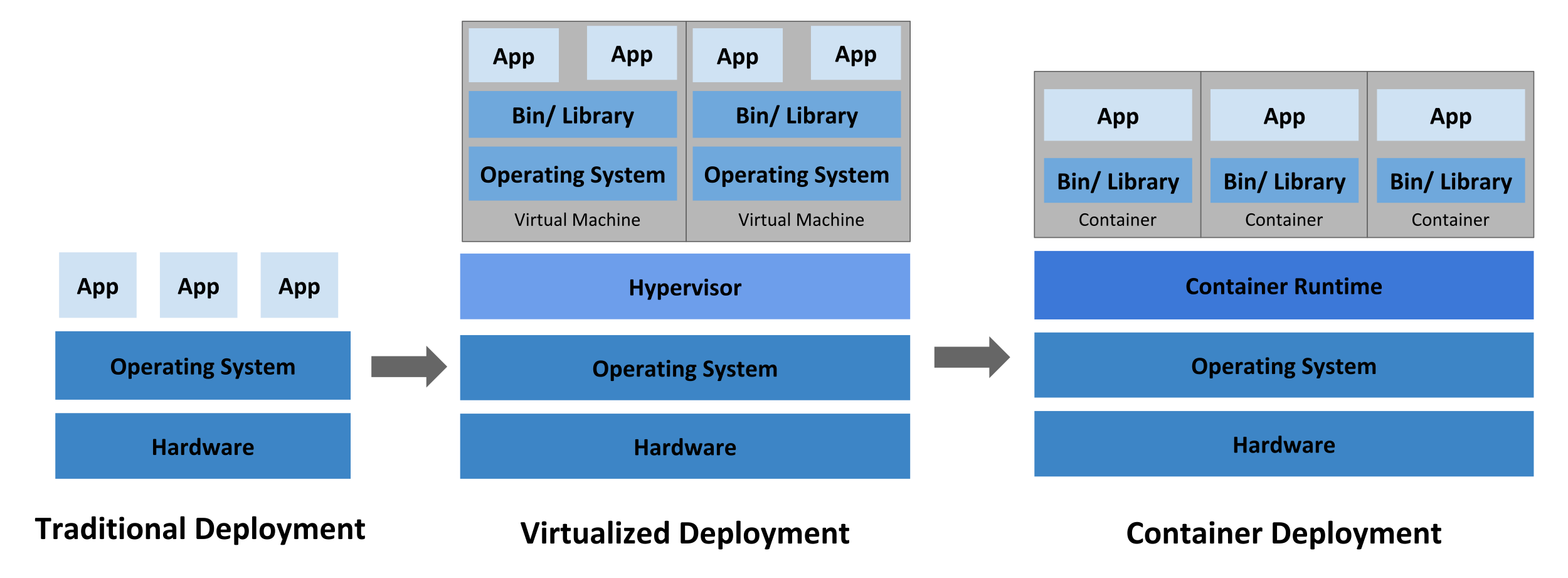
Kubernetes

# Overview

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. Developed by google and open sourced in 2014, it allows for production grade container orchestration, making scalable applications quickly.

# Concepts

## Types of Depolyment



### Traditional

Applications running on phyiscal dedicated servers with a single operating system (OS). There is no way to define resource boundaries for the different programs running on the server, so there were often resource allocation issues and security implications.

### Virtualized

Multiple virtual machines (VM) running on one physical server cpu. Applications can be isolated between VMs, providing better management of resources and security between applications.

Each VM is a full machine running its OS on top of virtualised hardware. VMs were often used in the cloud with all servers being run by a large sysops manager.

### Containers

Containers are similar to VMs however they have relaxed isolations properies, allowing them to share the same OS. Being able to share the same OS allow containers to be very lightweight.

Containers have their own filesystem and share of hardware resources, but as they are decoupled from the OS they are portable across clouds and OS distributions.

Some of the benefits of running applications in containers are:

* Agile application creation and deployment
* CI/CD due to quick container deployment and easy rollbacks due to image immutability
* Separate dev and ops since containers are created at build/release time instead of deployment, allowing issues to be quickly picked up by the dev team.
* Environmental consistency due to containers running the same on a laptop as the cloud
* Loosely coupled elastic micro-services, since applications can be broken into smaller pieces which can be deployed and managed dynamically
* Resource management, since containers can be scaled quickly and dynamically to keep efficiency high

## Kubernetes

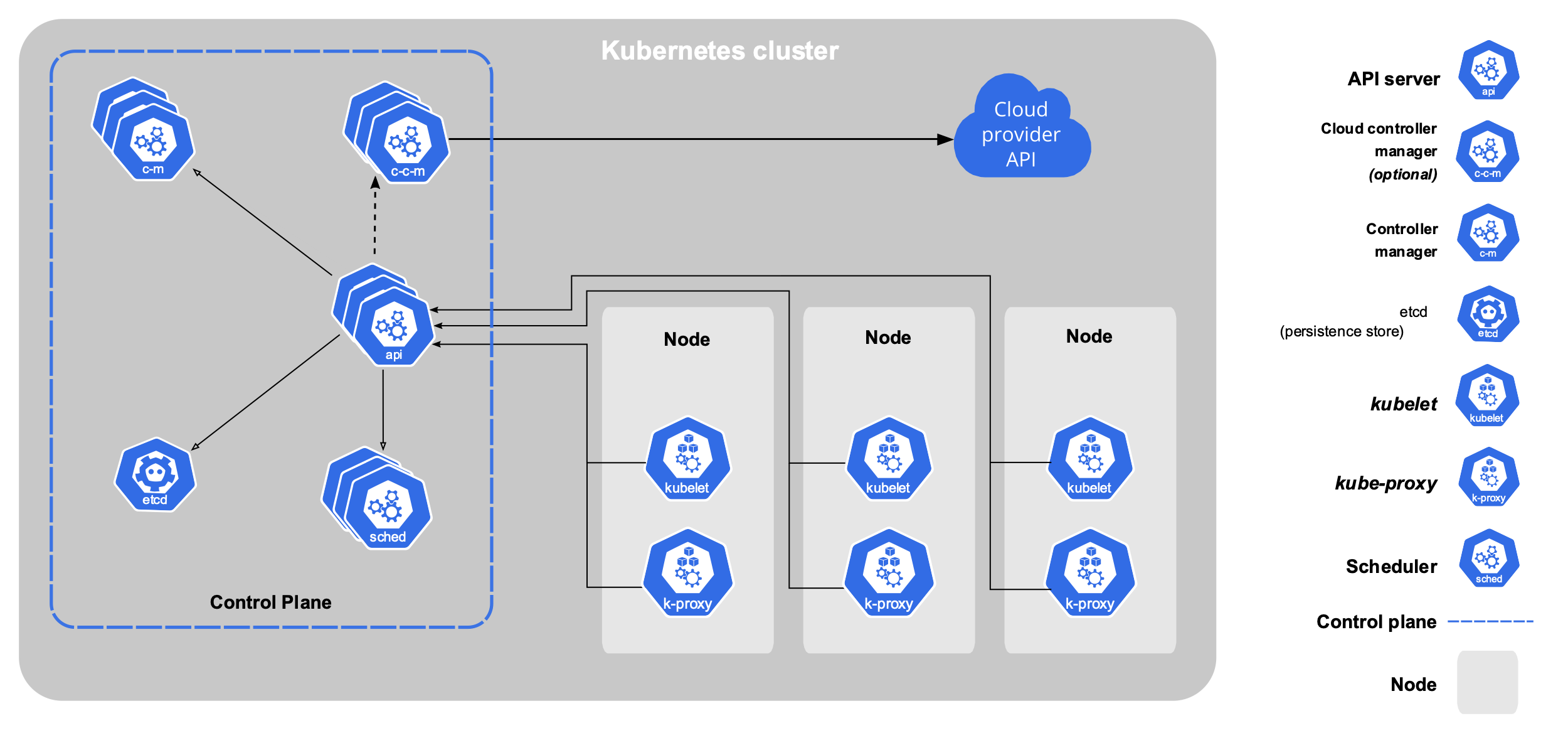
Kubernetes is a container management system, designed to manage distributed systems resiliently. It allows for automated scaling and self healing.

The key features of Kubernetes are:

* Service discovery and load balancing - kubernetes can expose a container using dns name or ip address, then load balance if traffic to the container is high
* Storage orchestration - automatically mount storage system
* Automated rollouts - describe the desired state for the deployed containers and kubernetes will change the actual state to match the desired state
* Automatic bin packing - provide a cluster of nodes to run containerised tasks and kubernetes will fit as best it can, even with limited resources
* Self-healing - restarts containers which fail, replace containers, and kill containers which do not respond to user defined health checks
* Secret and configuration management - store sensitive information such as ssh keys in kubernetes and then include in deployment without having the rebuild container images

## Components

A Kubernetes cluster is a set of worker machines called nodes, that run containerised applications in pods. Running a cluster across multiple computers provides fault tollerance and high availability.



### Control Plane

Control plane components make global decisions about the cluster, along with detecting and responding to cluster events. While control plane components can be run on any machine in the cluster it is common to run all components on the same dedicated control machine, which does not container any user applications.

The control plane components are:

* kube-apiserver - Exposes the Kubernetes HTTP API allowing for user management of the control plane components. The API allows a user to query and manipulate the state of objects in the Kubernetes API. Most opertations can be performed using the kubectl cli.
* etcd - Key value store used as Kubernetes backing store for all cluster data. It is important to ensure the backing store data is backed up, incase of coruption or failure.
* kube-scheduler - Watches for newly created Pods with no assigned node and selects a node for them to run on. Takes into account the resource requirements, system contraints, data locality, and many more factors.
* kube-controller-manager - Runs the controller porcesses. Logically each controller has a separate process, but to reduce complexity they are compiled into a single binary and run as one. Manages:
  + Node controller - notices and responding to when nodes go down
  + Replication controller - Maining correct number ofpods for every replication controller object in the system
  + Endpoints controller - joins servers and pods
  + Server Account and Token controllers - Creates default accounts and API access tokenes for new namespaces
* cloud-controller-manager - Embeds cloud-specific control logic. Links cluster into cloud providers API to allow kubernetes to manage cloud specific scaling etc:
  + Node controller - check with cloud provider to see if node has been deleted in the cloud after it stops responding.
  + Route controller - Setting up routes in the underlying cloud infrastructure.
  + Server controller - For creating, updating, and deleting cloud provider load balancers.

### Node Components

The node components run on every node to maintaine running pods and provide the Kubernetes runtime environment:

* kubelet - Agent making sure the containers are running in a pod.
* kube-proxy - Network proxy implementing part of the kubernetes service concept. Maintains network rules on the nodes, allowing network communication to Pods from network sessions inside or outside of the cluster.
* Container runtime - Responsible for running containers, commonly Docker, but can also handle many other such as containerd

### Addons

There are many addons which use the Kubernetes resources to implement cluster features, some of these are:

* DNS - Serves DNS records to Kubernetes services, all clusters should have a cluster DNS.
* Web UI - Allows users to manager and troubleshoot applications running in the cluster
* Container Resource Mointoring - provides time-series metrics about containers.
* Cluster-level Logging - Cluster level logging mechanism responisble for saving container logs to a central log store with search and browsing interface

## Objects

Kubernetes objects are persistent entities in the system which represent the state of the cluster. Specifically, they define:

* What containerized applications are running, and on what nodes
* The resources available to those applications
* Policies around how the applications behave, for example: restarts, upgrades, and fault tollerance.

The object is a record of intent, so once created, the system will constantly work to reach the desired state defined in the record. There are two parts to objects:

* spec - desired state defined when creating the object
* status - describes current state of object, supplied and updated by the kubernetes system

### Defining an object

YAML file containing basic information and spec of the object.

The required fields are:

* apiVersion - kubernetes api version
* kind - kind of object to create
* metadata- data that uniquely identifies object, including name, uid, and optional namespace
* spec - desired state of the object

For example:

**apiVersion**: apps/v1 *# for versions before 1.9.0 use apps/v1beta2*

**kind**: Deployment

**metadata**:

**name**: nginx-deployment

**spec**:

**selector**:

**matchLabels**:

**app**: nginx

**replicas**: 2 *# tells deployment to run 2 pods matching the template*

**template**:

**metadata**:

**labels**:

**app**: nginx

**spec**:

**containers**:

- **name**: nginx

**image**: nginx:1.14.2

**ports**:

- **containerPort**: 80

A common way to then deploy this object is using the kubectl apply command, for example:

kubectl apply -f ./application/deployment.yaml --record

Optional fields are:

* labels - (key/value) specify identifying attributes of objects with relevance to user but not the core system, alphanumeric with dashes between words. labels can be used to select objects. prefix can be set with / before name
* annotations - (key/value) used to describe objects, however cannot be used to select

Recommended labels:

* common-prefix/name - mysql
* common-prefix/instance - mysql-abc
* common-prefix/version - "5.7.21"
* common-prefix/component - database
* common-prefix/part-of - wordpress
* common-prefix/managed-by - chris

### Types of object

* Deployment
* Node
* Service
* Job

## Containers

Containers are repeatable, decoupled application images which can be run anywhere while producing the same behaviour. Typically, containers are refered to in pod definition by their name and registery location, which allows them to be downloaded and deployed. For example, a container could be referred to in a deployment like:

image: fictional.registry.example:10443/imagename

Kubernetes provides several important resources to containers:

* filesystem - combination of image and one or more volumes
* information about container iteself such as hostname (name of pod), and environment variables
* information about other objects in the cluster - services available for container at creation

### Lifecycle hooks

Containers are provided with lifecycle hooks, allowing them to be aware of events in their management lifecycle. The hooks provided are:

* PostStart - Immediatley after a container is created
* PreStop - Immediatley before a container is terminated, it is a blocking hook so much be completed before the signal to stop the container is sent

Container handle the hooks by implementing and registering a handler for the hook. There are two types:

* Exec - specific command is executed inside the container
* HTTP - executes a HTTP request against specific endpoint on the container

## Workloads

A workload is an application running on kubernetes, run inside a pod. Pods represent a set of running containers in the cluster. Pods can be managed by workload resources, which configure controllers that make sure the right number and kind of pods are running to match the state specified.

Workload resources include:

* Deployments
* StatefulSet
* DaemonSet
* Job

## Pods

Smallest deployable unit which can be managed in kubernetes. A pod is a group of one or more containers, with shared storage and network resources. The shared context of. pod is created using a set of linux namespaces, cgroups, and other isolation methods, in a smilar way to how Docker isolates containers.

Generally, pods run a single container, but hey can run multiple container which are tightly coupled and need to work together.

Pods allow applications to scale horizontally by increasing the number of pods per application and load balancing between them. This is referred to as replication.

Pods are created from pod templates, defined in deployment specs as 'templates'. When templates are updated, new pods are created to match the spec in the templates, with old ones being removed.

Each pod is assigned an IP address, containers inside pod can communicate on localhost.

### Lifecycle

A pods lifecycle is defined in phases:

* Pending - Accepted by cluster but one or more container has not yet been set up fully and is ready to run
* Running - Pod is bound to node and all containers have been created, at least on container is running
* Succeeded - All containers in pod have terminated in success and will not be restarted
* Failed - All containers in pod have termined but one has failed (non-zero status or terminated by system)
* Unknown - State of pod could not be obtained

While a pod is running, kubernetes can restart containers to handle some kinds of fault. However, pods are designed to be quickly deleted and deployed, so in general pod do not self heal, instead if they fail, they are simply replaced to heal the system.

Since pod states rely on containers, there are also container states which are kept track of:

* Waiting - Container is not either running or terminated.
* Running - Container is executing without issues
* Terminated - Container began execution and ran to completion or failed

### Probes

A probe is an optional diagnostic performed by the kubelet on a container. The kubelet calls a handler implemented by the container, which allows the container to signal its condition. Probes are defined in the container object definiton.

There are three types of handlers:

* Exec - Executes specific command inside the container, successful if 0 exit. For example, checking a file exists every 5 seconds:

livenessProbe:

exec:

command:

- cat

- /tmp/healthy

initialDelaySeconds: 5

periodSeconds: 5

* TCPSocketAction - TCP check against Pods ip address on a specific port, successful if port is open. For example, protecting a slow container with a startup probe, testing the opening of port 8080, once passed liveness and readiness probes will start. This probe will be performed a max of 30 times with 10 seconds inbetween:

startupProbe:

tcpSocket:

port: 8080

failureThreshold: 30

periodSeconds: 10

* HTTPGetAction - HTTP GET request against pods IP address on specific port and path, successful if response code 200 <= x < 400. For example, requesting /healthz every 3 seconds (adding custom header to filter from logs). Each probe has 2 seconds to timeout and if only one fails the pod will stop receiving traffic:

readinessProbe:

httpGet:

path: /healthz

port: 8080

httpHeaders:

- name: Custom-Header

value: Awesome

initialDelaySeconds: 3

periodSeconds: 3

timeoutSeconds: 2

failureThreshold: 1

There are three kinds of probe:

|  |  |  |
| --- | --- | --- |
| **Probe** | **Behaviour** | **When to use** |
| livenessProbe | Indicates if container is running, if it fails the kubelet kills the container, the container will then restart if it is subjected to a restart policy. If the container does not provide a liveness probe, default state is success. | Generally used when a container cannot crash on its own when unhealthy, or if specific behavior via probe indicates health. |
| readinessProbe | Indicates whether container is ready to respond to requests. Failure remove the pods IP address from the endpoints of all Services which match the Pod. Default state if failed, if container does not provide, default is success. | Generally used to only start sending traffic to a pod when it can process it, useful if container takes some time to load and traffic can be missed. Also good if you don’t want to kill a pod but want to stop it receiving traffic when unhealthy |
| startupProbe | Inidcates whether application within container is started, if startup probe is provided, all other are disabled until it succeeds. On failure, the kubelet kills the container and restarts if policy is in place. | Good for containers which take a long time to come into service. Instead of setting long liveness intervals, set startupProbe with long period. |

liveness and readiness probe options:

* initialDelaySeconds - number of seconds after the container has started before probes start. default 0
* periodSeconds - How often to perform probe. default 10
* timeoutSeconds - Number of seconds after which the probe timesout. default 1
* successThreshold - Minimum number of consecutive successes for the probe to be considered successful after having failed. default to 1
* failureThreshold - Number of times before to try probe before marking as unhealthy and restarting if liveness or marking unready if readiness

### Deletion

Pods should be allowed to gracefully terminate when they are no longer needed instead of being killed, so they can clean themselves up.

When deletion is requested, the cluster records and tracks the intended grace period before forcefully killing the pod.

Typically the container runtime sends a TERM signal to the main process of each container, although this can be changed to STOPSIGNAL if the container responds to that instead. Once the grace period has expired, a KILL signal is sent to any remaining processes and the Pod is deleted from the API server.

### Preset

PodPreset is an API resource for injecting additional runtime into a Pod at creation time. Label selectors are used to specify the pods to which a given PodPreset applies. It allows templates not to have to specifiy all information for every pod.

Pod preset must be enabled in the API:

settings.k8s.io/v1alpha1/podpreset

by:

settings.k8s.io/v1alpha1=true --runtime-config

## Services

When applications are deployed using Deployments, the containers in a pod will not actually be using the port specificed in the spec on the node. Instead, each container is given is assigned its own IP address from a flat, cluster wide IP address pool, which the port is then exposed on, allowing all containers to be accessed from any other pods or nodes in the cluster using IP.

While it would be possible to connect to pods directly, since pods are nonpermanent managing the IPs of a group of similar pods to connect to requires automating. Kubernetes uses Services to do this.

Services define a logical set of pods and a policy by which to access them. The set of pods targeted by a service is usually determined by a selector (label), and the service will continually scan for pods which match its selector. Note - when creating a deployment, the service will be selecting the template pod label, not the deployment labels. When created a service is given a unique IP address (clusterIP) which will not change while the service is alive. Pods can then talk to the service, IP knowing communication will be automatically load balanced out to the correct pods.

A service is a kubernetes object and is defined to map incoming requests to specific ports defined by port number or port name of the pod:

**apiVersion**: v1

**kind**: Service

**metadata**:

**name**: my-service

**spec**:

**selector**:

**app**: MyApp

**ports**:

- **name**: http

**protocol**: TCP

**port**: 80

**targetPort**: 9376

- **name**: https

**protocol**: TCP

**port**: 443

**targetPort**: 9377

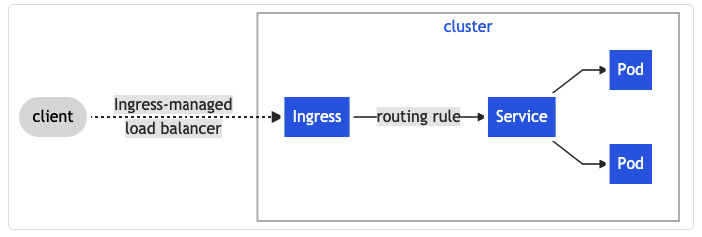
### Publishing Service

For some services, such as frontends, the service may need to be exposed on an external IP address. ServiceTypes allows the specification of the type of service:

* ClusterIP (default) - exposes on cluster-internal IP
* NodePort - exposes on each Node's UP at static port, external contact using <NodeIP>:<NodePort>
* LoadBalancer.- exposes externally using a cloud providers load balancer to balance between NodePorts (auto created)
* ExternalName - maps service to externalName field by returning CNAME record with its value

### Ingress

Ingress can also be used to expose a service by creating an entry point for the cluster, allowing multiple services to be used under the same IP. They are generally used to give services externally reachable URLs, load balance traffic, terminate SSL/TLS, and offer name based veritual hosting. Common ingress controllers are: ingress-nginx and trafiek.



Create an ingress object:

**apiVersion**: networking.k8s.io/v1

**kind**: Ingress

**metadata**:

**name**: minimal-ingress

**annotations**:

**nginx.ingress.kubernetes.io/rewrite-target**: /

**spec**:

**rules**:

- **http**:

**paths**:

- **path**: /testpath

**pathType**: Prefix

**backend**:

**service**:

**name**: test

**port**:

**number**: 80

Ingress rules contain:

* host (optional) - define traffic to select, if not included all traffic selected
* list of paths - each has associated backed defined with service.name, service.port.name or service.port.number. Path types can be Prefix or Exact, allowing for specific matching of routes, multiple matches will match the longest match.

A default backend if often created to handle any requests which are not selected.

Resource backends is an ObjectRef to another Kubernetes resource with the same namespace. Common usage is to ingress data to an object storage backend with static assets.

## Storage

Since volumes in dynamically created containers are ephemeral, kubernetes volume abstraction enable consistent volumes which can be shared between containers in a pod.

Kubernetes allows for a large number of volume types:

<https://kubernetes.io/docs/concepts/storage/volumes/>

A common storage option is emptyDir, which is a volume created when a Pod is assigned to a node, existing so long as the pod is running on the node. Mounted on whatever medium backs the node but can be mounted in memory.

### PersistentVolume

A PersistentVolume is a piece of storage in a cluster which as by provisioned using Storage Classes. PVs are plugin like Volumes but have a lifecycle independednt of Pods. PersistentVolumeClaims (PVCs) are requests for storage by a user, using up PV resources. PVs and PVCs allow storage for pods to be abstracted from the StorageClass.

A common PV StorageClass is AWS EBS, however it is important to note that it can only be bound to one aws ec2 node. To setup first create the volume using the aws api, ensure it is in the same region as the EC2 node instances

## Config Maps

A ConfigMap is an API object used to store non-configdentail data in key-value paris. Pods then consume the data as:

* environment variables (no auto updates)
* configuration files in a volume (auto updates)
* command and arg in container
* write code to run inside a pod that uses the kubernetes API to read a ConfigMap

ConfigMaps decouple containers from environment specific config, allowing for more portable applications.

### Define a config map

**apiVersion**: v1

**kind**: ConfigMap

**metadata**:

**name**: game-demo

**data**:

*# property-like keys; each key maps to a simple value*

**player\_initial\_lives**: "3"

**ui\_properties\_file\_name**: "user-interface.properties"

*# file-like keys*

**game.properties**: *|*

*enemy.types=aliens,monsters*

player.maximum-lives=5

**user-interface.properties**: *|*

*color.good=purple*

color.bad=yellow

allow.textmode=**true**

### Consume as files

Reference config map in Pod spec as volume:

**volumes**:

- **name**: foo

**configMap**:

**name**: myconfigmap

Mount config map volume in container using volumeMounts:

**volumeMounts**:

- **name**: foo

**mountPath**: "/etc/foo"

**readOnly**: **true**

Specific items from the configMap can be selected using:

**volumes**:

- **name**: config

**configMap**:

**name**: game-demo

**items**:

- **key**: "game.properties"

**path**: "game.properties"

- **key**: "user-interface.properties"

**path**: "user-interface.properties"

### Consume as ENV variables

Define env in container definition using configMapKeyRef:

**containers**:

- **name**: demo

**image**: alpine

**env**:

- **name**: PLAYER\_INITIAL\_LIVES

**valueFrom**:

**configMapKeyRef**:

**name**: game-demo

**key**: player\_initial\_lives

## Secrets

Store and manage senstive information in kubernetes instead of putting it in pod definition or container image. There are various different types of predefined secrets, along with standard user-define Opaque secrets:

<https://kubernetes.io/docs/concepts/configuration/secret/#secret-types>

### Create secret

**apiVersion**: v1

**kind**: Secret

**metadata**:

**name**: secret-dockercfg

**type**: kubernetes.io/dockercfg

**data**:

**.dockercfg**: *|*

*"<base64 encoded ~/.dockercfg file>"*

Values can either be in the 'data' field as base64 encoded strings, or in 'stringData' field as key-value pairs.

**apiVersion**: v1

**kind**: Secret

**metadata**:

**name**: mysecret

**type**: Opaque

**stringData**:

**username**: administrator

Once a secret object is created apply it to the cluster before deploying using

kubectl apply -f ./secret.yaml

### Use Secret

Similar to ConfigMaps, secrets can be mounted as files or exposed as environment variables.

To mount as file, first define a secret, then mount as volume:

**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: mypod

**spec**:

**containers**:

- **name**: mypod

**image**: redis

**volumeMounts**:

- **name**: foo

**mountPath**: "/etc/foo"

**readOnly**: **true**

**volumes**:

- **name**: foo

**secret**:

**secretName**: mysecret

Secrets can also be projected to specific paths in the mounted volume:

**volumes**:

- **name**: foo

**secret**:

**secretName**: mysecret

**items**:

- **key**: username

**path**: my-group/my-username

Using as environment variables using secretKeyRef:

**apiVersion**: v1

**kind**: Pod

**metadata**:

**name**: secret-env-pod

**spec**:

**containers**:

- **name**: mycontainer

**image**: redis

**env**:

- **name**: SECRET\_USERNAME

**valueFrom**:

**secretKeyRef**:

**name**: mysecret

**key**: username

- **name**: SECRET\_PASSWORD

**valueFrom**:

**secretKeyRef**:

**name**: mysecret

**key**: password

**restartPolicy**: Never

### Permissions

By default, secret permissions are 0644, it is possible to override this with defaultMode:

**volumes**:

- **name**: foo

**secret**:

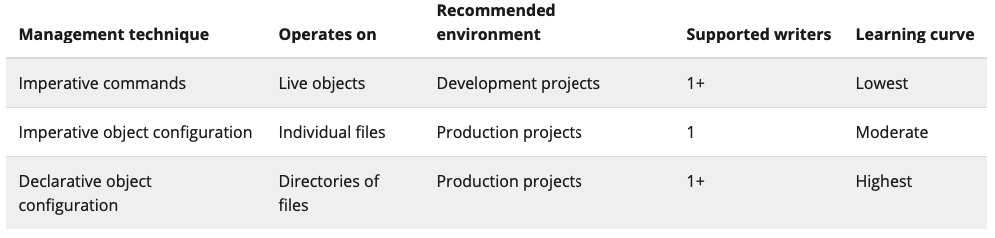
**secretName**: mysecret

**defaultMode**: 0400

# Management

kubectl cli support several different ways of creating and managing objects. Kubernetes objects should be managed using only one technique, mixing and matching will cause undefined behaviour.

## Techniques



### Imperative commands

User operates directly on live objects in a cluster, operations are provided to kubectl as arguments or flags. For example:

kubectl create deployment nginx --image nginx

Pros:

* commands are simple and easy to learn
* requires only one step to deploy

Cons:

* does not integrate with change review process
* no audit trail
* no source of records apart from whats live
* no template for create new similar objects

### Imperative object configuration

kubectl command specifies the opertation (create, replace, etc) and provide at least one filename with object definintion. For example:

kubectl create -f nginx.yaml

Pros:

* Object configuration stored in source control such as Git
* Object configuration can be integrated into review processes
* Template for creating new similar objects

Cons:

* Knowledge of schema required
* Additional step to writing config file

### Declarative object configuration

User operates on object configuration files stored locally, however user does not define the operations to be taken on the files. The operations are automatically detected per-object by kubectl.

kubectl can display differences on a set of configs, then apply them. For example:

kubectl diff -f configs/

kubectl apply -f configs/

Pros:

* Changes made directly to live objects are retained, even if they are not merged back into configuration files
* Declarative object configuration has better support for operating on directories and automatically detecting operation types

Cons:

* Hardered to debug and understand when results are unexpected
* Partial updates using diffs create complex merge and patch operations

## Node management

Nodes can either self-register to the control plane or be manuallyed added as a node object. When nodes are created, kubernetes create a node object internally, checks it has been registered on the API server, and if it is healthly it can then start running pods.

# Best Practice

## Config tips

* define configs with lastest stable API version
* store config files in git before deploying to allow for easy rollbacks
* group related configs into single file
* call kubectl on directory to apply many configs
* do not specify default values unnecessarily
* put object descriptions in annotations
* use semantic labels such as: { app: myapp, tier: frontend, phase: test, deployment: v3 }

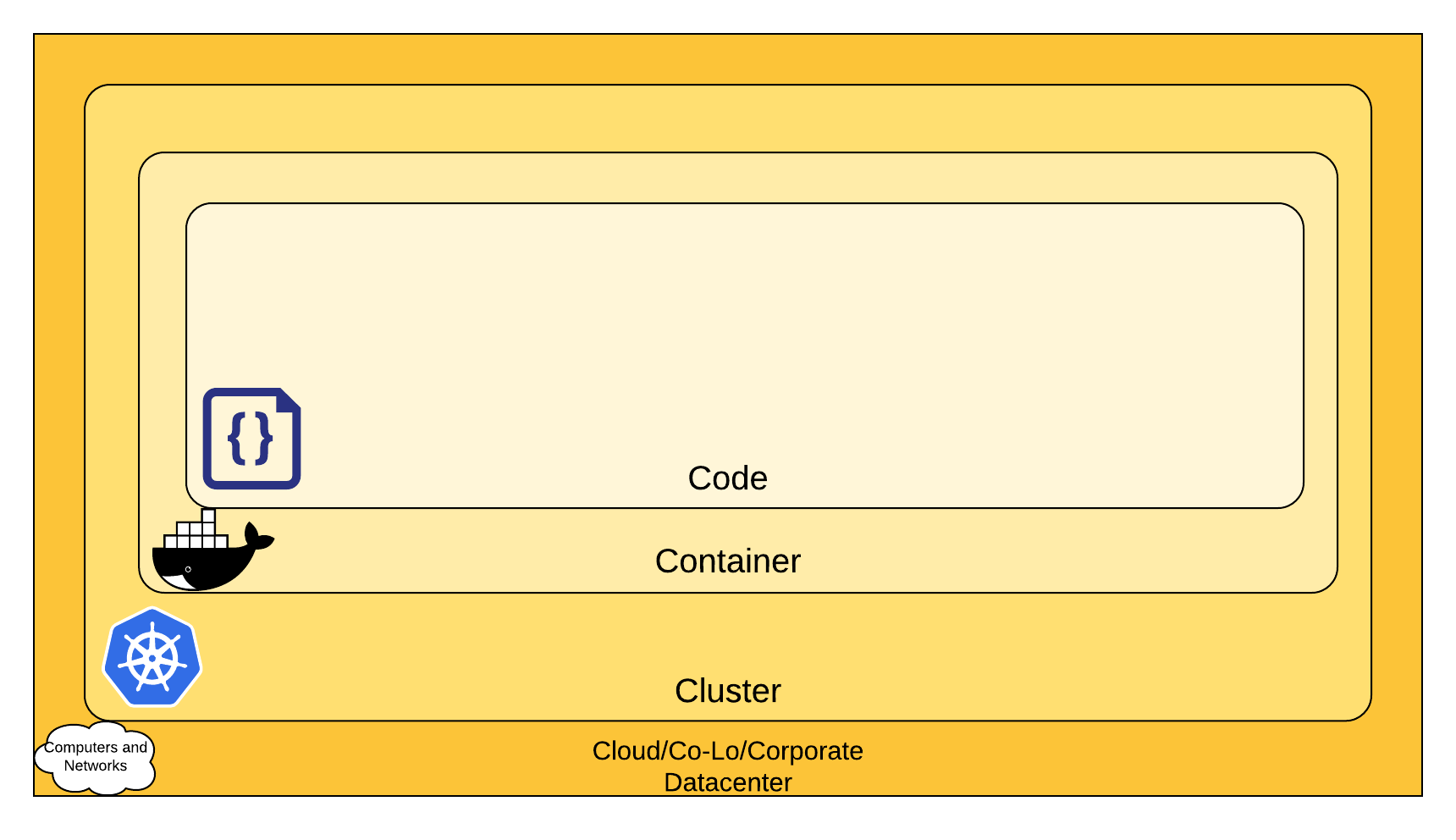
## Services

* Create a service before its backend workloads so env variables are provided to started containers
* Use cluster DNS server which listens for new services and allows access via DNS record
* use headless services when you don’t need kube-proxy load balancing

## Containers

* use imagePullPolicy appropriately

# Security



### Cloud

Cloud providers provide security documentation for how best to manage secuity in kubernetes infastructure. Main areas of concern are:

* Networks access to API Server - All access to the control plane should not be allow publically on the internet and should be restricted to the set of IP address needed to administer the cluster
* Network access to nodes - Nodes should be configured to only accept connections from the control plane on specific ports, generally nodes should not be exposed directly on the internet
* Kubernetes access to cloud provider API - Control plane should have minimum privilidge possible to complete its role
* Acces to etcd - only the control plane should be able to access the kubernetes etcd datastore
* etcd encryption - if possible encrypt all drives as rest, bare minimum etcd should be encrypted

### Cluster

Secure the components of a cluster:

<https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/>

Secure components in a cluster:

* RBAC Authorization
* Authentication
* Application secrets management
* Pod security
* QoS
* Network policys
* TLS of kubernetes ingress

### Container

Use docker best practices:

* Container vulnerability scanning and OS dependency security (i.e. scan container during build for known vulnerabilities)
* Image signing and enforcement
* Disallow privileged users - When constructing containers, create user that have the least level of os privilege possible for role

### Code

General code security practices:

* Access over TLS only
* Limited port ranges of communication - only expose nessecary ports
* regularly scan 3rd party dependencies
* static code analysis to check for unsafe coding practices
* Dynamic probing attacks