Kubernetes

Part 1

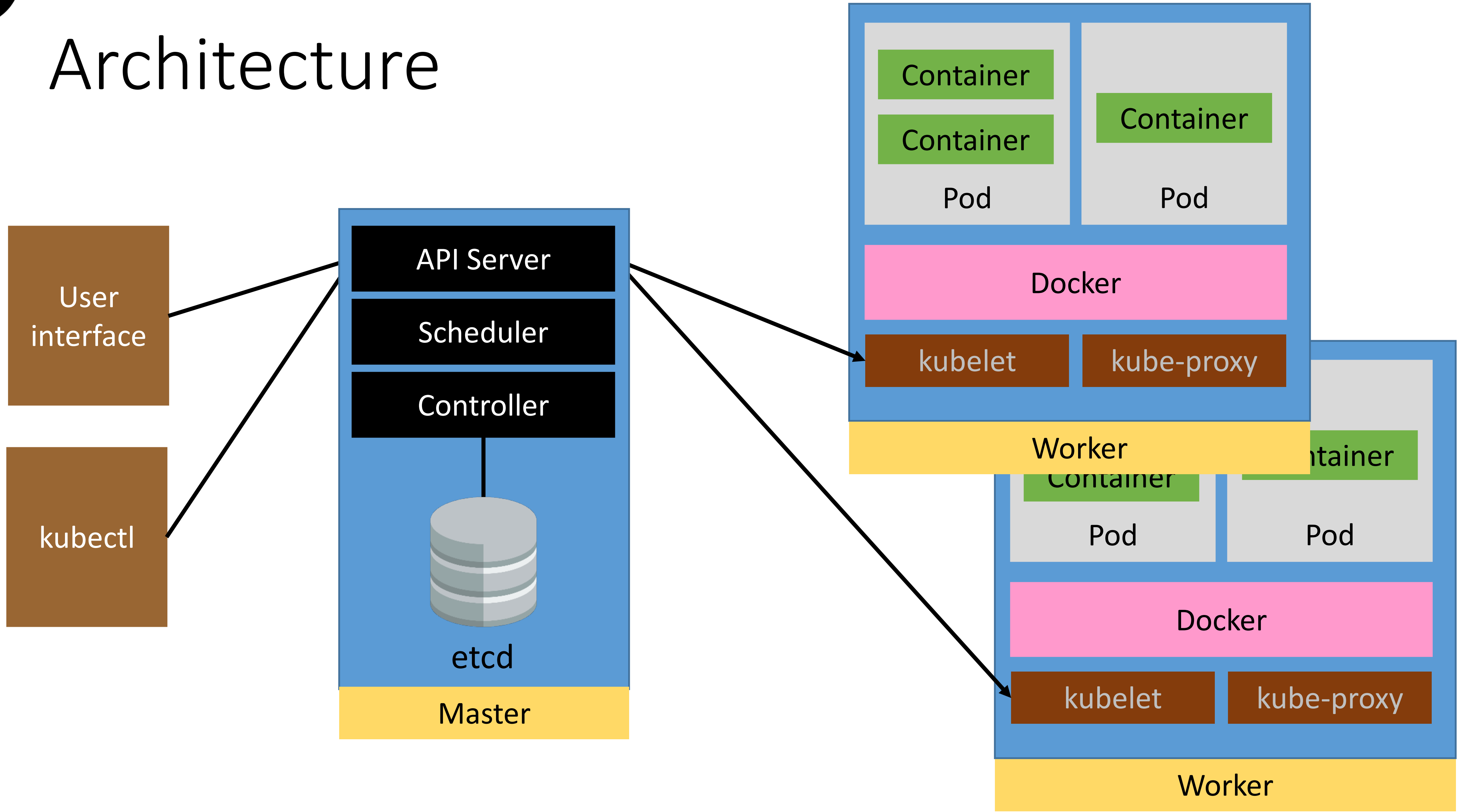
Cloud Native Application

- Cloud native computing
  - Segment application into microservices
  - Package each part of into its own container
  - Dynamically orchestrate those containers to optimize resource utilization

What is Kubernetes?

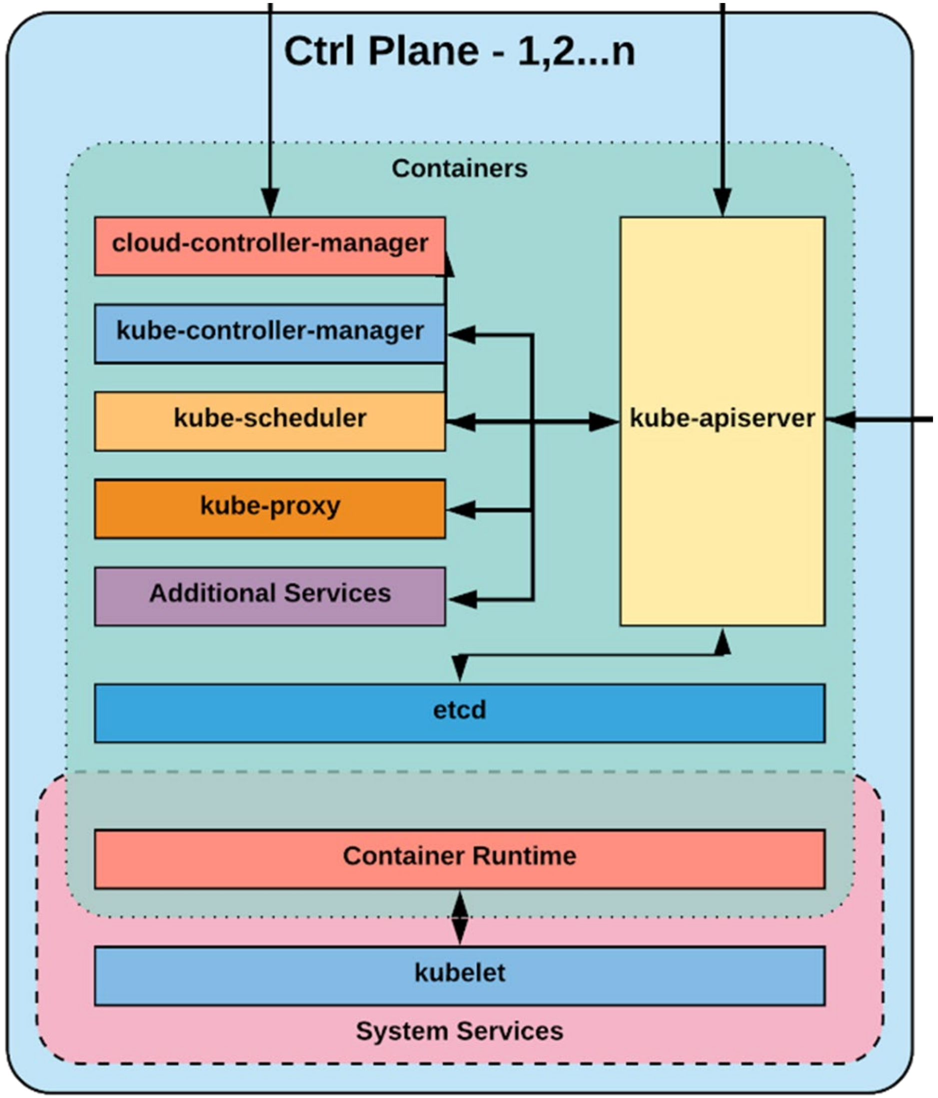
- Container orchestrator
  - Schedules and deploys containers
  - Recover from failure, keeping the actual state and desired state of an application in sync
  - Provides basic monitoring, logging, health checks
  - Enables containers to talk to each other
  - Scale workloads
- Project that started off as an internal Google project for managing containers
- Provides the same API across all cloud providers
  - Free from the underlying cloud platform

Architecture

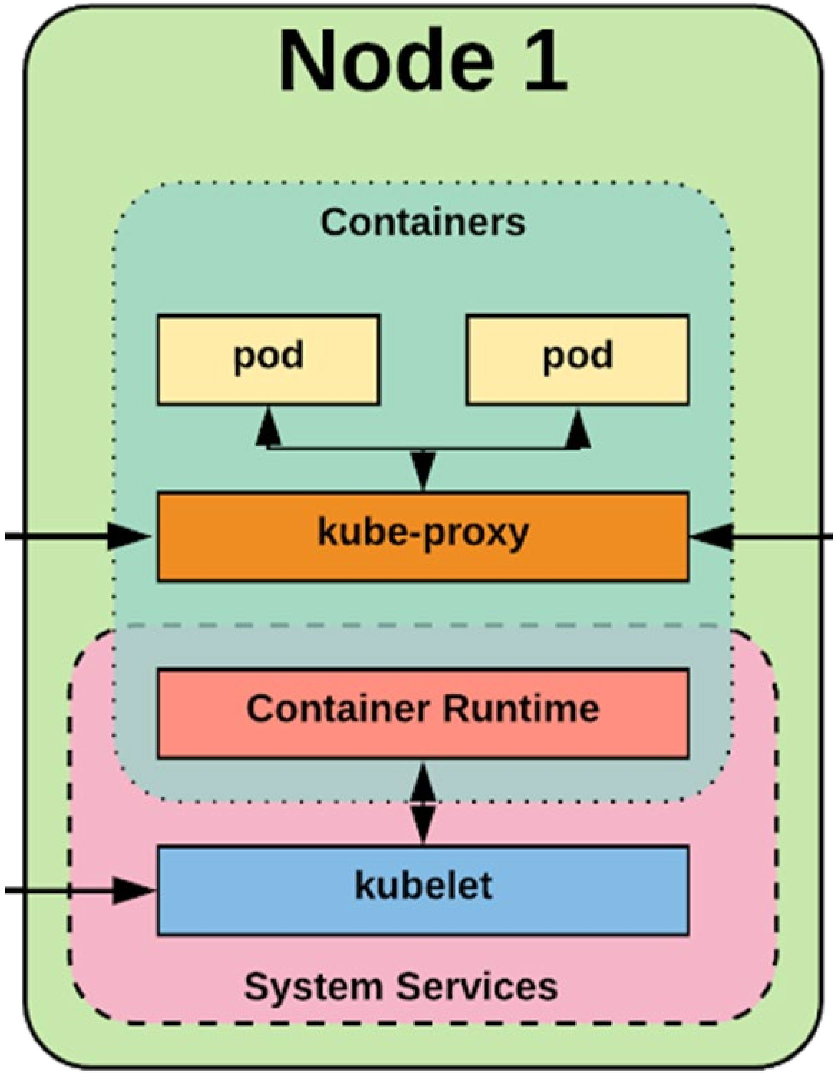


Control Pane Components

- API Server – receives REST request to create services, deploy Pods, etc
- Controller manager – runs a number of process to manages the cluster. Monitors the state of the cluster and steers the cluster towards the desired state
- Scheduler – evaluates workload requirements and schedule it on matching resources
- Etcd – a distributed key/value storage; used to store the cluster's state



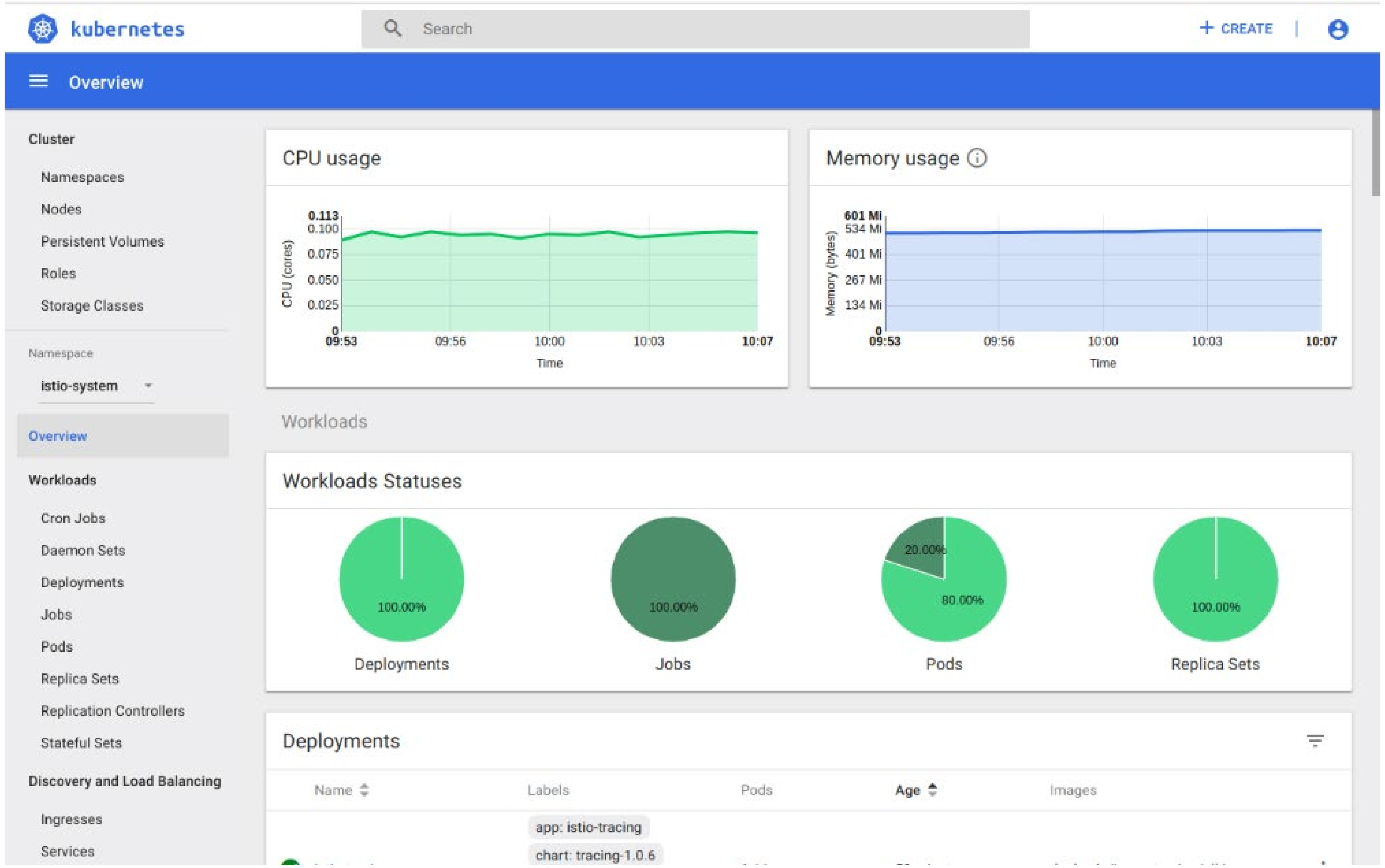
Node Components



- Kubelet – communicates with the API server in master. Takes orders from masters to schedule and manage Pods. Also reports the health of the Pods to the master node
- Kube-proxy – handles' the Pods networking. Performs connection forwarding and load balancing for services
- Container runtime – use to manage containers. Docker in our case

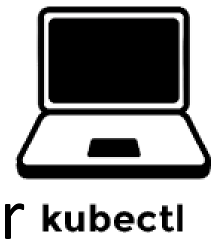
Interacting with Kubernetes

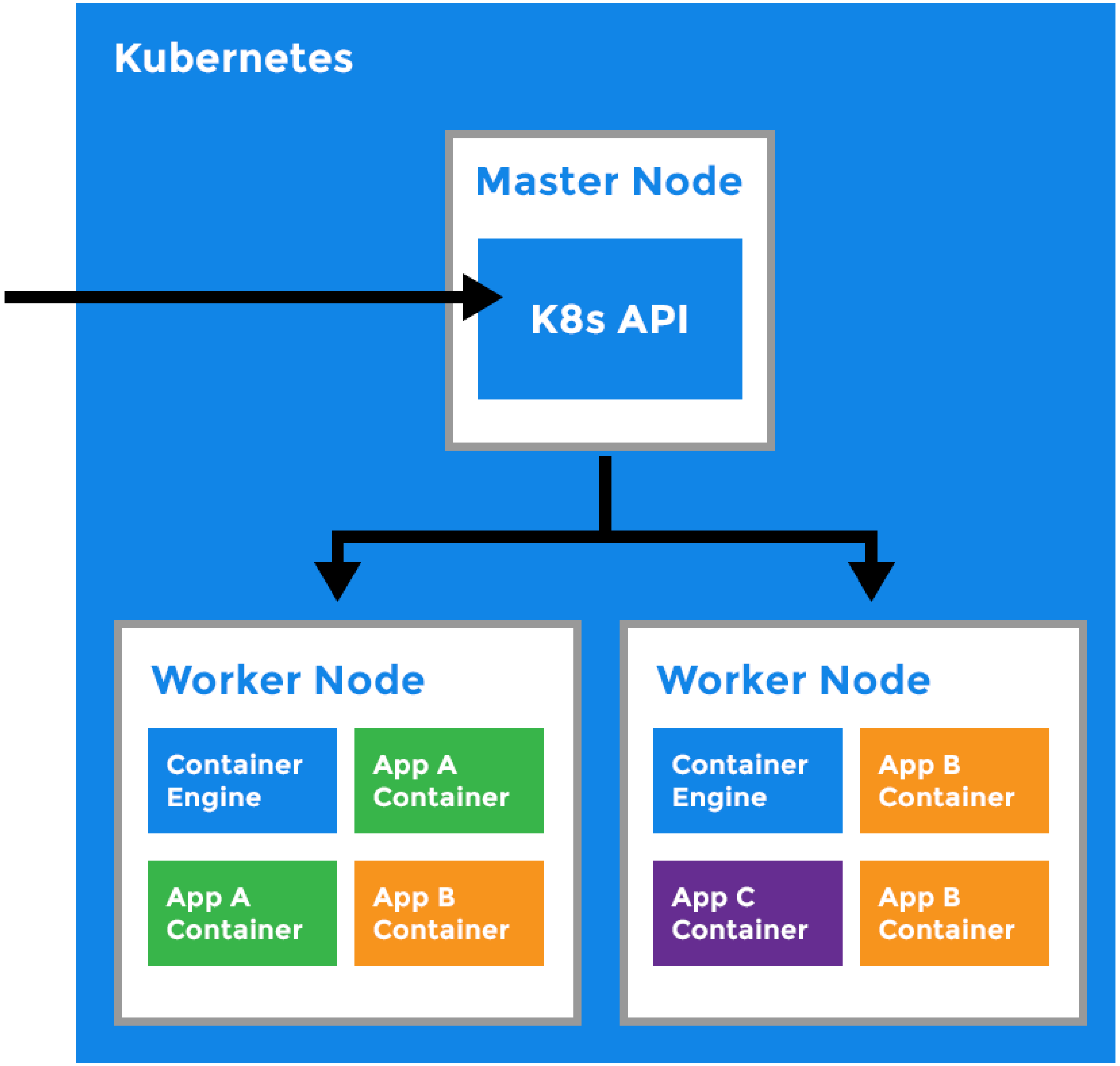
- Native Kubernetes dashboard
  - Not dashboard provided by cloud providers
  - May need to install
- **`kubectl`**
  - Command line
- Programmatic
  - REST API
  - Language specific client libraries



kubectl

- Command line tool for interacting with the cluster





- Gets cluster information from configuration file
  - Default location `$HOME/.kube/config`
- If file is in different location
  - Use `--kubeconfig` option
  - Set `KUBECONIG` environment variable

Image from https://blog.risingstack.com/what-is-kubernetes-how-to-get-started/

Kubernetes Command

Resource type	
pod	Pod
deploy	Deployment
svc	Service
ing	Ingress
pvc	Persistent volume claim

- Creating a resource

```
kubectl apply -f <yaml_file>
```

- Getting a resource

```
kubectl get <resource_type> <resource_name>
```

- Detailed information

```
kubectl describe <resource_type> <resource_name>
```

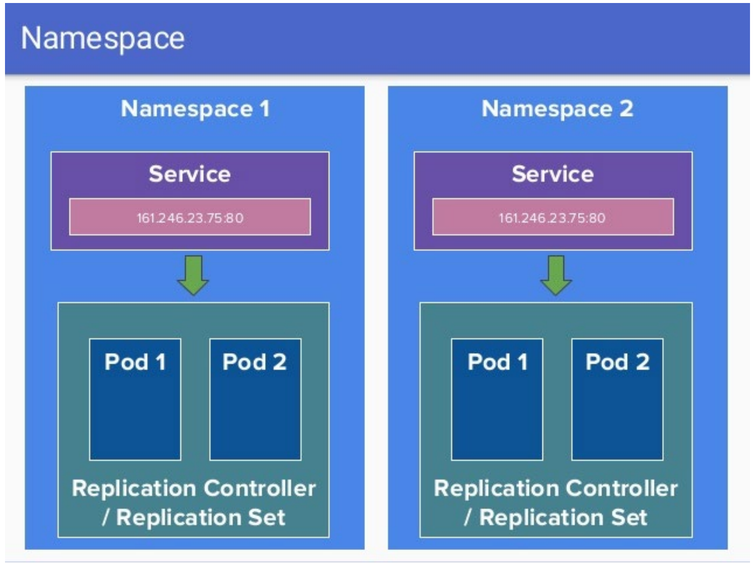
- Delete a resource

```
kubectl delete <resource_type> <resource_name>
kubectl delete -f <yaml_file>
```

Namespace

- A single Kubernetes cluster should be able to run application from multiple users
- For security reasons
  - Each user/application should only be allowed to access their own resources
  - Different policies to regulate their access
- Kubernetes uses namespace to isolate users and applications
  - Can restrict resources, access inside a namespace

- Default namespaces
  - `default` - the namespace to that Kubernetes put your deployment into if you did not specify any namespace
  - `kube-system` - for system
  - `kube-public` - accessible by all user



Defining and Using Namespace

- Create the namespace

```
kubectl apply -f namespace.yml
```

- Specify the namespace with -n

```
kubectl -n <name> <command>
```

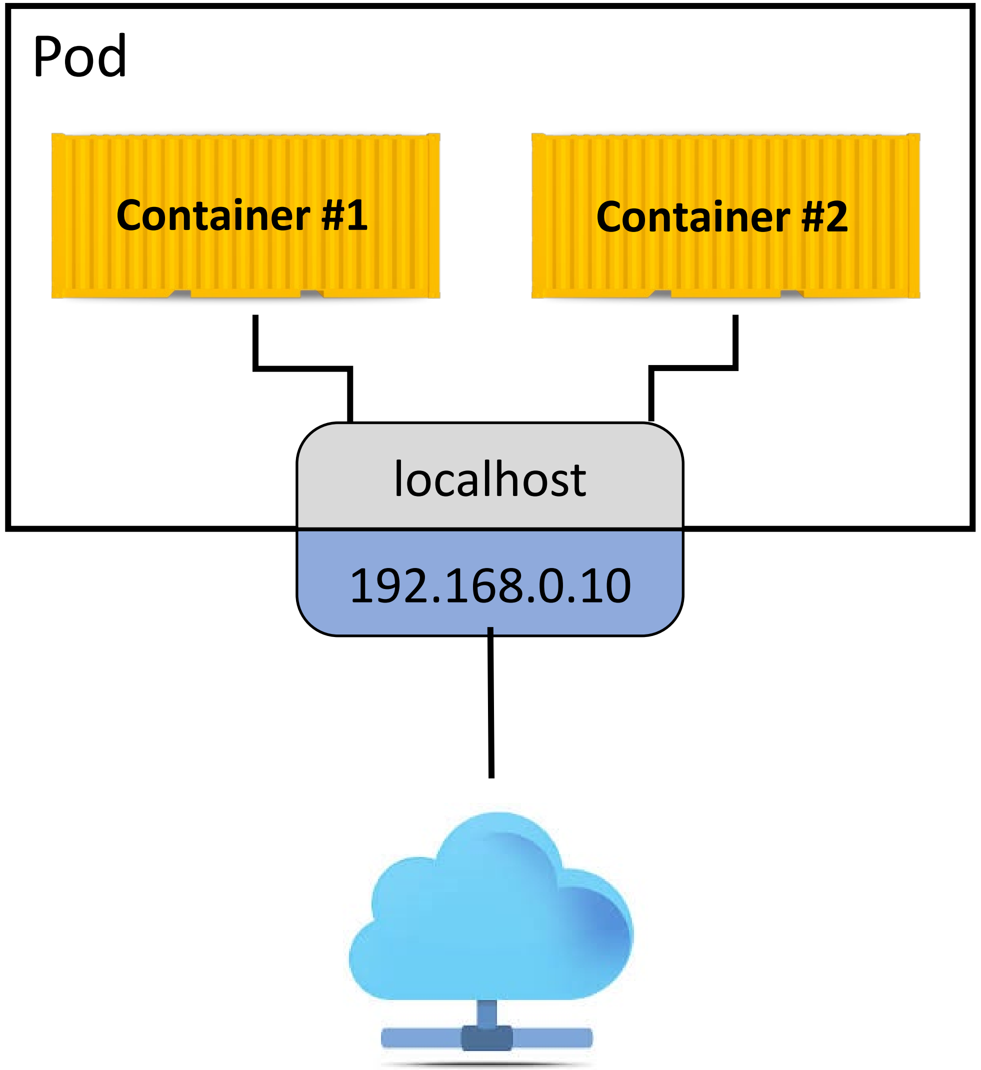
- Delete namespace

```
kubectl delete namespace/<name>
```

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: myapp
```

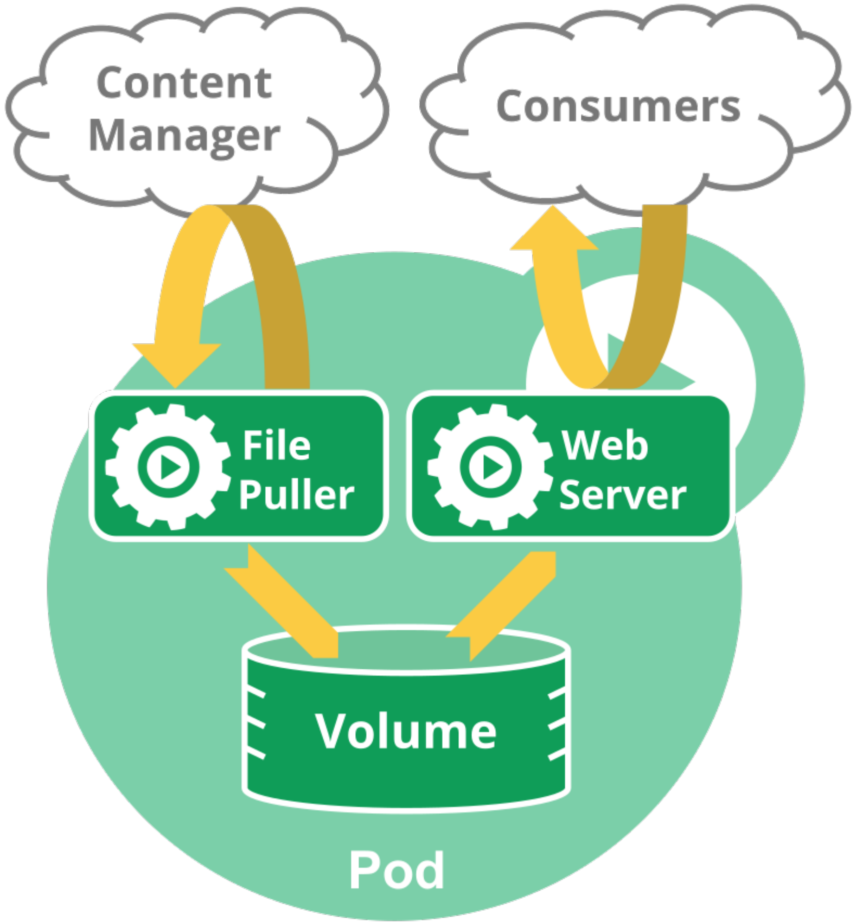
Namespace

Pod



- Is a "unit of work" in Kubernetes
  - Smallest schedulable entity
- Contains one or more containers
- Containers in a Pod share the same namespace
  - Allow containers inside a Pod to communicate with each other via localhost
- Containers in a Pod are either all running or they are not
  - Can never have a Pod with one or more failed containers
- Each Pod has an IP address
- Pods are ephemeral
  - They can die
  - They can be rescheduled by Kubernetes to another node

Example of Multi Container Pod



- Two containers
  - Web server displays content from a volume
  - File puller to ensure that the volume has the latest content by syncing it with some remote master
- The two containers are tightly coupled and should be schedule as a Pod

Image from https://kubernetes.io/docs/concepts/workloads/pods/pod/

Defining a Pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp
  namespace: default
  labels:
    version: v1
    zone: prod
spec:
  containers:
    - name: myapp
      image: myapp:sha256:fadfec....
      imagePullPolicy: Always
      ports:
        - containerPort: 3000
```

Mandatory fields

Type of object

Unique name of object

Defaults to ‘default’ namespace

Labels for filtering

Unique container name within the pod

Image tag

Image digest. More secure than using a tag

Where to get the image from

Port to expose on the Pod

Pod Management

- Create a Pod

```
kubectl apply -f pod.yml
```

- View all Pods

```
kubectl get pods -o wide
kubectl get pods -o yaml
```

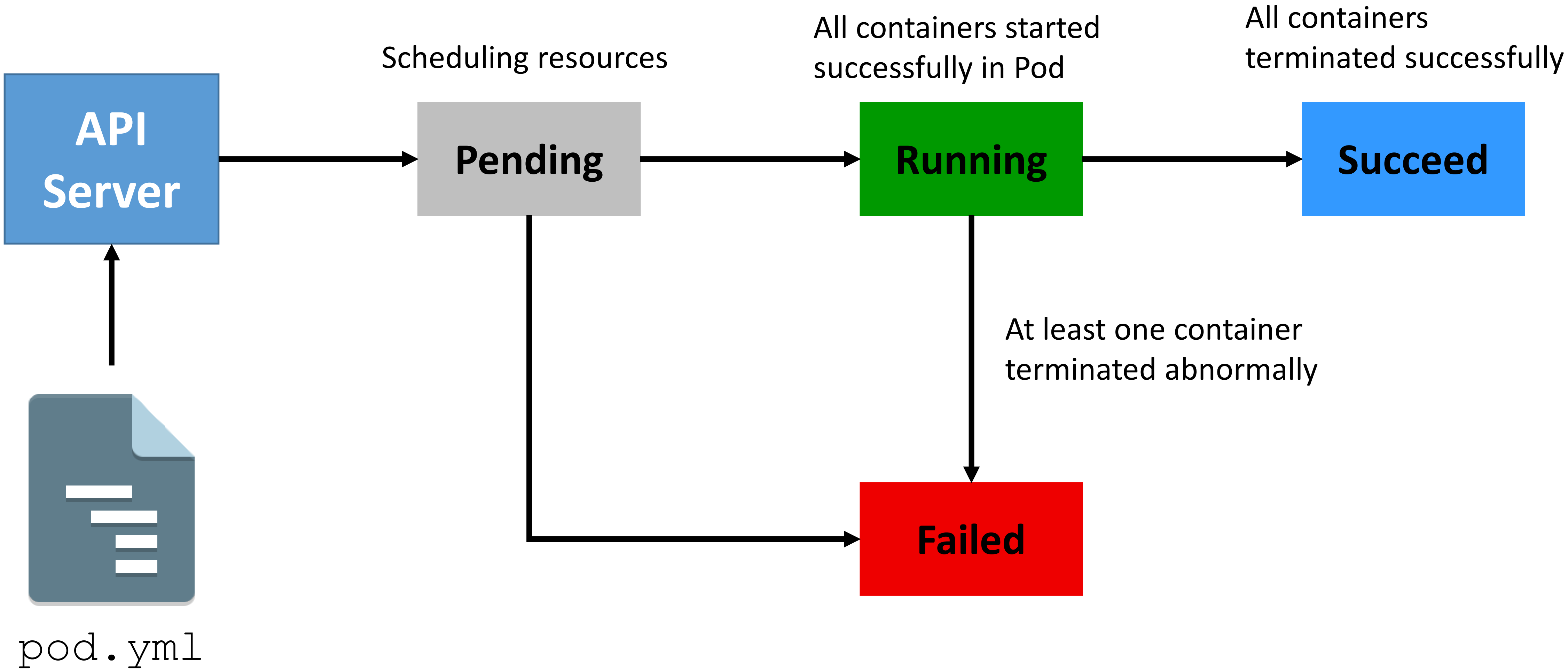
- Detail information about a pod

```
kubectl describe pods myapp-pod
```

- Delete a Pod

```
kubectl delete -f pod.yml
kubectl delete pod myapp-pod
```

Pod Lifecycle



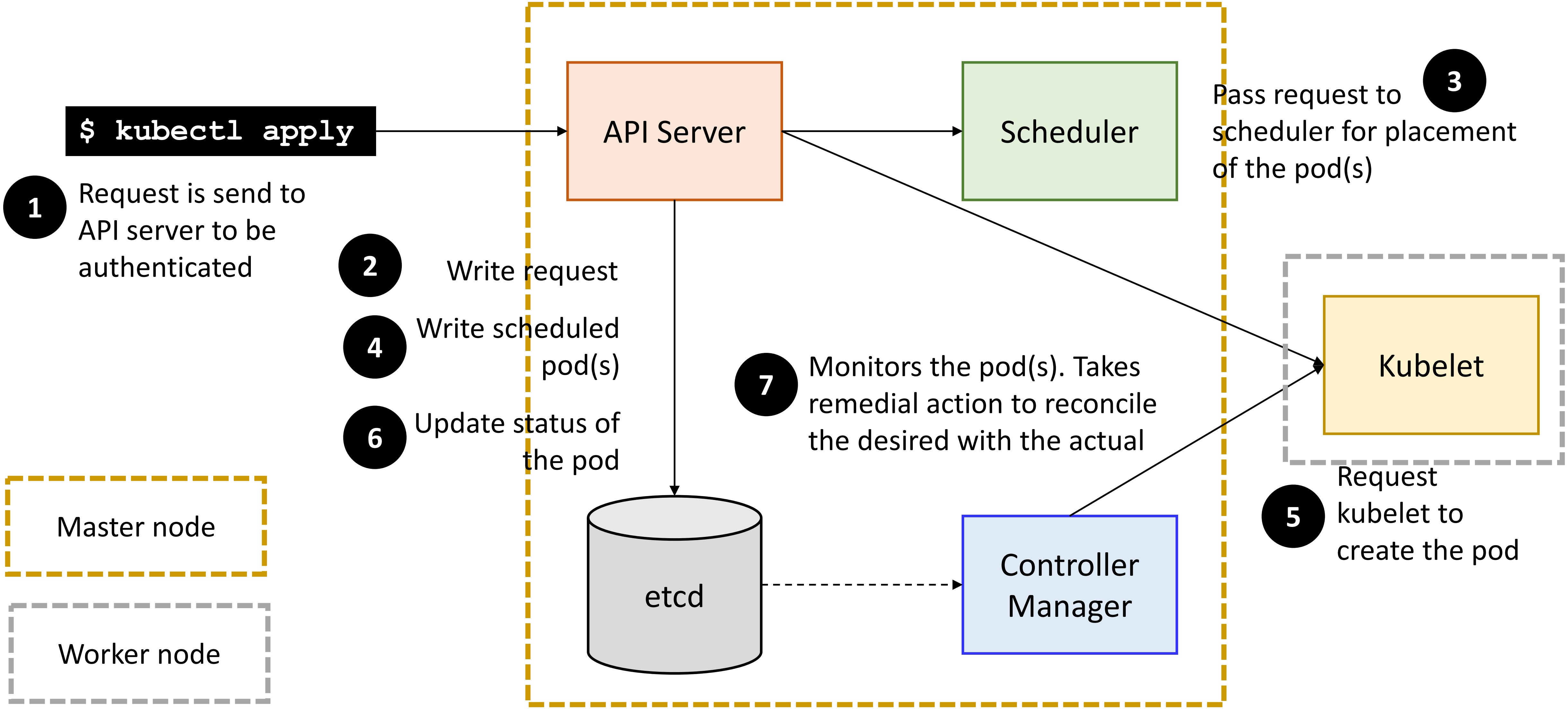
Accessing the Pod

- Pod's container port is not exposed
- To access it need to bind the port to the host port

```
kubectl port-forward pod/myapp-pod 8080:3000
```

  - Forwards traffic from port 8080 to Pods' port 3000
- Not a good way to access the application
  - Use for testing

How Kubernetes Provision Resources



Health Checks - Readiness

- Discover when a pod is ready to serve traffic
  - Eg. at startup - Pod is creating database
  - Eg. at steady state - scaling a large image and will not receive other request until the current one completes
- Will not route traffic to it until the pod is ready again. Traffic will be rerouted to other Pods

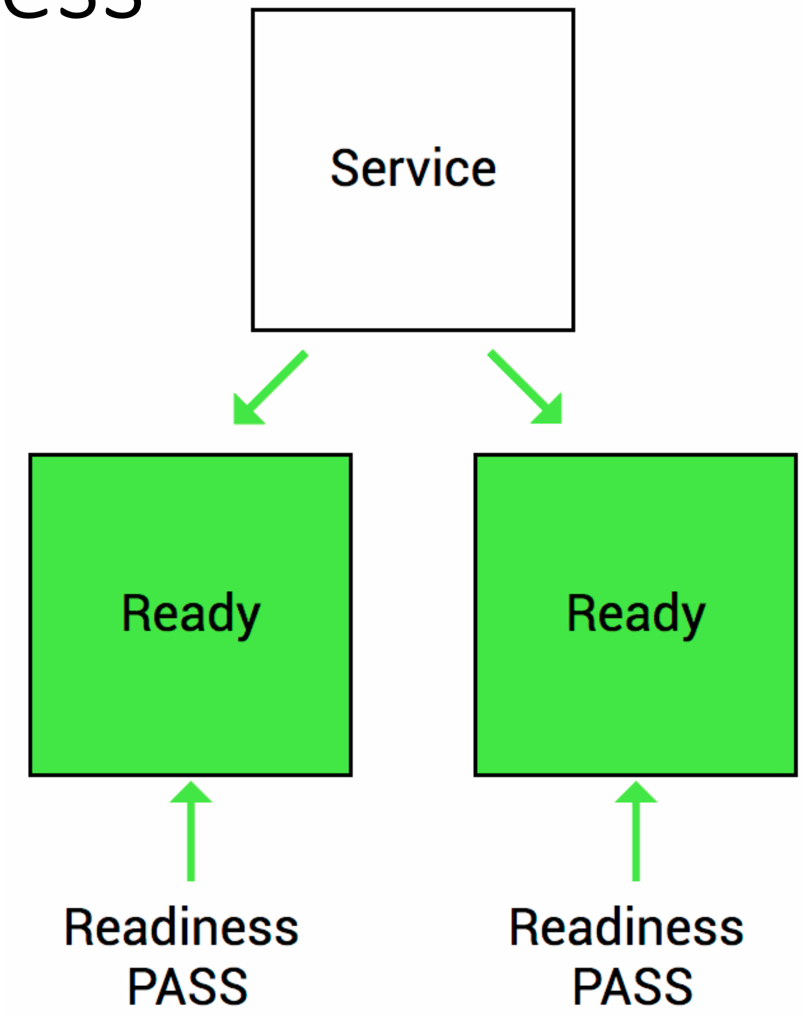


Image from https://cloud.google.com/blog/products/gcp/kubernetes-best-practices-setting-up-health-checks-with-readiness-and-liveness-probes

Health Checks - Liveness

- Checks if a Pod is dead or alive
- If Pod is dead, then remove the Pod and starts a new Pod to replace it
- Difference between liveness and readiness is that a Pod can fail readiness but is alive

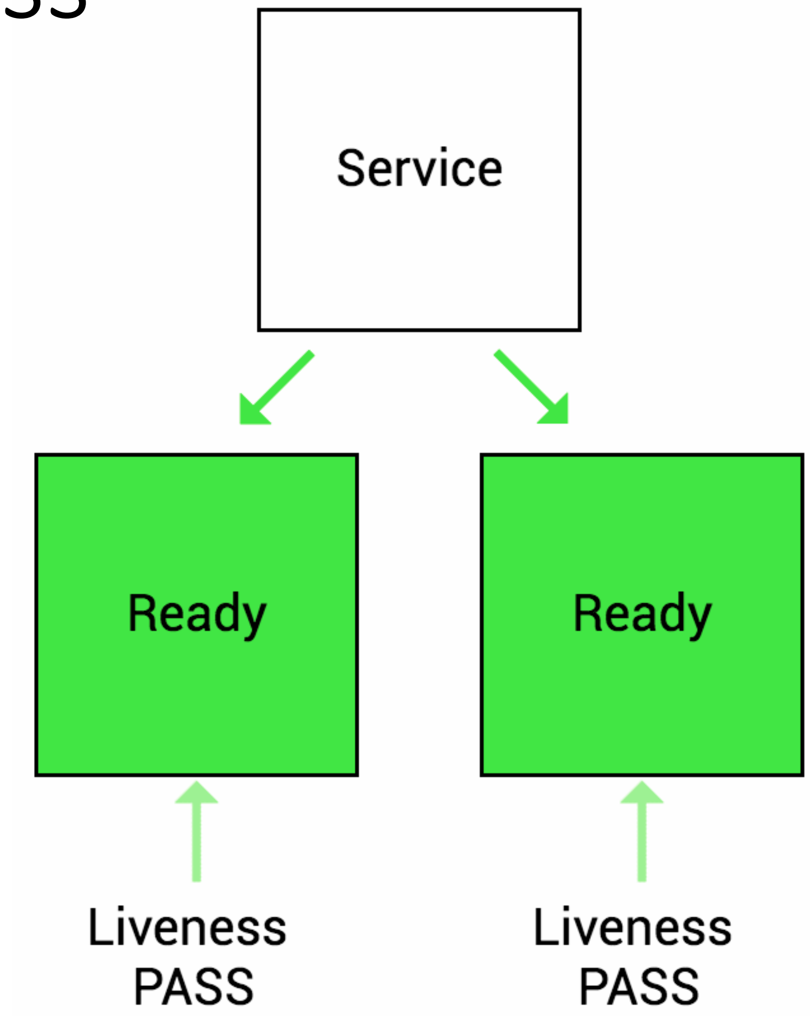


Image from https://cloud.google.com/blog/products/gcp/kubernetes-best-practices-setting-up-health-checks-with-readiness-and-liveness-probes

Defining Probes

```yaml
apiVersion: v1
...
spec:
  containers:
    - name: myapp
      image: myapp:sha256:fadfec....
      imagePullPolicy: Always
      ports:
        - name: app-port
          containerPort: 3000
      readinessProbe:
        httpGet:
          path: /ready
          port: app-port
        timeoutSeconds: 5
        failureThreshold: 1
```

If these request returns a status code of greater than 400 then it is consider a failure

```yaml
livenessProbe:
  httpGet:
    path: /
    port: app-port
  timeoutSeconds: 5
  failureThreshold: 3
  successThreshold: 1
```

Kubernetes

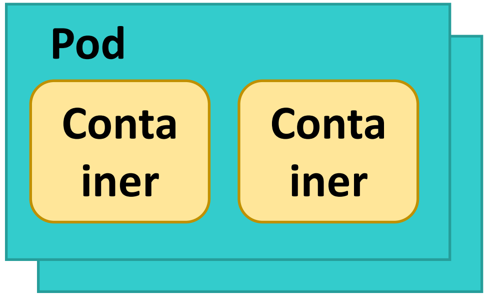


Image idea from https://www.slideshare.net/imesh/deep-dive-into-kubernetes-part-1

Deployments

- Almost always need more than a single Pod in production
- Deployments are used to create and deploy one or more Pods
- Deployment consist of
  - The number of Pods in the initial deployment
  - A template of the Pod which includes the Docker image, container port, etc.

Defining a Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp
spec:
  replicas: 2
  selector:
    matchLabels:
      name: myapp
  template:
```

Number of instances in deployment

Criteria to identify the pods belonging to this deployment

```yaml
template:
  metadata:
    name: myapp-pod
    labels:
      name: myapp
  spec:
    containers:
      - name: myapp
        image: myapp@sha256:...
        imagePullPolicy: Always
        ports:
          - containerPort: 3000
```

Pod definition

Deployment Management

- Create a deployment

```
kubectl apply -f deployment.yml
```

- View all deployments

```
kubectl get deploy -o wide
kubectl get deploy -o yaml
```

- Detail information about a deployments

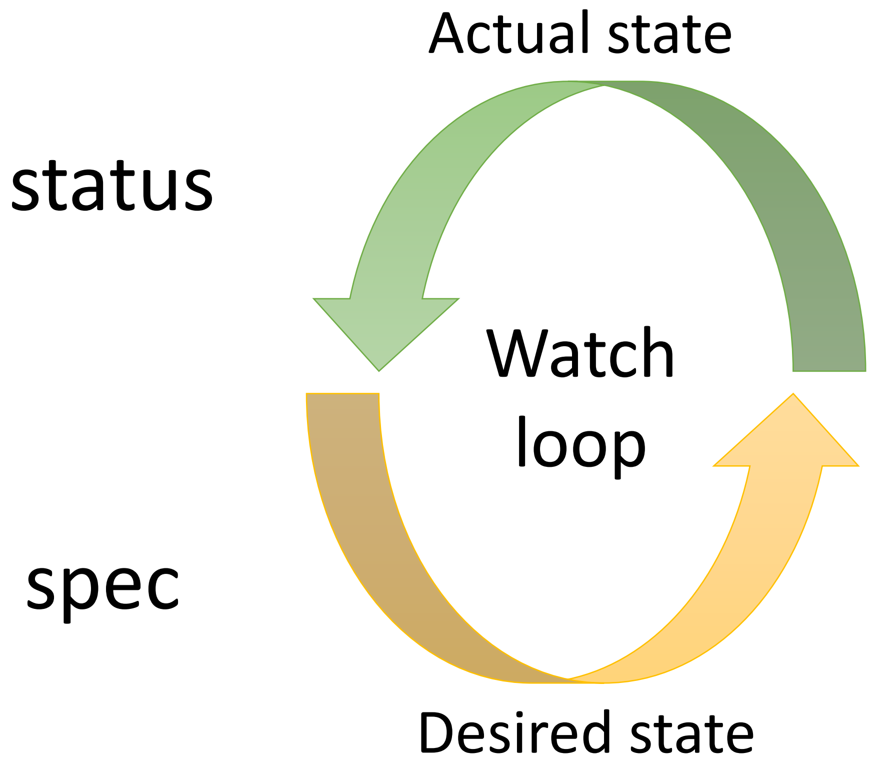
```
kubectl describe deploy myapp-pod
```

- Delete a deployment

```
kubectl delete -f deployment.yml
kubectl delete deploy myapp-deployment
```

Replica Sets

- Deployments uses replica sets to manage Pods
- Replica sets ensures that the desired number of Pods are running
  - As defined in the `replica` attribute
- Kubernetes will match the actual state against the desired state
  - If the actual number of instances is less than the specification, Kubernetes will provision additional Pods so that the actual matches the desired



Kubernetes

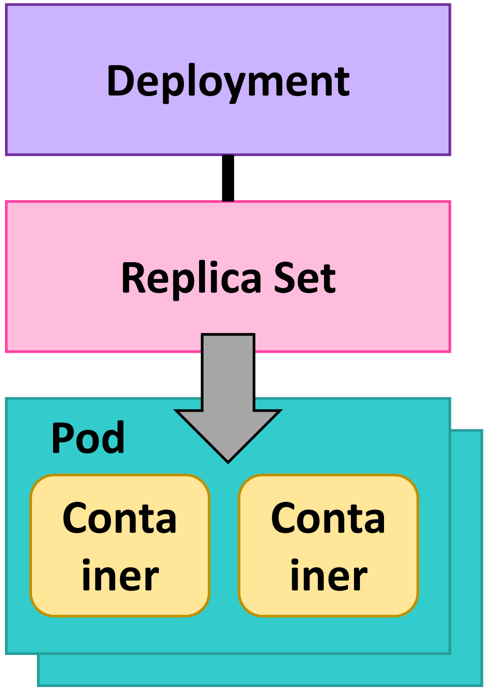


Image idea from https://www.slideshare.net/imesh/deep-dive-into-kubernetes-part-1

Passing Values into Containers

- Configurations can be passed into containers in a Pod as configuration maps and/or secrets
  - Key/value pair files
- Difference between ConfigMap and Secret is that the latter values are base 64 encoded
- Passed into the container as
  - Environment variables viz. bind the values to environment variables
  - Mounted as a volume into a container

ConfigMap and Secrets

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: myapp-config
data:
  db_name: northwind
  db_host: myserver
  db_port: 3306
```

```
apiVersion: v1
kind: Secret
metadata:
  name: myapp-secret
data:
  db_user: ZnJlZA==
  db_password: eWFiYWRhYmFkb28=
```

$$\underbrace{\texttt{echo -n 'fred' | base64}}_{\text{Encode value to base64}}$$

Injecting ConfigMap and Secret into Containers

```
containers:
- name: myapp
  image: myapp@sha256:...
  ...
  env:
  - name: DB_HOST
    valueFrom:
      configMapKeyRef:
        name: myapp-config
        key: db_host
  - name: DB_PASSWORD
    valueFrom:
      secretKeyRef:
        name: myapp-secret
        key: db_password
```

Bind the environment variable `DB_HOST` to the following value

`ConfigMap` name

Key from `ConfigMap`

Bind the environment variable `DB_PASSWORD` to the following value

`Secret` name

Key from `Secret`

Kubernetes

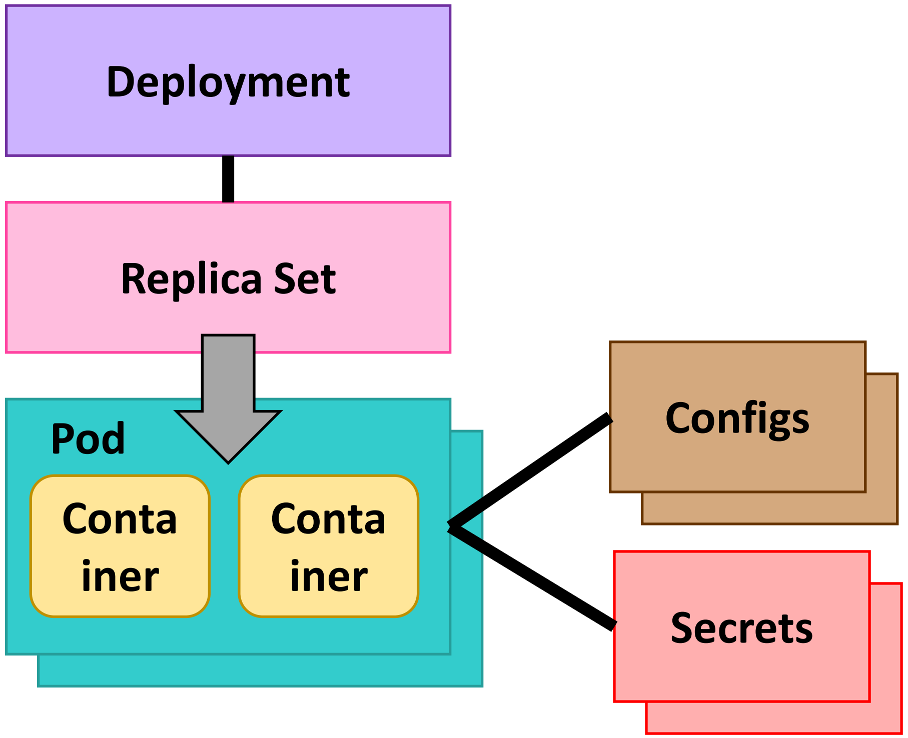
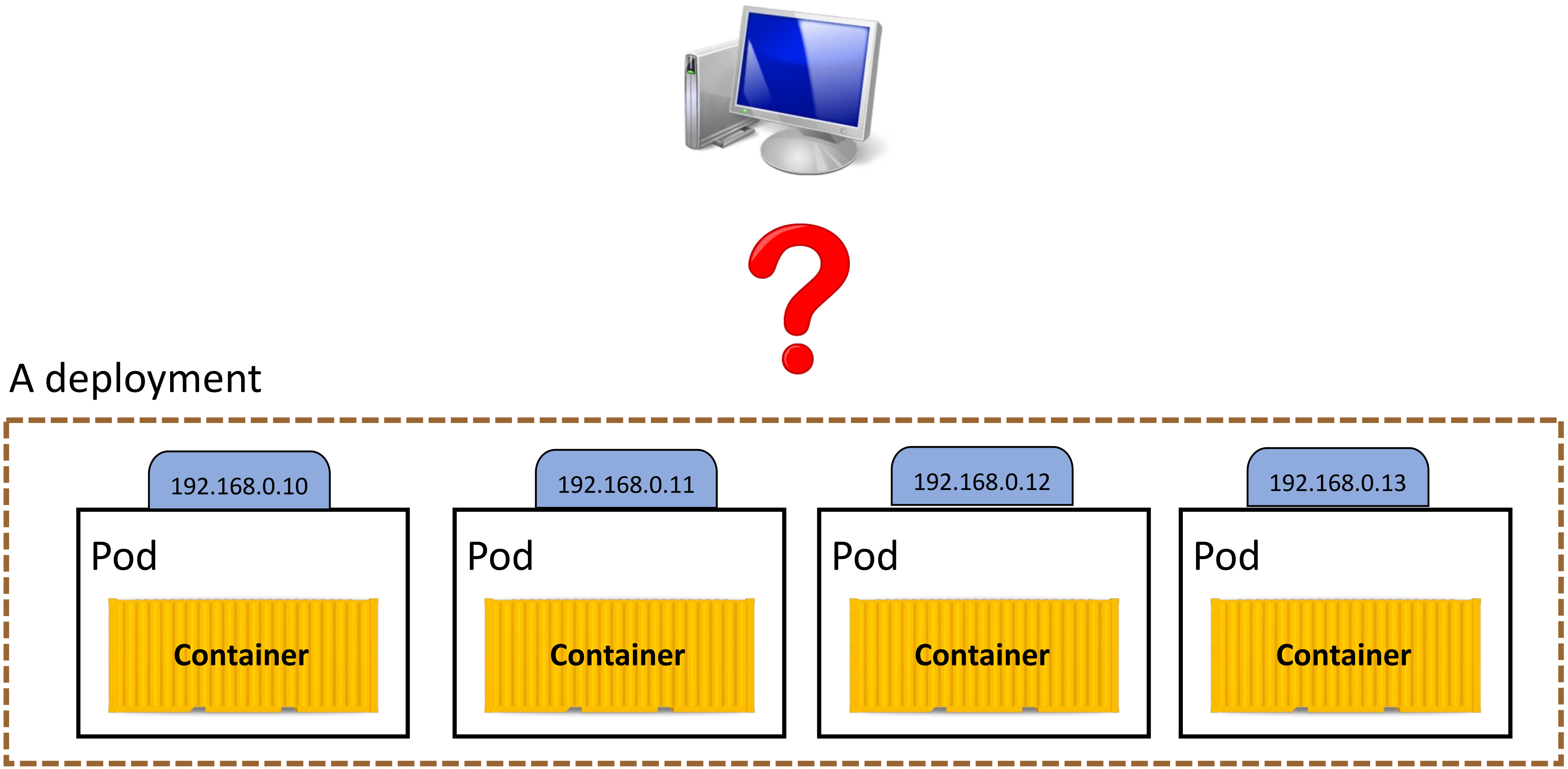


Image idea from https://www.slideshare.net/imesh/deep-dive-into-kubernetes-part-1

Accessing the Application



Pods are Ephemeral

- Pods are ephemeral
  - Can be reschedule to another node by the scheduler
  - Eg. when there is a node or network failure
- Clients cannot connect to pods directly via node
- Service provides a stable IP to the client
  - Acts as a proxy to a set of pods
  - Service keeps track of pods so clients connecting to them don’t have to
- When pods are reschedule to another node the service is responsible for redirecting the request to another Pod instances

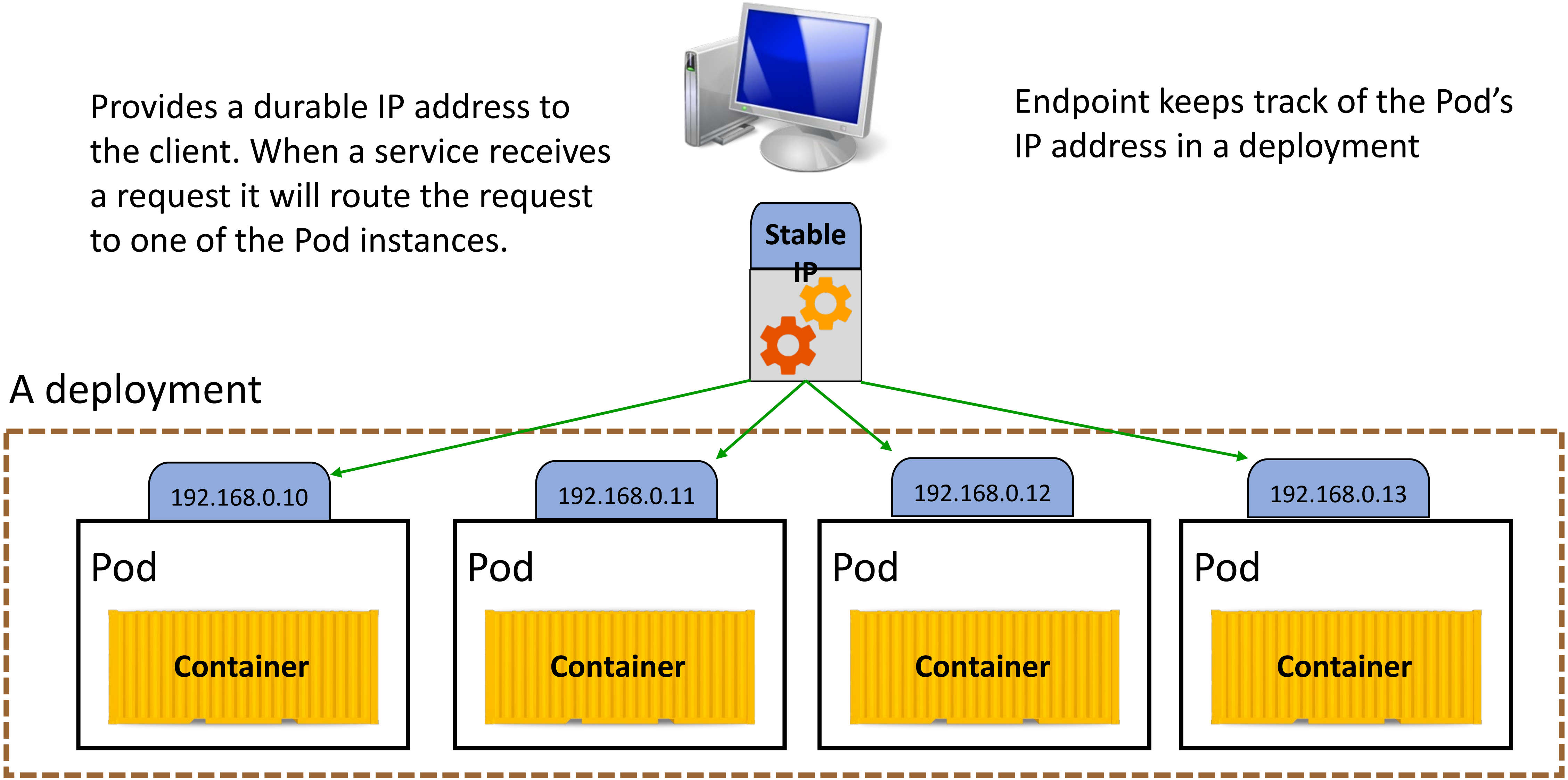
Service

- Well known endpoint for a set of Pods in a deployment
- Services will route a request to a Pod under that the service controls
  - Also provides simple load balancing
- Pods are selected to be in a service based on their labels
  - For a Pod to be in a service, the Pod only has to match some of its Pod labels
  - Pods and services are loosely coupled
- Services are durable, unlike Pods which are ephemeral
  - Static IP address
  - Static namespace DNS name

Accessing the Application

Provides a durable IP address to the client. When a service receives a request it will route the request to one of the Pod instances.

Endpoint keeps track of the Pod's IP address in a deployment



Defining a Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp
spec:
  type: ClusterIP
  selector:
    name: myapp
    version: v1
  ports:
    - name: http:
      port: 8080
      targetPort: 3000
      protocol: TCP
```

Service name

Specify the type of service that is exposed

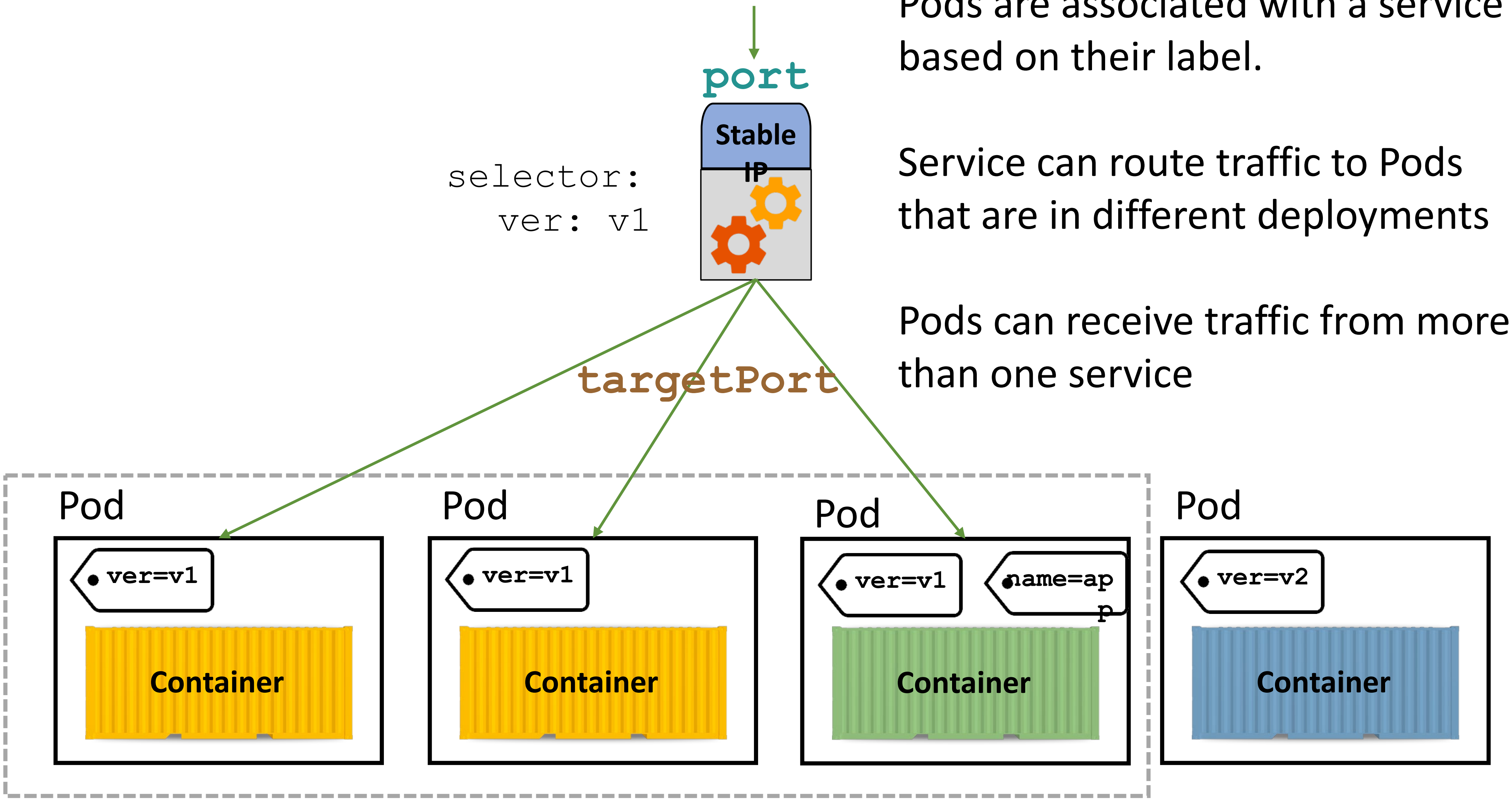
Route service to Pods that matches these labels

The port(s) that are exposed by this service

The port(s) that are exposed by this service

Route traffic from port (8080) to the Pod's port (targetPort 3000)

Services and Pods



Pods are associated with a service based on their label.

Service can route traffic to Pods that are in different deployments

Pods can receive traffic from more than one service

Service Type - `ClusterIP`

- Provision service IP address inside the cluster
- The IP address is not accessible from outside of the cluster
- This is the default

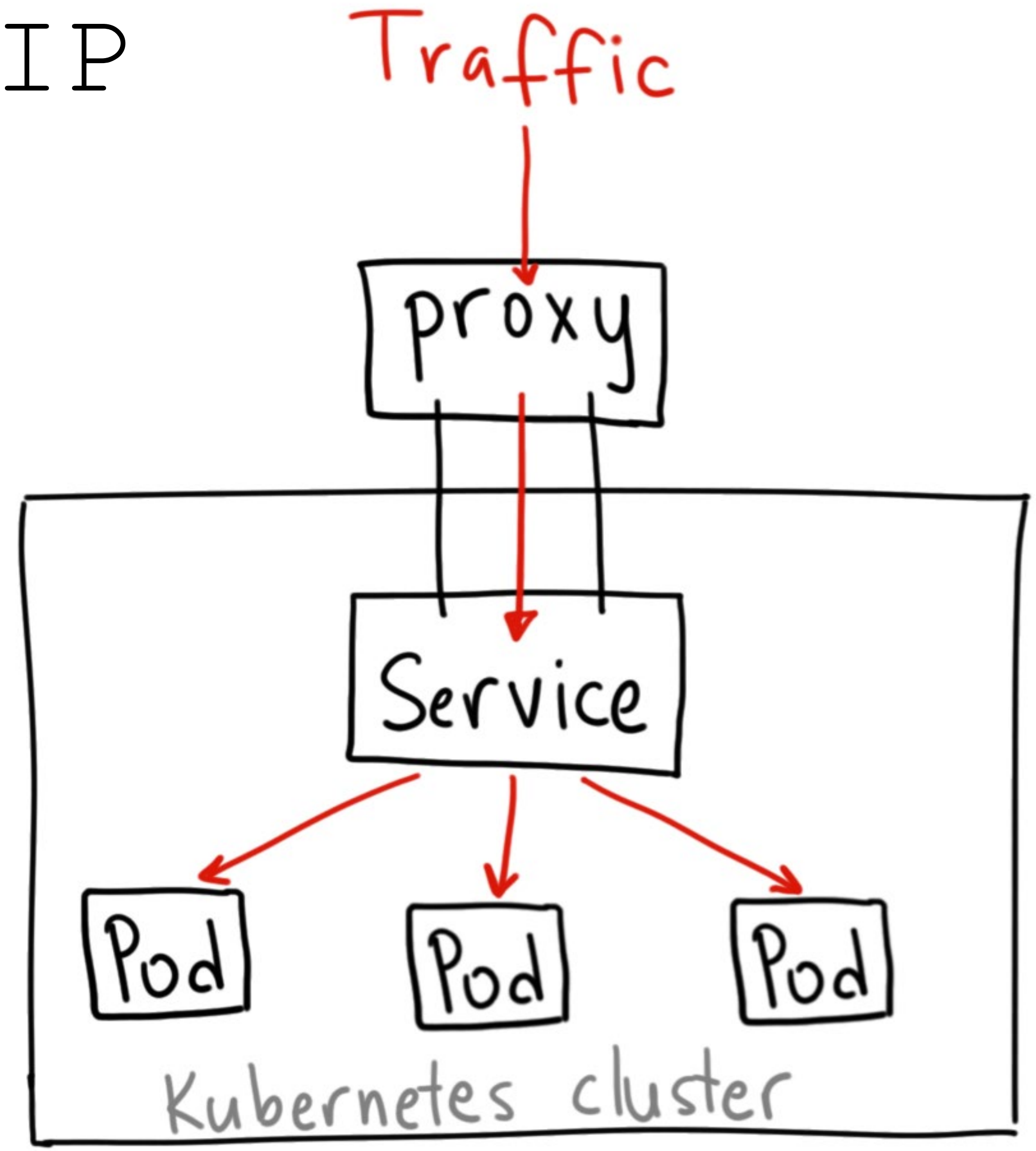


Image from https://medium.com/google-cloud/kubernetes-nodeport-vs-loadbalancer-vs-ingress-when-should-i-use-what-922f010849e0

Service Type - `NodePort`

- Opens a port, 30000 in this case, in all the node
- Traffic arriving at port 30000 on any of the nodes will be routed to the service
  - Expose as `ClusterIP`
- Make services appear local viz. the service can be accessed with localhost
  - Extra hop if the node does not have to Pod
- Need to pick a cluster node and the exposed port (node port) to access the service

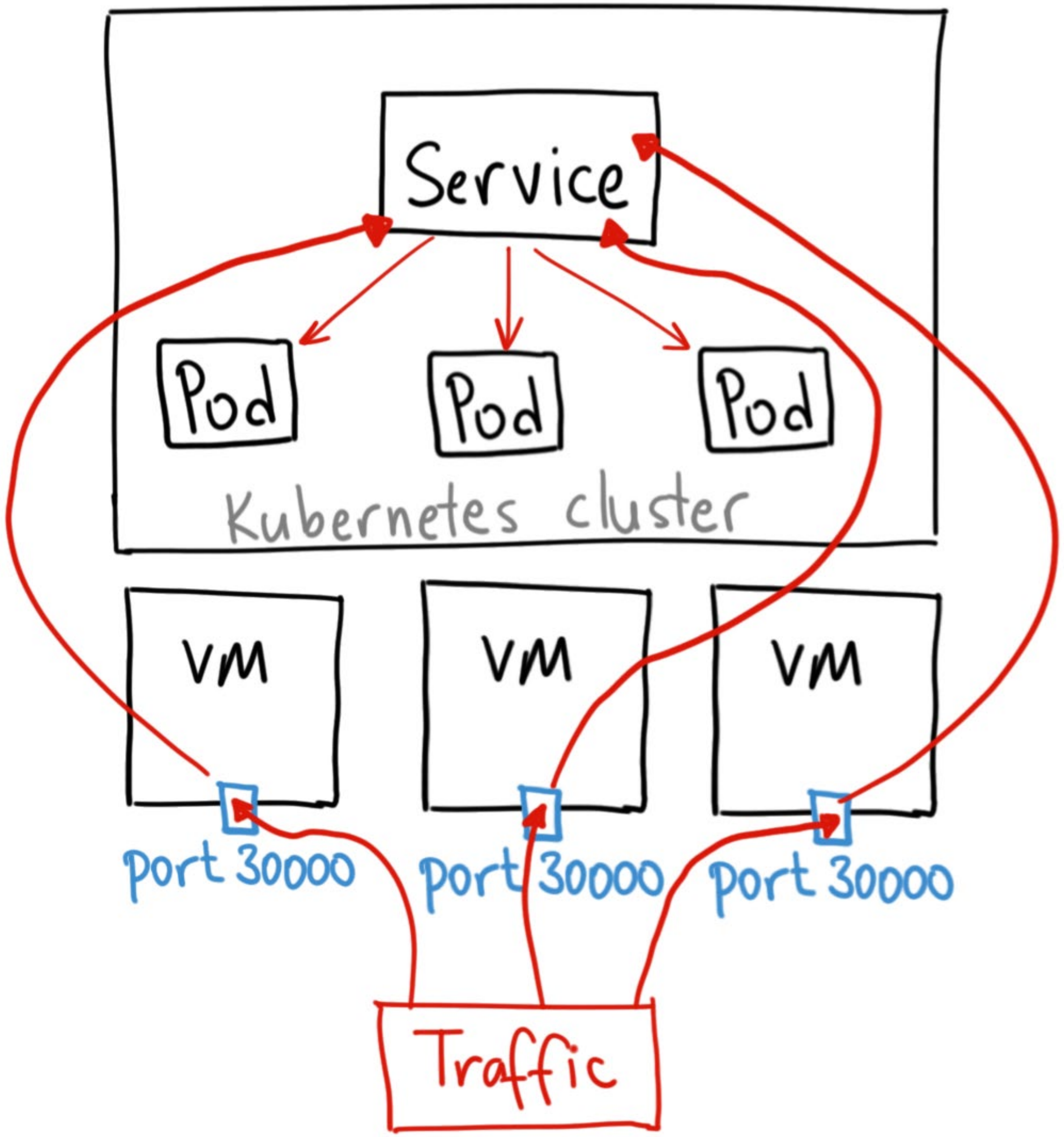


Image from https://medium.com/google-cloud/kubernetes-nodeport-vs-loadbalancer-vs-ingress-when-should-i-use-what-922f010849e0

Service Type - `LoadBalancer`

- A load balancer will route traffic to the service
- Traffic coming from the load balancer will be distributed to a node according to its routing policy
  - Exposed as `NodePort`
- The load balancer is accessible from outside of the cluster
- `LoadBalancer` service type will be provisioned by the underlying cloud platform
  - By the cloud controller manager

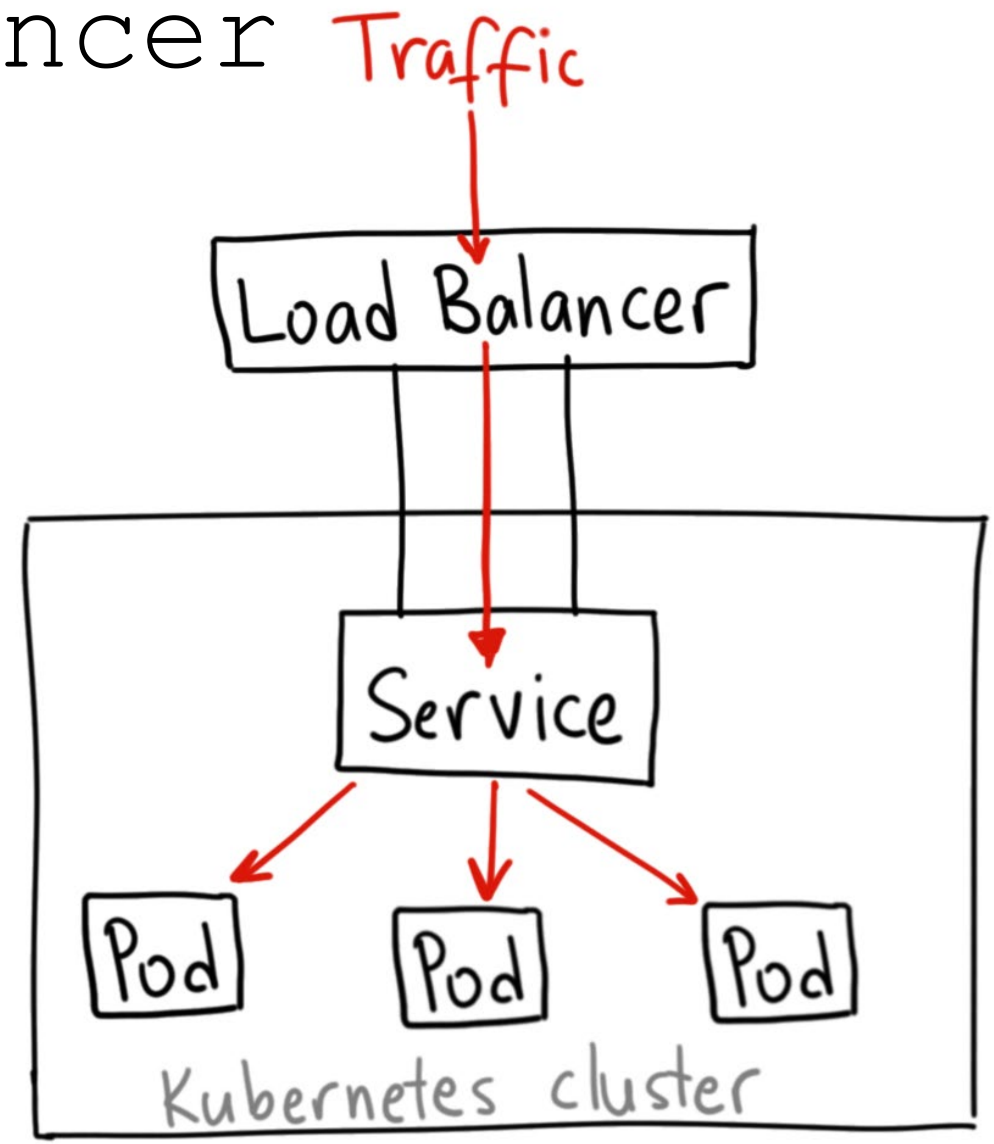
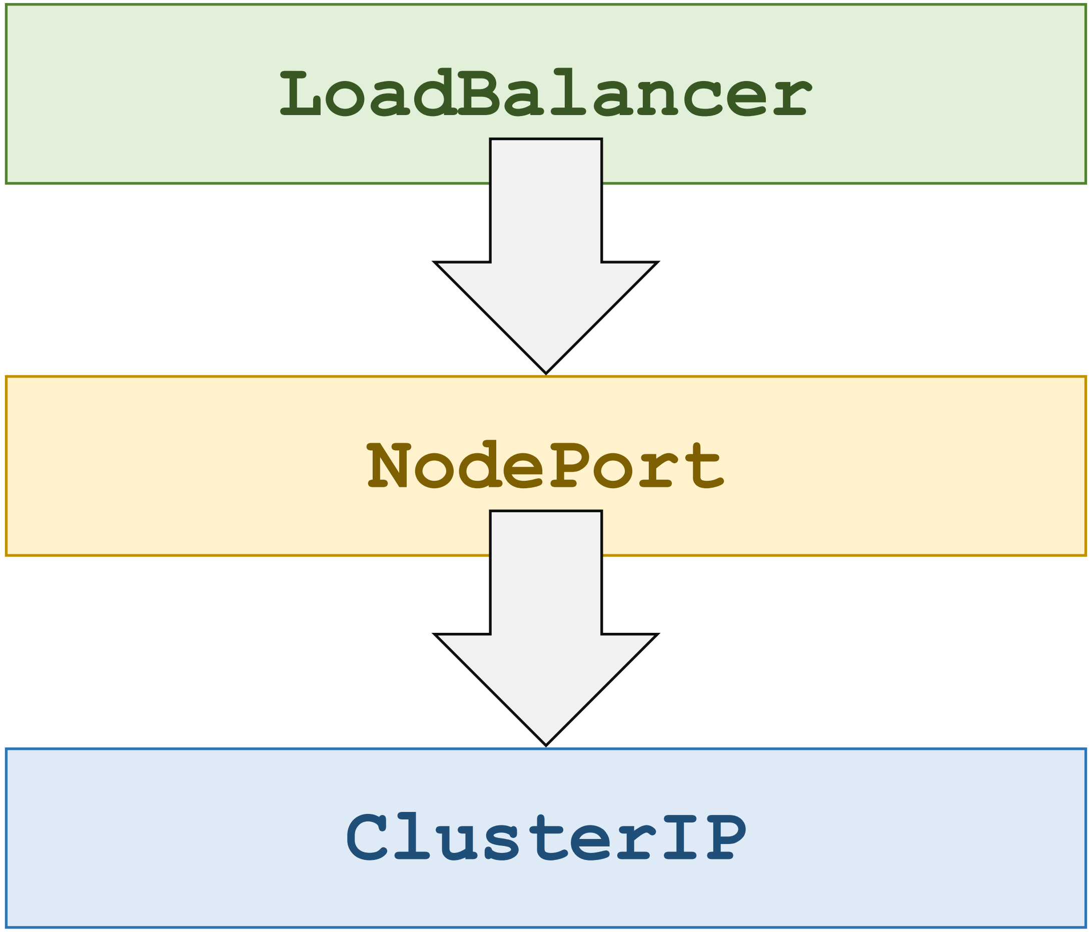


Image from https://medium.com/google-cloud/kubernetes-nodeport-vs-loadbalancer-vs-ingress-when-should-i-use-what-922f010849e0

Service



Pod to Service Communication

- Kubernetes creates an entry in its internal DNS (KubeDNS)

  `<service_name>.<namespace>.svc.cluster.local`

- A pod can access a service either with
  - service name if pods and service are in the same namespace
  - FQDN if service and pods are in different namespaces

Accessing the Service

- May need to access the service from outside the cluster
  - For testing
- Forward traffic from the host into the cluster's IP

```
kubectl port-forward svc/<service_name> 8080:3000
```

  - Port map 8080 from the host to port 3000 exposed by the service
- Or start a kube-proxy

```
kubectl proxy --port=8080
```

  - Access the service with the following URL

```
http://localhost:8080/api/v1/namespaces/<namespace>/services/
http:<service_name>:3000/proxy/
```

Kubernetes

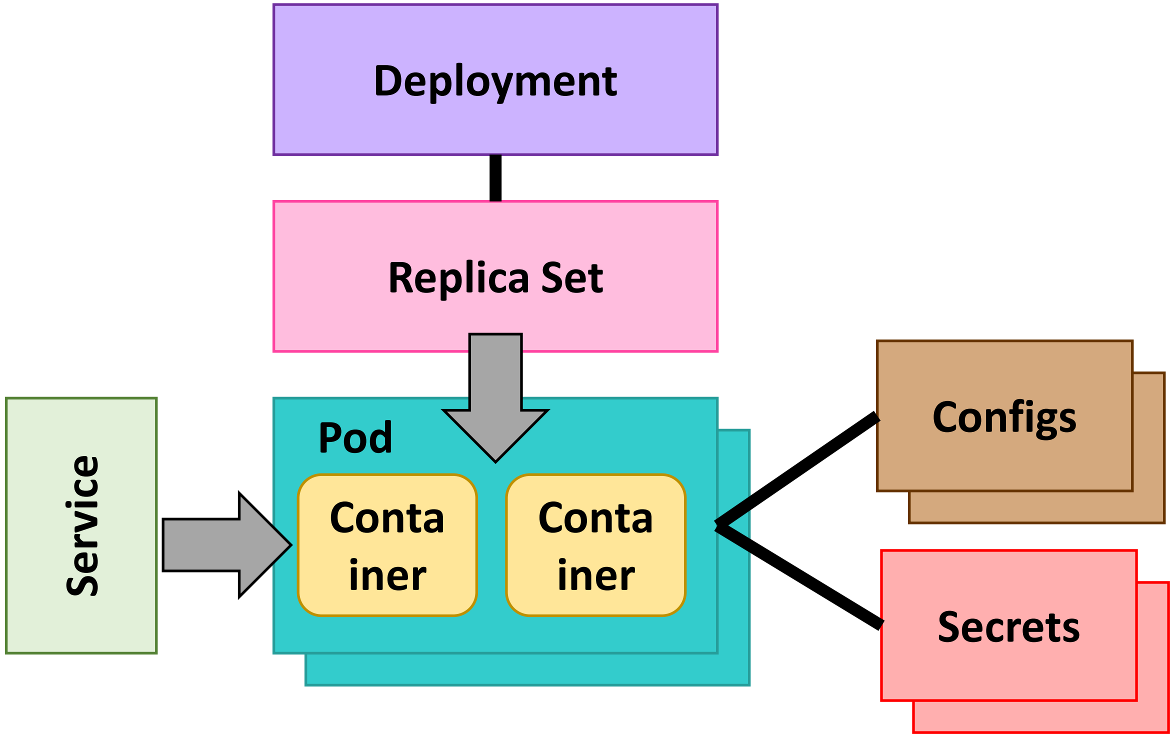
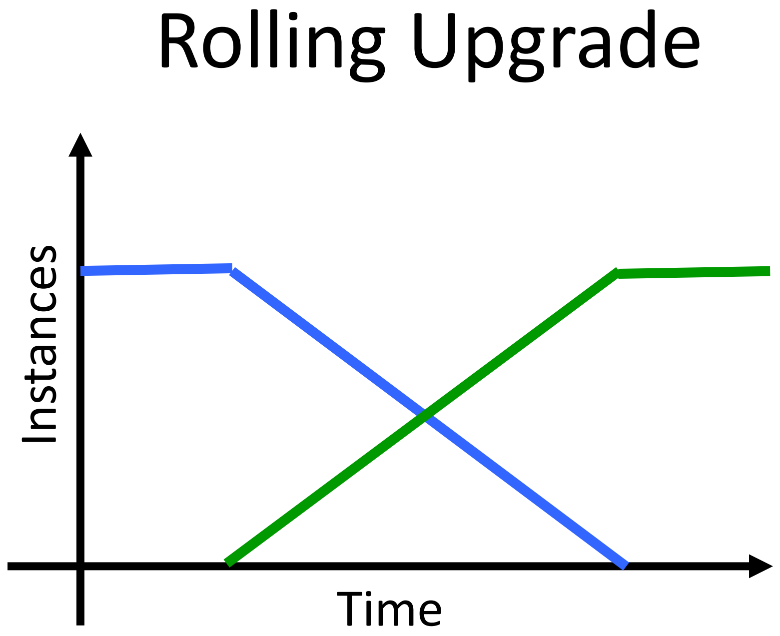


Image idea from https://www.slideshare.net/imesh/deep-dive-into-kubernetes-part-1

Rolling Updates

- Applications need to be updated
  - Pods with new images
- Rolling updates allow deployments to be updated without any downtime
  - Replace old Pods with the new gradually
  - When old Pods are no longer serving request
- Alternative to rolling updates is ‘recreate’
  - Kill all existing Pods before creating new ones

Rolling Deployment vs Recreate



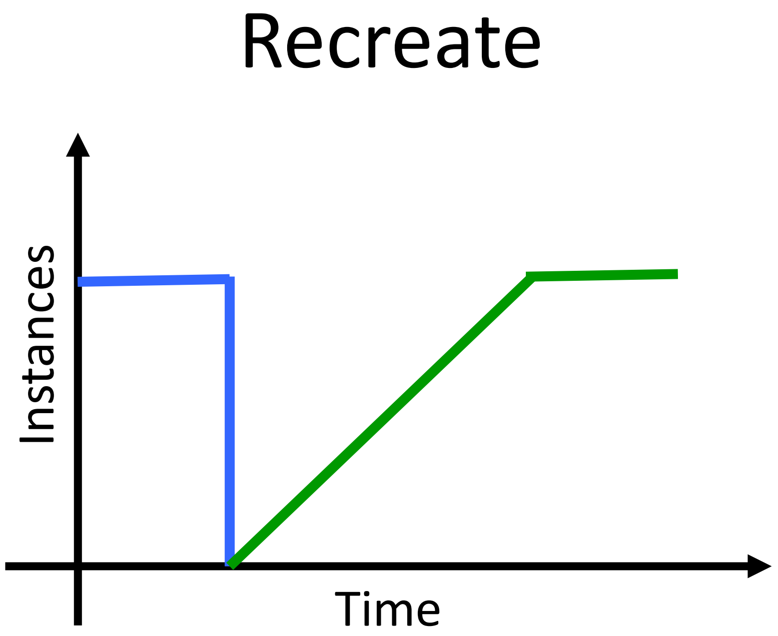
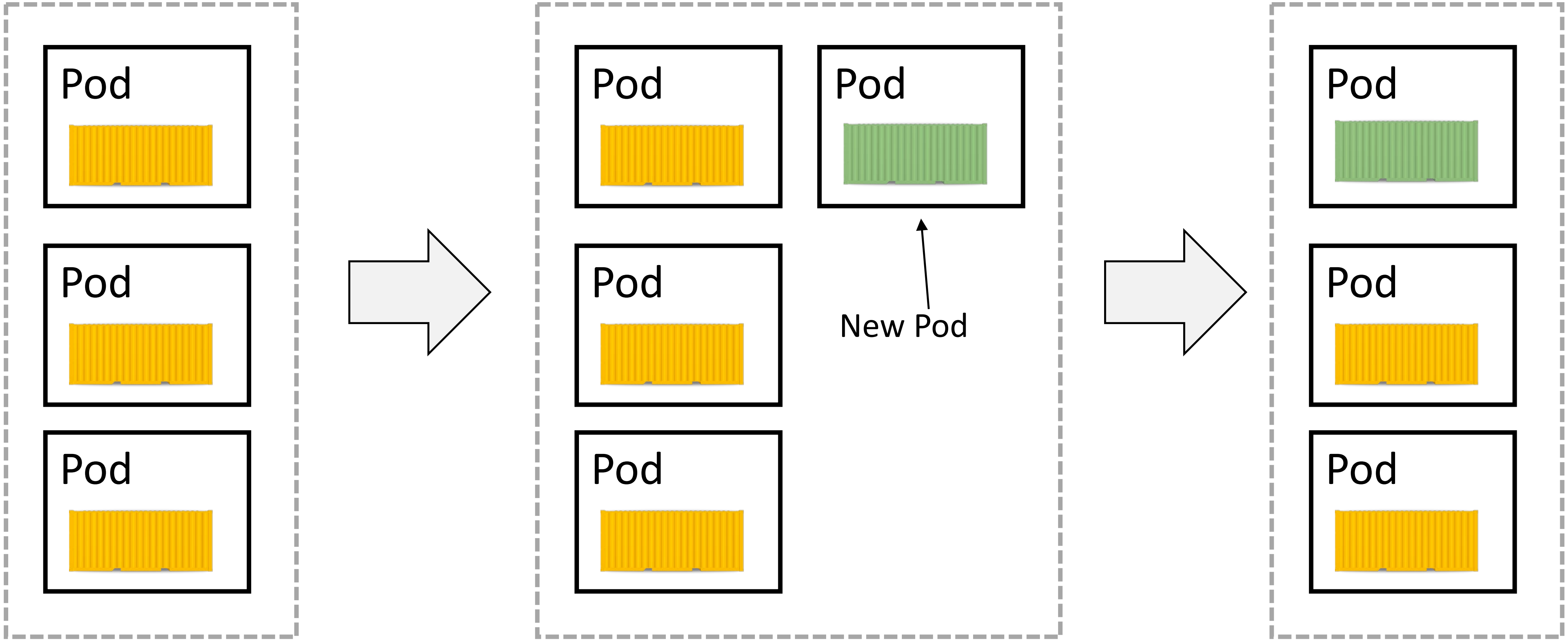


Image idea from https://speakerdeck.com/luxas/intro-to-the-cloud-native-world-of-kubernetes-november-2018

Rolling Update Illustrated

```
maxSurge: 1
maxUnavailable: 0
```



Deployment
Replica = 3

Defining a Rolling Update

Number of seconds for Kubernetes to wait for the application to be ready

Use the rolling update strategy for this deployment with 3 replicas

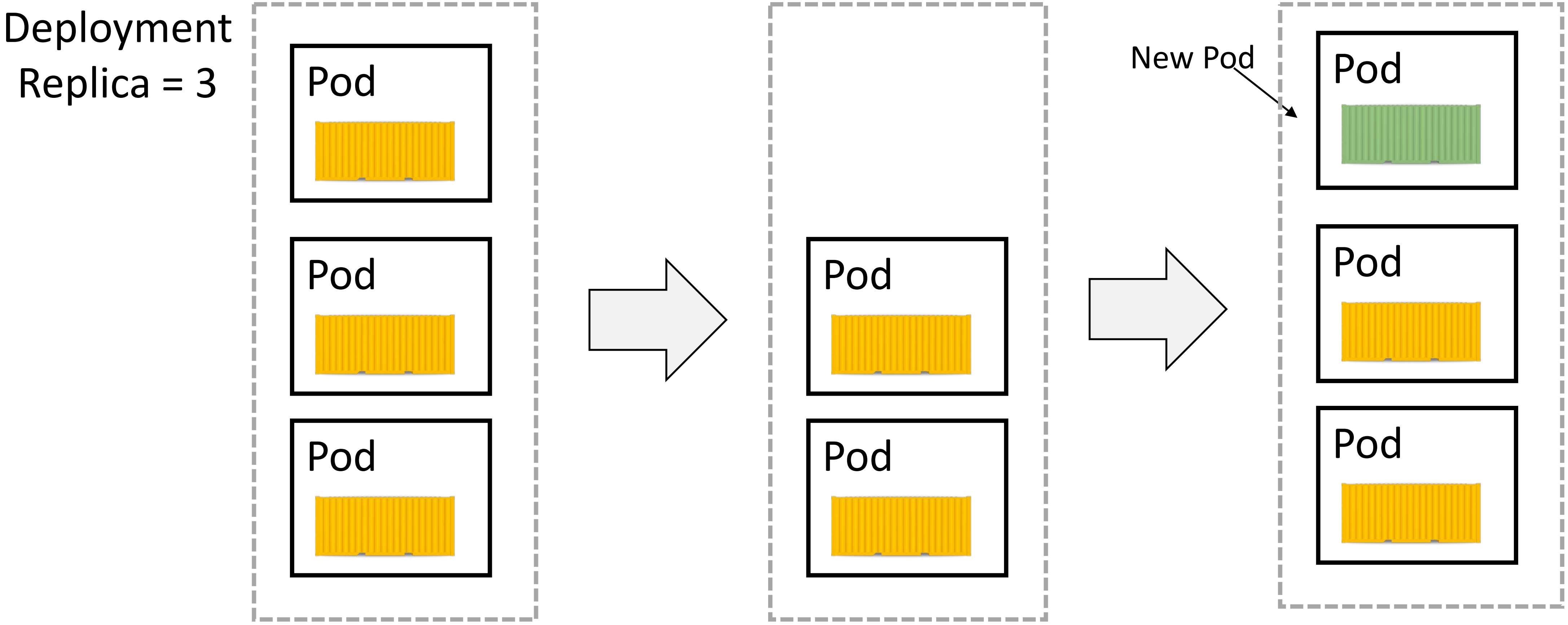
```
apiVersion: apps/v1
kind: Deployment
...
spec:
  replicas: 3
  minReadySeconds: 5
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
```

The policy for updating the deployment is controlled by `maxSurge` and `maxUnavailable`

Pods in this deployment will be updated one at a time (`maxSurge`). At any time the number of Pods will not fall below 3 (`maxUnavailable`)

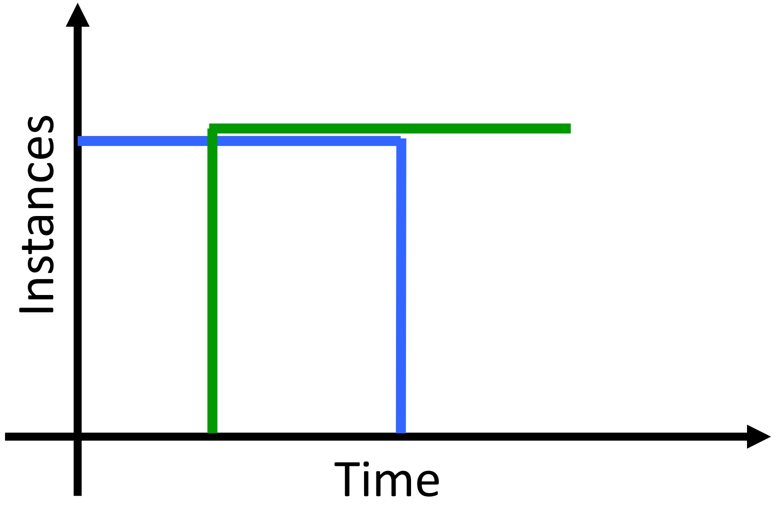
Rolling Update Illustrated

```
maxSurge: 0
maxUnavailable: 1
```

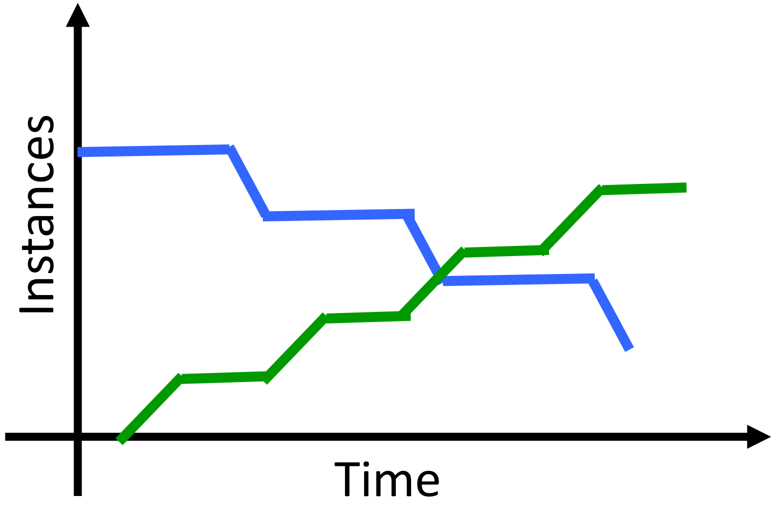



```
kubectl apply -f updated_deployment.yml
```

Rolling Updates




```
replicas: 5
maxSurge: 5
maxUnavailable: 0
```




```
replicas: 5
maxSurge: 1
maxUnavailable: 1
```

Image idea from https://speakerdeck.com/luxas/intro-to-the-cloud-native-world-of-kubernetes-november-2018

Rollback

```
kubectl apply -f dep-v1.yml
kubectl apply -f svc.yml

kubectl apply -f dep-v2.yml
```

```
kubectl rollout history deployment myapp-deployment

REVISION     CHANGE-CAUSE
1
2
```

```
kubectl rollout undo deployment myapp-deployment --to-revision=1
```

Managing Updates

- Apply an update

```
kubectl apply -f deployment-next.yml
```

- Check update status

```
kubectl rollout status deployment <deployment_name>
```

- See the revision history of the deployment

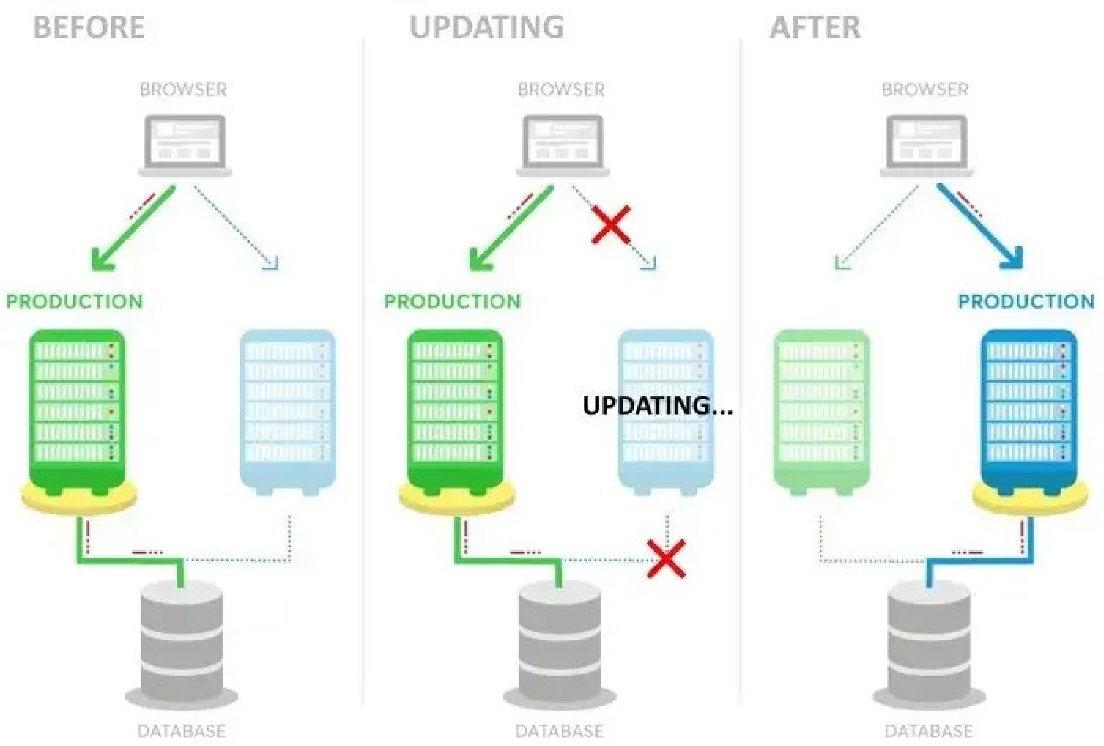
```
kubectl rollout history deployment <deployment_name>
```

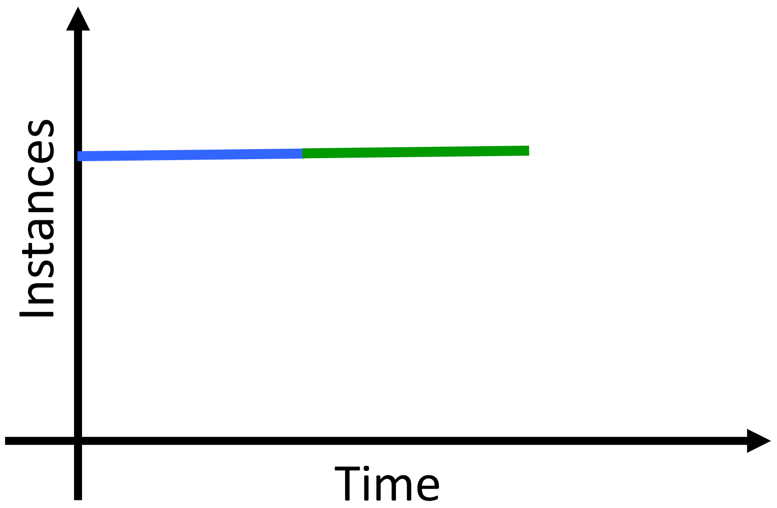
- Rollback to a previous version

```
kubectl rollout undo deployment <deployment_name> \
  --to-revision=<rev>
```

Blue Green Deployment

- Release model where 2 versions of an application is deployed side-by-side
  - Old version is called blue, the new version is called green
- When the green deployment is ready to receive traffic, reconfigure load balancer to forward request from blue to green
- Blue is maintained for a period before decommissioning
  - For rollback if there are issues with the green deployment





Example - Blue Green Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deploy-v1
  namespace: prod
spec:
  replicas: 3
  selector:
   matchLabels:
     name: myapp-po
     version: v1
  …
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deploy-v2
  namespace: prod
spec:
  replicas: 3
  selector:
   matchLabels:
     name: myapp-po
     version: v2
  …
```

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-svc
  namespace: prod
spec:
  type: ClusterIP
  selector:
    version: v1
```

Change Service selector from v1 to v2 when v2 pods are ready

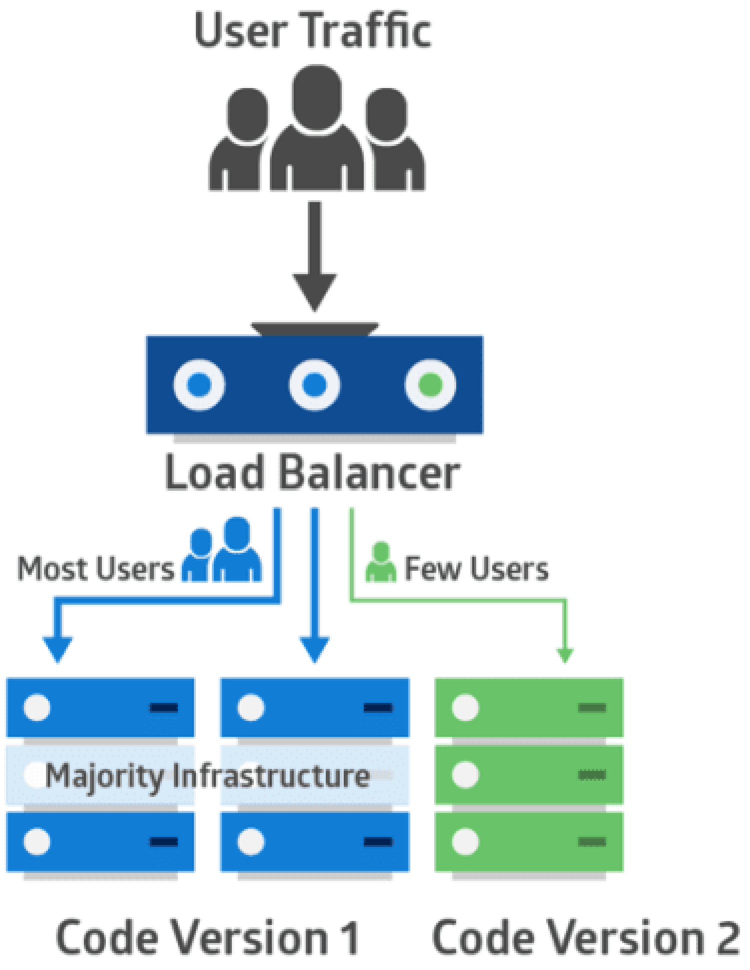
```
kubectl get po -n prod
  -l name=myapp-po,version=v2 \
  -o custom-columns='NAME:.metadata.name,READY:.status.conditions[?(@type=="Ready")].status'
```

Select pods with these labels only

Custom column for displaying pod conditions

Canary Deployment

- Making staged deployment where new releases is tested with a small subset of users
  - Part of request, e.g. 30% is served by the canary
- Objective of canary is to
  - Solicit feedback from users
  - Test the new release in production, if there are issues, will only affect a small percentage of the users



Canary Deployment

- Vanilla Kubernetes does not support canary deployment
- Use multiple deployment to mix different versions of the same application
  - Assign a common label to both deployments
  - Service forwards traffic to pods by selecting the common label
  - If the split is 1 in 4, 25% of the request goes to the new application, then deploy 1 new pod and 3 old pods
- Other methods of creating canary deployment
  - Ingress Nginx proprietary feature for marking an endpoint as canary
  - Istio using virtual service to split the traffic between different services

Example – Canary Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deploy-v1
  namespace: prod
spec:
  replicas: 3
  selector:
   matchLabels:
     name: myapp-po
     version: v1
  …
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deploy-v2
  namespace: prod
spec:
  replicas: 1
  selector:
   matchLabels:
     name: myapp-po
     version: v2
  …
```

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-svc
  namespace: prod
spec:
  type:
  selector: ClusterIP
    version: myapp-po
```

Common label on both the deployments

Use the service to randomly route incoming traffic to both the deployments using the common label(s)

Appendix

Architecture