



# Kubernetes

*Basics*



**IBM Cloud**



Everyone's container journey starts with one container....



At first the growth is easy to handle....





But soon you have many applications, many instances...

And that is why there is container orchestration



# What is container orchestration?

Management of the deployment, placement, and lifecycle of workload containers

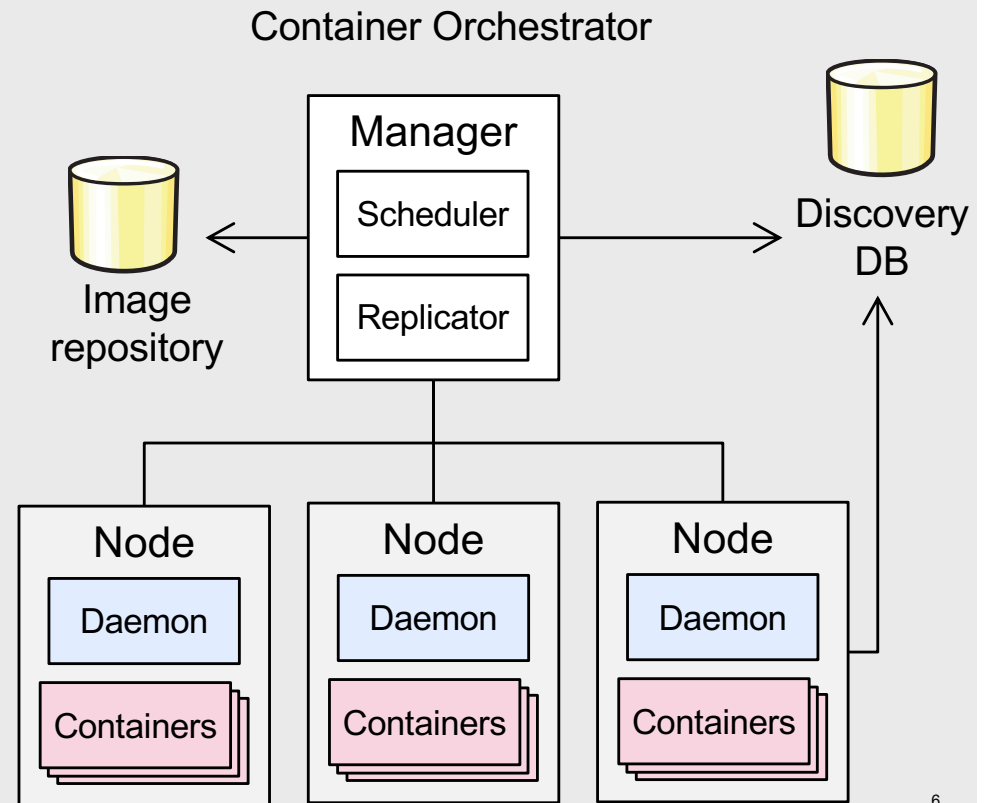
Cluster management creates unified targets for varied workload

Scheduling intelligently distributes containers across nodes

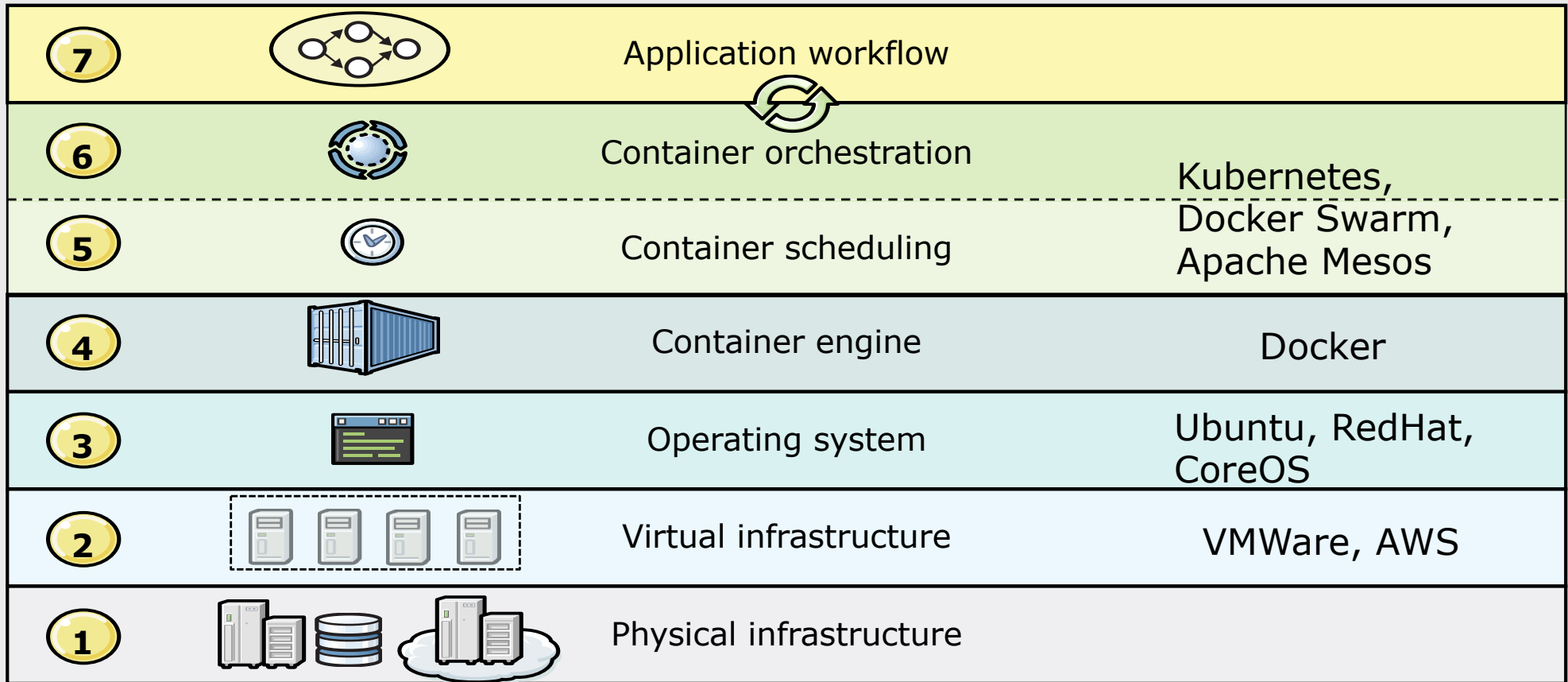
Service discovery knows where containers are located and provides a method to direct requests to them

Replication allows the workload to be scaled

Health management creates a method to ensure the application is assured to be viable by allowing unhealthy containers to be replaced



# Container ecosystem



# What is Kubernetes?



Fully open source container orchestrator inspired and informed by Google's experiences and internal systems

Unified API for deploying web applications, batch jobs, and databases maintaining and tracking the global view of the cluster

Supports multiple cloud and bare-metal environments

Manage applications, not machines providing a better framework to support rolling updates, canary deploys, and blue-green deployments

Designed for extensibility

Rich ecosystem of plug-ins for scheduling, storage, and networking

Open source project managed by the Linux Foundation



# Kubernetes strengths

## Clear governance model

- Managed by the Linux Foundation.
- Google is driving the product features and roadmap, while allowing the rest of the ecosystem to participate.

## Growing and vibrant ecosystem

- IBM, Huawei, Intel, and Red Hat are among the companies making prominent contributions to the project.

## Avoid dependency and vendor lock-in

- Active community participation and ecosystem support.

## Support for a wide range of deployment options

- Customers can choose between bare metal, virtualization, private, public, and hybrid cloud deployments
- Wide range of delivery models across on-premises and cloud-based services.

## Design is more operations-centric

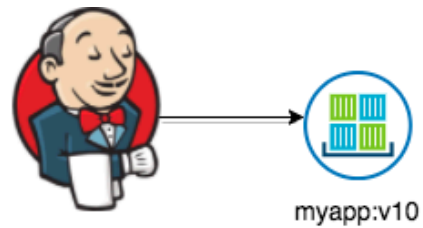
- First choice of DevOps teams.

# Kubernetes

*The Building Blocks*

# Immutability

Build Once - Deploy Everywhere



The same container image is built once and is moved between environments



Dev

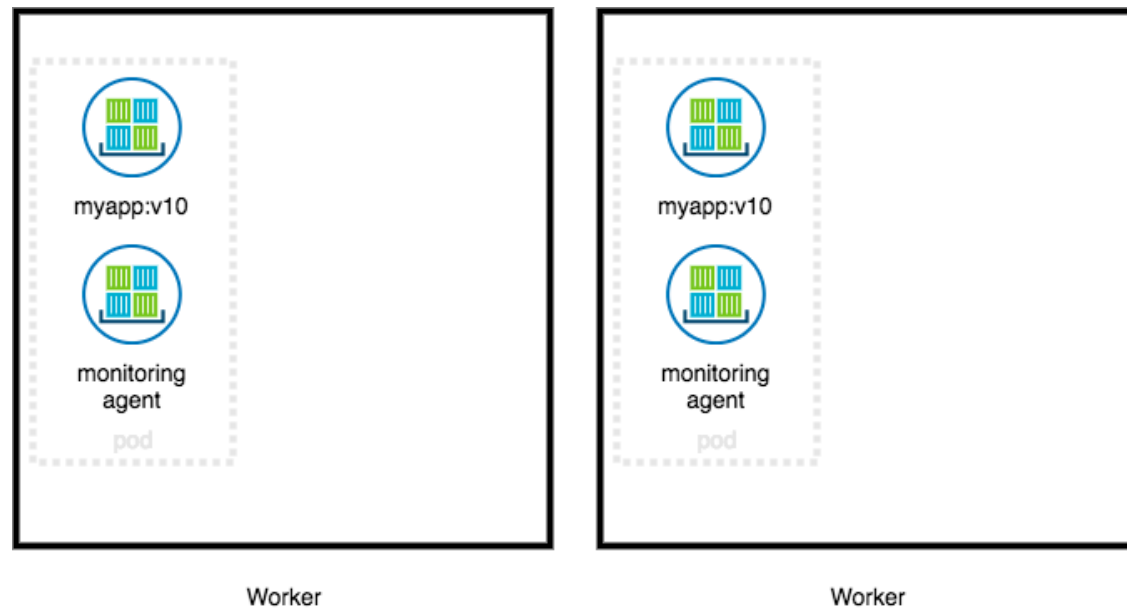


Prod

# Pod

A single unit of work in Kubernetes, which may consist of one or more containers

All containers in a pod are co-located and co-scheduled, and share the kernel namespace (process, storage, network, etc.)



# Pod Health Checking

Pods are automatically kept alive by “process check” checking the basic status of the main process for the application

To go beyond this Kubernetes allows you to create a liveness probe to provide additional means for identifying health.

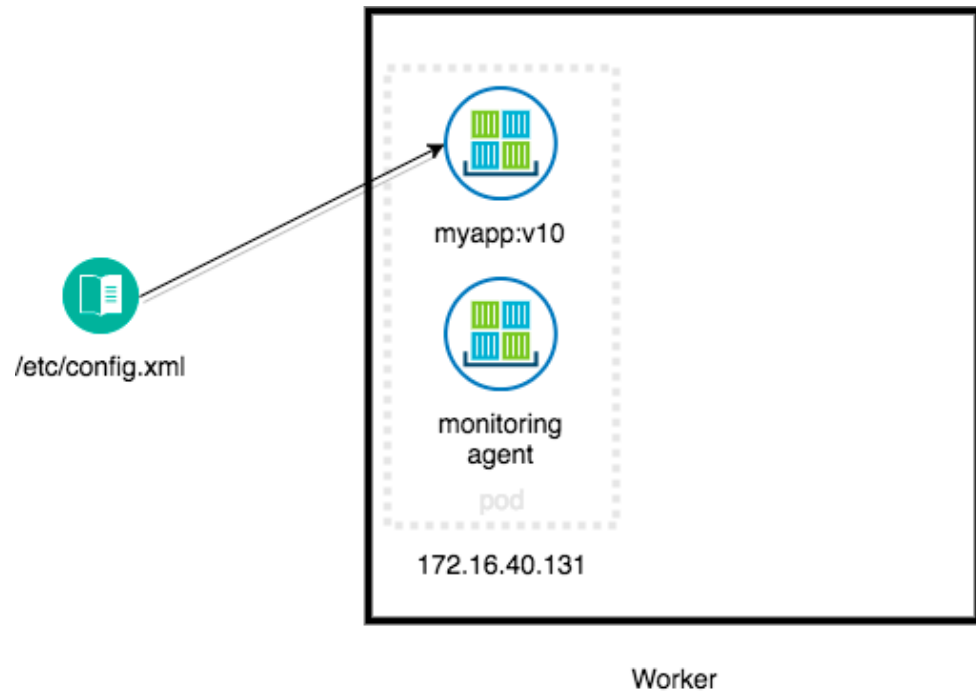
```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-http-healthcheck
spec:
  containers:
    - name: nginx
      image: nginx
      # defines the health checking
      livenessProbe:
        # an http probe
        httpGet:
          path: /_status/healthz
          port: 80
        # length of time to wait for a pod to initialize
        # after pod startup, before applying health checking
        initialDelaySeconds: 30
        timeoutSeconds: 1
      ports:
        - containerPort: 80
```

# Config Maps & Secrets

Share and store configurations, credentials and more

Store the configurations and secrets (credentials, certificates) in the K8s environment and mount them to the local filesystem within container(s)

The container image can move un-changed between environments (i.e. container immutability)

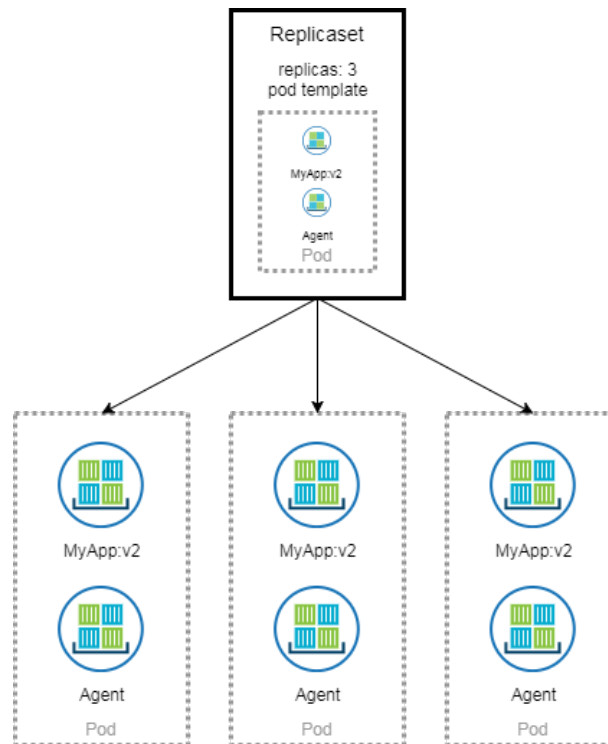


# Replicaset

Scale pods horizontally and provide resiliency

Replicasets run one-to-many instances of the desired pod

When possible the replica pod should be stateless or near-stateless



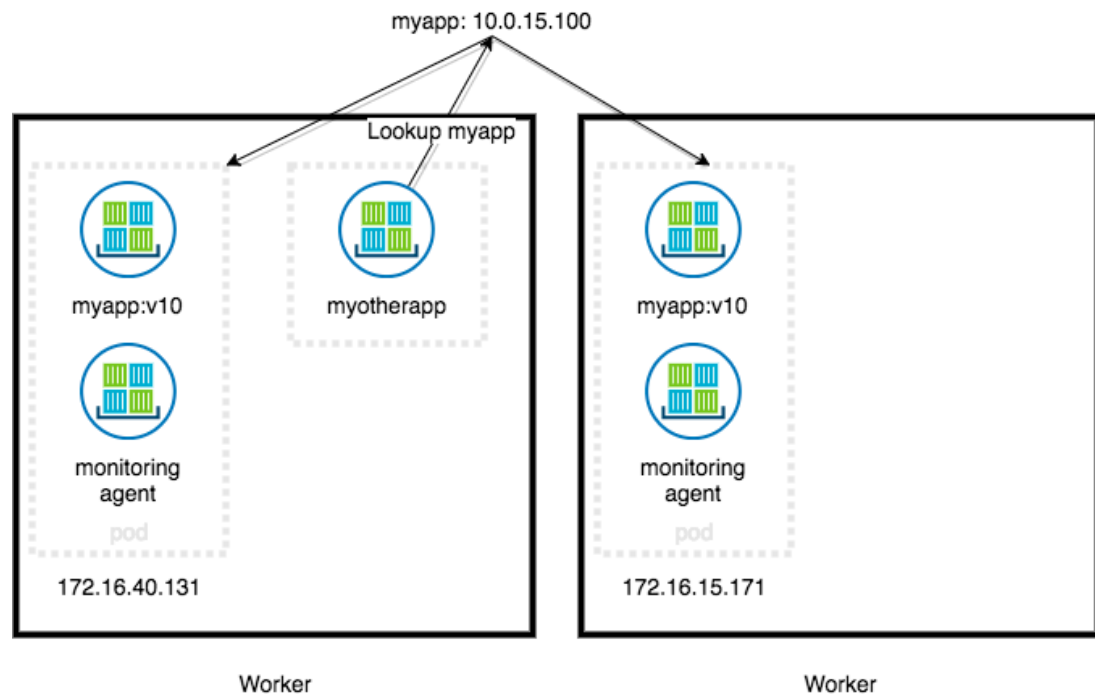
```
apiVersion: apps/v1beta2 # for versions before 1.8.0 use apps/v1beta1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # this replicas value is default
  # modify it according to your case
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
        - name: php-redis
          image: gcr.io/google_samples/gb-frontend:v3
          resources:
            requests:
              cpu: 100m
              memory: 100Mi
      env:
        - name: GET_HOSTS_FROM
          value: dns
          # If your cluster config does not include a dns service, then to
          # instead access environment variables to find service host
          # info, comment out the 'value: dns' line above, and uncomment the
          # line below.
          # value: env
      ports:
        - containerPort: 80
```

# Service Discovery

Kubernetes has an internal DNS that is used as a Service Registry.

A Service resource in Kubernetes results in an entry in the internal DNS

By default, a Service points to an internal Cluster IP that load balances between a set of healthy running pods

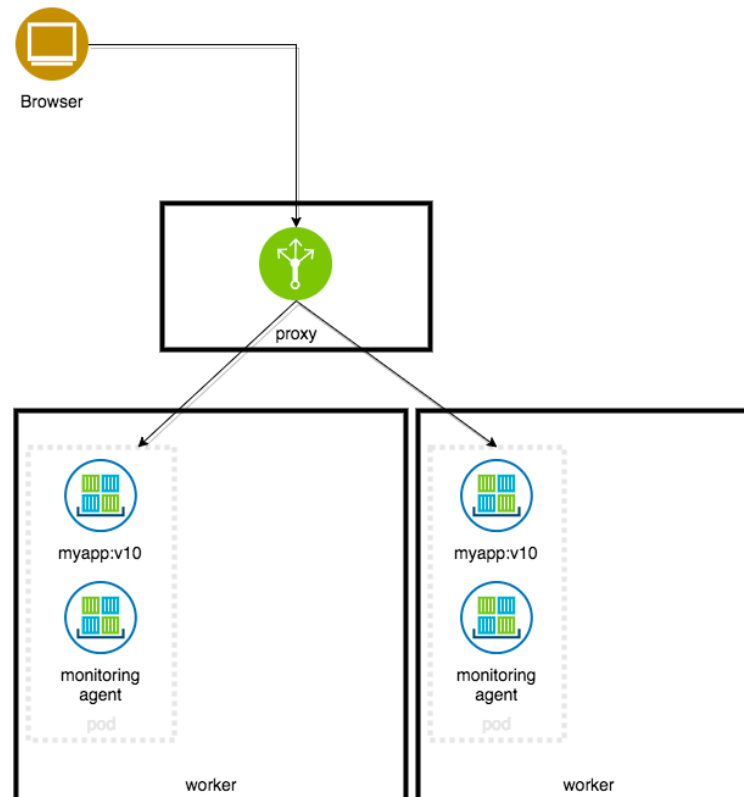




# Ingress Resources

External access to applications running in Kubernetes may be enabled through Ingress resources and proxy nodes

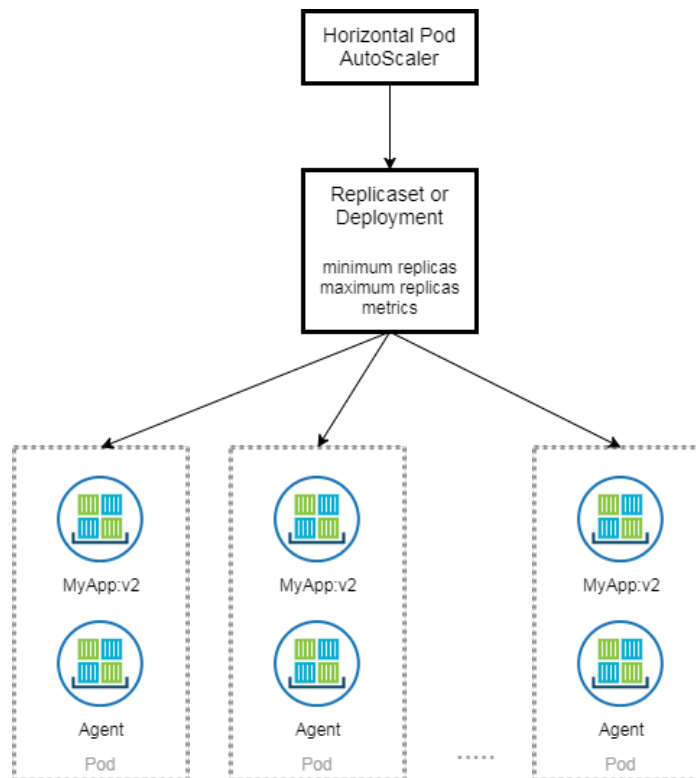
The proxy node(s) in ICP expose services defined in Kubernetes



# More on Scaling

## Horizontal Pod Auto-scaling (HPA)

Allows you to scale the number of running pods in a replicaset based upon resource (or application custom) metrics



```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1beta1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: 50
status:
  observedGeneration: 1
  lastScaleTime: <some-time>
  currentReplicas: 1
  desiredReplicas: 1
  currentMetrics:
  - type: Resource
    resource:
      name: cpu
      currentAverageUtilization: 0
      currentAverageValue: 0
```

# Statefulsets

Similar to replicaset for the purpose of scale or redundancy and/or, statefulsets run one-to-many instances of the desired pod

Unlike replicaset, statefulsets are intended for applications requiring state.

Valuable for applications that require:

- Stable, unique network identifiers
- Stable persistent storage
- Ordered graceful deployment and scaling
- Ordered graceful deletion and termination
- Ordered automated rolling updates

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
    - port: 80
      name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1beta2
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx # has to match .spec.template.metadata.labels
  serviceName: "nginx"
  replicas: 3 # by default is 1
  template:
    metadata:
      labels:
        app: nginx # has to match .spec.selector.matchLabels
    spec:
      terminationGracePeriodSeconds: 10
      containers:
        - name: nginx
          image: gcr.io/google_containers/nginx-slim:0.8
          ports:
            - containerPort: 80
              name: web
          volumeMounts:
            - name: www
              mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
    - metadata:
        name: www
      spec:
        accessModes: [ "ReadWriteOnce" ]
        storageClassName: my-storage-class
        resources:
          requests:
            storage: 1Gi
```

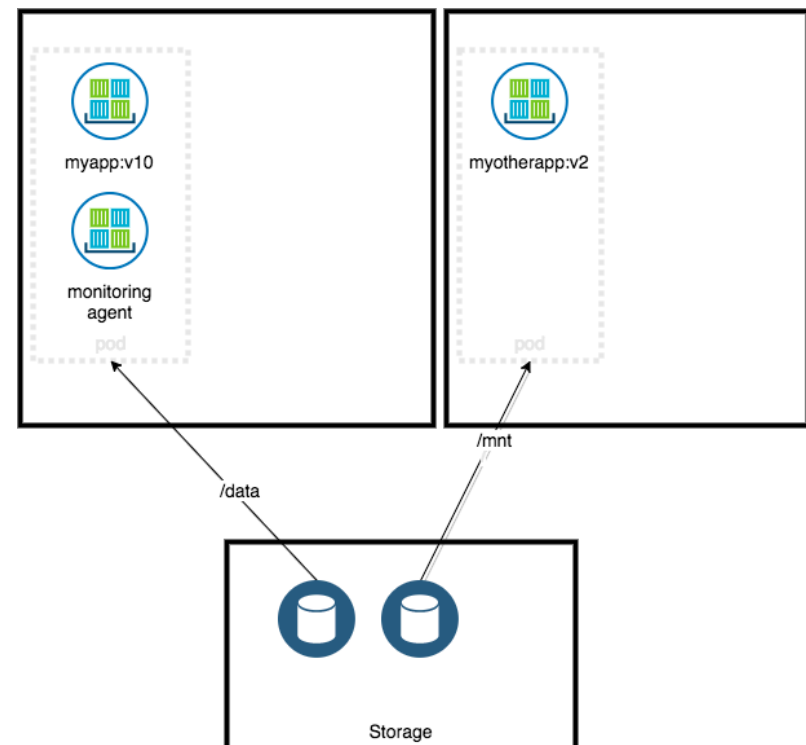
# Persistence & Storage

There are many types of persistent storage and many provider options

Some pods must be able to persist data so that if Kubernetes restarts them on the same or another node data loss is avoided

Kubernetes will re-attach the shared storage when the pod (re)starts

Storage providers support different retention and recycling policies and the definitions of these are not universal



# Kubernetes

*More on Persistent Storage*

# Persistent Storage Overview

Storage that persists beyond the lifecycle of the container allows workload to achieve state



**Docker**, containers and persistence



**Kubernetes**, pods and persistence



**Terminology** and the persistent lexicon



**Value** of the persistence solution

# It is about STATE not STORAGE

We strive to simplify our configuration management by moving towards immutability



## Containers

Containers are purposefully not Virtual Machines and do not carry the burden of a VM's resource management



## Stateless

Applications that containerize the best tend to be stateless with a small number of configuration parameters required for personalization



## State

Some services require backing storage to maintain the current operating state such as a database or applications that persist transactions



## Pervasive

As orchestration is used to place containerized workload throughout the private cloud it becomes important that the persistent storage is equally mobile

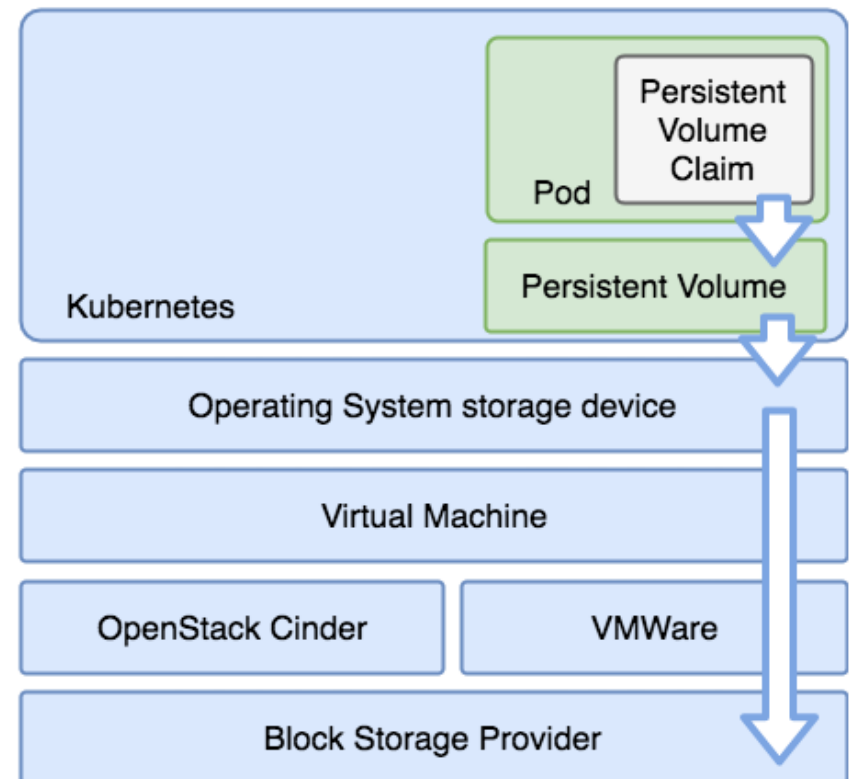
# Persistent Storage

## Solution components

**Persistent Volume** is a storage resource within the cluster. PVs have a lifecycle independent of any individual pod that uses it. This API object encapsulates the details of the storage implementation or cloud-provider-specific storage system.

A **Persistent Volume Claim** is a storage request, or claim, made by the developer. Claims request specific sizes of storage, as well as other aspects such as access modes.

A **StorageClass** describes an offering of storage and allow for the dynamically provisioning of PVs and PVCs based upon these controlled definitions.





# Kubernetes

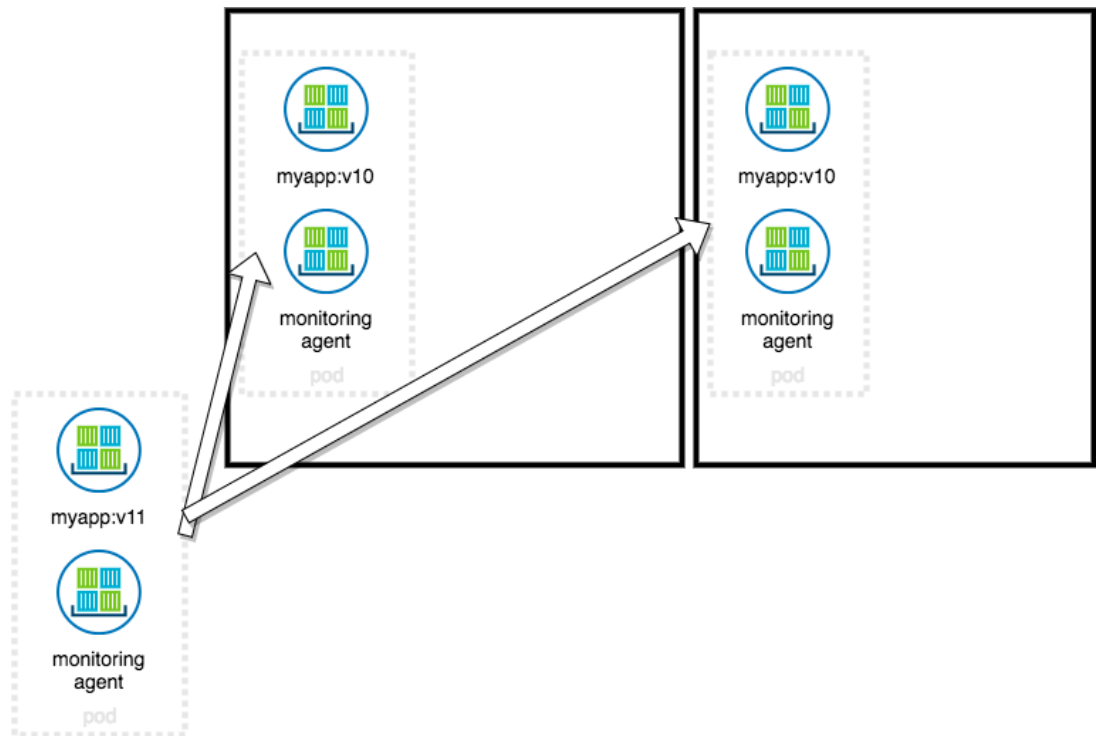
*Deploying Applications*

# Deployments

Deployments manage rolling updates to ReplicaSets and StatefulSets

When a new version of the application is available, the Deployment provides the ability to scale down the previous version of the application and scale up the new version in a controlled fashion with zero downtime

Enables rollback in the case of failure



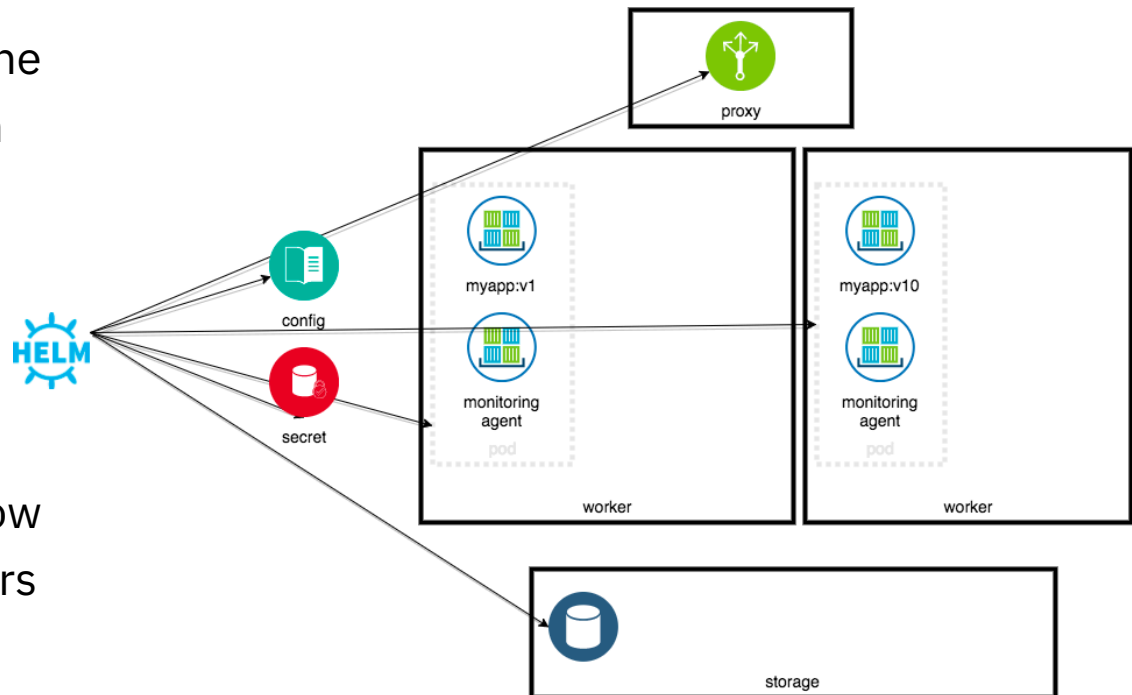
# Helm

An application deployment in Kubernetes is more than just the container image and includes configuration, secrets, services, ingress resources, and storage definitions

Helm allows you to manage all of the artifacts that build your application

Resources are templated and replaced dynamically at installed time

Tells Kubernetes all it needs to know about an application, its parameters and dependencies



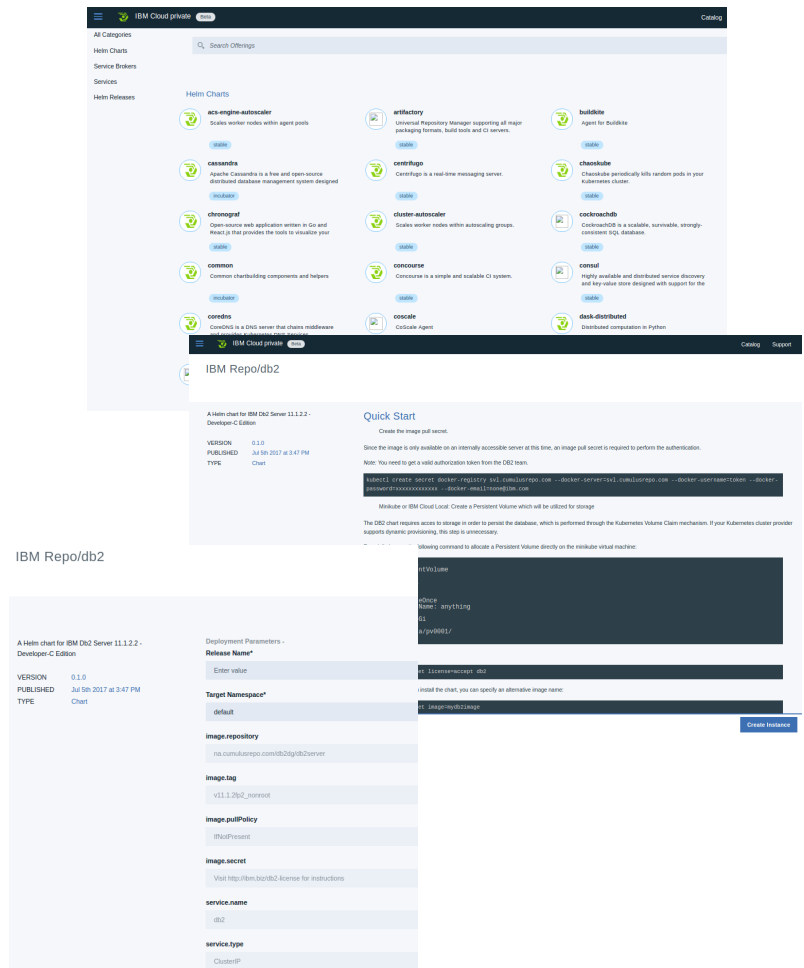
# Application Center components

*Application center provides a centralized location from which you can browse, and install packages in your cluster.*

**Helm:** A tool for managing Kubernetes charts. Charts are packages of pre-configured Kubernetes resources.

**Helm Repository:** A Helm chart repository is a location where packaged charts can be stored and shared.

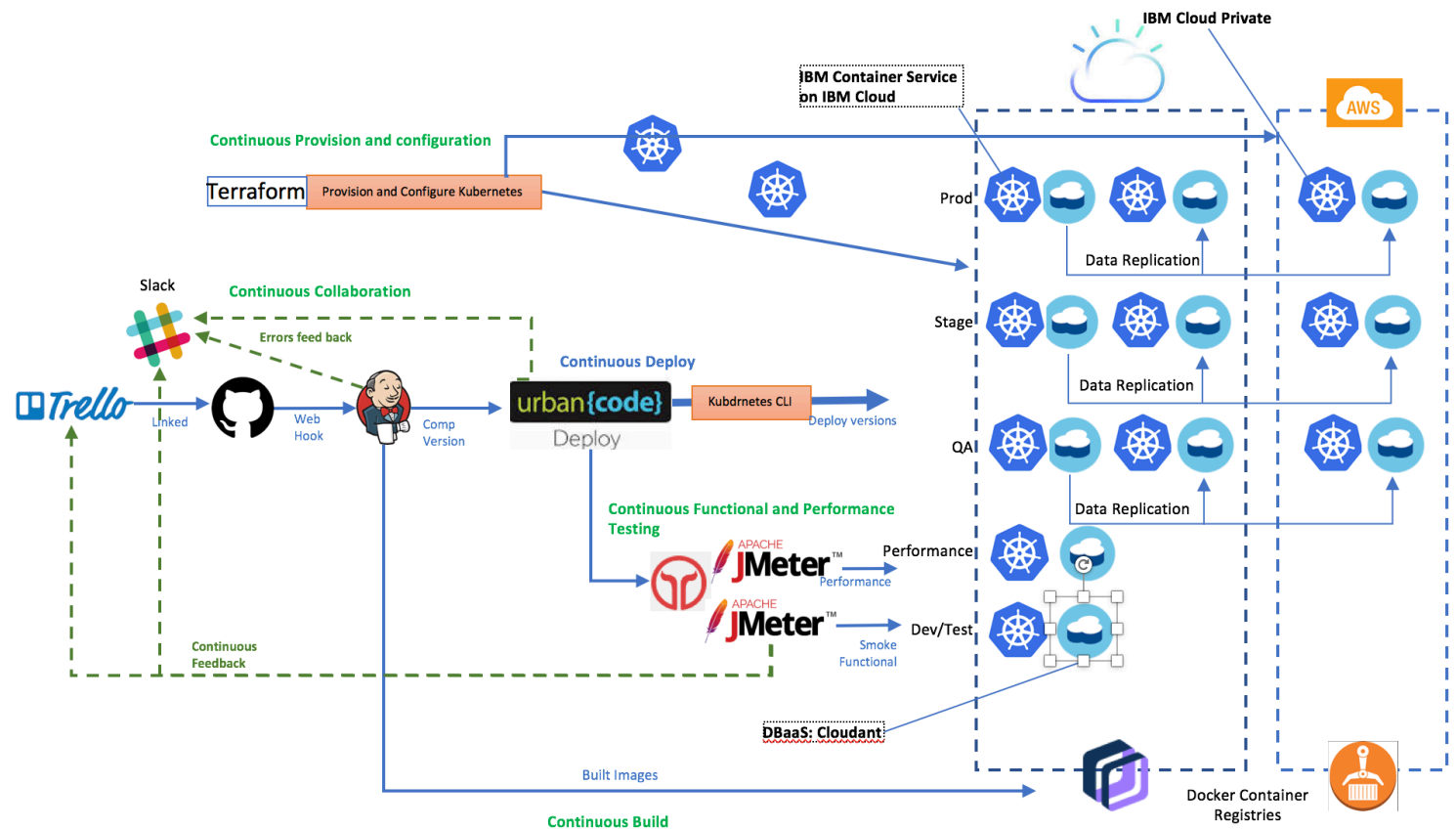
**Tiller:** Runs inside of the cluster, and manages releases (installations) of your charts.



# CI / CD with Jenkins

Build artifacts from Jenkins (the container image) are continuously built and deployed in ICP

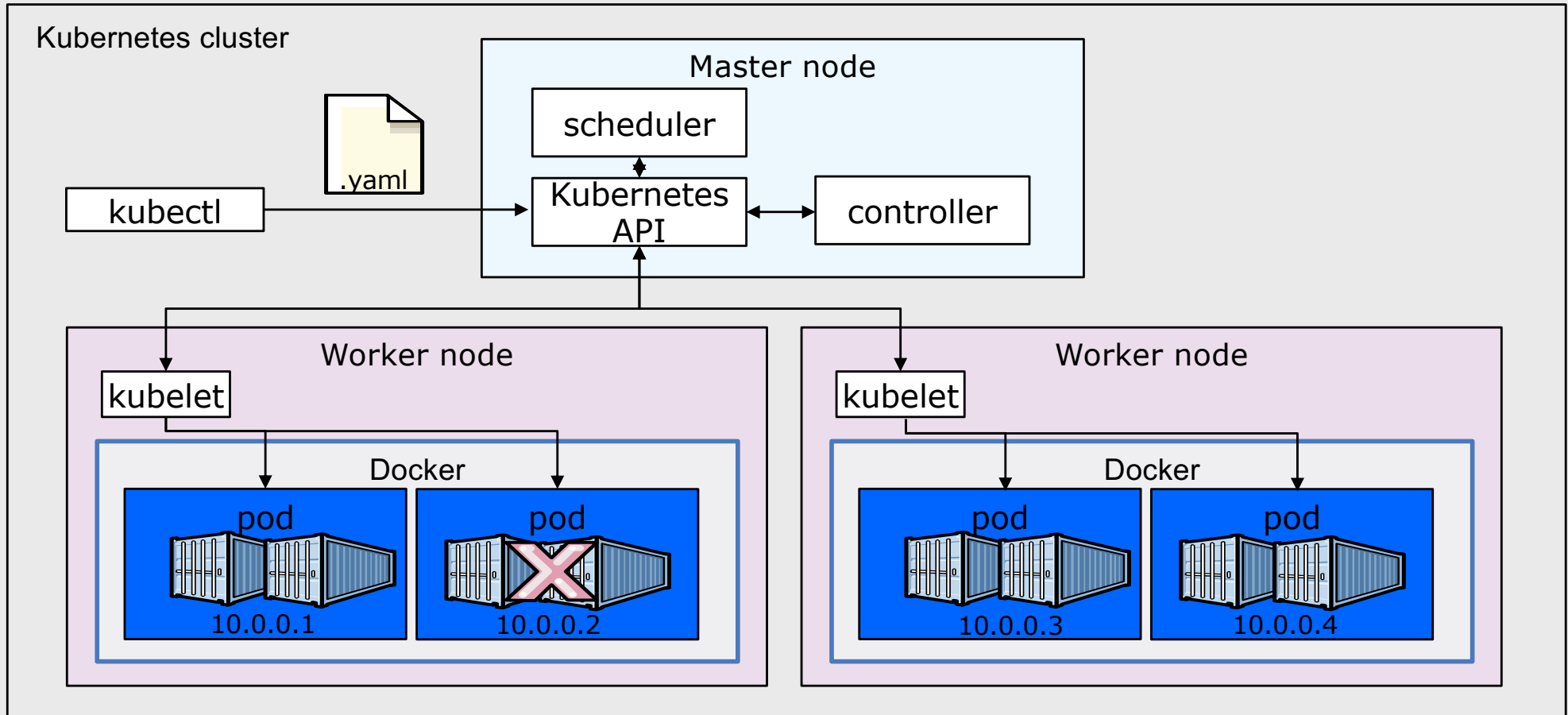
The same image is promoted between each environment (dev, QA, stage, prod)



# Kubernetes

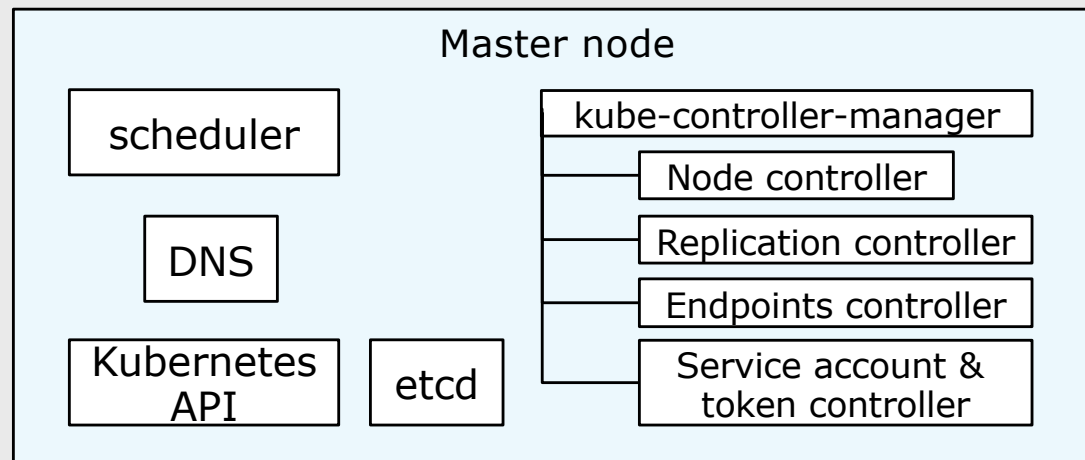
*The Cluster*

# Kubernetes cluster architecture



# Master Node components

- Make scheduling decisions for the cluster, and respond to cluster events, like a node failure
- Can run on any node in the cluster, but typically all master components run on the same virtual machine (vm), and do not run any container apps on that vm





# Master Node Components

## Etcd

- A highly-available key value store
- Stores all cluster data

## API Server

- Exposes API for managing Kubernetes
- Used by kubectl CLI

## Scheduler

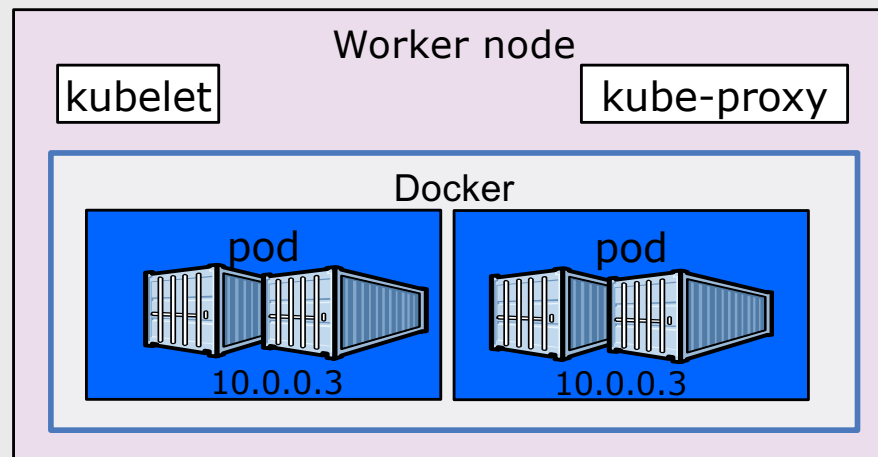
- Selects the worker node for each pods runs

## Controller manager

- Daemon that runs controllers (background threads that handle routine tasks in the cluster)
- Node Controller – Responsible for noticing and responding when nodes go down
- Endpoints Controller – Populates the Endpoints object (joins services and pods)
- Service Account and Token Controllers – Create default accounts and API access tokens for new namespaces

# Worker Node Components

- Provide the Kubernetes runtime environment; run on every node
- Maintain running pods



# Kubernetes

*Further Concepts*

# Naming in Kubernetes

## Name

- Each resource object by type has a unique name

## Namespace

- Resource isolation: Each namespace is a virtual cluster within the physical cluster
  - Resource objects are scoped within namespaces
  - Low-level resources are not in namespaces: nodes, persistent volumes, and namespaces themselves
  - Names of resources need to be unique within a namespace, but not across namespaces
- Resource quotas: Namespaces can divide cluster resources
- Initial namespaces
  - `default` – The default namespace for objects with no other namespace
  - `kube-system` – The namespace for objects created by the Kubernetes system

# Kubernetes configuring Containers and Resources

## Label

- Metadata assigned to Kubernetes resources (pods, services, etc.)
- Key-value pairs for identification
- Critical to Kubernetes

## Selector

- An expression that matches labels to identify related resources

# Kubectl commands

Support different approaches to working with Kubernetes objects:

- Imperative commands on live objects.
- Individual configuration files or directories of files.

**Important:** maintain a consistent approach when working with the same object; do not mix approaches.

Basic syntax:

`<verb> <objecttype> [<subtype>] <instancename>`

- Where the `<verb>` is an action such as: **create**, **run**, **expose**, **autoscale**.
- `<objecttype>` is the object type, such as a **service**.
- Some objects have subtypes. For example, a service has **ClusterIP**, **LoadBalancer**, **NodePort**.
- Use the **-h** flag to find the arguments and flags supported by a subtype
- `<instancename>` specifies the name of the object

# Kubectl command useful examples

## Get the state of a cluster

```
$ kubectl cluster-info
```

## Get all the nodes of a cluster

```
$ kubectl get nodes -o wide
```

## Get info about the pods of a cluster

```
$ kubectl get pods -o wide
```

## Get info about the replication controllers of a cluster

```
$ kubectl get rc -o wide
```

## Get info about the services of a cluster

```
$ kubectl get services
```

## Get full config info about a Service

```
$ kubectl get service  
NAME_OF_SERVICE -o json
```

## Get the IP of a Pod

```
$ kubectl get pod NAME_OF_POD -  
template={{.status.podIP}}
```

## Delete a Pod

```
$ kubectl delete pod NAME
```

## Delete a Service

```
$ kubectl delete service  
NAME_OF_SERVICE
```

# Resources

## Kubernetes tutorial

- <https://kubernetes.io/docs/tutorials/kubernetes-basics/>

## Introduction to container orchestration

- <https://www.exoscale.ch/syslog/2016/07/26/container-orch/>

## TNS Research: The Present State of Container Orchestration

- <https://thenewstack.io/tns-research-present-state-container-orchestration/>

## Large-scale cluster management at Google with Borg

- <https://research.google.com/pubs/pub43438.html>



