Mastering Kubernetes

Concepts, Development, and Operations

Training objectives

- Understand containers technology
- Understand Kubernetes design, architecture and main components
- Design and deploy Cloud Native Applications with Kubernetes

Methodology

- Teaching by examples
- Theoretical and Practical approach
- Step-by-step Lab Tutorial
- Learning Checkpoints

Prerequisites

- Linux knowledge and practice
- Computer programming concepts and methodologies

Expected background

- Development
- Operations
- DevOps
- QA Engineer
- IT Architect
- IT Project Manager
- IT Decisions maker

Practice

- It will take more than merely reading documentation to make you an expert.
- Instead, let's to practice.
- If you are attending a workshop or a trainer-led class, follow instructions from your instructor to access the cloud environment.
- If you are doing this on your own, you can use the Katakoda free service.

Modules

1. Containers Review
2. Kubernetes Essentials
3. Advanced Kubernetes
4. Applications Design Patterns
5. Adopting a container based solution

Module 1

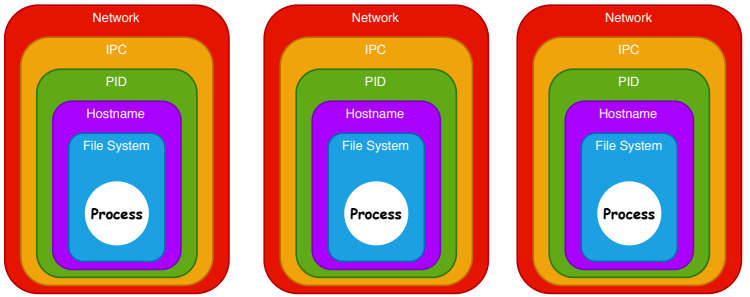
Containers Review

Linux Containers

- A container is a technique to isolate a process from other processes running on the same machine.
- It uses mostly standard Linux features, as for example (not limited to):
  - **namespaces:** a partition of system resources allowing sandboxing and isolation between processes.
  - **cgroups:** a kernel feature to control and limit the resource usage like CPU, memory, disk I/O, network, etc.
  - **layered filesytems:** a tecnique to allow multiple filesystems to be presented as a single filesystem.
- Containers existed in Linux/Unix before anyone cared of them: e.g. BSD Jails, Solaris Zones, LXC, etc.
- Docker just made using Linux containers easier and mass adoption followed.

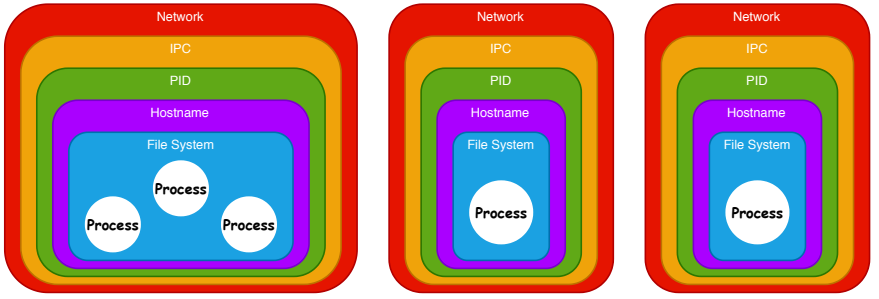
Container levels of isolation

Containers create multiple levels of isolation for a process



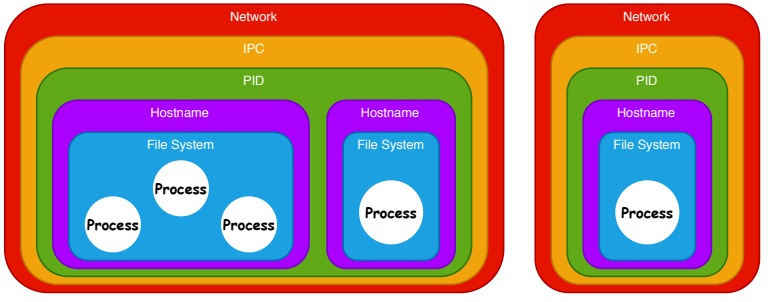
Multiple processes in a container

We can combine multiple processes in the same container



Sharing namespaces between containers

Containers can share some namespaces where keeping other private



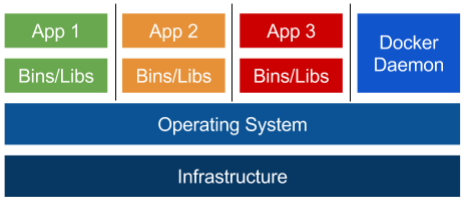
Containers advantages

Running an application within a container offers the following advantages:

- Images contain only the content needed to run the application. It is much more efficient with containers than with virtual machines which include the whole operating system.
- Improved performance since it is not running an entirely separate operating system. A container typically runs faster than a virtual machine.
- Applications running in that container can be isolated and secured from other processes on the host machine.
- With the application runtime included in the container, a container is capable of being run in multiple environments without any changes.

The Docker model

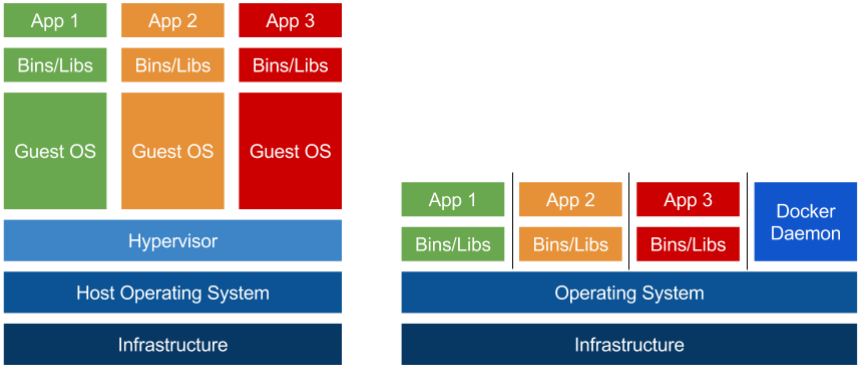
The Docker Engine is the Linux daemon responsible for management the containers



The Docker CLI is client command line tool to interact with Docker Engine

Containers vs Virtual Machines

A different way to achieve isolation between processes running on the same physical machine.



They are different but not mutually exclusive!

Running a container

Run an application by using the Docker client

```
docker run busybox:latest echo 'Hello Docker'
docker run busybox:latest whoami
```

When the user specifies an image, Docker looks first for the image on local host. If the image does not exist locally, then the image is pulled from the public image registry Docker Hub.

Run the Busybox container interactively

```
docker run -it busybox
```

Running a container 1

Run a container in daemon mode on a random port

```
docker run -d httpd
```

Run a container in daemon mode on a defined container port 80

```
docker run -d -p 80 httpd
```

Run a container in daemon mode and map the container port 80 to the host port 4000

```
docker run -d -p 4000:80 httpd
```

Running a container 3

Specify the container name

```
docker run -d -p 4000:80 --name webserver httpd
```

If you do not specify name, then a random name is used. See code here.

List running containers

```
docker ps
```

List all containers

```
docker ps -a
```

Running a container 4

By default, a container has no resource constraints and can use cpu, memory, and io as the host's kernel permits.

Control how much host resources a container can use

```
docker run -d -p 4000:80 --name webserver --memory 400m --cpus 0.5 httpd
```

Control a container

```
docker stop|start|restart webserver
```

Running a container 5

Remove a stopped container

```
docker rm webserver
```

Force remove a running container

```
docker rm -f webserver
```

Remove all containers

```
docker rm $(docker ps -aq)
```

Start a container and remove it when the container exits

```
docker run --rm -it busybox
```

Inspecting a container

Inspect a container

```
docker inspect webserver
```

See logs from a running container

```
docker logs -f webserver
```

Inspecting a container 1

Examine the processes running inside a container

```
docker top webserver
```

See the ports used by a daemonized container

```
docker port webserver
```

To investigate within a running Docker container attach a bash shell to the container

```
docker run -d -p 80:80 --name web httpd
docker exec -it web /bin/bash
```

Working with images

When the client specifies an image, Docker looks first for the image on local host.

If the image does not exist locally, then the image is pulled from the public image registry **Docker Hub**.

Search an image from the Docker Hub

```
docker search mysql
```

Pull an image from the Docker Hub

```
docker pull mysql/mysql-server
```

Working with images 1

List the local images

```
docker images
```

Start a container from the local image

```
docker run -it centos
```

Remove a locally stored image

```
docker rmi mysql/mysql-server
```

Layered Filesystems

Docker uses a layered storage architecture for the images and containers. A layered filesystem is made of many separate layers to be combined and presented to the user as a single layer, creating the illusion that all files can be changed, including the files belonging to a read only layer.

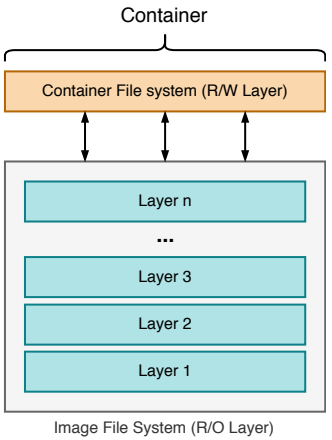
Some of supported layered file systems are:

- device mapper
- btrfs
- aufs
- overlay2

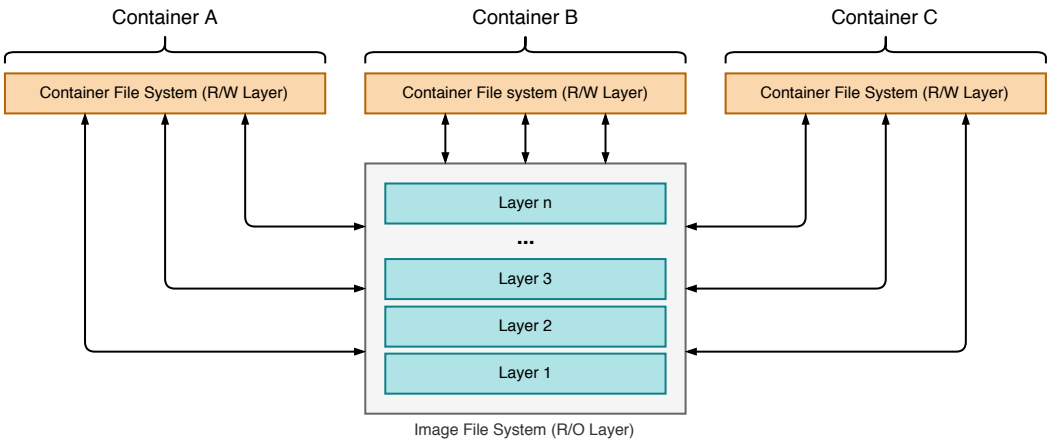
To check the used layered file system, inspect the docker engine

```
docker system info | grep -i "Storage Driver"
```

Layered Filesystem 1



Layered Filesystem 2



Persistent Volumes

- Data created in the read write container filesystem are not persistent
- Applications require persistent storage for using, capturing, or saving data beyond the container life cycle.
- Utilizing persistent volume storage is a way to keep data persistence.
- As best practice, it is high recommended to isolate the data need persistence.
- Persistent storage is an important use case, especially for things like databases.

Persistent Volumes 1

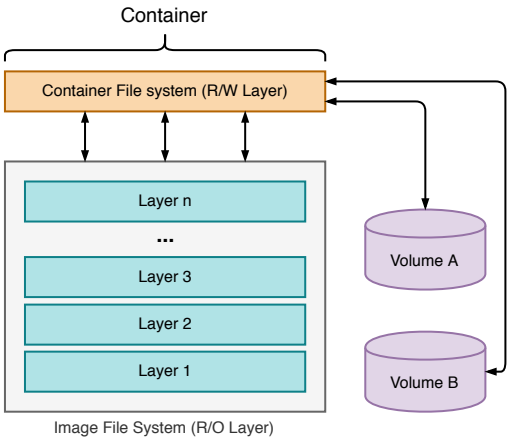
To achieve this goal, there are two different approaches:

1. **host based** volumes: reside on the same host where container is running
2. **shared** volumes: reside on a shared filesystem like NFS, GlusterFS or others

In the first case, data are persistent to the host, meaning if the container is moved to an another host, the content of the volume is no more accessible from the new container.

In the second case, the volume can be mounted by any machine, no matter the host is running the container. It provides data persistence across a cluster of hosts.

Persistent Volumes 2



Persistent Volumes 3

Persistent volumes are mapped from volumes defined into Dockerfile to filesystem on the host running the container.

Start a container from the nodejs image

```
docker run --name=nodejs \
   -p 80:8080 -d \
   -e MESSAGE="Hello" \
kalise/nodejs-web-app:latest
```

This will create a volume for /var/log dir since this dir has been declared as volume in the Dockerfile.

Inspect the container to check the volumes used by

```
docker inspect nodejs
```

Persistent volumes are created under /var/lib/docker/volumes directory.

Persistent Volumes 4

A volume persists even if the container itself is deleted.

```
docker rm -f nodejs
ls -l /var/lib/docker/volumes/84894a09fe25f503cd0f2d3a30eaa00a08d72190a92e2568d395cea5a277c456/
```

To manually delete a volume, find the volume and remove it

```
docker volume list
docker volume remove 84894a09fe25f503cd0f2d3a30eaa00a08d72190a92e2568d395cea5a277c456
```

or prune all non used volumes

```
docker volume prune
WARNING! This will remove all local volumes not used by at least one container.
Are you sure you want to continue? [y/N] y
Total reclaimed space: 250B
```

Persistent Volumes 5

Persistent volumes can be created before the container and then attached to the container

```
docker volume create --name myvolume
docker volume inspect myvolume
docker run --name=nodejs \
     -p 80:8080 -d \
     -e MESSAGE="Hello" \
     -v myvolume:/var/log \
     kalise/nodejs-web-app:latest
docker inspect nodejs
```

Persistent Volumes 6

Persistent volumes can be mounted on any directory of the host file system.

```
docker run --name=nodejs \
  -p 80:8080 -d \
  -e MESSAGE="Hello" \
  -v /logs:/var/log \
kalise/nodejs-web-app:latest
```

Volume data now are placed on the /logs host directory. This helps sharing data between containers and host itself.

The same volume could be mounted by another container, for example performing some analytics on the logs produced by the nodejs application.

However, multiple containers writing to a single shared volume can cause data corruption. Make sure the application is designed to write to shared data stores.

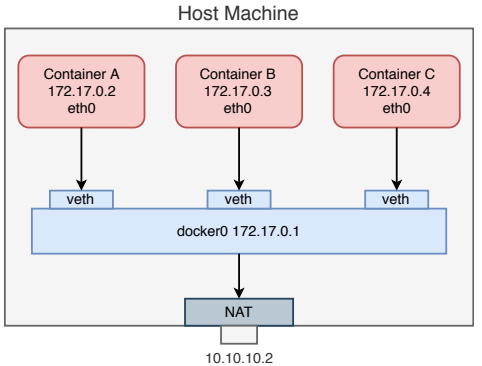
Registry

A Docker registry service is a storage and content delivery system containing tagged images.

Main registry service is the official Docker Hub but a user can build his own private registry behind the firewall.

Users interact with the registry by using push and pull commands.

Docker network model



- Each container will get its own private IP address
- Containers can talk each other via their IP addresses
- Containers can reach any destination the host machine is able to reach
- Containers can be reached from the external through the host machine

Default networks

Installing the Docker Engine, there are three default networks on the host machine

```
docker network list
NETWORK ID          NAME                DRIVER
34a43cf4f024        bridge              bridge
83af822c1610        host                host
02c1fcf12795        none                null
```

The bridge network represents the docker0 network present in all Docker installations as part of a host network stack

```
ifconfig
```

The docker0 interface belongs to a Linux bridge network created by docker at daemon start time.

```
brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.024267786b35       no
```

Default networks 1

Create a container and check the host is creating a new virtual Ethernet interface:

```
docker run -d -p 80:80 --name webserver httpd:latest
ifconfig
brctl show
```

Connect to the container just created to inspect its own network stack

```
docker exec -it webserver /bin/bash
/ # apt update && apt install net-tools -y
/ # ifconfig
/ # exit
```

Inspect the bridge network

```
docker network inspect bridge
```

Bridge network is the default option. Running a container, it automatically adds the new container to the bridge network.

Default networks 2

The other two options are:

1. host mode
2. none mode

In the host mode, the container shares the networking namespace of the host

```
docker run -d -p 80 --net=host --name webserver httpd
```

Inspect the host network

```
docker network inspect host
```

The none mode does not configure networking

```
docker run --net=none -it busybox
/ # ifconfig
```

Default networks 3

A container can be forced to reuse the network of another container

```
docker run -d -p 80:80 --name=web httpd:latest
docker run -d --net container:web -e MYSQL_RANDOM_ROOT_PASSWORD=true mysql:latest
```

Connect to the first container and check its network stack

```
docker exec -it web bash
/ # apt update && apt install net-tools -y
/ # ifconfig
/ # netstat -natp
/ # exit
```

Do the same for the second container.

Note: two containers sharing the same network cannot use the same port.

Sysadmin bonus: network namespaces

Docker networking leverages on a Linux kernel feature called *network namespaces*.

A Linux machine can have multiple network namespaces.

Each Docker container has its own network namespace.

Start a container on your Linux machine and check the network namespaces:

```
sudo ip netns
```

If no output is because Linux expects namespace listed under */var/run/netns* directory.

In order to use standard Linux tools, you should link docker namespace */var/run/docker/netns* directory

```
cd /var/run/
sudo ln -s /var/run/docker/netns .
sudo ip netns
```

Module 2

Kubernetes Essentials

What is Kubernetes?

kubernetes

Open Source platform for orchestrating containers in production.

Why Kubernetes?

Running containers in production:

- Clustering
- Scheduling
- High Availability
- Load Balancing
- Isolation

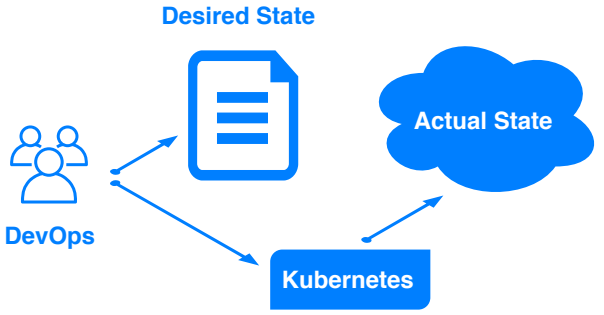
What does Kubernetes mean?

Ancient Greek word for *"Helmsman"* of the ship.

Who owns Kubernetes?

- Kubernetes was started by Google, open sourced in 2015, and now managed by the **Cloud Native Computing Foundation**.
- The Cloud Native Computing Foundation is a child entity of the **Linux Foundation** and operates as a vendor neutral governance group.

The Kubernetes Model

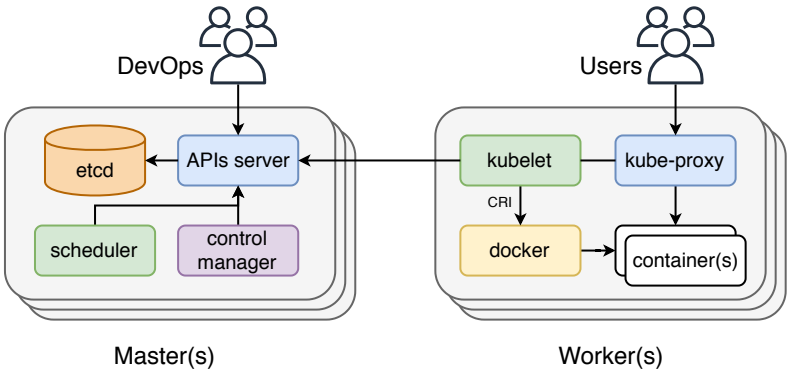


Tell Kubernetes about the desired state of your world.

Declarative and Self Healing

- Kubernetes: *I will always try and steer the system to your desired state.*
- You: *I want 3 healthy instances of my application to always be up & running.*
- Kubernetes: *Okay, I'll ensure there are always 3 instances up & running.*
- ...
- Kubernetes: *Oh look, one instance has died.*
- You: *What?!*
- Kubernetes: *Don't worry, I'm going to spin up a new one.*
- You: *Thanks!*

The Kubernetes Architecture



Multiple master and worker nodes for redundancy.

Control Plane: kube-apiserver

- Forward facing REST interface into the kubernetes control plane and datastore.
- All clients and other applications interact with kubernetes strictly through the APIs Server.
- Acts as the gatekeeper to the system by handling authentication and authorisation, request validation, and admission control.
- Behaves as the front-end to the backing etcd datastore.

Control Plane: etcd

Based on Raft Consensus to create a fault-tolerant consistent view of the cluster.

- Acts as the system datastore.
- Provides a strong, consistent and highly available key-value store for persisting state.
- Stores objects and config information.

Control Plane: kube-scheduler

- Evaluates workload requirements and attempts to place it on a matching resource.
- Workload requirements can include:
  - general hardware requirements
  - affinity and anti-affinity
  - taints and tolerations
  - labels
  - various custom resource requirements

Control Plane: kube-controller-manager

- Serves as the primary daemon that manages all core component control loops.
- Monitors the cluster state via the APIs server and steers the cluster towards the desired state.

Workers: kubelet

- Acts as agent responsible for managing the entire lifecycle of workloads.
- Reports events to the APIs server
- Interacts with the container runtime

Workers: kube-proxy

- Manages the network rules on each node.
- Performs connection forwarding and load balancing.
- Available running modes:
  - userspace
  - iptables
  - ipvs

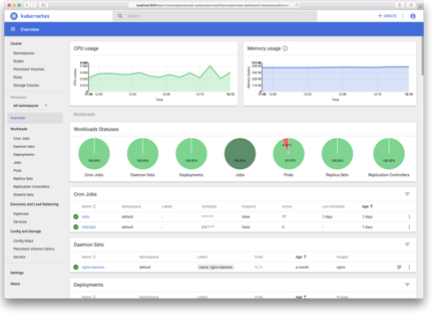
Workers: containers runtime

- A CRI (Container Runtime Interface) compatible application that executes and manages containers.
- Common options:
  - docker
  - containerd
  - cri-o

Additional components: DNS

- Provides cluster wide DNS service for Kubernetes.
- Current options is CoreDNS

Additional components: Dashboard

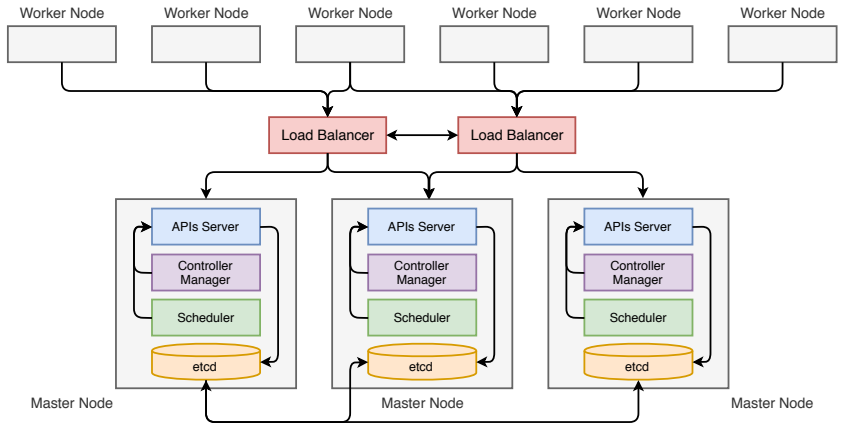


A general purpose (limited) web front end for the Kubernetes.

Additional components: Metric Server

- Provides metrics for Kubernetes
- Autoscaling based on CPU and RAM usage

High Avalaibility of Control Plane



Client: kubectl

Command Line for Kubernetes

```
$ kubectl version
```

The configuration file

```
$ cat .kube/config
```

tells the client where the APIs Server is and how to talk with it.

A pocket cheatsheet for `kubectl` can be found here.

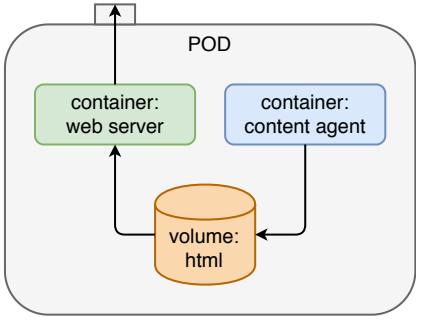
Namespaces

Logical partitions of the cluster:

- `user`: the private home namespace for a given user or group of users.
- `default`: the default namespace for any object without a namespace.
- `kube-system`: acts as the home for objects and resources created by Kubernetes itself.
- `kube-public`: readable by all users.

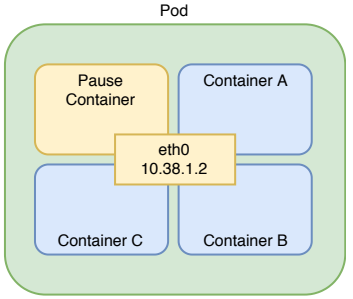
Running containers in Kubernetes

A Pod is a set of one or more tightly coupled containers



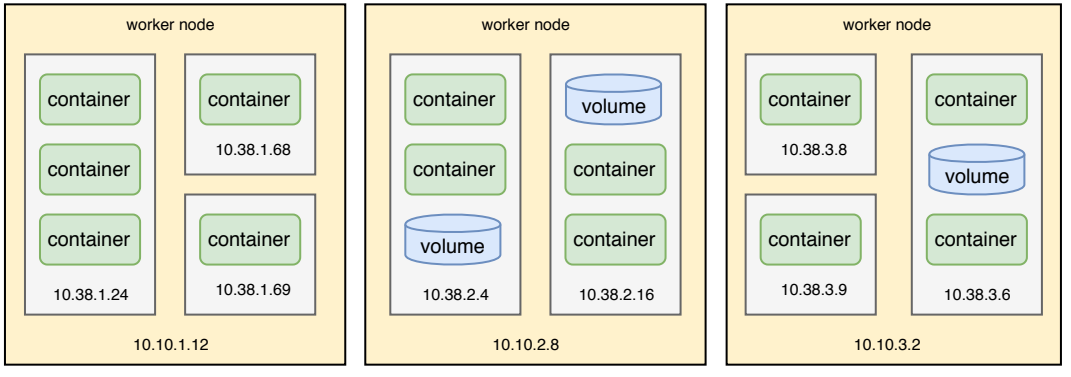
Containers in a Pod can share data through local volumes and the network interface.

The ghost Pause Container



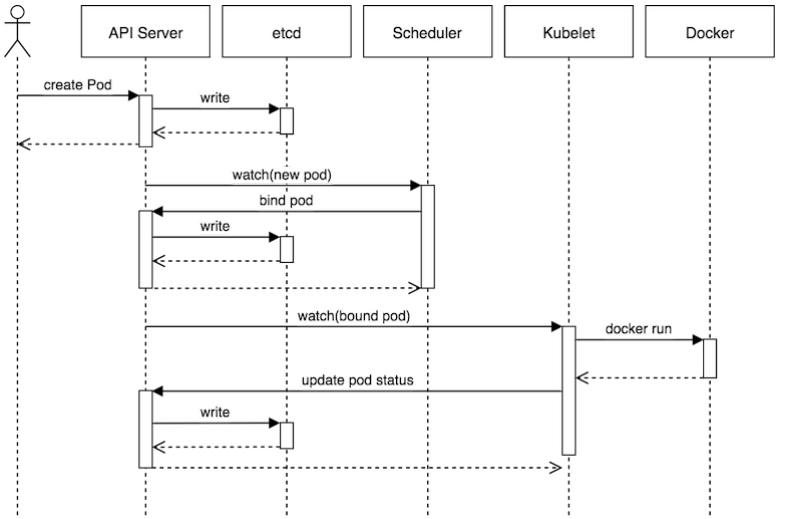
- Any pod comes with a 'Pause Container' sharing its network namespace with all the other containers in the pod.
- It contains a simple binary that goes to sleep and never wakes up. See the code here.
- When the other containers die and get restarted by kubelet, all the network setup, including the pod's IP address, will still be there because of the pause container.

Place pods on the worker nodes



Pods are the minimum units of placement in Kubernetes.

Behind the scenes



High scalable platform

As at time of writing, a single Kubernetes cluster supports:

- Up to 5,000 worker nodes
- Up to 150,000 pods
- Up to 300,000 containers

Federation of multiple clusters is possible (work in progress)

Health Check

- The kubelet agent constantly performs an health check on the containers in the pod.
- The `restartPolicy` property of the pod controls how kubelet behaves when a container exits:
  - `Always`: always restart an exited container (default)
  - `OnFailure`: restart the container only when exited with failure
  - `Never`: never restart the container

Liveness Probe

- When a container runs into some trouble conditions due to application code, it is still be considered running.
- To detect this kind of issues, the kubelet performs periodically a *Liveness Probe*
- The kubelet will restart the container if the probe fails.
- Instrument the application to expose a health check HTTP interface, e.g. `/healthz`.
- For no HTTP based applications, other checks are available:
  - TCP check
  - EXEC check
- A liveness probe can make your application more safe.

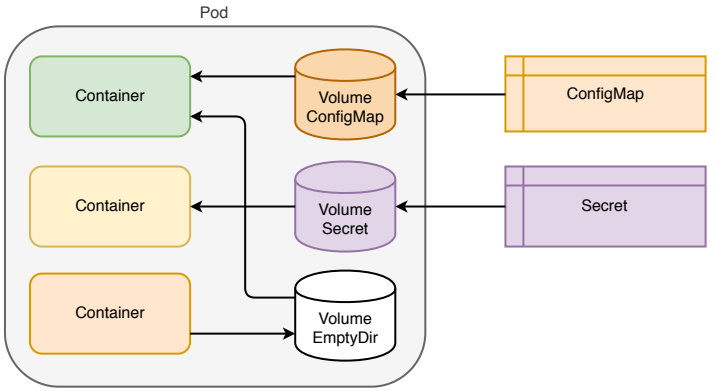
Readiness Probe

- A starting container may need time to load configuration, or connect a database before the first user's request can be served.
- To detect if an application is ready to serve user’s requests, the kubelet performs periodically a *Readiness Probe*
- The user's requests are sent to the application only if the readiness probe succeded.
- Instrument your application to expose a readiness HTTP interface, e.g. `/ready`.
- For no HTTP based applications, other checks are available:
  - TCP check
  - EXEC check
- A readiness probe can make your application more safe.

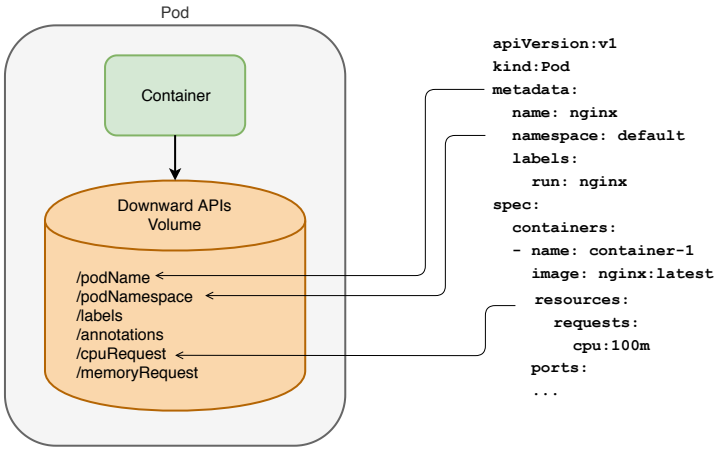
Life Cycle Hooks

- Processes running in containers, may require a start up procedure while other need a gentle and clean shutdown set of operations.
- If this is the case, define two additional life cycle hooks:
  - Post Start Hook: is executed after the container is created.
  - Pre Stop Hook: is executed immediately before a container is terminated.

Attaching Volumes to a Pod

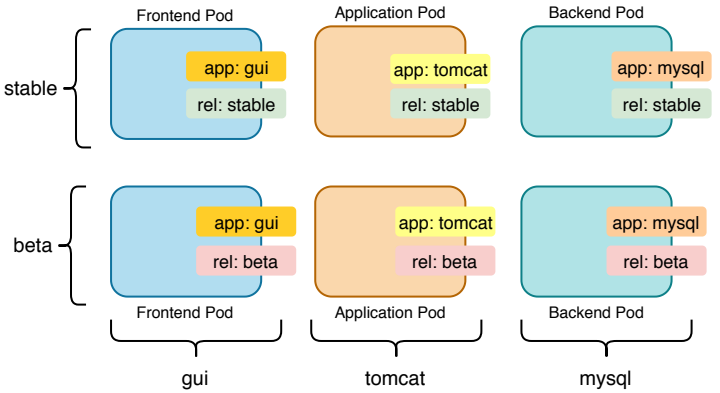


Downward APIs volume



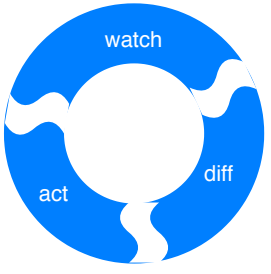

```
apiVersion:v1
kind:Pod
metadata:
  name: nginx
  namespace: default
  labels:
    run: nginx
spec:
  containers:
  - name: container-1
    image: nginx:latest
    resources:
      requests:
        cpu:100m
    ports:
    ...
```

Organizing pods with labels



Multi-dimensional labeling.

Controllers



An active reconciliation loop:

- Watch both the desired state and the actual state
- Compare the actual with the desired state
- Change the actual state if it diverges from the desired one

Embedded controllers

Running in the Controller Manager:

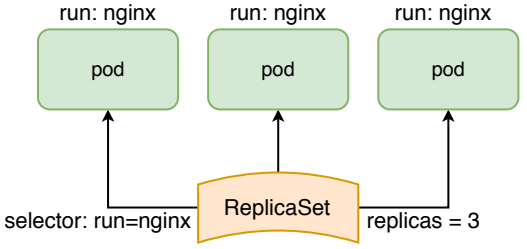
- Deployment
- ReplicaSet
- StetefulSet
- DaemonSet
- Node
- Service
- Endpoint
- …

Add-on controllers

Running as regular pods:

- Ingress Controller
- Network Controller
- Operators
- …

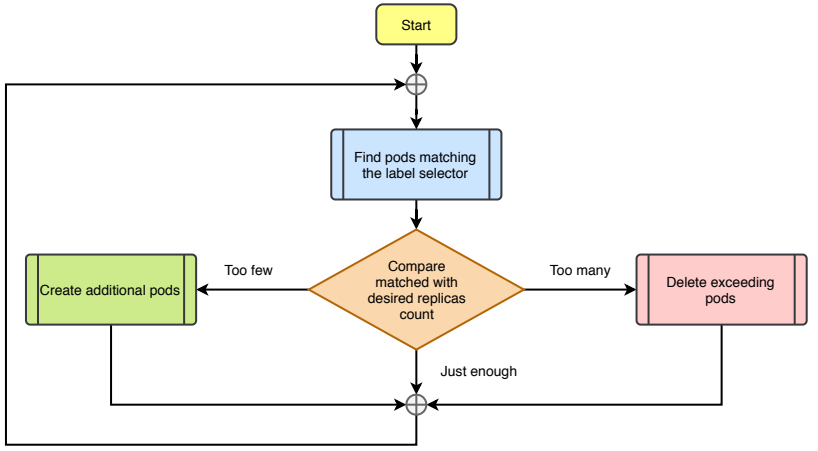
ReplicaSet



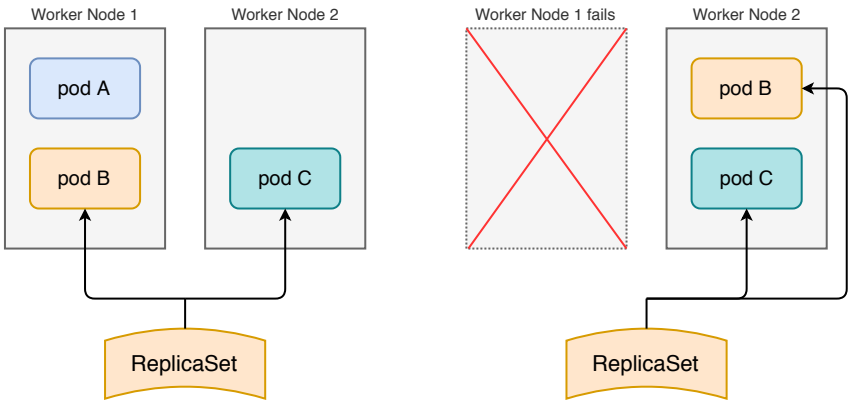
The three key parts of a ReplicaSet object:

- label selector
- replicas count
- pod template

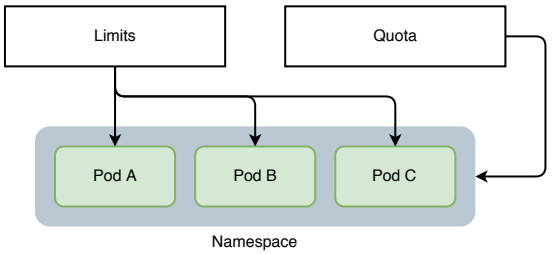
ReplicaSet Controller loop



Pod High Availability



Quotas and Limits

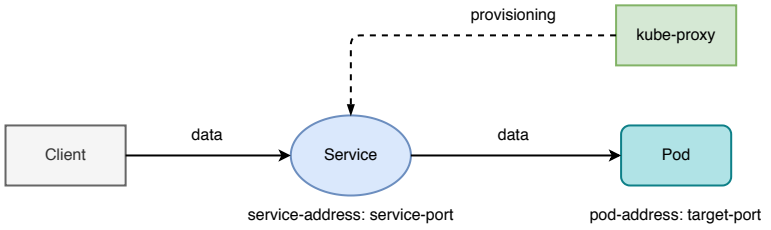


- Limits apply to individual pods.
- Quotas apply to all pods in the namespace.

Services

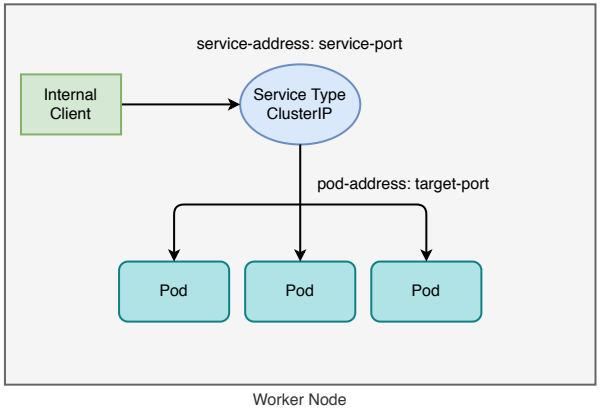
- Services enable clients to talk with pods.
- Service types:
  - ClusterIP
  - NodePort
  - LoadBalancer

Service provisioning



- Services are provisioned on the worker nodes by the kube-proxy:
  - userspace (deprecated)
  - iptables (current default)
  - ipvs (recently introduced)

ClusterIP service acts as proxy and load balancer(*) between clients and pods



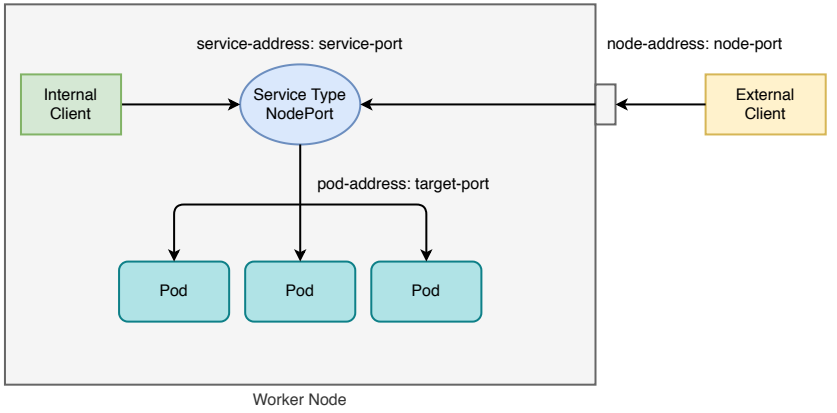
(*)IPtables do not implement a real Load Balancer!

When using the kube-proxy in iptable mode, the service is not implementing a real Load Balancer:

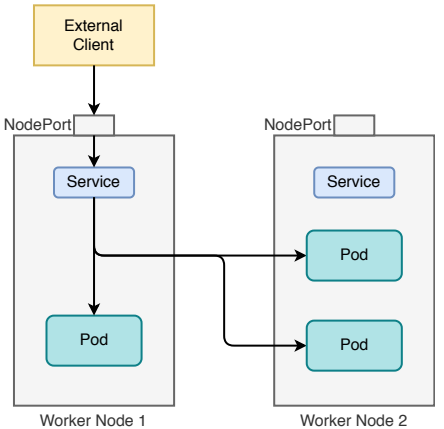
- The iptable are designed for firewalling and not for load balancing.
- However, the kube-proxy could craft a smart set of rules that could make iptables behave similar to a Load Balancer.
- If you have three pods, the kube-proxy writes the following rules:
  - select pod 1 as the destination with a probability of 0.33
  - select pod 2 as the destination with a probability of 0.33
  - select pod 3 as the destination with a probability of 0.33
- Since this is a probability, there's no guarantee that pod 2 is selected after pod 1 as the destination.

Use the kube-proxy in ipvs mode if you need for a real and performant Load Balancer between pods.

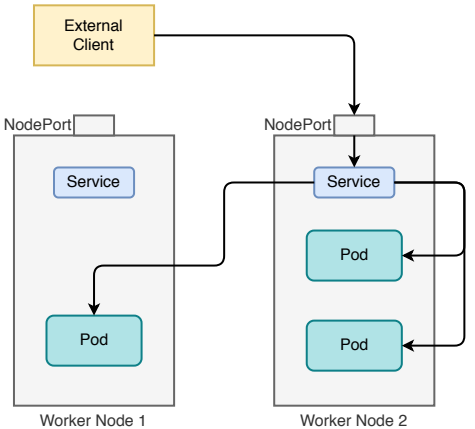
NodePort service maps a random node port with the service port



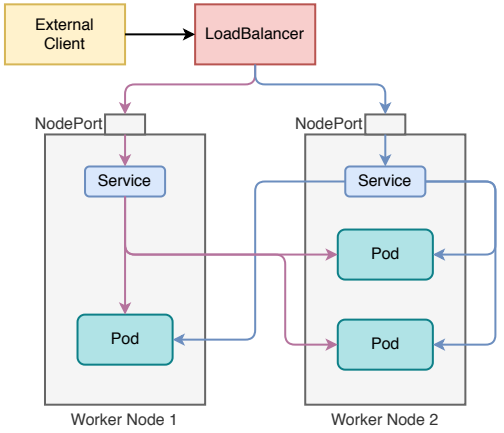
The NodePort is opened on all the worker nodes



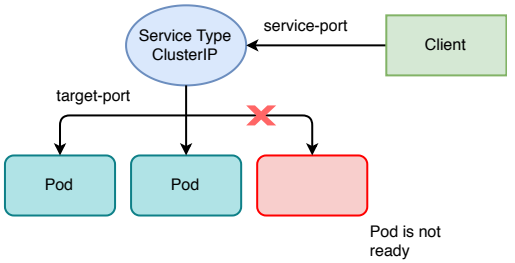
The NodePort is opened on all the worker nodes



LoadBalancer service dynamically creates a Load Balancer (Cloud Providers only)

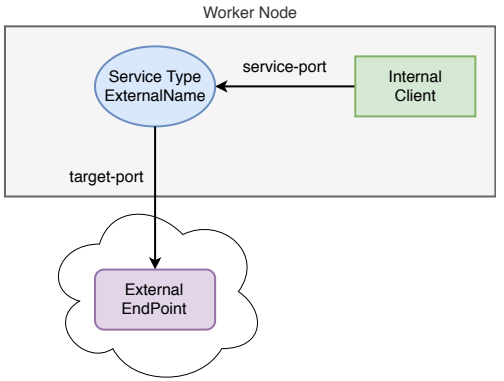


Service Endpoints



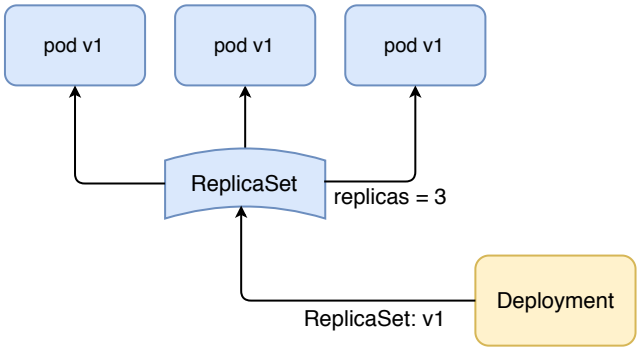
- Pairs of pod IP address and pod listening ports are called *Service endpoints*.
- A pod whose Readiness Probe fails is removed as service endpoint.
- Service endpoints are dynamically updated.

ExternalName service references endpoints outside the cluster



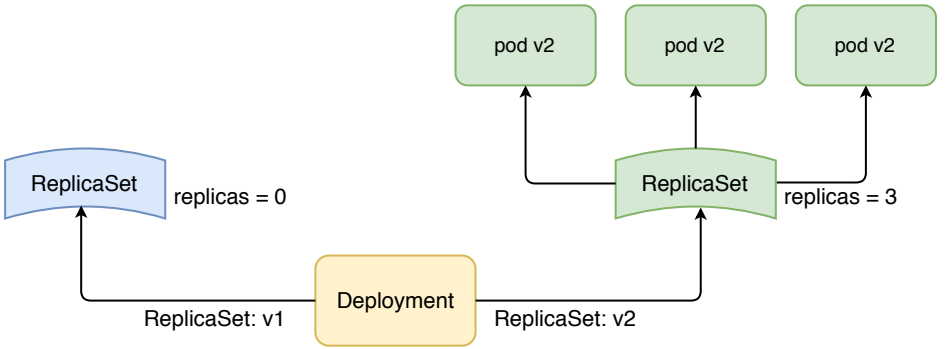
Use it when you need to consume external applications (e.g. traditional databases) from a pod

Application Deployment



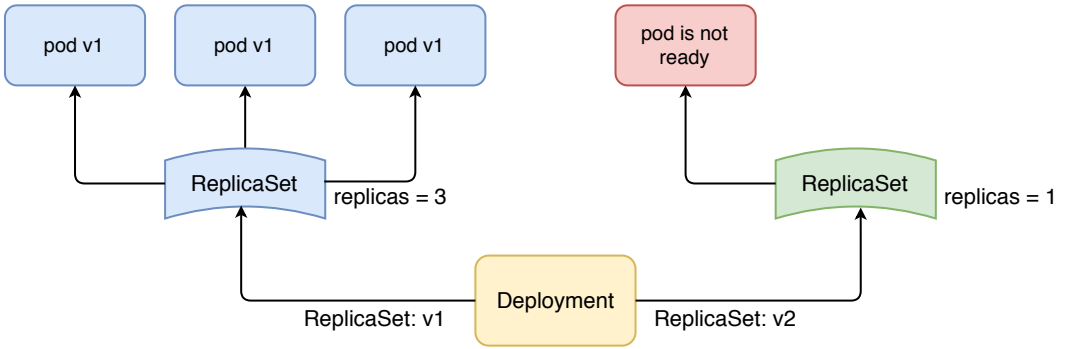
A deployment to deploy an application.

Application Update



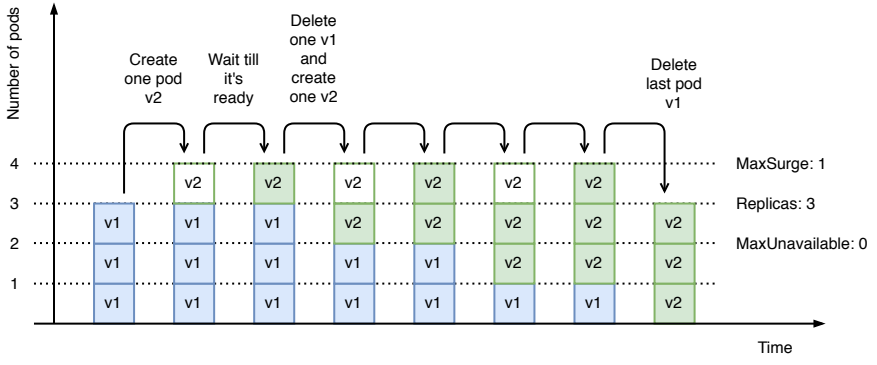
A deployment to perform an application update.

Failed Application Update



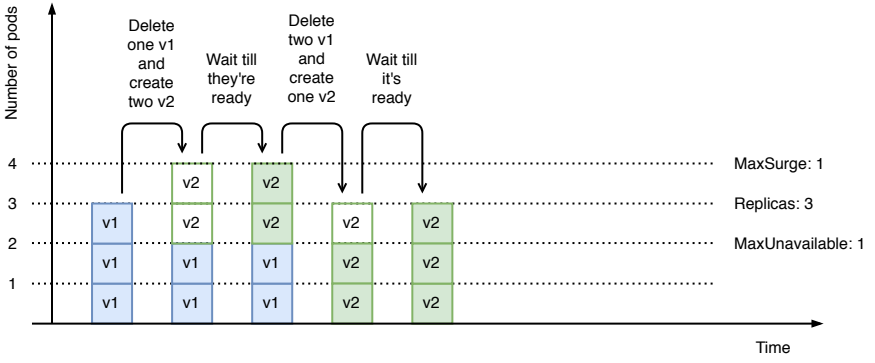
An application update blocked by a failing readiness probe.

Rolling Update



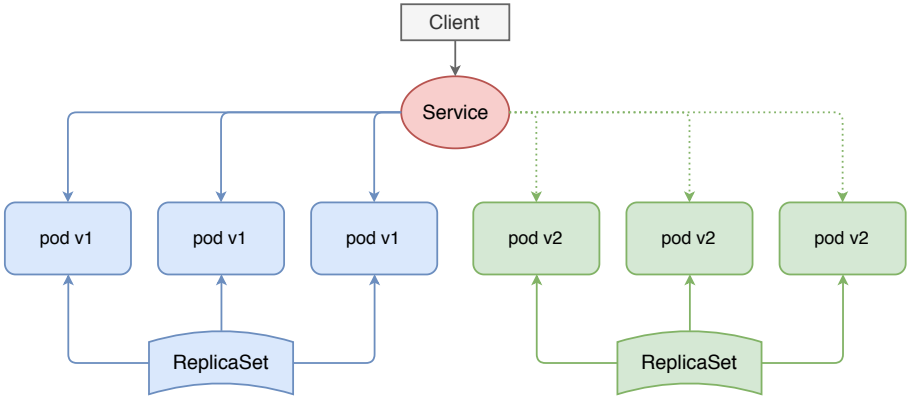
Rolling update with maxSurge=1 and maxUnavailable=0.

Rolling Update

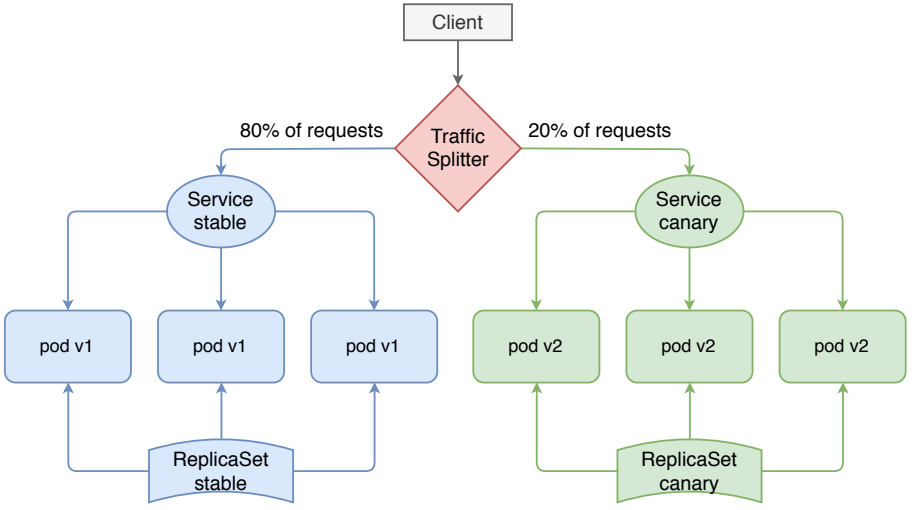


Rolling update with maxSurge=1 and maxUnavailable=1.

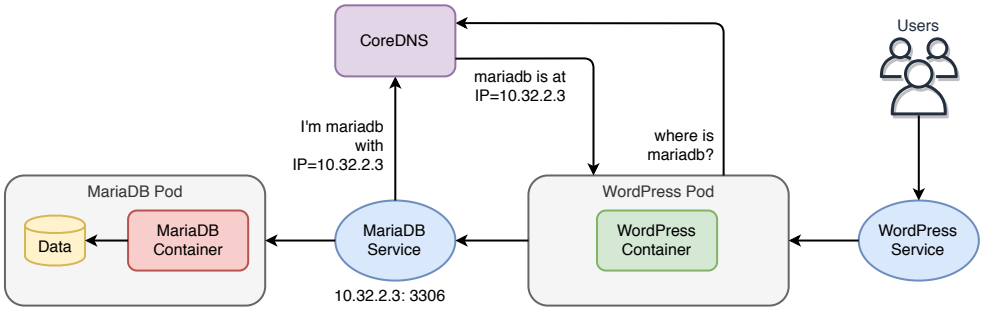
Blue/Green Update



Canary Update



Services discovery



Core DNS Service:

- Services registers themself to the Core DNS.
- Pods query the Core DNS for service discovery.

Module 3

Advanced Kubernetes

CNI: Container Network Interface

- Kubernetes networking is implemented via the Container Network Interface (CNI).
- CNI works as an interface between the container runtime and Kubernetes.
- Plugin model for multiple options:
  - Community implementations: Bridge, Router, Calico, Flannel, Open Virtual Switch, Multus, Romana, ...
  - Networking Vendor implementations

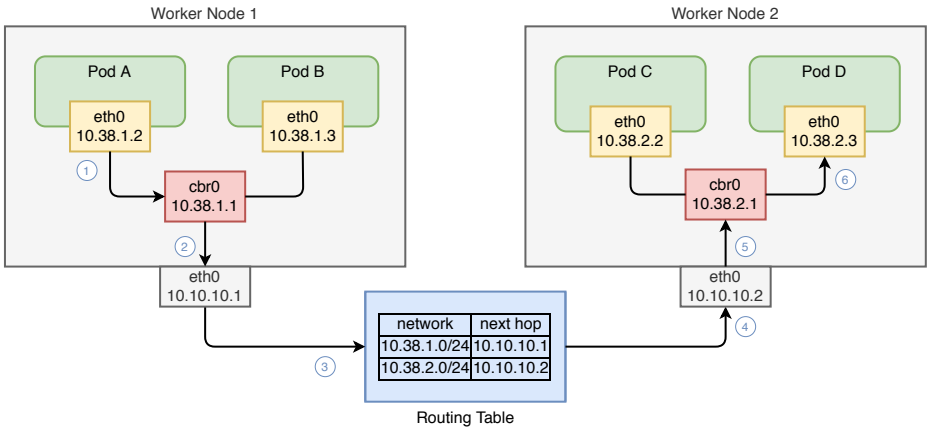
Kubernetes Network Model

- Pods Networks: a cluster-wide network and subnetworks used for pod-to-pod communication managed by the CNI plugin.
- Services Network: a cluster-wide range of virtual IPs for pod-to-service, external-to-service and service-to-pod communication.
- Hosts Network: a cluster wide network for host-to-host communication.

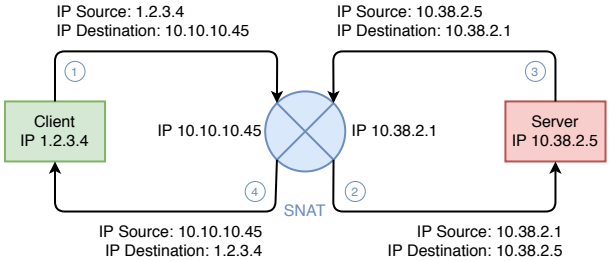
Pods Networks

- All containers within a pod can communicate with each other unimpeded.
- All pods can communicate with other pods without NAT.
- All nodes can communicate with pods without NAT.
- The IP that a pod sees itself as is the same IP that others see it as.

Pod-to-pod communication

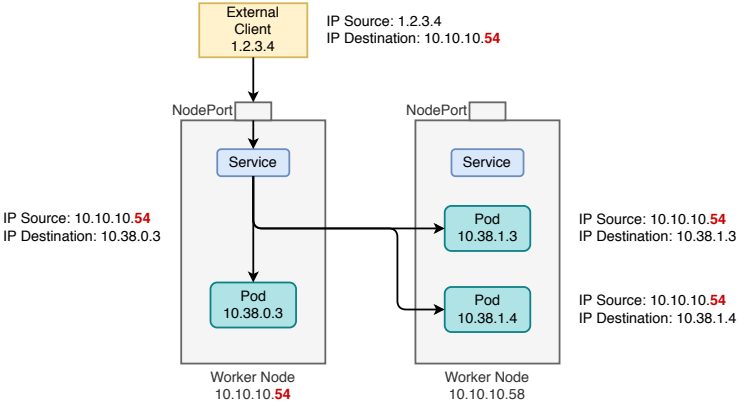


Source NAT



- With SNAT, the source IP address of client's request is lost.
- Sometimes, this isn't desidered (e.g. keep track of source IPs accessing a service)
- NodePort and LoadBalancer services do SNAT (and then the source IP address is lost)

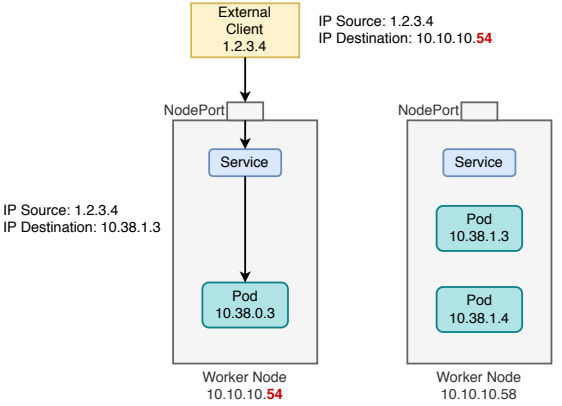
NodePort with externalTrafficPolicy: Cluster (default value)



Traffic is balanced regardless the worker node where pods are running.

Packets entering the worker node are SNATed and client source IP is lost.

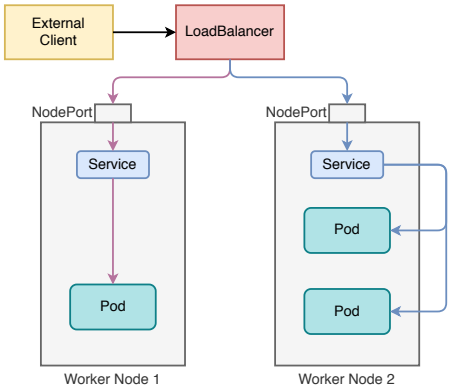
NodePort with externalTrafficPolicy: Local



Traffic is sent only to the single worker node and then no balanced.

Packets entering the worker node are not SNATed and client source IP is preserved.

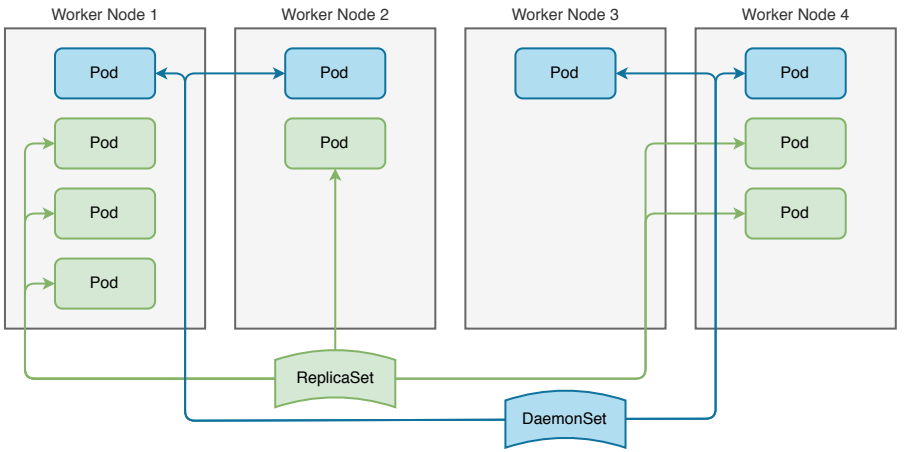
LoadBalancer with externalTrafficPolicy: Local



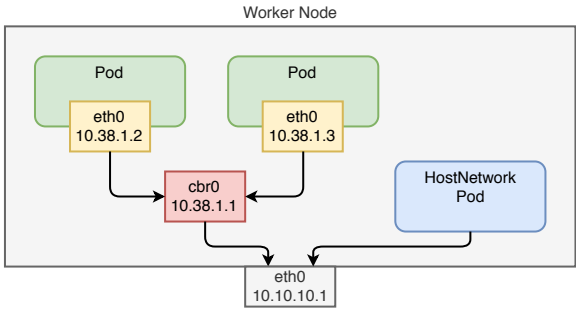
Traffic is sent only to the single worker node (balanced by the Load Balancer).

Packets entering the worker nodes are not SNATed and then the client source IP is preserved.

DaemonSet

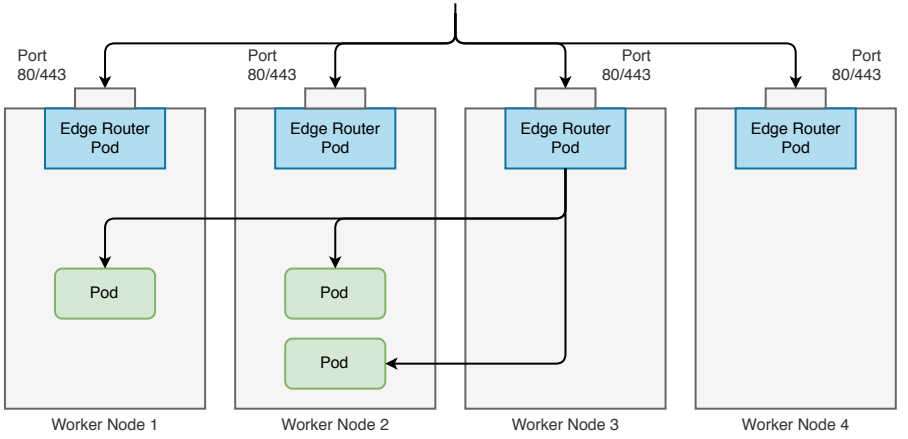

Worker Node 1
Worker Node 2
Worker Node 3
Worker Node 4
Pod
Pod
Pod
Pod
Pod
Pod
Pod
Pod
Pod
ReplicaSet
DaemonSet

HostNetwork Pods



Pods using the host's network namespace instead of their own.

Expose pods with a Reverse Proxy



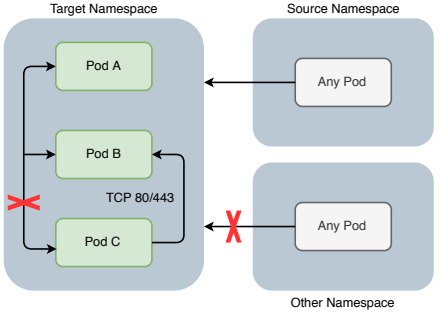
Discovering pods using a *headless* service

Need to populate the reverse proxy routing table in a dynamic fashion:

- Discover the pods as endpoints of a headless service.
- A headless service is a `ClusterIP` Service Type without a `ClusterIP` 😉
- We do not need for the `ClusterIP` because we want only the endpoints.
- Having a `ClusterIP` service type, the endpoints are discovered automatically.
- The APIs server notifies the endpoints to the reverse proxy.
- The reverse proxy updates its routing table according to.

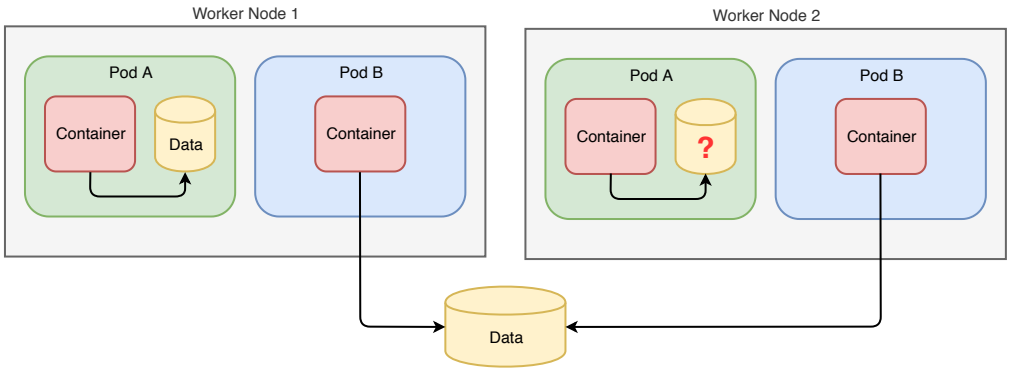
The reverse proxy uses only pod-2-pod communication without the usage of `iptables`.

Network Policy: restrict access to the pods



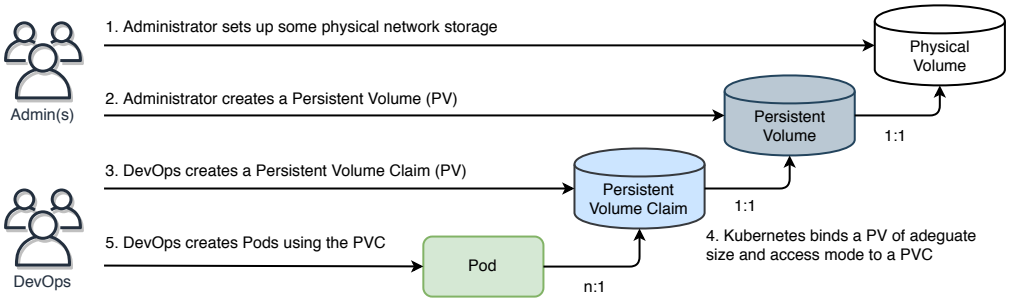
- Deny traffic from other namespaces while allowing all traffic from the namespaces the Pod is living in.
- Allow inbound traffic from only certain Pods within the same namespace.
- Define ingress rules for specific port numbers of an application.
- Check your preferred network plugin (CNI) documentation to see if it supports the network policies.

Persistent Volumes



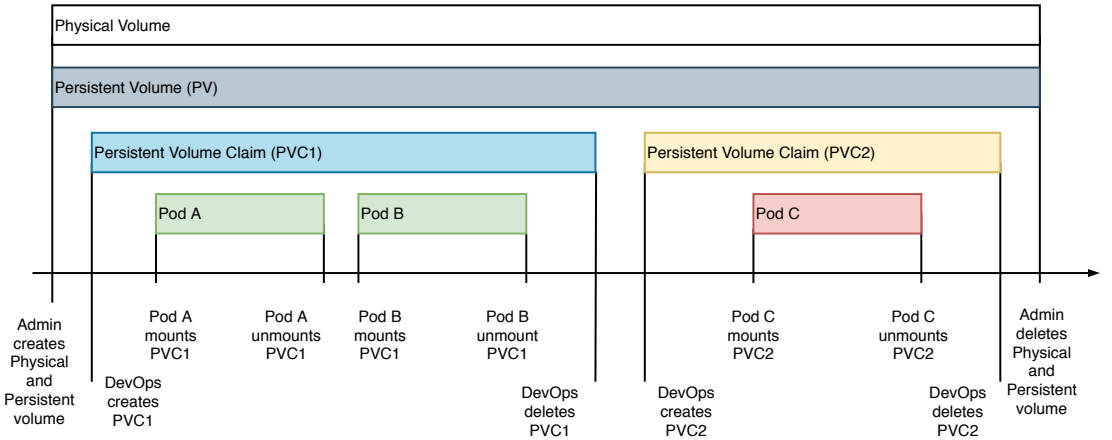
Persist data across pod replacement.

Manual Storage Provisioning



PVs are provisioned by cluster admins and consumed by pods through PVCs.

Persistent Volume Lifecycle



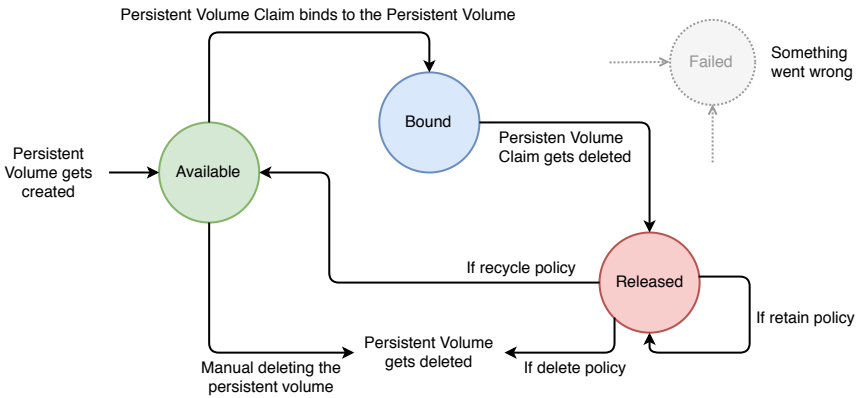
Volume Access Modes

- **Read Write Many** [RWX]: the volume can be mounted as read and write by multiple nodes at time
- **Read Only Many** [ROM]: the volume can be mounted as read only by multiple nodes at time
- **Read Write Once** [RWO]: the volume can be mounted as read and write by a single node at time

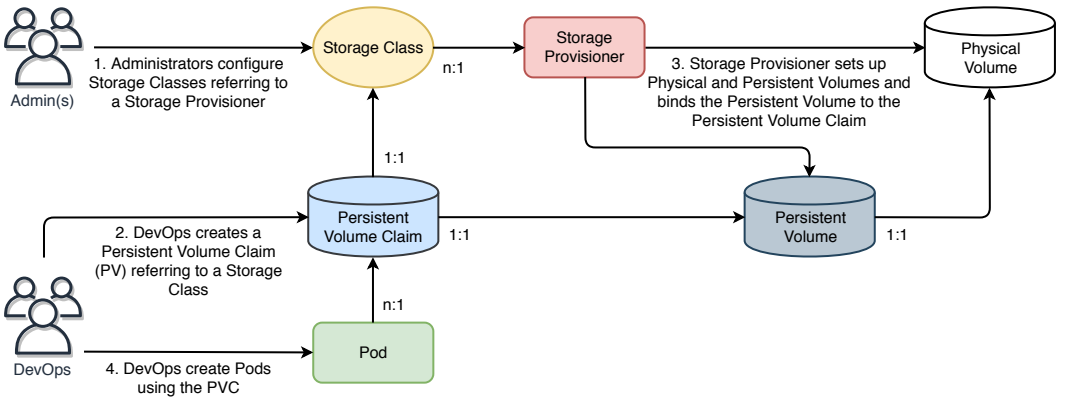
Persistent Volume Reclaim Policy

- **Retain**: the content of the physical volume still exists when the Peristent Volume is unbound
- **Recycle**: the content of the physical volume is deleted when the Persistent Volume is unbound
- **Delete**: both the content and the physical volume are deleted when the Persistent Volume is unbound

Persistent Volume State transitions

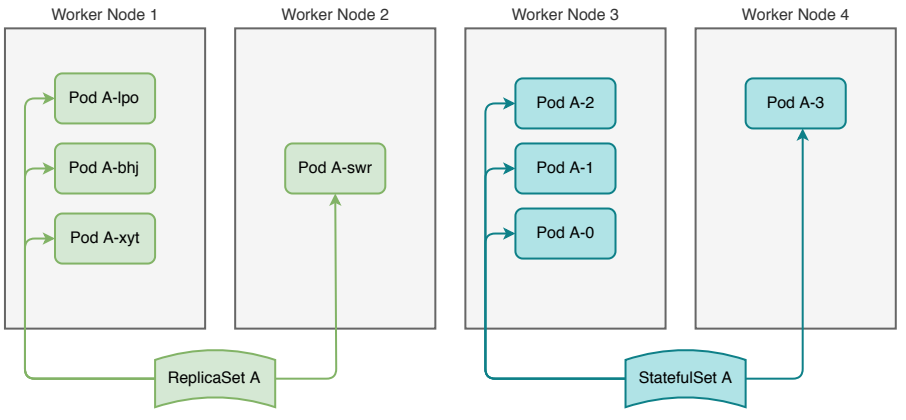


Dynamic Storage Provisioning



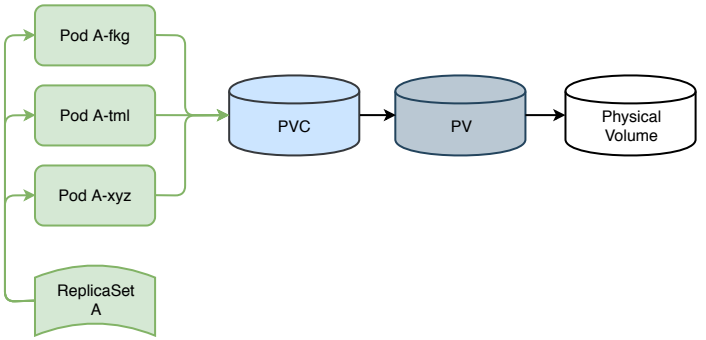
PVs are provisioned on request by the Storage Provisioner and consumed by pods through PVCs.

StatefulSet



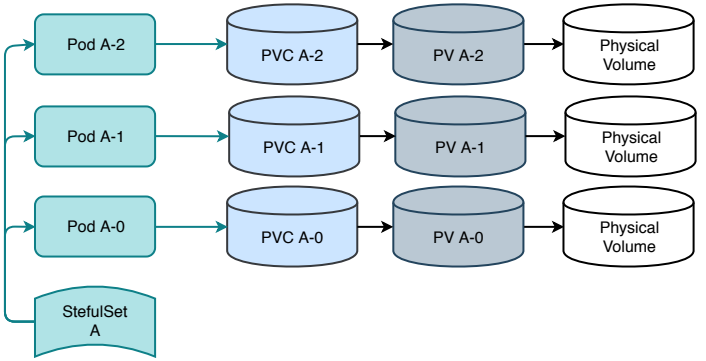
StatefulSet's pods have predictable names while ReplicaSet's pods have not.

Scaling Up ReplicaSet



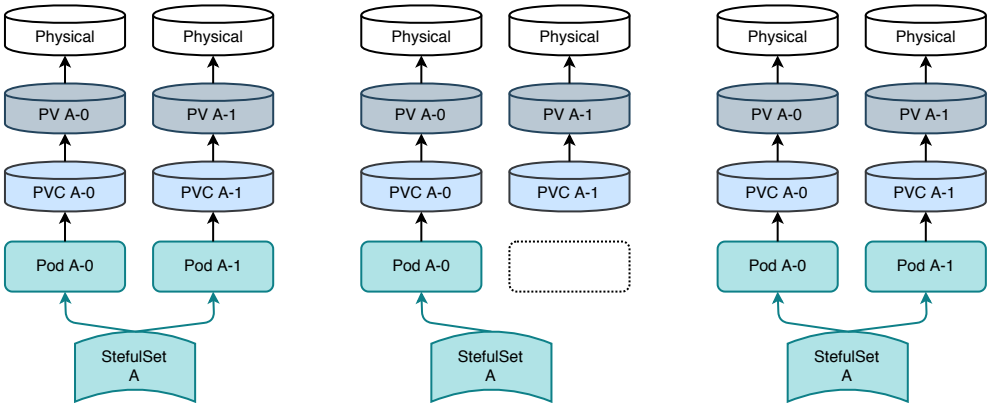
Scaling up a ReplicaSet does not scale PVCs.

Scaling Up StatefulSet



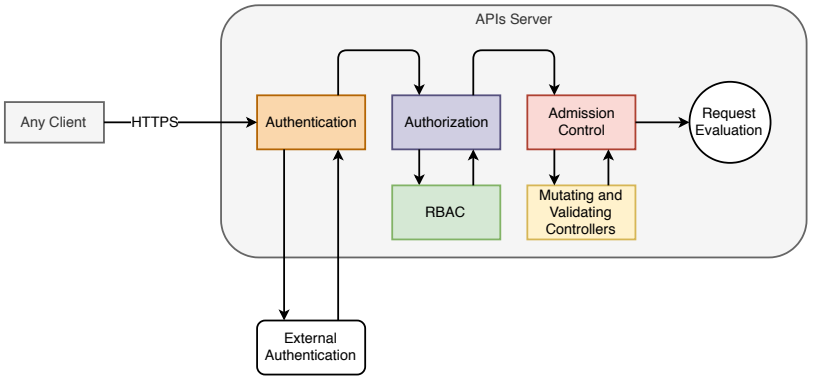
Scaling up a StatefulSet does scale PVCs too.

Scaling Down StatefulSet



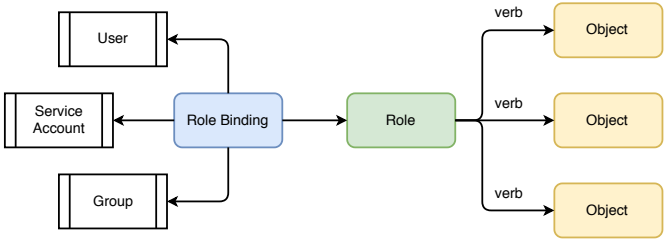
Scaling down a StatefulSet does not delete PVCs.

Authentication, Authorization, and Admission Control



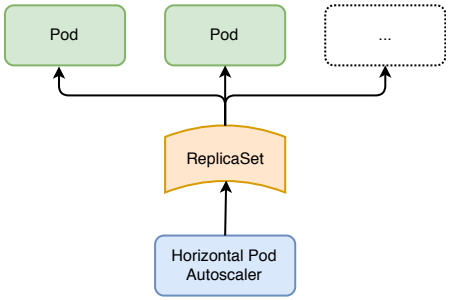
Any request sent to the APIs Server must pass security gates.

RBAC: Role Based Access Control



- At cluster level:
  - Cluster Roles
  - Cluster Role Bindings
- At namespace level:
  - Roles
  - Role Bindings

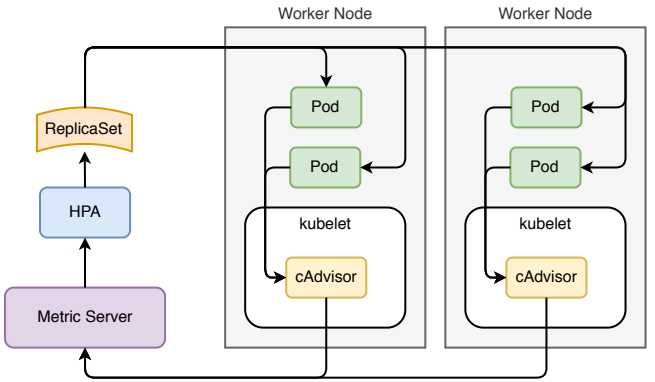
Horizontal Pod Autoscaler



Pods dynamic autoscaling based on metric

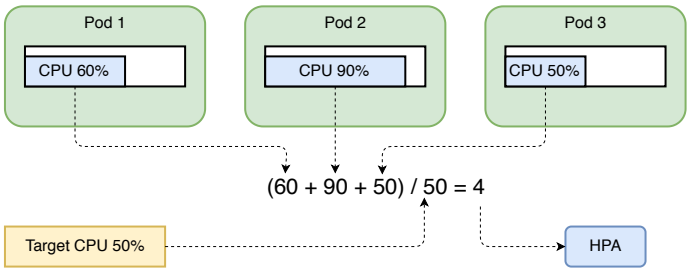
- Minimum and maximum number of replicas
- Resource to scale: ReplicaSet, StatefulSet, Deployment
- Metrics

Metric Server



- An autoscaler obtains metrics from the Metric Server and rescales the target resource.
- Pod's metrics are collected from cAdvisor and sent to the Metric Server.

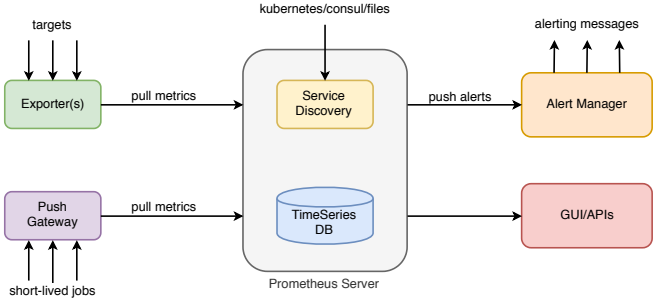
Autoscaling based on CPU



Monitoring with Prometheus

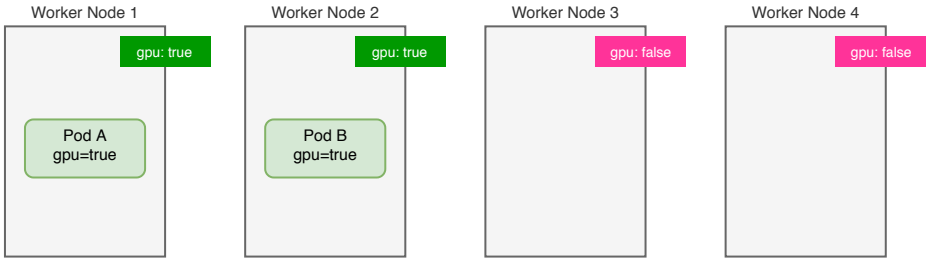
- **Prometheus** is an open source monitoring system well suited for highly dynamic service-oriented systems.
- It uses a multi-dimensional data model with time series data identified by a metric name and key/value pairs.
- **PromQL** is a flexible query language used by Prometheus.
- Time series collection happens via a **pull** model over HTTP(S).
- Pushing time series is supported via an intermediary gateway.
- Monitored targets are discovered via service discovery or static configuration.
- Advanced graphing and dashboarding is supported via **Grafana**.
- No reliance on cluster distributed storage.

Prometheus Architecture



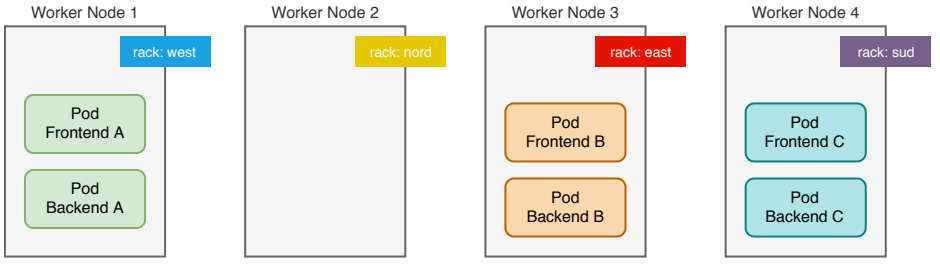
- Prometheus discovers targets either dinamically from Kubernetes or statically from config files.
- It scrapes metrics from targets via exporters or from short-lived jobs via Push Gateway.
- It stores all scraped samples locally on an internal TimeSeriesDB.
- It runs rules over this data via PromQL language to aggregate, record, and generate alerts.
- Alerts are dispatched via the Alert Manager.
- GUI and other APIs consumers can be used to visualize and manipulate the collected data.

Advanced Scheduling: Node Affinity



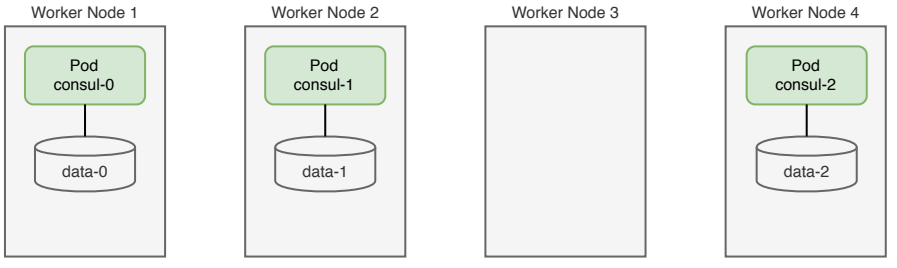
Node Affinity with required nodes where to schedule the pods.

Advanced Scheduling: Pod Affinity



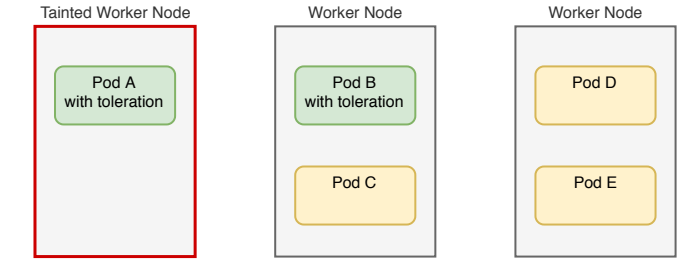
Colocation with Pod Affinity.

Advanced Scheduling: Pod AntiAffinity



Redundancy with Pod AntiAffinity.

Advanced Scheduling: Taints and tolerations



Taint: node-role/infra:NoSchedule

- Pods are not assigned to a tainted node unless they have a toleration for that taint.
- Tolerations does not exclude the pods to be assigned to nodes without taints.

Helm Package Manager

Deploy complex applications through a lot of yaml files can be a daunting task.

Helm is a tool for application management based on charts.

A Helm chart:

- describes a complex application
- provides repeatable application installation, upgrade, and remove
- rolls back to an older version of application release
- sets application parameters through variables

Module 4

Applications Design Patterns

Cloud Native Applications

Common motivations behind moving to cloud native application architectures:

- Agility
- Safety:
  - Visibility
  - Fault isolation
  - Fault tolerance
  - Automated recovery
- Scaling
- Automation

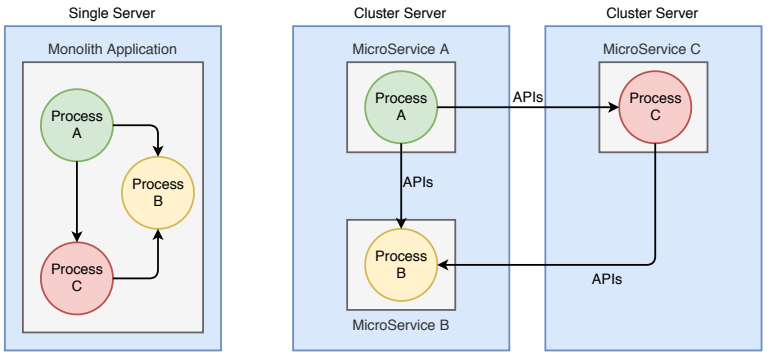
Defining Cloud Native Architectures

- New way to design software applications: 12Factor
- It describes an application archetype that optimizes for:
  - agility
  - safety
  - scaling
  - automation
- Applications that can be “throw away” with very little cost:
  - the "Pet vs Cattle" methafor
- The application environment is full disposable:
  - state is extracted to some backing service
- Scale up and down in a very simple and elastic manner that is easily automated

Microservices

- Turn monolithic systems into independently deployable lightweight units:
  - *"Do only one thing and do it very well"* - Do you remember the '70 UNIX philosophy?
- Units communicate each other over well defined APIs:
  - loosely coupled components
- Each unit usually represents a business capability:
  - smallest atomic service that delivers a business value

Migrate to Microservices



Turn monolithic systems into independently deployable lightweight units.

Benefits of Microservices

Microservice architectures meet Cloud Native Applications requirements in several ways:

- Agility:
  - decouple the associated development cycles
  - smaller teams, the "Two Pizzas Team" rule
  - technology mistakes are more recovereable
- Safety:
  - faults can be isolated quickly
  - finest grained monitoring
- Scaling:
  - each component can be scaled independently from others
- Automation:
  - declarative deployments
  - self healing

Challenges for Cloud Native Application adoption

- Cultural changes:
  - From Releases Management to Continous Delivery
- Organizational changes:
  - From Silos to DevOps
- Technical challenges:
  - Decomposing Monoliths
  - Decomposing Data
  - Containerization
  - Orchestration

Decomposing Monoliths

Traditional n-tier, monolithic enterprise applications rarely operate well when deployed to cloud infrastructure as they make assumptions Cloud Infrastructures simply cannot provide:

- Access to mounted, shared filesystems
- Peer-to-peer application server clustering
- Shared libraries
- Configuration files sitting in well-known locations
- Long-lived infrastructure

Patterns for decomposing Monoliths

- Build new features as Microservices:
  - Still have a backlog of features? Stop adding new code to the monolith! All new features will be built as microservices.
- Wrap the Monolith:
  - Create an APIs interface around the monolith that make it looks like a microservice.
- Strangle the Monolith:
  - Gradually create a new system around the edges of the old, letting it grow slowly until the old system is strangled.
- Any combination of the above.

Decomposing Data

- It’s not enough to decompose monolithic applications into microservices.
- Data models must also be decoupled.
- Database per service pattern:
  - each microservice encapsulates, governs, and protects its own domain model and persistent data store.
- However, data must often be recomposed in order to ask cross microservices questions.

A Taxonomy for Patterns

- Foundational Patterns
- Configuration Patterns
- Structural Patterns
- Behavorial Patterns

Foundational Patterns

- Designing containers
- Predictable demands
- Observability
- Life Cycle Conformance
- Security context

Build the smallest image possible

- Minimize number of layers.
- Remove unnecessary tools.
- Use the smallest base image possible:
  - scratch
  - alpine
  - distroless
- Optimize for cache
- Use Multi-Staged builds

Tag images using semantic versioning

- The Semantic Versioning Specification provides a clean way of handling software release version numbers.
- In this system, software has a three-part version number: **X.Y.Z**, where:
  - X is the major version, incremented only for incompatible API changes.
  - Y is the minor version, incremented for new features.
  - Z is the patch version, incremented for bug fixes.
- Every increment in the minor or patch version number must be for a backward-compatible change.
- Use the **latest** tag only for the most recent image. This tag is moved as soon as a new image is created.

Use the native logging mechanisms of containers

- On a traditional server, applications write logs to a specific file, e.g. `/var/logs/apache.logs` and handle log rotation to avoid filling up the disks.
- Containers offer an easy and standardized way to handle logs:
  - the application writes logs on *stdout* and *stderr*.
  - the container runtime captures these logs.
  - logs can be accessed by using the `docker logs` or `kubectl logs` commands.
  - logs are stored in the `/var/lib/docker/containers/id/id-json.log` file in JSON format.
- Using Kubernetes with an advanced logging system, e.g. Fluentd you might forward those logs to a database in order to centralize them.

Ensure that containers are immutable

- Avoid to treat containers as traditional servers:
  - update application inside of a running container
  - patch a running container when vulnerabilities arise
  - change configuration files in a running container
- Immutability makes deployments safer and more repeatable.

Ensure that containers are stateless (as much as you can)

- Stateless means that any state (persistent data of any kind) is stored outside of a container.
- This external storage can take several forms, depending on the needs:
  - To store files, use an external Object Store service, such as Minio.
  - To store information such as user sessions, use an external key-value store.
  - For databases, use an external disk attached to the container.
- By using these options, you remove the data from the container itself, meaning that the container can be cleanly shut down and destroyed at any time without fear of data loss.
- If a new container is created to replace the old one, just connect the new container to the same datastore.

Be smart with state

Stateless components are easy to:

- Scale:
  - to scale up, just add more copies
  - to scale down, instruct instances to terminate once they have completed their current task.
- Repair:
  - to repair a failed instance, simply terminate it and spin up a replacement.
- Roll-back:
  - terminate the bad one and launch instances of the old version instead.
- Load-Balance:
  - balancing across staless instances is easy since any instance can handle any request
  - balancing across stateful instances is much harder, since the state of the user's session forces only one instance to handle all requests from a given user.

Specify resources consumption

- Containers should have enough resources to actually run:
  - running a large application on a node with limited resources, it is possible for the node to run out of memory or CPU.
- Containers should limit resources they actually need:
  - avoid to spinning up more replicas to artificially decrease latency instead of make your code more efficient.
  - a bug can cause an application to go out of control and use 100% of the available CPU.

Observability

- To facilitate its management in production, an application must communicate its state to the overall system:
  - Is the application running?
  - Is it healthy?
  - Is it ready to receive traffic?
  - How is it behaving?
- Kubernetes implements two probes:
  - Liveness Probe
  - Readiness Probe
- In addition, you should instrument the application with metrics endpoints

Metrics endpoints

- Monitoring is an integral part of application management.
- In many ways, monitoring containerized applications follows the same principles that apply to monitoring of non-containerized applications.
- However, because containerized infrastructures tend to be highly dynamic, you cannot afford to reconfigure your monitoring system all the times.
- A popular option is Prometheus, a system that automatically discovers the endpoints it has to monitor.
- Prometheus scrapes the containers for metrics in a specific format for them:
  - Instrument your application to expose metric web interface, e.g. `/metrics`

Levels of isolation in Kubernetes

Kubernetes has several nested layers, each of which provides some level of isolation and security:

- Containers
- Pods
- Nodes
- Namespaces
- Cluster
- Infrastructure

Principle of least privilege

- Give a user or service account only those privileges which are essential to perform its intended function.
  - a user for the sole purpose of creating backups does not need to install software
  - an application serving web pages does not need to write files

Container Security context

- Specify the user ID under which the process in the container will run.
- Prevent the container from running as root.
- Give the container full access to the worker node kernel (be careful!).
- Configure fine grained privileges by adding or dropping capabilities to the container.
- Prevent the process inside the container from writing to the filesystem.

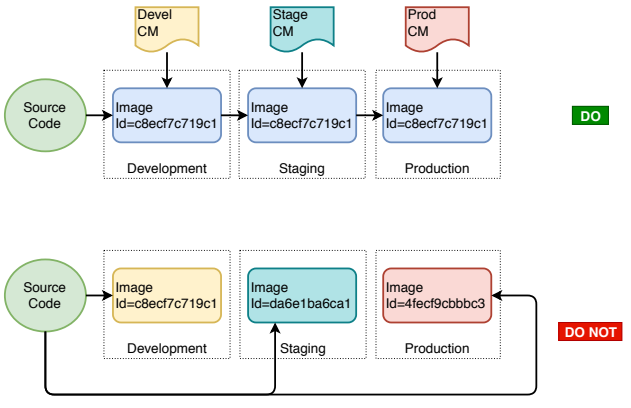
Configuration Patterns

- Configuration Maps
- Configuration Harvesting

Separate configuration from code

- Configuration is everything that is likely to vary between different deployments:
  - Development
  - Test
  - Staging
  - Production
- Keep a strict separation of configuration from code.
- Instead, pass configurations as:
  - Environment Variables
  - Configuration Maps
  - Secrets

Promote the same image to different environments



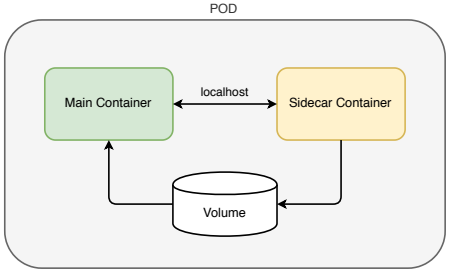
Configuration Harvesting

- Harvest configuration from the environment
  - From Downward APIs volume
  - From APIs Server

Structural Patterns

- Sidecar
- Adapter
- Initialiser
- Ambassador

The Sidecar Pattern

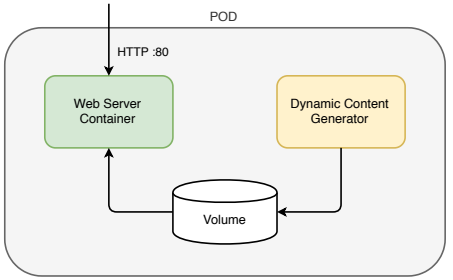


- In the Sidecar Pattern, we have multiple containers:
  - The main container: implements the core logic for the application
  - One or more supporting containers: augment and improve the application container
- Containers are part of the same Kubernetes Pod.
- Containers can share local volumes and communicate through localhost.

Common use cases for Sidecar Pattern

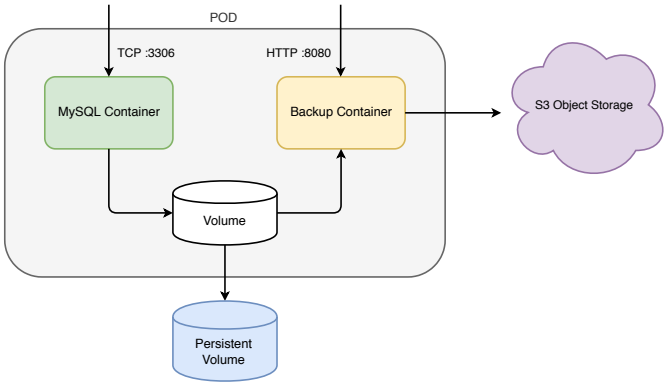
- Augmenting the application
- Implementing modularity: a.k.a *"Do only one thing and do it very well!"*
- Adding dynamic content
- Dynamic Configuration

Adding dynamic content with a Sidecar



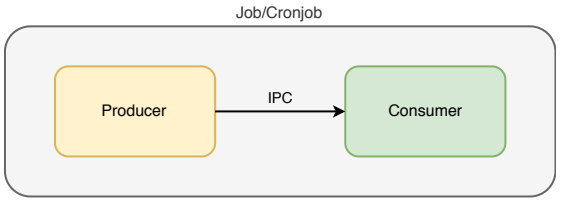
Use a sidecar to dynamically create web pages

Augmenting the application with a Sidecar



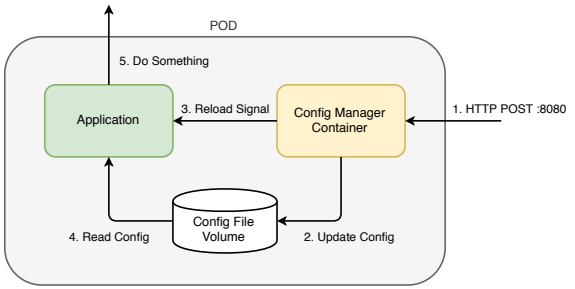
Use a sidecar to backup MySQL to Cloud Object Storage

Implementing modularity with Sidecar



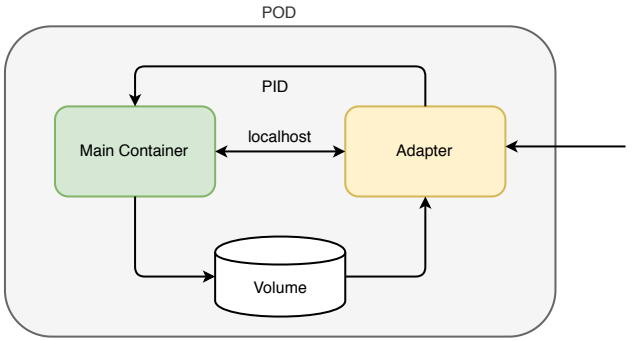
Use sidecar to implement modularity

Dynamic Configuration with Sidecar



Use sidecar to dynamically update configuration

The Adapter Pattern



- In the Adapter Pattern, we have two containers:
  - The main container: implements the core logic for the application
  - An adapter container: implements additional interfaces
- Containers are part of the same Kubernetes Pod.
- Containers can share local volumes and communicate through localhost or inter process communication.

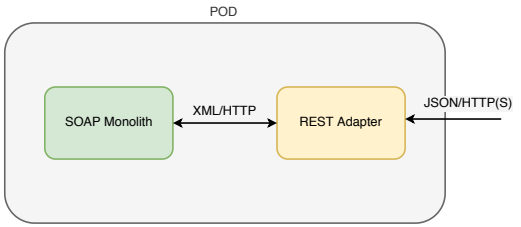
Common use cases for the Adapter Pattern

- Adapting an existing legacy application to Cloud Native Infrastructures
- Instrumenting and introspecting the application for:
  - logging
  - monitoring

Why not simply modify the application container itself?

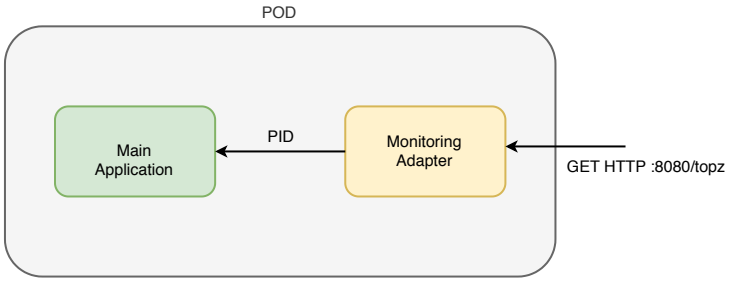
- If you're the developer responsible for the application container, then this might actually be a good solution.
- If you're reusing a container produced by another party, deriving a slightly modified image is significantly more expensive: patch, rebase, etc.
- If you're transitioning from Monolith to a Microservices architecture, then it may actually not be a great move.
- Additionally, decoupling the adapter into its own container allows for sharing and reuse.

Adapting an existing legacy application



Add an Adapter to wrap the monolith

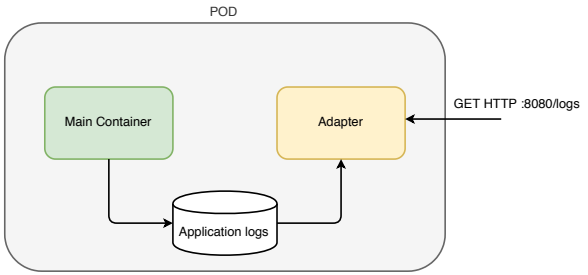
Instrument for Monitoring



Use an Adapter to instrument the main application for monitoring:

- HTTP GET `/topz` interface that provides a readout of resource usage, as in the `top` command
- Credits: brendandburns.

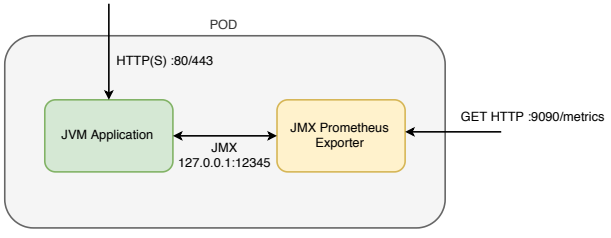
Instrument for Logging



Use an Adapter to expose a logging interface:

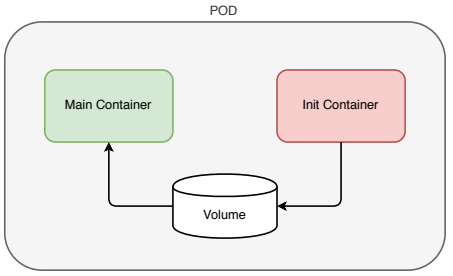
- Expose a HTTP `/logs` interface for applications not logging on `stdout` and `stderr`.
- Standardize logs format from different applications.

Prometheus Adapters



- Many Java applications expose metrics using JMX
- Rather than rewriting an application to expose metrics in the Prometheus format, use the Prometheus jmx_exporter.
- The `jmx_exporter gathers` metrics from an application through JMX and exposes them through a `/metrics` readonly endpoint that Prometheus can read.
- Limit the exposure of the read write JMX endpoint.

The Initializer Pattern

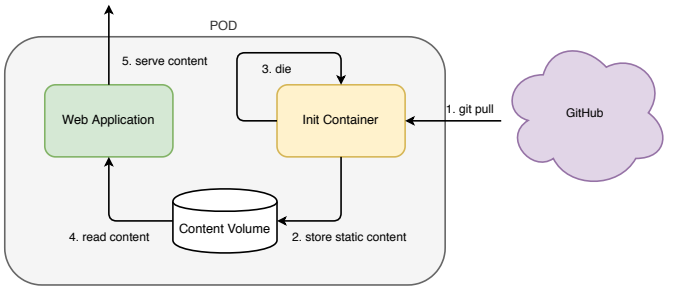


- In the Initializer Pattern:
  - The init container(s) are executed BEFORE the main container.
  - The main container is executed only AFTER the init container(s) end.
- All containers are part of the same Kubernetes Pod.
- Containers share local volumes.

Common use cases for the Initializer Pattern

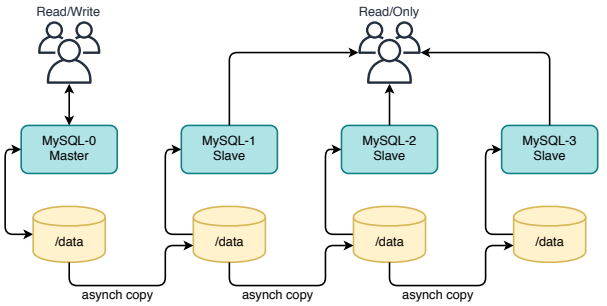
- Initialize the environment of the application
- Create configuration files
- Load static content

Load static content with Init Container



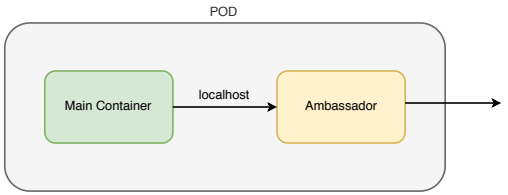
Use an init container to load static contents, e.g. images, video, html, css, from a Git.

Initialize the environment with Init Container



Use multiple init containers to initialize a replicated MySQL Cluster.

The Ambassador Pattern

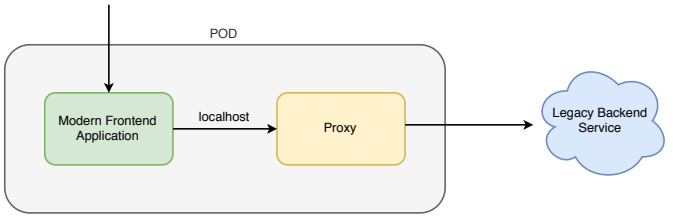


- In the Ambassador Pattern, we have two containers:
  - The main container: implement the core logic for the application
  - A proxy container: access to external services on behalf of main container
- Containers are part of the same Kubernetes Pod.
- Containers communicate through localhost.

Common use cases for the Ambassador Pattern

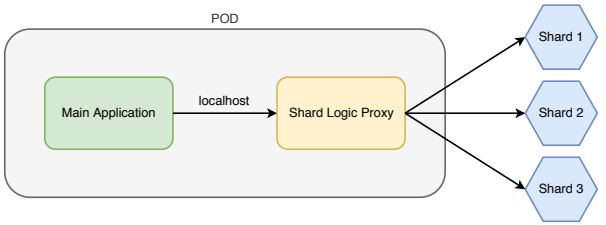
- Protocol translation
- Accessing legacy services
- Accessing sharded services
- A/B Testing and request splitting

Accessing legacy services with an Ambassador proxy



Use an Ambassador proxy to access a legacy backend.

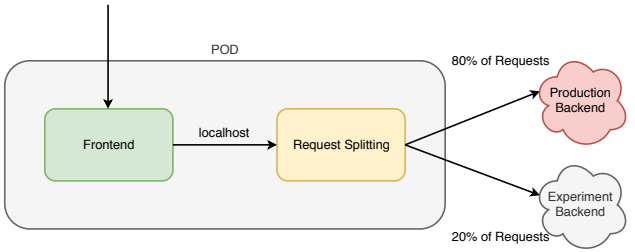
Accessing sharded services with an Ambassador



Use an Ambassador to access a sharded backend:

- Sharding splits up the data layer into multiple disjoint pieces, each hosted by a separate server
- Shard logic is required to recombine pieces togheter
- Implement shard logic in an ambassador container
- The main application is not aware of the sharded service

A/B Testing with an Ambassador



Use an Ambassador to perform experimentation or other forms of request splitting:

- The ambassador splits incoming requests between two (even more) backend services:
  - 80% of requests are sent to the production backend
  - 20% of requests are sent to the experiment backend
- The main application is not aware of the request splitting

Behavorial Patterns

- Distributed Stateless Services
- Distributed Statefull Services

Distributed Stateless Services

- Stateless services don’t require saved state to operate correctly.
- A stateless service can be made of multiple independent instances:
  - redundancy
  - scale
- Individual requests may be routed to any instance of the service.

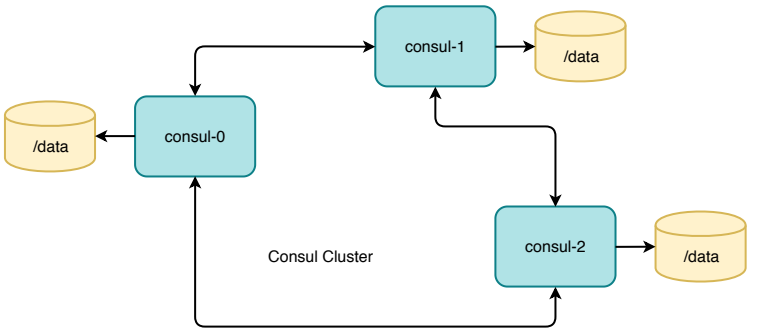
Distributed Statefull Services

- Statefull services require saved state to operate correctly.
- A Statefull service can be made of multiple coupled instances (cluster):
  - redundancy
  - scale
- State is distributed among different instances:
  - replicated
  - sharded
- Individual requests are routed to the master (leader) of the cluster.

Leader Election in Distributed Statefull Services

- A leader election protocol in operation:
  - initially the first leader is selected among all the instances
  - when it fails, another leader takes over
- A distributed consensus alghorithm, e.g. Raft is used to elect the leader and mantain the status.
- Odd number of instances: e.g. 3, 5, 7

Consul



- Here is a good use-case for Stateful sets!
- We are going to deploy a Consul cluster with 3 nodes
- Consul is a highly-available key/value store like etcd

Operators

- Concept introduced by CoreOS
- An operator represents **human operational knowledge in software,** to reliably manage an application:
  - Lifecycle Management: initialize, backup, restore, upgrade
  - Deploying and configuring replication
  - Reacting to failures when intervention is needed
  - Scaling up and down the systems

What are Operators made from?

- Operators combine two things:
  - **Custom Resource Definitions**
  - **Custom controllers** watching the corresponding resources and acting upon them
- A given operator can define one or multiple custom resources.
- The controller code, aka control loop, typically runs within the cluster but it could also run elsewhere.
- An operator is actually a custom controller for custom resources.

How Operators work?

- An operator creates one or more custom resources.
- The controller will watch the custom resources.
- Each time we create/update/delete one of the custom resources, the controller is notified.
- The controller compares the new desired state of the custom resources with their actual state.
- If required, the controller updates the actual state of the custom resources as in the changed desired state.

Why use Operators?

- Kubernetes gives us Deployments, StatefulSets, Services ...
- These mechanisms give us building blocks to deploy applications
- They work great for services that are made of *N* identical containers
- They also work great for some stateful applications like Consul, etcd ...
- They're not enough for complex services:
  - where different containers have different roles
  - where extra steps have to be taken when scaling or replacing containers

Use cases for Operators

- Systems with primary/secondary replication
  - Examples: MariaDB, MySQL, PostgreSQL, Redis …
- Systems where different groups of nodes have different roles
  - Examples: ElasticSearch, MongoDB …
- Systems with complex dependencies (that are themselves managed with operators)
  - Examples: Flink or Kafka, which both depend on Zookeeper
- Representing and managing external resources
  - Example: AWS Service Operator
- Managing complex cluster add-ons
  - Example: Istio operator
- Deploying and managing our applications' lifecycles

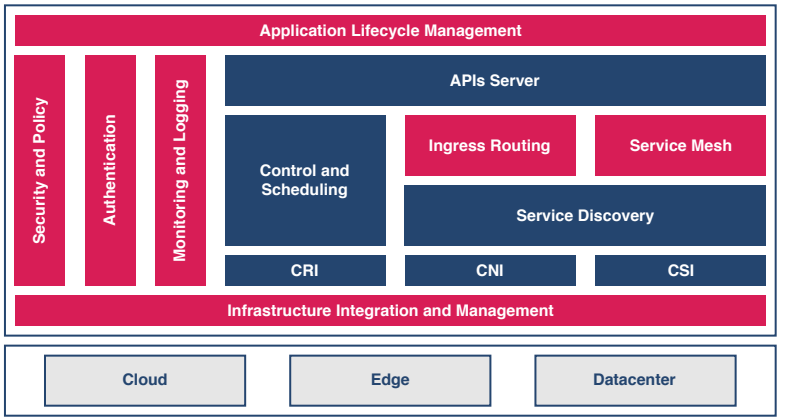
Operators development tools

- Operator Framework
- Kubebuilder
- Kubernetes Universal Declarative Operator

Module 5

Adopting a Container based solution

Kubernetes deployment in a Container as a Service (CaaS)



- Kubernetes offers some basic functionality but leaves the room for third-party tooling and integration.
- The blue blocks above represent the core components of Kubernetes in a general container solution.

Kubernetes Deployment Options

- PaaS Managed Kubernetes.
- IaaS Public Cloud.
- On-premises.
- Hybrid.

Managed Kubernetes

A managed Kubernetes cluster is hosted, maintained and managed by a commercial vendor:

- Google GKE
- Amazon EKS
- Azure AKS
- ...

Based on the PaaS concept. Often, it comes with other additional services already integrated such as:

- image registry
- load balancer
- persistent block storage
- health monitoring
- integrated logging and monitoring
- auto scaling of pods and nodes
- security and access control
- policy and governance

PaaS Managed Kubernetes (pros)

- Only exposes the worker nodes of the Kubernetes cluster to customers.
- Managing the master and etcd database itself is done by the vendor.
- Leave users to focus on the applications and avoid the burden of infrastructure maintenance.
- Scaling out a cluster by adding new nodes can be done within minutes.
- Upgrade of the cluster to the latest version done by the vendor.
- Additional services already integrated.

PaaS Managed Kubernetes (cons)

- Users have few options (or even nothing) to customize the Kubernetes cluster.
- Limited storage and network options.
- Latest version upgrade follows the vendor's schedule.
- Costs.

IaaS Public Cloud

- Public Cloud vendors offer the option to install Kubernetes on their IaaS platforms.
- Users are fully responsible for installing and maintaining the Kubernetes cluster.
- Other additional services such as registry, load-balancer, storage, ... are available as vendor's offer.

IaaS Public Cloud (pros)

- Users have full options to customize the Kubernetes cluster.
- Latest version upgrade is a user's choice.
- No need to manage the infrastructure: network, compute, and storage.

IaaS Public Cloud (cons)

- Users are fully responsible for installing and maintaining the Kubernetes cluster.
- Need to pay resources for the Control Plane, i.e. master nodes.
- Limited storage and network options (they depends on the vendor's offer).
- Other additional services need to be integrated.
- Costs.

On-premises

Enterprise users often prefer to run Kubernetes on their data center for a full control or deep customization.

Options:

- Upstream distributions:
  - DIY
  - Supported and/or managed: e.g. **Clastix Enterprise Kubernetes**
- Vendor distributions:
  - Red Hat OpenShift
  - VMware Tanzu
  - Suse Rancher

On-premises (pros)

- Deep customization (upstream distributions) or medium (vendor distributions).
- Full control on the installation and the configuration options.
- Latest version upgrade is a user's choice.
- Wide storage and network options.
- Privacy and ownership of data.
- Costs.

On-premises (cons)

- Users are fully responsible for installing and maintaining the cluster.
- Need to allocate resources for the Control Plane, i.e. master nodes.
- Infrastructure management, i.e. compute, storage, network.
- Other essential services such as registry, load-balancer, storage, ... need to be set and mantained.

Hybrid

It is an evolving use case where users set Kubernetes deployment that spans the on-premises datacenter and public clouds.

- It leverages on the overlay networking solutions and the concept of federated clusters in Kubernetes.
- Kubernetes federation can integrate clusters running across multiple zones and regions, or even clusters deployed at multiple clouds.
- Federation creates a mechanism for multi-cluster geographical replication.

Pros:

- multi-cluster geographical replication, which keeps the most critical services running even in the face of regional connectivity or data center failures.

Cons:

- complexity
- working in progress
- not yet production ready

Before adopting Kubernetes

The adoption of Cloud Native and Kubernetes can help on transforming IT from the gatekeepers to the innovators of the modern enterprise.

Here's what the CIO will need to consider for that transformation:

- Business Considerations
- Technology Considerations
- People and Process Considerations

Business Considerations

- Value assessment
- Legacy assessment
- Process assessment

Business Considerations: value assessment

- Make a strategic assessment of the business value of IT transformation and how Kubernetes can impact.
- Kubernetes is not a magic blue pill that cures all IT problems.
- Get a buy-in from all stackeholders which requires a careful evaluation of disruption and a plan for how to mitigate it.

Business Considerations: legacy assessment

- Kubernetes supports legacy applications although these are typically not ideal use case for it.
- Legacy applications often require a redesign process before to be Cloud Native.
- There may be a cost associated with architectural changes.
- If those costs are too high, do not move to Kubernetes just for the sake of using it.
- Prioritize the high-value workloads for migrating to Kubernetes.

Business Considerations: process assessment

- Adopting Cloud Native approach can provide agility to the IT and can help to deliver business value very fast.
- However, if the internal IT delivery process is not ready for Cloud Native, implement changes first to adopt Kubernetes.
- Huge value can be delivered introducing Kubernetes only if the process fits.

Technology Considerations

Adopting Cloud Native and Kubernetes require a change of technological mindset.

- Paradigm shift: resilience instead of reliability of applications.
- Architecture shift: infrastructure and applications are treated as cattle instead of pets.
- Storage shift: consume storage as service, e.g. Object Storage instead of files.
- Interaction shift: declarative approach instead of procedural.

People and Process Considerations

- Talents: a move to Kubernetes will require cross functional talents and skills.
- Cultural: time for silos between development and operational teams is over. Adopting DevOps is a must.
- Failure: a Cloud Native approach provides an opportunity for people to experiment, fail, and learn fast.

Technical challenges when adopting Kubernetes

- Containerization of the applications.
- Distributed architectures: design lightweight and modular services.
- Observation: monitoring, logging, and tracing.
- Kubernetes cluster setup, upgrade and maintenance.
- Storage and Network Integration.
- High Availability: Infrastructure, Control Plane, and Applications.
- Access control, sharing and isolation.
- Policy and governance.
- Multi clusters management.

Thanks!