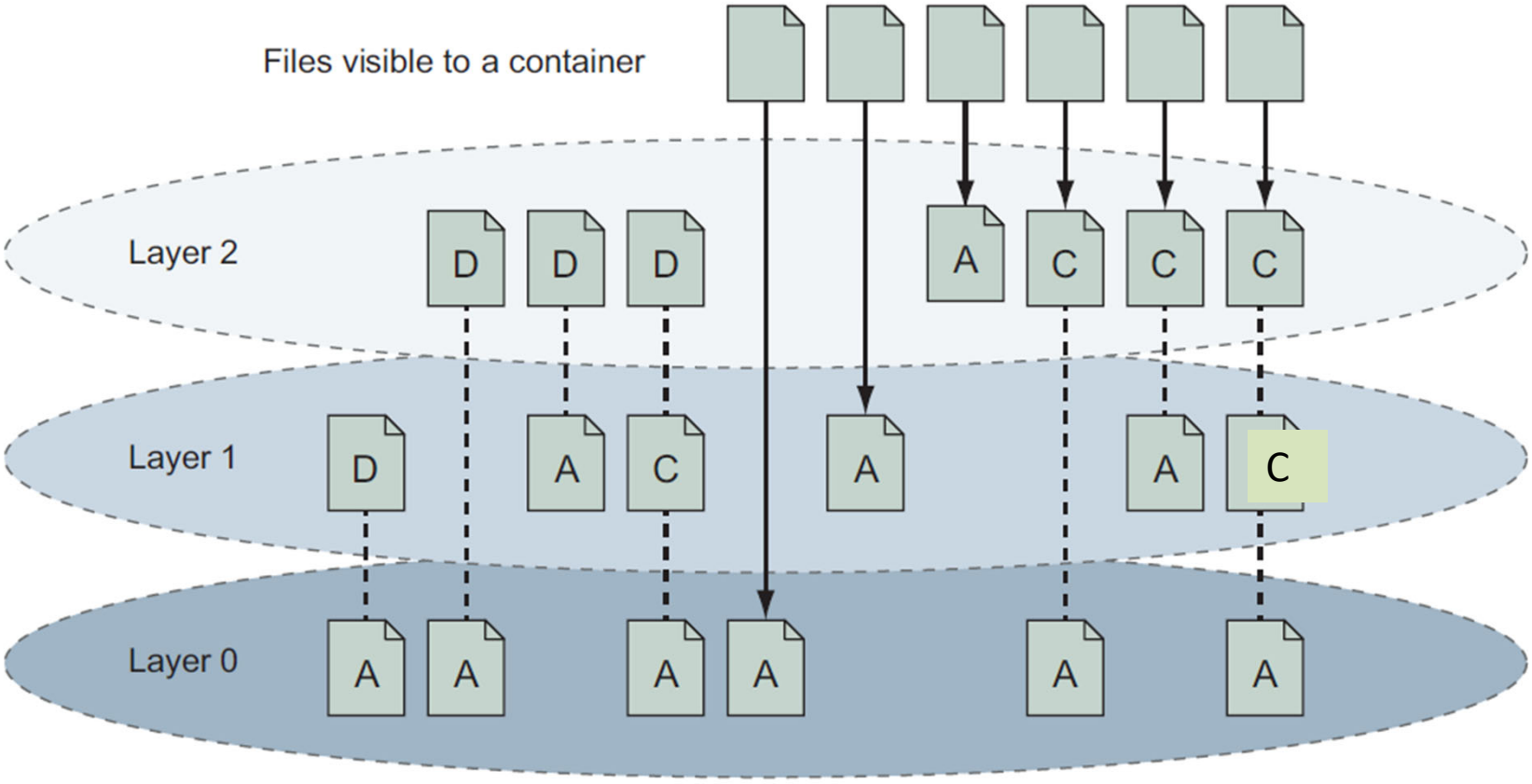
容器啟動後，其內應用程式對容器的所有改動，增刪，都只會發生在writable container這一層，不會動到原有的image



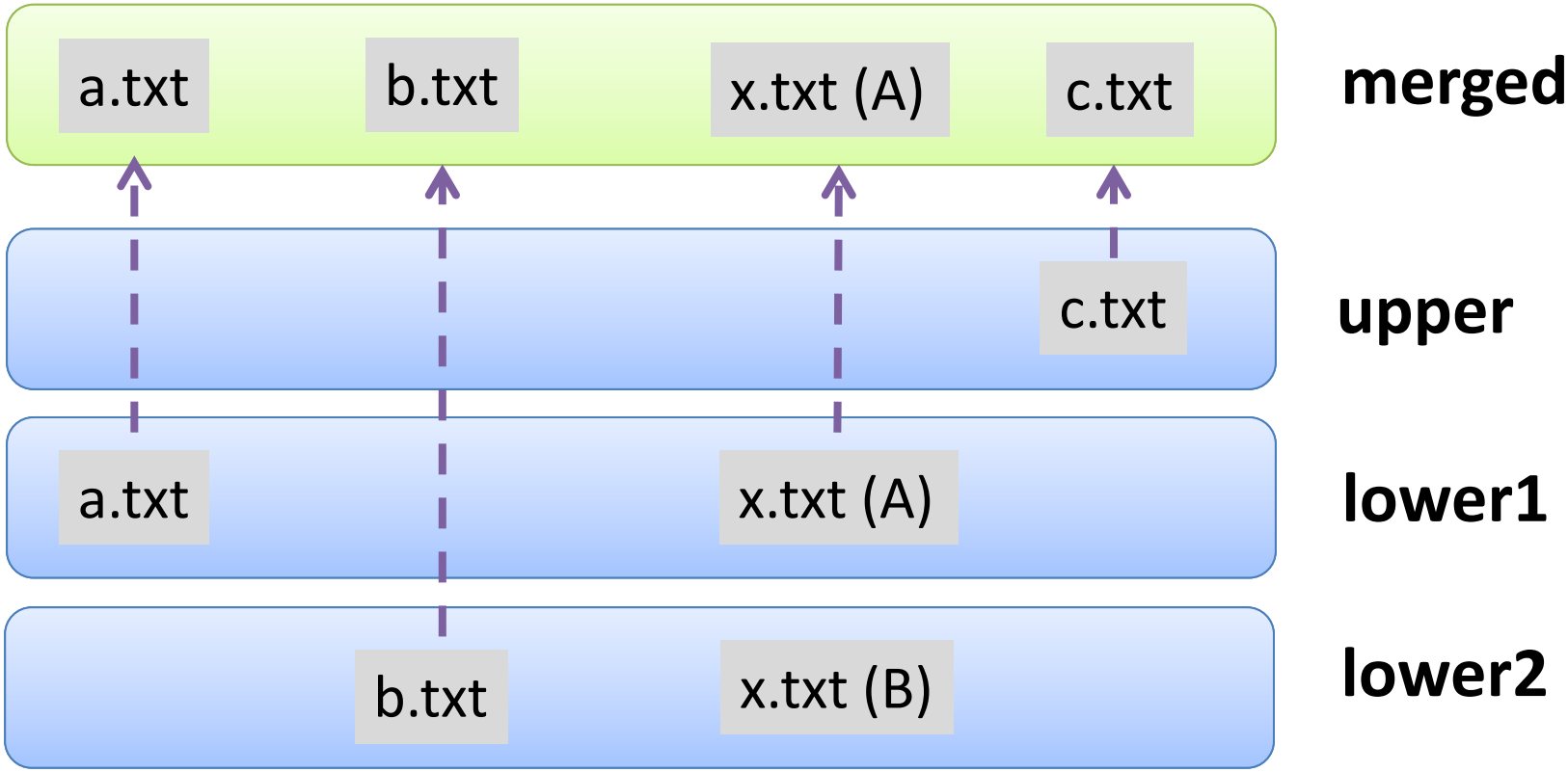
Zadok, E., Iyer, R., Joukov, N., Sivathanu, G., & Wright, C. P. (2006). On incremental file system development. ACM Transactions on Storage (TOS), 2(2), 161-196.

Union FS：增刪修改的套用

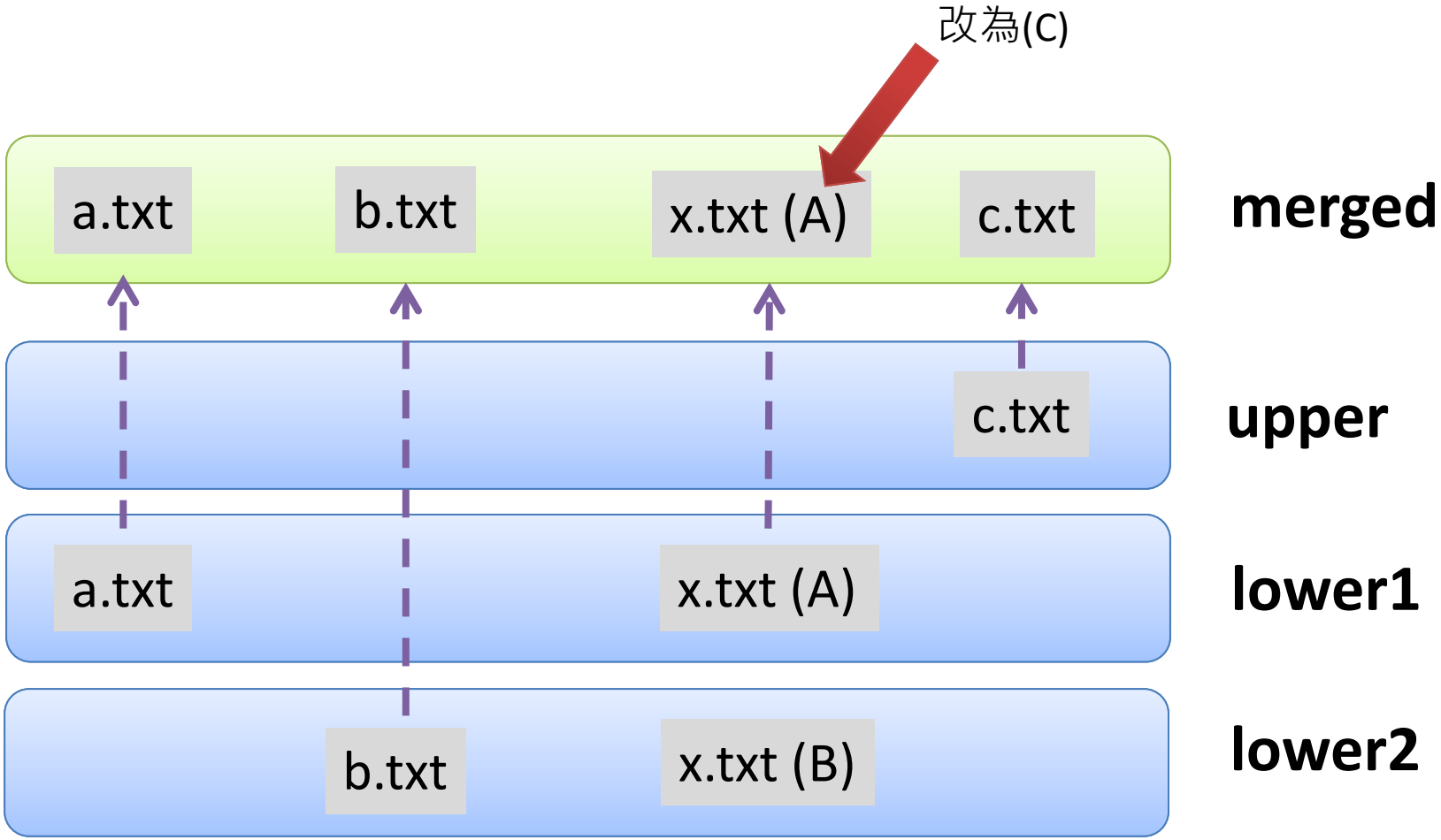
重要觀察點: 在不修改資料的前提下呈現出修改的效果



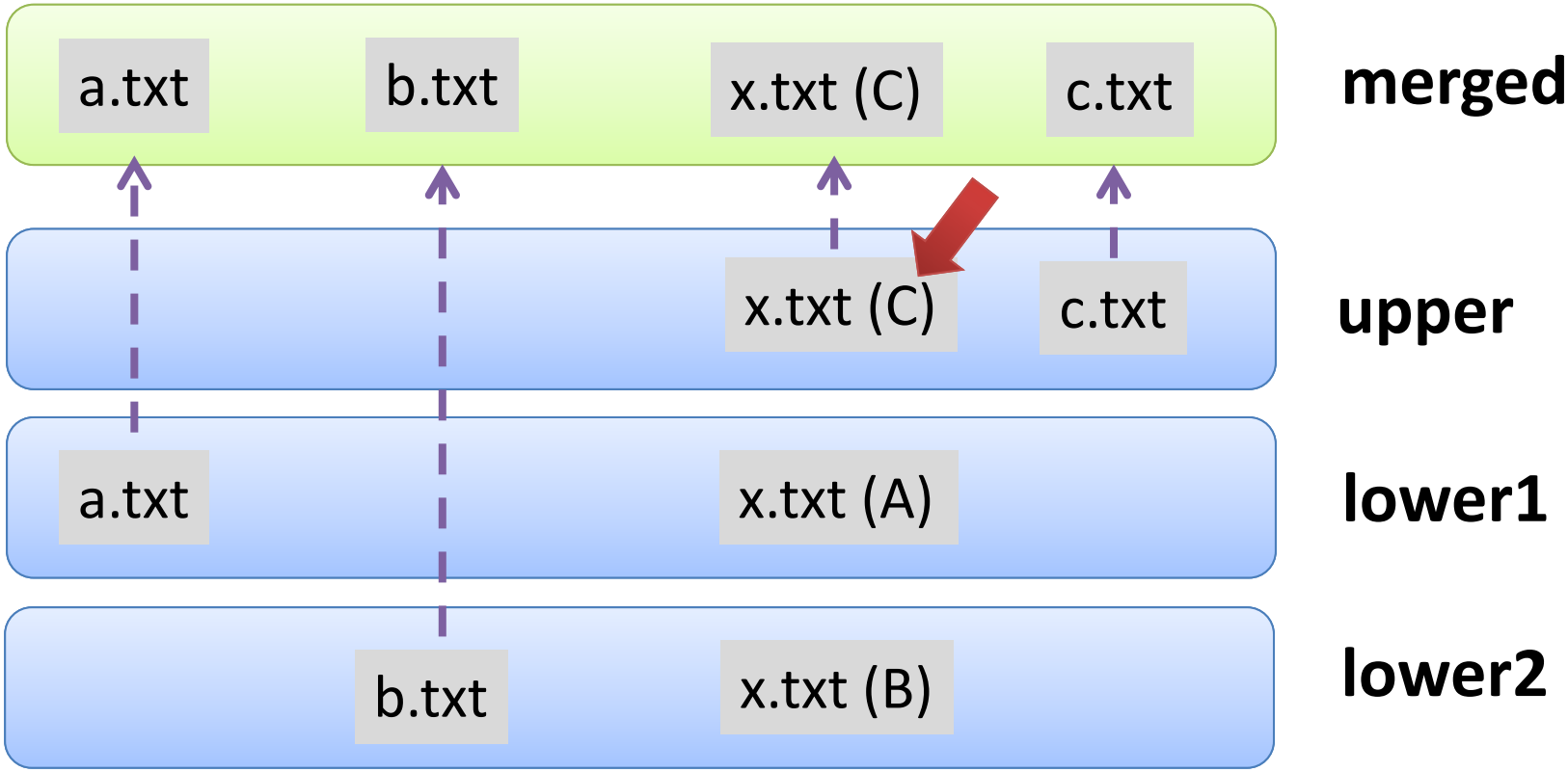
Demo


a.txt
b.txt
x.txt (A)
c.txt
merged
c.txt
upper
a.txt
x.txt (A)
lower1
b.txt
x.txt (B)
lower2

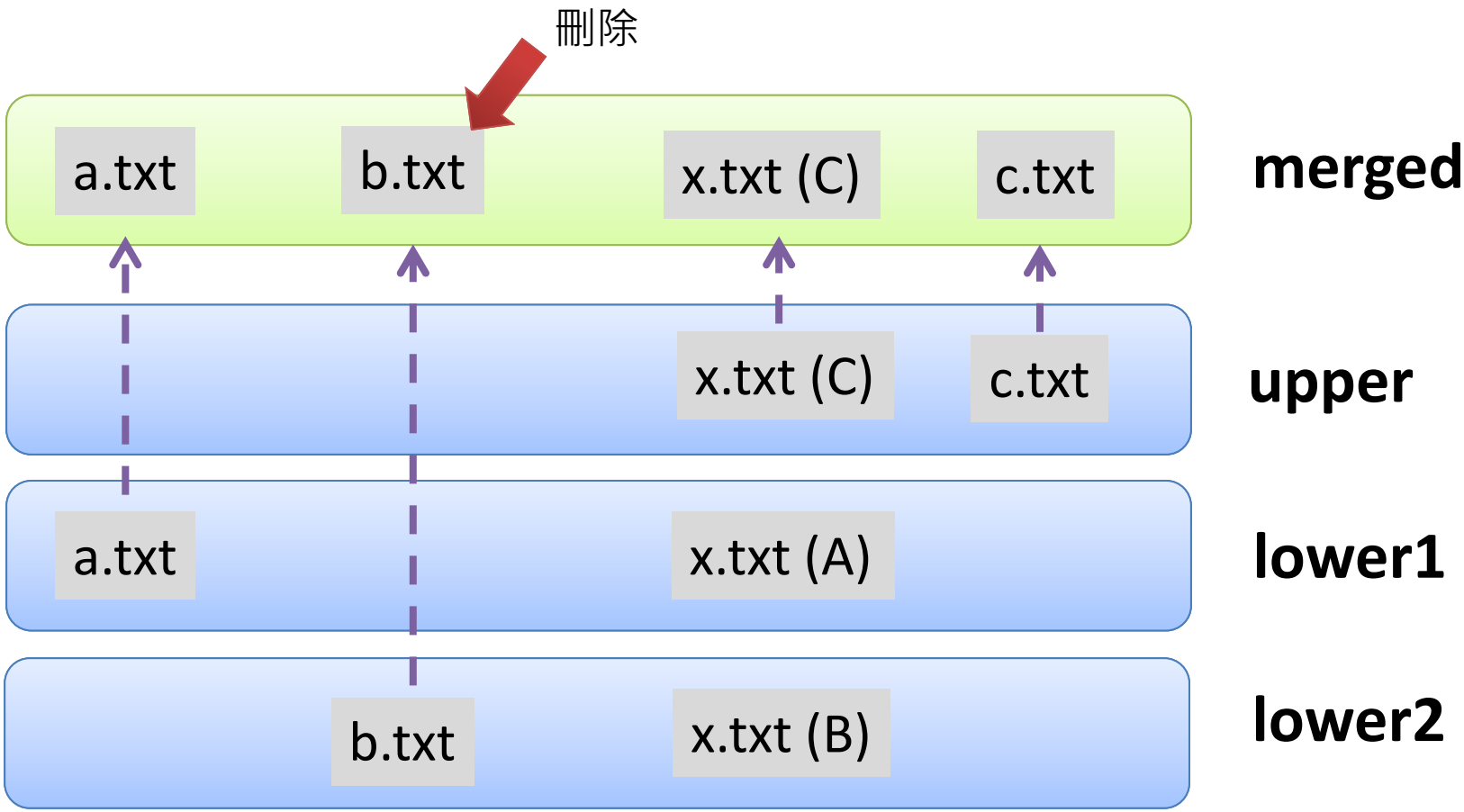
Demo: 如何記錄「修改歷程」



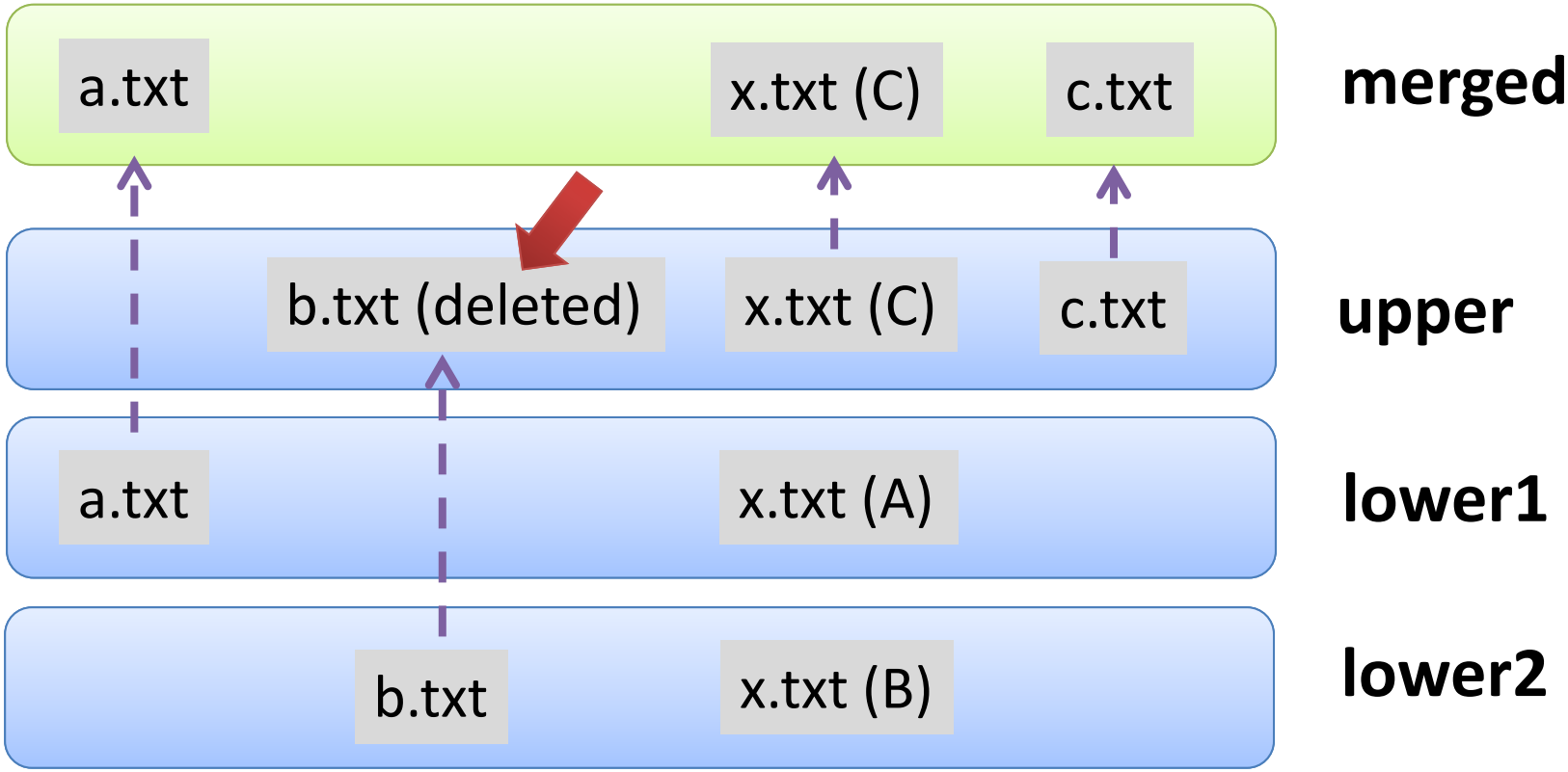
Demo: 如何記錄「修改歷程」



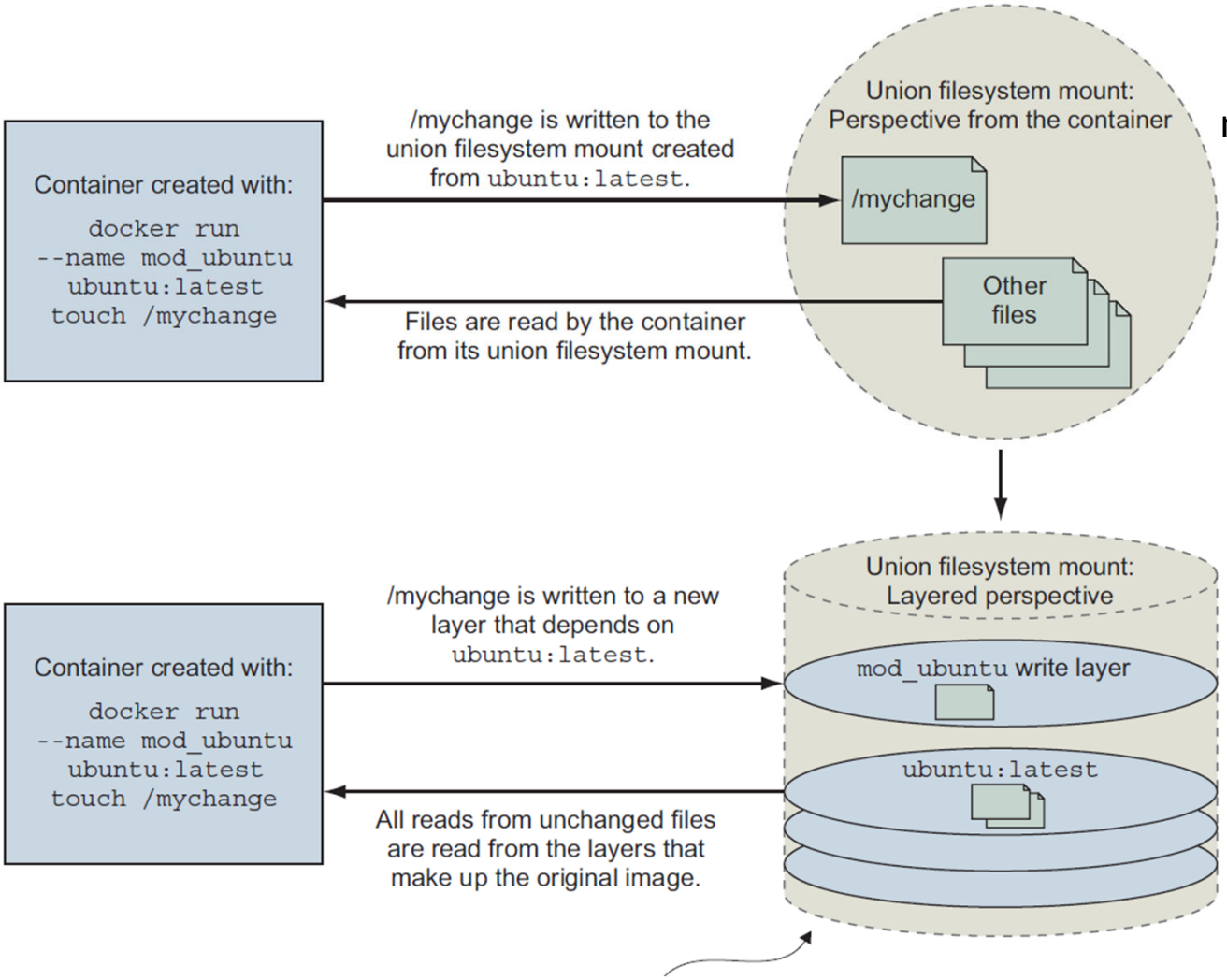
Demo: 如何記錄「刪除歷程」



Demo: 如何記錄「刪除歷程」



Union FS

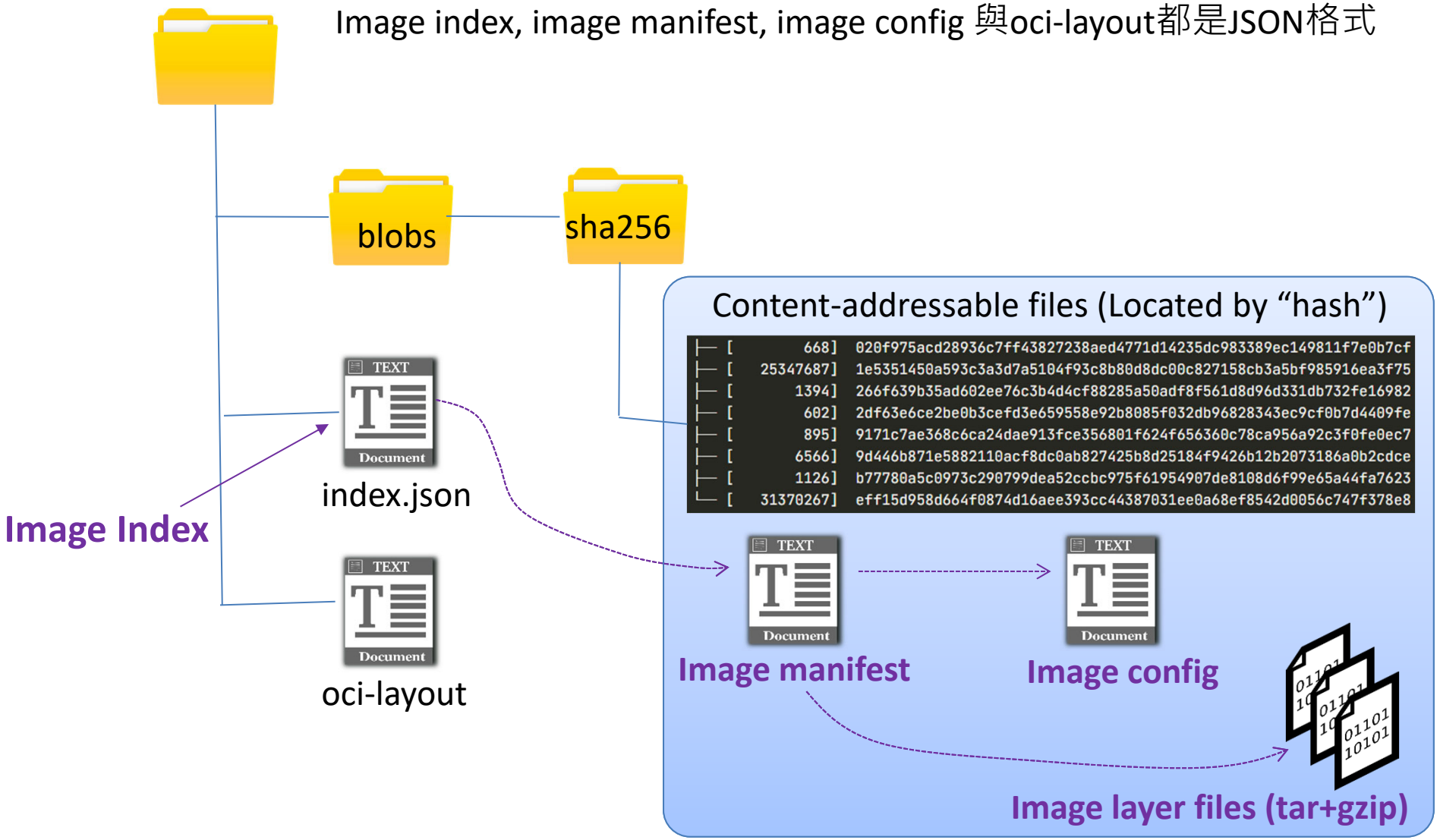


應用層觀點

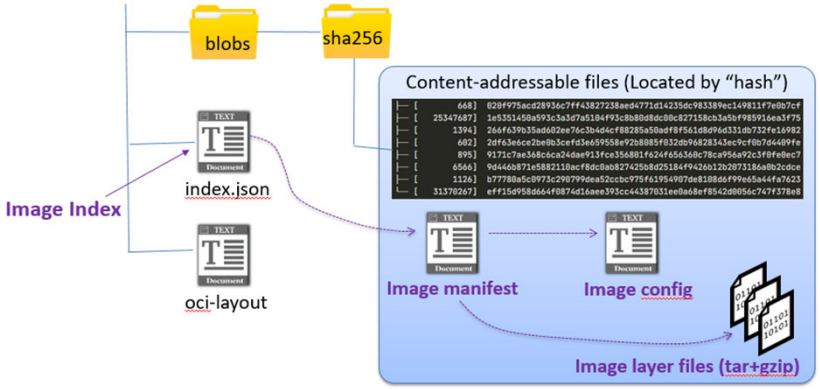
檔案層觀點

OCI Image Layout

Image index, image manifest, image config 與oci-layout都是JSON格式



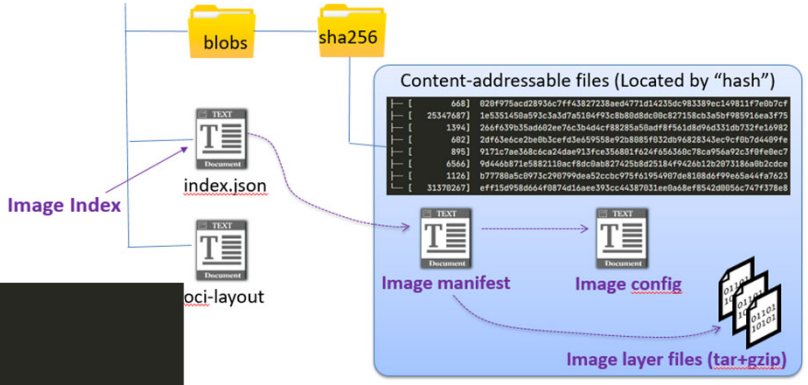
Demo



- 準備工具
  - tree、skopeo與jq
- 取得特定image
  - skopeo copy docker://photon oci:local_photon
- 查看目錄結構
  - cd local_photon
  - tree --du
  - 查看oci-layout
    - more oci-layout | jq
  - 從index.json找manifest
    - more index.json |jq

Demo

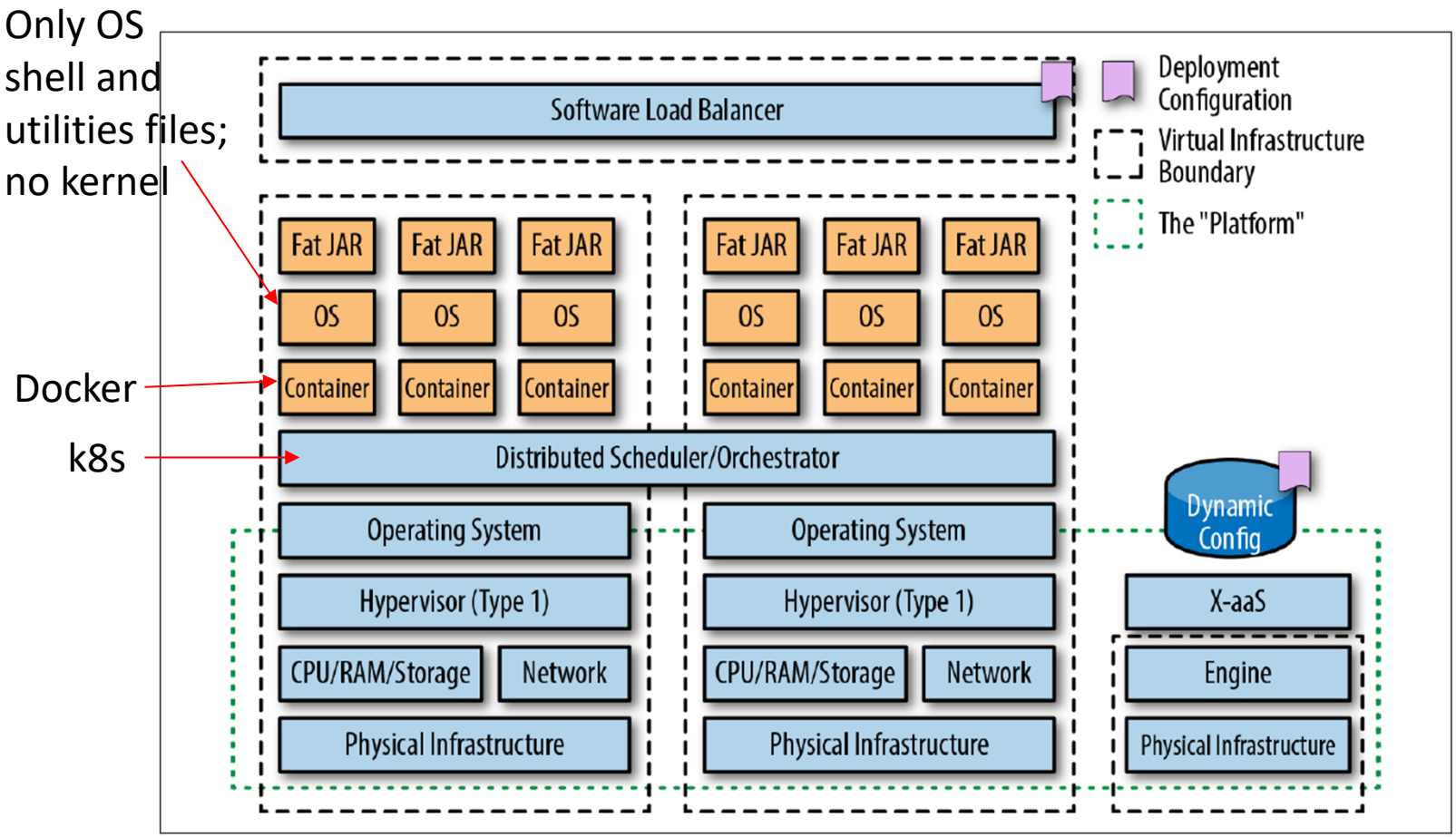
- 從manifest找config
- 從manifest找image layers

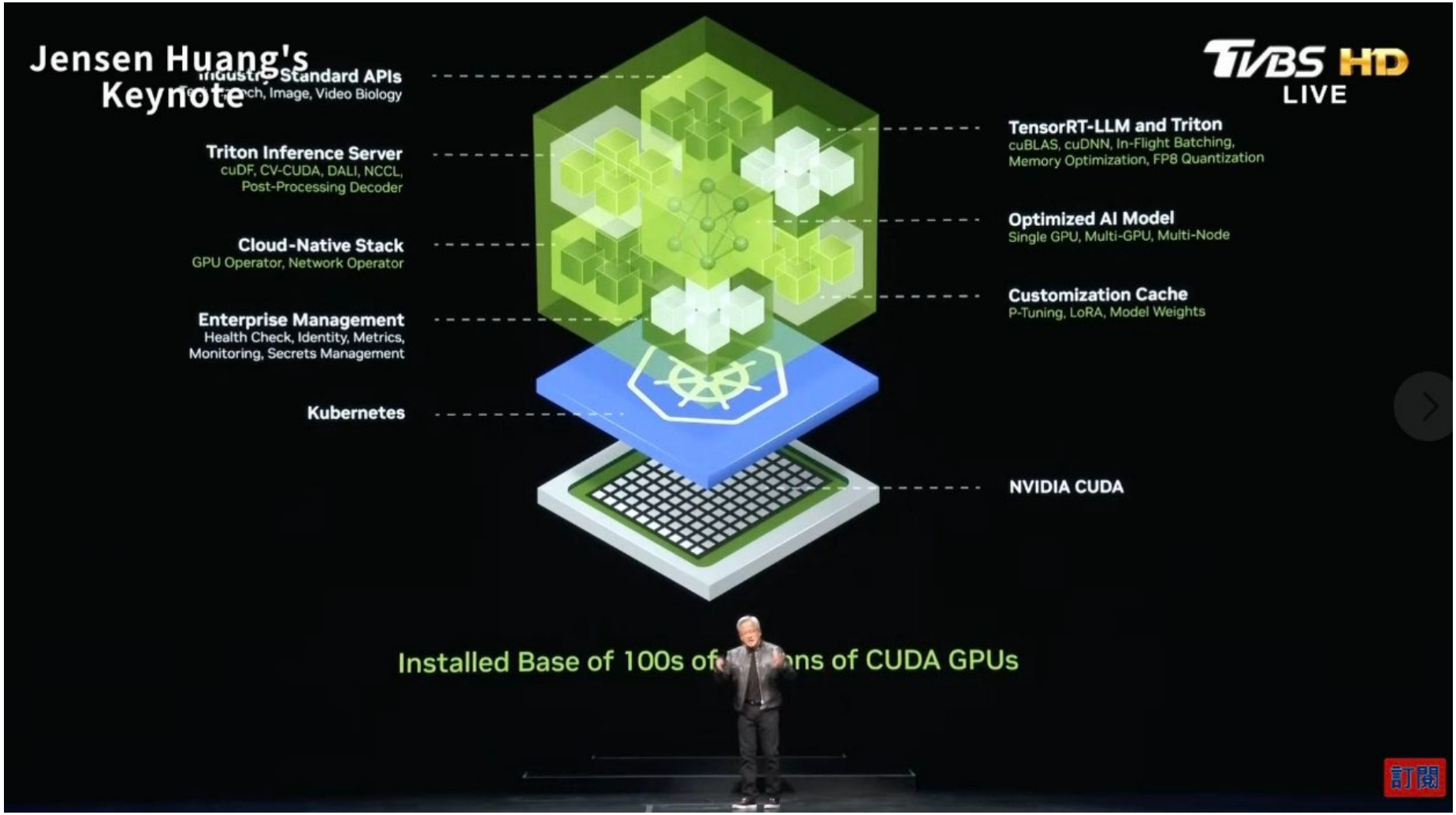



```
{
  "schemaVersion": 2,
  "config": {
    "mediaType": "application/vnd.oci.image.config.v1+json",
    "digest": "sha256:9d446b871e5882110acf8dc0ab827425b8d25184f942
    "size": 6566
  },
  "layers": [
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "digest": "sha256:eff15d958d664f0874d16aee393cc44387031ee0a6
      "size": 31370267
    },
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "digest": "sha256:1e5351450a593c3a3d7a5104f93c8b80d8dc00c827
      "size": 25347687
    },
```

詳細步驟可參考: https://vividcode.io/understanding-oci-image-spec/

Cloud Native 系統服務的運行

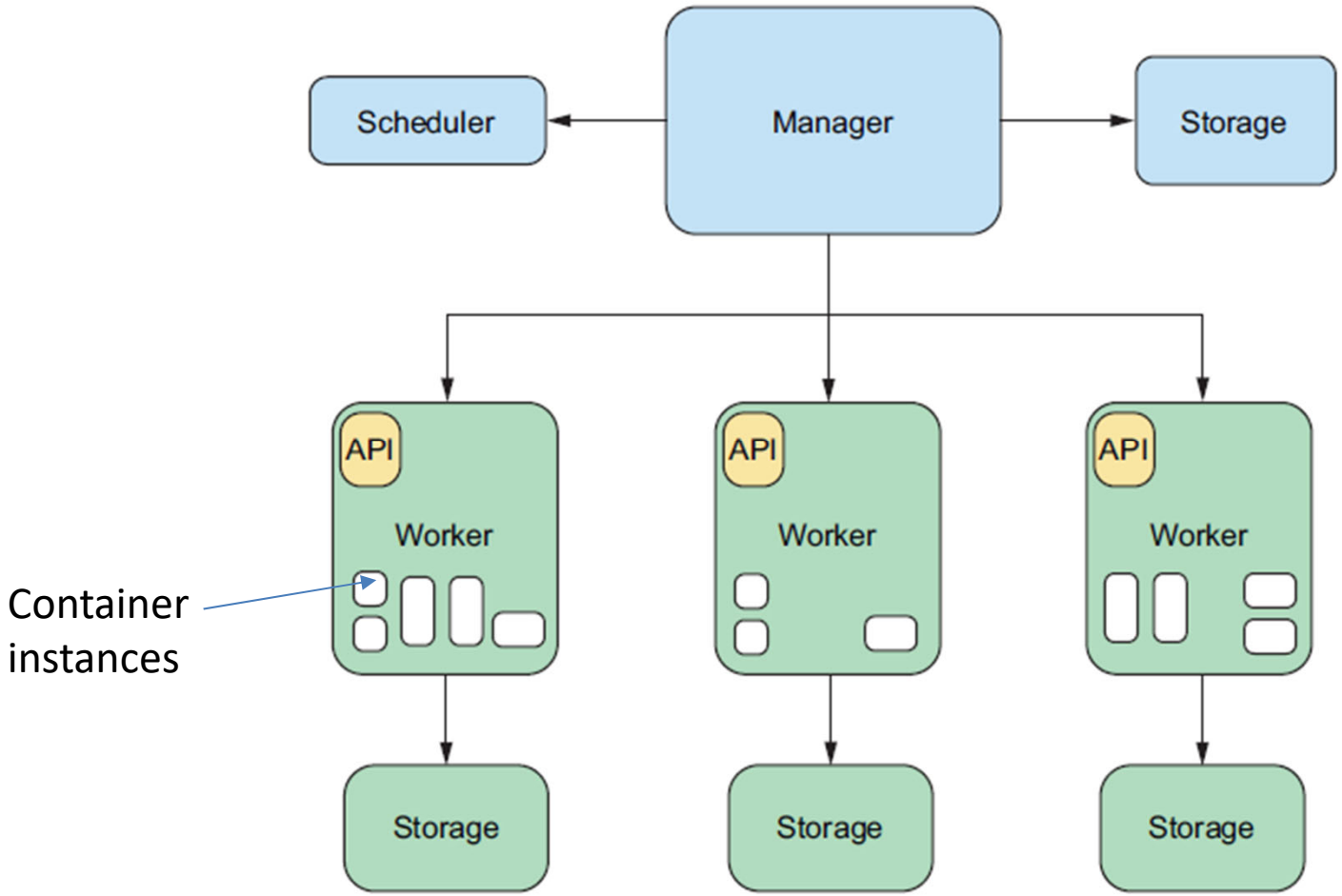



Jensen Huang's Keynote
TVBS HD LIVE
Industry Standard APIs
Text, Speech, Image, Video Biology
Triton Inference Server
cuDF, CV-CUDA, DALI, NCCL,
Post-Processing Decoder
Cloud-Native Stack
GPU Operator, Network Operator
Enterprise Management
Health Check, Identity, Metrics,
Monitoring, Secrets Management
Kubernetes
TensorRT-LLM and Triton
cuBLAS, cuDNN, In-Flight Batching,
Memory Optimization, FP8 Quantization
Optimized AI Model
Single GPU, Multi-GPU, Multi-Node
Customization Cache
P-Tuning, LoRA, Model Weights
NVIDIA CUDA
Installed Base of 100s of
ns of CUDA GPUs
訂閱

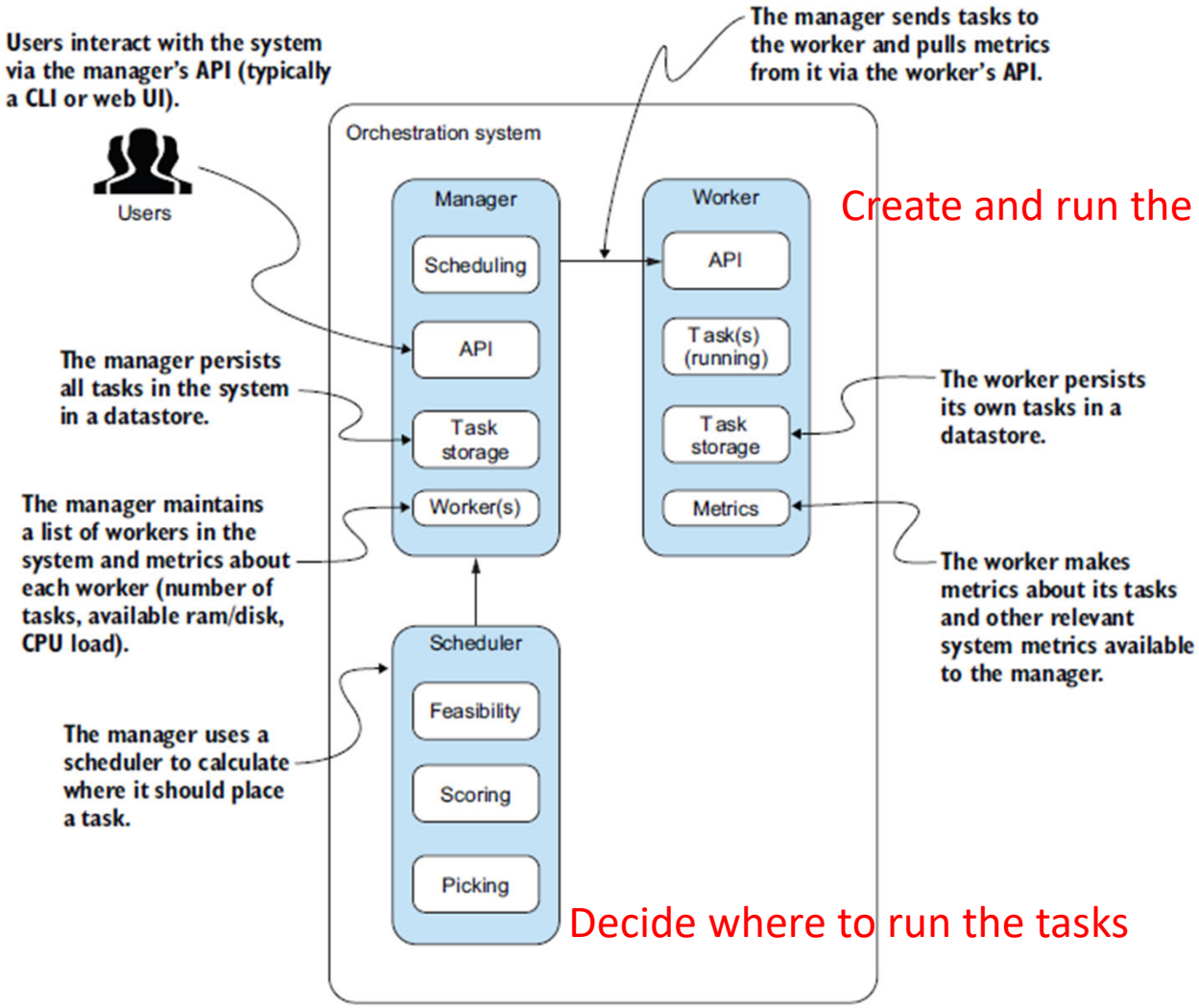
Container Orchestration

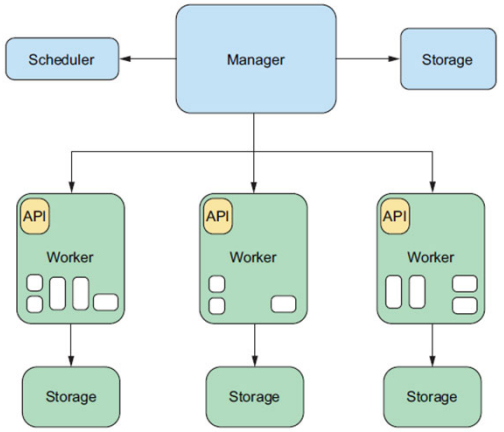
- Purpose
  - To manage the life cycle of containers at scale
- Tasks
  - Resource management
    - Placing containers on nodes that provide sufficient resources
    - Moving containers to other nodes if the resource limits is reached
    - Health monitoring and restarting
    - Scaling in or out
  - Networking
    - Providing mappings for containers to connect to networking
    - Internal load balancing between containers

Container Orchestration的一般性結構



Container Orchestration的一般性結構





Kubernetes

- An open source project for container orchestration
  - Contributed by Google in 2014
  - Borg: the platform behind Google Search, Gmail, and YouTube
  - Kubernetes leverages Borg's innovations and lessons learned
- Competitors
  - Apache Mesos
  - Docker Swarm

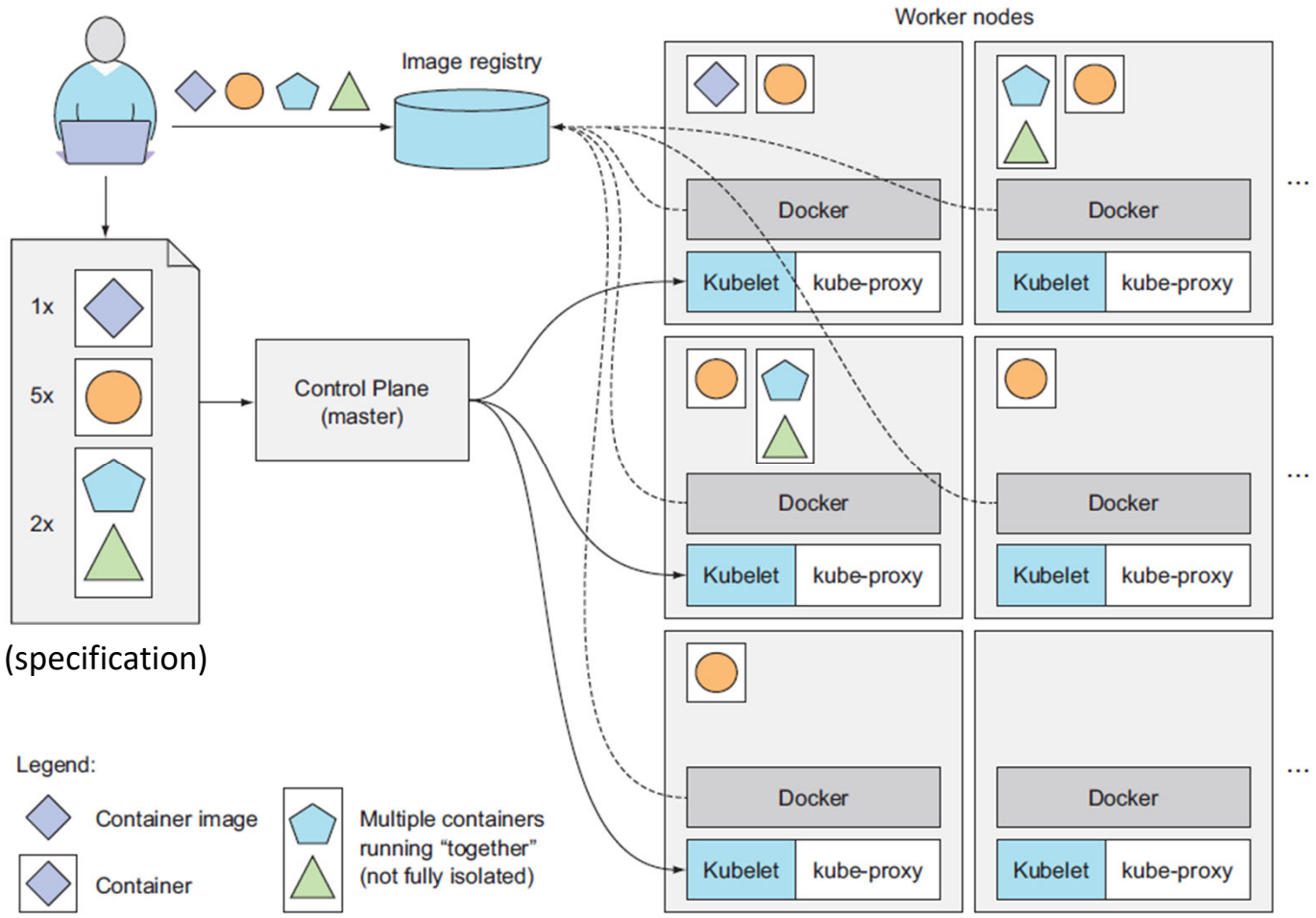
Kubernetes

- What Kubernetes does
  - Service-orientation
    - Service discovery via internal DNS
    - Runtime binding of address
  - Ad hoc cluster-wide LAN
  - Load balancing (horizontal scaling)
  - Self-healing (by restart)
  - Rollout and rollback
    - Change to specific version automatically and on demand

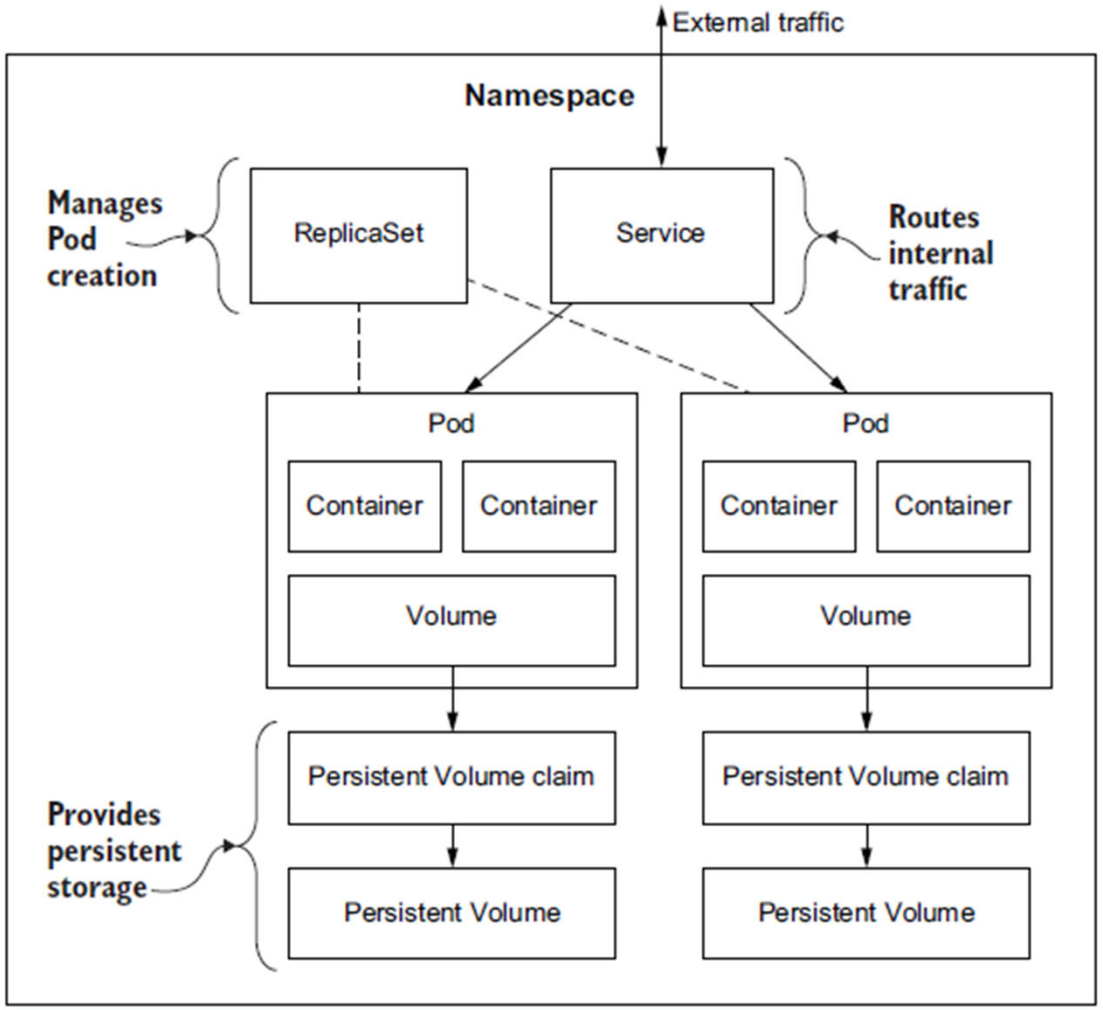
Kubernetes

- What Kubernetes doesn't do
  - Diagnose the problems/bugs
  - Cross-cutting application-level services
    - Transactions, ORM, RPC….
  - Build container image
    - Only pull OCI-compliant images from image registry
    - Should not build image in the cluster environment

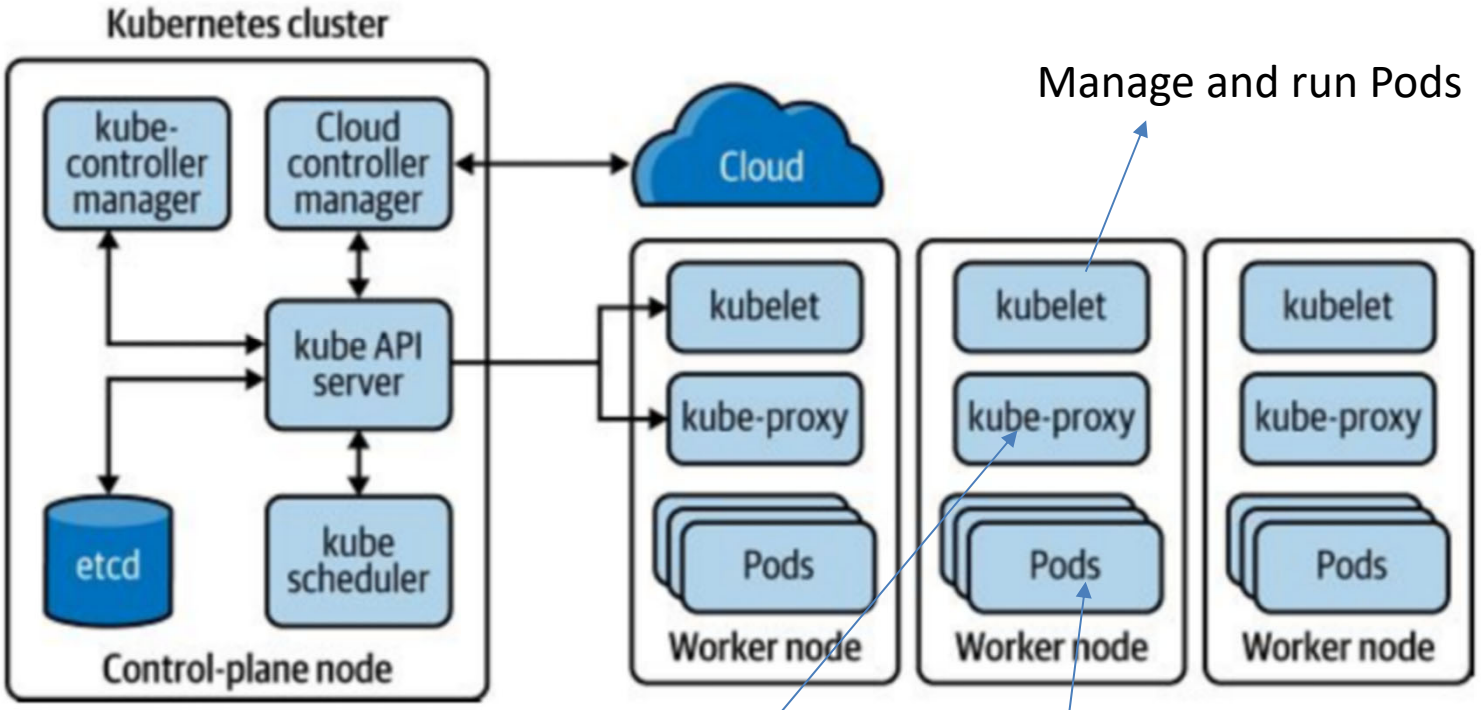
Kubernetes運行架構



Logical View

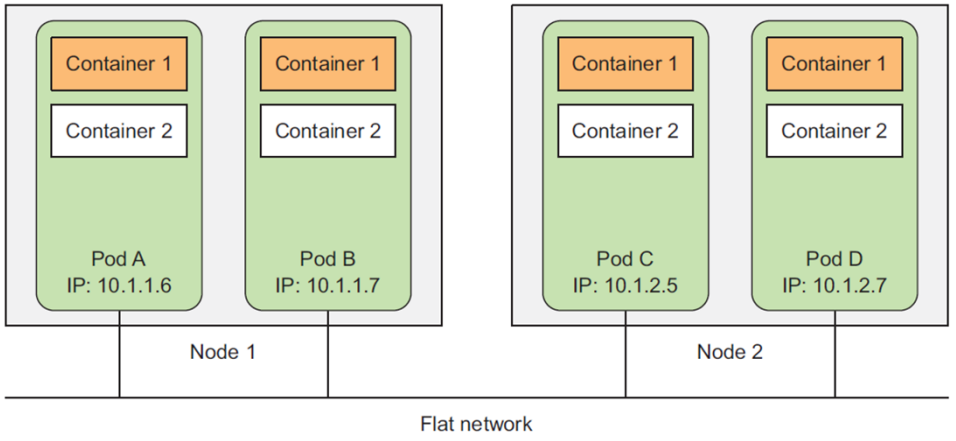


Kubernetes: a high-level view



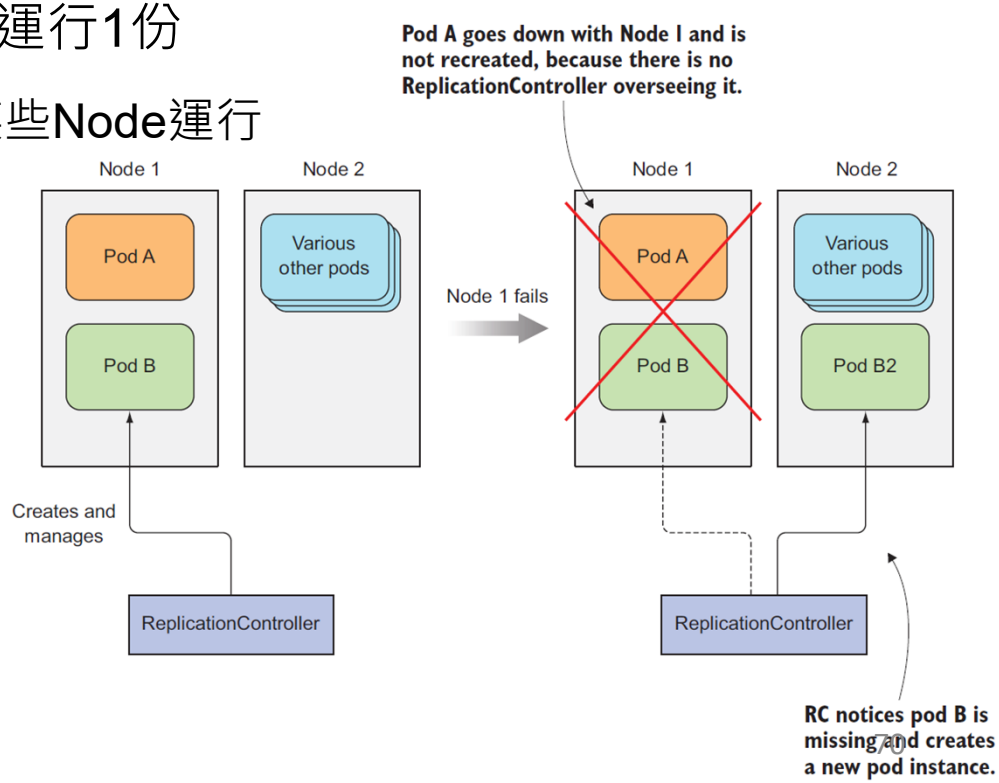
重要元素

- Pod
  - 一個Pod可放置1到多個container instances (通常是1個)
    - 同pod中的containers只有partial isolation
    - 同Pod中的containers共享資源
      - PID (disable by default), UTS, MNT, IPC, NET
      - Ex: share the same IP address (containers 可透過localhost:port來互相存取)
  - Pod是k8s做整體調控的基本單位
  - 屬於同k8s cluster中的Pods，概念上同屬於一個網路(no NAT)
    - 彼此可透過IP直接相互存取
- Node
  - 主機 (實體或虛擬)
  - 一個Node可內含多個Pods



重要元素

- ReplicaSet
  - 確保Pods (至少n個副本)的運行
  - Replication Controller的後繼者
- DaemonSet
  - 確保某個Pod在每個Node均運行1份
    - 也可以透過進階設定只在某些Node運行

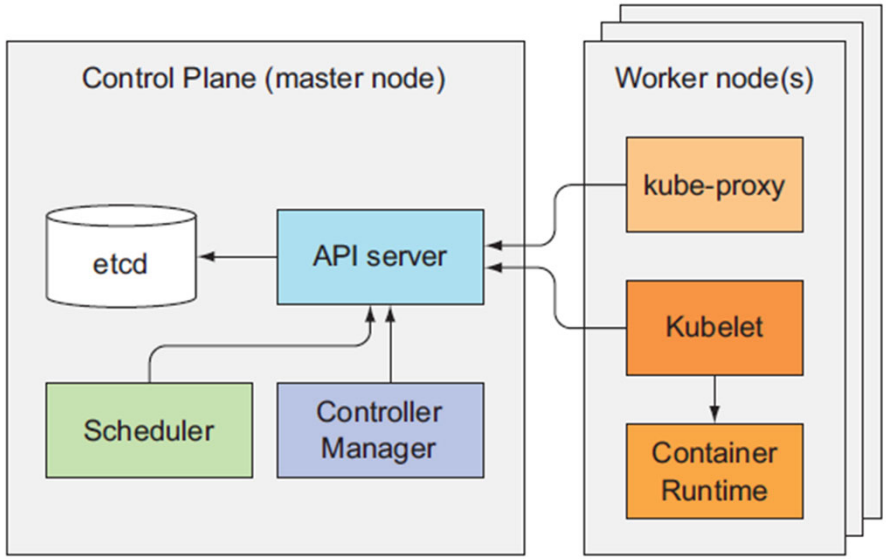


重要元素

- Service
  - 做為存取Pods的統一入口
    - Expose pods at a single and stable IP/Port
    - 如果是外部，需要配合定義對外接口
    - 內部可直接存取
- Volume
  - 用來bind pod外部檔案系統

Kubernetes Infrastructure

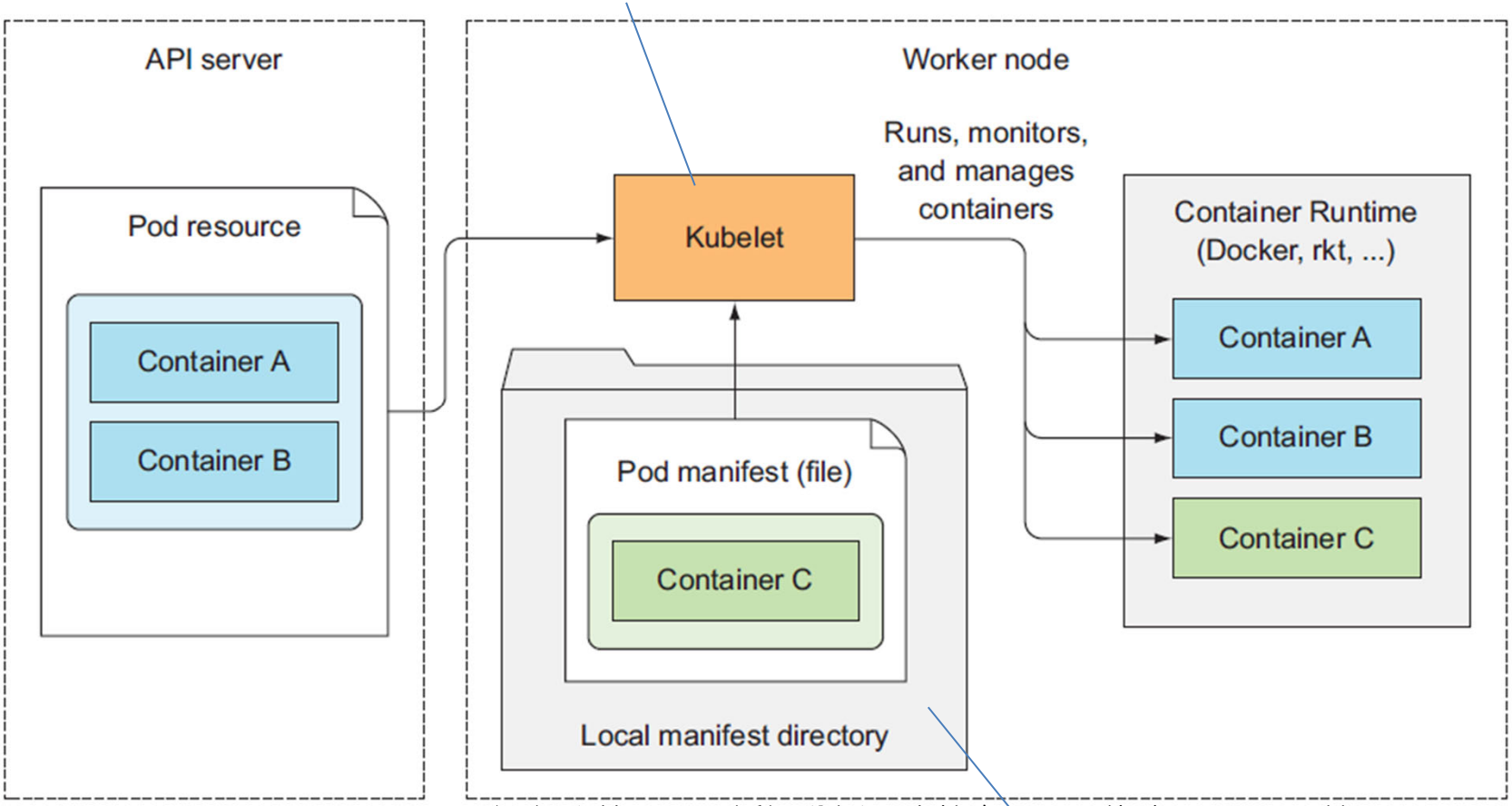
- Master Node: container cluster control pane
  - Kube-apiserver
  - etcd (KV store)
  - kube-scheduler (deploy new container to pods)
  - Kube-controller-manager
- Worker Node
  - Kubelet (local manager of pods)
  - Kube-proxy (network mapping)
  - Container runtime



K8s上所有的系統管理元件不直接溝通; 而是透過API Server溝通
Etcd也只被API Server維護
Master node上的系統元件，也可以變成pod方式運行
(此時master node上也要運行kubelet)

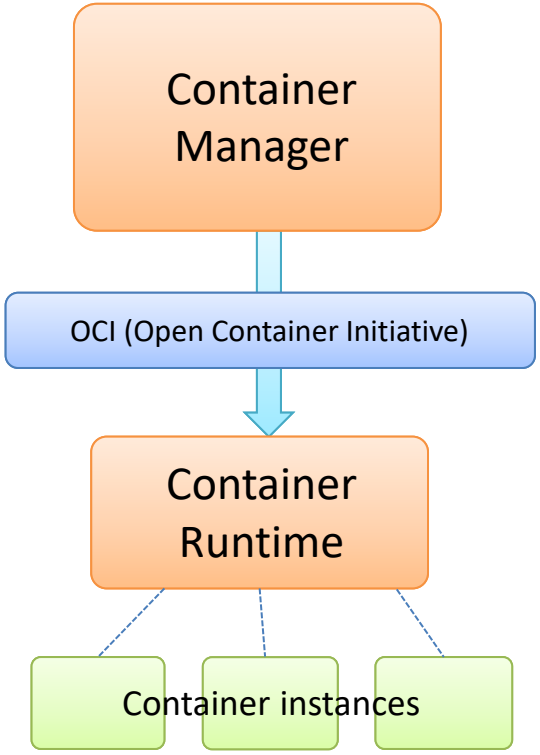
Kubelet功能

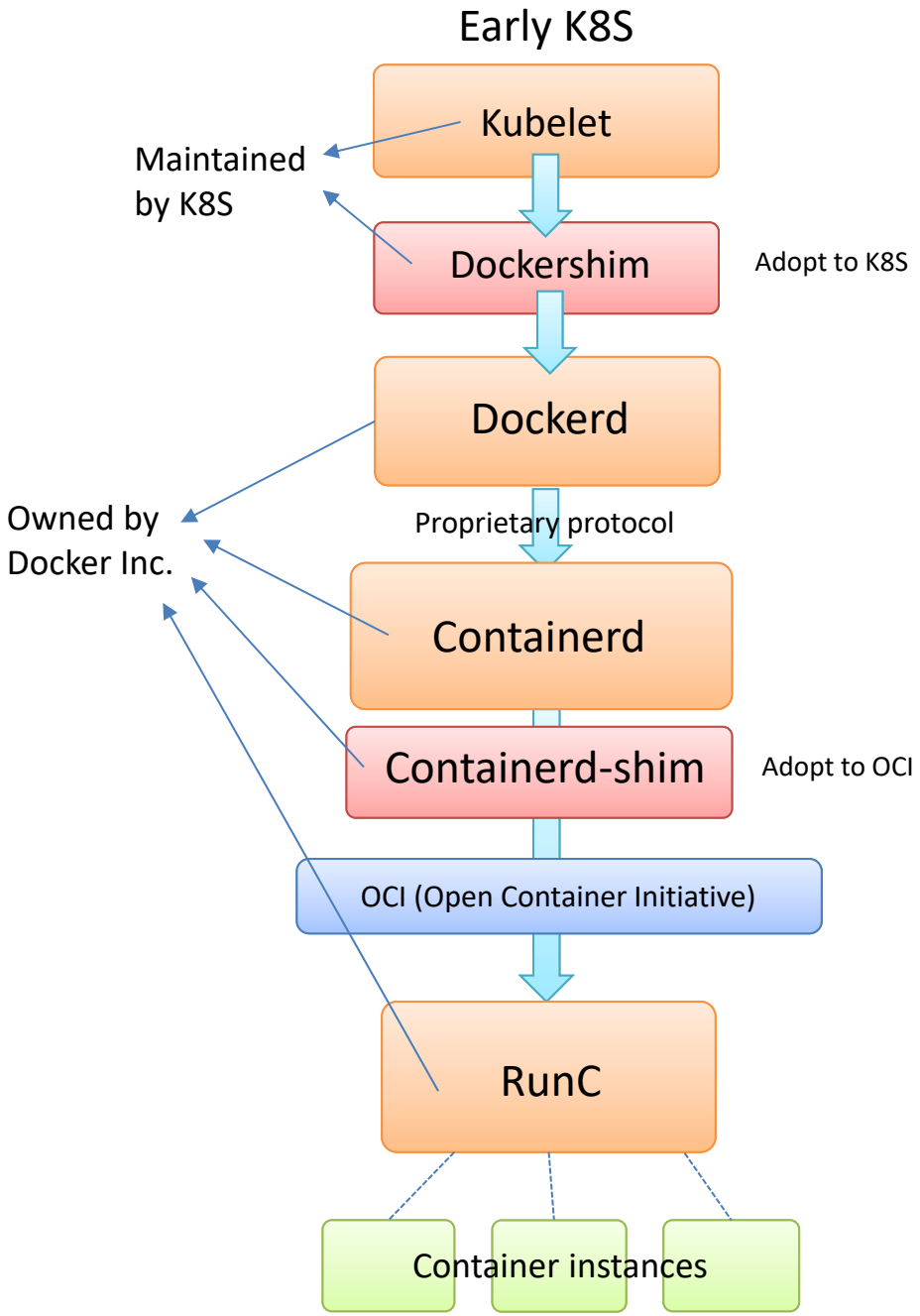
Kubelet不斷詢問API Server是否有要新加的pods，若有，就新建pod並下傳並執行裡面的containers
Pod(與裡面的containers)建好後，kubelet向API server註冊，並持續向API Server回報pod狀況



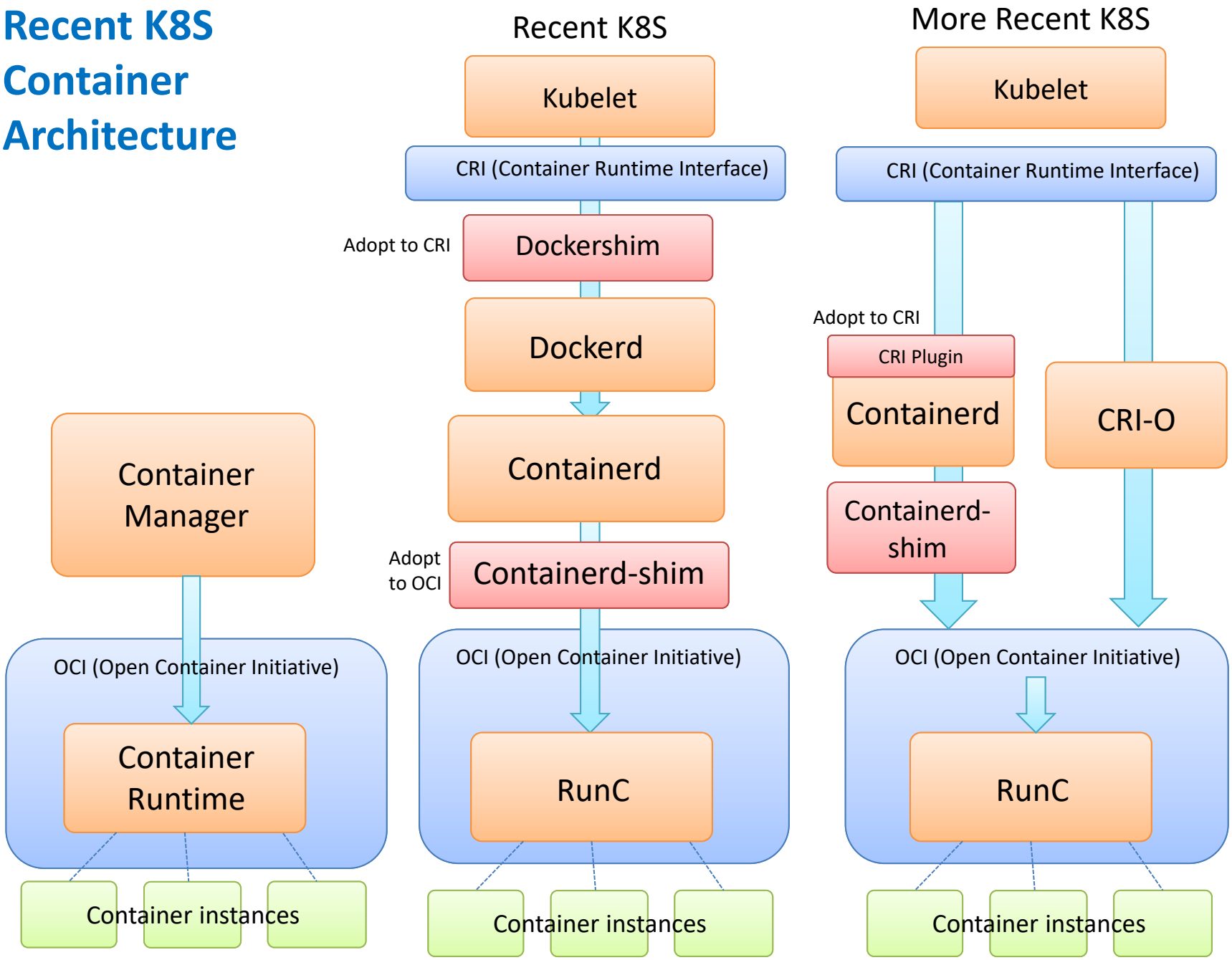
運行系統pod: 不透過分派，直接在local目錄建manifest，讓kubelet直接跑pod (bootstrapping)

Early K8S Container Architecture

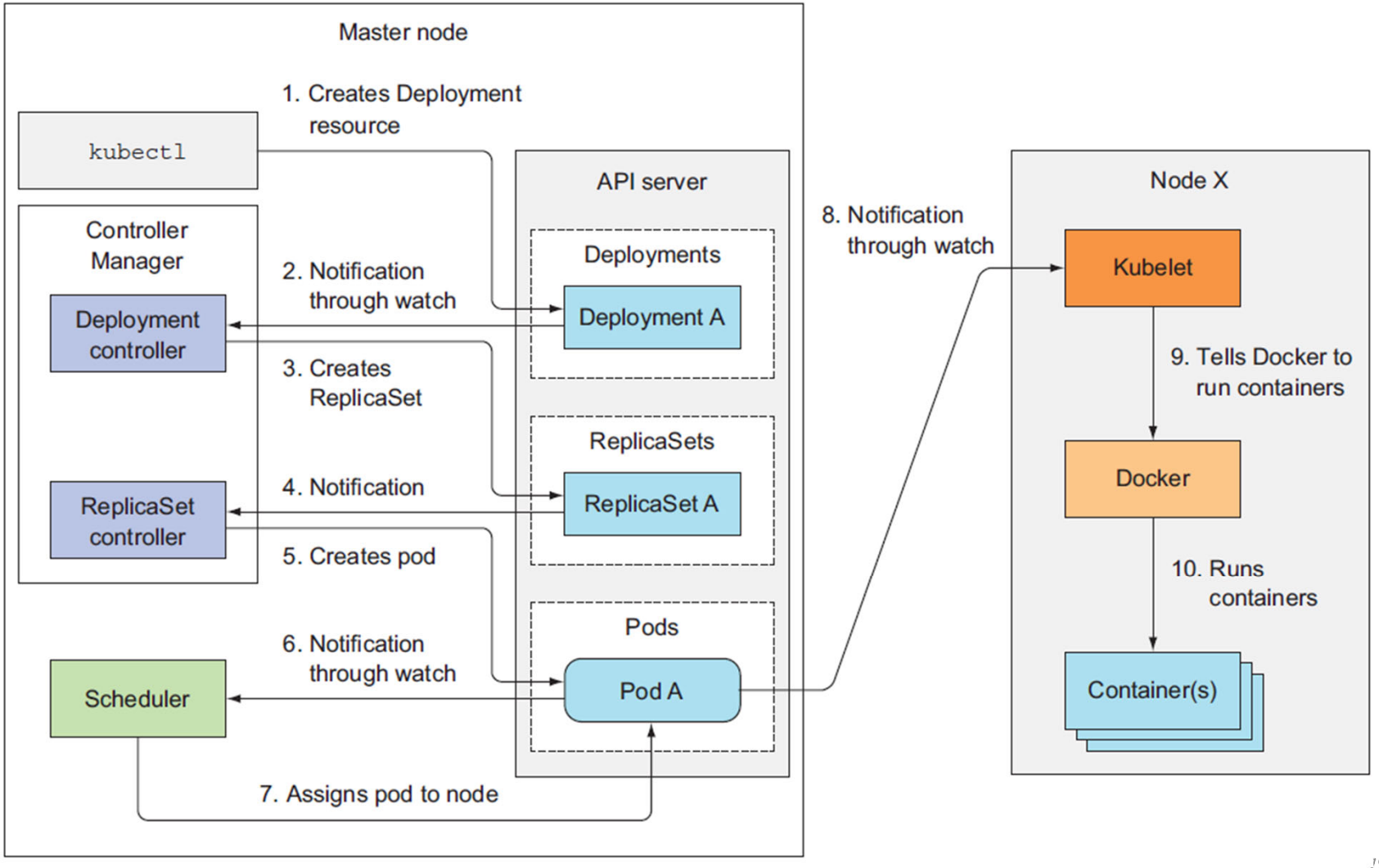




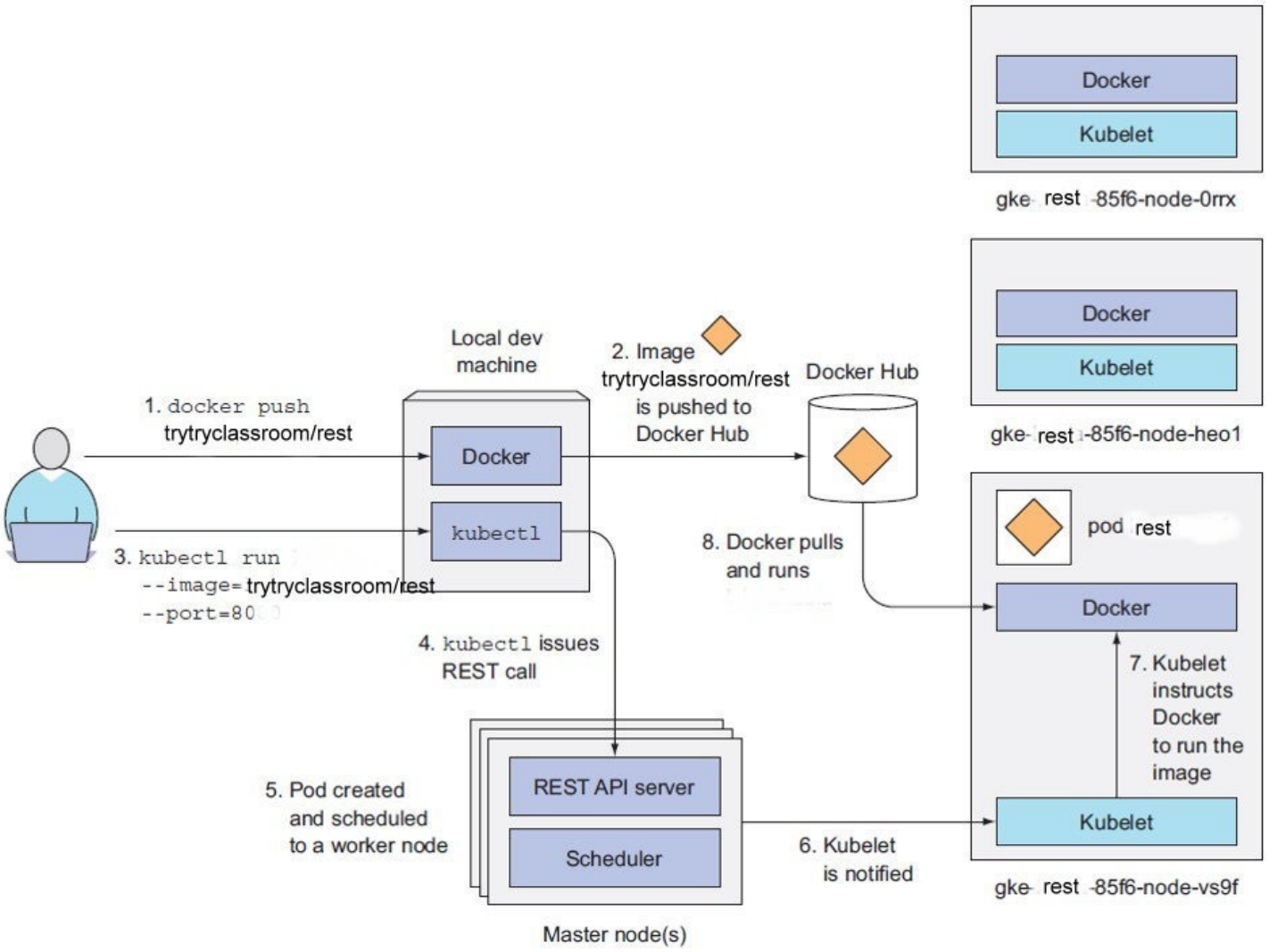
Recent K8S Container Architecture



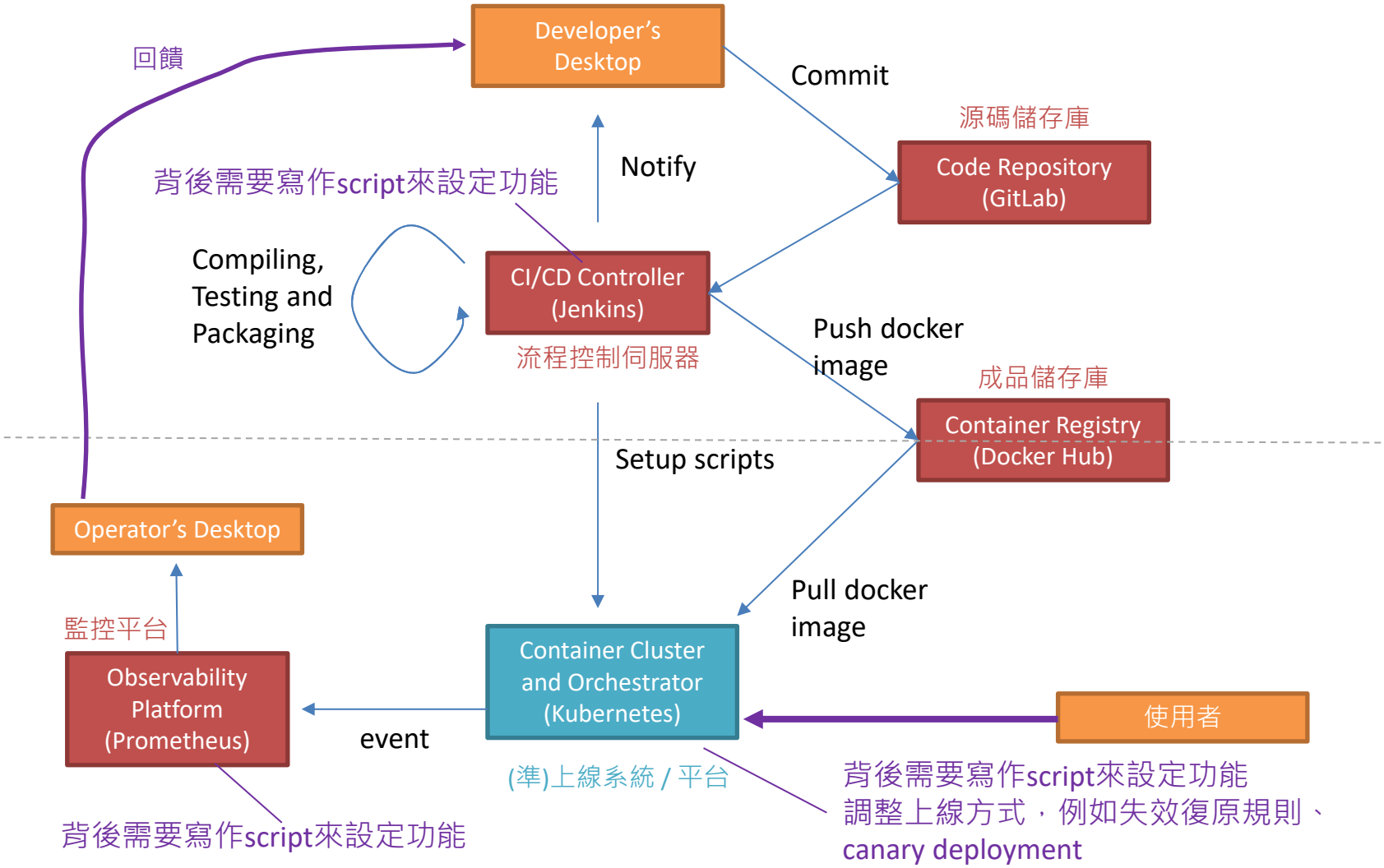
整體部署流程


Master node
1. Creates Deployment resource
kubectl
Controller Manager
Deployment controller
2. Notification through watch
3. Creates ReplicaSet
ReplicaSet controller
4. Notification
5. Creates pod
Scheduler
6. Notification through watch
7. Assigns pod to node
API server
Deployments
Deployment A
ReplicaSets
ReplicaSet A
Pods
Pod A
8. Notification through watch
Node X
Kubelet
9. Tells Docker to run containers
Docker
10. Runs containers
Container(s)

Example

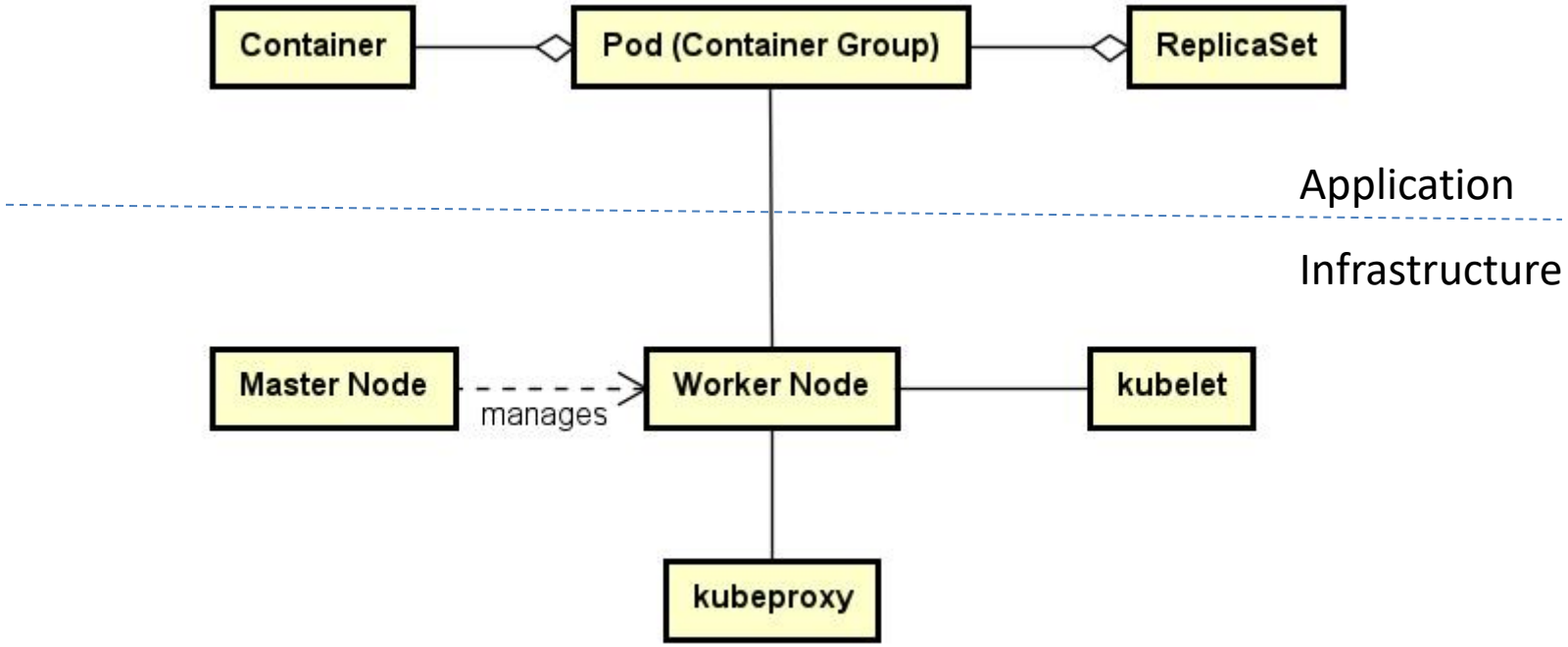

Docker
Kubelet
gke- rest -85f6-node-0rrx
Docker
Kubelet
gke- rest -85f6-node-heo1
Local dev machine
2. Image trytryclassroom/rest is pushed to Docker Hub
Docker Hub
1. docker push trytryclassroom/rest
Docker
kubectl
3. kubectl run --image= trytryclassroom/test --port=80
8. Docker pulls and runs
pod rest
Docker
7. Kubelet instructs Docker to run the image
4. kubectl issues REST call
5. Pod created and scheduled to a worker node
REST API server
Scheduler
Master node(s)
6. Kubelet is notified
Kubelet
gke- rest -85f6-node-vs9f

The “Cloud Native” CI/CD Pipeline



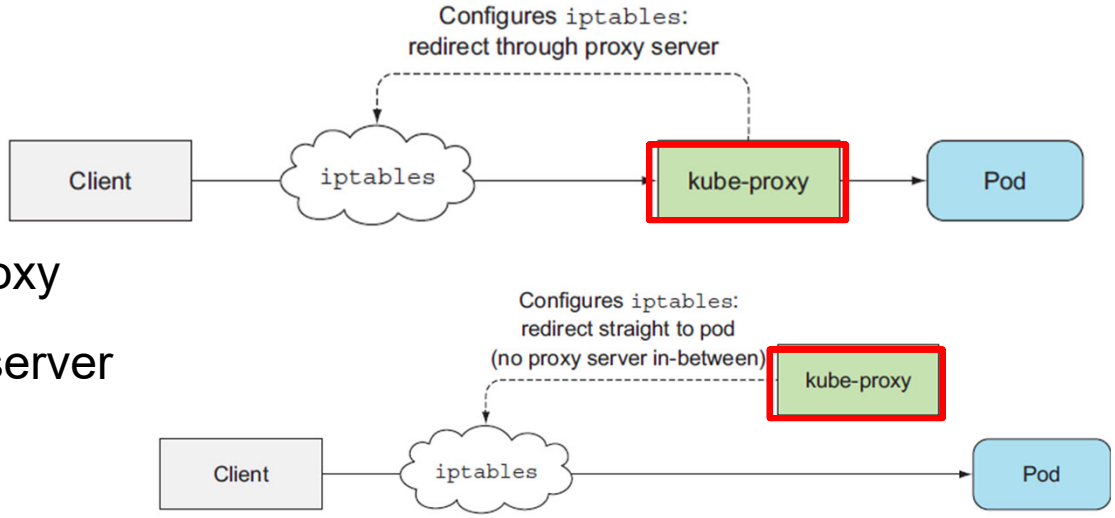
Summary

- Key concepts

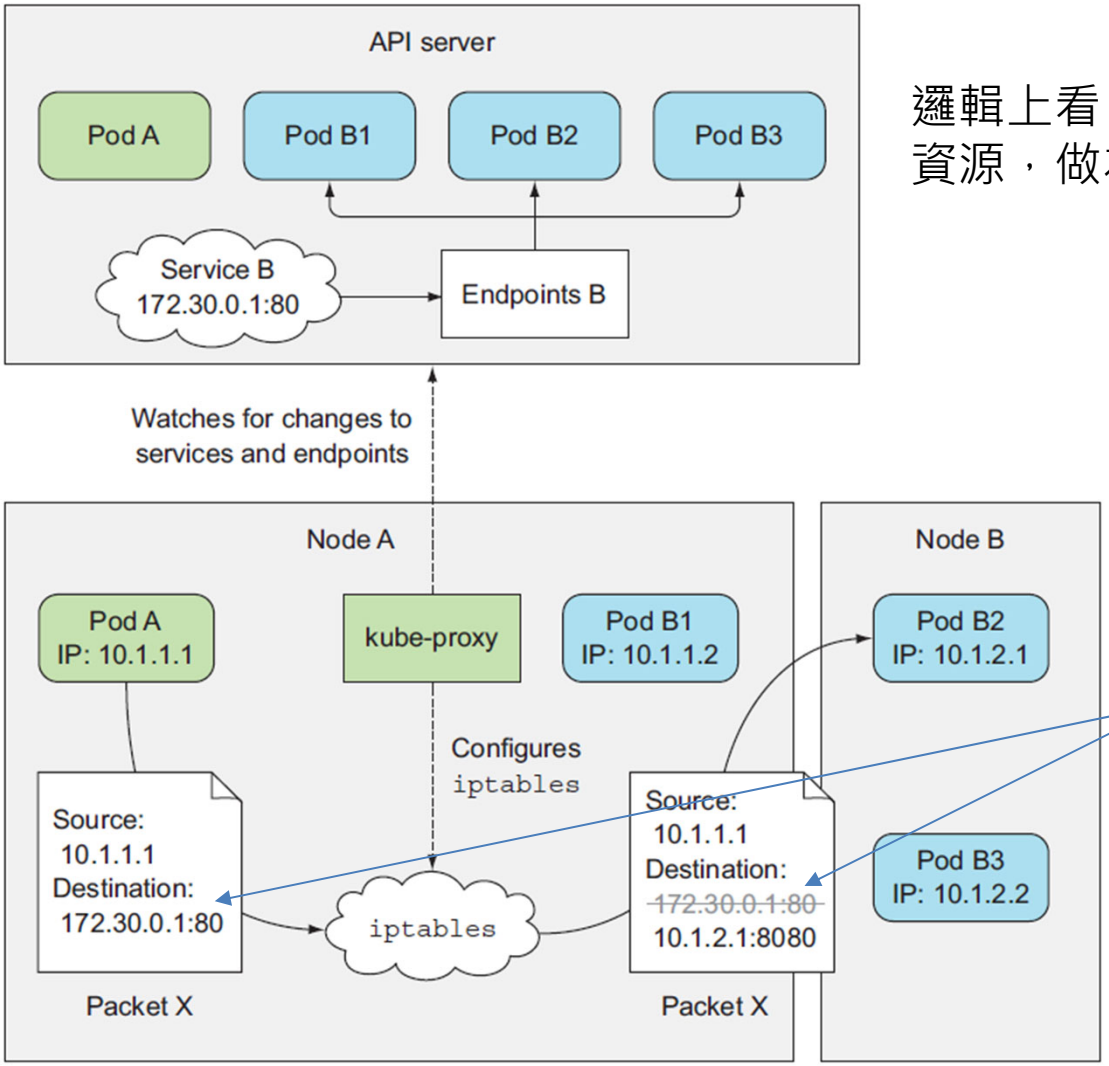


Kube-proxy

- 外部clients如何連接到直正的pods?
  - 外部clients 只知道 service的IP/Port
  - 後台的Pods只有private IPs
  - 如何找到、連到真正的pods?
- 二種方式
  - userspace proxy mode
    - 重導所有要求到kube-proxy
    - Kube-proxy as a proxy server
  - iptables proxy mode
    - kube-proxy動態設定iptables設定繞送路徑



Kube-proxy



邏輯上看，ServiceB是一個有特定IP/Port的資源，做為Pod B1-Pod B3存取入口

- 假設現Pod A是client要存取Service B
- Node 上的kube proxy修改iptables規則，將172.30.0.1:80 (Service B)對應到10.1.2.1:80 (Pod B2)

Q & A