Cloud-Native Applications
(Micro-service oriented)

Cloud-Native Applications: Motivation

- Elasticity and Ubiquity of Cloud Infrastructure (Data-Center-scale Computing) have enabled new generation of applications, use-cases and business opportunities:
  - Netflix, Airbnb, Spotify, Pinterest, Snapchat, Whatspp,…
- Example: Netflix:
  - Value Proposition (Competitive Advantage):
    - Low Cost Video Streaming with Superb User-experience at scale
  - Application Properties and Unique Requirements
    - > 100 millions of users in 190 countries (mostly in US)
    - Vast variance in Load, within minutes (evenings, campaigns, …)
    - 10,000s of servers
    - 1000s of daily application changes across 100s of functions
      - Video streaming, Catalog, Recommendations, subscription
      - ~1 update every minute

85

Cloud-Native Applications:
Requirements-driven Design Principles

- Low Cost + Variance in Load + Superb User Experience
  - Can't afford Over or Under Provisioning => Auto-Scaling Elasticity
- Large-scale Infrastructure + Inevitable Hardware Failures + Superb User Experience
  - Must accommodate HW failures w/o downtime => Design for Failure
- Frequent Application Updates + Large-scale + Superb User Experience
  - Can't afford redeploying everything everytime => Modularity
  - Can't afford testing everything everytime => Stable Internal APIs
- Frequent Application Updates + Low Cost + Superb User Experience
  - Can't afford manual QA/ admin effort for each update => Automation in Deployment

86

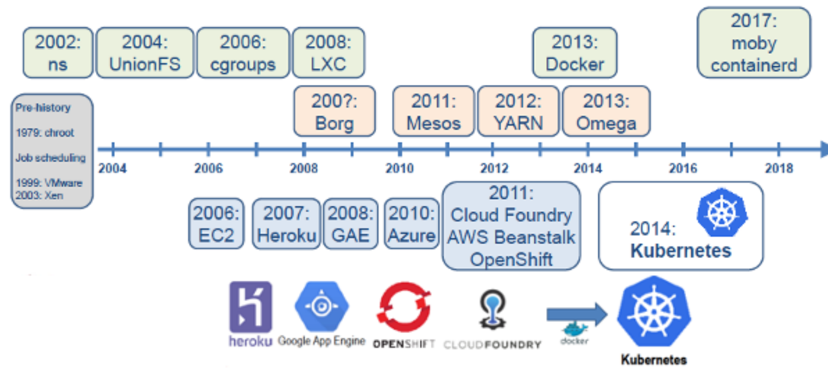
Cloud-Native Applications:
Design-driven Common Services

Design Principle	Common Service
Auto-Scaling	Horizontal Auto-Scaling
Auto-Scaling / Design for Failure	Elastic Load Balancing
Design for Failure	Replication
Design for Failure	Health Monitoring
Modularity	Decoupling into homogenous (micro)services*
Modularity	Unified packaging (across dev/test/prod) with Docker
API-driven composition	Discovery
API-driven composition	Routing
Automation	Fully programmable life cycle of components*
Automation	Observability (monitoring, tracing, etc)

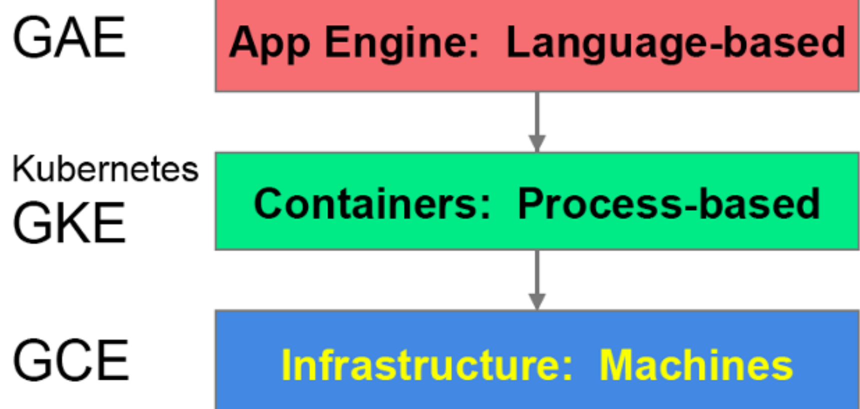
87

Evolution of Platforms for Cloud-Native Applications:

- New Platforms emerged, offering common services required by Cloud-Native Applications:
  - Auto-Scaling, Replication, Load-Balancing, Health Monitoring, Service Discovery, Application-Level Routing, Programmability
  - Started in form of “Platform as a Service” (PaaS)
  - Eventually generalized to the Container-based Orchestration approach



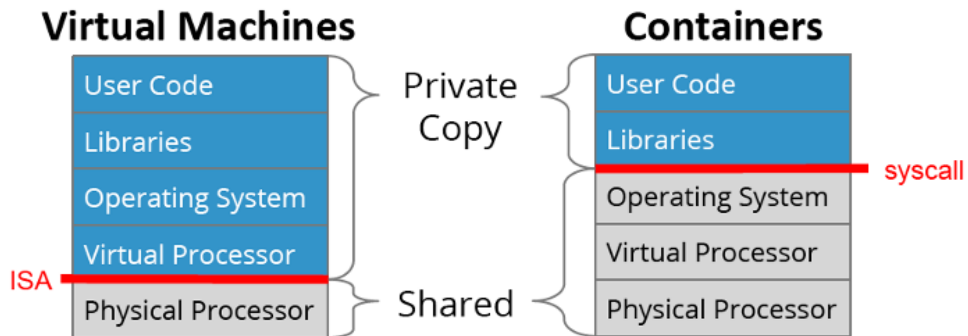
Different forms of
Cloud-based Computing Services/Offerings
from Google



Deploying Cloud-Native (Micro-service oriented) Applications

Container Technologies to our rescue !

VMs vs. Containers



Containers: less overhead, enable more “magic”

Virtual Machine vs. Container

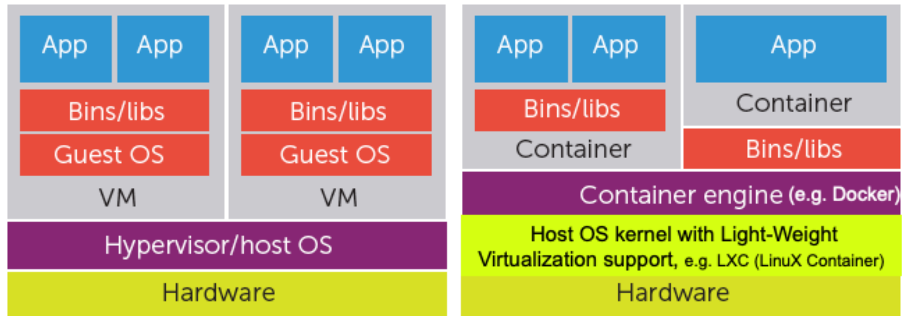
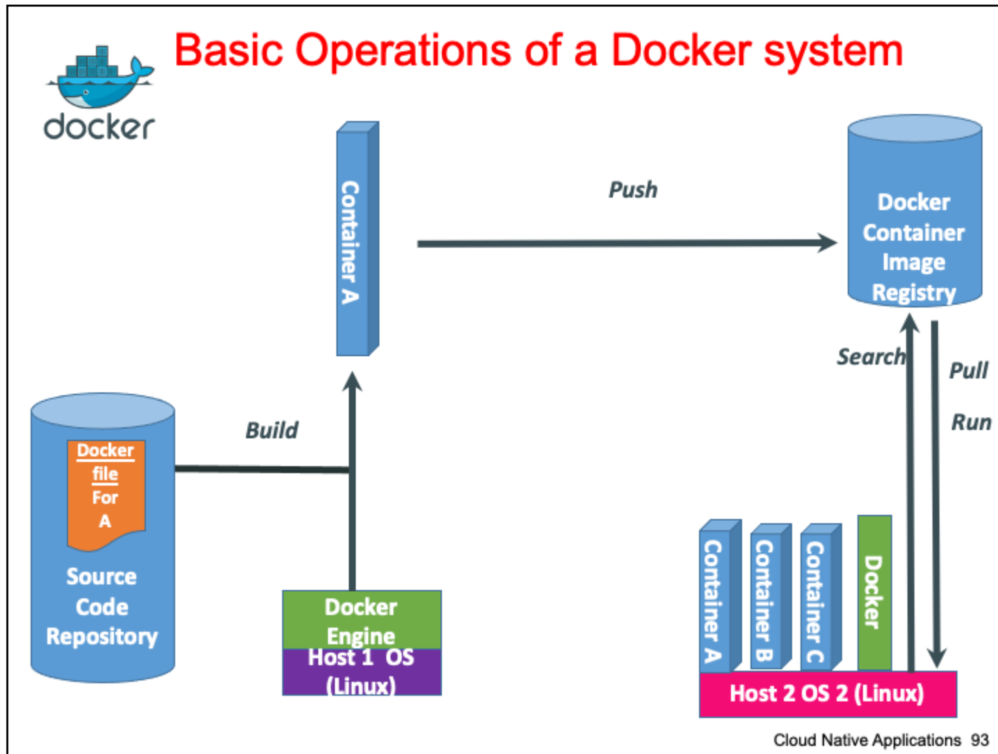


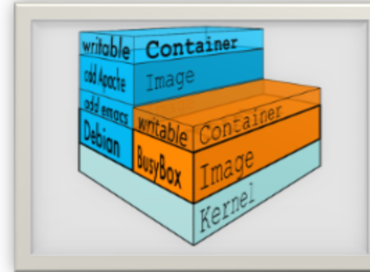
FIGURE 1. Virtualization architecture. The two possible scenarios, a traditional hypervisor architecture on the left and a container-based architecture on the right, differ in their management of guest operating system components.

Source: Claus Pahl, "Containerization and the PaaS Cloud," IEEE Cloud Computing Magazine, May/June 2015



https://www.digitalocean.com/community/tutorials/docker-explained-using-dockerfiles-to-automate-building-of-images

Docker Containers

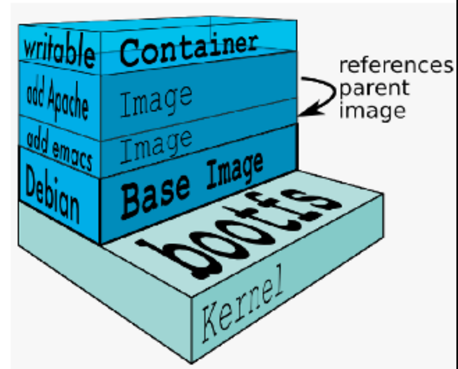


- Units of software delivery (ship it!)
  - run everywhere
    - regardless of kernel version
    - regardless of host distribution
    - (but container and host architecture must match*)
  - run anything
    - if it can run on the host, it can run in the container
    - i.e., if it can run on a Linux kernel, it can run

*Unless you emulate CPU with QEMU and binfmt

Docker Image structure

- NOT A Virtual Hark Disk (VHD) file
- NOT A FILESYSTEM
- uses a *Union File System*
- a read-only
- do not have state
- Basically a tar file
- Has a hierarchy
  - Arbitrary depth
- Fits into the Docker Registry



Cloud Native Applications 95

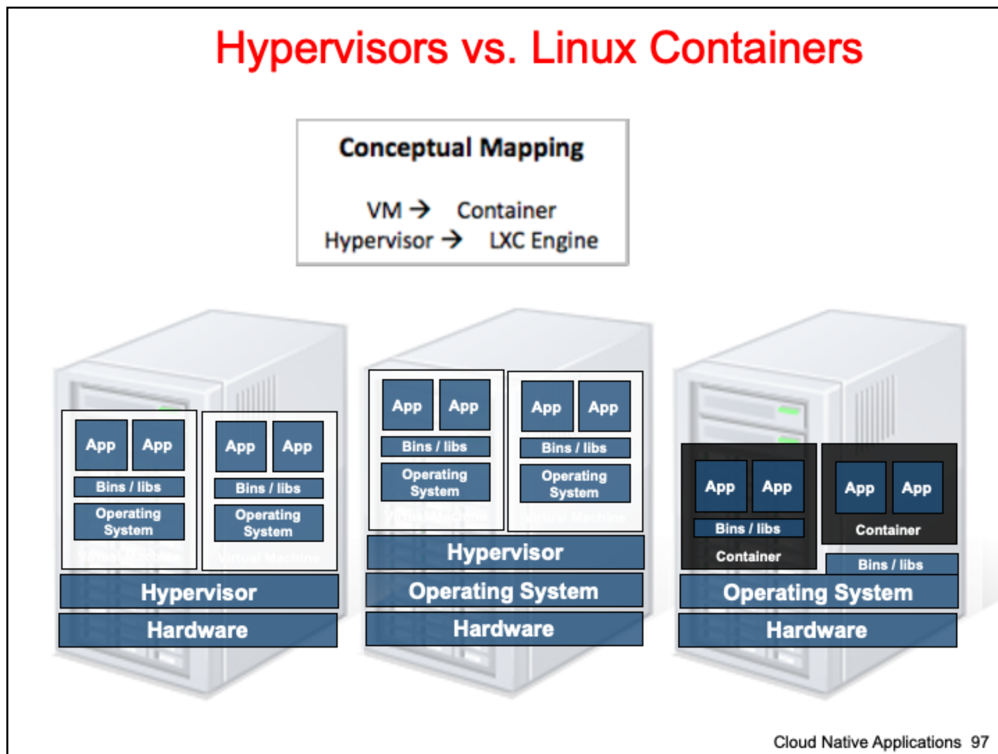
Google's Kubernetes: - Merging 2 Different Types of Containers

Docker

- It's about ***packaging***
- Control:
  - packages
  - versions
  - (some config)
- Layered file system
- ⇒ Prod matches testing

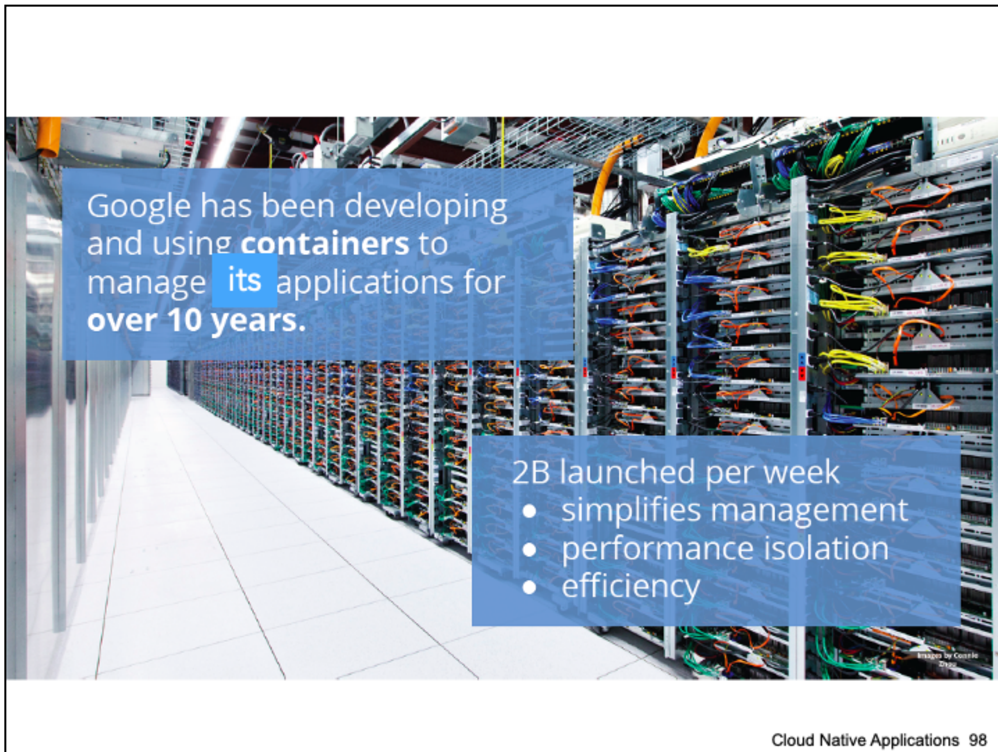
Linux Containers

- It's about ***isolation***
  … *performance isolation*
- not *security* isolation
  … use VMs for that
- Manage CPUs, memory, bandwidth, …
- Nested groups



Type 1 (bare-metal)
VMware ESX, Microsoft Hyper-V, Xen

Type 2 (hosted)
VMware Workstation, Microsoft Virtual PC,
Sun VirtualBox, QEMU, KVM


Google has been developing and using **containers** to manage its applications for **over 10 years.**
2B launched per week
● simplifies management
● performance isolation
● efficiency

Why Containers ?

- Ease of development:
  - User makes jobs based on containers ; the cluster/cloud schedule those jobs for them
    - User don't need to worry about machines or the OS
- High Resource Utilization: (more efficient)
  - The scheduler can pack many containers per machine
  - Can mix Live-services and Batch workload
    - Use Batch-job to fill in the holes
- Ease of Operation: (fewer staff per job)
  - e.g. All jobs use latest security patches
  - More shared code among projects (shared services and code)

Kubernetes (K8s)

κυβερνήτης: Greek for “pilot” or “helmsman of a ship”
The open-source cluster manager from Google

- Container Orchestrator
- Run Docker Container
- Support multiple cloud and bare-metal environments
- Inspired and informed by Google’s experiences & internal systems, e.g. the Borg scheduler
- Open source, written in Go:
  - Google has donated it to Cloud Native Computing Forum (CNCF)

Key: Kubernetes manages Applications NOT Machines

Kubernetes: Higher Level of Abstraction

Think About

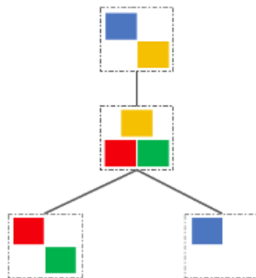
- Composition of services
- Load-balancing
- Names of services
- State management
- Monitoring and Logging
- Upgrading

Don't Worry About

- OS details
- Packages — no conflicts
- Machine sizes (much)
- Mixing languages
- Port conflicts

Kubernetes – Google's path towards "Cloud-native" Applications

- Kubernetes serves as a distributed platform for Hosting and Orchestrating Containers in a clustered environment
  - Support: Container Grouping (Pods), Replication, Scheduling, Load-Balancing, Auto-Healing, Scaling, Service Discovery, etc .
- Cloud-native Apps often structured as Interacting Microservices
  - Encapsulated states with APIs, like "Objects"
  - Mix of Programming Languages
  - Mix of Teams



Don't think of a container as the boundary of your application

"A container is more like a class in an object-oriented language."

--- Google's Brendan Burns

ns 102

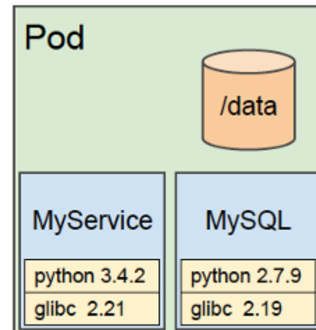
Services Model

- Each app lives in an environment of shared services
  - Storage, monitoring, logging
  - Master election, deployment, testing
- Services are *Abstract*
  - A "Service" is just a long-lived abstract name
  - Varied Implementations over time (versions)
    - Multiple versions running at a time
      - Required for "canary" testing -- deploy new version bit-by-bit gradually
    - Usually new versions are backward compatible
    - Sometimes not => must eventually update ALL clients
    - Running multiple versions gives clients/users some time to update
  - Services can be (and often are) updated independently
- Kubernetes routes to the right implementation

Managing Dependencies w/ K8s

Containers:
- Handle *package* dependencies
- Different versions, same machine
- No “DLL hell”

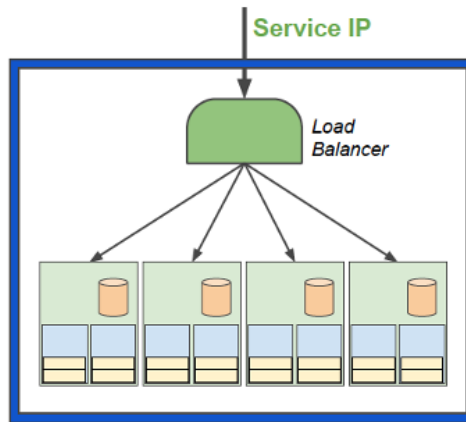
Pods:
- *Co-locate* containers
- Shared volumes
- IP address, independent port space
- Unit of deployment, migration



Running/ Managing Services w/ K8s

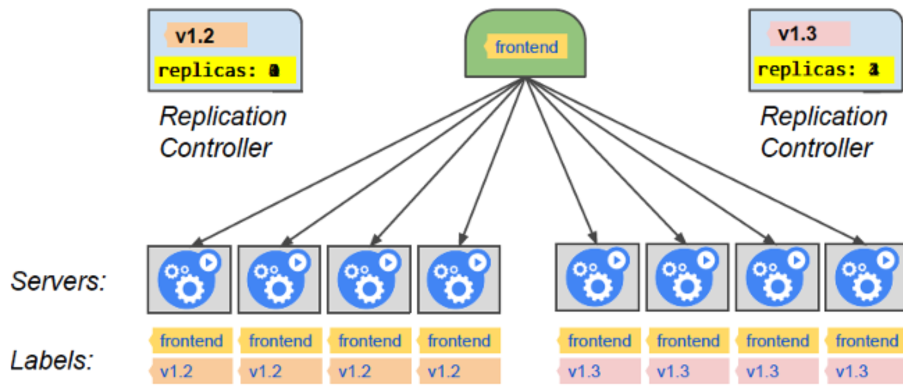
Services:

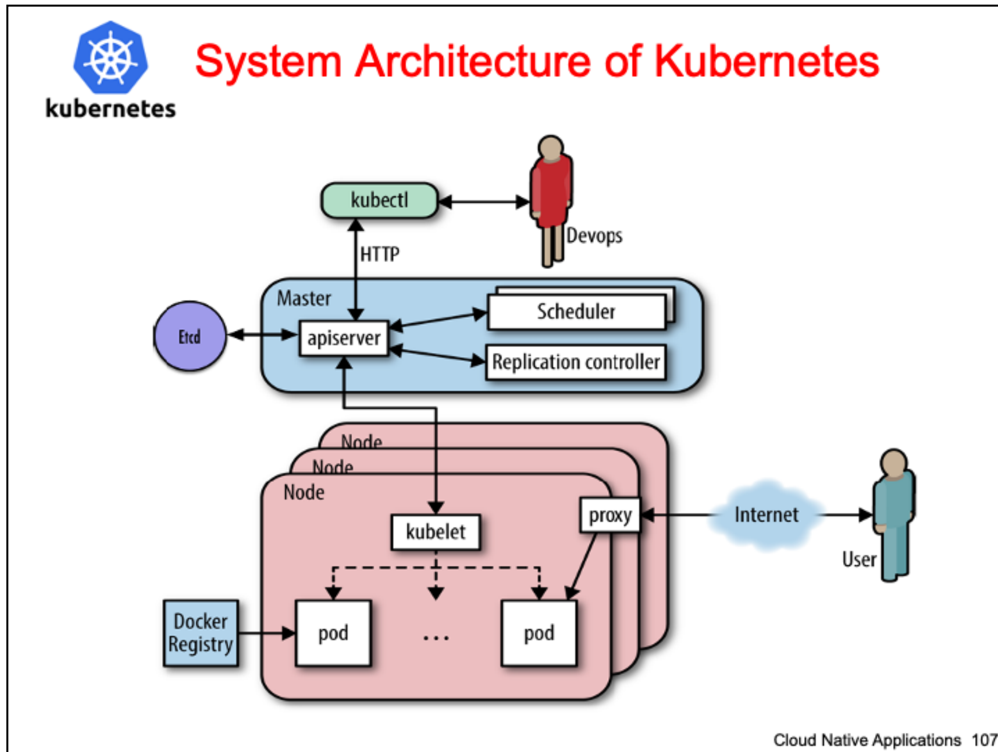
- Replicated pods
  - Source pod is a template
- Auto-restart member pods
- Abstract name (DNS)
- IP address for the service
  - in addition to the members
- Load balancing among replicas



Managing Service Dependenices w/ K8s

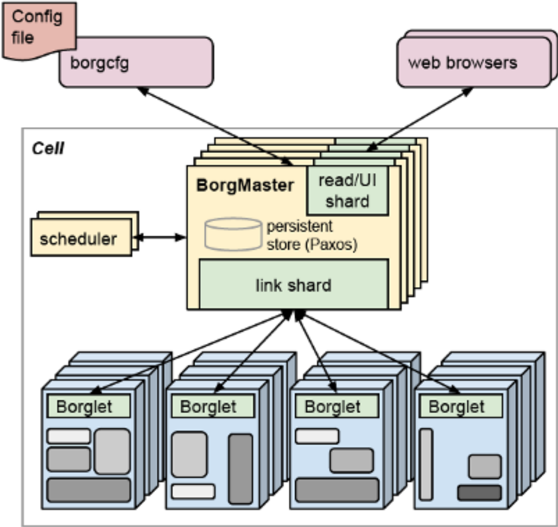
Example: Rolling Upgrade with Labels



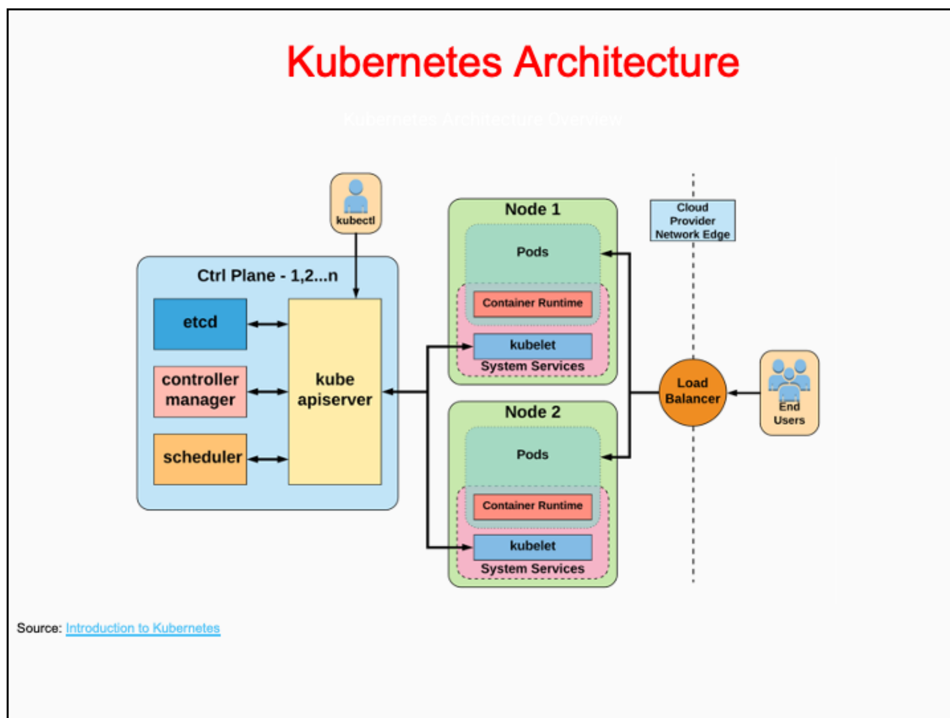


- Kube-apiserver
  - Gate keeper for everything in kubernetes
  - EVERYTHING interacts with kubernetes through the apiserver
- Etcd
  - Distributed storage back end for kubernetes
  - The apiserver is the only thing that talks to it
- Kube-controller-manager
  - The home of the core controllers
- kube-scheduler
  - handes placement

Recall - The Architecture of Google's Borg



A. Verma, L. Pedrosa, "Large-scale cluster management at Google with Borg", Eurosys 2015



Architecture Overview

Masters - Acts as the primary control plane for Kubernetes. Masters are responsible at a minimum for running the API Server, scheduler, and cluster controller. They commonly also manage storing cluster state, cloud-provider specific components and other cluster essential services.

Nodes - Are the 'workers' of a Kubernetes cluster. They run a minimal agent that manages the node itself, and are tasked with executing workloads as designated by the master.

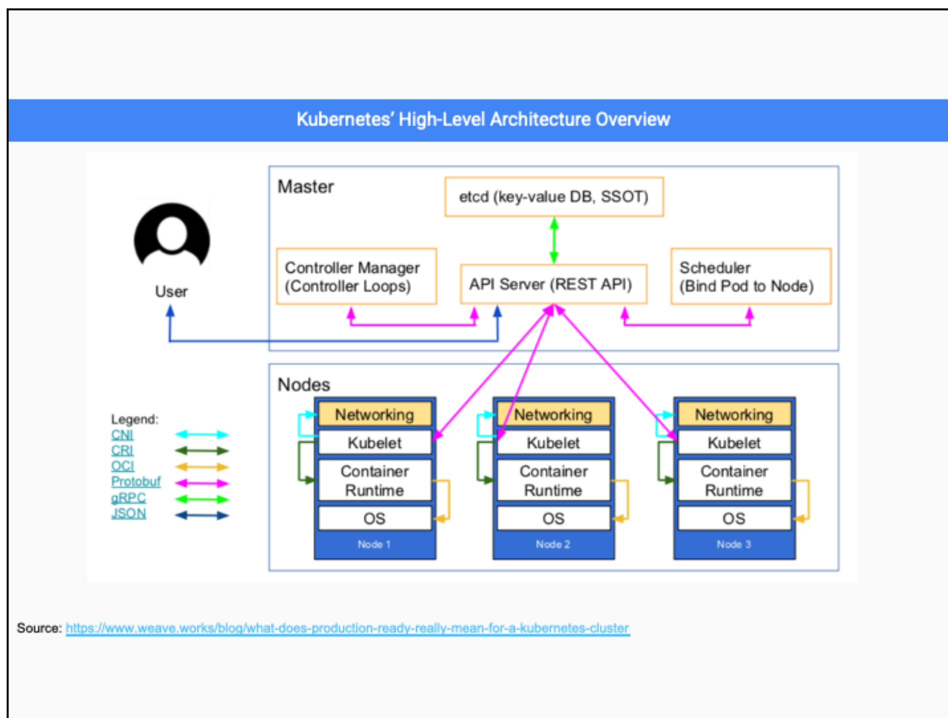
- Kube-apiserver
  - Gate keeper for everything in kubernetes
  - EVERYTHING interacts with kubernetes through the apiserver
- Etcd
  - Distributed storage back end for kubernetes
  - The apiserver is the only thing that talks to it
- Kube-controller-manager
  - The home of the core controllers
- kube-scheduler
  - handles pod placement

50K feet on how Kubernetes works

How Kubernetes works?

In Kubernetes, there is one (or more) master node and multiple worker nodes, each worker node can handle multiple pods. Pods are just a bunch of containers clustered together as a working unit. You can start designing your applications using pods. Once your pods are ready, you can specify pod definitions to the master node, and how many you want to deploy. From this point, Kubernetes is in control. It takes the pods and deploys them to the worker nods.

Source: https://itnext.io/successful-short-kubernetes-stories-for-devops-architects-677f8bfed803

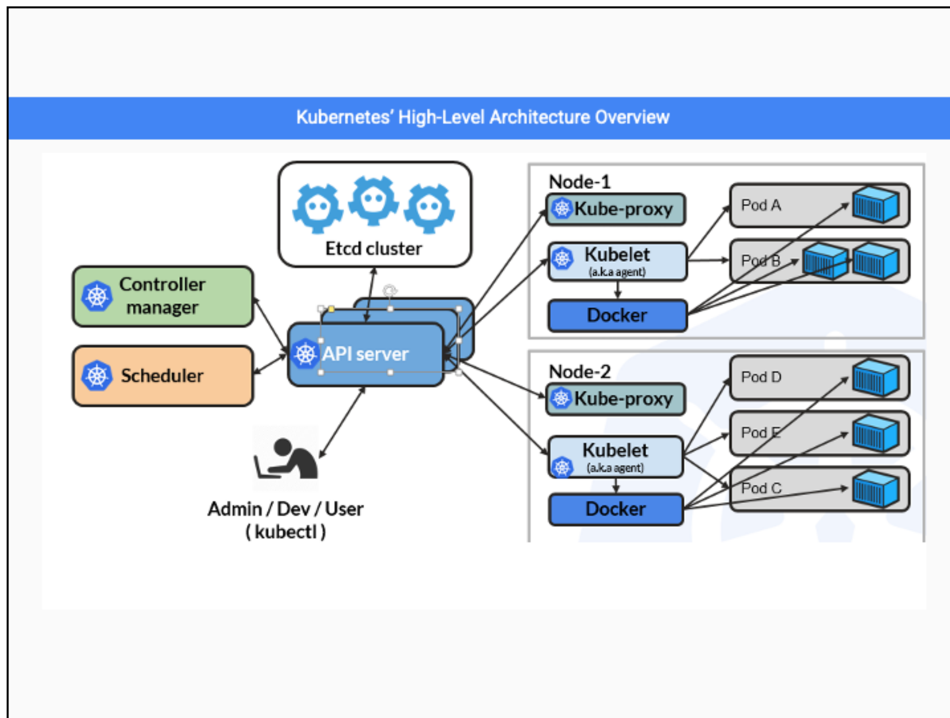


Architecture Overview

Masters - Acts as the primary control plane for Kubernetes. Masters are responsible at a minimum for running the API Server, scheduler, and cluster controller. They commonly also manage storing cluster state, cloud-provider specific components and other cluster essential services.

Nodes - Are the 'workers' of a Kubernetes cluster. They run a minimal agent that manages the node itself, and are tasked with executing workloads as designated by the master.

- Kube-apiserver
    - Gate keeper for everything in kubernetes
    - EVERYTHING interacts with kubernetes through the apiserver
- Etcd
    - Distributed storage back end for kubernetes
    - The apiserver is the only thing that talks to it
- Kube-controller-manager
    - The home of the core controllers
- kube-scheduler
    - handes placement



Architecture Overview

Masters - Acts as the primary control plane for Kubernetes. Masters are responsible at a minimum for running the API Server, scheduler, and cluster controller. They commonly also manage storing cluster state, cloud-provider specific components and other cluster essential services.

Nodes - Are the 'workers' of a Kubernetes cluster. They run a minimal agent that manages the node itself, and are tasked with executing workloads as designated by the master.

Core Concepts of Kubernetes

Cluster — Set of machines where pods are deployed, managed and scaled.

- Nodes are connected via a "flat" network.
- Typical cluster sizes range from 1-200 nodes.

Pod — A pod consists of one or more containers that are guaranteed to be co-located on the same machine.

- Share storage volumes and a networking stack.
- A pod is the basic unit of scheduling.

Controller — A controller is a reconciliation loop that drives actual cluster state toward the desired cluster state.

- **Replication Controller** — Handles replication and scaling by running a specified number of copies of a pod across the cluster.

Service — Set of pods that work together, such as one tier of a multi-tier application. Kubernetes provides:

- Service discovery
- Request routing by assigning a stable IP address and DNS name
- Load balancing

Label — The user can assign key-value pairs (called labels) to any API object in the system (e.g., pods, nodes).

- **Label selector** — A query against a label that returns matching objects.

Borg vs. Kubernetes Comparisons:

- Borg is a predecessor to Kubernetes
- Borg groups tasks by 'job' and simple numeric index; Kubernetes adds 'labels' for greater flexibility.
- Kubernetes allows for Docker and other containers, while Borg only allows LMCTFY
- Borg exposes the API of its various components to allow external program to access/control cluster-resource directly ; All accesses to a K8s-cluster must go through a single-point-of-contact: the API-server
- Single IP per machine in Borg complicates things
  - Because of Linux namespaces, VMs, IPv6, and software-defined networking, Kubernetes assigns every “pod” and service its own IP address
  - Allows developers to choose ports and removes the infrastructure complexity of managing ports
- Borg is not open source or available for use outside of Google unlike Kubernetes
  - Both work on bare metal, but Kubernetes can work on various cloud hosting providers, “such as Google Compute Engine.”

114

LMCTFY = Let Me Contain That For You ; Google’s in-house container technologies

Kubernetes is a portable, extensible open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation.

Jobs are restrictive as the only grouping mechanism for tasks

labels – arbitrary key/value pairs that users can attach to any object in the system.

One IP address per machine complicates things.

Thanks to the advent of Linux namespaces, VMs, IPv6, and software-defined networking, Kubernetes can take a more user-friendly approach that eliminates these complica- tions: every pod and service gets its own IP address, allowing developers to choose ports rather than requiring their soft- ware to adapt to the ones chosen by the infrastructure, and removes the infrastructure complexity of managing ports.

More Concepts of Kubernetes

- Controllers
  - Deployments →
  - ReplicaSet →
  - ReplicationController →
  - DaemonSet →
- StatefulSets →
- ConfigMaps →
- Secrets →
- Persistent Volumes (attaching storage to containers) →
- Life Cycle of Applications in Kubernetes →
  - Updating Pods
  - Rolling updates
  - Rollback

What is a Deployment in K8s ?

- A Deployment provides declarative updates for Pods and ReplicaSets, specifying the “desired” state of a deployment
  - The Deployment Controller is responsible to change the actual state to the desired state. This is the so-called “reconciliation” process.

Use Case of Deployments:

- Create a Deployment to rollout a ReplicaSet
- Declare the new state of the Pods
- Rollback to an earlier Deployment revision
- Scale up the Deployment to facilitate more Load
- Pause for Deployment to apply fixes to its PodTemplateSpec before resume it to start a new rollout

116

Kubernetes is a portable, extensible open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation.

Jobs are restrictive as the only grouping mechanism for tasks

labels – arbitrary key/value pairs that users can attach to any object in the system.

One IP address per machine complicates things.

Thanks to the advent of Linux namespaces, VMs, IPv6, and software-defined networking, Kubernetes can take a more user-friendly approach that eliminates these complica- tions: every pod and service gets its own IP address, allowing developers to choose ports rather than requiring their soft- ware to adapt to the ones chosen by the infrastructure, and removes the infrastructure complexity of managing ports.

Kubernetes Resources Explained (1)

Kubernetes resources explained (1)		
	Resource (abbr.) [API version]	Description
	Namespace* (ns) [v1]	Enables organizing resources into non-overlapping groups (for example, per tenant)
Deploying Workloads	Pod (po) [v1]	The basic (atomic) deployable unit containing one or more processes in co-located containers
	ReplicaSet	Keeps one or more pod replicas running
	ReplicationController	The older, less-powerful equivalent of a ReplicaSet
	Job	Runs pods that perform a completable task
	CronJob	Runs a scheduled job once or periodically
	DaemonSet	Runs one pod replica per node (on all nodes or only on those matching a node selector)
	StatefulSet	Runs stateful pods with a stable identity
	Deployment	Declarative deployment and updates of pods

Source: Kubernetes in Action book by Marko Lukša

Kubernetes Resources Explained (2)

	Resource (abbr.) [API version]	Description
Services	Service (svc) [v1]	Exposes one or more pods at a single and stable IP address and port pair
	Endpoints (ep) [v1]	Defines which pods (or other servers) are exposed through a service
	Ingress (ing) [extensions/v1beta1]	Exposes one or more services to external clients through a single externally reachable IP address
Config	ConfigMap (cm) [v1]	A key-value map for storing non-sensitive config options for apps and exposing it to them
	Secret [v1]	Like a ConfigMap, but for sensitive data
Storage	PersistentVolume* (pv) [v1]	Points to persistent storage that can be mounted into a pod through a PersistentVolumeClaim
	PersistentVolumeClaim (pvc) [v1]	A request for and claim to a PersistentVolume
	StorageClass* (sc) [storage.k8s.io/v1]	Defines the type of storage in a PersistentVolumeClaim

Source: Kubernetes in Action book by Marko Lukša

Kubernetes Resources Explained (3)

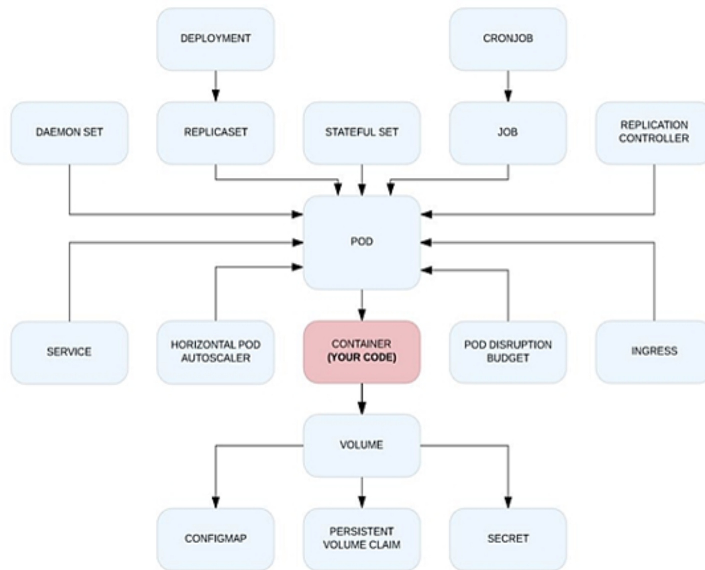
	Resource (abbr.) [API version]	Description
Scaling	HorizontalPodAutoscaler (hpa) [autoscaling/v2beta1**]	Automatically scales number of pod replicas based on CPU usage or another metric
	PodDisruptionBudget (pdb) [policy/v1beta1]	Defines the minimum number of pods that must remain running when evacuating nodes
Resources	LimitRange (limits) [v1]	Defines the min, max, default limits, and default requests for pods in a namespace
	ResourceQuota (quota) [v1]	Defines the amount of computational resources available to pods in the namespace
Cluster state	Node* (no) [v1]	Represents a Kubernetes worker node
	Cluster* [federation/v1beta1]	A Kubernetes cluster (used in cluster federation)
	ComponentStatus* (cs) [v1]	Status of a Control Plane component
	Event (ev) [v1]	A report of something that occurred in the cluster

Source: Kubernetes in Action book by Marko Lukša

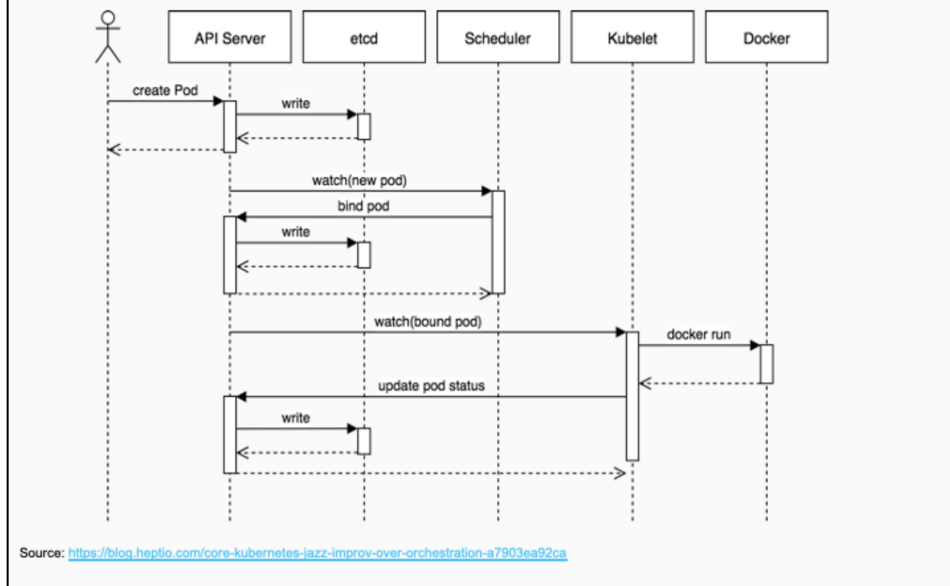
Kubernetes Resources Explained (4)

	Resource (abbr.) [API version]	Description
Security	ServiceAccount (sa) [v1]	An account used by apps running in pods
	Role [rbac.authorization.k8s.io/v1]	Defines which actions a subject may perform on which resources (per namespace)
	ClusterRole* [rbac.authorization.k8s.io/v1]	Like Role, but for cluster-level resources or to grant access to resources across all namespaces
	RoleBinding [rbac.authorization.k8s.io/v1]	Defines who can perform the actions defined in a Role or ClusterRole (within a namespace)
	ClusterRoleBinding* [rbac.authorization.k8s.io/v1]	Like RoleBinding, but across all namespaces
	PodSecurityPolicy* (psp) [extensions/v1beta1]	A cluster-level resource that defines which security-sensitive features pods can use
	NetworkPolicy (netpol) [networking.k8s.io/v1]	Isolates the network between pods by specifying which pods can connect to each other
	CertificateSigningRequest* (csr) [certificates.k8s.io/v1beta1]	A request for signing a public key certificate
Ext.	CustomResourceDefinition* (crd) [apiextensions.k8s.io/v1beta1]	Defines a custom resource, allowing users to create instances of the custom resource

Application Dependency on Kubernetes primitives



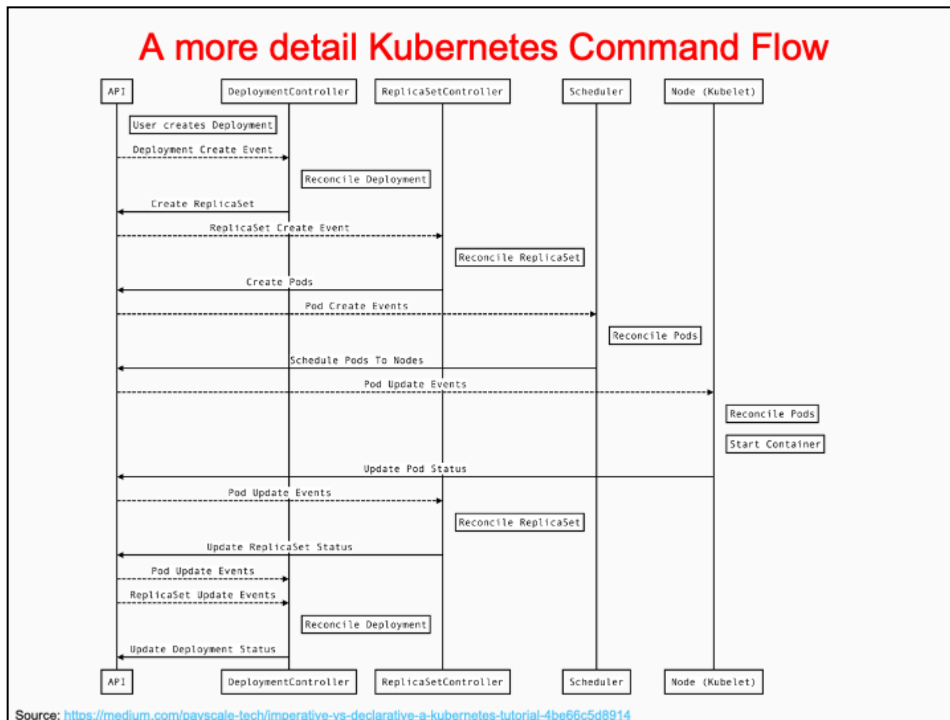
Source: Kubernetes effect by Bilgin Ibryam



1. **Client** sends a request for a Deployment to the **API Server** (for example using kubectl create -f deployment.yml)
2. **API Server** persists the Deployment to **etcd; etcd** returns 200 to **API Server; API Server** returns 200 to **Client** (obviously, the work isn't done here, there is a lot of background asynchronous stuff that happens next)
3. **Controller Manager** has a **watch** against the **API Server;** it sees that a Deployment has been created and it populates that

Informer to the **SharedCache**

4. **Deployment Controller** sees the **Deployment**, pulls it off the **queue**, creates a **ReplicaSet** and persists the ReplicaSet object back to the **API Server** and back to **etcd**
5. Repeat Step 3 for ReplicaSet
6. **ReplicaSet Controller** sees the **ReplicaSet**, pulls it off the queue, creates X number of **Pods** and persists the Pods object back to the **API Server** and back to **etcd**
7. Repeat Step 3 for Pods (but **Scheduler** does this part now)
8. **Scheduler** sees the **unscheduled pods**, and executes its **business logic** to fill out the **nodeName** field in the pod's **Spec** with the name of the **Schedulable Node** and persists the scheduled Pods object back to the **API Server** and back to **etcd**
9. Repeat Step 3 for Pods (but **Kubelet** this time**)**
10. **Kubelet** (on the scheduled node) sees it's supposed to have a Pod running on its machine and talks with the container runtime (**Docker**, etc.) to make that happen



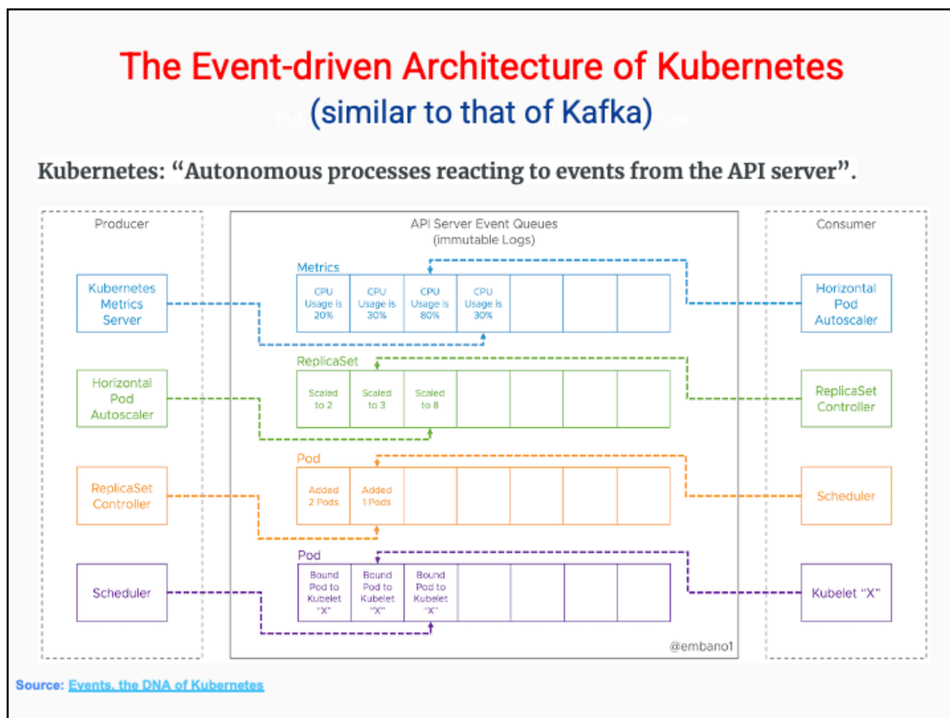
Source: https://medium.com/payscale-tech/imperative-vs-declarative-a-kubernetes-tutorial-4be66c5d8914

1. kubectl translates your imperative command into a declarative Kubernetes Deployment object. A Deployment is a higher-level API that allows rolling updates (see below).
2. kubectl sends the Deployment to the Kubernetes API server, kube-apiserver, which runs in-cluster.

3. kube-apiserver saves the Deployment to etcd (a distributed key-value store), which also runs in-cluster, and responds to kubectl.
4. Asynchronously, the Kubernetes controller manager, kube-controller-manager, which watches for Deployment events (among others), creates a ReplicaSet from the Deployment and sends it to kube-apiserver. A ReplicaSet is a version of a Deployment. During a rolling update, a new ReplicaSet will be created and progressively scaled out to the desired number of replicas, while the old one is scaled in to zero.
5. kube-apiserver saves the ReplicaSet to etcd.
6. Asynchronously, kube-controller-manager, creates two Pods (or more if we scale out) from the ReplicaSet and

sends them to kube-apiserver. **Pods are the basic unit of Kubernetes. They represent one or several containers sharing a Linux cgroup and namespaces.**

7. kube-apiserver saves the Pods to etcd.
8. Asynchronously, the Kubernetes scheduler, kube-scheduler, which watches for Pod events, updates each Pod to assign it to a Node and sends them back to kube-apiserver.
9. kube-apiserver saves the Pods to etcd.
10. Finally, the kubelet that runs on the assigned Node, always watching, actually starts the container.



How to understand Kubernetes

Think of the API server as an *immutable (replicated) log* (or queue), each representing a stream of events. Events are facts that can be causally related (*happened-before*) or not related at all (then we say they happened *concurrently*). *etcd* is important for the durability of events, but an implementation detail.

All processes (controllers), e.g. the scheduler, deployment controller, endpoint controller, Kubelet, etc. can be understood as *producers and/or consumers* of events (consumers can be producers as well, and vice versa).

Consumers specify the objects (and optionally namespace) they want to receive events from the API server. This is called a *watch* in Kubernetes. Think of the combination of

object+namespace as a dedicated (virtual) event queue the API server handles.

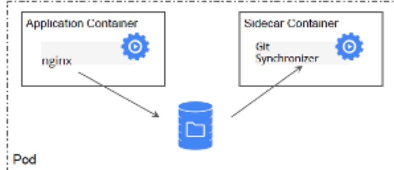
Consumers and producers don't know about each other as they're fully decoupled (by the queue) and autonomous. This makes the whole system extremely scalable, robust and extensible (adaptable to change).

Thus, by design, it's a fully *asynchronous and eventually consistent* platform. Information takes time to propagate from producers(s) to consumers(s). The diagram below shows this where the Horizontal Pod Autoscaler hasn't caught up with the events coming from the metrics server, which also affects the downstream chain of producers and consumers in our example.

Local and Distributed Abstractions/ Patterns

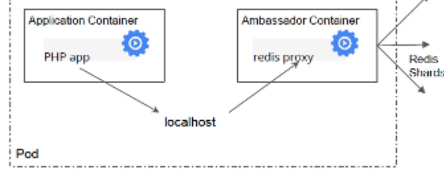
Sidecar Pattern

Sidecars extend and enhance



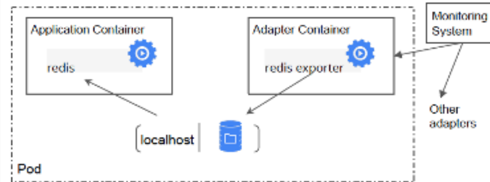
Ambassador Pattern

Ambassadors represent and present



Adapter Pattern

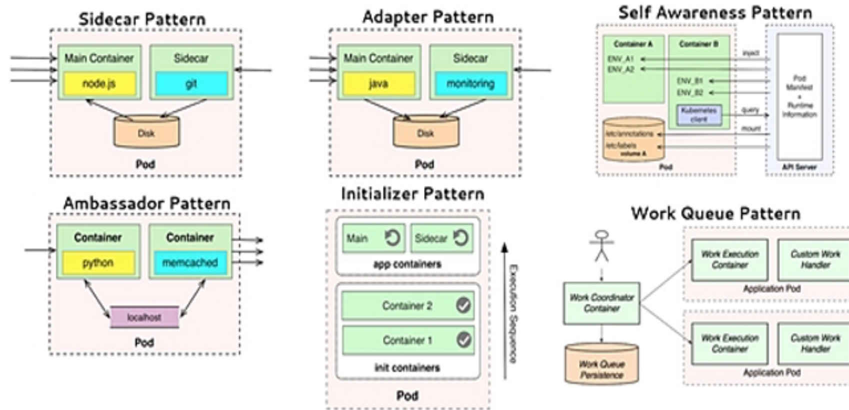
Adapters normalize and abstract



Source: Eric Brewer, ACM SoCC Keynote, 2015

Sidecars: Extend and Enhance
Ambassadors: Represent and Present
Adapters: Normalize and Abstract

Local and Distributed Abstractions/ Patterns



Source: Kubernetes effect by Bilgin Ibryam

Sidecars: Extend and Enhance
Ambassadors: Represent and Present
Adapters: Normalize and Abstract

Who "Manages" Kubernetes?

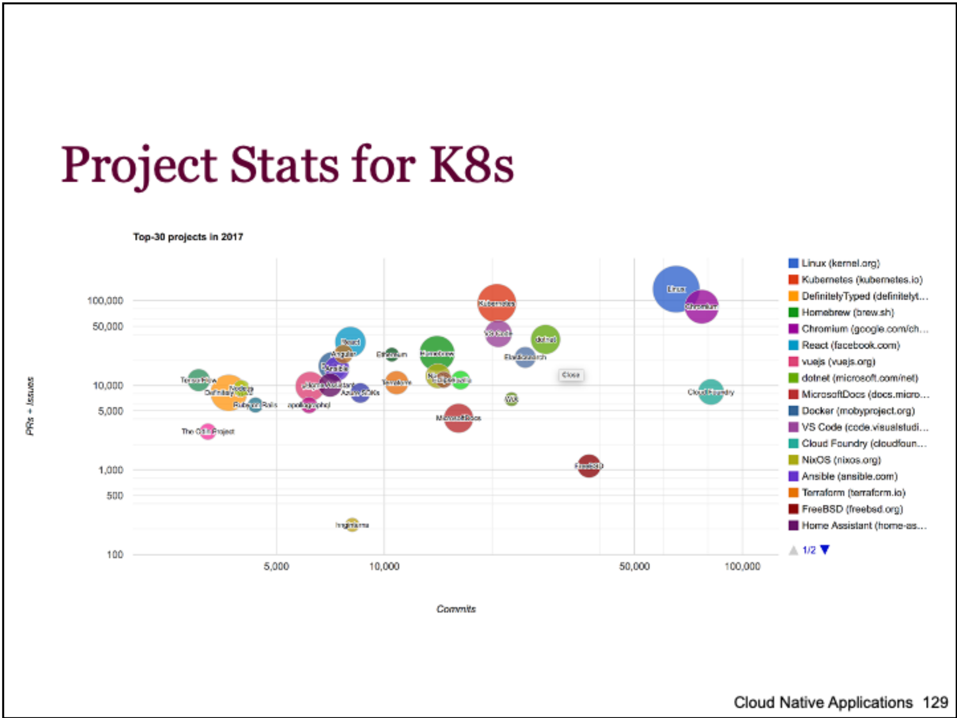
The CNCF is a child entity of the Linux Foundation and operates as a vendor neutral governance group.

- CNCF is a sub-foundation of the Linux Foundation
- A vendor neutral entity to manage "cloud native" projects
- Projects that are "cloud native" generally focus on:
  - containers
  - dynamic orchestration
  - enable "microservice" deployment methodologies
  - can be logging, monitoring, containerizers etc
- Kubernetes was the first project under the CNCF, and the first to be "graduated"

Project Stats for K8s

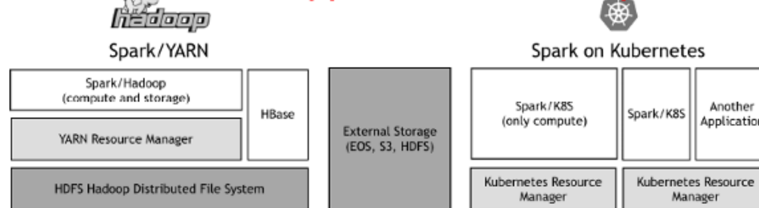
- Over 55,000 stars on Github
- 2000+ Contributors to K8s Core
- Most discussed Repository by a large margin
- **70,000+** users in Slack Team

- Very active project
- 49k starts on github
- 2000+ contributors to main core repo
- most discussed repo on github
- massive slack team with 60k+ users



- Graphic is a little out of date at this point, but for the most part still holds true
- Also one of the largest open source projects based on commits and PRs.

Challenges for Cloud-based K8s approach vs. Hadoop/YARN



- Development of Container-based technologies was geared towards Ephemeral (stateless) Compute-oriented Pods ;
  - Long-term state/ results need to be stored in External Persistent Storage
- For Big Data applications, complex stateful information and data stored on the persistent storage can be large while ephemeral compute nodes may need to be scaled separately
- Trade-offs between Data Locality and Compute Elasticity (Still remember the key idea of "Bringing Computation to the Data" to avoid network-transfer bottleneck ?)
- Deploying Large-scale Persistent (stateful) apps/ services, e.g. Hadoop, over Ephemeral-oriented K8s computing pods remains non-trivial but some solutions are emerging:
  - Cloud Native Storage Solutions include: Container Storage Interface (CSI for K8s), Rook w/ Ceph, Rook w/ Cassandra, Rook w/ CockroachDB etc + Commercial ones, e.g. Trident from NetApp
  - Some Hadoop-over-K8s efforts: e.g. BlueK8 and Kubedirectors, will Hadoop/YARN survive ?
- Steep learning curve for K8s, e.g. for writing/ configuring app package to be deployed over it:
  - YAML, Helm charts, Operators, etc.

Cloud Native Applications 130

1. kubectl translates your imperative command into a declarative Kubernetes Deployment object. A Deployment is a higher-level API that allows rolling updates (see below).
2. kubectl sends the Deployment to the Kubernetes API server, kube-apiserver, which runs in-cluster.

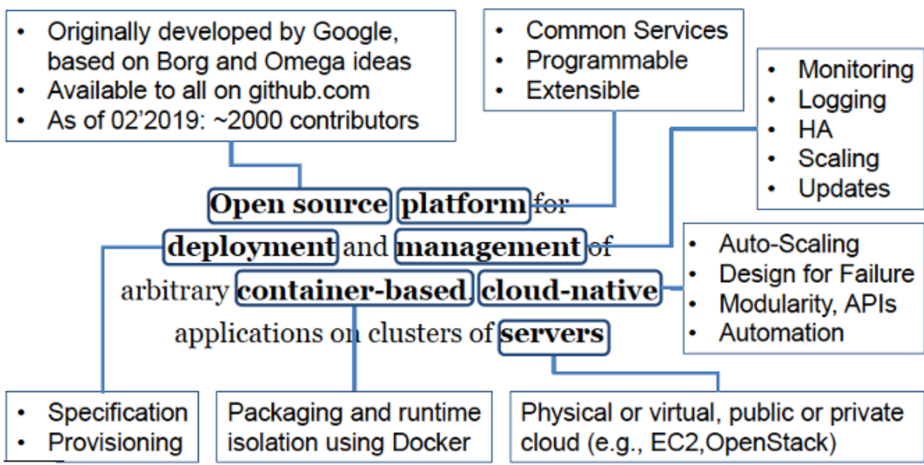
3. kube-apiserver saves the Deployment to etcd (a distributed key-value store), which also runs in-cluster, and responds to kubectl.
4. Asynchronously, the Kubernetes controller manager, kube-controller-manager, which watches for Deployment events (among others), creates a ReplicaSet from the Deployment and sends it to kube-apiserver. A ReplicaSet is a version of a Deployment. During a rolling update, a new ReplicaSet will be created and progressively scaled out to the desired number of replicas, while the old one is scaled in to zero.
5. kube-apiserver saves the ReplicaSet to etcd.
6. Asynchronously, kube-controller-manager, creates two Pods (or more if we scale out) from the ReplicaSet and

sends them to kube-apiserver. **Pods are the basic unit of Kubernetes. They represent one or several containers sharing a Linux cgroup and namespaces.**

7. kube-apiserver saves the Pods to etcd.
8. Asynchronously, the Kubernetes scheduler, kube-scheduler, which watches for Pod events, updates each Pod to assign it to a Node and sends them back to kube-apiserver.
9. kube-apiserver saves the Pods to etcd.
10. Finally, the kubelet that runs on the assigned Node, always watching, actually starts the container.

Summary of Kubernetes

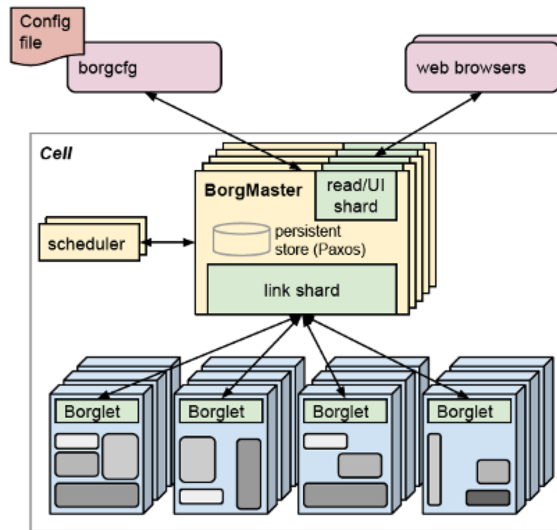
- An Open-source, Cloud-Native Container-based Platform



Backup Slides

More on Borg

High-level Architecture of Google's Borg



A. Verma, L. Pedrosa, "Large-scale cluster management at Google with Borg", Eurosys 2015

Borg Architecture - Borgmaster

- Each cell contains a Borgmaster
- Each Borgmaster consists of 2 processes:
    - Main Borgmaster process
    - Scheduler
- Multiple replicas of each Borgmaster
- Role of (elected leader) Borgmaster:
    - submission of job, termination of any of job's task

State of Borgmaster saved in Paxos-based store

Borg Architecture - Borglet

- Local Borg agent on every cell
  - starts/stops/restarts tasks
  - Manages local resources
  - Rolls over debug logs
- Polled by Borgmaster to get machine's current state
- If a Borglet does not respond to several poll messages, it is marked as down
  - Tasks re-distributed
  - If communication is restored, Borgmaster tells Borglet to kill rescheduled tasks

Borglet continues normal operation even if it cannot communicate with a Borgmaster replica or the replica fails. Currently running tasks and services can continue running

How does Borg work?

- Users submit "jobs"
  - Each "job" contains 1+ "task" that all run the same program/binary
  - Runs inside containers (not VMs as it would cost higher latency)
- Each "job" runs on one "cell"
  - A "cell" is a set of machines that run as one unit
- Two main types of jobs:
  - "Prod" job : long-running server jobs, higher priority
  - "Non-prod" job : quick batch jobs, lower priority

Cloud Native Applications 136

- Job properties:
  - Name
  - Owner
  - # Tasks
- Job program attributes:
  - Processor architecture
  - OS version
  - External IP address
- In fact, user can specify desired resource usage (CPU cores, RAM, disk space, disk access rate, TCP ports, etc). That said, we will later that Borg takes in other considerations at runtime for task scheduling.
-
- prod jobs 70% CPU reserved, 55% memory reserved

How does Borg work?

- **Allocs:**
  - Reserved set of resources in one machine
  - Can run multiple instances of a task, different tasks from many jobs, or future tasks
- **Priority and quota:**
  - Each job has a priority
  - Preemption disallowed between "prod" jobs.
  - Quota refers to vector of resource quantities for period of time
- **Support for naming and monitoring**

- Alloc example: web server instance and the corresponding logsaver task that copies logs from local disk to a distributed file system

- Quota example = 20 TiB of RAM at prod priority from now until the end of July in this particular cell
- Jobs whose quotas cannot be satisfied are immediately rejected upon submission

- Borg name service
- Borg also writes job size and task health information
- Helps for debugging (it seems)

Borg Architecture - Scalability

- Ultimate scalability limit is unknown
  - Single Borgmaster can manage thousands of borglets
  - Rates above 10,000 tasks per minute
  - Busy Borgmaster uses 10-14 CPU cores and 50GiB RAM

- Scheduling requires computing feasibility checks and score. This can become a huge bottleneck for scalability.

- Without these techniques, Borg takes more than 3 days to schedule a cell's entire workload from scratch. With these enabled, it only takes a few hundred seconds.

Isolation

- Security:
  - Linux *chroot* command used for process isolation
  - Standard sandboxing techniques used for running external software
- Performance:
  - Borg makes explicit distinction between LS (latency-intensive) tasks and batch tasks. Helps for priority-based preemption
  - Borg uses notion of compressible resources (CPU cycles, disk I/O bandwidth) and non-compressible resources (RAM, disk space)

- Chroot is an operation that changes the apparent root directory for the current running process and their children. A program that is run in such a modified environment cannot access files and commands outside that environmental directory tree. This modified environment is called a *chroot jail*.

Borg Architecture - Scheduling

- Borgmaster adds new jobs to a pending queue after recording it in the Paxos store
- A scheduler (primarily operates on tasks) scans and assigns tasks to machines
  - Feasibility checking
  - Scoring
- E-PVM vs “best-fit”
  - E-PVM leaves headroom for load-spikes but has increased fragmentation
  - Best-fit fills machines as tightly as possible, but hurts “bursty loads”
- Current model is a hybrid of both
  - Borg will kill lower priority tasks until it finds room for an assigned task

feasibility checking, to find machines on which the task could run, and scoring, which picks one of the feasible machines.

In feasibility checking, the scheduler finds a set of machines that meet the task’s constraints and also have enough “available” resource

The score takes into account user-specified preferences, but is mostly driven by built-in criteria such as minimizing the number and priority of preempted tasks, picking machines that already have a copy of the task’s packages, spreading tasks across power and failure domains, and pack- ing quality including putting a mix of high and low priority

Scheduler prefers to assign tasks to machines that already have packages (program and data) installed to run the tasks

Hybrid model prvides 3-5% better packing efficiency

Techniques used by Borg for Scalability:

- Score caching
  - Feasibility and scoring is expensive, scores are cached until properties of the machine or task change
- Equivalence class
  - A group of tasks with identical requirements
  - Stems from tasks within a job having identical requirements and constraints
  - Easier than determining feasibility for every pending task and scoring every machine
- Relaxed randomization
  - Scheduler examines machines in a random order until it has found enough feasible machines to score, and selects the best from this set
  - Speeds up assignments of tasks to machines

- Scheduling requires computing feasibility checks and score. This can become a huge bottleneck for scalability.

- Without these techniques, Borg takes more than 3 days to schedule a cell's entire workload from scratch. With these enabled, it only takes a few hundred seconds.

Borg - Achieving Availability

- To mitigate inevitable failures, Borg will:
  - Automatically reschedule evicted tasks
  - Reduce correlated failures by distributing across failure domains
  - Limits downtime due to maintenance
  - Use “declarative desired-state representations and idem-potent mutating operations” to ease resubmission of forgotten requests
  - Avoid task to machine pairings that cause crashes
  - Use a logsaver to recover critical data written to a local disk
- Achieve 99.99% availability in practice

- Automatically reschedule evicted tasks
- Reduce correlated failures by distributing across failure domains
- Limits downtime due to maintenance
- Use “declarative desired-state representations and idem-potent mutating operations” to ease resubmission of forgotten requests
- Avoid task to machine pairings that cause crashes
- Use a logsaver to recover critical data written to a local disk

Utilization - Main Goal of Borg

- Efficient utilization is very important for Google:
  - A few percentage improvement can save millions of dollars!
- Cell Sharing
- Large Cells
- Fine-grained Resource Requests
- Resource Reclamation

Borg/Kubernetes Comparisons:

- Borg is a predecessor to Kubernetes
- Borg groups work by 'job'; Kubernetes adds 'labels' for greater flexibility.
- Kubernetes allows for Docker and other containers, while Borg only seems to allow LMCTFY
- Single IP per machine in Borg complicates things
  - Because of Linux namespaces, VMs, IPv6, and software-defined networking, Kubernetes assigns every "pod" and service its own IP address
  - Allows developers to choose ports and removes the infrastructure complexity of managing ports
- Borg is not open source or available for use outside of Google unlike Kubernetes
  - Both work on bare metal, but Kubernetes can work on various cloud hosting providers, "such as Google Compute Engine."

Kubernetes is a portable, extensible open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation.

Jobs are restrictive as the only grouping mechanism for tasks

labels – arbitrary key/value pairs that users can attach to any object in the system.

One IP address per machine complicates things.

Thanks to the advent of Linux namespaces, VMs, IPv6, and software-defined networking, Kubernetes can take a more user-friendly approach that eliminates these complica- tions: every pod and service gets its own IP address, allowing developers to choose ports rather than requiring their soft- ware to adapt to the ones chosen by the infrastructure, and removes the infrastructure complexity of managing ports.

How does the Borgmaster handle load spikes while minimizing fragmentations?

- Hybrid of E-PVM (worst-fit) and best-fit model
  - E-PVM leaves headroom for large spikes, has large fragmentation
  - Best-fit model fills machines as tightly as possible
    - Pre-empts by killing lower priority tasks and add it back to pending queue
  - Large cells used to decrease fragmentation

Techniques Borg uses for managing utilization

- Cell-sharing: sharing prod and non-prod tasks
  - Resource reclaiming
  - Not sharing prod and non-prod work would increase machine needs by 20-30%
- Large cells: to allow large computations and decrease fragmentation
  - splitting up jobs and distributing them requires significantly more machines
- Fine-grained resource requests
  - fixed size containers/VMs not ideal
  - instead there are “buckets” of CPU/memory requirements
- Resource reclamation: jobs specify limits
  - Borg can kill tasks that use more RAM or disk space than requested
  - Throttle CPU usage
  - Prioritize prod tasks over non-prod

Evaluation metric: cell compaction, given a workload, find out how small a cell it could be fitted into by removing machines until the workload could no longer fit

increasing utilization by a few percent- age points can save millions of dollars

partly a practical matter: we found ourselves consuming 200 000 Borg CPU cores for our ex- periments at one point—even at Google’s scale, this is a non-trivial investment.

deliberately leave significant headroom for workload growth, occasional “black swan” events, load spikes, machine failures, hardware upgrades, and large-scale partial failures (e.g., a power supply bus duct)

Why is it important to have isolation, and how does Borg implement it?

- To protect an app from Noisy, Nosy and Messy neighbors
- Sharing machines between applications increases utilization, but isolation is needed to prevent tasks from interfering
  - Security: rogue tasks can affect other tasks, and information should not be visible between tasks
  - Performance:
    - Utilization can be decreased by users inflating resource requests to prevent interference
    - Again, rogue tasks can affect your task
- Security: Linux chroot jail is the primary security isolation mechanism
- Performance: Linux cgroup-based container
  - Also appclass is used to help with overload and overcommitment
  - High priority LS (latency-sensitive) tasks