

Kubernetes

Kubernetes (Admin)



Table of contents

1. Architecture
2. Installation
3. Configuration and operational maintenance
4. Cluster Upgrade
5. Day to day actions, make your Kubernetes users happy and aware
6. Extensibility: Operators, CRD and API Servers
7. Federation, Service Mesh, Security



Logistics

- Schedule
- Lunch
- Other questions?



Architecture

Table of contents

- *Architecture*
- Installation
- Configuration and operational maintenance
- Cluster Upgrade
- Day to day actions
- Extensibility: Operators, CRD and API Servers
- Federation, Service Mesh, Security



Chapter content

- Introduction
- Architecture
- Control Plane
- Workers

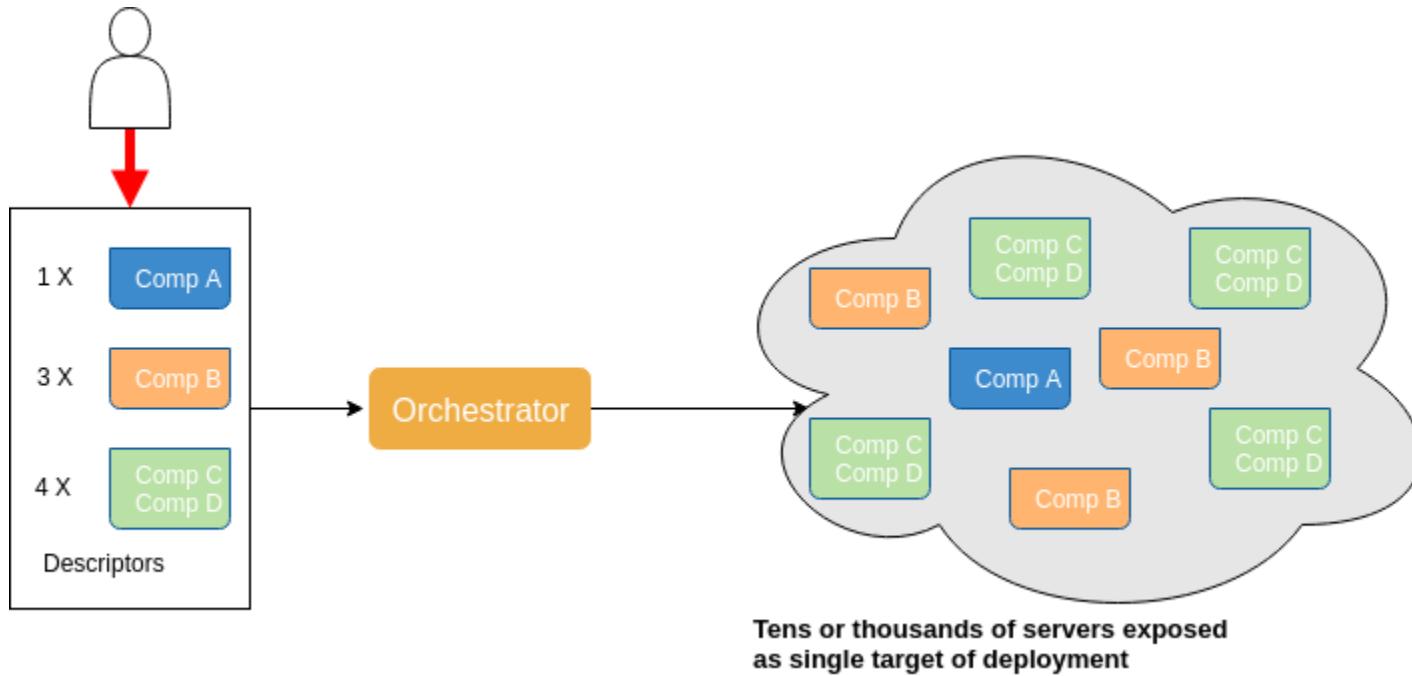
How to ...

- How to deal with container errors/unexpected stops?
- How to deal with node crash?
- How to deal with node maintenance?
- How to deal with scale up/down?
- How to deal with the multiplication of components to manage/deploy?
- How to deal with your apps updates?

Why use an orchestrator?

- Orchestrators have been built to answer all these questions by design
- Orchestrators are often called "Datacenter Operating Systems"
- They are tools built to ease the management of containerized workloads

The orchestrator from a final user perspective



Placement and container lifecycles

■ The orchestrator will choose where to host your containers according to constraints :

- technical constraints : network, hardware (GPU, HDD/SDD)
- co-localisation (affinity, anti-affinity, ...)
- number of replicas/copies
- CPU/memory/storage resources availability

Failover management

- The orchestrator will restart components that failed :
 - containers
 - nodes/hosts
- The orchestrator offers native high availability for control plane components

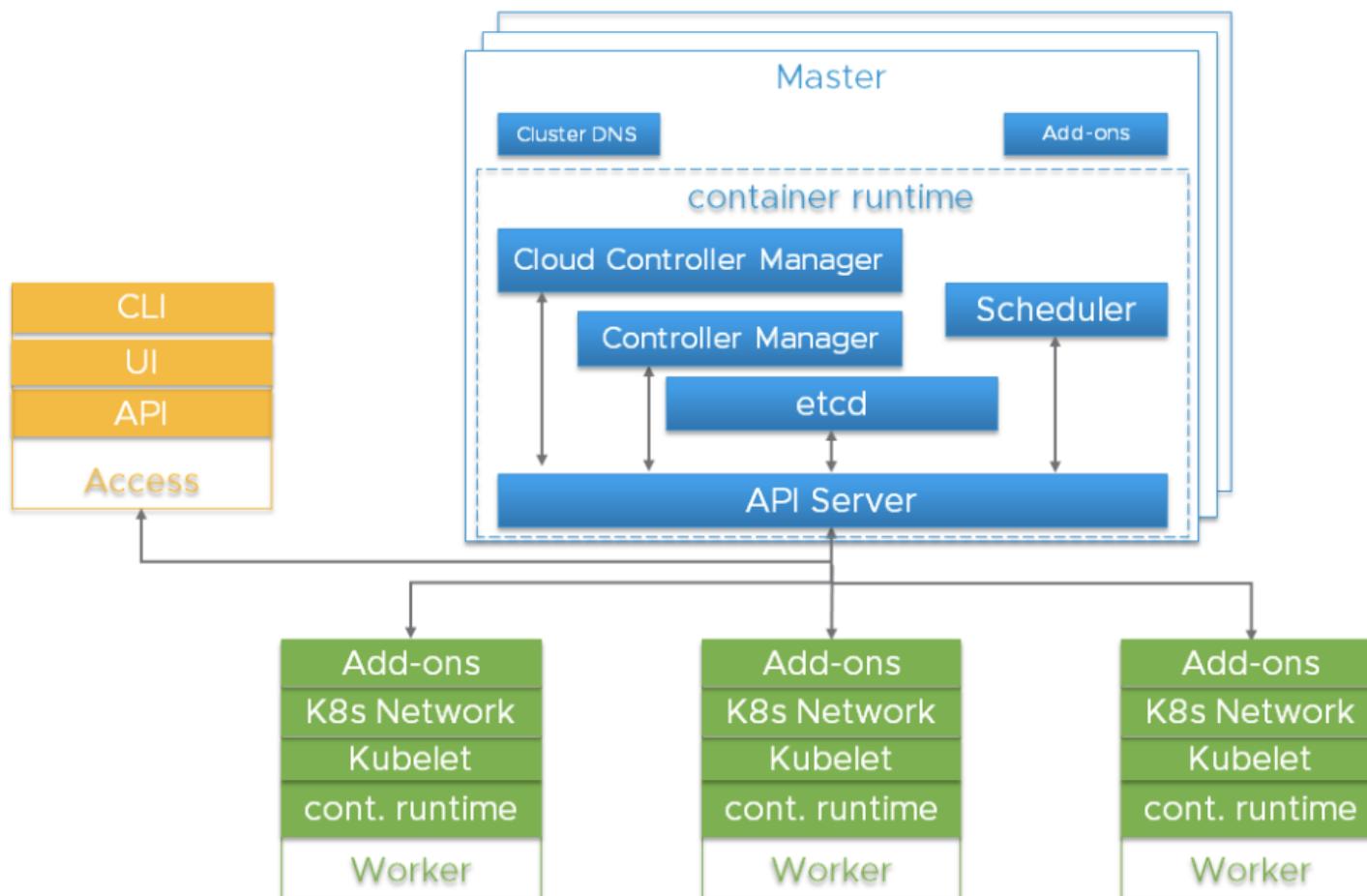
Networking features

- Load Balancing
- Service Discovery
- Networks shared by all containers accross nodes (most of the time **overlays**) so that containers don't need anymore to be connected using node ports

Orchestration features

- Distributed storage management
- Configuration and Secrets
- Manual and auto scaling
- Interaction through a command line and a REST API
- Text descriptors that can be managed like the rest of the application code
- ***Role Based Access Control*** : management of who can do what

Kubernetes architecture



-- Source: [kubeadm Cluster Creation Internals - Lucas Käldström](#)

Control Plane

- Number of masters defines your fault tolerance
 - $2*n + 1$ master nodes are needed to handle n masters failure
- Etcd can be hosted inside or outside the cluster
- If the control plane fails:
 - No updates/modifications can be applied to existing workloads and configurations
 - Workloads are not moved when a worker node fails
- But:
 - Workloads stay online
 - Containers can be restarted by the kubelet on the same node if required because it is not managed by a controller

Etcd - Overview

- A ***distributed hierarchical key/value store*** from CoreOS/RedHat
- Now a CNCF Project
- Programmed in Go
- Two exposed ports
 - 2379: client communication through a gRPC/Rest API
 - 2380: peer communication
- Node fault tolerance implemented with the Raft protocol

Etcd - Raft protocol

- **Raft**: an *understandable* distributed consensus algorithm
- Node roles: *leader, candidate or follower*
- Logs replication
 - Followers write to their log on leader request
 - Leader commits writes to log when follower majority acknowledges
 - Leader notifies followers of the new consensus
 - The leader *never* erases committed logs

*If you want to further understand raft you can check
Raft or docker container run -p 80:80
aaabbb000/nginx-raftscope*

Etcd in Kubernetes

Etcd versions:

- [Kubernetes Changelog](#) points out which etcd version you should use
- Kubernetes <v1.6: etcd API v2
- Kubernetes =>v1.6: etcd API v3 (with gRPC/protobuf)

With APIv3 and gRPC, api-server can register watchers on etcd resources

*Only the API Server should interact with etcd
(except for backup/restore)*

API Server

- Handles ***all interactions*** with the cluster
 - both external and internal
- Main front to the **etcd database** and so to the cluster desired state
 - authorizes and validates all requests
- Exposes the API
 - JSON for external interactions
 - gRPC/Protobuf for intra-cluster communications

API Server - Request flow

1. API Groups and version negotiation
2. Authentication
3. Authorization
4. Admission control
5. Storage to etcd

API servers are running active-active and always query etcd with quorum read making sure they see the last consistent state

API Server - Reminders API groups

The Kubernetes API is divided in **API groups** specified in a **REST path** and in the ***apiVersion*** field of a serialized object

- Core group (or legacy) at the REST path `/api/v1` and uses `apiVersion: v1`
- The named groups are at REST path `/apis/$GROUP_NAME/$VERSION`, and use `apiVersion: $GROUP_NAME/$VERSION` (e.g. `apiVersion: batch/v1`)
 - Full list available [here](#)

API Server - Admission controllers (1/3)

- Intercept requests to the API Server
- Two kinds of **Admission controllers** executed in order
 - **Mutating**: Make changes to a request/resource prior persistence
 - **Validating**: Validate the request

API Server - Admission controllers (2/3)

Admission controllers are enabled/disabled with parameters passed to the kube-apiserver command line.

- Enabling admission controller **NamespaceLifecycle**

```
kube-apiserver --enable-admission-plugins=NamespaceLifecycle ...
```

- Disabling admission controller **PodNodeSelector**

```
kube-apiserver --disable-admission-plugins=PodNodeSelector ...
```

API Server - Admission controllers (3/3)

Examples of included admission controllers:

- **DefaultStorageClass** applies a default storage class on pvc which don't have one
- **LimitRanger** ensures respect of the **LimitRange** constraint in the target namespace
- **NamespaceLifecycle** rejects creation in non-existent or being deleted namespace and deletion of **default**, **kube-system**, **kube-public** namespaces

Full list available [here](#).

Scheduler

- Defines where to run workloads based on available resources
- Runs in active/passive mode
 - Current active scheduler is flagged in **etcd**
- Two steps
 - Filtering
 - Ranking
 - If same priority, choose random

Scheduler - Filtering

Filter Predicates in Kubernetes

- PodFitsResources
- CheckNodeMemoryPressure
- CheckNodeDiskPressure
- PodFitsHostPorts
- HostName
- MatchNodeSelector
- NoDiskConflict
- NoVolumeZoneConflict
- MaxEBSVolumeCount
- MaxGCEPDVolumeCount

Scheduler - Ranking (1/2)

For example, suppose there are two priority functions, `priorityFunc1` and `priorityFunc2` with weighting factors `weight1` and `weight2` respectively, the final score of some `NodeA` is:

```
finalScoreNodeA = (weight1 * priorityFunc1) +  
(weight2 * priorityFunc2)
```

After the scores of all nodes are calculated, the node with highest score is chosen as the host of the Pod. If there are more than one node with equal highest scores, a random one among them is chosen.

Scheduler - Ranking (2/2)

- Default ranking strategies:
 - SelectorSpreadPriority
 - InterPodAffinityPriority
 - LeastRequestedPriority
 - BalancedResourceAllocation
 - NodePreferAvoidPodsPriority
 - NodeAffinityPriority
 - TaintTolerationPriority
 - ImageLocalityPriority

Scheduler - Server Configuration

You can change the default scheduler policy by specifying `--policy-config-file` to the kube-scheduler Example `policy.json`:

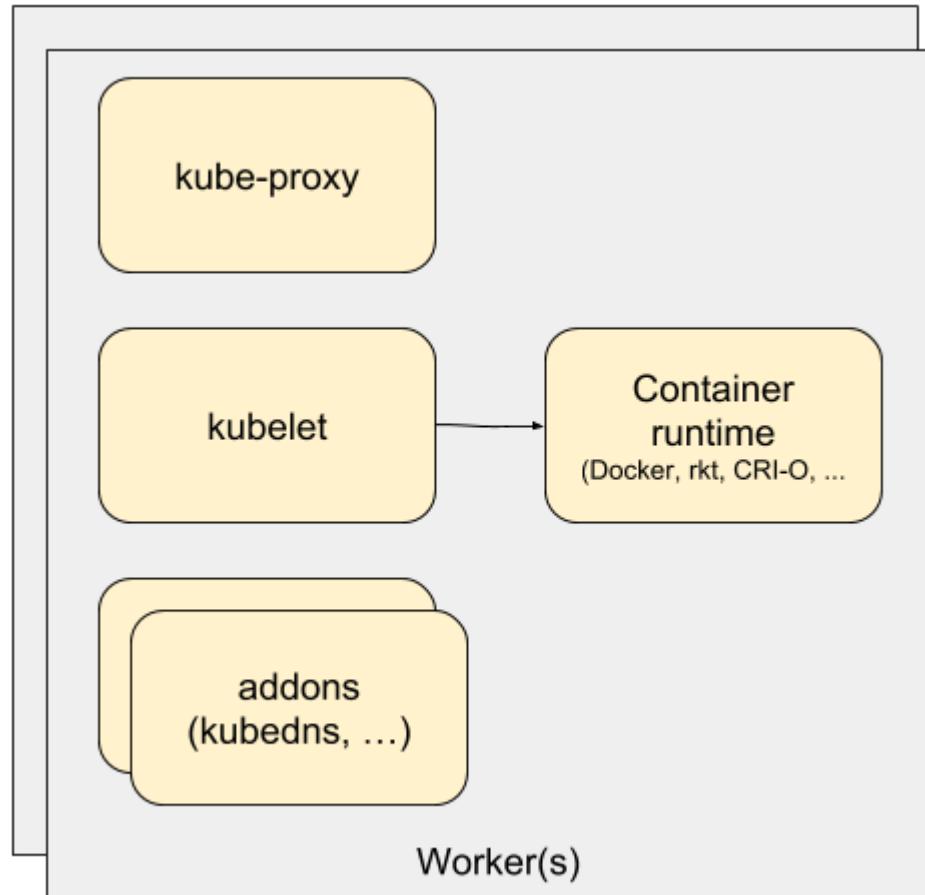
```
{  
    "kind": "Policy",  
    "apiVersion": "v1",  
    "predicates": [  
        { "name": "PodFitsHostPorts" },  
        { "name": "PodFitsResources" },  
    ],  
    "priorities": [  
        { "name": "LeastRequestedPriority", "weight": 1 },  
        { "name": "BalancedResourceAllocation", "weight": 1 },  
    ],  
    "hardPodAffinitySymmetricWeight": 10  
}
```



Controller manager

- Implements the control loop for workloads ([ReplicationController](#), [ReplicaSet](#), [Deployment](#), ...) and other resources
- Runs in active/passive mode
 - Current active controller is flagged in [etcd](#)

Workers



kube-proxy

Network traffic routing accross the nodes depends on the network addon. But egress, ingress of services are taken care of by the **kube-proxy** component.

kube-proxy configures node network for Kubernetes services

- It uses:
 - **iptables**
 - **IPVS** - GA since v1.11

Kubelet (1/2)

- It is the **Node agent**, it runs on a Node and make sure that containers included in assigned **PodSpecs** are running and healthy.
- **PodSpecs** are provided to the **kubelet** by querying the **API Server**
- These communications use a **kubeconfig** like any **API Server** client

Kubelet (2/2)

There are also other ways to provide **PodSpecs** to **kubelet**:

- File: `pod-manifest-path` passed to the command line is monitored every 20s by default
- HTTP Endpoint: `manifest-url` passed to the command line and checked by the kubelet every 20s by default
- HTTP Server: the kubelet expose an API use to interact with **Pods** which can be used to create new **Pods**

Warning: *The HTTP Server on the kubelet allows creation of pods **without authentication**. It should be disabled or at least secured*

Container runtime

The `kubelet` handles container runtimes which implements the **Container Runtime Interface (CRI)**

It supports:

- docker
- containerd
- rkt (deprecated)
- cri-o

It's also possible to use more than one container runtime in parallel. See [Chapter 2 - CRI](#)

Warning: check the compatible/endorsed version of your runtime





Lab 1: Container runtime & Kubelet

Network communications summary

Master node(s)

Protocol	Direction	Port Range	Purpose	Used By
TCP	Inbound	6443	Kubernetes API server	All
TCP	Inbound	2379	etcd client API	kube-apiserver
TCP	Inbound	2380	etcd server peer	etcd
TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	10251	kube-scheduler	Self
TCP	Inbound	10252	kube-controller-manager	Self

Worker node(s)

Protocol	Direction	Port Range	Purpose	Used By
TCP	Inbound	10250	Kubelet API	Self, Control plane
TCP	Inbound	30000-32767	NodePort Services	All



Installation

Table of contents

- Architecture
- *Installation*
- Configuration and operational maintenance
- Cluster Upgrade
- Day to day actions
- Extensibility: Operators, CRD and API Servers
- Federation, Service Mesh, Security

Chapter content

- Installation steps
- Distributions
- Network solutions
- Container Runtimes
- DNS
- Storage

Installation steps - Summary

1. Infrastructure provisioning
2. Certificates creation
3. Kubernetes configuration files
4. Etcd installation
5. Control plane members installation
6. Worker members installation
7. kubectl configuration
8. Addons installation (network, dns, ...)

Installation steps - Preparation

- How do I manage the certificates?
- Local or distant etcd?
- Which specs and operating system for my nodes?
- What size? How many containers?
- Which distribution should I use?
- Which network solution?
- Which container runtime?
- Which dns addon?
- How do I manage the storage provisioning?

Installation steps - Certificate management

- All communications throughout the clusters are encrypted
- Every components should have its own key pair and certificate
- All these certificates are not signed by the same CA

Default CN	Parent CA	kind
kube-etcd	etcd-ca	server, client
kube-etcd-peer	etcd-ca	server, client
kube-etcd-healthcheck-client	etcd-ca	client
kube-apiserver-etcd-client	etcd-ca	client
kube-apiserver	kubernetes-ca	server
kube-apiserver-kubelet-client	kubernetes-ca	client
front-proxy-client	kubernetes-front-proxy-client	client



Lab 2.1: Certificates

Installation steps - Etcd

- Kubernetes can use an external etcd cluster
 - etcd nodes must be installed and managed
- Or use an embedded one, hosted in the control plane nodes
 - etcd nodes are tied to control plane nodes
 - run as static pod
- Etcd node count is not meant to be changed
 - Think ahead before choosing how many etcd **Node** you use
 - Usually 3 or 5

Installation steps - Node specs

- **Virtual or physical** machines
- x86_64 architecture
- **Api-server** and **etcd** need at least 1 core and 1GB memory for 10s of nodes
- Nodes **do not** have to be identical (CPU, Memory, ...)

Installation steps - Node OS

- x86_64 Operating system
- Compatible with chosen container runtime
- With a well defined and applied upgrade strategy (especially for the kernel which is shared)
- Check compatibility matrix of the chosen distribution

e.g. for **kubeadm**:

- *Ubuntu 16.04+*
- *Debian 9*
- *CentOS/RHEL 7*
- *Fedora 25/26 (best-effort)*
- *...*

Cluster sizing - High availability considerations

Component	role	effect of loss	recommended instances
etcd	maintains state of all Kubernetes objects	Loss of storage catastrophic. Loss of majority = Kubernetes loses control plane, API Server depends on etcd, read-only API calls not requiring a quorum may work, and existing workloads may continue to run.	odd number, 3+
API Server	provides API used internally and externally	Unable to stop, start, update new pods, services, replication controller. Scheduler and Controller Manager depend on API Server. Workloads continue if they are not dependent on API calls (operators, custom controllers, CRDs, etc.)	2+
kube-scheduler	places pods on nodes	No pod placements, priority and preemption	2+
kube-controller-manager	runs many controllers	Core control loops that regulate state cease, in-tree cloud provider integration breaks.	2+
cloud-controller-manager (CCM)	Integration for out-of-tree cloud providers	Cloud provider integration breaks	1
Add-ons (e.g., DNS)	varies	varies	Depends on add-on, e.g. 2+ for DNS

Cluster sizing - Nodes number

When creating a cluster, you should respect these constraints:

- No more than 5000 nodes
- No more than 150000 total pods
- No more than 300000 total containers
- No more than 100 pods per node

Distributions

More than 50 Kubernetes **platforms** and **distributions**

- On premises:
 - openshift (ocp/okd)
 - kubeadm, kube-spray, kops
 - tectonic
 - Rancher 2
- Managed:
 - GKE (Google)
 - AKS (Azure)
 - EKS (Amazon)
 - Ovh Kubernetes-as-a-service

Distributions - kubeadm (1/2)

- Distributed as a single go binary
- Maintained by the Kubernetes community
- Since **Kubernetes v1.11** can deploy a **High Availability** cluster
- Follows the Kubernetes release cycle

Distributions - kubeadm (2/2)

- kubeadm will:
 - Generate certificate chains
 - Deploy kubelet on control plane/worker nodes
 - Deploy etcd cluster member and control plane components as static pods
 - Provide a simple way to join worker nodes to the cluster
- kubeadm won't:
 - Create VMs
 - Configure network infrastructure
 - Install container runtime on machines

Distributions - kubespray (1/2)

- A set of **ansible playbooks**
- In **kubernetes-incubator**
- Not synchronized with Kubernetes release cycle but master up to date

Distributions - kubespray (2/2)

- kubespray will:
 - Generate certificate chains
 - Deploy etcd on control plane nodes
 - Deploy kubelet on control plane/worker nodes
 - Deploy control plane components as static pods
 - Provide some playbook to add nodes later
- kubespray won't:
 - Create VMs
 - Configure network infrastructure
 - Install container runtime on machines

Distributions - kops (1/2)

- A tool distributed as a single binary
- Maintained by the Kubernetes community
- Not synchronized with Kubernetes release cycle
- Create a cluster on **AWS**, **GCE** or **DigitalOcean**
- VMware vSphere support in progress

Distributions - kops (2/2)

- kops will:
 - Create VMs
 - Configure network infrastructure
 - Install container runtime on machines
 - Generate certificate chains
 - Deploy etcd on control plane nodes
 - Deploy kubelet on control plane/worker nodes
 - Deploy control plane components as static pods
 - Provide some playbook to add nodes later
- kops won't:
 - Work on every on-premises configurations

Distributions - Openshift

- A complete and highly integrated Kubernetes distribution created by **Redhat**
- Available:
 - with Redhat support: **Openshift Container Platform**
 - as Opensource: **OKD**
- Includes:
 - a docker registry
 - metrics with Hawkular or Prometheus
 - CI/CD with jenkins integration
 - a kind of PaaS with S2I

Distributions - Rancher 2

- Rancher now exclusively deploys and manages multiple Kubernetes
- Kubernetes deployments on bare-metal and cloud
- Existing cluster management
- Includes:
 - complete UI
 - RBAC and multi tenancy
 - Monitoring and alerting
 - Application catalog (helm)
 - (simple) Pipeline engine

Distributions - Summary (1/2)

Distribution	HA	Supported	Supported network addons	Container Runtime	Registry
kubeadm	Yes [1]	Community	Calico, Canal, Flannel, Kube-router, Romana, Weave, JuniperContrail/TungstenFabric	docker (cri-containerd), cri-o, frakti, rkt	No
kubespray	Yes	Community	Calico, Canal, Flannel, Cilium, Contiv, Weave	docker, rkt [2]	No
kops	Yes	Community	Calico, Canal, Flannel, kopeio-vxlan, kube-router, Cilium, romana, weave, amazon-vpc-routed-eni	docker	No
openshift	Yes	Redhat	Openshift *, Flannel, Nuage Networks	docker, cri-o	Yes
rancher 2	Yes	Rancher Labs	Calico, Canal, Flannel	docker	No
tectonic	Yes	CoreOS/Redhat	Calico, Flannel	docker	Yes (Quay)
PKS	Yes	Pivotal	Flannel, NSX-T	docker	Yes (Harbor)

[1]: Introduced in kubeadm 1.11, relies on an external LB

[2]: Only on control plane nodes, workloads still run on docker

Distributions - Summary (2/2)

Decision helpers:

- Do you need enterprise support?
- Do you need UI for your end users?
- Do you want to manage your control plane?
- Is it easy to setup? to maintain?
- Do you want to run Kubernetes on premise?
- Which cloud provider is the best for your needs? (costs / auto upgrade / geographical area / ...)

Distributions - CNCF Conformance test

The CNCF provides a ***conformance test suite***

This test suite is developed by Heptio, the source code and docs are available on github

It is provided as a binary which deploys resources on the cluster testing expected behaviour of controllers, scheduler, etc.

sonobuoy run

View tests progress and logs

sonobuoy status

sonobuoy logs

It produces a report available through this command:

sonobuoy retrieve .



Lab 2.2: Installation

Network solutions

As stated in the [official Kubernetes documentation](#):

Kubernetes imposes the following fundamental requirements on any networking implementation (barring any intentional network segmentation policies):

- *all containers can communicate with all other containers without NAT*
- *all nodes can communicate with all containers (and vice-versa) without NAT*
- *the IP that a container sees itself as is the same IP that others see it as*

To fulfill these requirements two kinds of network plugins can be used:

- **kubenet** is the default and doesn't itself implements cross-node networking
- **cni plugin** and deploy a network addon on the cluster

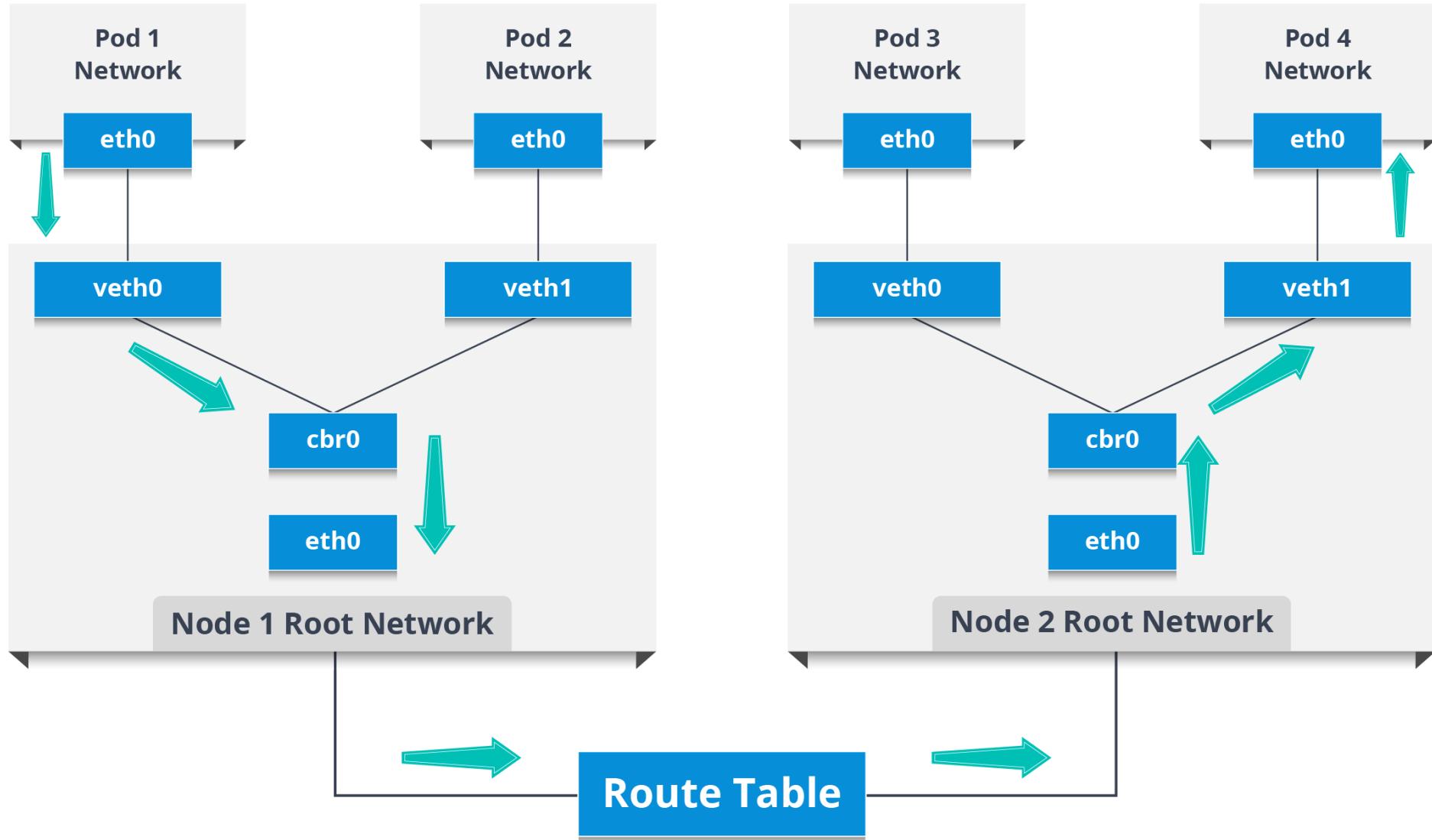
Network solutions - Reminders (1/2)

Layer		Protocol Data Unit (PDU)	Protocol Examples	Function
L7	Application	Data	HTTP	<u>High-level APIs, including resource sharing, remote file access</u>
L6	Presentation			Translation of data between a networking service and an application; including character encoding, data compression and encryption/decryption
L5	Session			<u>Managing communication sessions, i.e. continuous exchange of information in the form of multiple back-and-forth transmissions between two nodes</u>
L4	Transport	Segment, Datagram	TCP/UDP	Reliable transmission of data segments between points on a network, including segmentation, acknowledgement and multiplexing
L3	Network	Packet	IP	Structuring and managing a multi-node network, including addressing, routing and traffic control
L2	Data link	Frame	Ethernet	Reliable transmission of data frames between two nodes connected by a physical layer
L1	Physical	Symbol		Transmission and reception of raw bit streams over a physical medium

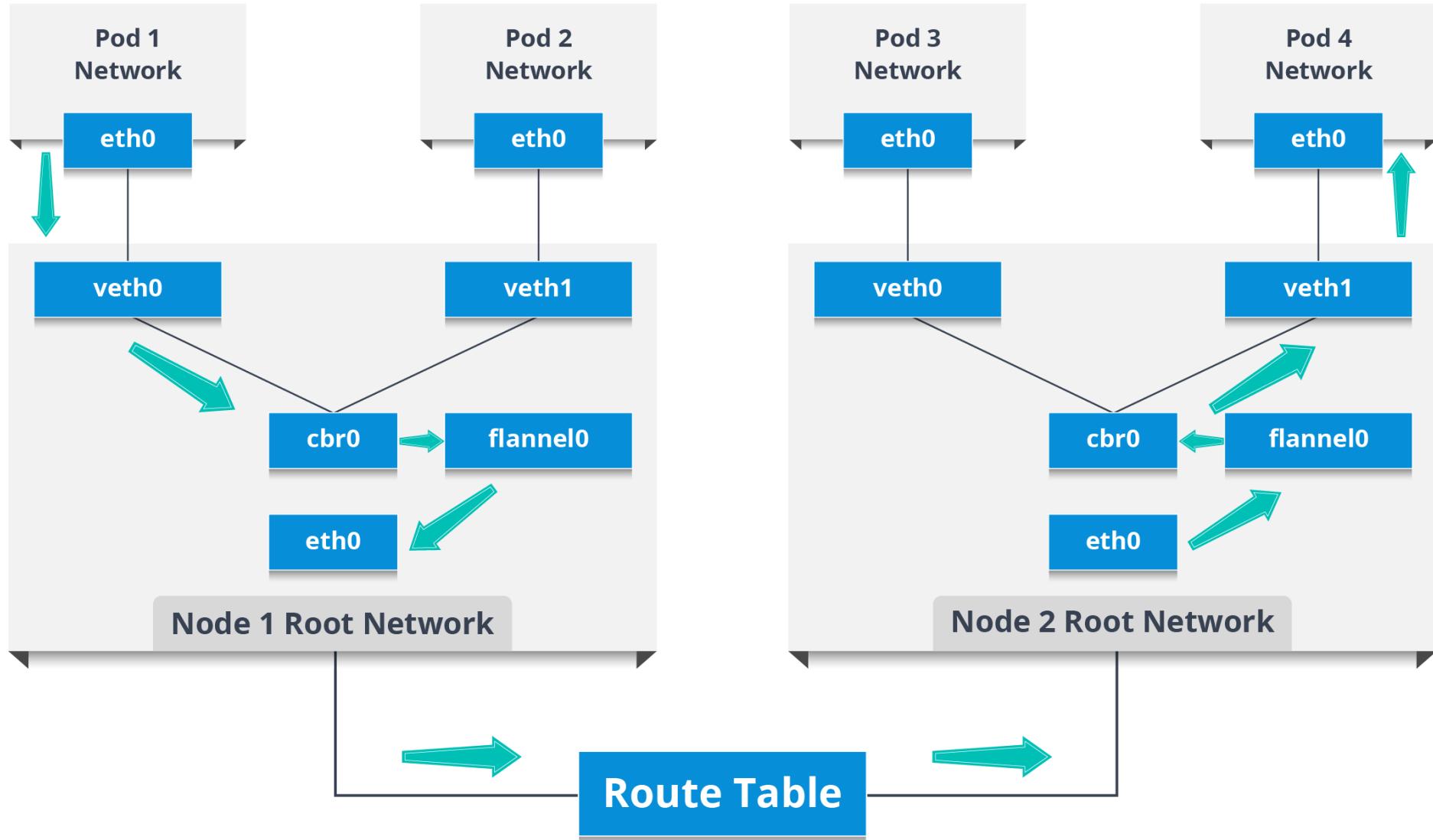
Network solutions - Reminders (2/2)

- **Overlay network**: An overlay network decouples network services from the underlying infrastructure by encapsulating one packet inside of another packet
- **BGP**: Border Gateway Protocol is a protocol used to exchange routing and reachability information
- **Vxlan**: Virtual Extensible LAN encapsulates L2 frames within L4 UDP datagrams
- **eBPF**: extended Berkeley Packet Filter provides advanced rules to filter traffic without leaving kernel space

Network solutions - Native routing



Network solutions - Overlay network



Network solutions - Kubenet

- Kubenet creates a `cbr0` network bridge interface
- All created pods are then affected a veth pair and the pod end of the pair is assigned an IP address allocated from a configured range

But **kubenet** comes with some requirements. Either:

1. A route declaration exists for each node pods CIDR range (GKE is implemented this way)
2. All interfaces are on the same ethernet switching infrastructure (L2) **but**
'it seems to work, but has not been thoroughly tested'

Network solutions - CNI (1/3)

There are many addons supporting **CNI** and providing the needed network connectivity to your cluster

- Flannel
- Weave net
- Project Calico
- Canal
- Cilium
- Romana
- ...
- All of these use some mechanic (initContainers, ...) to put the CNI plugin binary on the kubelet host
- Each of these comes with a different approach, features and performances

Network solutions - CNI (2/3)

A CNI plugin is a simple command invoked by the kubelet which should handle 4 event types:

- ADD: sent when a new container is created, the plugin has to setup the network namespace, interface, routes, ...
- DEL: sent when a container is deleted
- GET: used to get informations on a previously created container (currently not used by kubelet)
- VERSION: used to get the supported CNI versions

The event type and payloads are passed through environment variables

Network solutions - CNI (3/3)

To install a CNI Plugin, you must:

1. Put its config file in `/etc/cni/net.d/`
2. Put its executable command in `/opt/cni/bin`

Configuration example for Flannel:

```
{  
  "name": "cbr0",  
  "plugins": [{"  
    "type": "flannel",  
    "delegate": {  
      "hairpinMode": true,  
      "isDefaultGateway": true  
    }  
  }]  
}
```

Network solutions - Flannel

Flannel is a simple network overlay from **CoreOS**.

- It supports different backends:
 - to encapsulate traffic between nodes (vxlan, udp)
 - to route traffic between nodes on the same L2 network (host-gw)

Flannel stores its state in **etcd**

It doesn't support **Network Policies**

Network solutions - Project Calico

Project Calico provides network connectivity for various platforms (Kubernetes, Openstack, Mesos, ...)

- Uses BGP to enable routing between PODs
- Supports IP in IP encapsulation if the network drops traffic to/from unknown addresses
- Supports network policies

Network solutions - Cilium

Cilium uses eBPF to leverage a high performance network solution

- Stores its state in **etcd**
- Supports overlay network (default) or native routing
- Default Kubernetes network policies + cilium specifics available

Network solutions - WeaveNet

WeaveNet provides an overlay network using vxlan

- Doesn't store on etcd, but on a local named volume
- Supports network policies

Network solutions - Canal

Canal combines Flannel and Calico

- Flannel provides the overlay or non-overlay network connectivity options
- Calico provides network policies enforcement

Network solutions - CNI Summary

Solution	Encapsulation	Network Policies	IPv6	State
Flannel	vxlan	No	No	etcd
Calico	IP in IP	Yes	Yes	etcd
WeaveNet	vxlan	Yes	Yes	node local
Cilium	vxlan, geneve	Yes + specifics	Yes	etcd
Canal	vxlan	Yes	Yes	etcd

Decision helpers:

- Do you need network policies?
- Is the backend solution compatible with your existing network?
- Is the addons supported in your Kubernetes distribution?



Lab 2.3: Network solution

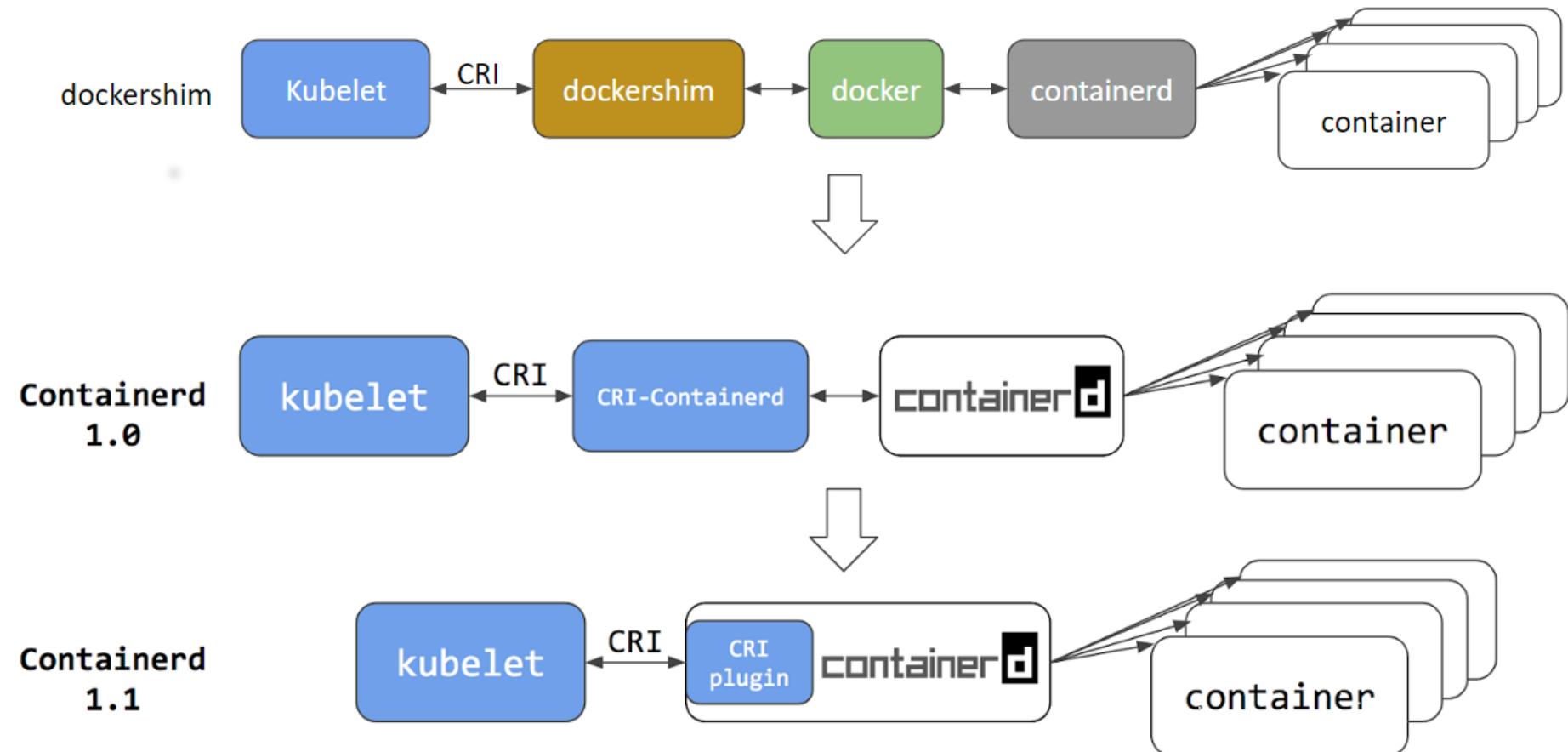
Container Runtimes

At the beginning of the **Kubernetes** project, **Docker** was the standard and only supported container runtime. With the **OCI standards** and other container runtimes being created, the community decided to create the **Container Runtime Interface (CRI)**.

This allowed new compatible runtimes to be created and used with Kubernetes:

- Containerd
- CRI-O
- RKT
- gVisor
- frakti

Container Runtimes - Containerd



Container Runtimes - CRI-O

cri-o was designed specifically to be used as the Kubernetes Container Runtime.

The project began at **Redhat** in 2016.

cri-o is considered stable as of Kubernetes 1.9 and has been used to run Openshift Online Platform since August 2018.

It can use any OCI runtime that implements the OCI runtime spec and defaults to using runc.

Container Runtimes - Others

There are a few other runtimes available for Kubernetes

- **gvisor**: a container runtime using a userspace implementation of syscalls to isolate untrusted/unsafe workload from the kernel
- **rkt**: another container runtime, not really usable in production with Kubernetes
- **frakti** (KataContainers): an hypervisor based runtime

Container Runtimes - Kubelet (1/2)

kubelet is the agent that runs on each nodes and manage **Pods** and container lifecycle. It comes with a default configuration from the Kubernetes distribution, but may be configured.

```
$ kubelet -h
Usage:
  kubelet [flags]

Flags:
      --alsologtostderr                                log to standard error
as well as files
      --bootstrap-kubeconfig string                   Path to a kubeconfig
file that will be used to get client certificate for kubelet. If the file
specified by --kubeconfig does not exist, the bootstrap kubeconfig is used to
request a client certificate from the API server. On success, a kubeconfig file
referencing the generated client certificate and key is written to the path
specified by --kubeconfig. The client certificate and key file will be stored in
the directory pointed by --cert-dir.
      [...]
```

Container Runtimes - Kubelet CRI-O

Kubelet parameters are stored in /etc/kubernetes/kubelet.env file.

```
# cat /etc/kubernetes/kubelet.env | grep KUBELET_ARGS
KUBELET_ARGS="--pod-manifest-path=/etc/kubernetes/manifests
--pod-infra-container-image=gcr.io/google_containers/pause-amd64:3.0
--cluster-dns=10.233.0.3 --cluster-domain=cluster.local
--resolv-conf=/etc/resolv.conf --kubeconfig=/etc/kubernetes/node-kubeconfig.yaml
--require-kubeconfig"
```

To use **CRI-O**, the following additional parameters to **KUBELET_ARGS** are required:

```
--experimental-cri=true - Use Container Runtime Interface.release.
--container-runtime=remote - Use remote runtime with provided socket.
--container-runtime-endpoint=unix:///var/run/crio/crio.sock - Socket for remote
runtime (default crio socket localization).
--runtime-request-timeout=10m - Optional but useful. Some requests, especially
pulling huge images, may take longer than default (2 minutes) and will cause an
error.
```

DNS (1/2)

kubedns is a custom component used to provide service name resolution inside the cluster.

As of Kubernetes **1.10** this component was made pluggable and **CoreDNS** introduced as beta.

Started in Kubernetes **1.11**, **CoreDNS** became the default for **kubeadm** replacing **kubedns**.

It's still possible to use kubedns

e.g with kubeadm:

```
kubeadm init --feature-gates=CoreDNS=false
```

DNS (2/2)

- **CoreDNS** provides new features:
 - Making an alias for an external name
 - Dynamically adding services to another domain, without running another server
 - Adding an arbitrary entry inside the cluster domain



Lab 2.4: DNS

Storage

With Kubernetes, storage is provided to pods by **PV/PVC**.

To satisfy the PVCs, at least one **StorageClass** is needed.

A **StorageClass** definition must contain at least these informations:

- provisioner: the plugin used for provisioning PVs. This field is mandatory.
- parameters: used to pass custom parameters to the underlying plugin. These are provisioner specific.
- reclaimPolicy: defines the reclaimPolicy for PVs of this **StorageClass**. Default is delete.

Storage - Provisioners

There are many options when choosing a storage provisioner

- On premise:
 - Cinder (OpenStack block storage)
 - GlusterFS / CephFS / Rook
 - Portworx / StorageOS
 - NetApp Trident
 - NFS
 - ...
- Cloud solutions:
 - AWSElasticBlockStore
 - AzureDisk, AzureFile
 - GCEPersistentDisk

Storage - StorageClass example

GlusterFS StorageClass example:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/glusterfs
parameters:
  resturl: "http://127.0.0.1:8081"
  clusterid: "630372ccdc720a92c681fb928f27b53f"
  restauthenabled: "true"
  restuser: "admin"
  secretNamespace: "default"
  secretName: "heketi-secret"
  gidMin: "40000"
  gidMax: "50000"
  volumetype: "replicate:3"
```

Storage - Reclaim policy

- When a volume is no longer needed, the associated **PVC** can be deleted
- The **ReclaimPolicy** of the **PersistentVolume** define what happens to the **PersistentVolume** once freed :
- Freed **PVs** can be :
 - **Retained**: the **PV** is no longer used but can't be bound to another **PersistentVolumeClaim** until an operator release it
 - **Recycled** (deprecated): the **PersistentVolume** is cleaned, i.e. datas are deleted. Once cleaned it can be associated to a new **PVC**
 - **Deleted**: the **PersistentVolume** is deleted (e.g. AzureDisk, GC Disk, AWS EBS...)

Storage - Access Modes

- **ReadWriteOnce** (RWO): the volume can be mounted as read-write by a single node
- **ReadOnlyMany** (ROX): the volume can be mounted read-only by many nodes
- **ReadWriteMany** (RWX): the volume can be mounted as read-write by many nodes

⚠ A volume can only be mounted using one access mode at a time, even if it supports many



Lab 2.5: Storage



Configuration and operational maintenance

Table of contents

- Architecture
- Installation
- *Configuration and operational maintenance*
- Cluster Upgrade
- Day to day actions
- Extensibility: Operators, CRD and API Servers
- Federation, Service Mesh, Security

Chapter content

- Logs centralization
- Metrics centralization
- Backup / Restore
- Tools for capacity planning (node resources consumption, namespaces, requests, limits)
- Garbage Collection
- Connectivity of a Kubernetes cluster with the rest of your infrastructure
- LoadBalancers
- Ingress
- Schedulers
- Troubleshooting the cluster

Logs centralization - Why

Why should we centralize logs?

Kubernetes deploy pods an available nodes:

- There is no way to predict where your application will be running
- ssh into the machine or use a network share to access the logs are not viable options

Instead, each running process writes its event stream, unbuffered, to stdout [...] each process' stream will be captured by the execution environment, collated together with all other streams from the app, and routed to one or more final destinations for viewing and long-term archival. -- <https://12factor.net/logs>

Logs centralization - How

Kubernetes doesn't provide this out of the box, but:

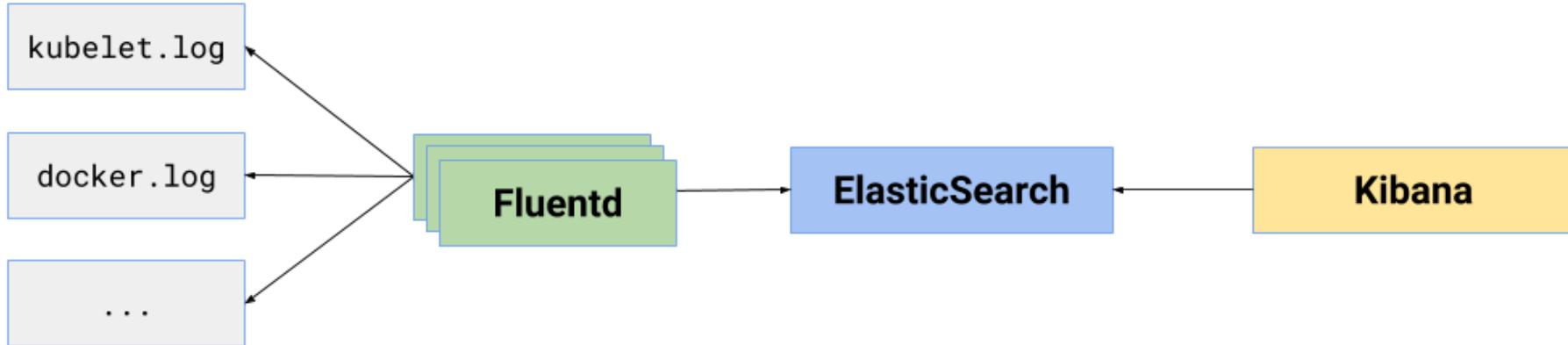
- Kubernetes hosted by cloud providers usually come with their own solution:
 - **Stackdriver** for **Google Cloud**
 - **Log Analytics** for **Azure**
- Some distributions provide it:
 - **OpenShift** provides an integrated **EFK** stack
 - **Rancher 2.0** provides a pluggable architecture and supports many solutions (embedded ElasticSearch, external ElasticSearch, Splunk, Kafka, Syslog)
- On-premises you are on your own (nearly):
 - There are many docs and articles explaining how to use **EFK** with **Kubernetes**

Logs centralization - What

There are multiple logs producers to address when you run Kubernetes:

- Kubernetes components:
 - Kubelet/Node
 - Kube API Server
 - Kube Controller manager
 - Kube Scheduler
- Kubernetes addons (deployed as usual workloads most of the time):
 - Network plugins
 - Ingress controllers
- Hosted workloads
 - All user deployed pods

Logs centralization - EFK



These components are used to:

- **Fluentd**: Collect logs data from files or other sources...
- **ElasticSearch**: Store, index and expose logs
- **Kibana**: Display logs, search, create specific dashboards

Logs centralization - EFK Deployment

- **Fluentd** should be deployed as a **DaemonSet**
- **ElasticSearch** can be deployed :
 - inside the cluster as a **StatefulSet** or as an **Operator** (covered in chapter 6)
 - outside the cluster, using an externally managed instance
- **Kibana** can be deployed as a standard **Deployment**

Logs centralization - Fluentd configuration

Fluentd configuration uses mainly three directives:

- **source**: determines the input sources
- **match**: determines the output destinations
- **filter**: determines the event processing pipelines

Logs centralization - Fluentd source

Example source configuration for container logs:

```
# CRI Log Example:  
# 2016-02-17T00:04:05.931087621Z stdout F [info:2016-02-16T16:04:05.930-08:00]  
Some log text here  
<source>  
  @id fluentd-containers.log  
  @type tail  
  path /var/log/containers/*.log  
  pos_file /var/log/es-containers.log.pos  
  tag raw.kubernetes.*  
  read_from_head true  
<parse>  
  @type multi_format  
  <pattern>  
    format /^(<time>.+) (<stream>stdout|stderr) [<log>.*]$/  
    time_format %Y-%m-%dT%H:%M:%S.%N%:z  
  </pattern>  
</parse>  
</source>
```

Source: <https://github.com/kubernetes/kubernetes/tree/master/cluster/addons/fluentd-elasticsearch>

Logs centralization - Fluentd filter

Example filter configuration:

```
# Enriches records with Kubernetes metadata
<filter kubernetes.**>
  @type kubernetes_metadata
</filter>
```

The **Kubernetes** fluentd plugin is used to extract the namespace, pod name & container name which are added to the log message as a Kubernetes field object & the Docker container ID is also added under the **docker** field object.

Logs centralization - Fluentbit

As an alternative deployment pattern, you could use **Fluentbit** along with **Fluentd**:

- Designed to be a very lightweight forwarder
- Deployed on each nodes
- Collects and forwards logs to a **Fluentd** instance
- Adds pods/namespace/container metadata with the right plugin

The **Fluentd** instance then acts as an aggregator.

Logs centralization - Cluster

Additional tasks for configuring fluentd:

- Check docker (or the container runtime) log config
- Check kubelet log config
- Check control plane components log config
- Write a **ConfigMap** with all needed files





TP 3.1 : EFK Deployment

Metrics centralization

Metrics are split into two pipelines for monitoring Kubernetes:

- A **core metrics pipeline**:
 - Used by core components (e.g. scheduler and horizontal pod autoscaling)
 - Not meant to be integrated with third-party monitoring systems
 - Previously implemented by **Heapster**, now by **Kubernetes** components (Kubelet/cAdvisor, metrics-server)
 - Served by the **API Server**
- A **monitoring pipeline**:
 - Used for collecting various metrics and exposing them to end-users
 - Provided by third-party components
 - Opensource Kubernetes does not ship with a monitoring pipeline
 - Can be served by the **API Server**

Metrics centralization - Heapster/Metrics server

- **Heapster**
 - Before Kubernetes 1.11, `kubectl top` exposed metrics from **Heapster** addon
 - As of Kubernetes 1.11, **Heapster** is deprecated and is removed in 1.13 [1]
- **Metrics server**
 - As a replacement users are encouraged to use metrics-server instead, potentially supplemented by a third-party monitoring solution, such as Prometheus
 - The metrics server adds the `metrics.k8s.io/v1beta1` API to your cluster using the aggregation layer (covered in [Chapter 6](#))

Metrics centralization - Heapster

Using **Heapster**, metrics centralization is based on a **Deployment** and exposition on a **Service** named **heapster**.

Heapster supports different backends for storing datas (influxdb, standalone, ...).

See [Running Heapster on Kubernetes](#)

The `kubectl` command uses this service to gather **Pod** and **Node** metrics.

```
kubectl top nodes
```

```
kubectl top pods
```

Metrics centralization - Metrics server

Metrics server goals:

- Provide access to cluster metrics the same way as other API objects
- Storing metrics as their lifecycle differs from other objects and shouldn't be stored in **etcd**

Other considerations:

- Metrics are required for some Kubernetes components (HPA, scheduler, kubectl top), so it will be run by default in all clusters.
- Only one instance running on the cluster, vertically autoscaled by addon-resizer
- No long term storage of metrics, such use case should be addressed with a **Monitoring pipeline**

Metrics centralization - Monitoring pipeline - Why?

Why do you need a separated monitoring pipeline?

- Flexible monitoring pipeline, as core **Kubernetes** components do not need to rely on it
- Data collected by the monitoring pipeline may differ from those used by **Kubernetes** cluster
- Expose custom metrics to **Horizontal Pod Autoscaler**

Metrics centralization - Collected datas

Example collected datas:

- core system metrics
- non-core system metrics
- service metrics from user application containers
- service metrics from Kubernetes infrastructure containers; these metrics are exposed using Prometheus instrumentation



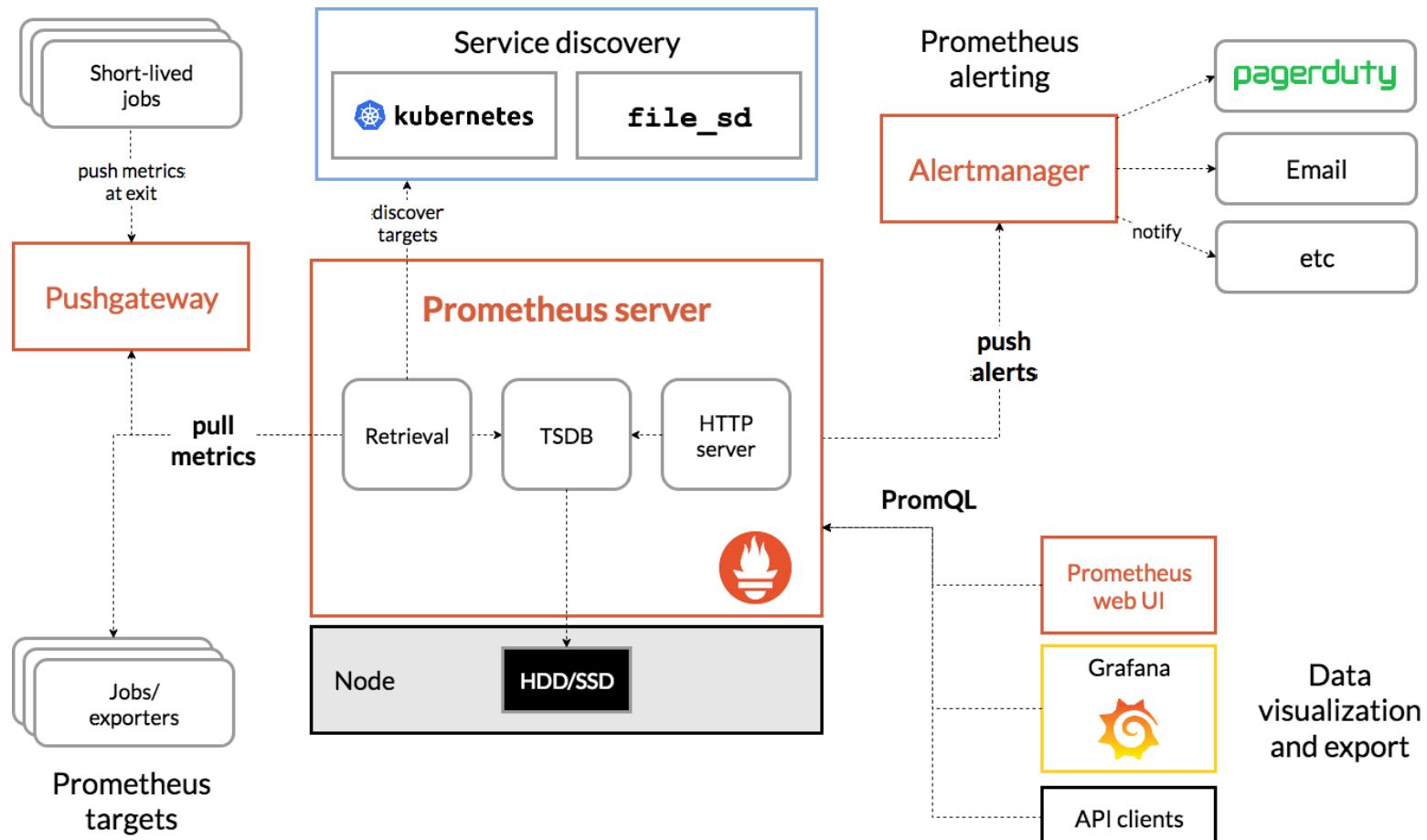
Metrics centralization - Prometheus overview

Prometheus is an **Opensource** project comprising many subprojects:

- A timeseries database
- A dedicated query language: **PromQL**
- A "standard" metric format: **OpenMetrics**
- Exporters and libraries to expose metrics from many languages/frameworks/applications
- An **alertmanager**

Metrics are scraped (or pulled) from an http(s) endpoint

Metrics centralization - Prometheus



Source:

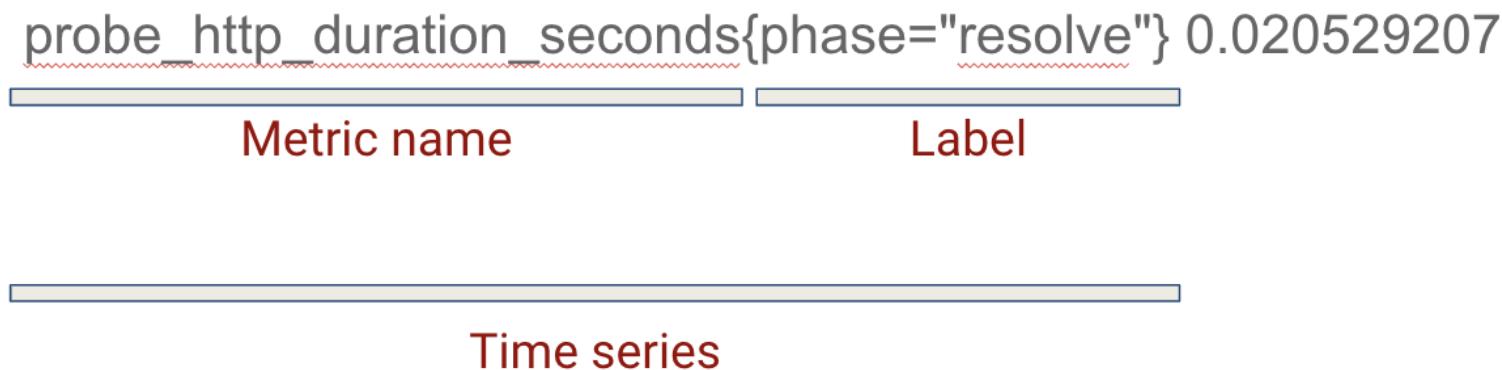
<https://prometheus.io/docs/introduction/overview/>

Metrics centralization - Prometheus metric types

Basic counters	Sampling counters
Counter	Histogram
Gauge	Summary

Metrics centralization - Prometheus metric example

```
# HELP http_requests_total Total number of HTTP requests made.  
# TYPE http_requests_total counter  
http_requests_total{code="200",handler="graph",method="get"} 2  
http_requests_total{code="200",handler="label_values",method="get"} 2  
http_requests_total{code="200",handler="prometheus",method="get"} 1072  
http_requests_total{code="200",handler="query",method="get"} 2  
http_requests_total{code="200",handler="static",method="get"} 46
```



Metrics centralization - Prometheus sources

- Exporters:
 - **Hardware/Host**: Node exporter, IBM Z HMC exporter, ...
 - **Webserver**: Apache, nginx, ...
 - **Messaging systems**: Kafka, RabbitMQ, ...
 - **Storage**: Ceph, Gluster, Hadoop HDFS, ...
 - **Databases**: Mongodb, MSSQL, MySQL, Redis, PostgreSQL, Oracle DB, ...
- Software exposing Prometheus metrics:
 - **Kubernetes components**: Kube-apiserver, kube-state-metrics, kubelet, kube-controller-manager, kube-scheduler
 - Others: etcd, Minio, Traefik, Weave Flux, ...

Metrics centralization - Prometheus deployment

The recommended way to deploy Prometheus on **Kubernetes** is the **Prometheus Operator**. It uses Custom Resource Definitions (CRD) to declare desired state:

- **Prometheus**: defines a desired Prometheus setup to run in the cluster
- **ServiceMonitor**: defines how a dynamic set of services should be monitored using a selector
- **AlertManager**: defines a desired Alertmanager setup to run in the cluster
- **PrometheusRule**: defines a desired Prometheus rule to be consumed by one or more Prometheus instances.

We'll cover Operators and CRD on **Chapter 6**

Metrics centralization - Prometheus CRD

Example Prometheus CRD:

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: prometheus
spec:
  serviceMonitorSelector:
    matchLabels:
      team: frontend
  version: v2.5.0
```

serviceMonitorSelector lets you specify which **ServiceMonitor** will be scraped by this instance

Metrics centralization - Prometheus Service Monitor

Example Service Monitor CRD:

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: example-app
  labels:
    team: frontend
spec:
  selector:
    matchLabels:
      app: example-app
  endpoints:
  - port: web
```

- The label selector is used to select **Services** and their underlying **Endpoint** objects
- The endpoint port specifies the port on which the metrics are exposed



TP 3.2 : Prometheus Deployment

Backup / Restore (1/2)

Why?

- **Disaster recovery**: to be able to restore the cluster state if the **etcd** cluster gets corrupted or lost
- **Upgrade tests**: to create full duplicated environments to test cluster or critical components upgrade or restore a partial state to test
- **Environment replication**: restore full or partial state to replicate some your setup on another cluster
- **Scheduled migration**: to migrate from one cluster to another

Backup / Restore (2/2)

What?

- Root certificate and key
- Etcd datas
- Volume datas

How?

- With stateless workloads:
 - the common way is to snapshot **etcd** state
 - another possibility is to export all kube resources (pods, deploy, ...)
- However, with stateful workloads, volume datas should also be backed up.

Backup with etcd (1/2)

Make sure you're able to connect to etcd:

```
ETCDCTL_API=3 etcdctl --cacert=ca.crt --cert=client.crt --key=client.key \
--endpoints https://ETCD_HOSTNAME:2379 endpoint status
```

- **ETCDCTL_API=3**: Communications with **etcd** should use the v3 API
- **--cacert=ca.crt**: CA certificate used to trust **etcd** https endpoint
- **--cert=client.crt, --key=client.key**: client key pair, signed by the CA
- **--endpoints https://ETCD_HOSTNAME:2379**: **etcd** is reachable at **ETCD_HOSTNAME:2379**
- **endpoint status**: Get a status of the cluster (members, state, ...)

Backup with etcd (2/2)

Then create a snapshot of **etcd** state:

```
ETCDCTL_API=3 etcdctl --endpoints $ENDPOINT snapshot save snapshotdb
```

And verify it:

```
ETCDCTL_API=3 etcdctl --write-out=table snapshot status snapshotdb
+-----+-----+-----+
| HASH | REVISION | TOTAL KEYS | TOTAL SIZE |
+-----+-----+-----+
| fe01cf57 | 10 | 7 | 2.1 MB |
+-----+-----+-----+
```

Note: certificate parameters are omitted for clarity

Restore with etcd

Restoring **etcd** state from a snapshot resets membership metadata (member ID and cluster ID); the member loses its former identity

The following command initializes a new cluster with **snapshot.db**, it must be run on each host

```
ETCDCTL_API=3 etcdctl snapshot restore snapshot.db \
  --name m1 \
  --initial-cluster
m1=http://host1:2380,m2=http://host2:2380,m3=http://host3:2380 \
  --initial-cluster-token etcd-cluster-1 \
  --initial-advertise-peer-urls http://host1:2380
```

Then **etcd** is ready to be started with this command (again on each host):

```
etcd \
  --name m1 \
  --listen-client-urls http://host1:2379 \
  --advertise-client-urls http://host1:2379 \
  --listen-peer-urls http://host1:2380 &
```

Backup/Restore with **velero** (Heptio)

Velero gives you tools to back up and restore your **Kubernetes cluster resources** and **persistent volumes**. Velero lets you:

- Take backups of your cluster and restore in case of loss.
- Copy cluster resources to other clusters.
- Replicate your production environment for development and testing environments.

Velero consists of:

- A **server** that runs on your cluster
- A **command-line client** that runs locally

Backup/Restore - velero compatibility

Velero will use a **storage provider** to **store the backups**:

- AWS S3
- Azure Blob Storage
- Google Cloud Storage

And supports these **storage solutions** for volume **snapshots**:

- AWS EBS
- Azure Managed Disks
- Google Compute Engine Disks
- Restic
- Portworx
- DigitalOcean

Backup/Restore - velero installation

Get the latest release for your platform:

<https://github.com/heptio/velero/releases>

This package includes:

- descriptors for deploying server part (example for minio setup):
 - Service accounts, CRD,: `config/common/00-prereqs.yml`
 - Minio deployment: `minio/00-minio-deployment.yml`
 - Velero server configuration: `minio/05-ark-backupstoragelocation.yml`
 - Velero server deployment: `minio/20-ark-deployment.yml`
 - Restic daemonset: `minio/30-restic-daemonset.yml`
- `velero` cli command

Backup/Restore - velero usage

Create a backup for any object that matches the `app=nginx` label selector:

```
velero backup create nginx-backup --selector app=nginx
```

Create regularly scheduled backups based on a cron expression using the `app=nginx` label selector:

```
velero schedule create nginx-daily --schedule="0 1 * * *" --selector app=nginx
```

Restore your lost resources:

```
velero restore create --from-backup nginx-backup
```



TP 3.3 : Backup/Restore

Tools for capacity planning

The easiest way to see cluster free and used capacity is to use `kubectl top nodes` or `kubectl top pods` when one of **Heapster** or **Metrics server** is deployed.

NAME	CPU(cores)	CPU%	MEMORY(bytes)	MEMORY%
node-1	39m	4%	1216Mi	46%
node-2	41m	4%	669Mi	25%
node-3	55m	5%	1342Mi	50%

However, this doesn't help with **capacity planning**:

- Only shows instant usage
- Doesn't show or enforce threshold on resources used

Tools for capacity planning - Requests/Limits

To manage **Capacity planning** Operators need workloads to declare necessary resources (CPU, memory).

To do so, **Kubernetes API** uses two concepts:

- **Requests**: Describes the **minimum amount** of compute resources **required**
- **Limits**: Describes the **maximum amount** of compute resources **allowed**

Tools for capacity planning - Requests

- **Requests**
 - Used by the **Scheduler** to ensure **Pod** fits on a **Node**
 - The **Pod** remains in the **PENDING** state as long as the resource request cannot be satisfied

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
too-much-cpu  0/1     Pending   0          4s
```

```
$ kubectl describe pods | tail -n 4
Events:
  Type      Reason          Age           From            Message
  ----      ----          ----         ----           -----
  Warning   FailedScheduling 11s (x8 over 75s) default-scheduler 0/3 nodes are
available: 3 Insufficient cpu.
```

Tools for capacity planning - Limits

- **Limits**

- Used by the **Kubelet** to enforce thresholds on resources used
- The **Pod** might be **TERMINATED** if memory usage goes over the defined limit
- The Container CPU use is being throttled when CPU usage goes over the defined limit

```
$ kubectl describe po | grep State -A4
  State:          Waiting
    Reason:        CrashLoopBackOff
  Last State:     Terminated
    Reason:        OOMKilled
    Exit Code:     1
  Started:        Tue, 11 Dec 2018 15:28:34 +0100
  Finished:       Tue, 11 Dec 2018 15:28:34 +0100
```

Tools for capacity planning - Requests/Limits units

For **Memory**, limits and requests are expressed in bytes as a plain integer or as a fixed-point integer using one of these suffixes: **E, P, T, G, M, K**. Or the power-of-two equivalents: **Ei, Pi, Ti, Gi, Mi, Ki**.

For example, the following represent roughly the same value:

```
128974848, 129e6, 129M, 123Mi
```

For **CPU**, limits and requests are expressed in CPU units. One CPU unit is equivalent **to 1 core**.

It can be expressed as a millicpu **100m** or with a decimal **0.1**.

Tools for capacity planning - Requests/Limits example

Example Pod defining CPU/Memory requests and limits:

```
apiVersion: v1
kind: Pod
metadata:
  name: too-much-cpu
spec:
  containers:
  - name: too-much-cpu
    image: centos
    resources:
      requests:
        cpu: "500m"
        memory: "128m"
      limits:
        cpu: "1.5"
        memory: "1Gi"
```

Tools for capacity planning - QoS definition

When available resources on a **Node** become scarce, **Kubernetes** might evict workloads to ensure stability of the **Node**.

To make decisions about which **Pods** to evict, **Kubernetes** assigns QoS classes to them at their creation.

There are 3 QoS classes:

- **Guaranteed**: Every **Container** in the Pod has equal value for limits and requests (CPU and Memory)
- **Burstable**: Doesn't meet Guaranteed criteria but at least one **Container** has a CPU or Memory request
- **BestEffort**: Containers in the Pod do not have any memory or CPU limits or requests

Tools for capacity planning - QoS result

Then **kubelet** ranks and evicts **Pods** in the following order:

- **BestEffort** or **Burstable** Pods whose usage of a starved resource exceeds its request
- **Guaranteed** pods and **Burstable** pods whose usage is beneath requests are evicted last

This mechanism only works with already scheduled workloads but doesn't enforce preemption for pending pods.



TP 3.4 : Capacity planning

Garbage Collection (1/2)

Node garbage collection is done by the **kubelet** and occurs on 5 condition types:

- `memory.available`
 - `nodefs.available`
 - `nodefs.inodesFree`
 - `imagefs.available`
 - `imagefs.inodesFree`
-
- `nodefs` is used by the **kubelet** for volumes, daemon logs, etc
 - `imagefs` is used by the **container runtime** for storing images and container writable layers

Garbage Collection (2/2)

There are two kinds of eviction types:

- **Hard**: no grace period, immediate action to reclaim the starved resource is taken
- **Soft**: no action is taken until a grace period has been exceeded
 - Grace period is configured with flag `eviction-soft-grace-period`
 - When soft evicting Pods the `eviction-max-pod-grace-period` flag describes the grace period to use when terminating pods

Garbage Collection - Configuration

Eviction thresholds are configurable with a flag on **kubelet** startup command:

```
--eviction-hard=memory.available<500Mi,  
nodefs.available<1Gi,imagefs.available<100Gi
```

or with the **kubelet** config file:

```
kind: KubeletConfiguration  
apiVersion: kubelet.config.k8s.io/v1beta1  
evictionHard:  
  memory.available: "500Mi"  
  nodefs.available: "1Gi"  
  imagefs.available: "100Gi"
```

Garbage Collection - Default values

The **kubelet** has the following default **hard eviction** thresholds:

- memory.available<100Mi
- nodefs.available<10%
- nodefs.inodesFree<5%
- imagefs.available<15%

Garbage Collection - Node condition

When an eviction threshold has been met (soft or hard), the **kubelet** reports the corresponding **Node** status

Node Condition	Eviction Signal
MemoryPressure	<code>memory.available</code>
DiskPressure	<code>nodefs.available</code> , <code>nodefs.inodesFree</code> , <code>imagefs.available</code> , or <code>imagefs.inodesFree</code>

Garbage Collection - Reclaiming resources

The **kubelet** then tries to reclaim resources with the following process

1. The **kubelet** removes dead Pods and theirs containers, and deletes all unused images
2. If insufficient resources were reclaimed, the **kubelet** will now evict end-user **Pods**
 - Pods are sorted according to their QoS and on their total disk usage: local volumes + logs & writable layer of all their containers.

Note: Having both **nodefs** and **imagefs** on the **Node**, only the actions concerning the right filesystem are taken

e.g.: for **imagefs** eviction threshold, it deletes all unused images



TP 3.5 : Garbage collection

Connectivity with the rest of your infrastructure

To expose services outside the cluster, some interactions with external infrastructure is needed. But **Kubernetes** provides some tools to help you:

- **LoadBalancers**
 - Cloud
 - On premises
- **Ingress**

LoadBalancers - Cloud

- Many cloud providers propose an integration of **Kubernetes** with their internal LoadBalancing features (AWS, Azure, GCE)
- Set `.spec.type` to **LoadBalancer**, then **Kubernetes** will interact with the Cloud Provider api and provision automatically a **LoadBalancer**
- Note :
 - This provisioning is asynchrone
 - The external IP will appear as "Pending" while the **LoadBalancer** is being provisioned
 - Technically, the **LoadBalancer** will redirect the traffic to the service using a **NodePort** (you don't have to configure the **NodePort**, it is done implicitly)

LoadBalancers - On premises

There are few projects which aim at providing LoadBalancer service type (with the appropriate controller) for **On Premises** Kubernetes deployments:

- Metallb
 - Young project, you should treat it as **a beta system**
 - Project has known **incompatibilities** with IPVS mode in kube-proxy and some network addons
- F5
 - Relies on **specific networking** to route traffic from BIG IP to the cluster

LoadBalancers - Limitations

- Exposing services through **NodePort** is not really elegant
- You can manually configure external loadbalancers not managed by Kubernetes (HAProxy, F5, ...) ... only if you have access to such loadbalancers
- Kubernetes services work at the **TCP** level, hence cannot provide features relying on cookies and other **HTTP** features
- **Ingress** can! ☺

Ingress

- What is an Ingress?
- Which Ingress Controller?
- How to deploy/manage them?

Ingress description

Ingress is used to manage traffic coming

- from outside the cluster
- to service inside the cluster

It provides SSL Termination, Loadbalancing, name-based virtual hosting

- **Ingress** is the resource describing how the traffic should be handled
- The **Ingress Controller** watches Ingresses and updates the Reverse Proxy configuration

Example Ingress

The following **Ingress** handles all requests

- with path `/my-app`
- header `Host: my-app.mydomain.com`
- and sends them to the service named `my-app`

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: my-app
spec:
  rules:
  - host: my-app.mydomain.com
    http:
      paths:
      - path: /my-app
        backend:
          serviceName: my-app
          servicePort: 80
```

Which Ingress Controller (1/2)

There are many Ingress controllers available for **Kubernetes**:

- Traefik from Containous
- Contour from Heptio
- Istio from Google and IBM, in partnership with the Envoy team at Lyft
- Ambassador from Datawire
- Nginx from NGINX Inc
- Kong from KongHQ
- HAProxy from HAProxy Technologies
- F5 BIG-IP Controller from F5 Networks

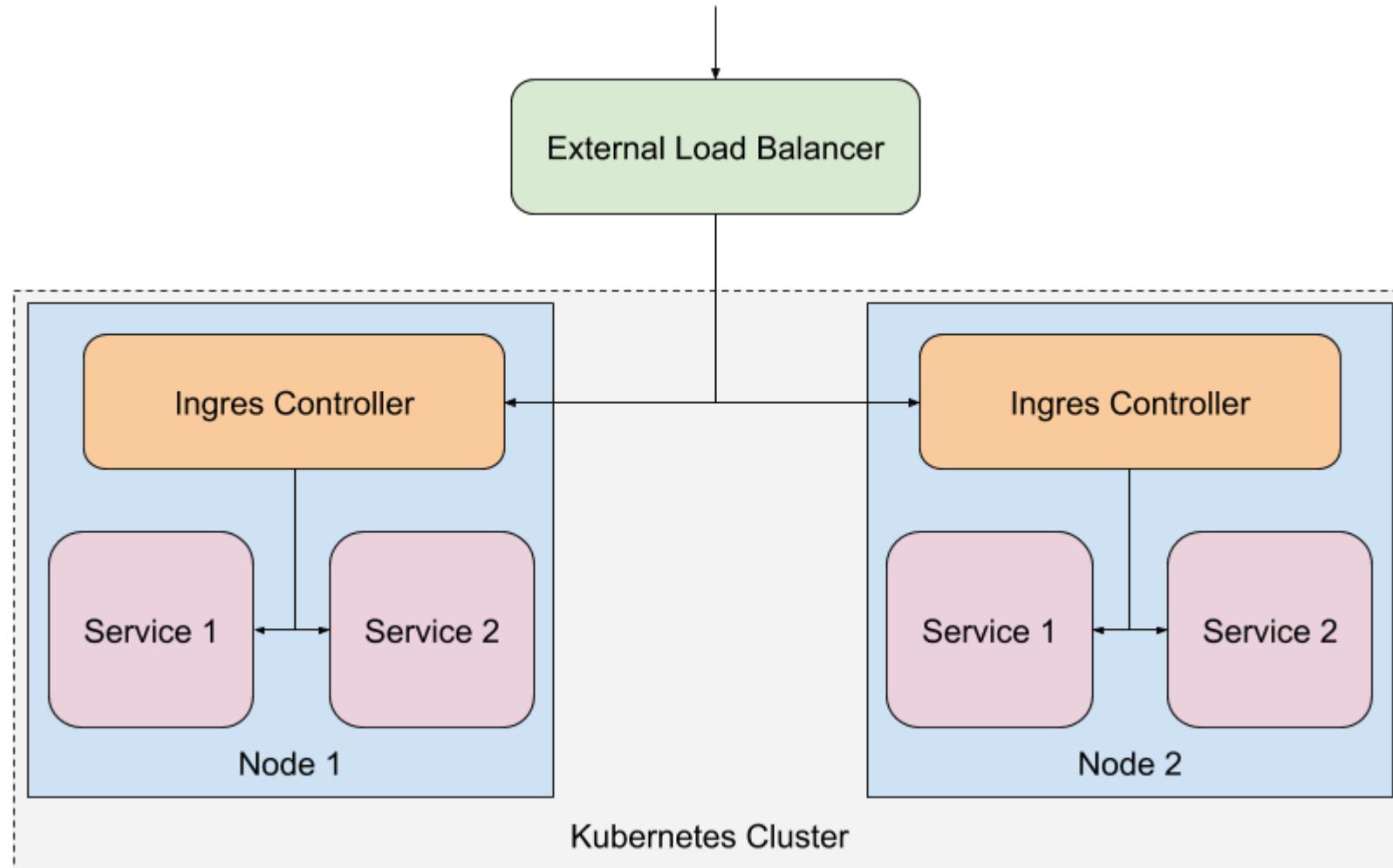
Which Ingress Controller (2/2)

Name	Company	Opensource	Commercial support	Description
Traefik	Containous	Yes	Yes	Let's Encrypt, secrets, http2, websocket
Contour	Heptio	Yes	-	Custom Ingress resource, many loadbalancing algorithms
Istio	Istio authors	Yes	Yes	More than a simple Ingress
Ambassador	Datawire	Yes	Yes	gRPC, http2, istio integration
Nginx	NGINX Inc	Yes	Yes	-
HAProxy	HAProxy Technologies	Yes	Yes	Many loadbalancing algorithms
F5 BIG-IP Controller	F5 Networks	No	Yes	-

How to choose

- Do you need specific protocols (http2, gRPC, tcp, ...)?
- Do you need find fine grained traffic control (For A/B Testing, ...)?
- Is it already proof tested?
- Is it compatible with monitoring (Prometheus metrics export, ...)?
- Do you want to stick to vanilla Ingress resources?

Integration with existing infrastructure (1/3)



Integration with existing infrastructure (2/3)

Reference implementation architectures usually propose:

- A front **LoadBalancer** outside of the cluster
- Distributing traffic to an **Ingress Controller**
 - Exposed as Deployment + NodePort Service or DaemonSet + HostPort container
- The Ingress Controller is able to distribute traffic accross cluster services based on L7 informations (Host, Cookie, ...)
- The front loadbalancer can be used
 - to distribute traffic among nodes
 - in front of several **Kubernetes** instances when "cluster upgrade" means "create new and destroy old"

Integration with existing infrastructure (3/3)

To manage efficiently your SSL certificates you can:

- Use a wildcard certificate and configure SSL offloading
 - on your front LoadBalancer (outside of the cluster)
 - on your Ingress controller (inside of the cluster)
- Use Let's encrypt with your Ingress controller (if compatible)
- Do some tricky implementation like with Vault's PKI secrets engine & dynamic X.509 certificates ❤



TP 3.6 : Ingress

Scheduling

There are few ways to control how scheduler will assign **Pods** to **Node**:

- **NodeSelector**
 - **Pod** will be assigned to **Nodes** with a label matching selector
 - will be deprecated in favor of **NodeAffinity**
- **NodeAffinity**: can express requirement and/or preference
- **PodAffinity**: to colocate **Pods**
- **PodAntiAffinity**: to make sure two **Pods** won't be on the same **Node**

Node affinity

There are two kinds of affinities:

- preferredDuringSchedulingIgnoredDuringExecution
- requiredDuringSchedulingIgnoredDuringExecution

A third one requiredDuringSchedulingRequiredDuringExecution is planned, it will just add eviction if the **Node** doesn't match the selector anymore.

Node affinity operators

Node selector example for affinity

```
nodeSelectorTerms:  
  - matchExpressions:  
    - key: kubernetes.io/env  
      operator: In  
      values:  
        - qa
```

Available operators are: **In**, **NotIn**, **Exists**, **DoesNotExist**, **Gt**, **Lt**

Node affinity example

Pod with Node affinity example

```
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: kubernetes.io/e2e-az-name
                operator: In
                values:
                  - e2e-az1
  containers:
    - name: with-node-affinity
      image: k8s.gcr.io/pause:2.0
```

Pod affinity/anti-affinity

Pod affinity/anti-affinity mechanism is used to make sure **Pods**:

- Run on different **Node** or set of **Nodes**
- Run on the same **Node** or set of **Nodes**

It can be used to :

- Spread Pods of a **ReplicaSet** or **StatefulSet** on different **Nodes** and improve resilience
- Colocate some **Pods** on the same **Node** or availability zone to reduce latency

More examples [here](#)

Pod affinity example

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
              - key: security
                operator: In
                values:
                  - S1
      topologyKey: failure-domain.beta.kubernetes.io/zone
  containers:
    - name: with-pod-affinity
      image: k8s.gcr.io/pause:2.0
```

Pod anti-affinity example

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-anti-affinity
spec:
  affinity:
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 100
          podAffinityTerm:
            labelSelector:
              matchExpressions:
                - key: security
                  operator: In
                  values:
                    - S2
            topologyKey: kubernetes.io/hostname
  containers:
    - name: with-pod-anti-affinity
      image: k8s.gcr.io/pause:2.0
```





TP 3.8 : Affinity

Taints and Tolerations

Taints are used to mark some **Nodes** as not useable by **Pods** without the corresponding **Tolerance**.

It can be used to:

- Dedicate some nodes to some users or usage
- Reserve some nodes with special hardware

Taints example

```
$ kubectl taint nodes node1 key=value:NoSchedule
```

The taint has key **key**, value **value**, and taint effect **NoSchedule**.

There are 3 possible effects for **Taints**:

- **NoExecute**: Pods without the Toleration will be evicted and won't be scheduled on this Node
- **NoSchedule**: Pods without the Toleration won't be scheduled on this Node
- **PreferNoSchedule**: The scheduler will try not to schedule Pods without the Toleration on this Node

Toleration example

The corresponding Tolerations can then be added to Pods:

```
tolerations:  
- key: "key"  
  operator: "Equal"  
  value: "value"  
  effect: "NoExecute"  
  tolerationSeconds: 3600
```

The additional **tolerationSeconds** is used to delay the eviction of the Pod



TP 3.9 : Taints & Tolerations

Custom schedulers

We've already seen that it's possible to use custom rules for scheduling with a custom **Policy** definition passed to the **kube-scheduler** by specifying `--policy-config-file`.

If affinity/anti-affinity and other mechanism are not sufficient, it's also possible to have more than one scheduler on the same cluster.

Pods can then be configured to use the **default-scheduler** or your custom scheduler.

Custom scheduler example

Custom scheduler deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    component: scheduler
  name: my-scheduler
spec:
  selector:
    [...]
  spec:
    serviceAccountName: my-scheduler
    containers:
      - name: kube-second-scheduler
        command:
          - /usr/local/bin/kube-scheduler
          - --address=0.0.0.0
          - --leader-elect=false
          - --scheduler-name=my-scheduler
          - --policy-config-file=policy.yml
        image: gcr.io/my-gcp-project/my-kube-scheduler:1.0
```

Custom scheduler usage

Pod using the custom scheduler

```
apiVersion: v1
kind: Pod
metadata:
  name: annotation-second-scheduler
  labels:
    name: multischeduler-example
spec:
  schedulerName: my-scheduler
  containers:
  - name: pod-with-second-annotation-container
    image: k8s.gcr.io/pause:2.0
```



TP 3.10 : Schedulers

Troubleshooting a cluster

Common steps:

- Know if components are launched as systemd services or static pods
 - Check systemd services status: `systemctl status <service>`
 - Check static pods status: `kubectl get pods -n kube-system`
 - If api-server is unavailable check container status: `docker container ls -a`
- Check service logs either
 - On the nodes with: `journalctl -u <service> -f`
 - On your log centralization solution

Troubleshooting API server

- Symptom: API server unavailable
- How to tell:

```
$ kubectl get pods # For example...
The connection to the server
<api-server-address>:<api-server-port> was refused - did you specify the right
host or port?
```

- Probable cause:
 - kube-apiserver down
 - etcd cluster unhealthy
- Remediation
 - Check api-server logs and fix it
 - Heal the etcd cluster

Troubleshooting scheduler

- Symptom: Pods stay in PENDING status
- How to tell:

```
$ kubectl get pods
NAME                  READY   STATUS    RESTARTS   AGE
whoami-56d4574f6c-2125s   0/1     Pending   0          4s
```

- Probable cause:
 - kube-scheduler unavailable
 - not enough resources to satisfy pod creation
- Remediation
 - Check kube-scheduler logs and fix it
 - Check resources requests

Troubleshooting controller-manager

- Symptom: Pods, ReplicaSets, Endpoints are not created
- How to tell:

```
$ kubectl get deploy
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
whoami    1          0          0           0           10s
```

- Probable cause:
 - kube-controller-manager unavailable
- Remediation
 - Check kube-controller-manager logs and fix it

Troubleshooting node

- Symptom: Node not ready
- How to tell:

```
$ kubectl get nodes
NAME        STATUS   ROLES      AGE        VERSION
node1       Ready    master     31m       v1.11.3
node2       NotReady <none>    27m       v1.11.3
```

- Probable cause:
 - kubelet down on node
 - kubelet cannot reach api-server
- Remediation
 - check kubelet service logs
 - fix kubelet service
 - fix kubelet <-> kube-apiserver connectivity

Troubleshooting DNS

- Symptom: DNS not available
- How to tell:

```
# Inside a pod, try to resolve a service by name
$ nslookup catalogue-db
;; connection timed out; no servers could be reached
```

- Probable cause:
 - kube-dns or core-dns pods not available
- Remediation
 - check that a cluster dns is deployed (either coredns or kube-dns)
 - fix cluster dns deployment





TP 3.11 : Troubleshooting

Cluster Upgrade

Table of contents

- Architecture
- Installation
- Configuration and operational maintenance
- *Cluster Upgrade*
- Day to day actions
- Extensibility: Operators, CRD and API Servers
- Federation, Service Mesh, Security

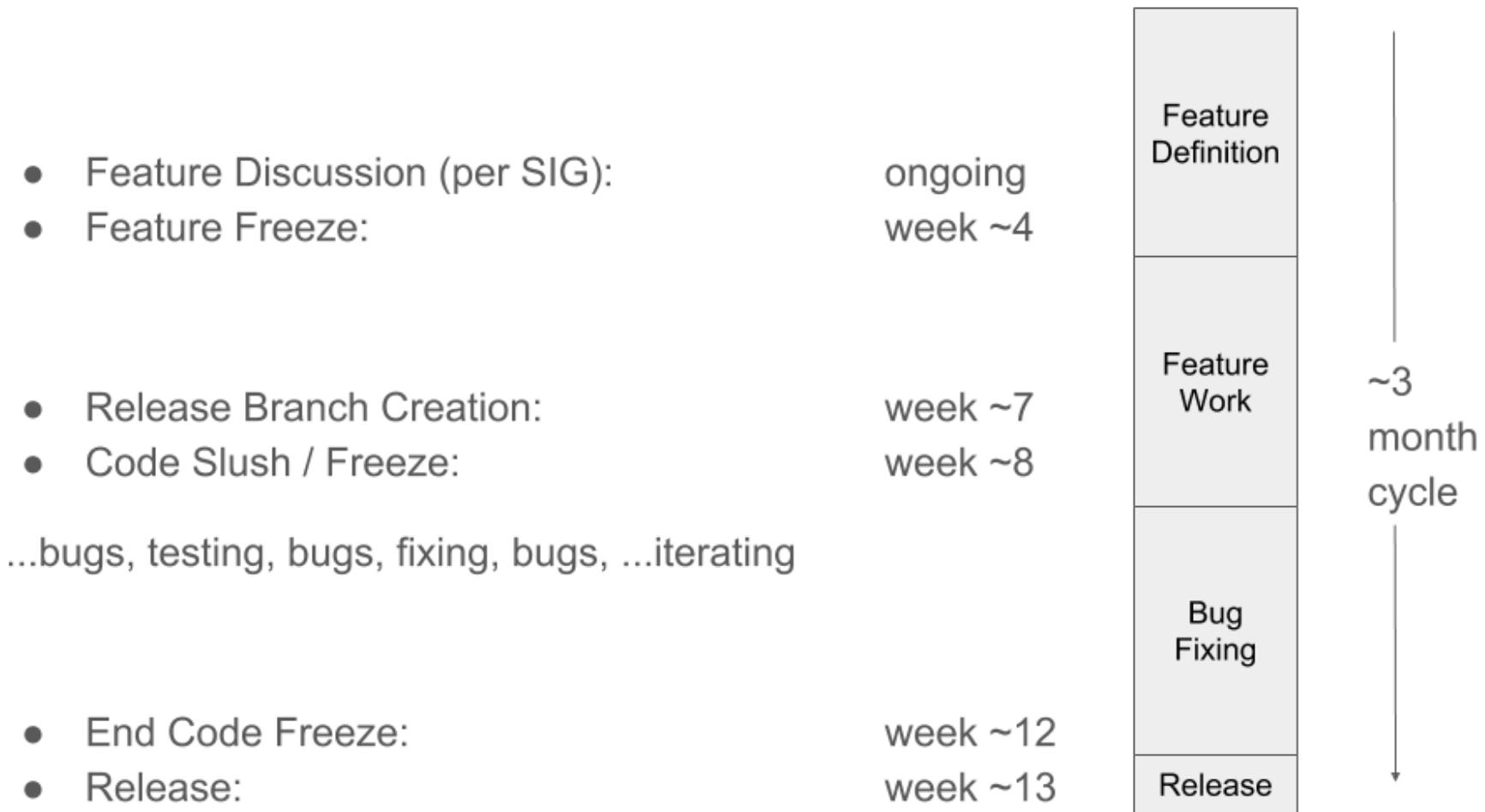
Chapter content

- Kubernetes Release Cycle
- Upgrade process
- Upgrade with kubeadm
- Upgrade with kops
- Upgrade with kubespray
- Upgrade with gke
- Upgrade with eks

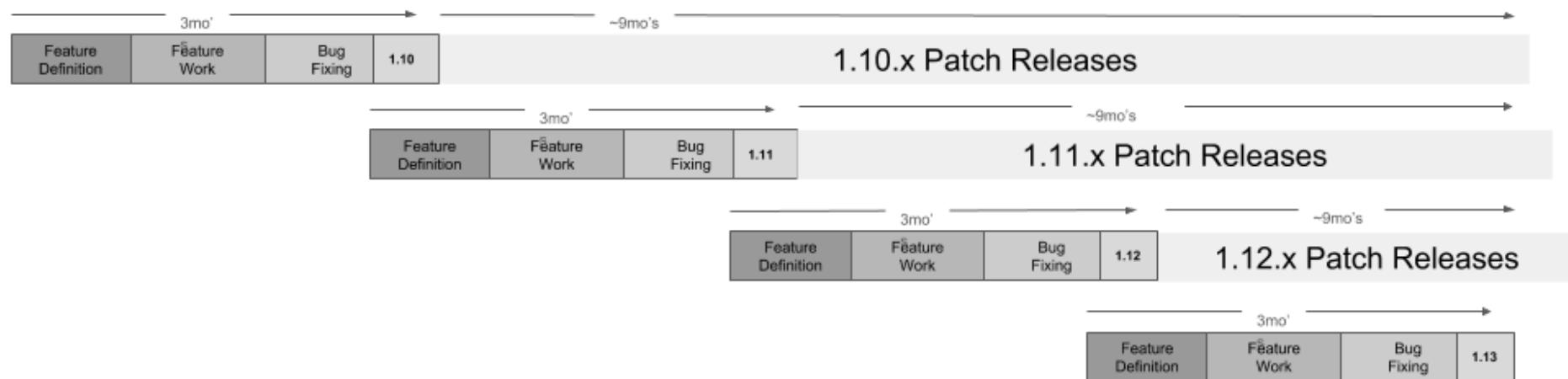
Kubernetes Release process

- No LTS (Long Term Support) version
 - Only 3 minor version supported at a time
 - Plan to upgrade **AT LEAST** every nine months
- Driven by **SIG (Special Interest Group)** Release team

Kubernetes Release Cycle



Kubernetes Release Lifecycle illustration



Kubernetes versions support

Kubernetes version	Release month	End-of-life-month
v1.6.x	March 2017	December 2017
v1.7.x	June 2017	March 2018
v1.8.x	September 2017	June 2018
v1.9.x	December 2017	September 2018
v1.10.x	March 2018	December 2018
v1.11.x	June 2018	March 2019
v1.12.x	September 2018	June 2019
v1.13.x	December 2018	September 2019

Upgrade patterns

- Recreate and destroy the whole cluster
 - Easy for stateless workloads and with a front LB
 - Requires more infrastructure
 - Can be used to validate current cluster workloads on new version before upgrade
- Upgrade existant cluster
 - Highly dependent on the tool used for creating the cluster
 - Well documented

Standard upgrade process steps (1/2)

- Keep these in mind:
 - You have to keep your cluster up to date
 - You have to read the release notes

Standard upgrade process steps (2/2)

- Upgrading is done with these steps:
 1. Backup etcd
 2. Migrate your etcd backup
 3. Upgrade the control plane
 - Workloads will still be online
 - Control plane will be unavailable (no cluster state modification is possible)
 4. Upgrade worker nodes
 - Rolling-update or with double node number
 - It might be a good time to patch/update your nodes if you don't do it on a more regular basis

Upgrade with kubeadm

- Upgrade with **kubeadm** is well documented
 - Even for high availability setups since 1.11

Upgrade steps:

- Upgrade the control plane
- Upgrade master and node packages (infrastructure pods)
- Upgrade kubelet on each nodes

Upgrade with kubeadm - Control plane (1/2)

- Use `kubeadm upgrade plan <version>` to preview the upgrade steps

```
[preflight] Running pre-flight checks.
```

```
[...]
```

Components that must be upgraded manually after you have upgraded the control plane with '`kubeadm upgrade apply`':

COMPONENT	CURRENT	AVAILABLE
Kubelet	2 x v1.11.1 1 x v1.11.3	v1.12.0
		v1.12.0

Upgrade to the latest experimental version:

COMPONENT	CURRENT	AVAILABLE
API Server	v1.11.3	v1.12.0
Controller Manager	v1.11.3	v1.12.0
Scheduler	v1.11.3	v1.12.0
Kube Proxy	v1.11.3	v1.12.0
CoreDNS	1.1.3	1.2.2
Etcd	3.2.18	3.2.24

You can now apply the upgrade by executing the following command:

```
kubeadm upgrade apply v1.12.0
```

Upgrade with kubeadm - Control plane (2/2)

When the upgrade has been done with **kubeadm** you should also upgrade your network CNI plugin.

These steps are not covered here as they are dependent on the chosen network CNI

Upgrade with kubeadm - Node kubelet

On each node:

- Evict workloads and mark it as unschedulable:
 - `kubectl drain $NODE`
 - You might need to add `--ignore-daemonsets`
 - `kubectl drain $NODE --ignore-daemonsets`
- Upgrade `kubelet` and `kubeadm`
- Upgrade `kubelet` config
 - `kubeadm upgrade node config --kubelet-version <version>`
- Restart the `kubelet`: `systemctl restart kubelet`
- Mark node schedulable: `kubectl uncordon $NODE`





TP 4.1 : Kubeadm upgrade

Upgrade with kops

Manual upgrade steps:

- Update the version in the cluster declaration
 - `kops edit cluster` or `kops upgrade cluster $NAME`
- Apply the changes on the infrastructure declaration
 - `kops update cluster --yes`
- Check what needs reboot
 - `kops rolling update cluster`
- Restart the nodes
 - `kops rolling update cluster --yes`

Upgrade with kubespray

- There is a dedicated playbook for safe upgrades. Launch it with:
 - `ansible-playbook upgrade-cluster.yml -b -i <inventory> -e <kubernetes_version>`
- The upgrade does the following steps:
 - docker
 - etcd
 - kubelet and kube-proxy
 - network_plugin
 - kube-apiserver, kube-scheduler and kube-controller-manager
 - Add-ons (e.g.: kube-dns)

Upgrade with GKE

Upgrade with gke are taken care off by GCP as two steps

- Upgrade control plane
 - During this phase, the control plane will be unavailable
 - The upgrade take ~5-10 minutes
- Rolling update of the nodes
 - Change the version defined on the node pool
 - GCP will then drain, shutdown and restart the node with the new version

Upgrade with EKS - Control Plane

- Upgrade from the console or with the cli
 - `aws eks update-cluster-version --name <name> --kubernetes-version <version>`
- `kube-proxy` update must be done manually by patching the **DaemonSet** to the required version
- You might also need to switch from `kube-dns` to CoreDNS
 - In case you're upgrading from 1.10 to 1.11

Upgrade with EKS - Worker Nodes

Here is an overview of the two methods available to upgrade nodes on EKS

- Migrating to a new worker node group
- Updating an existing worker node group

Upgrade with EKS - New worker node group

1. Launch a new worker node group
2. Record the `NodeInstanceRole` to be add theses nodes to your cluster
3. Update the security groups for both worker node groups to allow communication
4. Map the new workers on cluster RBAC by editing the **aws-auth ConfigMap**
5. Deactivate cluster autoscaling (if present)
6. Taint old nodes
7. Check that critical pods are scheduled to avoid unavailability (dns, fluentd, ...)
8. Drain old nodes
9. Remove old workers from cluster auth by editing the **aws-auth ConfigMap**
10. Reactivate cluster autoscaling (if present)

Upgrade with EKS - Update existing nodes

1. Check that critical pods are scheduled to avoid unavailability (dns, fluentd, ...)
2. Deactivate cluster autoscaling (if present)
3. Update your worker node group stack with the AWS CloudFormation console
4. Specify the details of your instance group:
 - Desired capacity
 - Max size
 - Instance type
 - Image id
5. Wait for the update to complete
6. Reactivate cluster autoscaling (if present)



Day to day actions

Table of contents

- Architecture
- Installation
- Configuration and operational maintenance
- Cluster Upgrade
- *Day to day actions*
- Extensibility: Operators, CRD and API Servers
- Federation, Service Mesh, Security

Chapter content

- Namespaces and Instances
- NetworkPolicies
- Common Admission Controllers
- LimitRanges
- Quotas
- Security contexts
- PodSecurityPolicy
- Authentication/Authorization/RBAC
- Persistent Volumes

Namespaces and cluster instances (1/2)

- How do I split my environments/apps on kubernetes?
 - Clusters are a mean to mutualise resources (cpu, memory, storage, ...)
 - Namespaces provide isolation within a cluster
- But
 - Namespace isolation isn't that strong with current container runtimes
 - One single mistake could jeopardize the entire cluster

Namespaces and cluster instances (2/2)

- It depends on the maturity, autonomy and responsibility of the teams
- Some patterns:
 - Clusters reflecting the operator teams separation
 - Separate clusters depending on how users interact with them (manually, with tools)
- Things to consider
 - You need to be able to test changes applied to infrastructure components (service mesh, ingress controller, ...)
 - Creating a new cluster (or replicating an existing one) should be easy
 - Use RBAC/Quotas/... to keep everybody satisfied

NetworkPolicies

- By default **Pods** accept traffic from any source
- It may be useful to restrict access to/from some **Pods**
- To do so you'll use a **NetworkPolicy**
 - Pods become isolated when they are matched by a **NetworkPolicy**
 - You must have a **network plugin** which supports **NetworkPolicy**
 - Otherwise you'll be able to create the object, but it will have no effect
- **egress**: traffic originating from the selected **Pods**
- **ingress**: traffic sent to the selected **Pods**

NetworkPolicy example

network-policy.yml:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: api-allow
spec:
  podSelector:
    matchLabels:
      app: review
      role: api
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: review
```

This policy only allow access to **Pods** with labels `app=review` and `role=api` from **Pods** with label `app=review`

There are a lot of example of **NetworkPolicies** on this github repository:
<https://github.com/ahmetb/kubernetes-network-policy-recipes>

Isolating namespaces from each other

deny-from-other-namespaces.yaml

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  namespace: my-project
  name: deny-from-other-namespaces
spec:
  podSelector:
    matchLabels:
  ingress:
    - from:
      - podSelector: {}
```

- This policy is deployed on **Namespace my-project**
- **podSelector.matchLabels** is empty, so it applies on all **Pods** in the **Namespace**
- **ingress.from.podSelector** is empty, so it allows traffic from all **Pods** in the **Namespace**

NetworkPolicies - Compatible/incompatible plugins

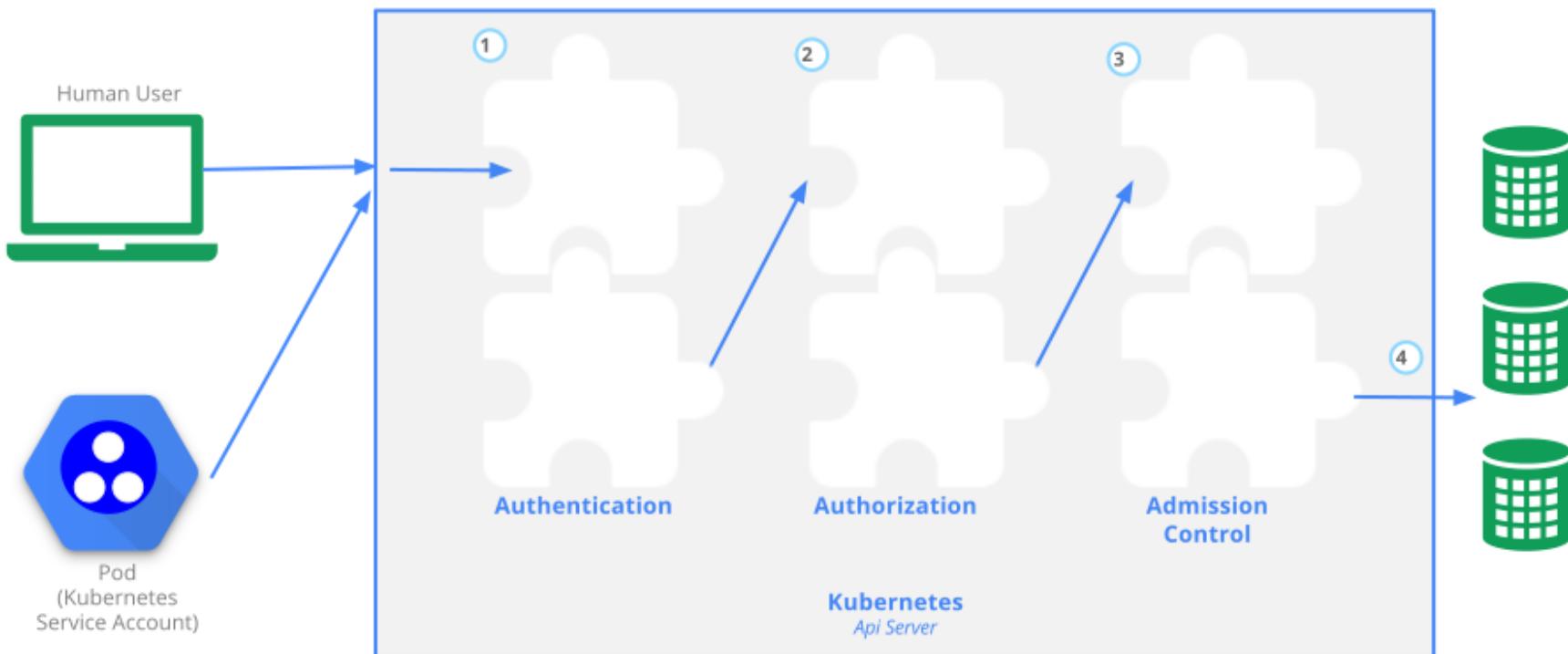
- Incompatible plugins:
 - Flannel
 - Canal with Flannel
- Compatible plugins:
 - Weave
 - Canal with Calico
 - Cilium



TP 5.1: Network policies

Common admission controllers

When sending requests to the cluster to create or modify resources, the resource isn't immediately created in the store. The following diagram shows the different steps followed by the request.



Common admission controllers - All of them

- **AlwaysPullImages**: sets imagePullPolicy to Always on every **Pods**
- **DefaultStorageClass**: sets **DefaultStorageClass** on **PVC**
- **DefaultTolerationSeconds**: sets the default forgiveness toleration for **Pods**
- **DenyEscalatingExec**: denies exec/attach access to **Pods** having privileged access to host
- **ExtendedResourceToleration**: adds tolerations to **Pods** requesting extended resources
- **ImagePolicyWebhook**: delegate admission decision based on image
- **LimitPodHardAntiAffinityTopology**: limits **AntiAffinity** to kubernetes.io/hostname
- **LimitRanger**: applies **LimitRange** controls

Common admission controllers - All of them

- **NamespaceAutoProvision**: creates **Namespace** on demand
- **NamespaceExists**: checks **Namespace** existence
- **NamespaceLifecycle**: prevents deletion of reserved **Namespace** and denies resources creation on a terminating **Namespace**
- **NodeRestriction**: limits what a **kubelet** can modify (owned **Pods** and **Node**)
- **OwnerReferencesPermissionEnforcement**: make **delete** permission necessary to modify `metadata.ownerReferences`
- **PodNodeSelector**: limits what node selectors may be used within a namespace
- **PersistentVolumeClaimResize**: adds check on which **PVC** can be resized
- **PodPreset**: injects the **PodPreset** fields in new **Pods**

Common admission controllers - All of them

- **PodSecurityPolicy**: applies the policies on new **Pods**
- **PodTolerationRestriction**: checks coherence between **Pod** and **Namespace** toleration
- **Priority**: applies the priority value to **Pods**, also rejects **Pods** without **PriorityClass**
- **ResourceQuota**: enforces **ResourceQuota**
- **SecurityContextDeny**: denies creation of **Pods** with some securityContext values, maybe used if **PSP** is not an option
- **ServiceAccount**: automates **ServiceAccount** management
- **StorageObjectInUseProtection**: places finalizers on new created **PV**

Common admission controllers - Deprecated

- **AlwaysAdmit / AlwaysDeny**
- **DenyExecOnPrivileged**: merged in **DenyEscalatingExec**
- **PersistentVolumeLabel**: attaches zone/region labels to **PV**, replaced by **cloud controller manager**

Common admission controllers - Alpha/Beta

- **EventRateLimit** (alpha): limits requests per scope (**Namespace**, **User**, ...)
- **Initializers** (alpha): sets the pending initializers by modifying the metadata of the resource to be created
- **MutatingAdmissionWebhook** (beta in 1.9): calls any mutating webhooks which match the request
- **ValidatingAdmissionWebhook** (alpha in 1.8; beta in 1.9): calls any validating webhooks which match the request

Common admission controllers - Configuration

For each version of Kubernetes there are default sets of activated/deactivated plugins. Configuration of activated/deactivated controllers is done via command-line parameters to the api server:

- Activated: `kube-apiserver --enable-admission-plugins=...`
- Deactivated: `kube-apiserver --disable-admission-plugins=...`

You can see which admission plugins are activated with:

- `kube-apiserver -h | grep enable-admission-plugins`

Common admission controllers - Default

In Kubernetes 1.13 here are the default activated ones:

- NamespaceLifecycle
- LimitRanger
- ServiceAccount
- PersistentVolumeClaimResize
- DefaultStorageClass
- DefaultTolerationSeconds
- MutatingAdmissionWebhook
- ValidatingAdmissionWebhook
- ResourceQuota
- Priority



TP 5.2: Admission controllers

LimitRanges

- **Kubernetes** best practices include defining **Request/Limits** for workloads
 - It helps the scheduler
 - It's a good way to make resource usage explicit
 - Using Quotas imposes every container to define a memory limit
- **LimitRanges** are used to ensure default, min, max request/limit values for cpu and memory usage
- How?
 - Create an object LimitRange in the concerned Namespace

LimitRanges example

limit-range.yml:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: limit-range
  namespace: limit
spec:
  limits:
  - default:
    cpu: 1
    memory: 512Mi
  defaultRequest:
    cpu: 0.5
    memory: 256Mi
  type: Container
```

LimitRanges other properties

- Besides default values, LimitRanges can be set for the following:
 - `min/max`: applied to CPU or memory
 - `maxLimitRequestRatio`: the max authorized ratio between limit and request
 - `type`: the limit can be applied to individual `Container` or the whole Pod

Keep in mind that:

$$\min \leq \text{request} \leq \text{limit} \leq \max$$

LimitRanges behaviour

- If you set only `min` and `max`, `default` and `defaultRequest` values will be automatically set with the `max` value
- If a container defines a limit but no request, the container request is set to the container limit as default
- If a container defines a request but not a limit, the container limit is set to the default limit

Quotas

When sharing a cluster between projects/teams, you should define some resource **Quotas**. These **Quotas** apply on:

- Compute resources
- Extended resources
- Storage resources
- Resource count

Quotas - How

- **Quotas** are enforced by an admission plugin on resource creation.
- To use quotas with your cluster, you must:
 - Enable the **ResourceQuota** on API Server startup: `--enable-admission-plugins=ResourceQuota,...`
 - Create a **ResourceQuota** in the targeted **Namespace**
- Then the API server will refuse to create the resource if it would result in exceeding the defined **Quotas**



Quota example

quota-example.yml:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: quota-mem-cpu
  namespace: quotas
spec:
  hard:
    requests.cpu: "2"
    requests.memory: 2Gi
    limits.cpu: "4"
    limits.memory: 4Gi
```

Compute resource quotas

Compute resources quota can be set on:

- `limits.cpu`
- `limits.memory`
- `requests.cpu`
- `requests.memory`

Keep in mind the default values assigned when only a subset of `limits`, `requests` are defined on **Pods** or **LimitRange**

Extended resource quotas

As of **Kubernetes** 1.10, it's possible to define some **Extended resources** on **Nodes**.

*For example, you can define
`requests.nvidia.com/gpu: 4`*

The definition and attribution of these resources isn't really straightforward at this time, you need to interact directly with the API.

However it's possible to set **Quotas** on these extended resources.

Storage resource quotas

Quotas can be set on storage resources:

- `requests.storage`: sum of storage used by the PVC
- `persistentvolumeclaims`: number of PVC
- `<storage-class-name>.storageclass.storage.k8s.io/requests.storage`
- `<storage-class-name>.storageclass.storage.k8s.io/persistentvolumeclaims`
- `requests.ephemeral-storage` (1.8 as alpha)
- `limits.ephemeral-storage` (1.8 as alpha)

Object count quotas

Quotas can be set on object count (1.9), with this syntax:
count/<resource>. <group>

The following resources are supported:

- configmaps / secrets
- persistentvolumeclaims
- pods: The total number of pods in a non-terminal state that can exist in the namespace
- replicationcontrollers
- resourcequotas
- services / services.loadbalancers / services.nodeports



Quota scopes

Quotas set a scope to restrict their field of application:

- **Terminating / NotTerminating**
- **BestEffort / NotBestEffort**

Quotas and PriorityClass

And as of **Kubernetes** 1.12, **Quotas** can be applied to **PriorityClass**

Example of **ResourceQuota** applied to a **PriorityClass**:

```
...
scopeSelector:
  matchExpressions:
  - operator : In
    scopeName: PriorityClass
    values: ["high"]
...
...
```



TP 5.3 & 5.4: LimitRanges & Quotas

SecurityContext

A security context is used to define some security properties of the process running inside containers.

- **Pod** and **Container** properties
 - **runAsUser**: the first process of the **Pod** containers will run as **uid**
 - **runAsGroup**: default **gid** of files created in volumes
 - **runAsNonRoot**: if set to true, the **Kubelet** won't allow **root** to run the container
 - **seLinuxOptions**: SELinux context to be applied to all containers
 - **supplementalGroups**: other **gid** associated to the user running containers

SecurityContext properties (2/2)

- **Pod** specifics:
 - **fsGroup**: default **gid** of the process and change the group owner of the files for some volume types
- **Container** specifics:
 - **allowPrivilegeEscalation**
 - **capabilities**: capabilities to add/drop
 - **privileged**: if set to true, the container will run as privileged
 - **readOnlyRootFilesystem**

SecurityContext example

pod-with-security-context.yml:

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    fsGroup: 2000
  volumes:
  - name: sec-ctx-vol
    emptyDir: {}
  containers:
  - name: sec-ctx-demo
    image: gcr.io/node-hello:1.0
    volumeMounts:
    - name: sec-ctx-vol
      mountPath: /data/demo
  securityContext:
    allowPrivilegeEscalation: false
```



PodSecurityPolicy

PodSecurityPolicy is used to enforce some **SecurityContext** properties and other security constraints on **Pods** running in the same **Namespace**

- Still in beta in 1.13
- Relies on the admission plugin **PodSecurityPolicy**
- Must be enabled with the flag on the api server `--enable-admission-plugins=PodSecurityPolicy`

PodSecurityPolicy - What

Properties regarding privileged process and process capabilities:

- **privileged**: Running of privileged containers
- **allowPrivilegeEscalation**,
defaultAllowPrivilegeEscalation: Restricting escalation to root privileges
- **defaultAddCapabilities**, **requiredDropCapabilities**,
allowedCapabilities: Linux capabilities

PodSecurityPolicy - What

Properties related to host:

- **hostPID, hostIPC**: Usage of host namespaces
- **hostNetwork, hostPorts**: Usage of host networking and ports
- **allowedHostPaths**: Usage of the host filesystem
- **allowedProcMountTypes**: The Allowed Proc Mount types for the container

PodSecurityPolicy - What

Volume/FS related properties:

- **volumes**: Usage of volume types
- **allowedFlexVolumes**: White list of Flexvolume drivers
- **fsGroup**: Allocating an FSGroup that owns the pod's volumes
- **readOnlyRootFilesystem**: Requiring the use of a read only root file system
- **runAsUser**, **runAsGroup**, **supplementalGroups**: The user and group IDs of the container

PodSecurityPolicy - What

Properties related to SELinux/AppArmor/...:

- **seLinux**: The SELinux context of the container
- with custom **annotations**
 - The AppArmor profile used by containers
 - The seccomp profile used by containers
 - The sysctl profile used by containers

PodSecurityPolicy example

podsecuritypolicy.yml:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: podsecuritypolicy
spec:
  privileged: false
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  volumes:
    - '*'
```



TP 5.5 & 5.6: Security context & PodSecurityPolicy

Authentication/Authorization/RBAC

- Security model:
 - Authentication: know who emitted the request
 - Authorization: is the requests allowed?
- 4 modes:
 - Previously ABAC (Attribute-based access control)
 - Now RBAC (Role-Based access control)
 - Node
 - Webhook

Authentication - User types

- Normal users
 - Can't be created by simple API calls
 - Managed outside the cluster
- Service accounts
 - Bound to a **Namespace**
 - Created automatically or by API calls
 - Managed inside the cluster with **Secret**

Authentication strategies

Whatever authentication method used, plugins used for authentication will add some of the following information to requests made to the API Server:

- Username
- UID
- Groups
- Extra fields

Authentication methods

- X509 cert
- Static tokens
- Bootstrap tokens
- Static password file
- Service Account tokens
- OpenID Connect tokens
- Authentication webhook
- Authentication proxy

Authentication - X509 Cert

X509 authentication is enabled by passing the argument `--client-ca-file=ca.crt`

- This CA certificate is used to authenticate client certs presented to the API server
- The CN of the certificate is used as the username for the request
- The O fields of the certificates are used as complementary groups

Authentication - Static tokens

Static token authentication is enabled by passing the argument `--token-auth-file=...`

The file content should be something like:

```
token, user, uid, "group1, group2, group3"
```

- There is no time validity limit for tokens
- Changes to the token file need API Server restart to be effective
- The API Server will expect an **Authorization** header with value **Bearer TOKEN**

Example:

```
Authorization: Bearer a9fa448e-f972-4f73-9cb6-df242e788459
```

Authentication - Static password file

Static password file authentication is enabled by passing the argument `--basic-auth-file=...`

The file content should be something like:

```
password,user,uid,"group1,group2,group3"
```

- As for token authentication:
 - There is no time validity limit for passwords
 - Changes to the password file need API Server restart to be effective
- The API Server will expect an **Authorization** header with value basic auth

Example:

```
Authorization: Basic BASE64ENCODED(USER:PASSWORD)
```

Authentication - Service accounts (1/2)

Service accounts are automatically created and associated with **Pods** when the **ServiceAccount** admission controller is enabled.

Then the following secret will be created in the **Namespace** of the SA:

```
apiVersion: v1
data:
  ca.crt: (APISERVER'S CA BASE64 ENCODED)
  namespace: ZGVmYXVsdA== # default encoded in base64
  token: (BEARER TOKEN BASE64 ENCODED)
kind: Secret
metadata:
  # ...
type: kubernetes.io/service-account-token
```

Authentication - Service accounts (2/2)

When authenticating with the previous token, the following informations will be added to the request:

- Username: `system:serviceaccount:(NAMESPACE):(SERVICEACCOUNT)`
- Groups: `system:serviceaccounts` and `system:serviceaccounts:(NAMESPACE)`

WARNING: - Access to the secret allows to use the Service Account - Take care of what Service Accounts are allowed to do

Authentication - Bootstrap tokens

Bootstrap tokens are temporary tokens managed by the cluster which can be used to make a certificate request to the API Server. The request can be approved on the server side with the CSR resource.

- The tokens are of the form `[a-z0-9]{6}.[a-z0-9]{16}`:
 - First is a Token ID
 - Second component is the Token Secret
- The feature is currently in beta
- Bootstrap token are enabled via `--enable-bootstrap-token-auth` on the API Server
- You should enable the TokenCleaner controller on the controller manager with the flag `--controllers=*,tokencleaner`

Authentication OpenID Connect

- OpenID Connect is a flavor of OAuth2 supported by some providers (Azure Active Directory, Salesforce and Google)
- When the user authenticates, it returns an additional field **id_token**
- This **id_token** is sent in requests to the API Server
- To enable this authentication mode, you must add the following flags to the API Server:
 - **--oidc-issuer-url**: URL of the provider which allows the API server to discover public signing keys
 - **--oidc-client-id**: A client id that all tokens must be issued for
 - There are many other options which can be used to configure the token validation

Authentication Webhook

Authentication webhook are used to delegate token validation to another server. Add the flag `--authentication-token-webhook-config-file`

```
apiVersion: v1
kind: Config
clusters:
  - name: name-of-remote-authn-service
    cluster:
      certificate-authority: /path/to/remote-ca.pem
      # URL of remote service to query. Must use 'https://authn.example.com/authenticate
    server: https://authn.example.com/authenticate
# users refers to the API server's webhook configuration.
users:
  - name: name-of-api-server
    user:
      client-certificate: apiserver-client-cert.pem
      client-key: apiserver-client-key.pem
current-context: webhook
contexts:
  - context:
      cluster: name-of-remote-authn-service
      user: name-of-api-server
      name: webhook
```

Authentication Webhook - Request

When receiving a query, the `kube-apiserver` will build and send a query to the configured server

```
{  
  "apiVersion": "authentication.k8s.io/v1beta1",  
  "kind": "TokenReview",  
  "spec": {  
    "token": "(BEARERTOKEN)"  
  }  
}
```

Authentication Webhook - Response

Success response:

```
{  
  "apiVersion": "authentication.k8s.io/v1beta1",  
  "kind": "TokenReview",  
  "status": {  
    "authenticated": true,  
    "user": {  
      "username": "user@mycompany.com",  
      "uid": "1000",  
      "groups": [ "developers" ],  
      "extra": { "extrafield1": [ "..." ] }  
    }  
  }  
}
```

Failure response:

```
{  
  "apiVersion": "authentication.k8s.io/v1beta1",  
  "kind": "TokenReview",  
  "status": {  
    "authenticated": false  
  }  
}
```

Authentication proxy

With this mode, the API server delegates the authentication to a front server. The authentication server sends user information through the use of HTTP headers.

This authentication method uses these flags:

- `--requestheader-username-headers=X-Remote-User`
- `--requestheader-group-headers=X-Remote-Group`
- `--requestheader-extra-headers-prefix=X-Remote-Extra-`

To prevent the use of forged headers, the authenticating proxy must present a client certificate.

*WARNING: Do not reuse this CA in another context
and take care when using this authentication method.*

Authorization (1/3)

- (ABAC)
- RBAC
- Webhook
- Node

Authorization (2/3)

Request attributes:

- user
- group
- extra
- Request Path (if not API resource)
- API
 - Request verb (get, list, create, update, patch, watch, proxy, redirect, delete, deletecollection)
 - Resource / Subresource
 - Namespace
 - API Group

Authorization (3/3)

Complementary specialized verbs on some resources:

- **PodSecurityPolicy**: use on `policy/podsecuritypolicies`
- **RBAC**: bind on `role, clusterrole` in `rbac.authorization.k8s.io`
- **Authentication layer**: `impersonate` on `users, group, serviceaccounts` in the core API group and `userextras` in the `authentication.k8s.io` API group

RBAC and Role binding

Three kind of resources involved:

- User: who?
- Role/ClusterRole: what is allowed?
- RoleBinding/ClusterRoleBinding: Role affectation to user

RBAC - Role

- A role is a set of permission on some resources
- Permissions are additive (only Allow rules)
- Defines permissions on resources within a **Namespace**

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

RBAC - RoleBinding

read-pods-role-binding.yml

```
# This role binding allows "jane" to read pods in the "default" namespace.
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: jane # Name is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role #this must be Role or ClusterRole
  name: pod-reader # this must match the name of the Role or ClusterRole you wish
  to bind to
  apiGroup: rbac.authorization.k8s.io
```

RBAC - ClusterRole/ClusterRoleBinding

A **ClusterRole** is used to grant permissions on resources:

- Cluster scoped resources (See `kubectl api-resources`)
- Namespaced resources accross all **Namespaces**
- Non-resource endpoints (like `/healthz`)

`secret-reader-cluster-role.yml`

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: secret-reader
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

RBAC - ClusterRole aggregation (1/2)

cluster-role-aggregate.yaml

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: monitoring
aggregationRule:
  clusterRoleSelectors:
  - matchLabels:
      rbac.example.com/aggregate-to-monitoring: "true"
rules: [] # Rules are automatically filled in by the controller manager.
```

RBAC - ClusterRole aggregation (2/2)

cluster-role-aggregated.yaml

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: monitoring-endpoints
  labels:
    rbac.example.com/aggregate-to-monitoring: "true"
# These rules will be added to the "monitoring" role.
rules:
- apiGroups: [""]
  resources: ["services", "endpoints", "pods"]
  verbs: ["get", "list", "watch"]
```

RBAC - Complement

To keep your cluster safe and healthy, you should limit access at least to these resources:

- **Role**
- **RoleBinding**
- **ClusterRoleBinding**
- **PodSecurityPolicy**
- **LimitRanges**
- **ResourceQuotas**
- **NetworkPolicy**



TP 5.7: RBAC

Authorization - Webhook

This authorization mode, makes the API server sending a query to an external service when determining user privileges.

To be activated, the Webhook must be configured with the flag `--authorization-webhook-config-file=webhook-conf.yml`

```
apiVersion: v1
kind: Config
clusters:
  - name: name-of-remote-authz-service
    cluster:
      certificate-authority: ca-remote-svc.pem
      server: https://remote.authz.svc.com/authorize
users:
  - name: name-of-api-server
    user:
      client-certificate: webhook-client-cert.pem
      client-key: webhook-client-key.pem
current-context: webhook
contexts:
  - context:
      cluster: name-of-remote-authz-service
      user: name-of-api-server
      name: webhook
```

Authorization - Webhook example

subject-access-review.json

```
{  
  "apiVersion": "authorization.k8s.io/v1beta1",  
  "kind": "SubjectAccessReview",  
  "spec": {  
    "resourceAttributes": {  
      "namespace": "kittensandponies",  
      "verb": "get",  
      "group": "unicorn.example.org",  
      "resource": "pods"  
    },  
    "user": "jane",  
    "group": [  
      "group1",  
      "group2"  
    ]  
  }  
}
```

Authorization - Webhook result

access-denied.json

```
{  
  "apiVersion": "authorization.k8s.io/v1beta1",  
  "kind": "SubjectAccessReview",  
  "status": {  
    "allowed": false,  
    "reason": "user does not have read access to the namespace"  
  }  
}
```

Authorization - Node

The **Node** authorizer is used to allow **kubelets** to perform required API operations:

- Read: services, endpoints, nodes, pods, secrets, configmap, pvc/pv
- Write: nodes, node status, pods, pods status, events
- Read/Write access to CSR API for TLS bootstrapping
- Create: tokenreviews, subjectaccessreviews for delegated auth/authz checks

Used with the **NodeRestriction** admission plugin, the **Node** authorizer limits access to owned resources.

Persistent volumes

- Create different **storageClass** related to users needs
- Take care of the freed resources
- Use **Quotas** to keep resources available and make capacity planning sustainable



Extensibility: Operators, CRD and API Servers

Table of contents

- Architecture
- Installation
- Configuration and operational maintenance
- Cluster Upgrade
- Day to day actions
- *Extensibility: Operators, CRD and API Servers*
- Federation, Service Mesh, Security

Chapter content

- Controllers
- Dynamic Admission Controllers
- Custom Resource Definition
- Operators
- API Servers/Aggregation layer

Scenarios

Imagine that you want to:

- Control user label when a resource is created
- Reject namespace deletion if not empty
- Inject a container automatically when a pod is created
- Create a Postgresql object to deploy a database, create a database, a user, etc..

Introduction

Kubernetes is highly configurable and extensible. It allows you to create:

- your own API objects (**CRD**)
- your own mechanism to control resources (**Admission Controllers**)
- and a lot of other things using **Operators**.

Controllers (1/2)

In Kubernetes, a controller is a control loop that watches the shared state of the cluster through the apiserver and makes changes attempting to move the current state towards the desired state

- Mainly in Golang
- It's recommended to use a dedicated service account (limited access)
- Keep it Simple

Controllers (2/2)

Example: You want to watch namespace creation...

```
namespaceInformer := cache.NewSharedIndexInformer(  
    &cache.ListWatch{  
        ListFunc: func(options metav1.ListOptions) (runtime.Object, error) {  
            return kclient.CoreV1().Namespaces().List(options)  
        },  
        WatchFunc: func(options metav1.ListOptions) (watch.Interface, error) {  
            return kclient.CoreV1().Namespaces().Watch(options)  
        },  
    },  
    &v1.Namespace{},  
    3*time.Minute,  
    cache.Indexers{cache.NamespaceIndex: cache.MetaNamespaceIndexFunc},  
)  
  
...  
  
namespaceInformer.AddEventHandler(cache.ResourceEventHandlerFuncs{  
    AddFunc: namespaceWatcher.createCustomerRules,  
})
```

If you want to deep dive into controllers, please [read this article](#)

Dynamic Admission Controllers (1/4)

Admission webhooks are *HTTP callbacks that receive admission requests and do something with them.*

2 native types in admissionregistration.k8s.io/v1beta1:

- **Validating admissionwebhook**
- **Mutating admissionwebhook**

Note that Initializers is still in alpha and will be deprecated. [source](#)

The principle is simple, each time you have an event, api-server will trigger an internal service in SSL to validate, deny, or modify a resource.

Dynamic Admission Controllers (2/4)

Use cases:

- You can use Validating admission webhook to control if annotations are well defined
- You can use Validating admission webhook to avoid deletion of non empty namespace
- Istio use Mutating admission webhook to inject Envoy container inside each pod
- You can use Mutating admission webhook to inject Vault init container

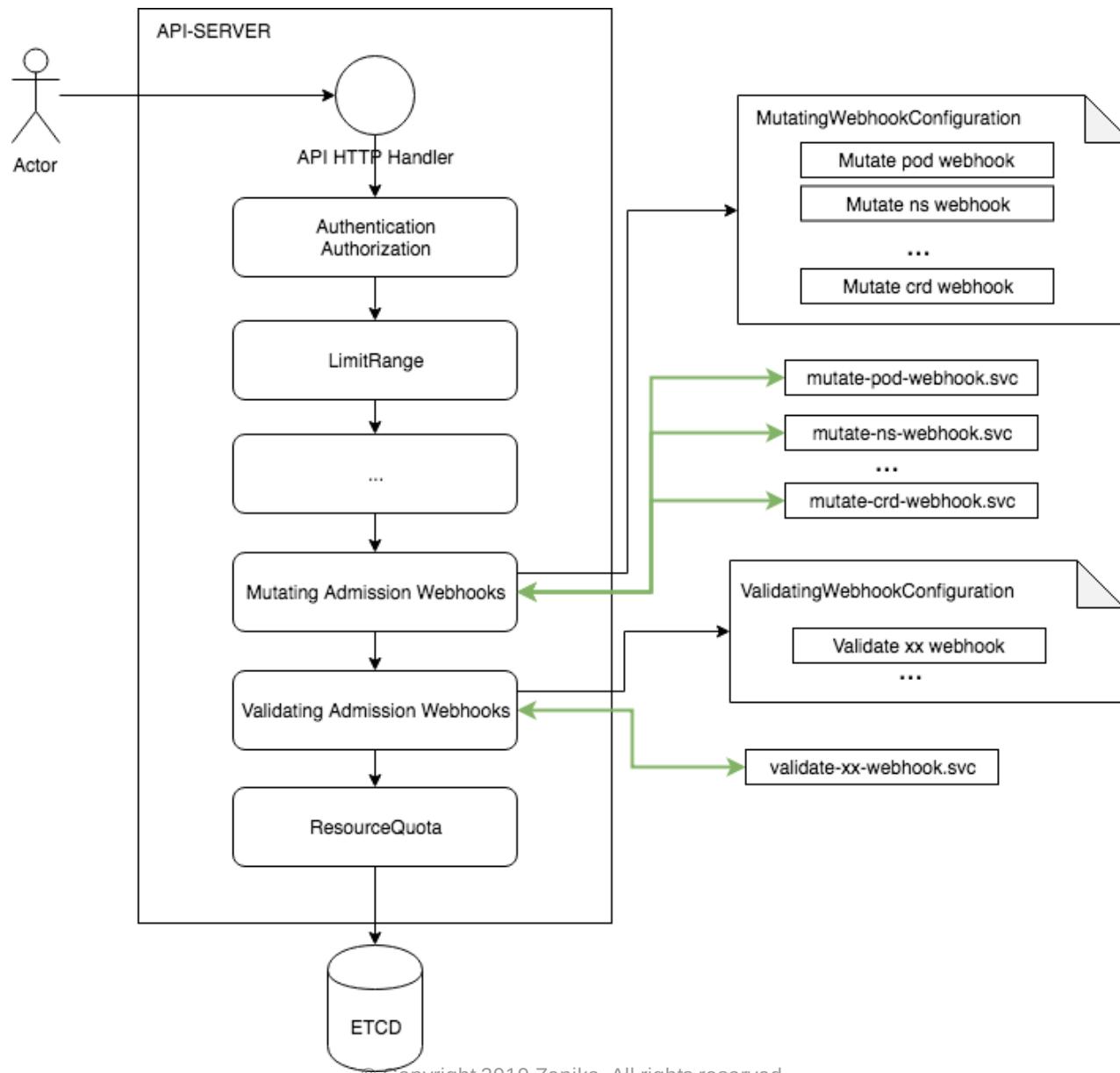
If you want to deep dive into Dynamic admission controllers, you can find a lot of information on the [IBM medium blog](#)

Dynamic Admission Controllers (3/4)

Example of configuration:

```
apiVersion: admissionregistration.k8s.io/v1beta1
kind: MutatingWebhookConfiguration
metadata:
  name: pod-mutating-webhook
webhooks:
- clientConfig:
    caBundle: xxxxxxxxxxxxxxxx==
    service:
      name: pod-mutating-webhook
      namespace: kube-system
      path: /mutate
    failurePolicy: Ignore
    name: pod-mutating-webhook.custom.com
    namespaceSelector: {}
rules:
- apiGroups:
  - "v1"
  apiVersions:
  - v1
  operations:
  - CREATE
  resources:
  - pods
```

Dynamic Admission Controllers (4/4)



Custom Resource Definition (1/2)

The `CustomResourceDefinition` API resource allows you to define custom resources. Defining a CRD object creates a new custom resource with a name and schema that you specify. The Kubernetes API serves and handles the storage of your custom resource.

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * * */5"
  image: my-awesome-cron-image
```

```
$ kubectl get crontab
```

NAME	AGE
my-new-cron-object	6s

Custom Resource Definition (2/2)

Resource definition example:

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below, and be in the form: <plural>.<group>
  name: crontabs.stable.example.com
spec:
  # group name to use for REST API: /apis/<group>/<version>
  group: stable.example.com
  versions:
    - name: v1
      served: true
      storage: true
  # either Namespaced or Cluster
  scope: Namespaced
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
    shortNames:
      - ct
```





TP 6.1 : Custom Resource Definition

Operators (1/2)

Kubernetes Operators is a concept created by [CoreOS](#)

The main function of an Operator is to:

- Read from a custom object that represents your application instance
- Make changes to what is running to match this desired state

Example:

```
apiVersion: "vault.security.coreos.com/v1alpha1"
kind: "VaultService"
metadata:
  name: "example"
spec:
  nodes: 2
  version: "0.9.1-0"
```

Operators (2/2)

Operator = controller pattern + API extension + single-app focus

How to create your own?

- From scratch: Create your own CRD and controller [example](#)
- Using Kubebuilder: [example](#)
- Using operator-sdk: [example](#)

An awesome list of existing operator can be found on Github:
<https://github.com/operator-framework/awesome-operators>

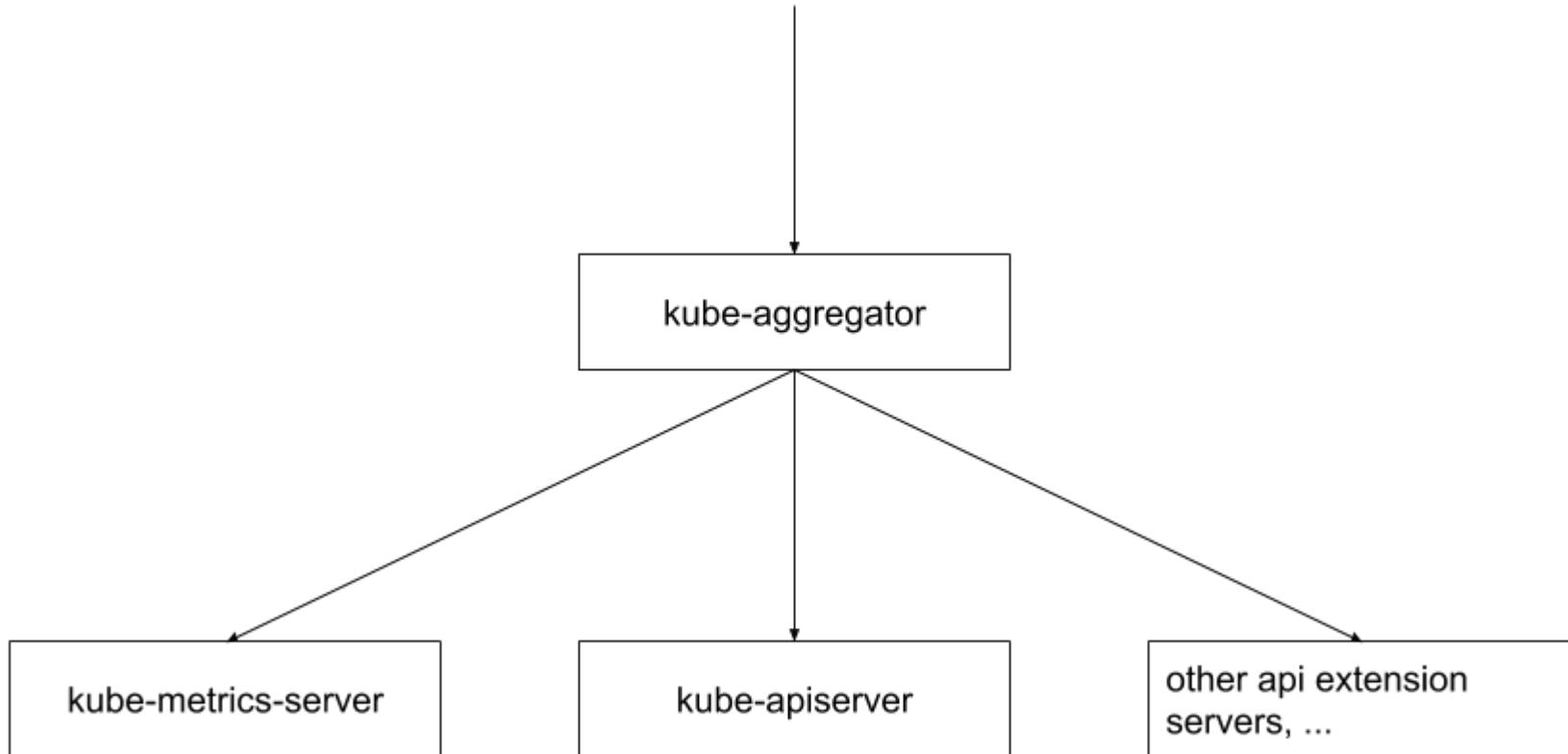
API Servers/Aggregation layer

CRD is a really good way to expose new API resources and use Kubernetes to store their state.

However, sometimes you might need to expose new API resources and handle their storage in a specific way.

e.g: Exposing metrics as API resources, but not using etcd to store them

Aggregation layer - kube-aggregator



API Server configuration

To activate the aggregation layer, you must add the following flags to the kube-apiserver:

```
--requestheader-client-ca-file=<path to aggregator CA cert>
--requestheader-allowed-names=front-proxy-client
--requestheader-extra-headers-prefix=X-Remote-Extra-
--requestheader-group-headers=X-Remote-Group
--requestheader-username-headers=X-Remote-User
--proxy-client-cert-file=<path to aggregator proxy cert>
--proxy-client-key-file=<path to aggregator proxy key>
```

If **kube-proxy** is not running on host running the API server, add the flag: **--enable-aggregator-routing=true**

API service example

Then to register new `api-resources` you must declare an **APIService** to the kube-apiserver

`metrics-apiservice.yaml`

```
apiVersion: apiregistration.k8s.io/v1beta1
kind: APIService
metadata:
  name: v1beta1.metrics.k8s.io
spec:
  service:
    name: metrics-server
    namespace: kube-system
  group: metrics.k8s.io
  version: v1beta1
  insecureSkipTLSVerify: true
  groupPriorityMinimum: 100
  versionPriority: 100
```





TP 6.2 : Aggregation layer

Federation, Service Mesh, Security

Table of contents

- Architecture
- Installation
- Configuration and operational maintenance
- Cluster Upgrade
- Day to day actions
- Extensibility: Operators, CRD and API Servers
- *Federation, Service Mesh, Security*

Chapter content

- Federation
 - Principles
 - History
- Service Mesh
- Security

Federation

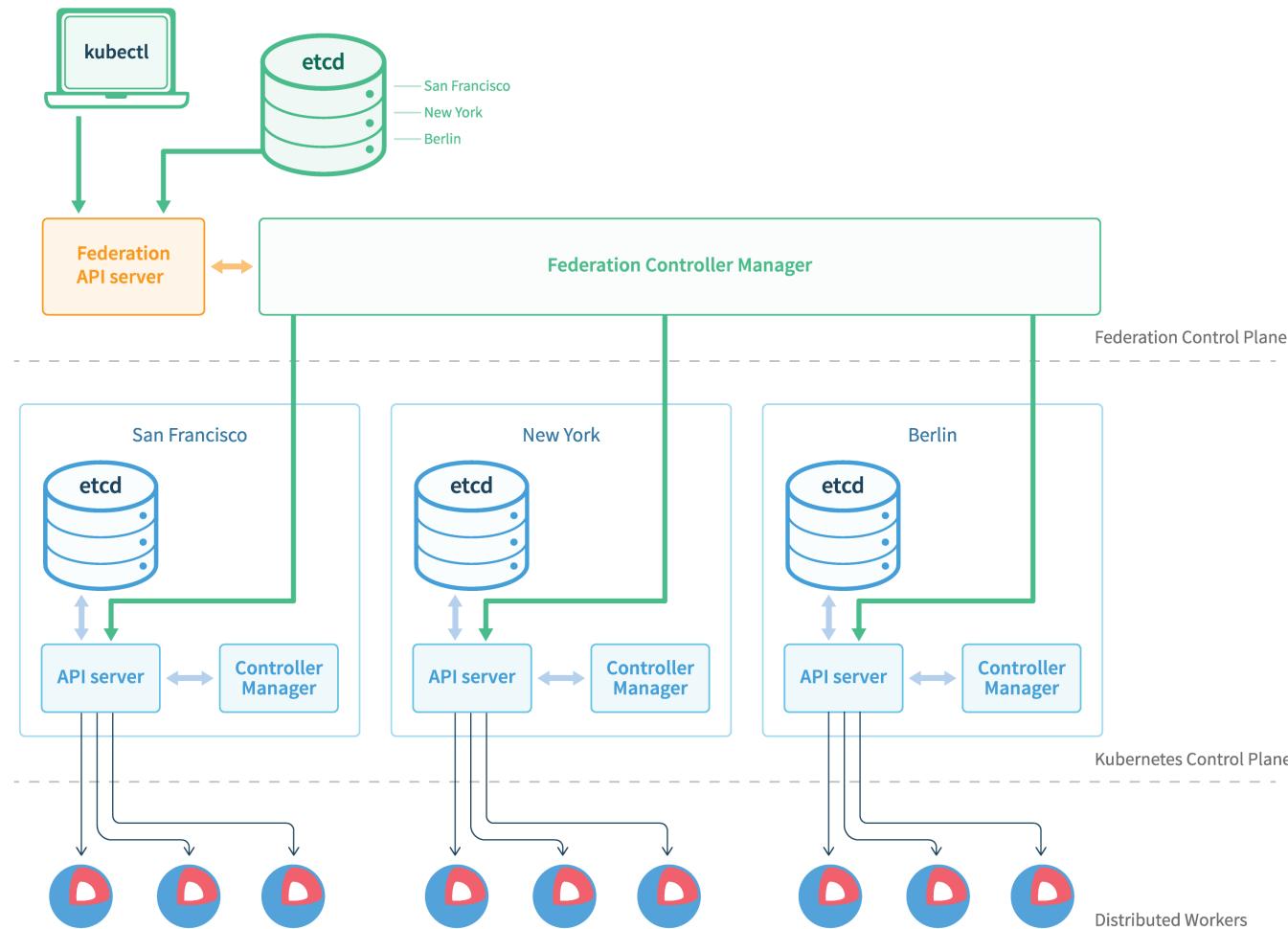
- The cluster state is stored in etcd
- etcd uses the Raft consensus algorithm
- Consensus performance, especially commit latency, is **limited by two physical constraints**:
 - network IO latency
 - disk IO latency

Thus it's not very efficient to have a cluster with etcd nodes far from each other

Federation principles

- The first proposed solution was to use a standard **Control Plane**:
 - Exposing the standard APIs
 - Distributing the workloads on clusters instead of **Nodes**
 - Easing service discovery at federation level
- A v1 Federated cluster can be set up with the help of the **kubefed** tool

Federation v1 architecture



-- Source: <https://coreos.com/blog/kubernetes-cluster-federation.html>

© Copyright 2019 Zenika. All rights reserved

Federation history

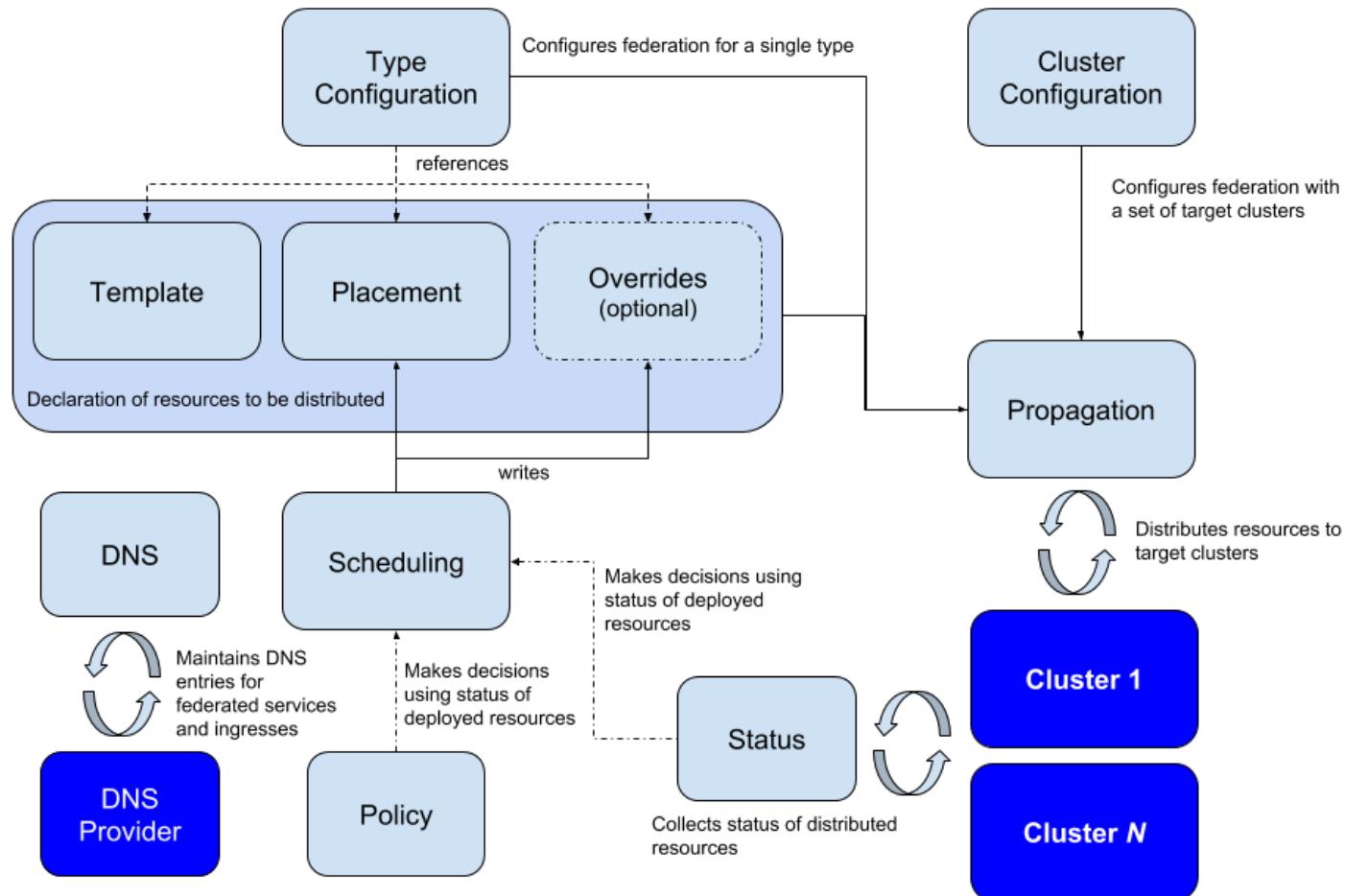
- This first federation solution was not viable
 - Difficulties in re-implementing the **Kubernetes** API at the cluster level
 - Federation-specific extensions were stored in annotations
 - Limited flexibility in federated types, placement and reconciliation
 - Due to 1:1 emulation of the **Kubernetes** API
 - No settled path to GA, and general confusion on API maturity
 - For example, **Deployments** are GA in **Kubernetes** but not even Beta in **Federation v1**.

Federation v1 status

From the **Kubernetes** documentation:

Federation V1, the current **Kubernetes** federation API which reuses the **Kubernetes** API resources ‘as is’, is currently considered alpha for many of its features. There is no clear path to evolve the API to GA; however, there is a **Federation V2** effort in progress to implement a dedicated federation API apart from the **Kubernetes** API.

Federation v2 concepts



-- Source: <https://kubernetes.io/blog/2018/12/12/kubernetes-federation-evolution/>

© Copyright 2019 Zenika. All rights reserved

Federation v2 status

- Alpha/Beta release ideally due before end of Q1 2019
 - Must-include features for Alpha
 - Update docs (readme) later or add to the list.
 - Must-include features for Beta
 - Finalise the API constructions, Propagation
- Needs more elaboration: put up matrix of features/maturity in README and as guide for next discussion

This information is extracted from the community sig-multicloud meeting notes

-- Source: <https://tinyurl.com/federation-v2-notes>



Service mesh

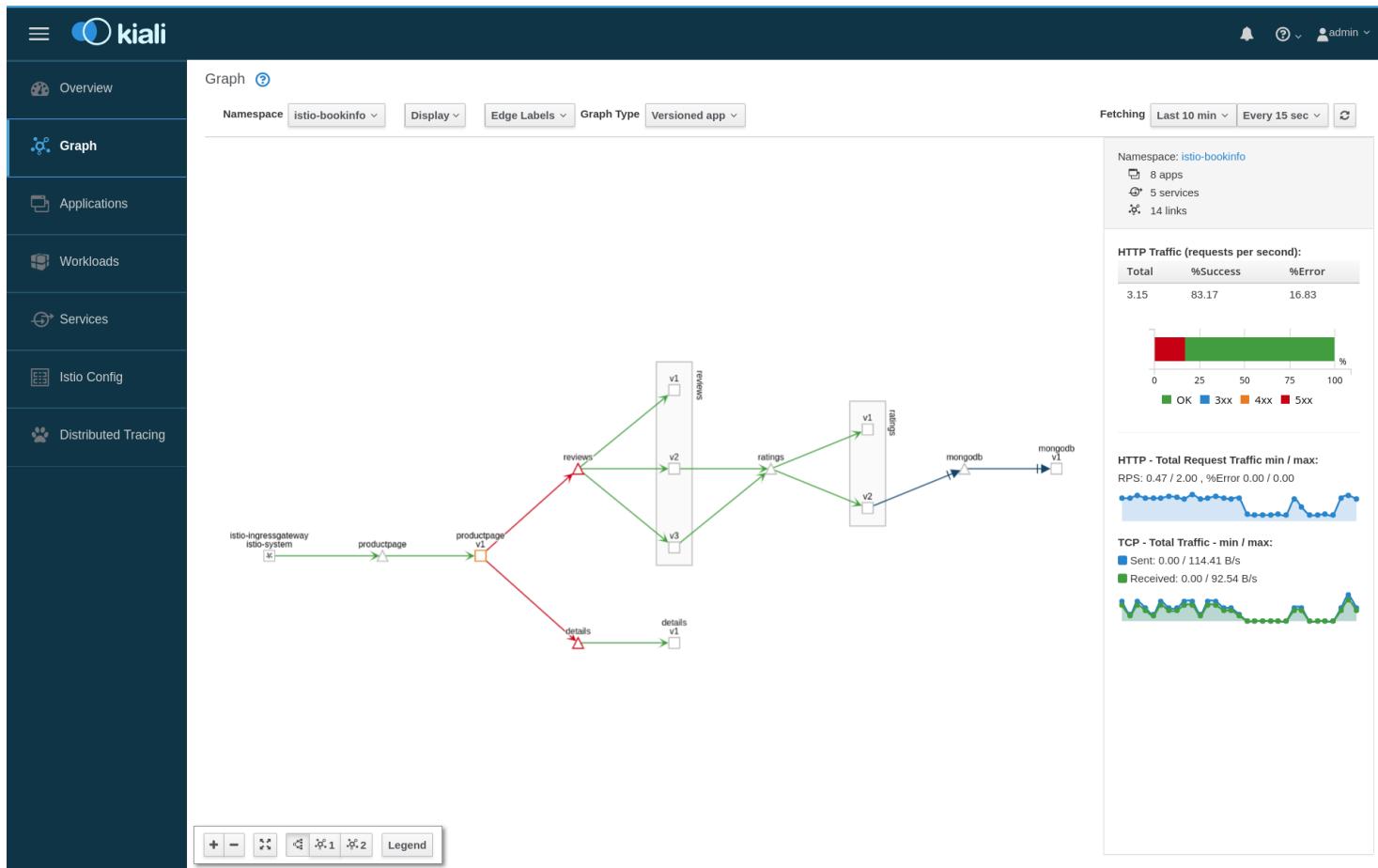
- Dedicated infrastructure layer for service-to-service communication
- Addresses safety, reliability
- Might also be used for/as:
 - Circuit breaker
 - Traffic splitter/shaper
 - Distributed tracing
 - TLS encryption
 - ...

-- Source: <https://blog.buoyant.io/2017/04/25/whats-a-service-mesh-and-why-do-i-need-one/>

Service mesh - Deployment example

- Service mesh usually (at least **Istio** does) relies on the **Mutating Webhook** admission controller
 - When a **Pod** is created an **Envoy** container is added to it
 - All communications go through this proxy
 - The "state" (configuration) of these proxies is stored in etcd
 - The **Pilot** component of Istio manages **Envoy** proxies configuration

Service mesh - Observability



-- Source: <https://www.kiali.io/>

Service mesh - Observability - How it works

- **Envoy** injects a Correlation Id in a coming request if not present
- The application must convey the Correlation Id in output requests
- Traffic metrics are built from the Correlation Ids
- Visualisation is built upon these metrics

Correlation Id: Unique identifier value that is attached to requests and messages that allow reference to a particular transaction or event chain.



Security

- There are many tools and articles available on how to secure a **Kubernetes** cluster:
 - Security
 - Securing Kubernetes
 - kube-bench

Security - Cluster

- Use TLS for all communications
- Enable RBAC, disable ABAC
 - Limit service account and users rights
- Limit access to trusted registries
- Audit your cluster for vulnerabilities/misconfigurations
- Update your cluster regularly

Security - Workloads

- Run containers as NonRoot
- Use **PodSecurityPolicies** and Linux security features to apply least privilege to **Workloads**
- Audit manifests with tools like **kubesc**
 - Generates risk scores for resources
 - Available as **kubectl** plugin
 - Available online: <https://kubesc.io/>
- Use **NetworkPolicies**

Security - Images

- Scan images for vulnerabilities
 - Clair
 - Aquasec microscanner
 - Anchore
- Sign images and manage the whole software supply chain
 - Notary
 - Grafeas

Encryption at rest - Why

- A **Kubernetes** cluster stores its state in etcd.
- By default, this data is unencrypted, including the **Secrets**.
- Encrypting data at rest is fairly simple and should be activated.

See: *Encrypting Secret Data at Rest*

Encryption at rest - How 1/2

- Generate an encryption key: `ENCRYPTION_KEY=$(head -c 32 /dev/urandom | base64)`
- Create an encryption config: `encryption-config.yaml`

```
kind: EncryptionConfig
apiVersion: v1
resources:
  - resources:
    - secrets
providers:
  - aescbc:
    keys:
      - name: key1
        secret: ${ENCRYPTION_KEY}
  - identity: {}
```

Encryption at rest - How 2/2

- Configure the `kube-apiserver`
 - Pass the `--encryption-provider-config` flag pointing to the created `encryption-config.yaml`
- Any new **Secret** will then be encrypted
- To make sure all **Secrets** are now encrypted, launch: `kubectl get secrets --all-namespaces -o json | kubectl replace -f -`



Summary

Installation

- Automate cluster installation
- Do not begin with a giant cluster
- Use a distribution which fits your need
- Choose and deploy:
 - A network addon
 - At least one storage provisioner
- Centralize logs and metrics

Day to day actions

- Keep up with **Kubernetes**
 - Automate cluster upgrade
- Keep your **Nodes** up to date
- Teach users how to deploy on **Kubernetes**
 - Liveness/Readiness probes
 - Resources limits/requests
 - ...
- Use **Quotas/LimitRanges** to guide users
- Configure garbage collection

Security

- Apply security at all level
 - Host OS
 - Cluster
 - Docker images
 - Workloads

Questions

