IoT: Cloud Services

Containers

Motivation

- The Internet of Things, the highly connected network of smart devices, such as environmental sensors, medical trackers, home appliances, and industrial devices, is growing rapidly.
- By 2020, a predicted 20 billion devices will be connected, which is more than twice the number of PCs, smartphones, and tablets combined.
- Containers are a lightweight approach to virtualization that developers can apply to rapidly develop, test, deploy, and update IoT applications at scale.
- IoT applications target a wide variety of device platforms. During prototyping and early phases of development, IoT projects are often developed using generic microcontroller-based development boards or single-board computers (SBCs) like the Raspberry Pi.

From microservices to containers

- Agile software development is a broadly adopted methodology in enterprises today.
- "full-stack-responsibility" for the individual services.
  - infrastructures need to scale differently and the self-service model for projects is taking center stage.
- Containers are the foundation for this.
  - Along with proper DevOps and orchestration.

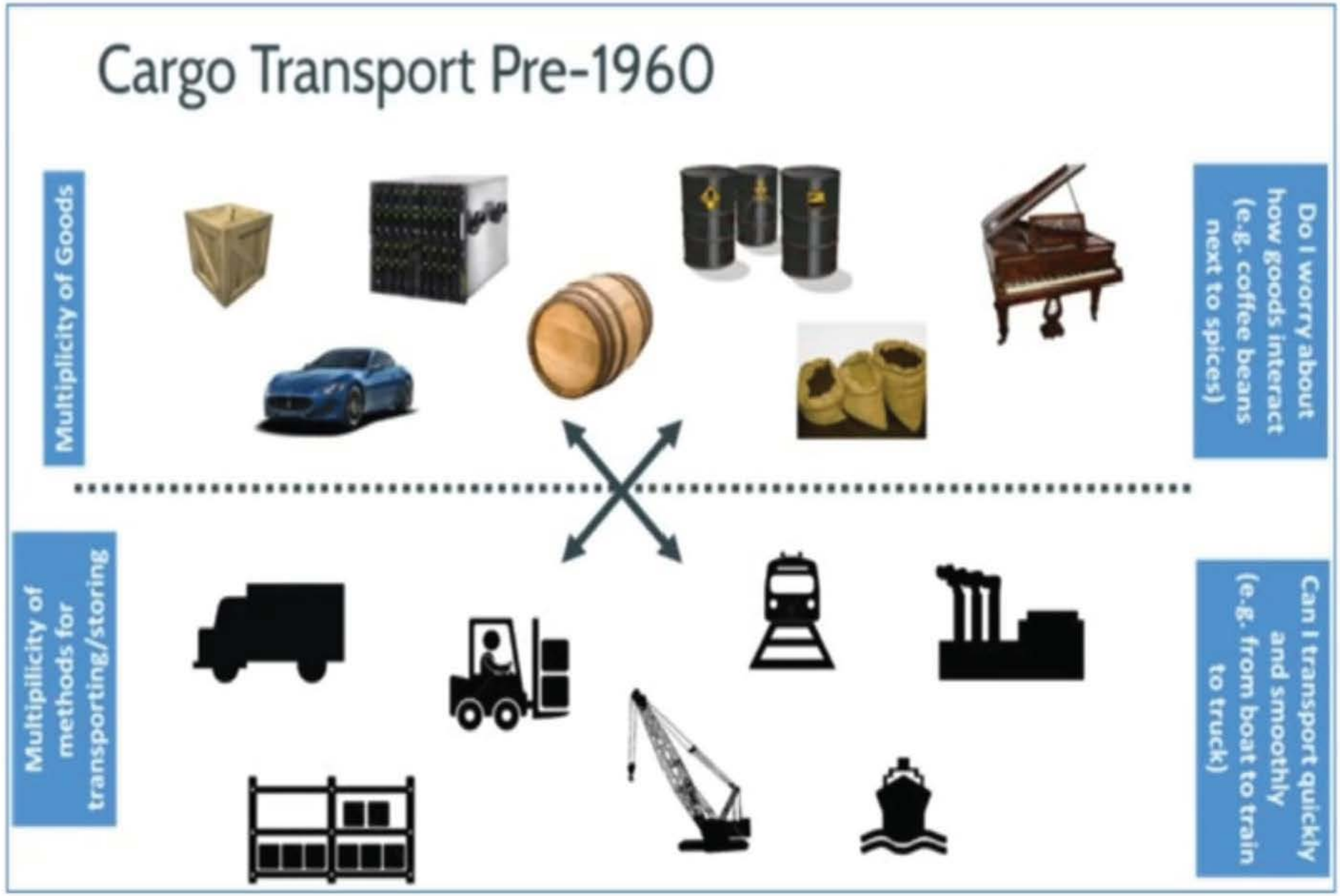
Containers - New Degree of Freedom

New concept of virtualization solution for PaaS and IaaS due to container's increased density, isolation, elasticity and rapid provisioning

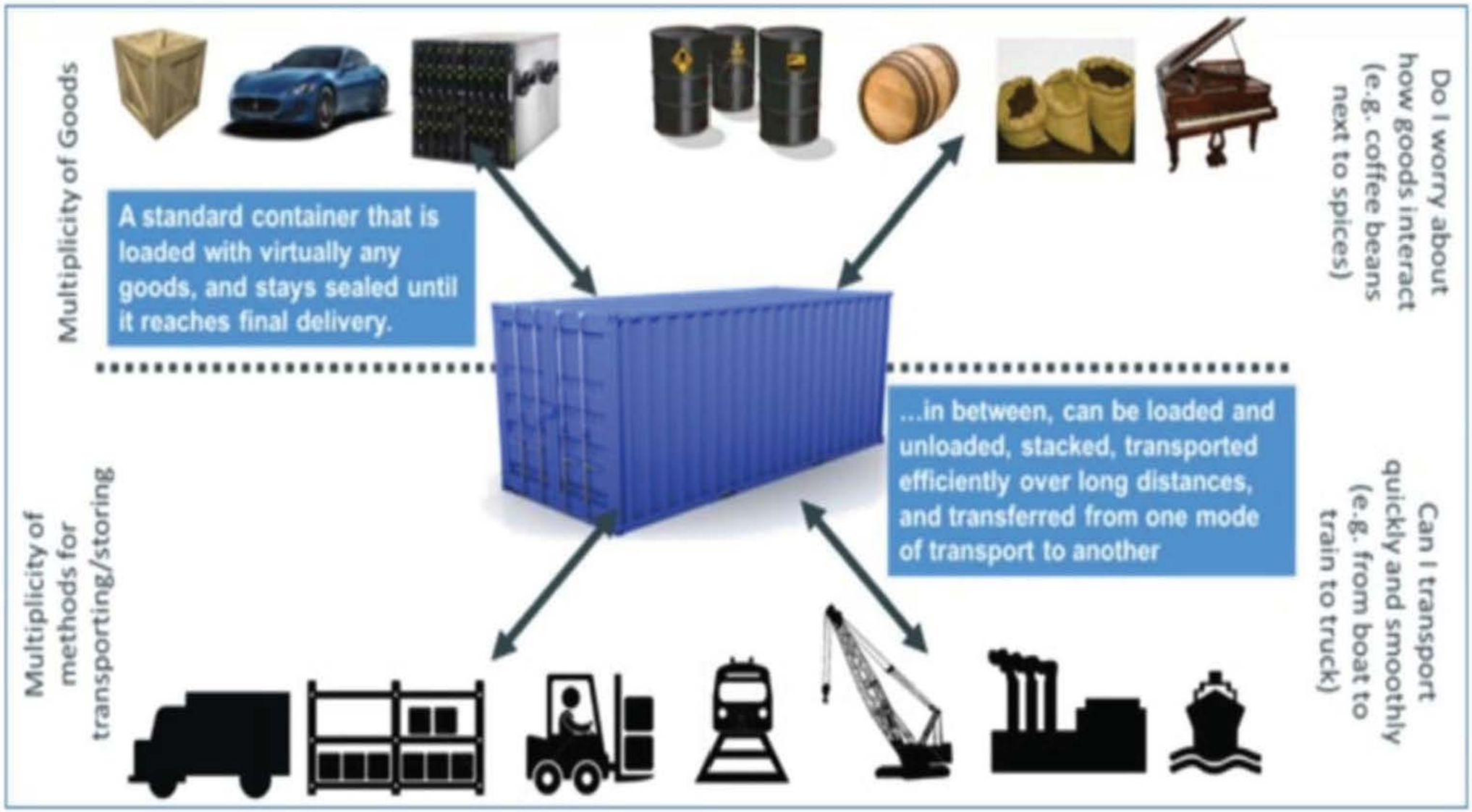
Containerization:

- Use lightweight packages instead of full VMs
- Move from a single large monolithic app to composition of microservices
- Containerize different parts of an application
- Move parts of apps into different types of cloud infrastructure
- Simplify migration of applications between private, public and hybrid clouds

History of Cargo Transportation

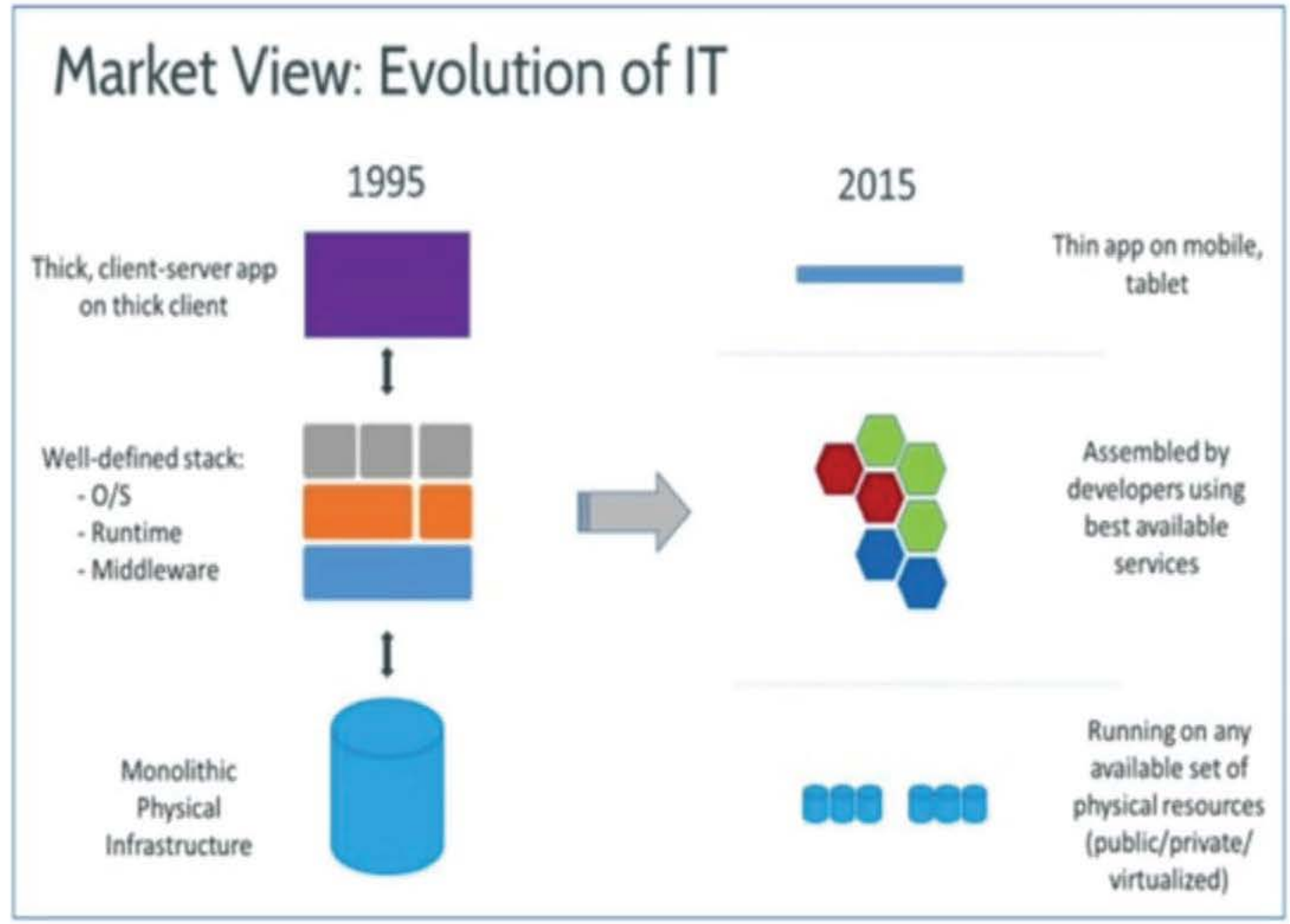


Solution to shipping Challenge

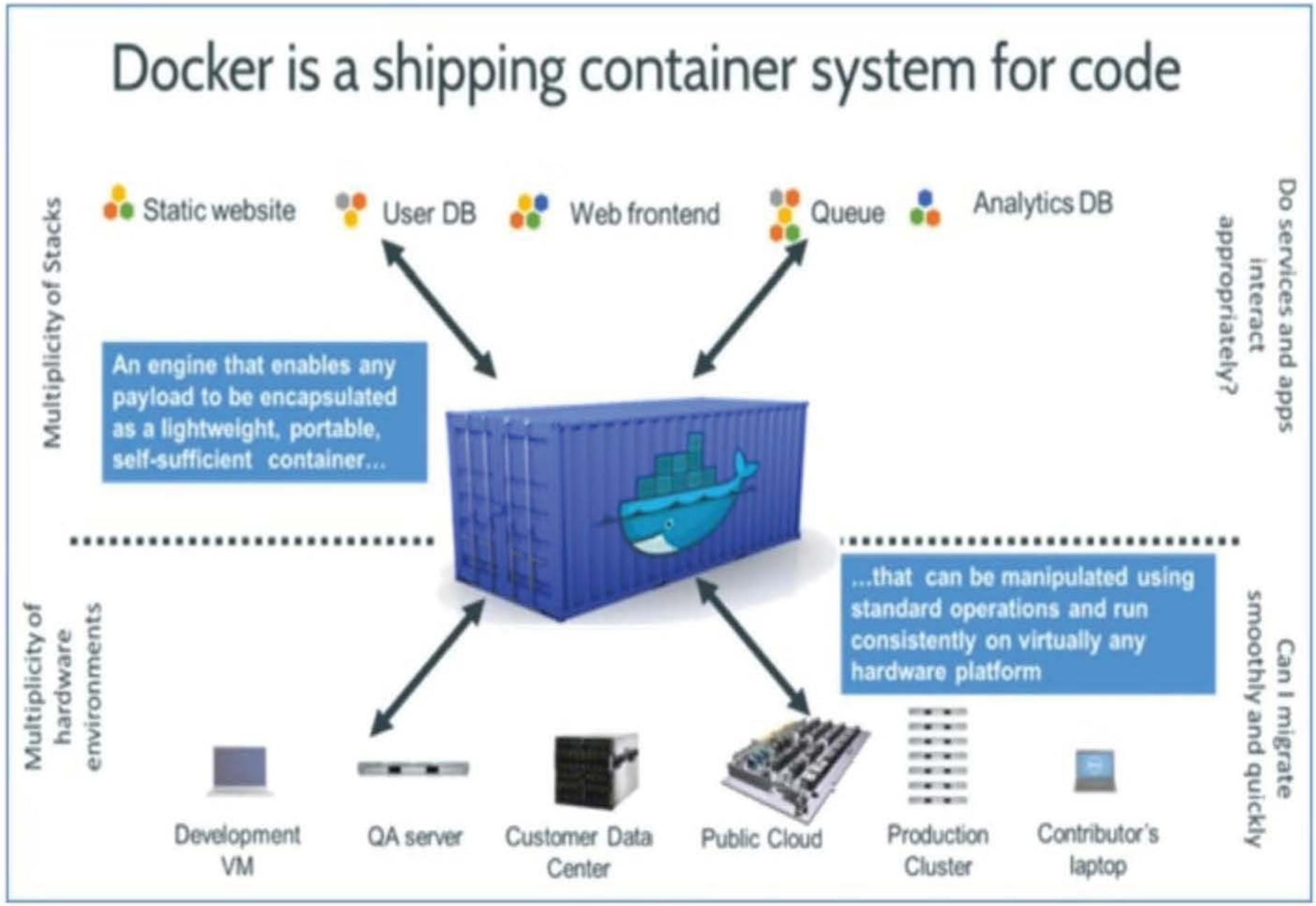


Analogy with Virtualization

- Historical way: $M mainframes
- Modern way: virtual machines
- Problem: performance overhead



The Solution to software shipping



IoT: Cloud Services

Containers or VMs

Why use Containers?

- Reduces build & deploy times
- Cost control and granularity
- Libraries dependency can be complicated
- Infrastructure configuration = spaghetti

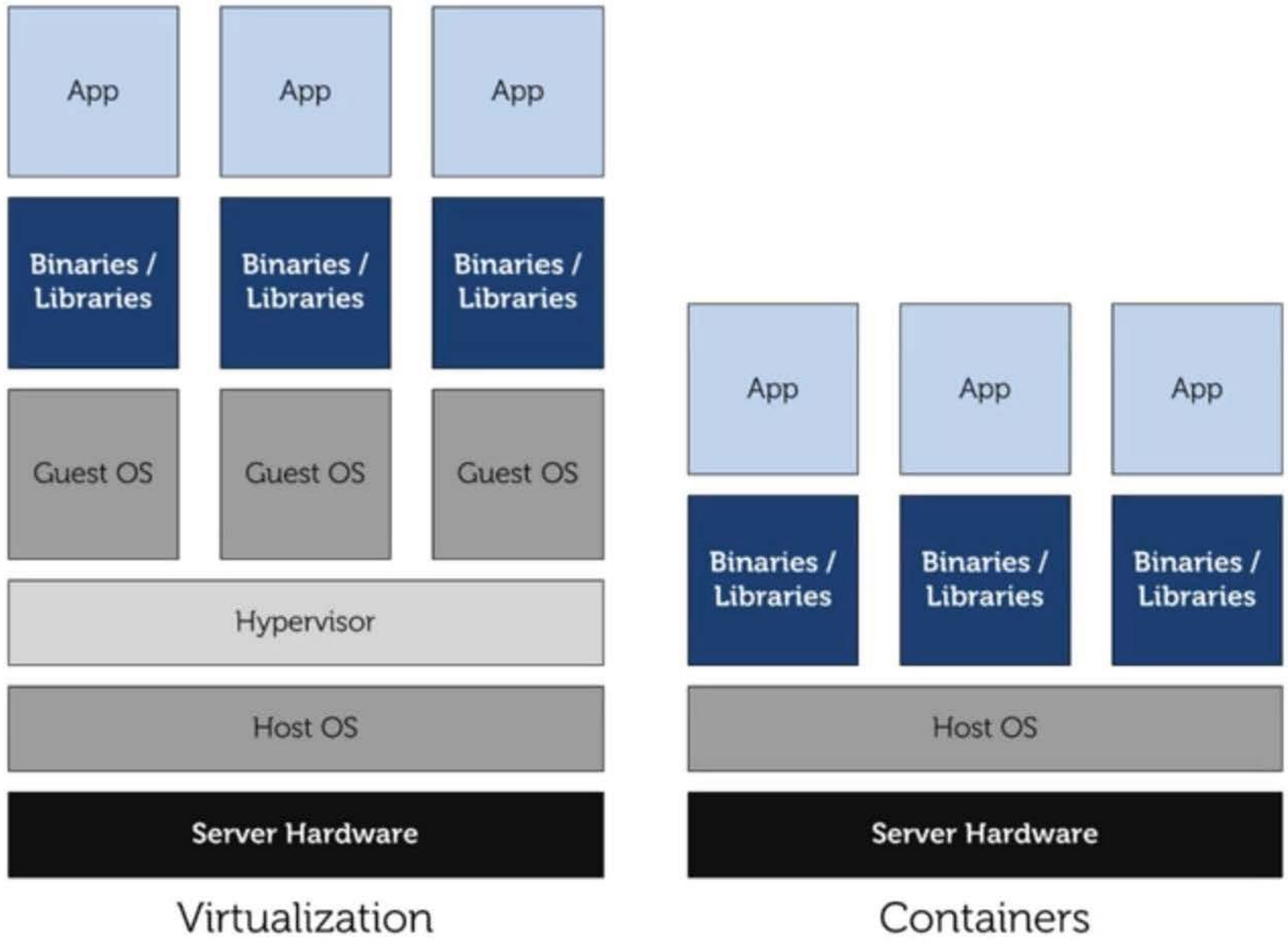
Containers vs VMs

Container technology simplifies cloud portability.

- Run same application in different environments

1. A container encapsulates applications and defines their interface with the surrounding system
2. In a virtual machine: a full OS install with the associated overhead of virtualized device drivers, etc.,
   - Containers use and share the OS and device drivers of the host.
3. Virtual machines have a full OS with its own memory management, device drivers, daemons, etc.
   - Containers share the host's OS and are therefore lighter weight.

Containers vs VMs



Containers can and can't

Can

- Get shell (i.e. ssh)
- Own process space
- Own network interface
- Run as root
- Install packages
- Run services
- …

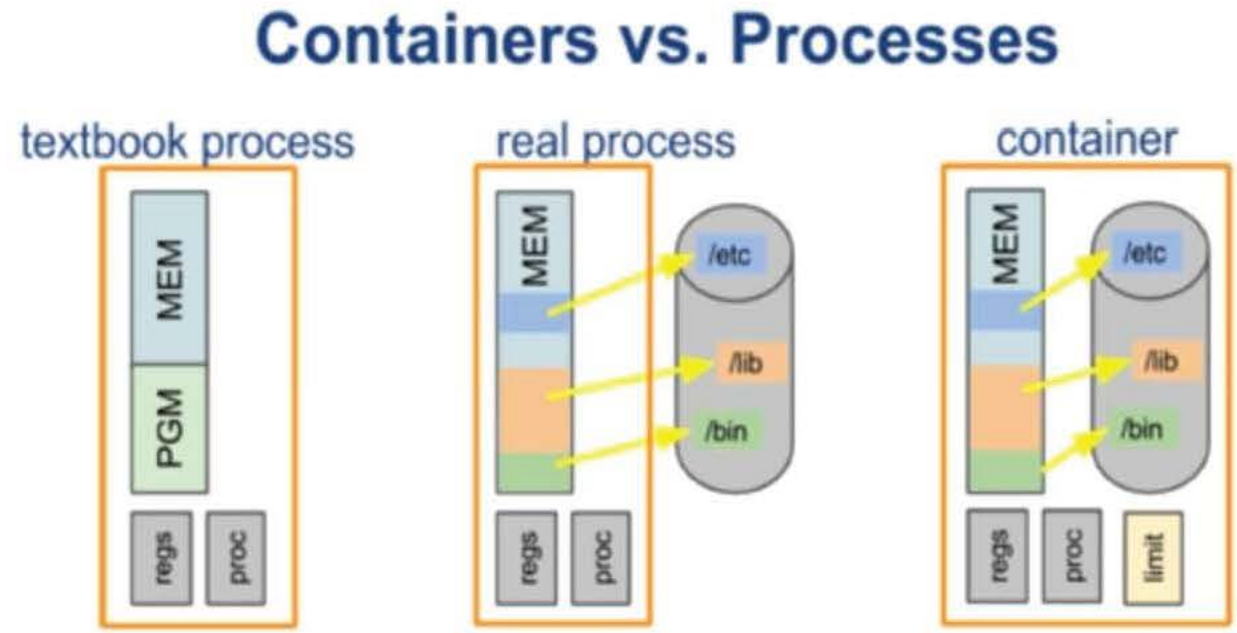
Can't

- Use the host kernel
- Boot a different OS
- Have its own modules
- Doesn't need *init* as PID 1

Containers vs Processes

Containers are processes with their full environment.

- A computer science textbook will define a process as having its own address space, program, CPU state, and process table entry.
- The program text is actually memory mapped from the filesystem into the process address space and often consists of dozens of shared libraries in addition to the program itself, thus all these files are really part of the process.

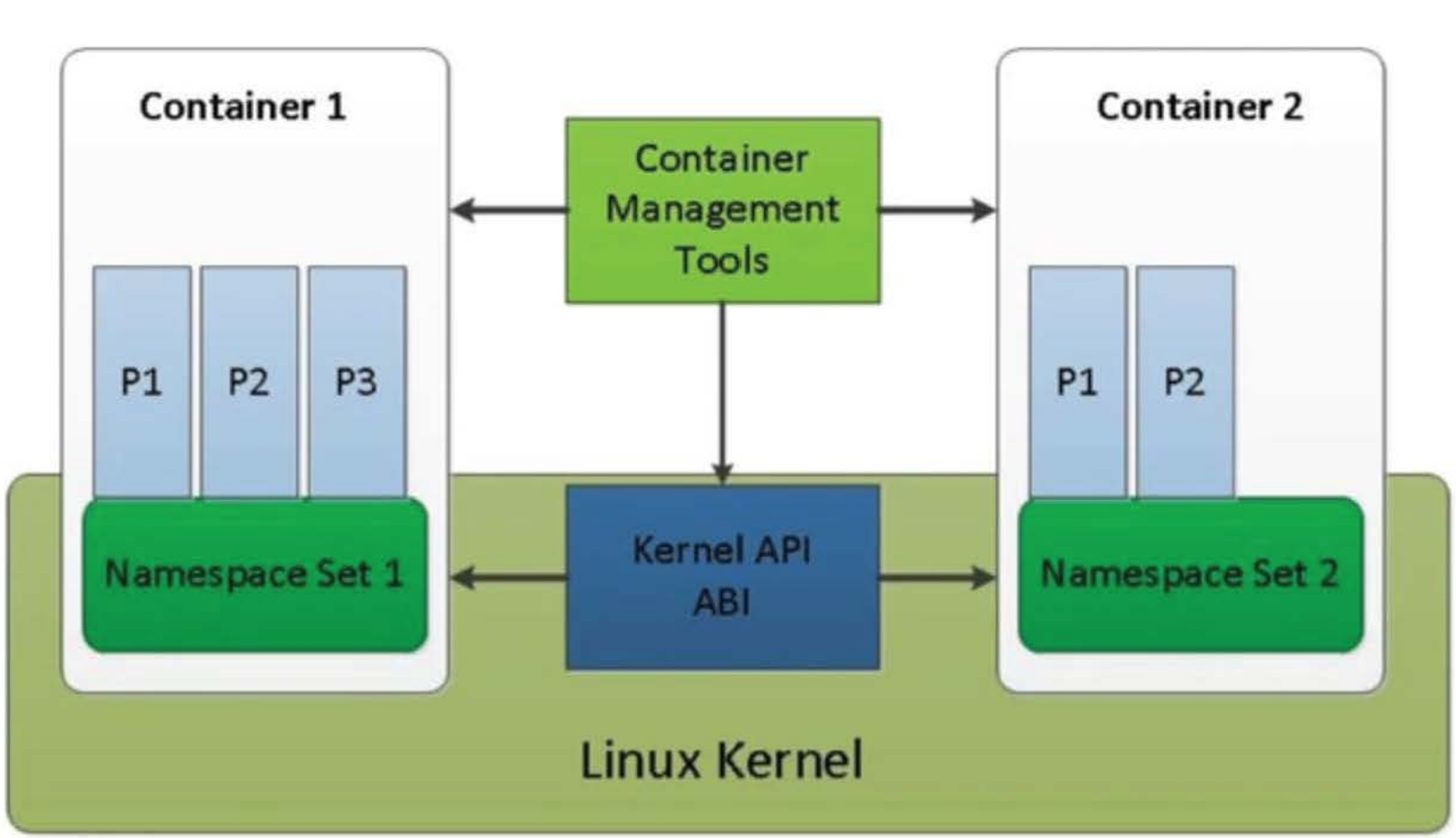


IoT: Cloud Services

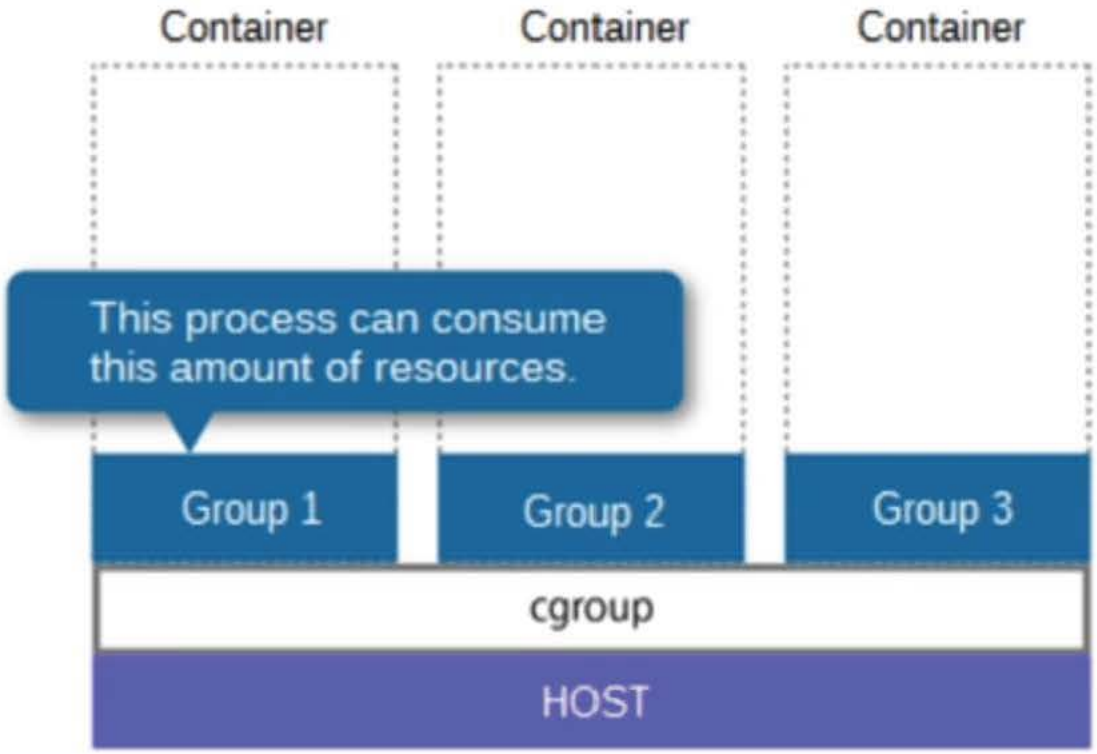
Containers - behind the hood

LinuX Containers (LXC)

- Operation System level virtualization method for Linux
- LXC (**L**inu**X** **C**ontainers) is an OS virtual for running isolated Linux systems (containers) on a single control host



Is containers a new thing?



Cgroups is a mechanism to control resources per hierarchical groups of processes

cgroups (abbreviated from **control groups**) is a Linux kernel feature to limit, account, and isolate resource usage (CPU, memory, disk I/O, etc.) of process groups.

- This work was started by engineers at Google in 2006 under the name "*process containers*".
- In late 2007, it was renamed to "Control Groups" due to the confusion caused by multiple meanings of the term "container" in the Linux kernel, and merged into kernel version 2.6.24.
- Since then, new features and controllers have been added.

Cgroups design goals

- Provide a unified interface to many different use cases, from controlling single processes to whole operating system-level virtualization
- Cgroups provides:
  - ***Resource limitation:*** groups can be set to not exceed a set memory limit — this also includes file system cache.
    - The original paper was presented at Linux Symposium and can be found at ***Containers:*** *Challenges with the memory resource controller and its performance.*
- ***Prioritization****:* some groups may get a larger share of CPU or disk I/O throughput.
- ***Accounting:*** to measure how much resources certain systems use for e.g. billing purposes.
- ***Control:*** freezing groups or check-pointing and restarting.

Namespaces

The purpose of each namespace is to wrap a particular global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource.

Cgroups vs namespaces

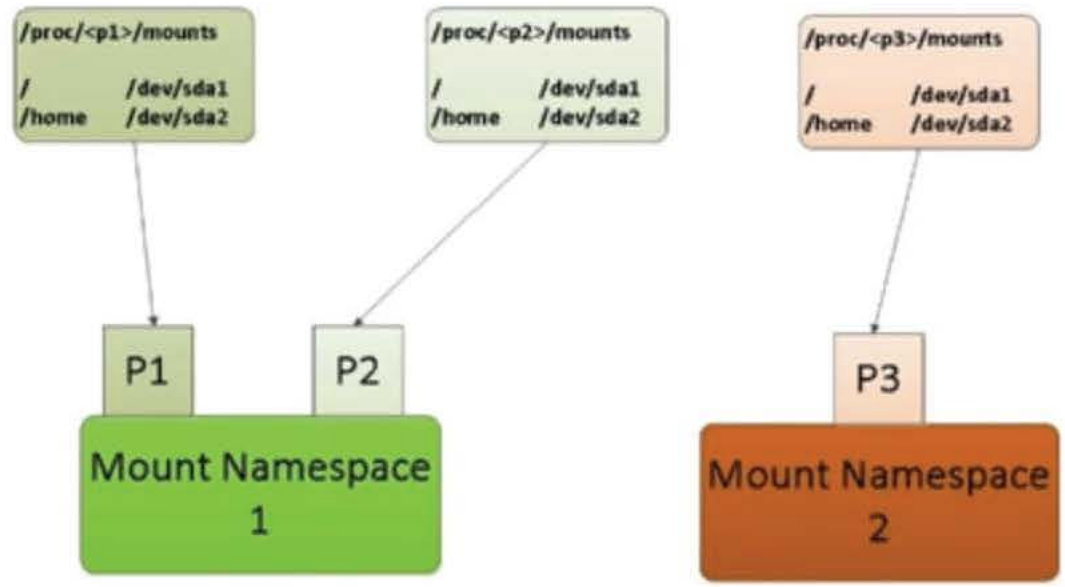
- **Cgroups**: limits how much you can use
- **Namespaces**: limits how much you can see (and use)

Namespaces

- mnt
- uts
- ipc
- net
- pid
- usr

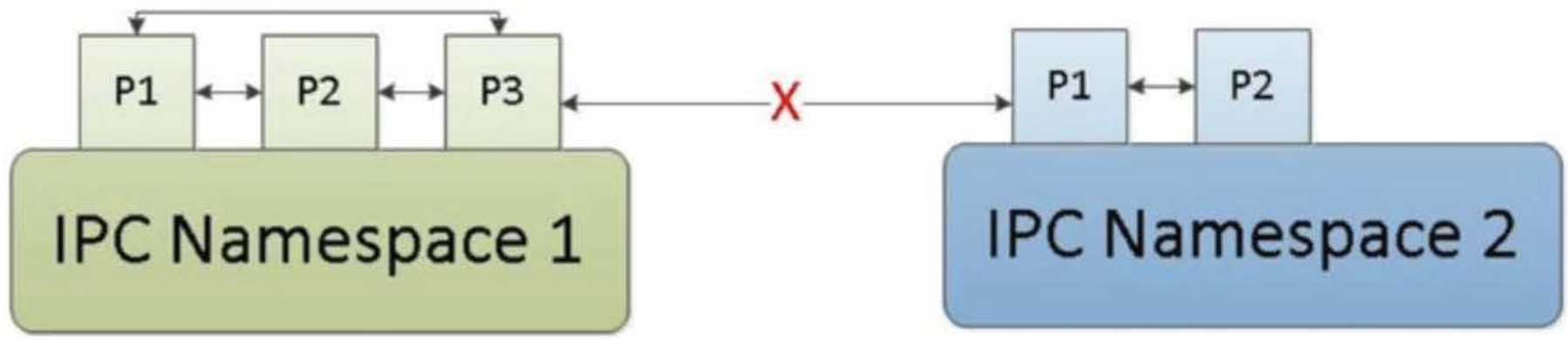
Mount namespace

- The “mount namespace” isolates the set of filesystem mount points seen by a group of processes
  - processes in a different mount namespaces can have different view of the filesystem hierarchy.
- Mounts can be private or shared
- The tasks running in each mount namespace doing mount/unmount will not affect the file system layout of the other mount namespace.



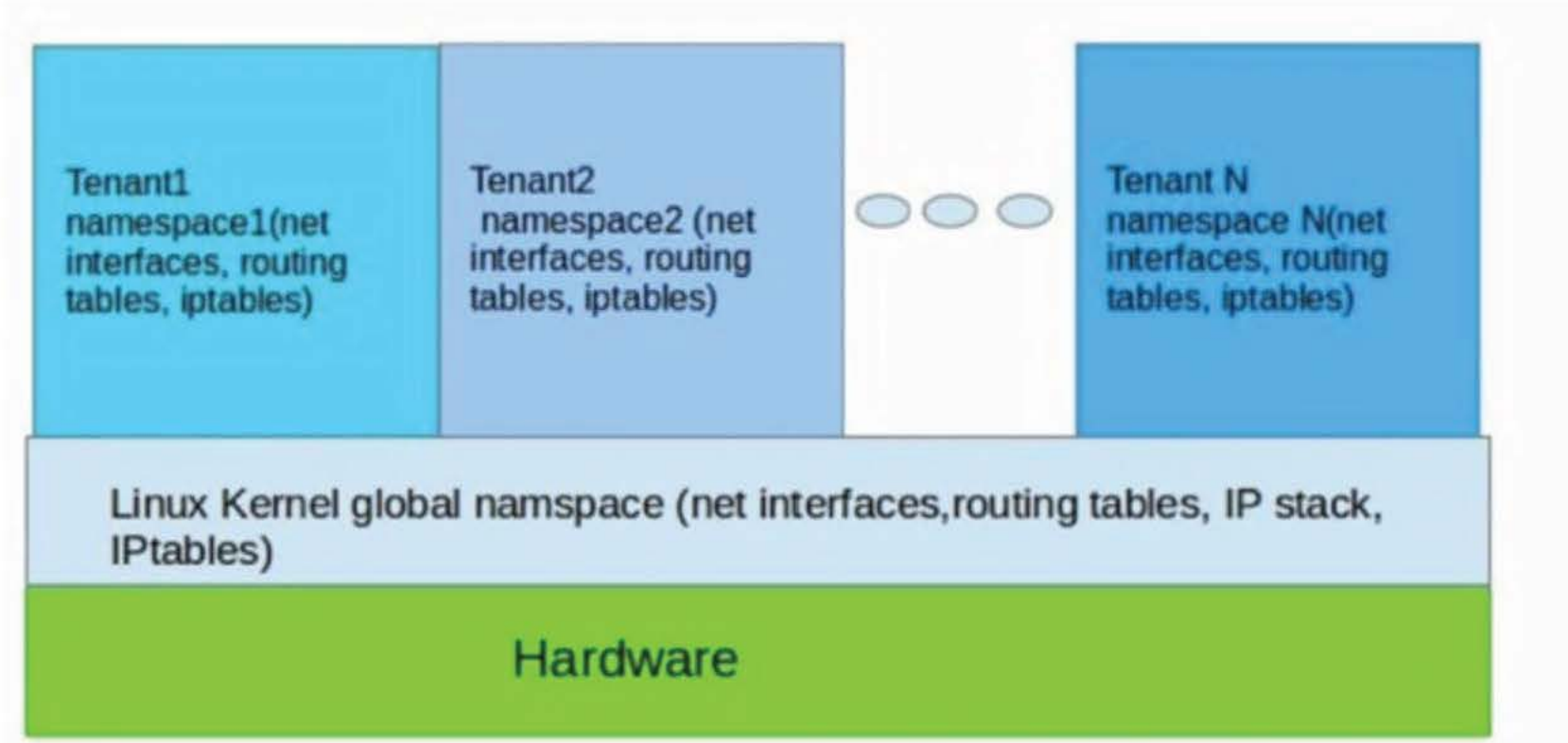
IPC Namespace

- IPC namespace isolates the inter-process communication resources.
- A process or a group of processes have their own:
  - IPC shared memory
  - IPC semaphore
  - IPC message queues



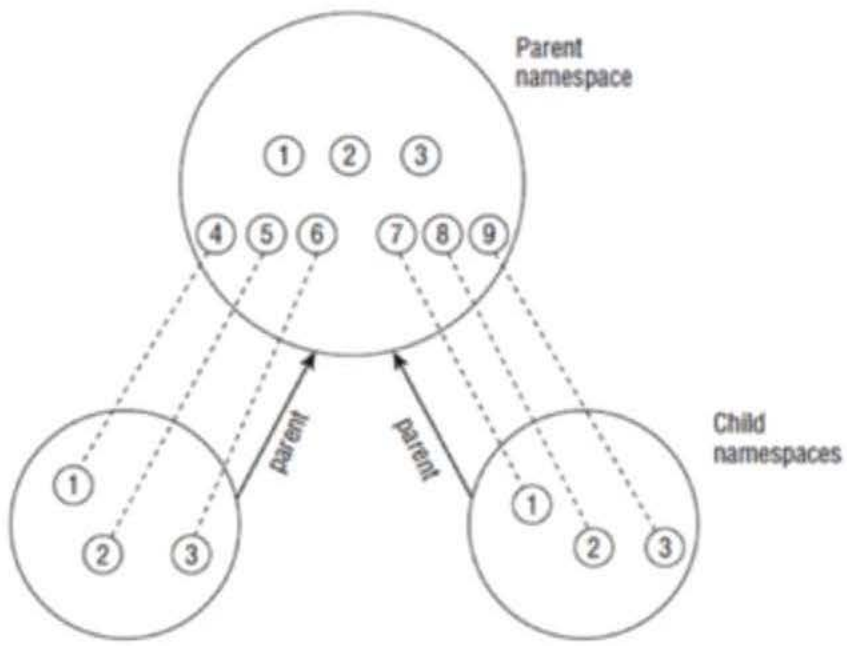
Network Namespace

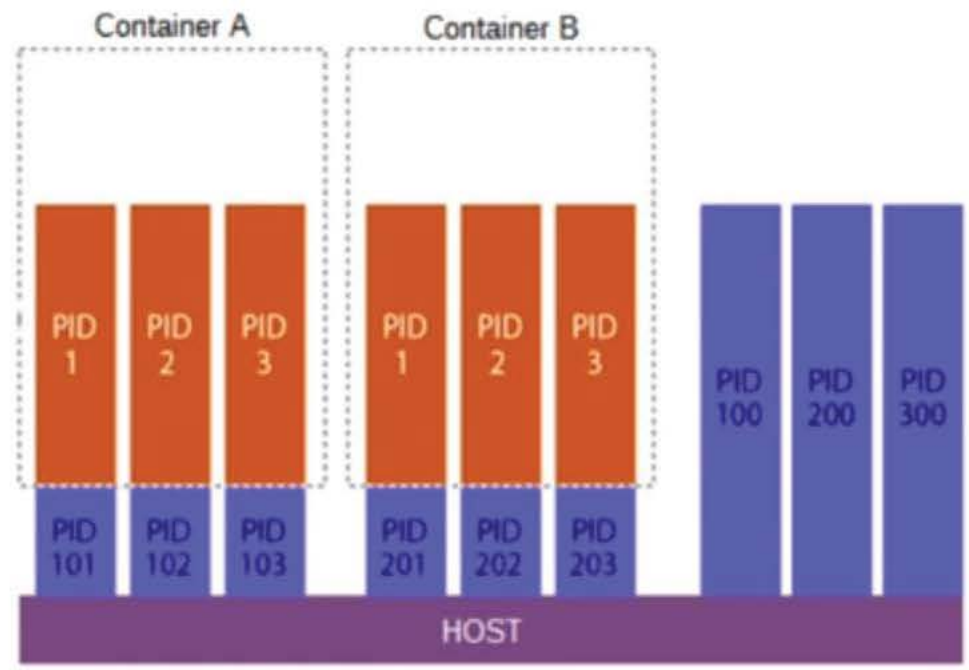
Net namespace isolates the networking related resources
Every net namespace has independent
    Network devices, IP addresses, firewall rules, routing table, sockets etc.



PID Namespace

- PID namespace isolates the Process ID, implemented as a hierarchy.
- PID namespace is a hierarchy comprised of “Parent” and “Child”.
  - The parent pid-ns can have many children pid-ns. Each Child pid-ns only has one parent pidns.





Other Namespaces

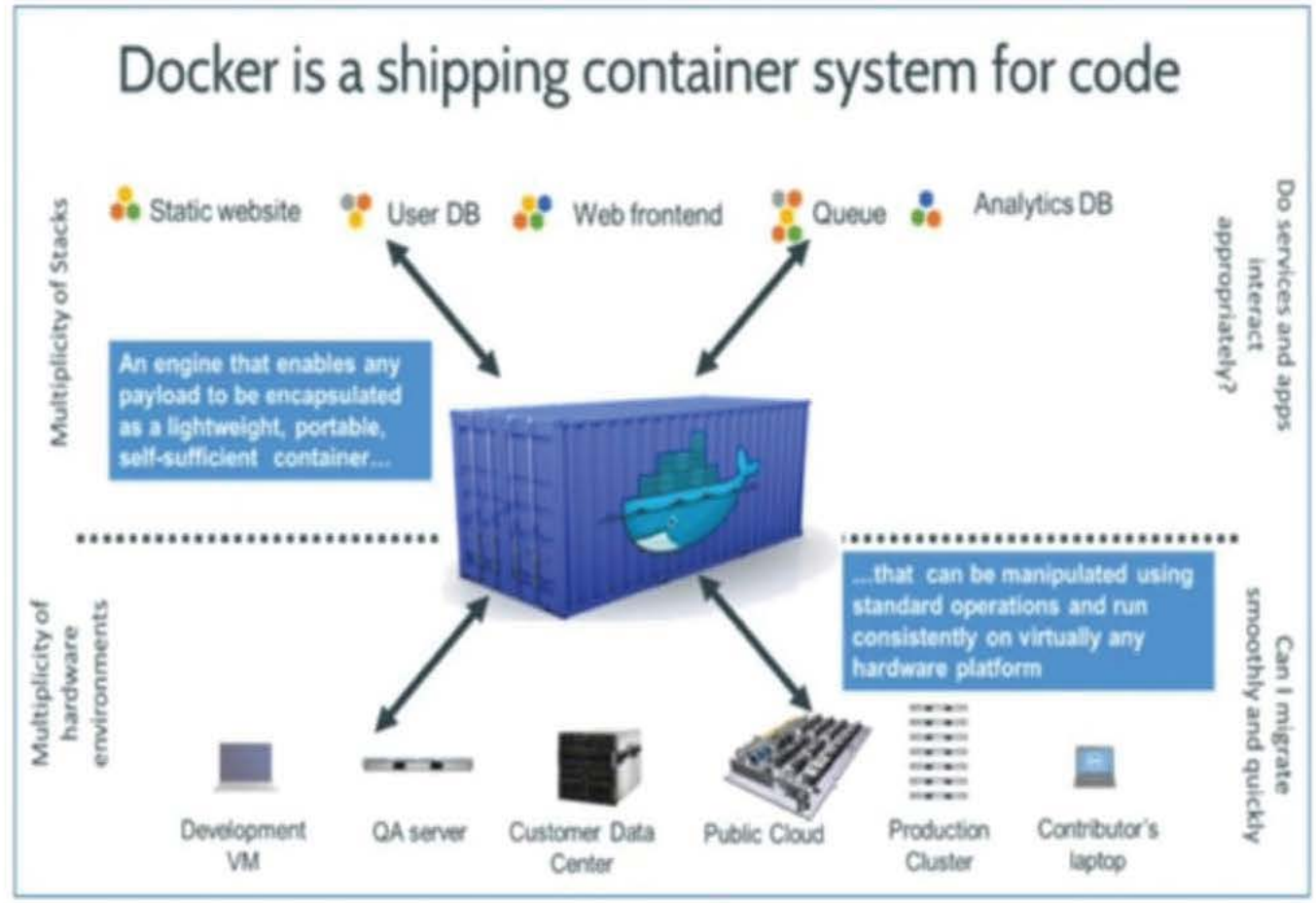
- UTS namespace
  - UTS = UNIX Time-sharing System
  - Each process has a separate copy of the hostname and the (now mostly unused) NIS domain name isolation!
  - In containers: useful for init and config scripts that tailor their actions based on the names
- User namespace
  - kuid/kgid: Original uid/gid, Global
  - uid/gid: user id in user namespace, will be translated to kuid/kgid finally
  - Only parent User NS has rights to set map

IoT: Cloud Services

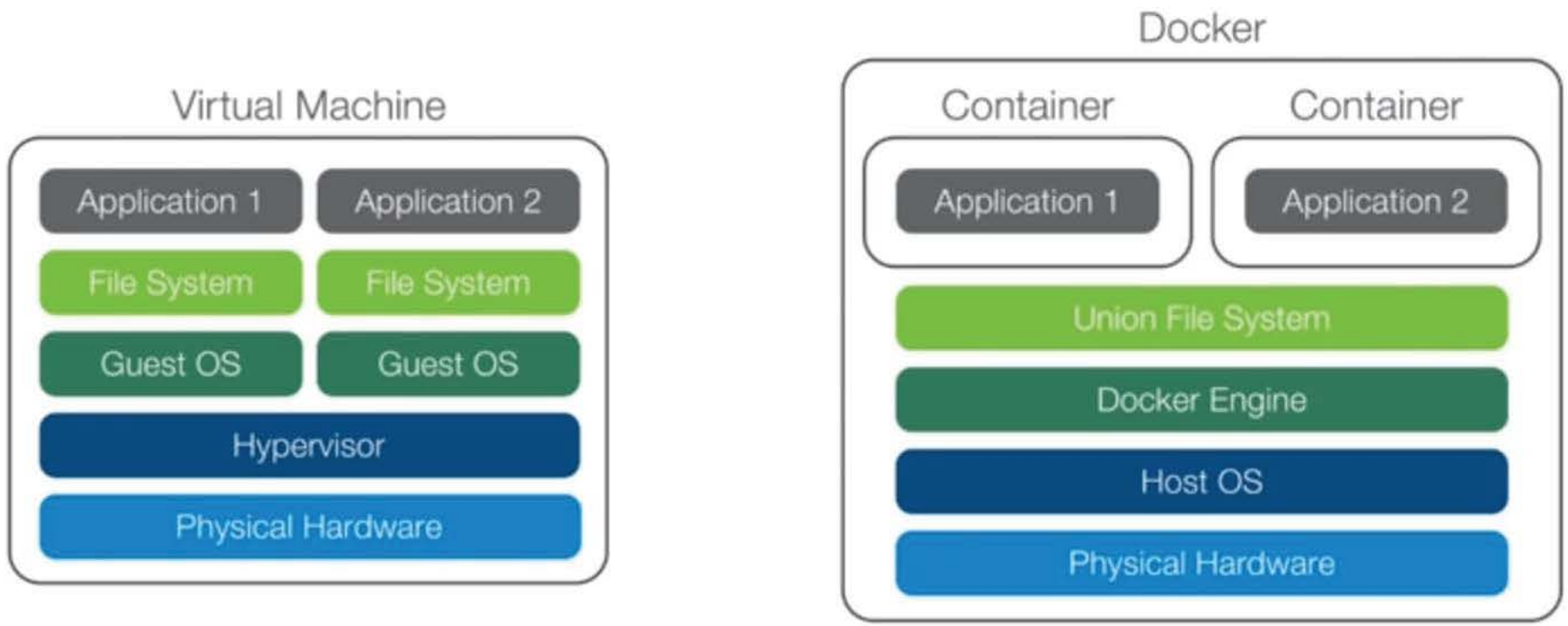
Docker

So what is Docker?

Docker is an open-source project that automates the deployment of applications inside software containers, by providing an additional layer of abstraction and automation of operating system–level virtualization on Linux.



VMs vs Containers vs Docker



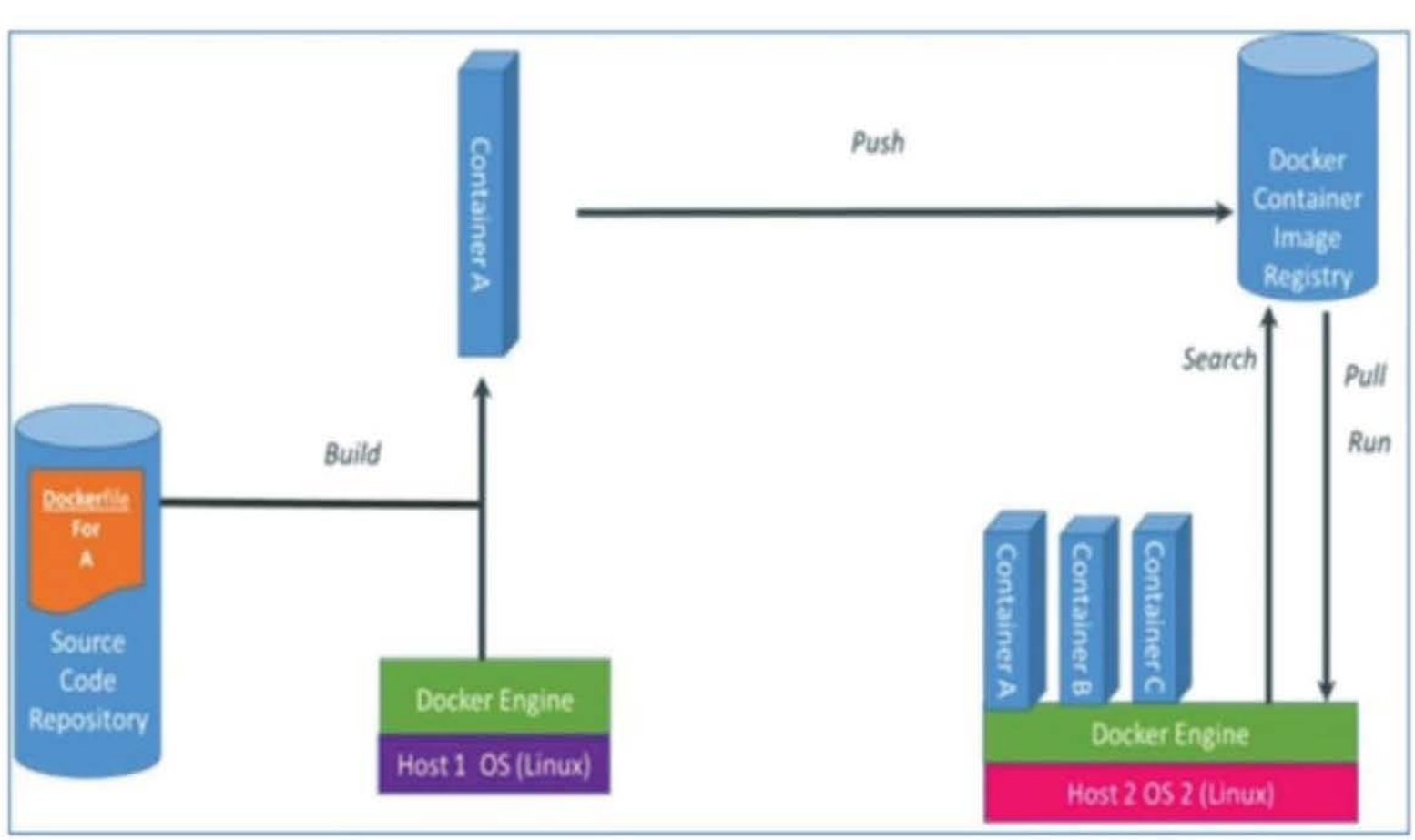
Why Docker?

Docker makes it easy to build, modify, publish, search, and run containers.

- With Docker, a container comprises both an application and all of its dependencies.
- Containers can either be created manually or, if a source code repository contains a DockerFile, automatically.
- Subsequent modifications to a baseline Docker image can be committed to a new container using the Docker Commit Function and then Pushed to a Central Registry.

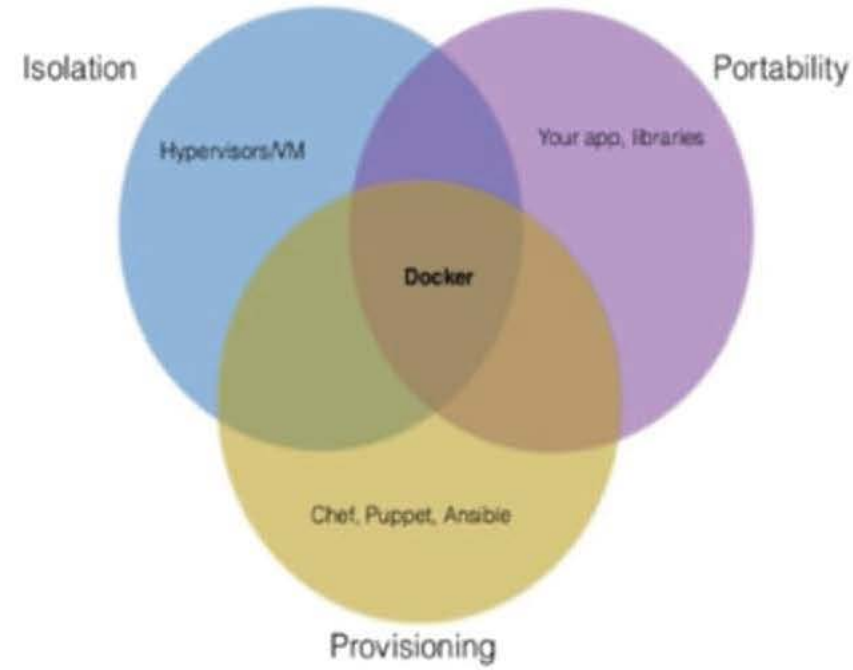
How Docker Works

- Containers can be found in a Docker Registry (either public or private), using Docker Search.
- Containers can be pulled from the registry using Docker Pull and can be run, started, stopped, etc.



Docker use cases

- Development Environment
- Environments for Integration Tests
- Quick evaluation of software
- Microservices
- Multi-Tenancy
- Unified execution environment (dev test to prod (local, VM, cloud, ...)



IoT: Cloud Services

Orchestration & Kubernetes

Container Orchestration

- Orchestrate containers to facilitate microservices architecture
- Provision containers
- Manage container dependencies
- Enable discovery
- Handle container failure
- Scale Containers

Why one to choose?

Kubernetes

Kubernetes is based on Google's and Red Hat's experience of many years working with Linux containers.

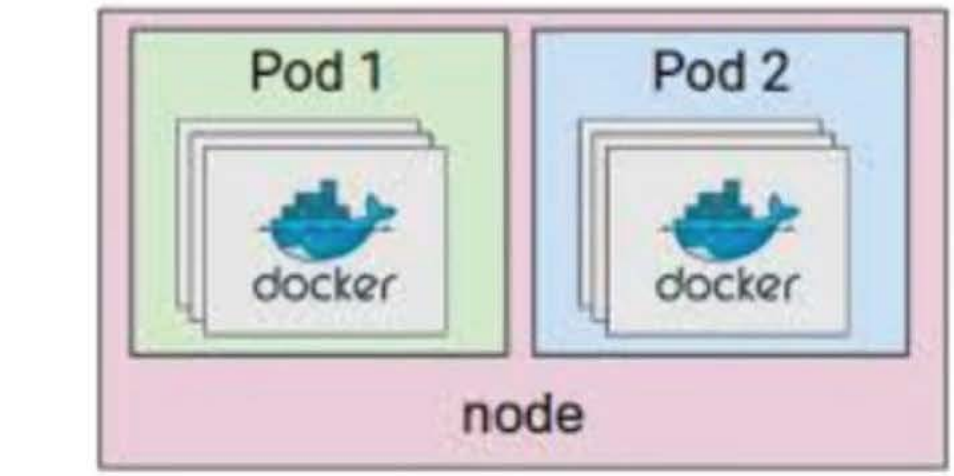
Some abilities:

- Mount persistent volumes that allows to move containers without loosing data,
- it has load balancer integrated
- it uses `Etcd` for service discovery.

Kubernetes uses a different CLI, different API and different YAML definitions.

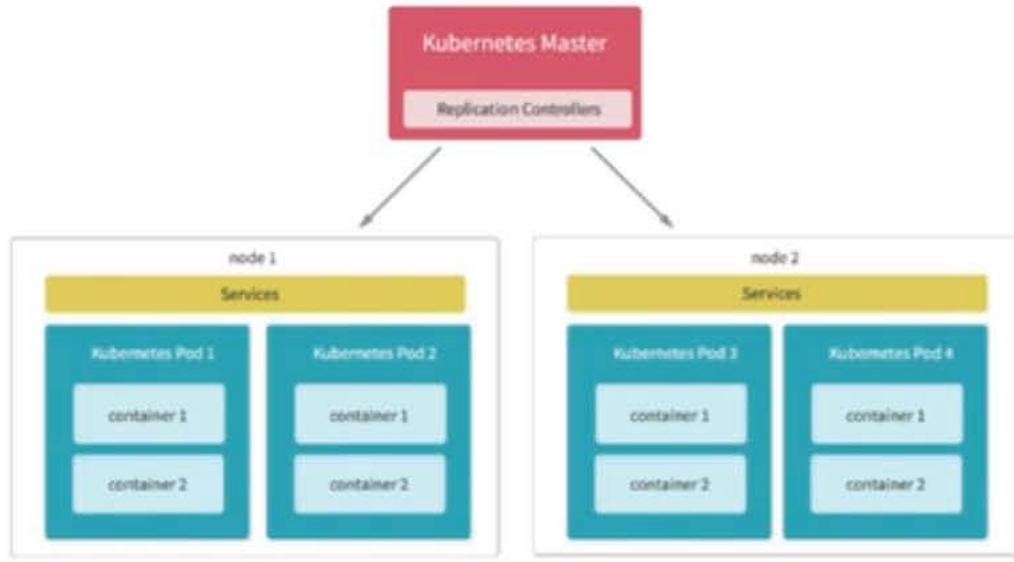
- cannot use Docker CLI

Pods



- ***Pods***:
  - Pods are groups of containers that are deployed and scheduled together.
  - A pod will typically include 1 to 5 containers which work together to provide a service.
  - Kubernetes will run other containers to provide logging and monitoring services.
  - Pods are treated as ephemeral in Kubernetes;

Kubernetes: Services & Replication Controllers



- ***Services***
  - Services are stable endpoints that can be addressed by name.
  - Services can be connected to pods by using label selectors; for example my “DB” service may connect to several “C*” pods identified by the label selector “type”: “Cassandra”.
  - The service will automatically round-robin requests between the pods.

- **Replication Controllers**
  - Replication controllers are used to instantiate pods in Kubernetes
  - They control and monitor the number of running pods (called replicas) for a service.

Kubernetes: Net space and Labels

- ***Flat Networking Space***:
  - Containers within a pod share an IP address, but the address space is “flat” across all pods,
    - all pods can talk to each other without any Network Address Translation (NAT).
  - This makes multi-host clusters much more easy to manage, at the cost of not supporting links and making single host (or, more accurately, single pod) networking a little more tricky.
  - As containers in the same pod share an IP, they can communicate by using ports on the localhost address.
- ***Labels***:
  - Labels are key-value pairs attached to objects in Kubernetes, primarily pods, used to describe identifying characteristics of the object
    - e.g. version: dev and tier: frontend.
  - Labels are not normally unique; they are expected to identify groups of containers.
  - Label selectors can then be used to identify objects or groups of objects, for example all the pods in the frontend tier with environment set to production.

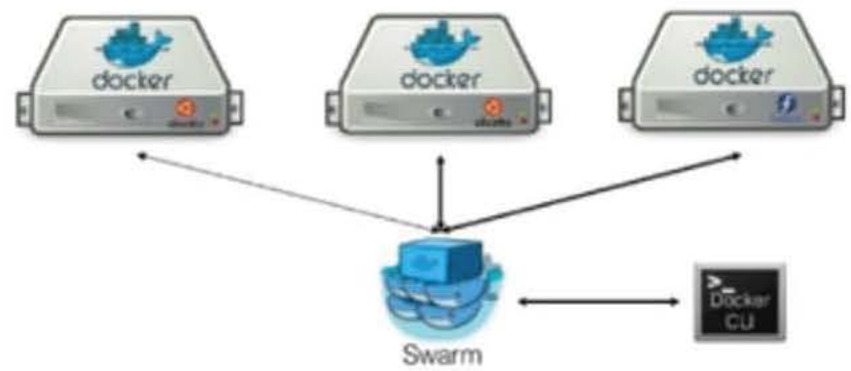
IoT: Cloud Services

Orchestration (next)

Docker Swarm

- A native clustering for Docker.
- Exposes standard Docker API
  - meaning that any tool that you used to communicate with Docker (Docker CLI, Docker Compose, Dokku, Krane, and so on) can work equally well with Docker Swarm.
- Bound by the limitations of Docker API

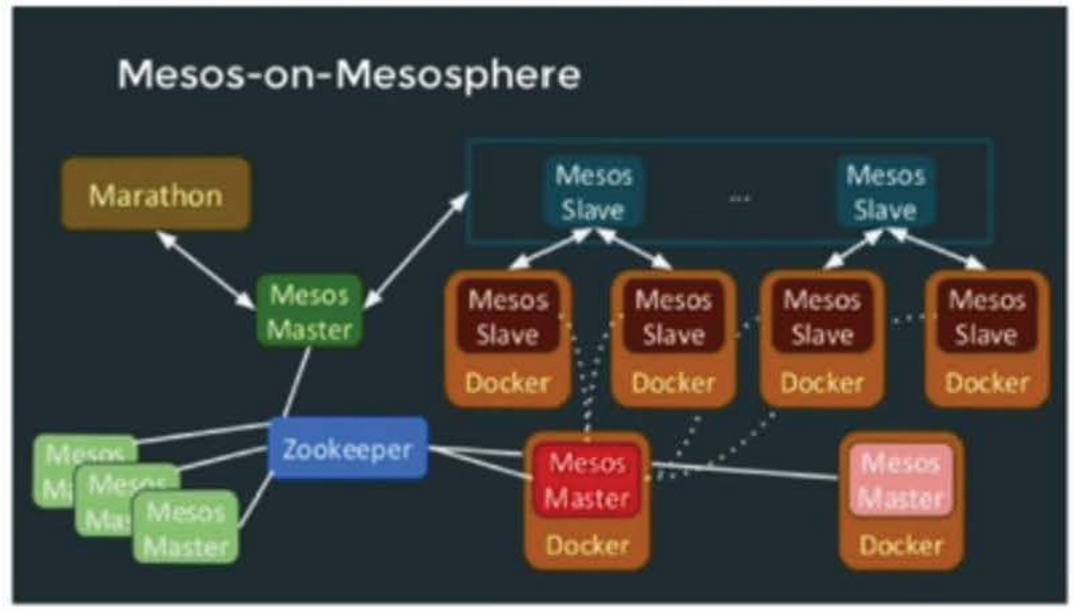
With Docker Swarm



Docker Swarm: Architecture

- Architecture:
  - each host runs a Swarm ***agent*** and one host runs a Swarm ***manager*** (on small test clusters this host may also run an agent).
  - The manager is responsible for the orchestration and scheduling of containers on the hosts.
  - Swarm can be run in a high-availability mode where one of etcd, Consul or ZooKeeper is used to handle fail-over to a back-up manager.
  - There are several different methods for how hosts are found and added to a cluster, which is known as *discovery* in Swarm.
    - By default, *token* based discovery is used, where the addresses of hosts are kept in a list stored on the Docker Hub.

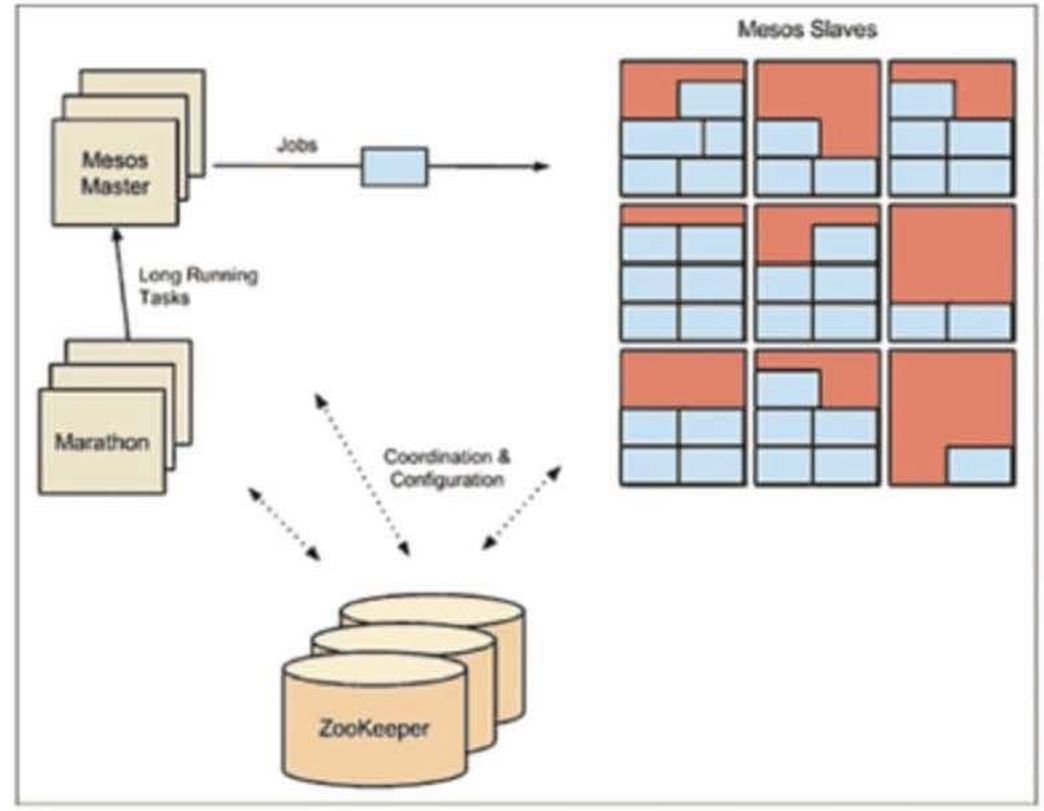
Mesos



- Apache Mesos (https://mesos.apache.org) is an open-source cluster manager. It's designed to scale to very large clusters involving hundreds or thousands of hosts.
  - Mesos supports diverse workloads from multiple tenants; one user's Docker containers may be running next to another user's Hadoop tasks.
- Apache Mesos was started as a project at the University of Berkeley before becoming the underlying infrastructure used to power Twitter and an important tool at many major companies such as eBay and Airbnb.
- With Mesos, we can run many frameworks simultaneously:
  - Marathon and Chronos are the most well known

Mesos Architecture

- ***Mesos Agent Nodes*** – Responsible for actually running tasks. All agents submit a list of their available resources to the master.
  - There will typically be 10s to 1000s of agent nodes.
- ***Mesos Master*** – The master is responsible for sending tasks to the agents.
  - maintains a list of available resources and makes "offers" of them to frameworks.
  - decides how many resources to offer based on an allocation strategy. There will typically be 2 or 4 stand-by masters ready to take over in case of a failure.
- ***ZooKeeper*** – Used in elections and for looking up address of current master.
  - Typically 3 or 5 ZooKeeper instances will be running to ensure availability and handle failures.



Mesos with Marathon: Architecture

- ***Frameworks*** – Frameworks co-ordinate with the master to schedule tasks onto agent nodes. Frameworks are composed of:
  - *executor* process which runs on the agents and takes care of running the tasks
  - *scheduler* which registers with the master and selects which resources to use based on offers from the master.
  - There may be multiple frameworks running on a Mesos cluster for different kinds of task. Users wishing to submit jobs interact with frameworks rather than directly with Mesos.
- **Marathon** is designed to start, monitor and scale long-running applications.
  - Marathon is designed to be flexible about the applications it launches,
    - It can even be used to start other complementary frameworks such Chronos ("cron" for the datacenter).
  - It makes a good choice of framework for running Docker containers, which are directly supported in Marathon.
  - Marathon supports various affinity and constraint rules.
  - Clients interact with Marathon through a REST API.
  - Other features include support for health checks and an event stream that can be used to integrate with load-balancers or for analyzing metrics.

IoT: Cloud Services

Container Conclusion

Containers Pros

Containers are immutable

The OS, library versions, configurations, folders, and application are all wrapped inside the container.

Guarantee that the same image that was tested in QA will reach the production environment with the same behavior.

Containers are lightweight

The memory footprint of a container is small.

Instead of hundreds or thousands of MBs, the container will only allocate the memory for the main process.

Containers are fast

Can start a container as fast as a typical Linux process takes to start.

Instead of minutes, you can start a new container in few seconds.

However, many users are still treating containers just like typical virtual machines and forget that containers have an important characteristic: **Containers are disposable**.

Things to avoid in Docker containers

- Do not store persistent data in containers:
  - containers can be stopped, destroyed, replaced.
- Do not ship the app in pieces:
- Do not create large images
  - Large image is harder to distribute
  - Avoid “updates” (e.g. `yum update`) that downloads may files to a new image
- Do not use single layer image
  - To make an effective layered file system, always create your base image layer for your OS, another layer for the username definitions, another layer for the runtime installation, and another layer for the configuration, and another layer for the app.
  - Makes it easier to recreate, manage and distribute the image
- Do not create images from running containers
  - E.g. do not use “`docker commit`” to create an image. This method is not reproducible

References and Acknowledgements

- Some images have been borrowed from: : http://scm.zoomquiet.io/data/20131004215734/index.html
- Docker Engine and Docker Hub: http://blog.docker.com/category/registry-2/
- Microservices: http://martinfowler.com/articles/microservices.html
- Swarm vs Fleet vs Kubernetes vs Mesos: https://www.oreilly.com/ideas/swarm-v-fleet-v-kubernetes-v-mesos
- 10 things to avoid in docker containers: http://developers.redhat.com/blog/2016/02/24/10-things-to-avoid-in-docker-containers/
- There are more public information that have been used…

IoT: Cloud Services

Serverless

What is serverless?

- Serverless architectures refer to applications that significantly depend on
  - third-party services (knows as Backend as a Service or "BaaS"), or
  - custom code that's run in ephemeral containers (Function as a Service or "FaaS")

Mobile Backend as a Service (MBaaS)

- Serverless was first used to describe applications that significantly or fully depend on 3rd party applications / services (‘in the cloud’) to manage server-side logic and state.
- Typically ‘*rich client*’ applications (think single page web apps, or mobile apps) that use the vast ecosystem of cloud accessible services
  - Think of single page web apps, or mobile apps that use the vast ecosystem of cloud accessible databases (like Parse, Firebase), authentication services (Auth0, AWS Cognito), etc

Functions as a Service (FaaS)

- Apps where some amount of server-side logic is still written by the application developer but unlike traditional architectures is run in stateless compute containers that are
  - event-triggered,
  - ephemeral (may only last for one invocation),
  - and fully managed by a 3rd party.
- Building and supporting a ‘Serverless’ application is not looking after the hardware or the processes
  - they are outsourced to a vendor.

Serverless Vendors

- Amazon Lambda - Run code without thinking about servers. Pay for only the compute time you consume.
- Google Cloud Functions - Lightweight, event-based, asynchronous compute solution that allows you to create small, single-purpose functions that respond to cloud events without the need to manage a server or a runtime environment.
- Azure Functions - Listen and react to events across your stack.
- IBM OpenWhisk - Distributed compute service to execute application logic in response to events.
- And many others…
  - A good list can be found: https://github.com/anaibol/awesome-serverless

IoT: Cloud Services

Deeper on Serverless

Serverless main concept

- The phrase “*serverless*” doesn’t mean servers are no longer involved.
- It simply means that developers no longer have to think "that much" about them.
- Computing resources get used as services without having to manage around physical capacities or limits.
- Serverless allows you to NOT think about servers.
  - Which means you no longer have to deal with over/under capacity, deployments, scaling and fault tolerance, OS or language updates, metrics, and logging.

Serverless vs Containers

- Take containers as an example. When you deploy a Docker container to Amazon ECS you still need to think about the hosting Cluster that your container will run on. You need to consider such questions as:
- Which Cluster would be best placed to run this container?
- Does the Cluster have capacity for my container's resource needs (CPU, memory)? If not how should I expand it?
- What is my strategy for deploying multiple instances of the container across multiple machines in the Cluster?
- If the Cluster has multiple types of machine within it, do I need to be concerned about that when I choose my deployment strategy?
- What are the security constraints of the Cluster, and do they need to be changed in order to properly host my container?

Benefits of Serverless

- The hosting provider
  - figures out the all the allocate questions for you dynamically
  - Guarantees it will have sufficient capacity for your needs
- Do not have to spend money upfront over-provisioning your host environment
- Not being constrained down the road by under-provisioning the environment.
- Not paying for idle
- Reliability and Availability are build in.

The Idea

- By composing and combining different services together in a loose orchestration developers can now build complex systems very quickly and spend most of their time focusing on their core business problem.
- These serverless systems can scale, grow and evolve without developers or solution architects having to worry about remembering to patch that web server yet again.
- A good serverless architecture can speed up development time and help to produce a more robust product.

IoT: Cloud Services

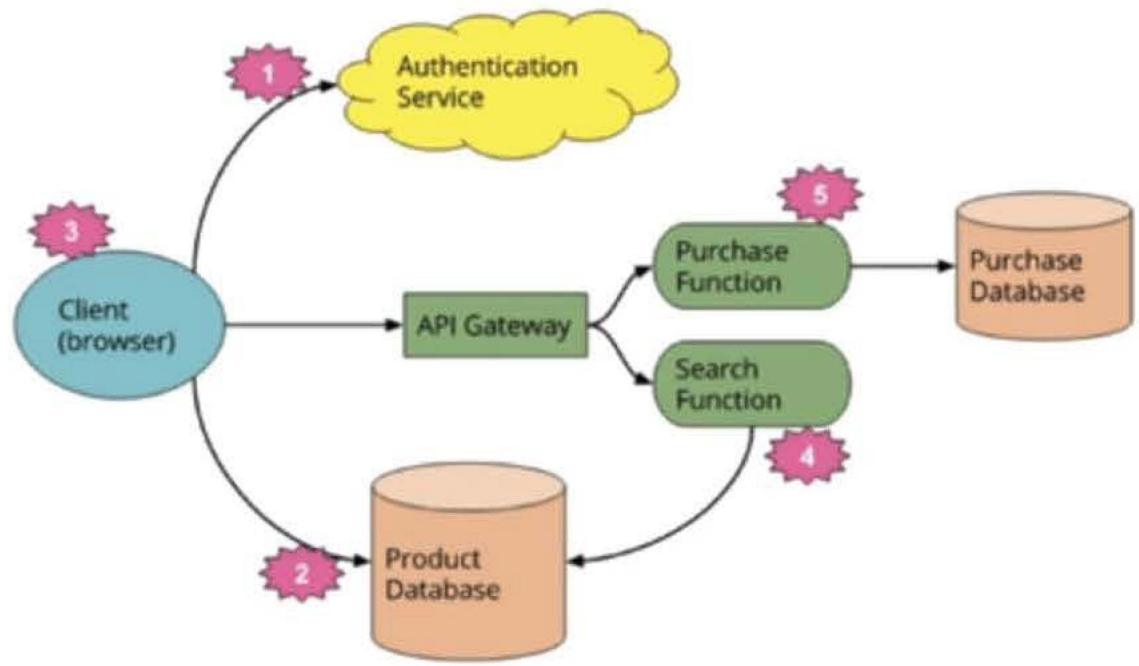
Serverless Examples

UI Driven app: The classic design

- A traditional 3-tier client-oriented system with server-side logic. A good example is a typical ecommerce app (dare I say an online pet store?).
- For example implemented in Java on the server side, with a HTML / Javascript component as the client
- With this architecture the client can be relatively unintelligent, with much of the logic in the system - authentication, page navigation, searching, transactions - implemented by the server application.

Serverless redesign of the Pet Store

- Deleted the authentication logic replaced it with a third party BaaS service.
- Client can have direct access to a subset of our database (for product listings), which itself is fully 3rd party hosted (e.g. AWS Dynamo.)
- Likely to have a different security profile for the client accessing the database in this way from any server resources that may access the database.

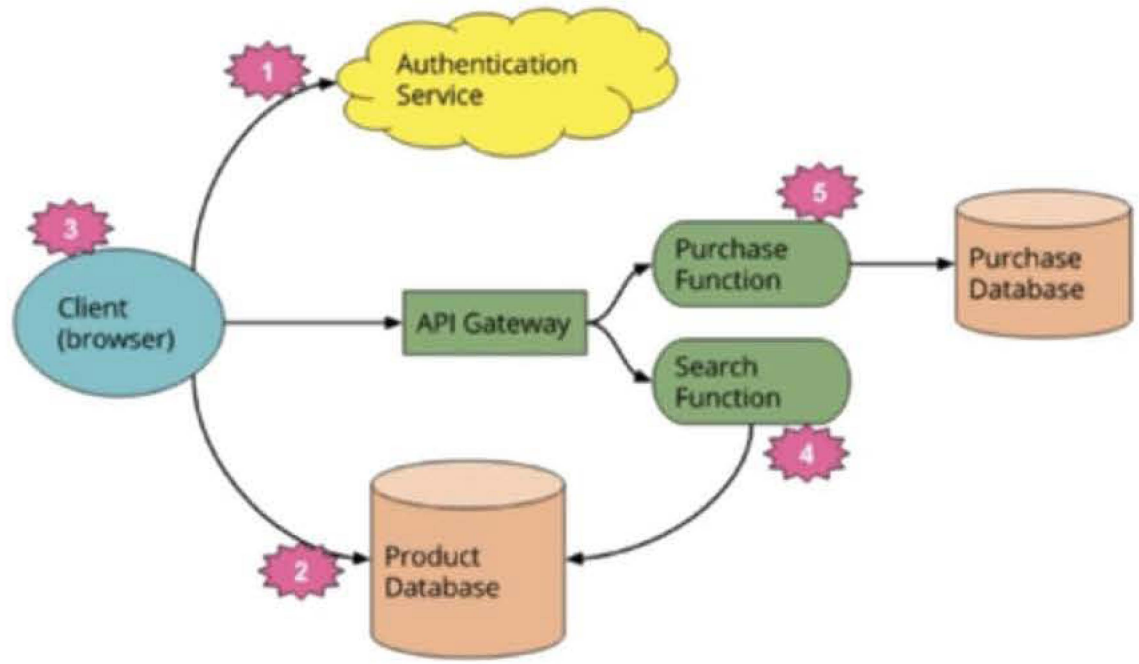


UI Driven app: The classic design

- A traditional 3-tier client-oriented system with server-side logic. A good example is a typical ecommerce app (dare I say an online pet store?).
- For example implemented in Java on the server side, with a HTML / Javascript component as the client
- With this architecture the client can be relatively unintelligent, with much of the logic in the system - authentication, page navigation, searching, transactions - implemented by the server application.

Serverless redesign of the Pet Store

- Deleted the authentication logic replaced it with a third party BaaS service.
- Client can have direct access to a subset of our database (for product listings), which itself is fully 3rd party hosted (e.g. AWS Dynamo.)
  - Likely to have a different security profile for the client accessing the database in this way from any server resources that may access the database.



A serverless view

- These previous two points imply a very important third - some logic that was in the Pet Store server is now within the client,
  - E.g.: keeping track of a user session, understanding the UX structure of the application (e.g. page navigation), reading from a database and translating that into a usable view, etc. The client is in fact well on its way to becoming a Single Page Application.
- Some UX related functionality can be kept in the server,
  - if it's compute intensive or requires access to significant amounts of data. An example here is 'search'.
  - For the search feature instead of having an always-running server we can implement a FaaS function that responds to http requests via an API Gateway (described later.) We can have both the client, and the server function, read from the same database for product data.

Another view

- Since the original server was implemented in Java, and AWS Lambda (our FaaS vendor of choice in this instance) supports functions implemented in Java, we can port the search code from the Pet Store server to the Pet Store Search function without a complete re-write.
- Finally we may replace our ‘purchase’ functionality with another FaaS function, choosing to keep it on the the server-side for security reasons, rather than re-implement it in the client. It too is fronted by API Gateway.