Kubernetes

Kubernetes Agenda

- A bit about Kubernetes
- Terminology
- K8S in depth
- Demos (For most sections)
- Hands on

What is Kubernetes?

- The meaning of `Kubernetes` is **Captain** in Greek
- `Kubernetes` is also referred to as K8S (K **ubernete** S)
- Open-source container-orchestration software
- Supply automation for deployment, scaling, operation and more
- `Kubernetes` is a platform which manage the containers for you (deploy health-check etc)
- Originally Developed by Google (2013) and v1.0 released to the public in 07/2015
- Was based upon Google `Borg` (Large Cluster Management) developed internally by Google
- Today is managed by the Cloud Native Computing Foundation (CNCF)
- The most contributed project by Unix community after the Linux Kernel

What is Kubernetes? [batteries-included]

- Think of Kubernetes like a higher-level language for your application architecture.
  - You describe what you need and it tries to resolve it
  - K8S acts as an engine to **resolve state** using the **abstracted resources** to **deploy and manage** your application.
- It is `declarative`, and not `imperative`.
- You tell it or “declare” what you want, and it figures out the rest.

`imperative`	you tell it what you want it **in every step**
`declarative`	you tell it what you want it to be, and **it figures it out**

- K8S gives you: **self-healing** and **seamless upgrading or rollback** of applications

What can Kubernetes REALLY do?

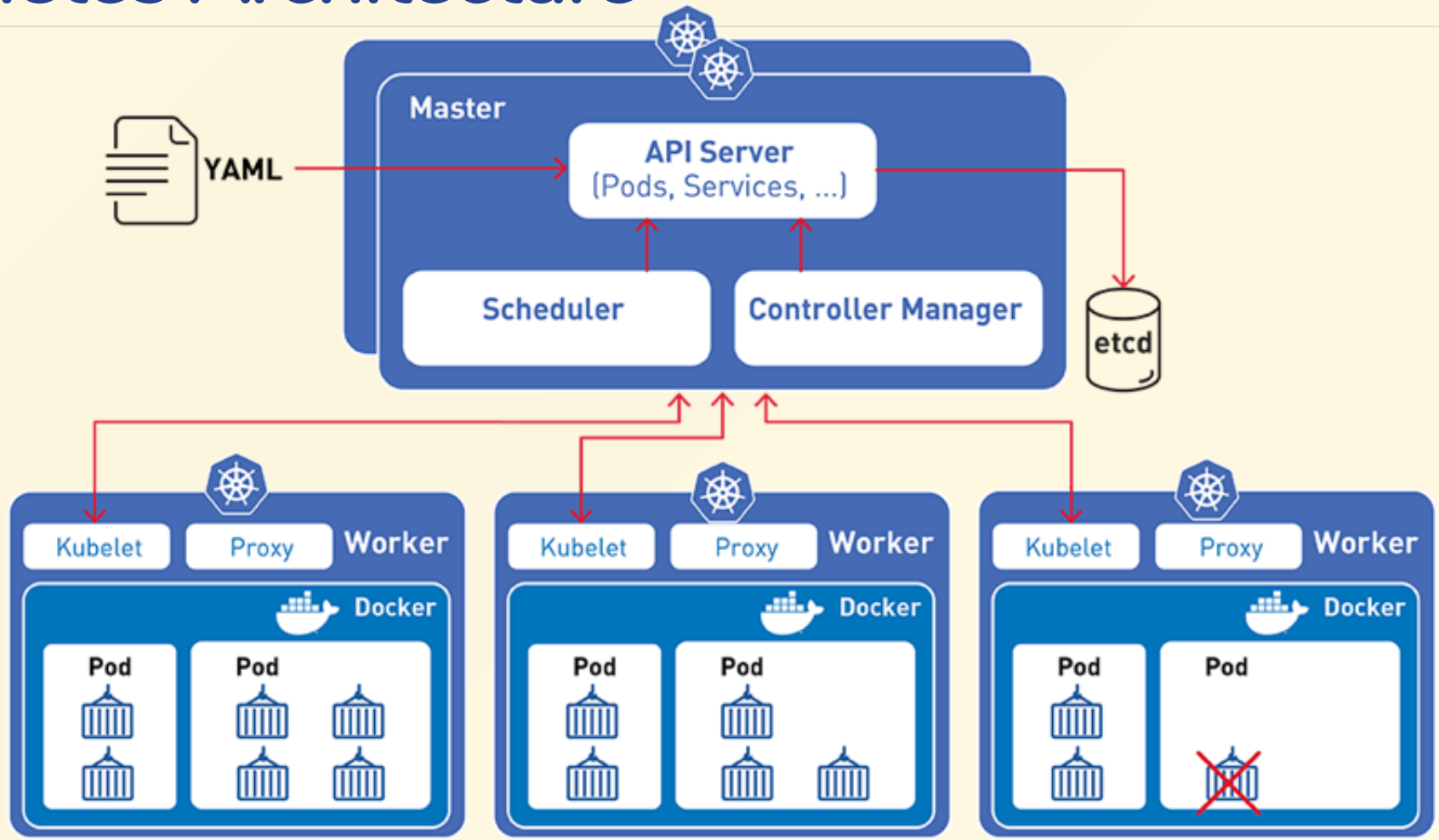
- Autoscale Workloads
- Blue/Green Deployments
- Manage Stateless and Stateful Applications
- Provide native methods of service discovery (replaceable)
- Easily integrate and support 3rd party apps (ex Helm, containers)
- **The killing feature:**

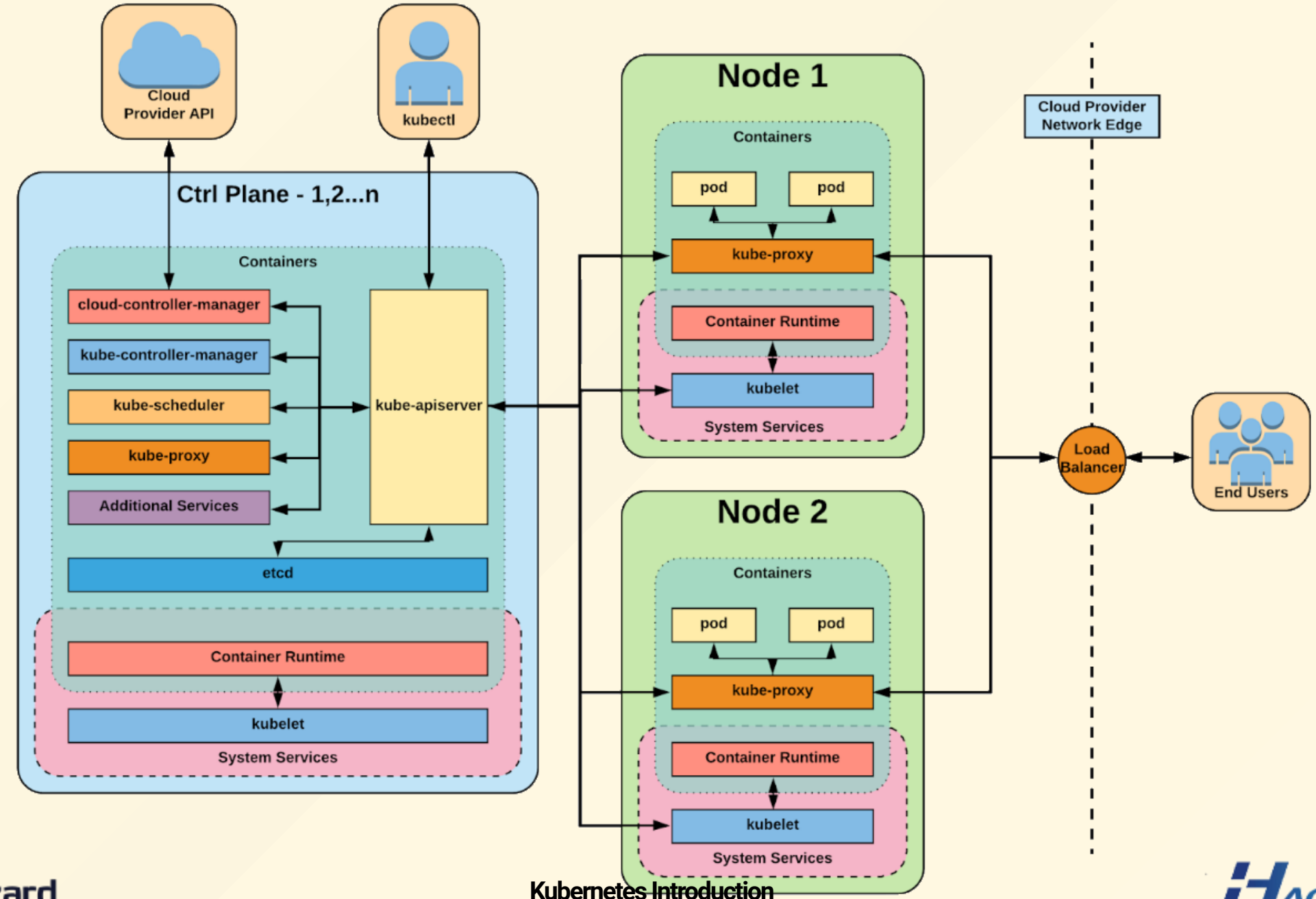
  Use the SAME API across everywhere .... (bare metal & cloud providers)

Some Numbers

- Over 61,000 stars on Github
- ~2400 Contributors to K8s Core
- Most discussed repo on github
- 1000+ pull requests
- ~90,000 commits !!!!

Kubernetes Architecture





Kubernetes resources

- The Kubernetes API defines a set of resources
- Resources are organized by type, `kind`
  Some common resource `kinds` are:

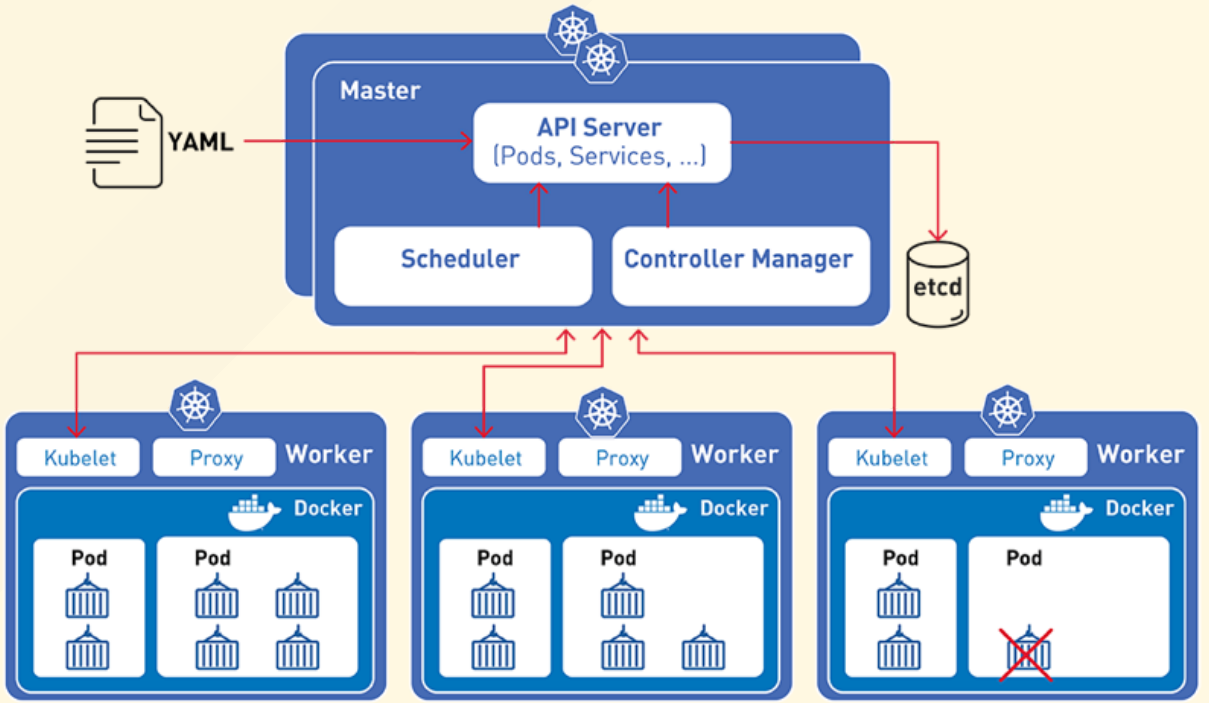
Kind	Description
`node`	a machine — physical or virtual — in our cluster
`pod`	group of containers running together on a host
`service`	stable network endpoint to connect to one or multiple containers
`namespace`	isolated group of things
`replicaset`	set of containers which can be scaled
`secret`	sensitive data to be passed to a container

Terminology / Main features

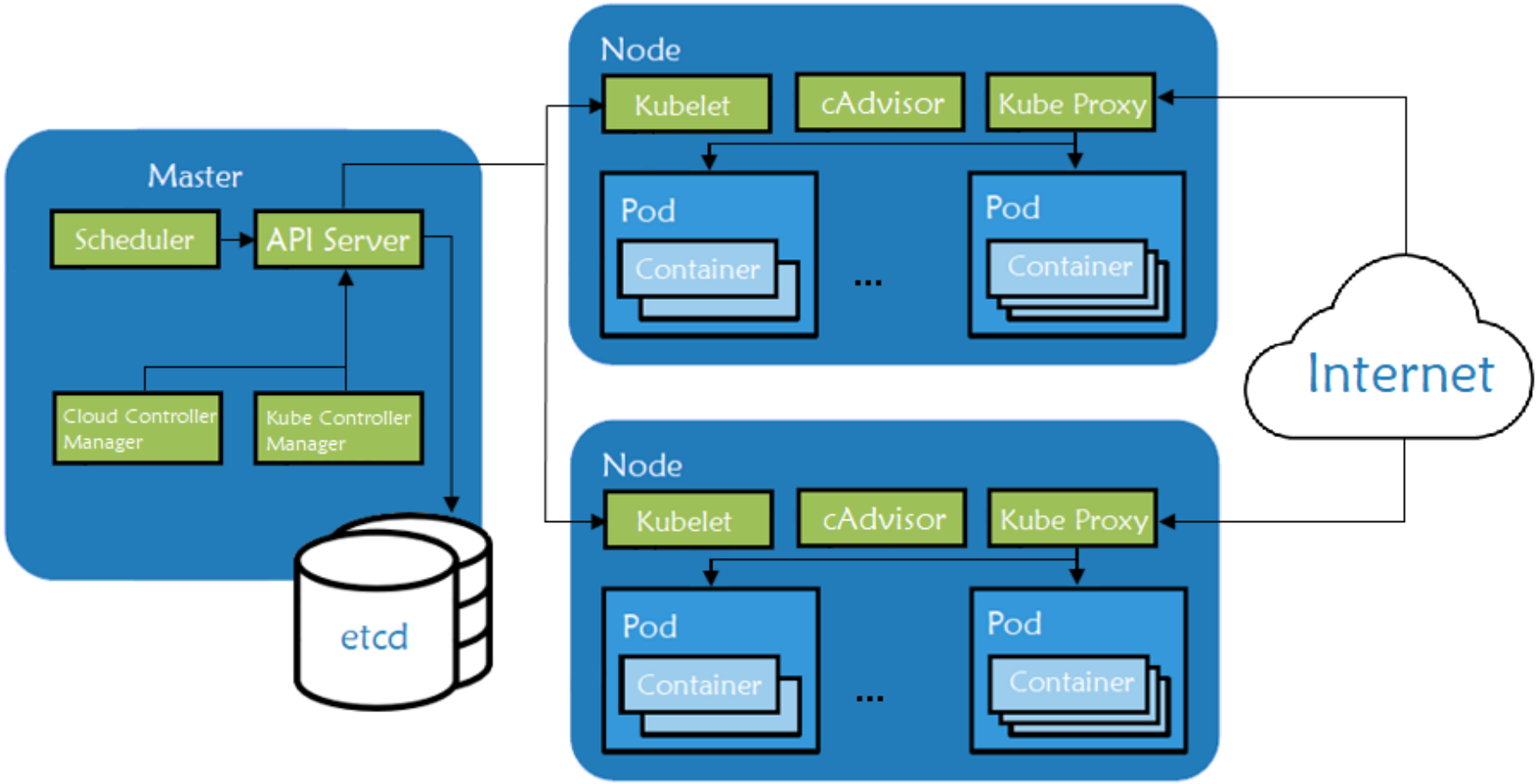
- Cluster
- Nodes / Master node
- Pod
- Replica Sets / Deployments
- Services
- Volumes
- Namespaces
- ConfigMaps and Secrets
- Labels and selectors
- etcd

Terminology - Cluster

- When you deploy Kubernetes, you get a `cluster`.
- A `cluster` is a **set of machines**, called **nodes**, that run containerized applications managed by Kubernetes



Terminology - Nodes (architecture)



Terminology - Nodes

- A node is a `worker machine` in Kubernetes, previously known as a **minion**.
- A node may be a VM or physical machine, depending on the cluster
- The services on a each node include the container runtime, kubelet and kube-proxy
- k8s has a unique node named **Master Node** (Master Components)
- Node is a top-level resource in the Kubernetes REST API
- Each node contains the services necessary to run pods and is managed by the master components
  - Container Engine (typically Docker)
  - `kubelet` (aka “node agent”)
  - `kube-proxy` (a necessary but not sufficient network component)

Terminology - Master Node (Master Components)

- **Master components** in Kubernetes is a collection of services
- **Master components** provide the cluster's control plane
- **Master components** manage the cluster (ex: scheduling, deploying)
- **Master components** can run on any node but usually deployed to a dedicated node
- The can be multiple masters nodes for high availability
- The main components of the **Master Node** (in addition to worker node components) are:
  - kube-apiserver
  - etcd (a highly available key/value store; the `"database"` of Kubernetes)
  - kube-scheduler
  - kube-controller-manager
  - Node Controller

Terminology - Pod

- The most basic component of k8s
- Pod is a **group of containers** (one or more) which is **deployed** to the host machine
- Pod describe the application running on k8s
- **Pods serve as unit of deployment for horizontal scaling, and replication**
- Each pod get its unique ip in the cluster
- Pods can communicate with each other using the Pod IP address,
  Developer should use Services and not Pod IP
- Pods can share storage (volumes), network, host resources (cpu, memory)
- Pods are managed through the **k8s API**
- Pods run in a shared context (Linux namespaces, cgroups)
- Applications with in the same pod can use *localhost* for communication

Terminology - Pod (sample - yaml file)

- `kind: pod`

```yaml
apiVersion: v1
kind: Pod <-----------------------------------<<
metadata:
  name: pod1
  labels:
    tier: frontend
spec:
  containers:
  - name: hello1
    image: gcr.io/google-samples/hello-app:2.0
    imagePullPolicy: Always
    command: ["echo", "SUCCESS"]
```

```
# pull the image and create a container
$ kubectl create -f <file name>
```

Basic pod commands

```
# Get k8s kinds (resources) like pods, deployments, replicateSets
kubectl get <...>

# Describe the given pod
kubectl describe pod <pod>

# Create new service (= expose new port)
kubectl expose pod <pod> --port=<port> --name=<name>

# Port forward from the pod to the host
kubectl port-forward <pod> <port>

# Execute command on the pod
kubectl exec <pod> --command ...

#Execute interactive pod, run pod with shell - !!!! Usefull for debugging pods
kubectl run -i -tty <name> --image=... --restart=Never --sh
```

Terminology - Replica Sets

- The aim of `ReplicaSet` is to maintain a stable set of replica Pods running at any given time
- `ReplicaSet` is a collection of definitions which specify the pods.
- It contains pod templates for creating or updating new pods.
- In production its much better to use `Deployment` than `ReplicaSet`, deployment has more features
- To get the ReplicaSet information:

```
kubectl get rs
```

Terminology - Replica Sets

```
apiVersion: apps/v1 # our API version
kind: ReplicaSet   # The kind we are creating
Metadata: # Specify all Metadata like name, labels
  name: some-name
  labels:
    app: some-App
    tier: some-Tier
Spec:
  replicas: 3 # Here is where we tell k8s how many replicas we want
  Selector: # This is our label selector field.
    matchLabels:
      tier: some-Tier
    matchExpressions:
      - {key: tier, operator: In, values: [some-Tier]} # we are using the set-based operators
  template:
    metadata:
      labels:
        app: some-App
        tier: someTier
    Spec: # This spec section should look like spec in a pod definition
      Containers:
```

Terminology - Deployments

- `Deployment` is the pod(s) specifications (labels, replicas, network etc)
- `Deployment` provides declarative updates for `Pods` and `ReplicaSets`.
- `Deployment` control and update the state of the Pods in the ReplicaSet
- Main `Deployment` use cases
  - Update ReplicaSet for rollback
  - Update Pods (add/remove/update)
  - Scale up/down
  - Clean old ReplicaSet

Terminology - Deployments

- `Deployment` ensures that only a **certain number of Pods are down** while they are being created or updated.
- Every time deployment is noticing changes, a `ReplicaSet` is created to reflect the changes to the desired Pods
- During `update`,
  by default, at least 75% of the desired number of Pods are up (25% max unavailable).
- During `creation`,
  by default, at most 125% of the desired number of Pods are up (25% max surge).

Terminology - Deployments (yaml fields)

revisionHistoryLimit

- The number of previous iterations of the Deployment to retain.

strategy:

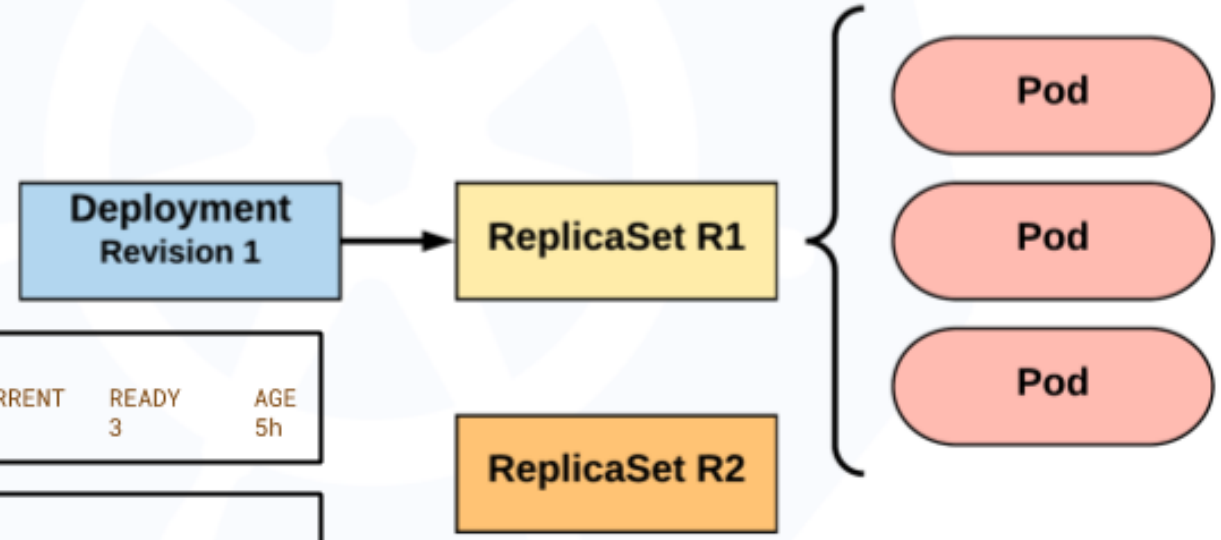
- Describes the method of updating the Pods based on the type.
- Valid options are Recreate or RollingUpdate.

Recreate	All existing Pods are killed before the new ones are created.
RollingUpdate	Cycles through updating the Pods according to the parameters: maxSurge and maxUnavailable.

RollingUpdate Deployment

Updating pod template generates a new **ReplicaSet** revision.

R1 pod-template-hash:
676677fff
R2 pod-template-hash:
54f7ff7d6d



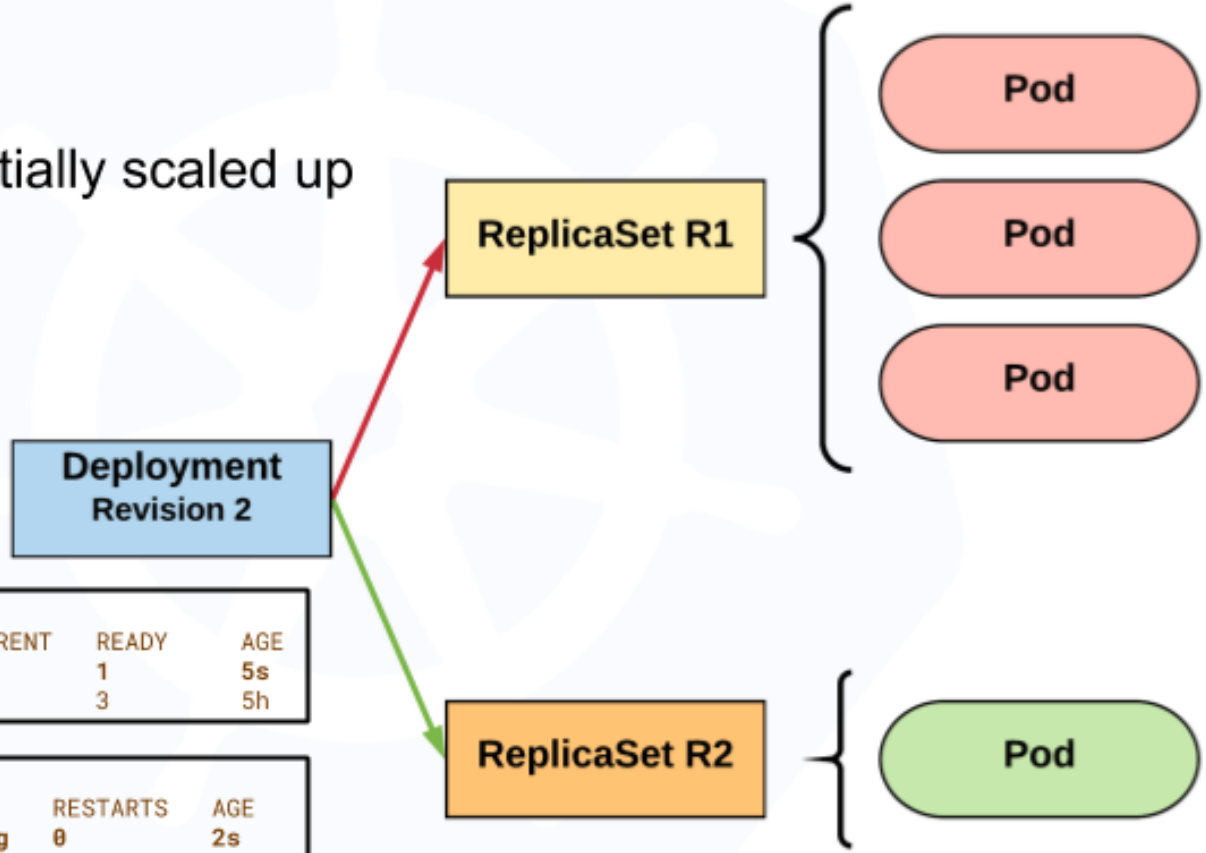

```
$ kubectl get replicaset
NAME                  DESIRED   CURRENT   READY     AGE
mydep-6766777fff      3         3         3         5h
```

```
$ kubectl get pods
NAME                        READY     STATUS    RESTARTS   AGE
mydep-6766777fff-9r2zn      1/1       Running   0          5h
mydep-6766777fff-hsfz9      1/1       Running   0          5h
mydep-6766777fff-sjxhf      1/1       Running   0          5h
```

RollingUpdate Deployment

New **ReplicaSet** is initially scaled up based on `maxSurge`.

R1 pod-template-hash:
`676677fff`
R2 pod-template-hash:
`54f7ff7d6d`



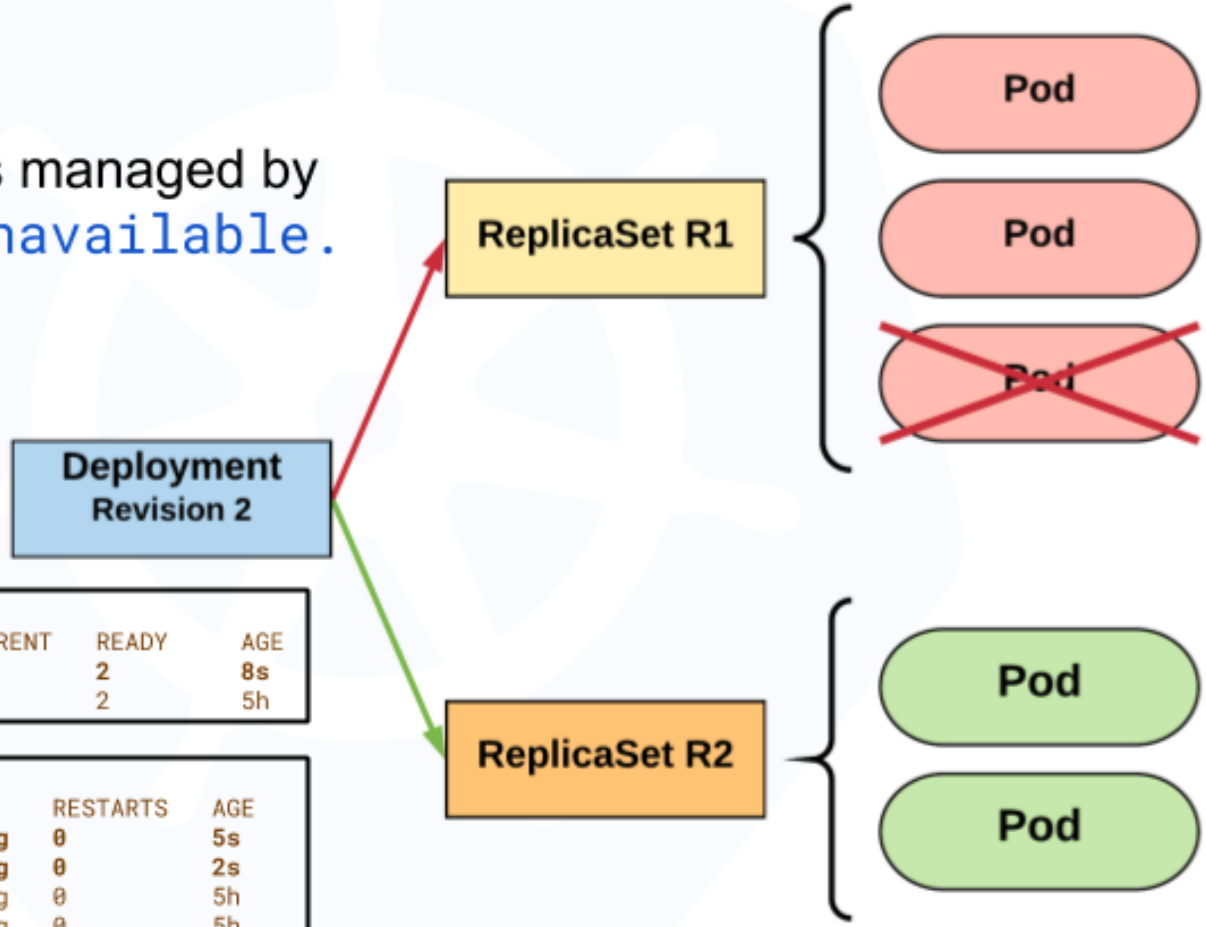

```
$ kubectl get replicaset
NAME                  DESIRED   CURRENT   READY   AGE
mydep-54f7ff7d6d      1         1         1       5s
mydep-6766777fff      2         3         3       5h
```

```
$ kubectl get pods
NAME                        READY   STATUS    RESTARTS   AGE
mydep-54f7ff7d6d-9gvll      1/1     Running   0          2s
mydep-6766777fff-9r2zn      1/1     Running   0          5h
mydep-6766777fff-hsfz9      1/1     Running   0          5h
mydep-6766777fff-sjxhf      1/1     Running   0          5h
```

RollingUpdate Deployment

Phase out of old Pods managed by `maxSurge` and `maxUnavailable`.

R1 pod-template-hash:
`676677fff`
R2 pod-template-hash:
`54f7ff7d6d`



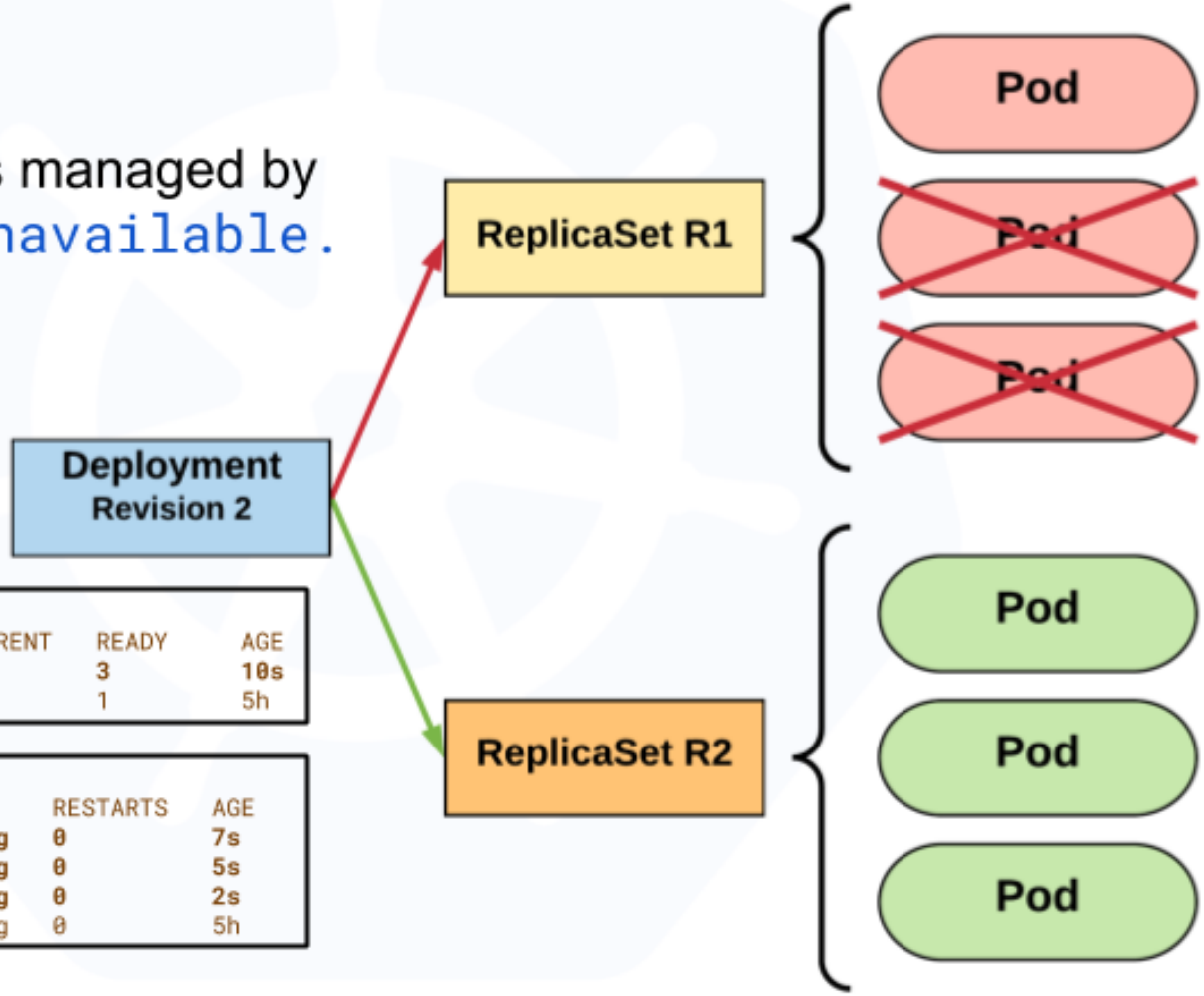

```
$ kubectl get replicaset
NAME                  DESIRED   CURRENT   READY   AGE
mydep-54f7ff7d6d      2         2         2       8s
mydep-6766777fff      2         2         2       5h
```

```
$ kubectl get pods
NAME                        READY   STATUS    RESTARTS   AGE
mydep-54f7ff7d6d-9gvll      1/1     Running   0          5s
mydep-54f7ff7d6d-cqvlq      1/1     Running   0          2s
mydep-6766777fff-9r2zn      1/1     Running   0          5h
mydep-6766777fff-hsfz9      1/1     Running   0          5h
```

RollingUpdate Deployment

Phase out of old Pods managed by `maxSurge` and `maxUnavailable`.

R1 pod-template-hash:
`676677fff`
R2 pod-template-hash:
`54f7ff7d6d`



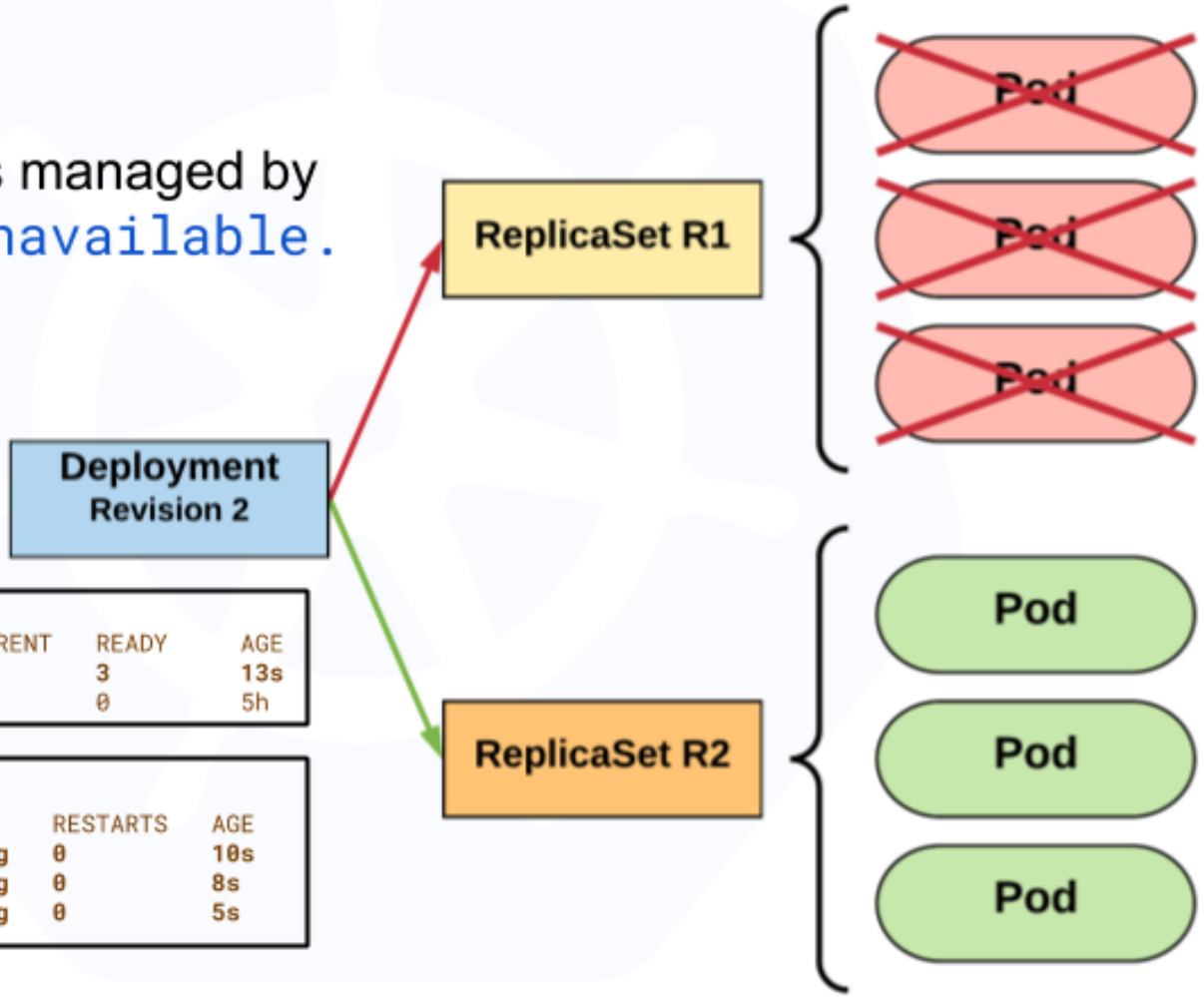

```
$ kubectl get replicaset
NAME                  DESIRED   CURRENT   READY   AGE
mydep-54f7ff7d6d      3         3         3       10s
mydep-6766777fff      0         1         1       5h
```

```
$ kubectl get pods
NAME                        READY   STATUS    RESTARTS   AGE
mydep-54f7ff7d6d-9gvll      1/1     Running   0          7s
mydep-54f7ff7d6d-cqvlq      1/1     Running   0          5s
mydep-54f7ff7d6d-gccr6      1/1     Running   0          2s
mydep-6766777fff-9r2zn      1/1     Running   0          5h
```

RollingUpdate Deployment

Phase out of old Pods managed by `maxSurge` and `maxUnavailable`.

R1 pod-template-hash:
676677fff
R2 pod-template-hash:
54f7ff7d6d



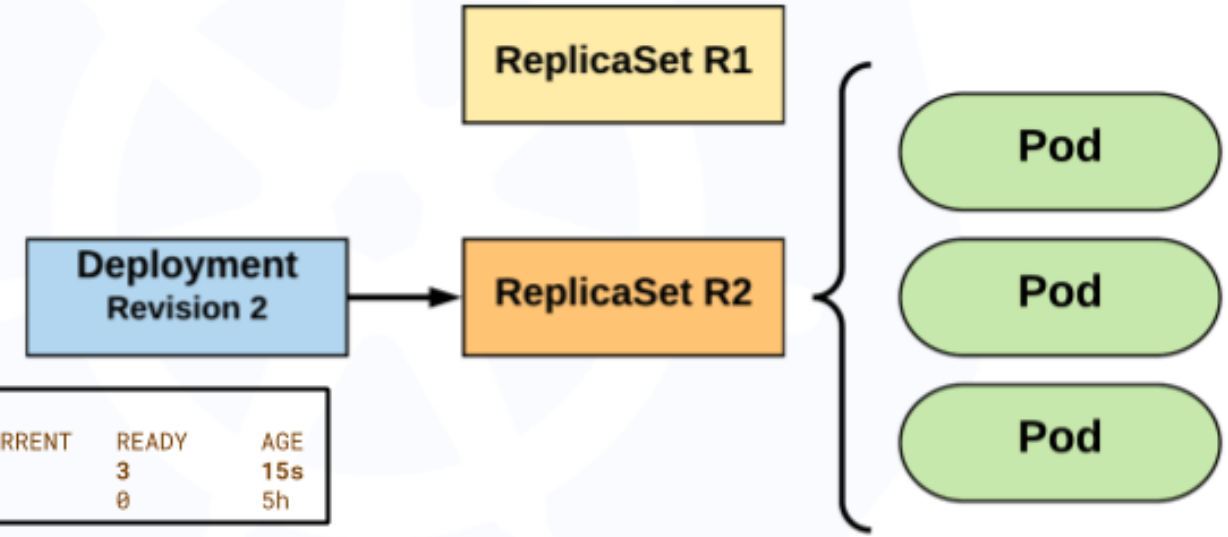

```
$ kubectl get replicaset
NAME                  DESIRED   CURRENT   READY   AGE
mydep-54f7ff7d6d      3         3         3       13s
mydep-6766777fff      0         0         0       5h
```

```
$ kubectl get pods
NAME                  READY   STATUS    RESTARTS   AGE
mydep-54f7ff7d6d-9gvll   1/1     Running   0          10s
mydep-54f7ff7d6d-cqvlq   1/1     Running   0          8s
mydep-54f7ff7d6d-gccr6   1/1     Running   0          5s
```

RollingUpdate Deployment

Updated to new deployment revision completed.

R1 pod-template-hash:
676677fff
R2 pod-template-hash:
54f7ff7d6d




```
$ kubectl get replicaset
NAME                  DESIRED   CURRENT   READY   AGE
mydep-54f7ff7d6d      3         3         3       15s
mydep-6766777fff      0         0         0       5h
```

```
$ kubectl get pods
NAME                     READY   STATUS    RESTARTS   AGE
mydep-54f7ff7d6d-9gvll   1/1     Running   0          12s
mydep-54f7ff7d6d-cqvlq   1/1     Running   0          10s
mydep-54f7ff7d6d-gccr6   1/1     Running   0          7s
```

Terminology - Deployments

- Create X Deployments units of nginx

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

Terminology - Deployments

- Use Deployment

```
kubectl scale deployment.v1.apps/nginx-deployment --replicas=10 --record=true
```

- Get details of your Deployment:

```
kubectl describe deployments
```

Terminology - Services

- A `Service` is an **abstraction** which defines a
  - **Logical set** of Pods and a
  - **Policy** by which to access them (aka micro-service).
- Every service in K8S is actually a Load Balancer
- The set of Pods is accessed by a selector
- A `Service` in Kubernetes is a REST object
- A `Service` is responsible for enabling network access to a set of pods.
  - Pods can connect each other via direct network requests, while `Service` is used for exposing services to the "world"

Terminology - Services

- Kubernetes assigns Service an IP address (aka “cluster IP”), which is used by the `Service` proxies
- `Services` "expose" the pods to the world
- `Services` serve as load balance layer
- Every node in a Kubernetes cluster runs a `kube-proxy`.
  kube-proxy is responsible for implementing a form of virtual IP for Services
- Services can define multiple ports and event static IPs
- When we deploy pods k8s they get the desired labels and the Service use selectors to match those labels (pods label).
- When deployemnt is changes the service is auto updated and reflacting those changes (for example if we add more pods)

Terminology - Services Ingress

- An ingress is a **host name and path mapping** to Service
- An Ingress is an object that allows access to your Kubernetes services from outside the Kubernetes cluster.
- Ingress is configured by creating a **collection of rules** that define which inbound connections reach which services.
- An ingress is usually but not only **LoadBalancer(s)** for exposing Node services.
- A basic deployment is usually build upon:
  - Deployment
  - Service
  - Ingress rules

Terminology - Services

```yaml
kind: Service
apiVersion: v1

metadata:
  name: hostname-service

spec:
  type: NodePort
  selector:
    app: echo-hostname
  ports:
    - nodePort: 30163
      port: 8080
      targetPort: 80
```

Make the service available to network requests from external clients

Forward requests to pods with label of this value

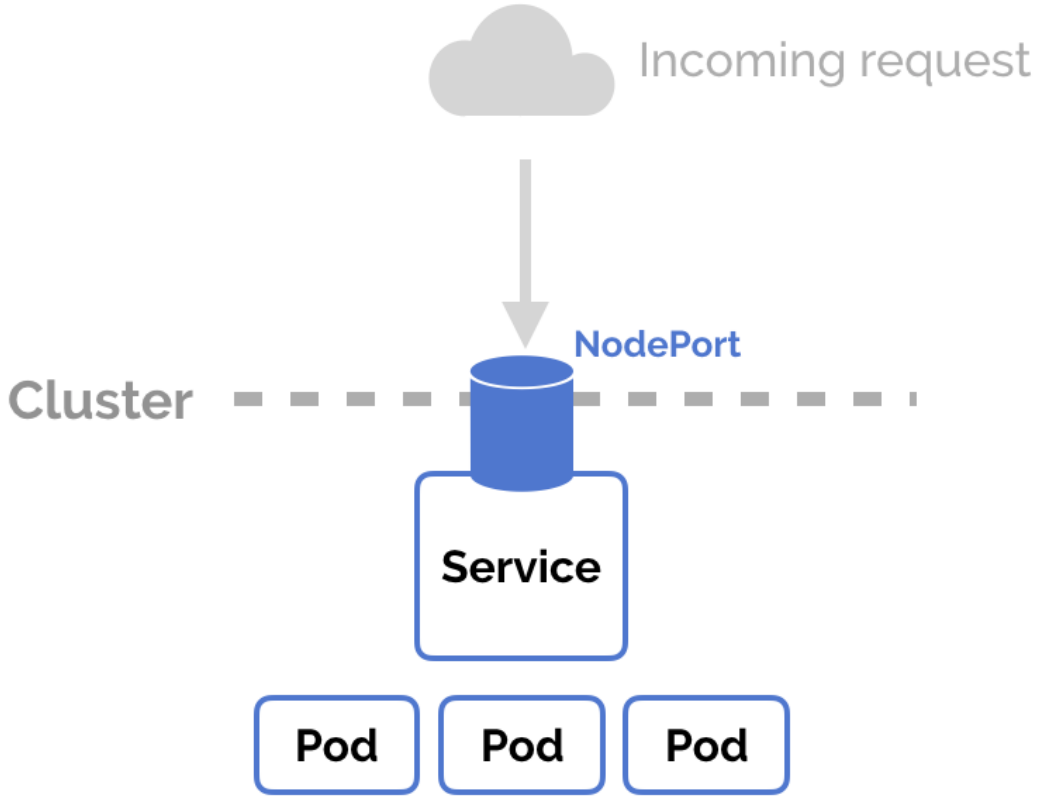
nodePort
access service via this external port number

port
port number exposed internally in cluster

targetPort
port that containers are listening on

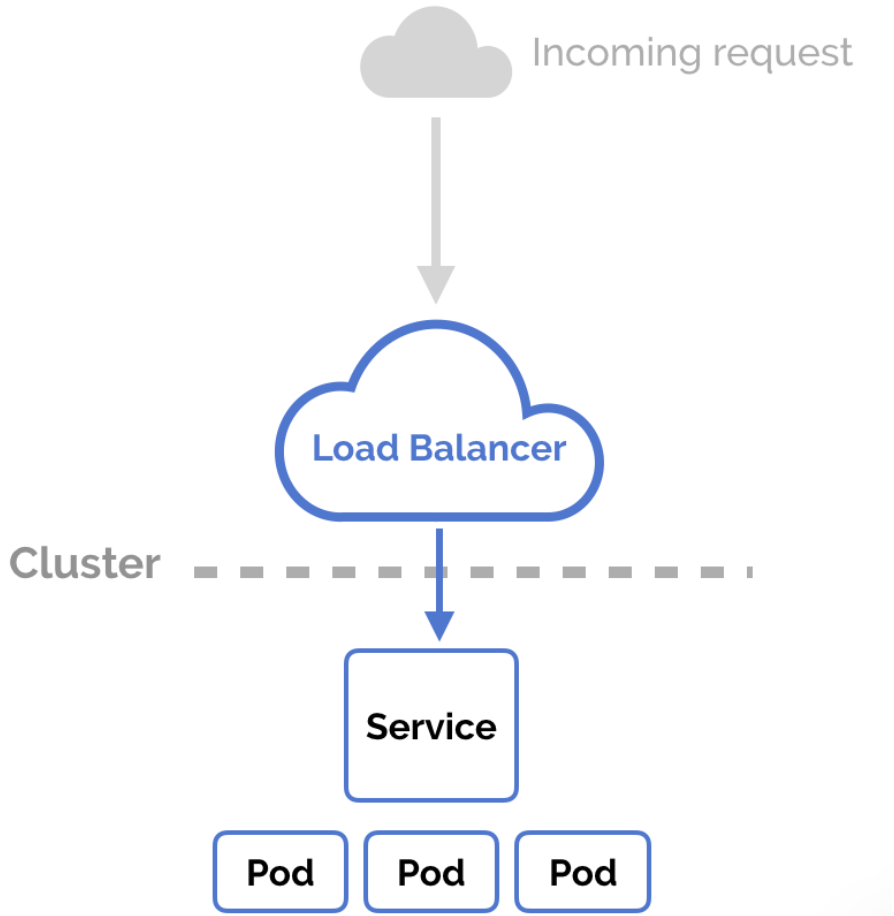
Terminology - Services NodePort

- `NodePort` is a configuration setting you declare in a service's YAML.
- Set the service spec's type to `NodePort`. Then, Kubernetes will allocate a specific port on each Node to that service, and any request to your cluster on that port gets forwarded to the service.



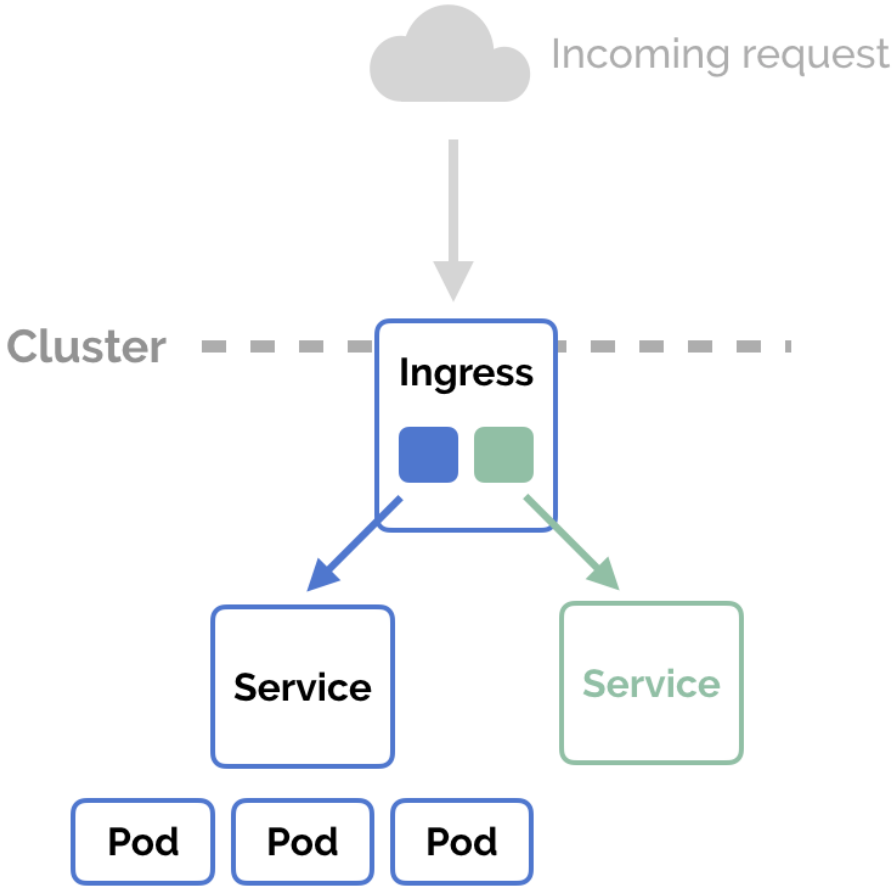
Terminology - Services LoadBalancer

- `LoadBalancer` is a usually a cloud provider LoadBalancer.
- Every time you want to expose a service to the outside world, you have to create a new LoadBalancer and get an IP address.



Terminology - Services Ingress

`Ingress` Will be covered later on in the network section



Terminology - Services yaml

```yaml
kind: Service
apiVersion: v1
metadata:
  name: hostname-service
spec:
  # Expose the service on a static port on each node so that we can access the service from outside the cluster
  type: NodePort

  # When the node receives a request on the static port (30163) "select pods with the label 'app' set to 'echo-hostname'"
  # and forward the request to one of them
  selector:
    app: echo-hostname

  # Three types of ports for a service
  #    - nodePort - a static port assigned on each the node
  #    - port - port exposed internally in the cluster
  #    - targetPort - the container port to send requests to
  ports:
    - nodePort: 30163
      port: 8080
      targetPort: 80
```

Terminology - Services (ServiceTypes)

Kubernetes ServiceTypes allow you to specify what kind of Service you want.

Node types	Description
`ClusterIP` [default]	Exposes the Service on a cluster-internal IP, which makes the Service only reachable from within the cluster.
`NodePort`	Exposes the Service on each Node's IP at a static port (the NodePort).
`LoadBalancer`	Exposes the Service externally using a cloud provider's load balancer. NodePort and ClusterIP Services, to which the external load balancer routes,are automatically created.
`ExternalName`	Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record

Terminology - Volumes

- In Docker volumes are set of mapping paths from the host to the container and vice versa
- In k8s, volumes has an explicit lifetime - the same as the Pod that encloses it.
  For example: when Pod ceases to exist, the volume will cease to exist as well.
- k8s supports multiple types of volumes
  - My Favorite: **gitRepo (deprecated)**

Terminology - Namespaces

- Kubernetes supports multiple virtual clusters deployed on the same physical cluster.
  (multiple separate sub groups)
- These virtual clusters are called `namespaces`.
- Namespaces are a way to divide cluster resources between multiple users (via resource quota).
- Namespaces provide a scope for names.
  Names of resources need to be unique within a namespace
- Namespaces can not be nested inside one another and each Kubernetes resource can only be in one namespace.

Terminology - Namespaces

- Kubernetes starts with three initial namespaces:

Namespaces	Description
`default`	The default namespace for objects with no other namespace
`kube-system`	The namespace for objects created by the Kubernetes system
`kube-public`	This namespace is created automatically and is readable by all users (including those not authenticated).

Terminology - ConfigMaps / Secrets

- `ConfigMaps` are used to pass and share key value pairs between pods
- `Secrets` are similar to config maps and store sensitive data

```
# Create CongifMap explicitly
kubectl create configmap my-password --from-literal='password=123'

# Create CongifMap from properties files
kubectl create configmap <cofig name> --from-file=<folder name>

# View config maps
kubectl get configmaps
```

Terminology - ConfigMaps / Secrets

- Create CongifMap from file

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
    name: config-value01
    namespace: default
data:
    day: 'Sunday'
    year: '2019'
    rainning: 'false'
```

```
# Create the config maps
kubectl create -f configmap.yaml
```

Terminology - ConfigMaps / Secrets

- Read the config maps
- `env:` - Select a specific variable we wish to use

```
...
env:
  - name: week-day
    valueFrom: <-------------------------------
      configMapKeyRef:
        name: <.....>
        key: <.....>
```

- `envFrom` - Select all the defined variables (in case we have multiple ones)

```
...
envFrom:
  ConfigMapRef:
    name: <.....>
```

Terminology - ConfigMaps / Secrets

> Note: Unlike ConfigMaps you must also specify the **type** of Secret you are creating.

There are 3 types:

Type	Description
docker-registry	Credentials used to interact with a container registry.
generic	Equivalent to Opaque. Used for unstructured data.
tls	A TLS key pair (PEM Format) that accepts a cert (--cert) and key (--key).

Terminology - ConfigMaps / Secrets

- Define secrets

```
# Create files needed for the secrets
echo -n 'admin' > ./username.txt
echo -n 'secret' > ./password.txt

# Create the secrets from the 2 seperate files
kubectl create secret generic app-user-pass --from-file=./username.txt --from-file=./password.txt

# Verify that the secret is created
kubectl get secrets
```

Terminology - Labels and selectors

- Labels are `key/value` pairs which attached and identify objects and resources in k8s.
- Every object in k8s should have a label
- Labels are used to organize and to select subsets of objects
- Each Key must be unique for a given object.
- Same objects will have the same label(s), for example instances in replica will have the same label
- Labels are used to mark releases, environments, versions ....

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: label-demo
  labels:
    environment: production
    app: nginx
```

Terminology - Selectors

- Selectors use labels to filter or select objects.

```
apiVersion: v1
kind: Pod
metadata:
  name: example
  labels:
    app: app1
    env: prod
spec:
  containers:
  - name: app1
    image: app1/app2
    ports:
    - containerPort: 80
  nodeSelector:
    key: value
```

Terminology - Labels and selectors

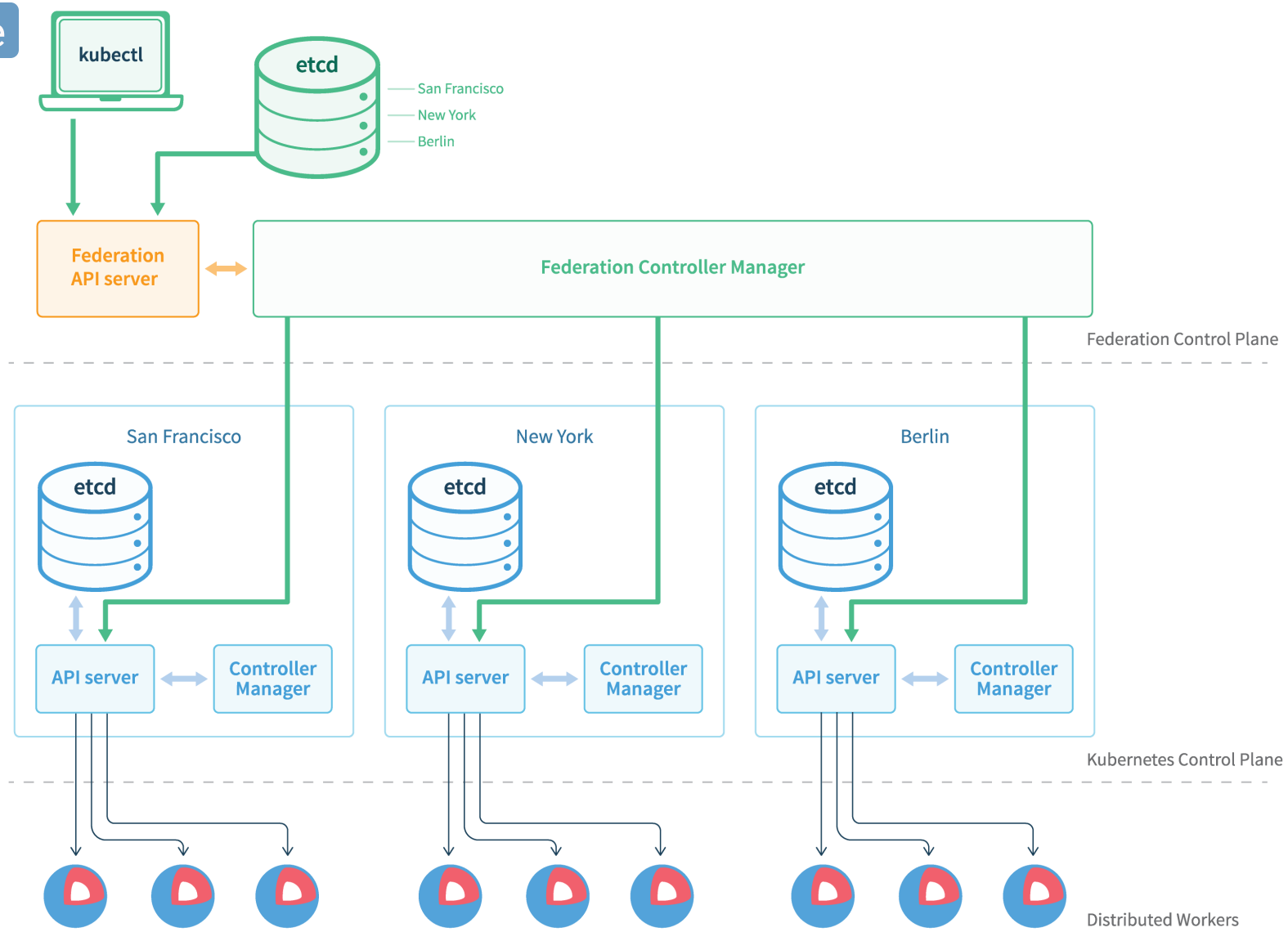
- Selectors types:
  - equality-based
  - set-based

```
apiVersion: v1                  | selector:           |  selector:
kind: Pod                       |   matchLabels:      |    matchExpressions:
metadata:                       |     gpu: nvidia     |    - key: gpu
  name: label-demo              |                     |      operator: in
  labels:                       |                     |      values: [“nvidia”]
    environment: production     |                     |
    app: nginx                  |                     |
```

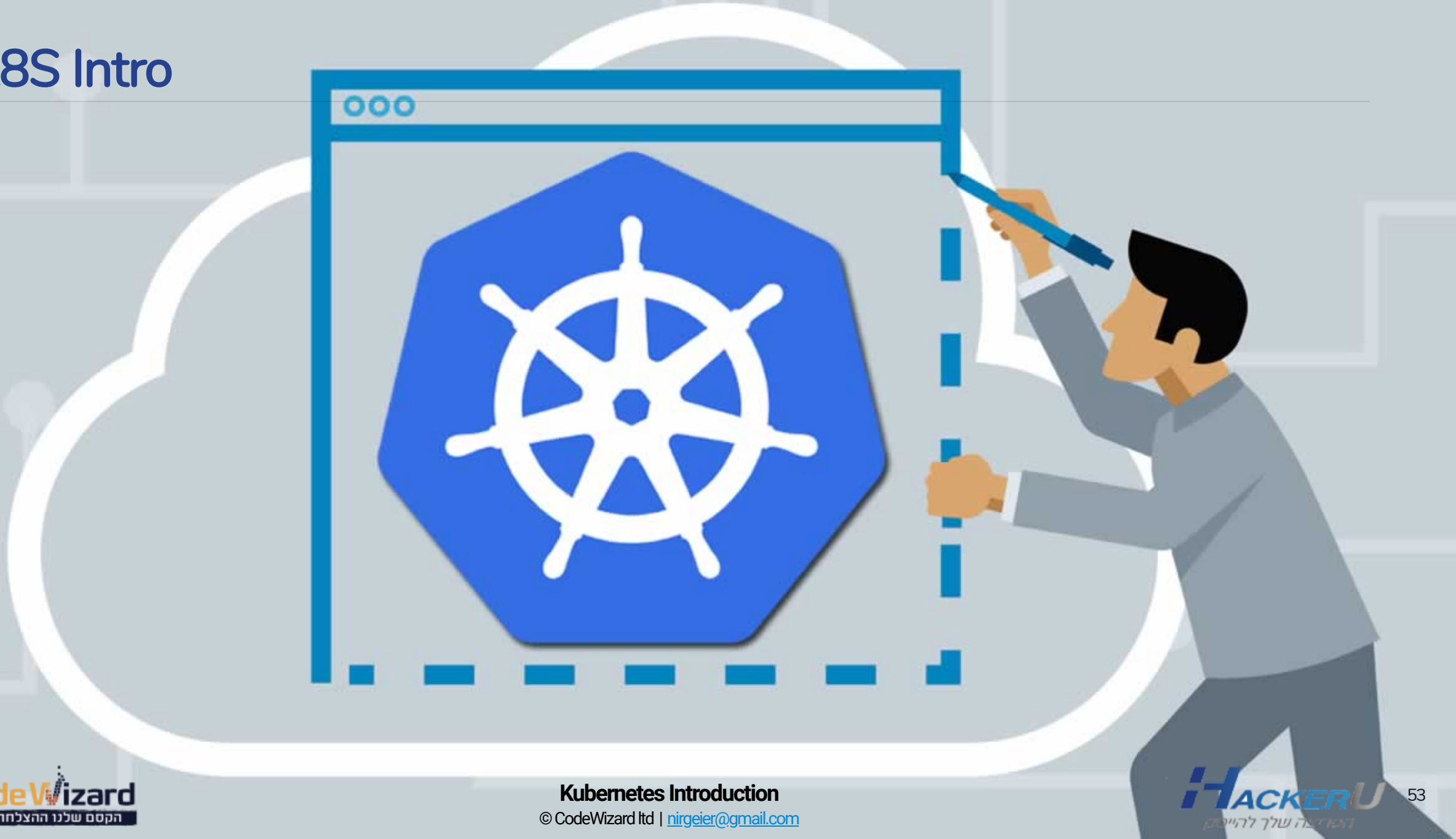
Terminology - etcd

- `etcd` is a **consistent** and **highly-available** key value store used by Kubernetes
- `etcd` is string all cluster data.
- In production environment etcd should be replicated (there are several ways to implement it)

etcd sample

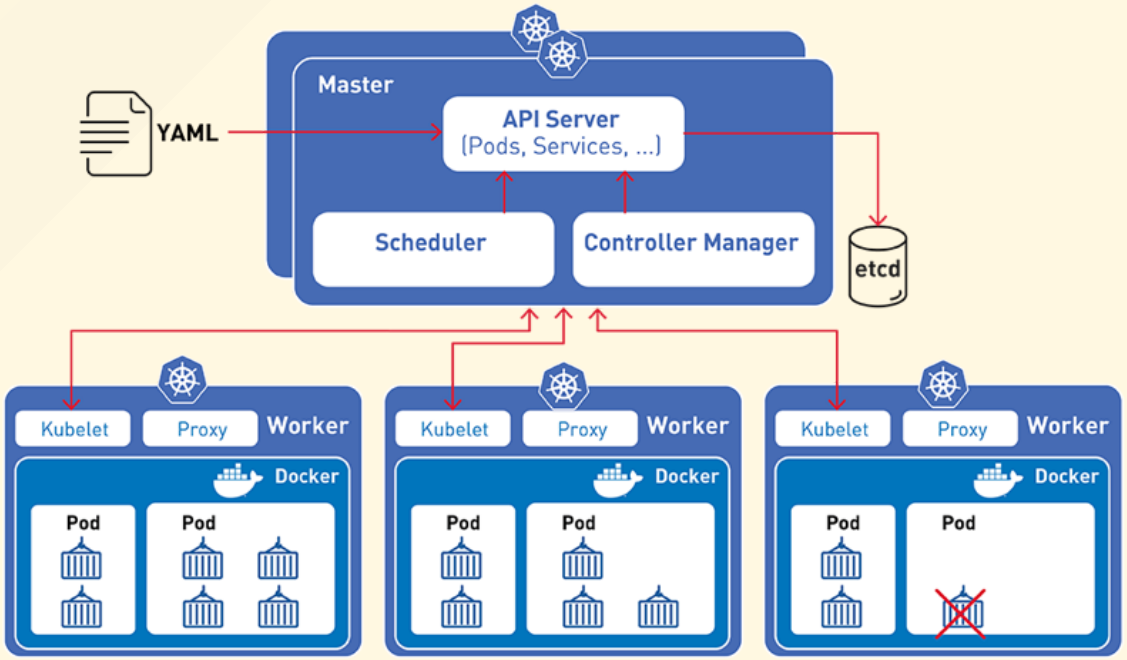

kubectl
etcd
San Francisco
New York
Berlin
Federation API server
Federation Controller Manager
Federation Control Plane
San Francisco
New York
Berlin
etcd
etcd
etcd
API server
Controller Manager
API server
Controller Manager
API server
Controller Manager
Kubernetes Control Plane
Distributed Workers

K8S Intro



How K8S works?

- K8s is build upon 4 main units
  - Pods
  - Services
  - Deployments / Replica sets
- User is interacting with the API
- K8S use yaml as the declarative language



How K8S works?

- User describe the requirements in a yaml file (`deployment`)
- The yaml is passed and processed by the API
- The K8S is responsible (auto managed) of creating, updating, scaling, healing and more
- The result is a cluster with the required objects
- The scheduler is responsible for the actual deployment
- In order to reach the pods (deployment units) we create `Service`.

How K8S Deployment works?

- Deployment describe the cluster
- When we change the deployment, k8s initiate a rollback.
  - K8S notice the change
  - K8S gradually updates the number of pods and not all of them on the same time or we will loose our service
  - Pods has a grace period when they are deleted (default 30s)
  - K8S use the following methods to define healthy pods:
    - liveness check
    - readiness check

How K8S Deployment works?

- K8S create new Pod
- K8S check for liveness
  The first check is liveness check. Once the check pass - which mean that the deploy passed and the pod is live and running it continue to the next phase
- K8S now check for readiness check. Once this check pass the pod is added to the LB
- If K8S need to delete old pods it will give it a `termination grace period` (default 30s) before its being removed to finish processing current requests
  - The pod is moved to `termination state`
  - Once the termination period passed the pod is being deleted
- K8S repeat this process for every required pods

How K8S Scheduler works? [kube-scheduler]

- The scheduler is the component which is responsible to deploy the required pod (on which machine)
- The scheduler is doing the following:
  - When we create a new pod the scheduler is "watching" for pods changes
  - It also monitor the state of all the machines (CPU, memory, storage etc)
- Once the scheduler notice a change it decide where to deploy the new pod based upon:
  - Predicates
  - Priorities

How K8S Scheduler works? [kube-scheduler]

- `Predicates`
  - Hard rule which must be accomplished like (OS, minimum memory etc). Collection of
    - System requirements (like memory)
    - User requirements (like - deploy only to VM with SSD drive)

How K8S Scheduler works? [kube-scheduler]

- `Priorities`

  Soft rules (recommendations) like:

  - It would be nice if there will be a single pod on every VM
  - It would be nice if you can deploy pods to different LB etc...
  - Priorities usually include rules like spreading, taint (the machine is not 100% but still we allow to deploy to it)
    For example: We prefer to deploy to a 100% percent healthy machine but if the only place left for deploy is a sick machine deploy anyway
  - We can define scoring to each rule

How K8S Scheduler works? [kube-scheduler]

- kube-scheduler selects a node for the pod in a 2-step operation:
  - Filtering
  - Scoring
  - **sort(filtering (nodes))** and choose the #1 node

How Volumes and Storage works in K8S.

- A `Volume` is an object in each pod
- A `Volume` is associated with the pod and mapped to a path in the container
- There are different volumes in K8S
  - EmptyDir - Temporary volume, the folder is on the same machine as the pod
  - Mounted storage, locally or in the cloud which are shared between pods
  - Persistence volume (NFS, disks) etc. This volume lives even when the pod dies
- In order to "combine" the different volume types K8S use `Persistence Volume Claim`

Persistence Volume Claim

- There are two ways PVs may be provisioned:
  - Statically
  - Dynamically.
- When Pod requires volume (storage) the K8S allocate (create) volume for each pod separately and mount them to the container.
- This allow more separately, flexibility between different storage types for example when we deploy to different cloud providers

Secrets Management

- Secret is used for storing certificates, passwords etc
- A secret is a key value pair
  - The value can be string or file content
- Secret can be stored in a volumes and we can declare specific secret volumes for serving sensitive data.
- Secret can also be an environment variable
- The content of the secret is stored in the etcd as plain text

Secrets Management

- In latest versions K8S added `Key management store integration` for using encryption keys to encrypt sensitive data (ex. Azure key-vault)
- In case we use `Key management store integration` the data is encrypted before its being stored in the etcd

Pods lifecycle

Pods lifecycle & Phases

- A pod is the atomic unit of scheduling
- When we create the pod the first phase is `Pending`.
  Pending means that the Pod is "ready" (ex: yaml parsed, object are ready) but not yet deployed (not schedules)
- The scheduler "find" home for the pod as explained earlier `sort(filter(nodes))`
  - `Creating` - If required, the image is pulled from the registry
  - `Running` - Once the image is pulled the pod is running. During this phase in case we have error (failure) K8S try to restart the container several times.

Pods lifecycle & Phases (Errors)

- In case of failure - when the retries doesn't work the the pod is pod state is changed to `CrashLoopBackOff`. K8S will try pause and might retry later.
- We can see the current states using `kubectl get pods`
- When pod stuck in `Pending` is usually due to lack of resources for deploying.
- When pod stuck in `Creating` is usually due to bad image
- `kubectl describe pods` will show the process of the pods

Pods lifecycle & Phases (Advanced)

- Hooks
  - `PostStart` : Hook which is executed right after the container starts
  - `PreStop` : Hook which is executed before the container dies
- Init container
  - Init containers, run **before** the app containers are started.
  - Init containers are exactly like regular containers, except:
    - Init containers always run to completion.
    - Each init container must complete successfully before the next one starts.

Pods (Advanced stuff - Probes)

- A Probe is a diagnostic performed periodically by the kubelet on a Container.
  There are three types of probes for checking the pods:
  - `ExecAction`: Executes a specified command inside the Container.
    The diagnostic is considered successful if the command exits with a status code of 0.
  - `TCPSocketAction`: Performs a TCP check against the Container's IP address on a specified port.
    The diagnostic is considered successful if the port is open.
  - `HTTPGetAction`: Performs an HTTP Get request.
    The diagnostic is considered successful if the response has a status code greater than or equal to 200 and less than 400.

How Configuration management works in K8S.

- K8S use a `ConfigMaps`
- `ConfigMaps` are key-value pairs and can be created using yaml or properties file(s)
- `ConfigMaps` can be used as file or as environment variables
- `ConfigMaps` & `Secrets` are like but K8S handle them differently
- We define the configuration during the deployment
- Configuration change execute a rollup

Monitoring & Alerting

- K8S expose many metrics for us (CPU, memory, disk etc)
- `Prometheus` is the leading tool for collecting data
  - We usually import the Prometheus and then add our monitoring metrics
  - Prometheus expose the metrics and we build the alerts based upon the metrics
- `Grafana` is used for visualization the metrics

K8s API

- The K8S API is layer in the Master node
- The API server is the central management entity
- The API Server is the **only** component which is connected to the etcd

kubectl

This is the api for end users for interacting with the k8c cluster

kubelet

This component is part of each node and is used internally by the workers and the master

kubeadm

Kubeadm is a tool for creating Kubernetes clusters.

Rolling updates / Blue green deploy

- With blue/green deployments a **new version** (green) is deployed alongside the **existing version** (blue).
- Once the deploy is done then the ingress/router to the app is updated to switch to the new version (green).
- Sample code

Networking (Basics)

- In K8S we use `Network Policies`
- `Network Policies` is a specification of how groups of pods are allowed to communicate with each other and other network endpoints.
- `NetworkPolicy` resources use labels to select pods and define rules which specify what traffic is allowed to the selected pods.
- The main policy types are
  - Ingress
  - Egress

Networking (Basics)

- Ingress
  Incoming traffic, **whitelist ingress rules**.
  Each rule allows traffic which matches both the **from** and ports sections.

- Egress
  Outgoing traffic, **whitelist egress rules**.
  Each rule allows traffic which matches both the **to** and ports sections.

Networking (Basics)

- By default, if no policies exist in a namespace, then **all ingress and egress** traffic is allowed to and from pods in the namespace.

Network Policies

Policy	Description
`allow-all`	Allow all traffic to all pods in a namespace
`default-deny`	Block all traffic to all pods in a namespace

Networking (Basics - Ingress)

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy

policyTypes:
- Ingress
- Egress
ingress:
- from:
- ipBlock:
    cidr: 172.17.0.0/16
    except:
    - 172.17.1.0/24
- namespaceSelector:
    matchLabels:
      project: myproject
- podSelector:
    matchLabels:
      role: frontend
ports:
- protocol: TCP
  port: 6379
```

Networking (Basics - Egress)

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy

policyTypes:
- Ingress
- Egress
egress:
- to:
  - ipBlock:
      cidr: 10.0.0.0/24
  ports:
  - protocol: TCP
    port: 5978
```

THE END.