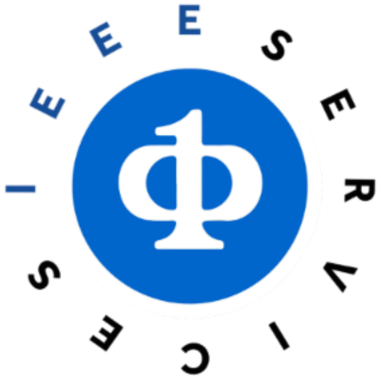
Introduction to Kubernetes

Instructor:
John J Rofrano
Senior Technical Staff Member, DevOps Champion
IBM T.J. Watson Research Center
rofrano@us.ibm.com
@JohnRofrano



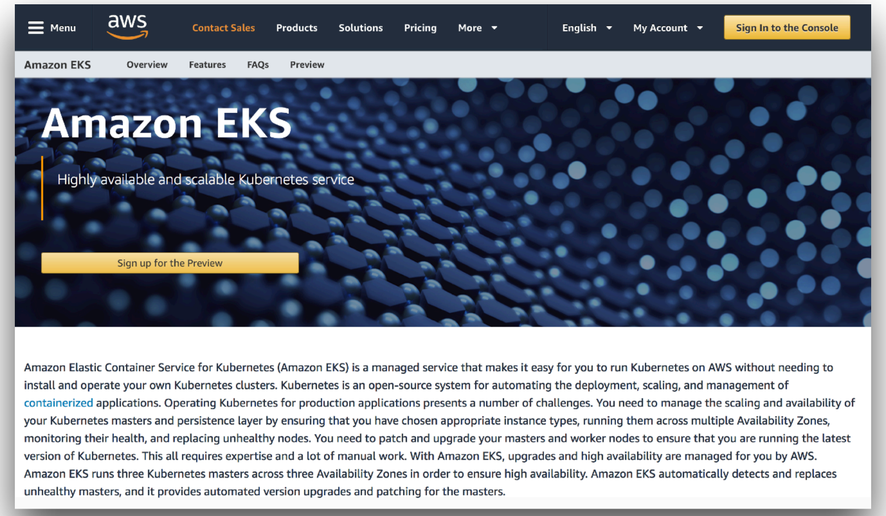
What will you lean?

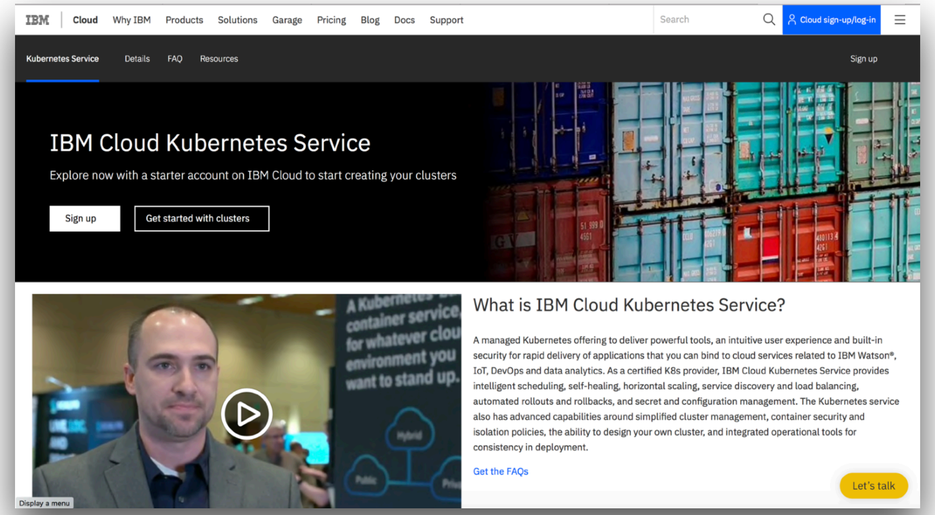
- What is Kubernetes
- Why would you use a Kubernetes for deployment
- Kubernetes Architecture Overview
- How to deploy your own containers in Kubernetes

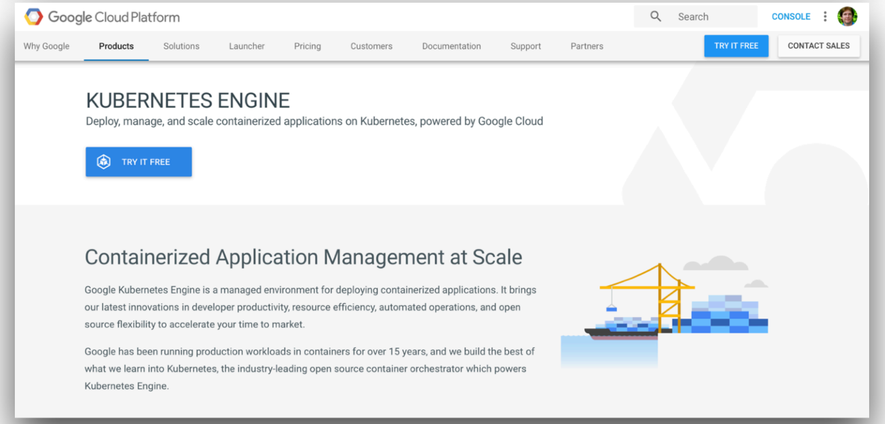
Kubernetes

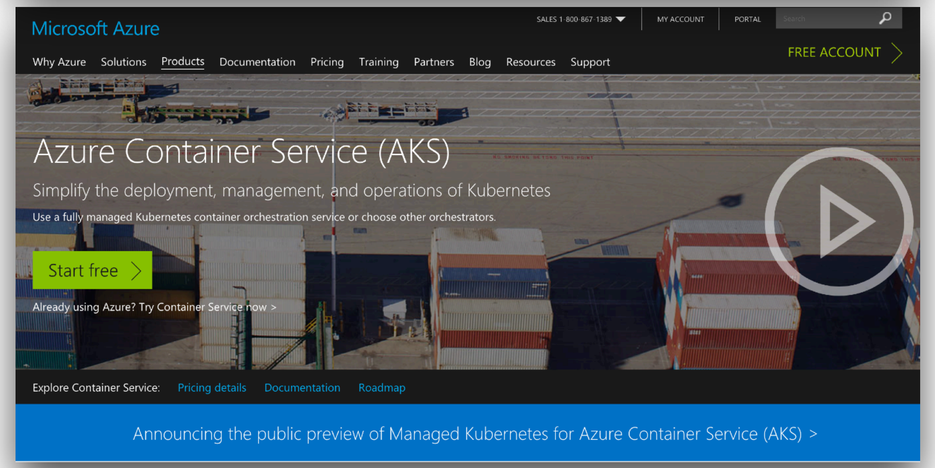
Kubernetes is the de facto Orchestrator for Containers

Every major cloud provider support Kubernetes









Why choose Kubernetes?

- An open-source system for automating deployment, scaling, and management of containerized applications.
- No Vendor Lock-In
- Large Community of support
- Robust platform for container orchestration
- Based on 15 years of experience of running production workloads at Google, combined with best-of-breed ideas and practices from the community

Kubernetes is an Orchestration Platform

- **Scheduling**
  decide where containers run
- **Lifecycle and Health**
  keep containers running and restart them if they fail
- **Scaling**
  grow and shrink deployments as needed
- **Naming and Discovery**
  help containers find each other
- **Load Balancing**
  distribute traffic across containers

...and a whole lot more

Why Do You Need Container Orchestration?

- Deploy applications to servers without worrying about specific servers
- Scale the application horizontally up and down
- Restore the application if the server on which it worked fails
  - This is called container auto-healing or rescheduling

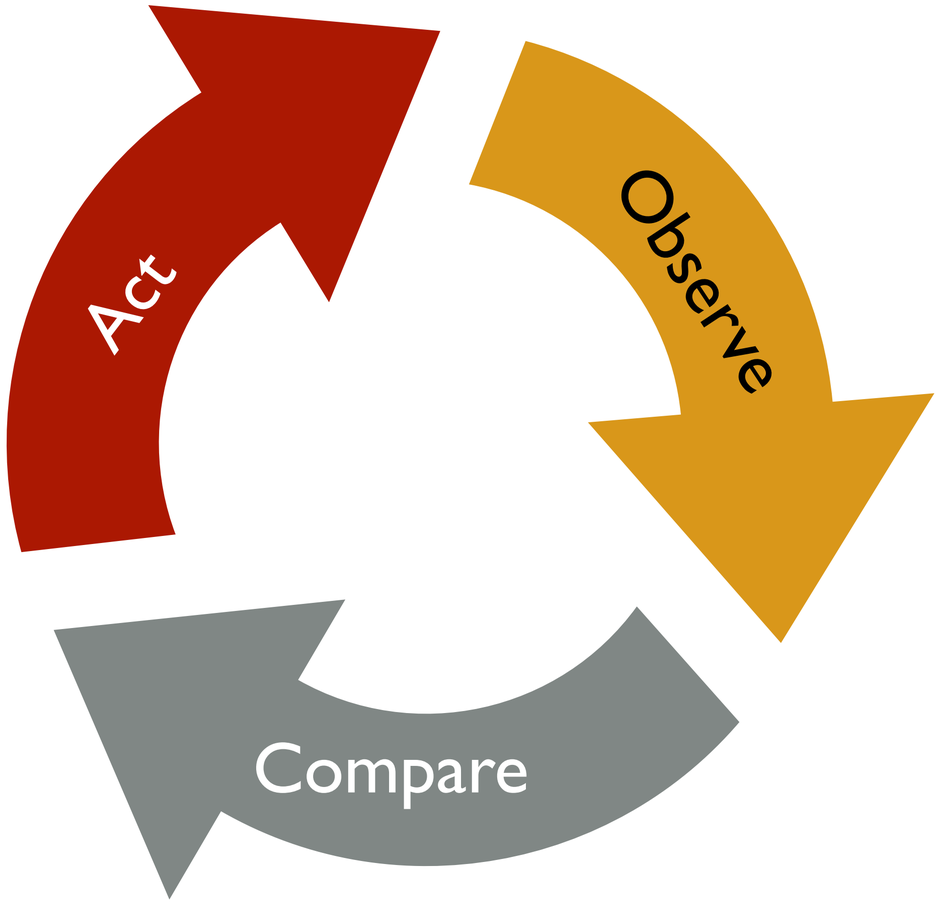
Kubernetes has a Declarative API

Kubernetes is a Declarative Model

You express the desired state

Kubernetes maintains it

What could be simpler?

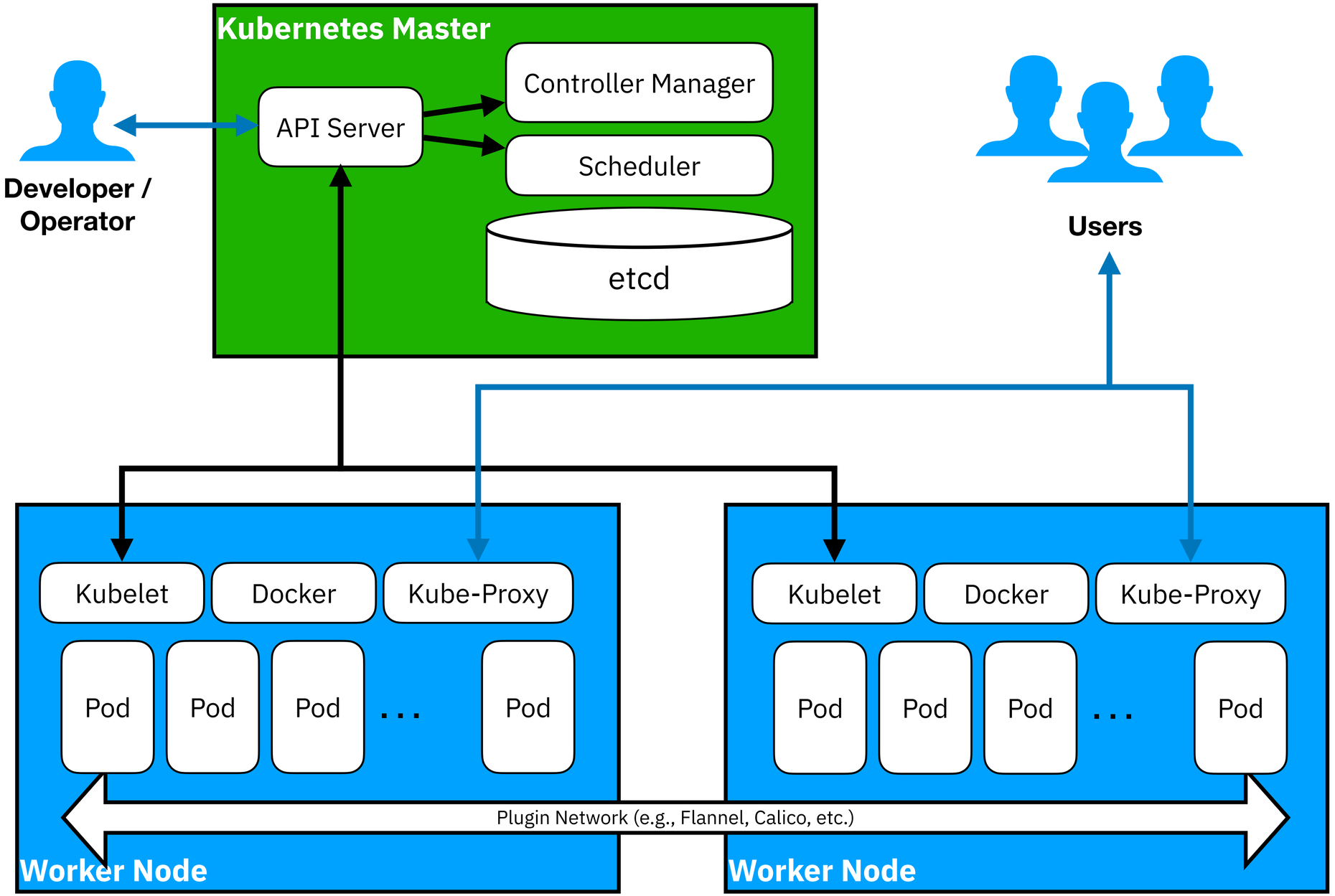


Kubernetes Architecture

https://github.com/kubernetes/community/blob/master/contributors/design-proposals/architecture/architecture.md

Kubernetes Architecture

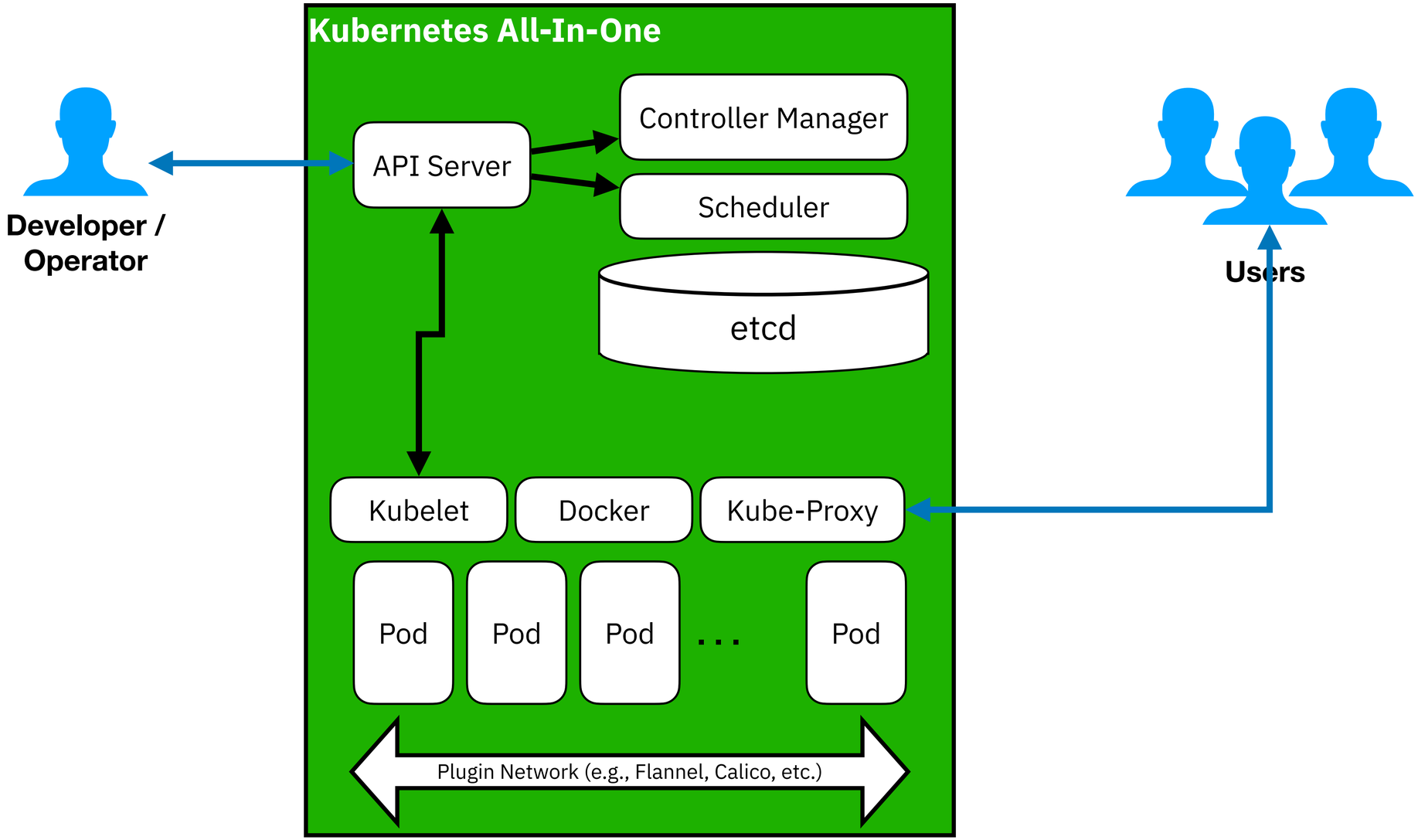
- There can be one or more Master Nodes (HA)
- There can be zero or more Worker Nodes
- Everyone communicates via the API Server
- Kubelet on each Worker Node acting as an agent
- Everything can be on one node for development use



Source: wikipedia

Kubernetes All-In-One

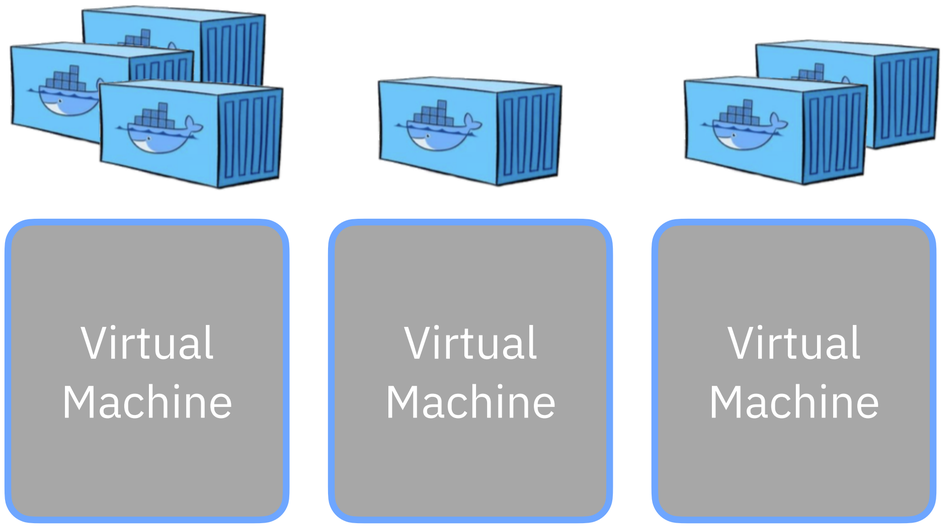
- You can run Kubernetes all in one VM for development work (not for production!)
- **MiniKube** is great for setting this up
- **MiniShift** will deploy a development version of RedHat OpenShift which uses Kubernetes



Kubernetes is the new "Cloud OS"

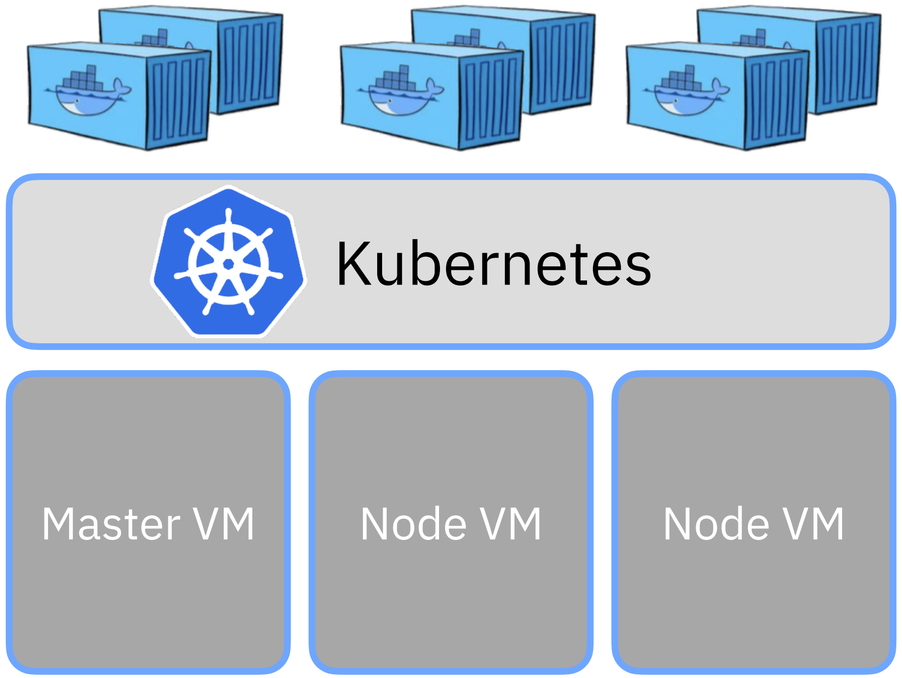
Managing Containers with VMs

- SysAdmins must decide where to place containers
- Workload balancing is manual

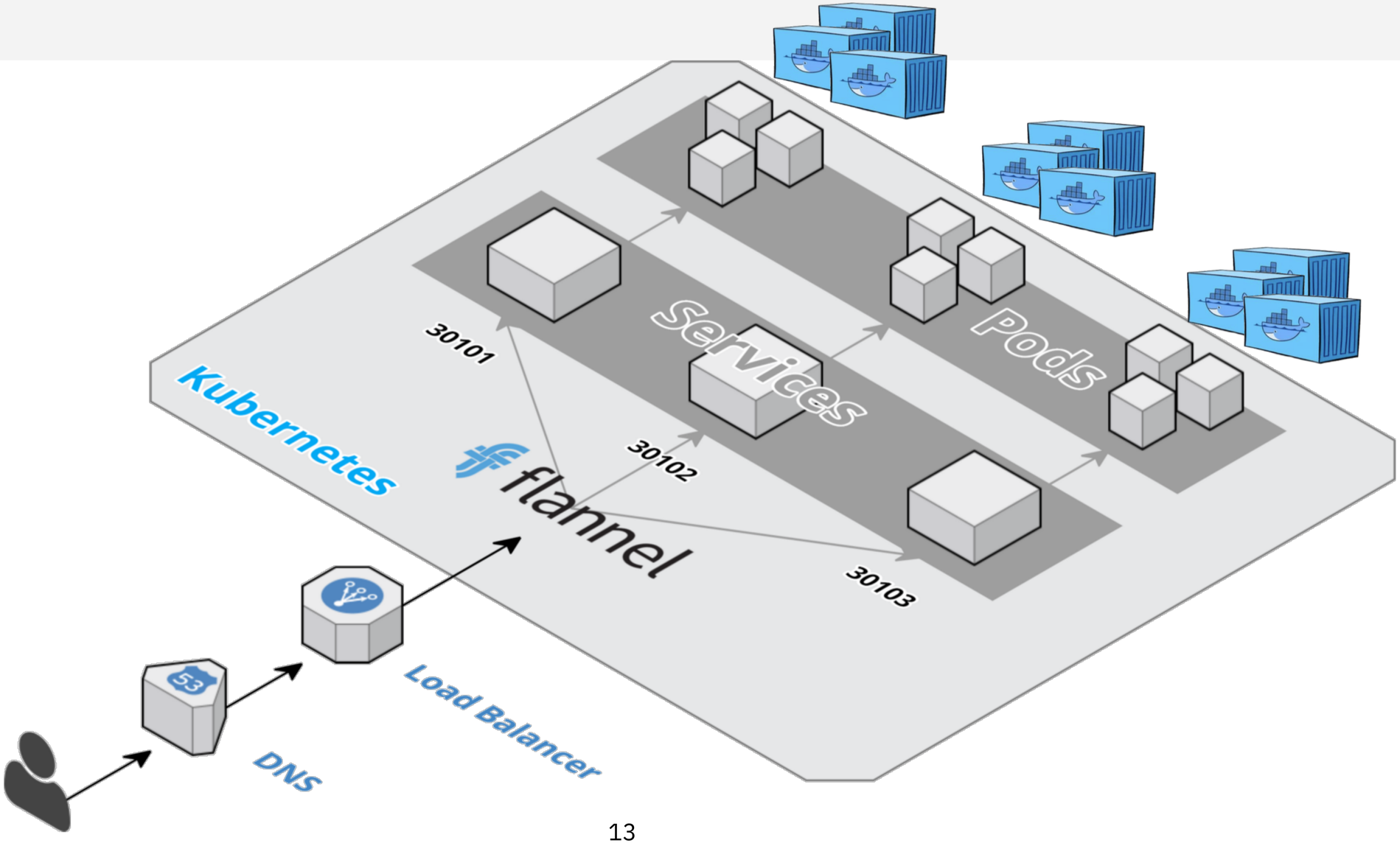


Managing Containers with Kubernetes

- Kubernetes schedules and optimizes workloads automatically



Kubernetes @ 20,000 ft.

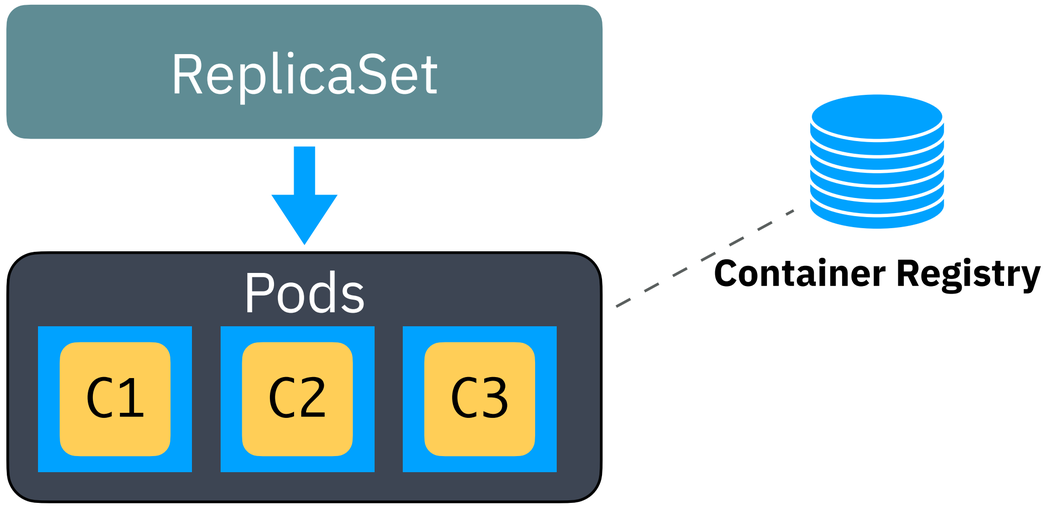


Kubernetes Pods

- Pods
  - Containers run in Pods
  - The Pod is the smallest deployment possible
  - Containers are deployed from a container registry (public or private)

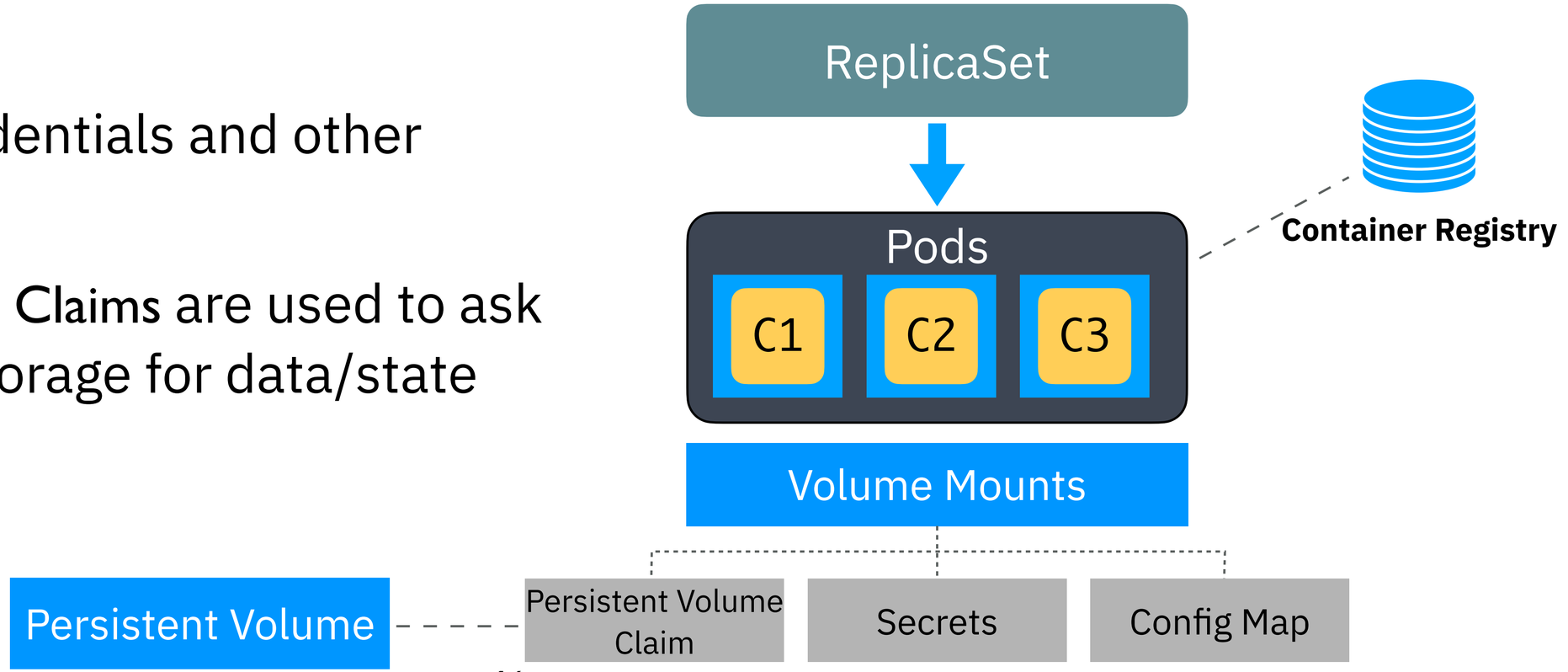
Kubernetes ReplicaSets

- ReplicaSet
  - ReplicaSets allow multiple copies if containers to be deployed
  - Each one in it's own Pod
  - If a container dies, the ReplicaSet will spawn a new one



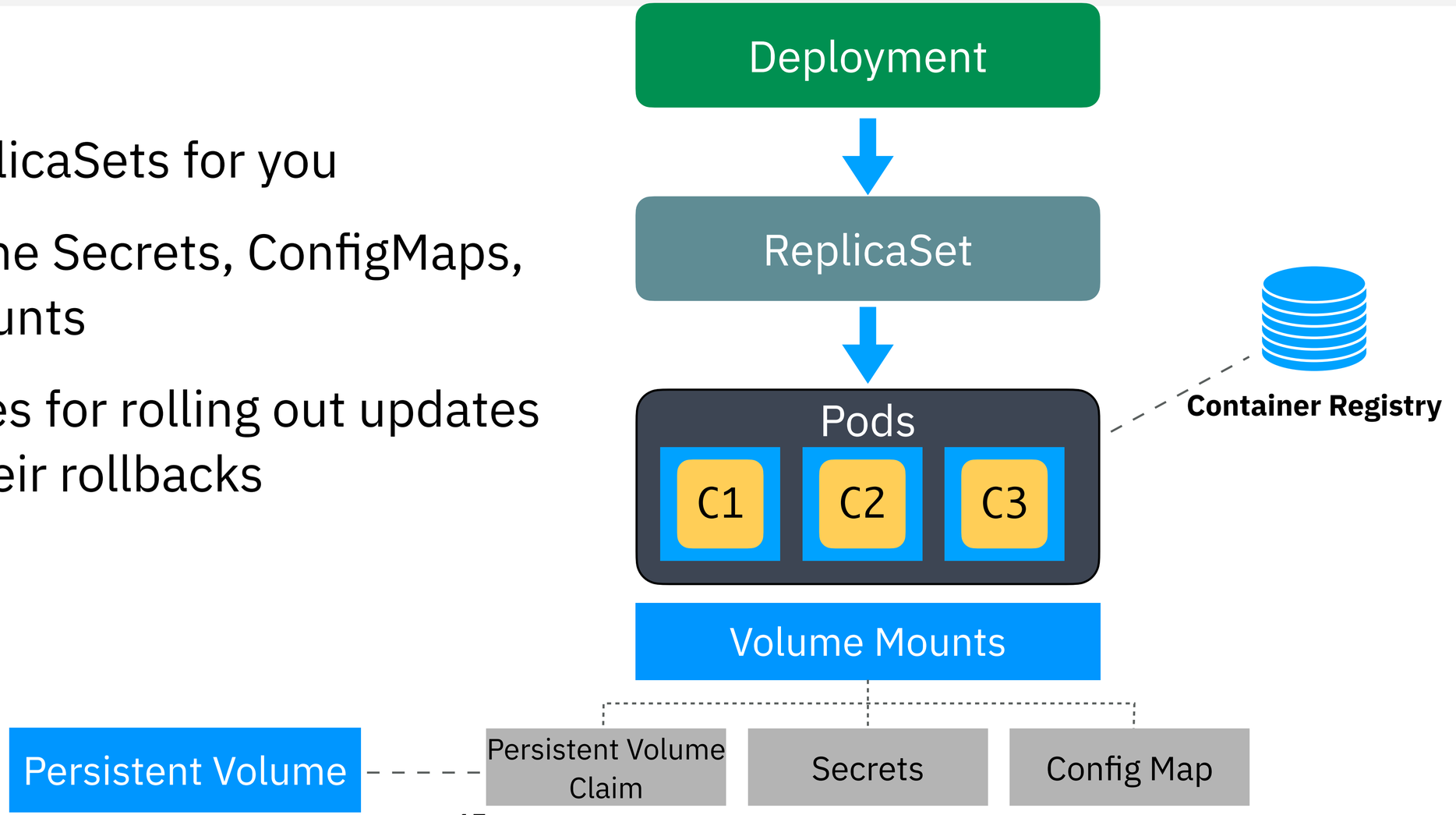
Kubernetes Volume Mounts

- Volumes
  - **ConfigMaps** hold configuration parameters
  - **Secrets** hold credentials and other secrets
  - **Persistent Volume Claims** are used to ask for persistent storage for data/state



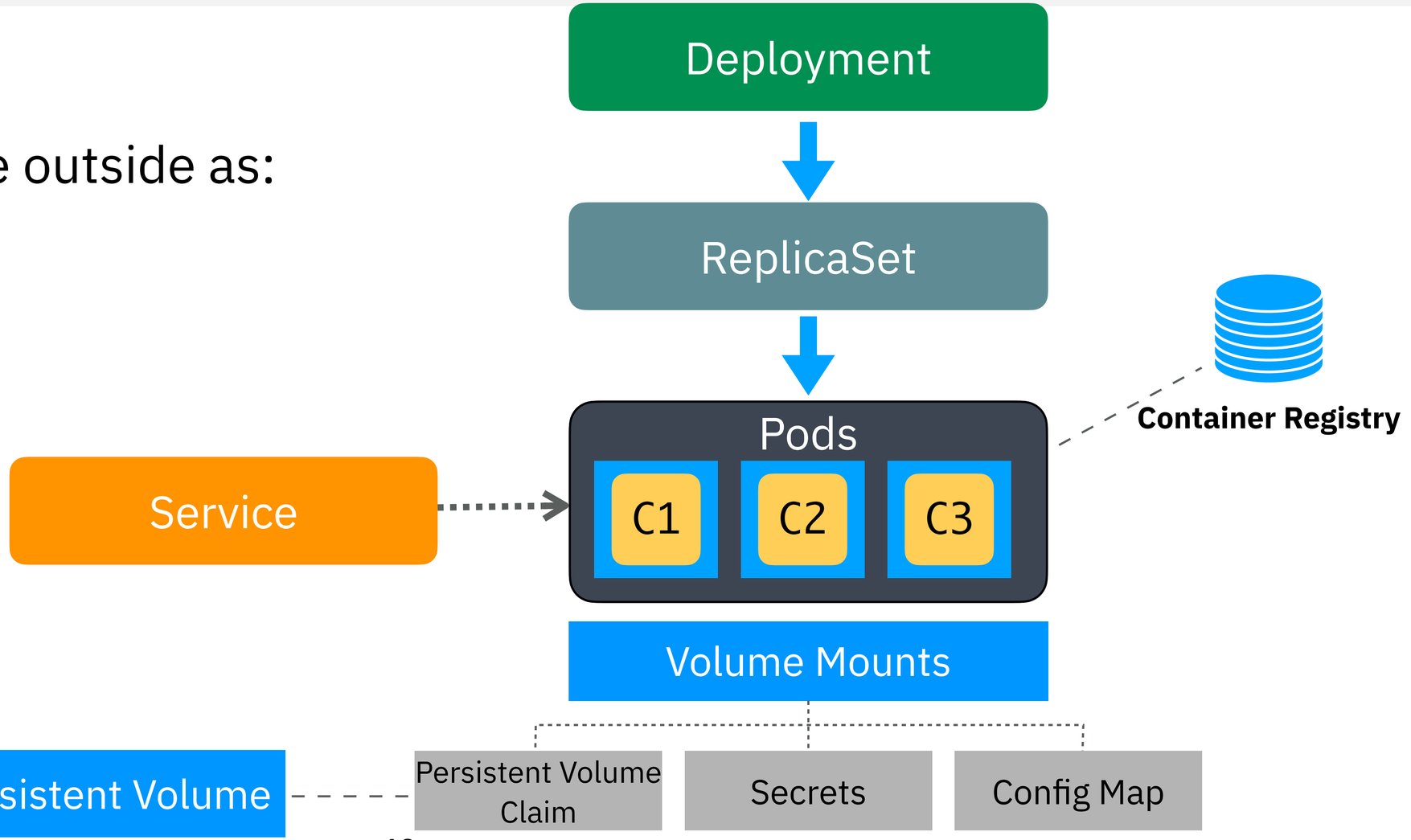
Kubernetes Deployments

- Deployment
  - Sets up the ReplicaSets for you
  - Also specified the Secrets, ConfigMaps, and Volume Mounts
  - Provides features for rolling out updates and handling their rollbacks



Kubernetes Service

- Service
  - Exposes Pods to the outside as:
    - ClusterIP
    - NodePort
    - Load Balancer

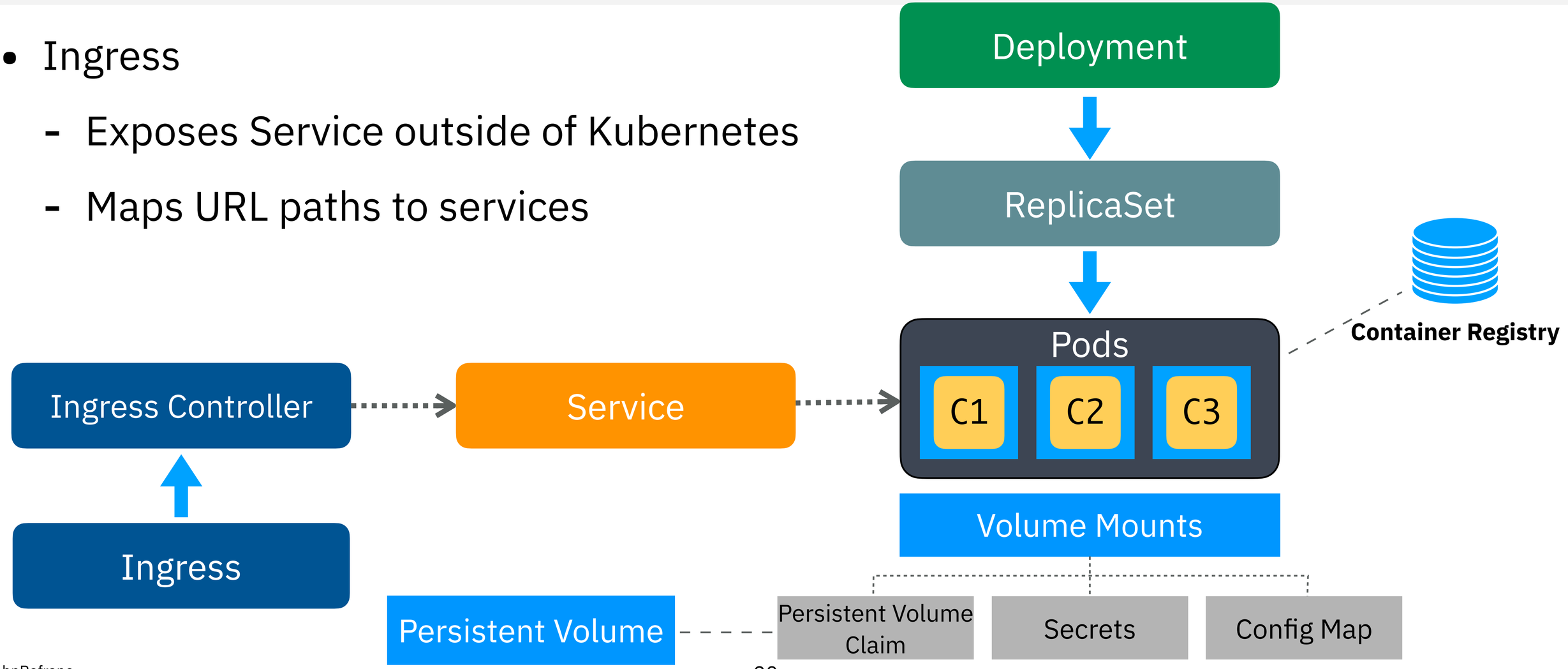


Types of Services

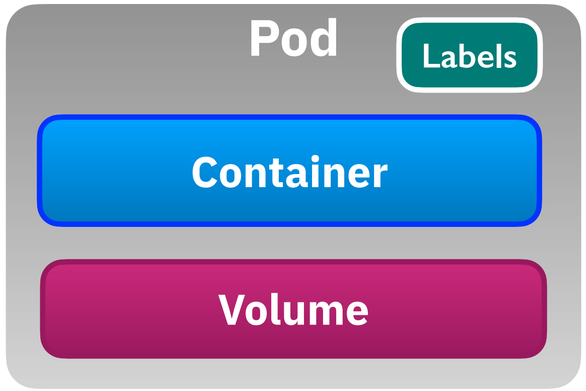
- **LoadBalancer** — Only available via cloud providers. Front end for service that balances the load across multiple backends from a single IP address
- **NodePort** — Exposes the service as an arbitrary port on every worker node in the cluster
- **ClusterIP** — Service is only accessible from other services within the cluster (no external exposure)

Kubernetes Ingress Controller

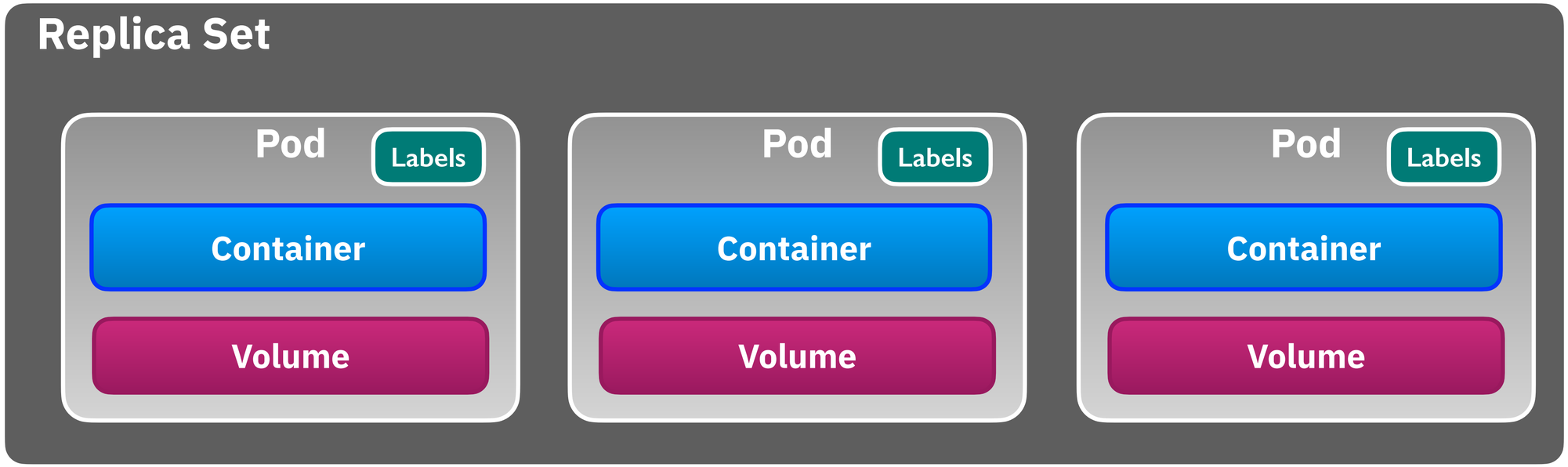
- Ingress
  - Exposes Service outside of Kubernetes
  - Maps URL paths to services



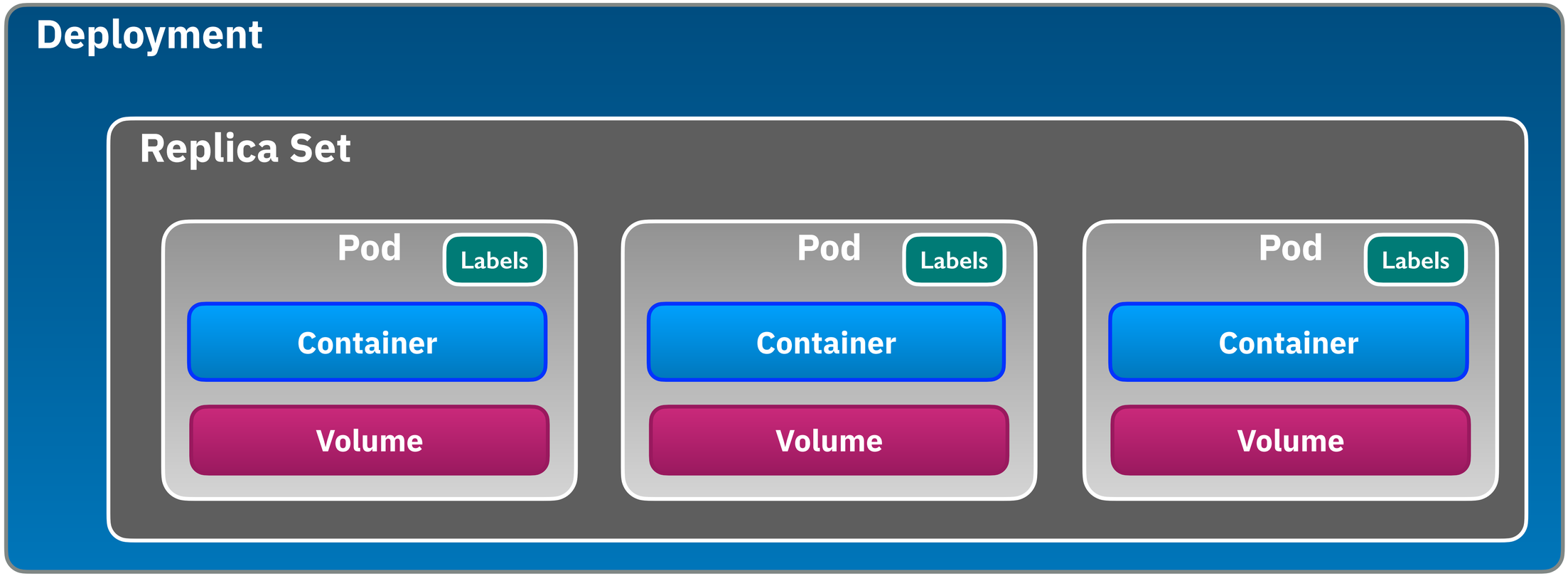
Services Map to Pods via Labels



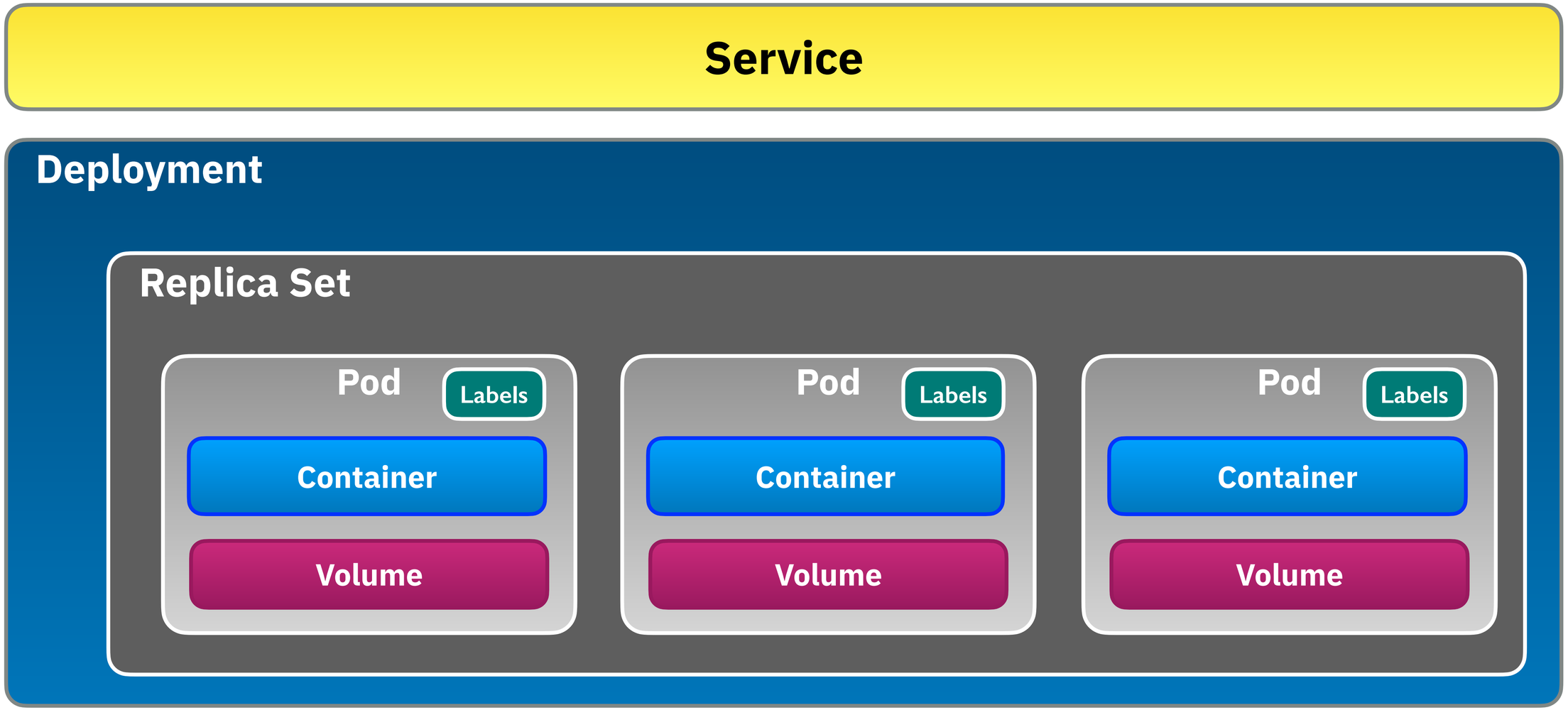
Services Map to Pods via Labels



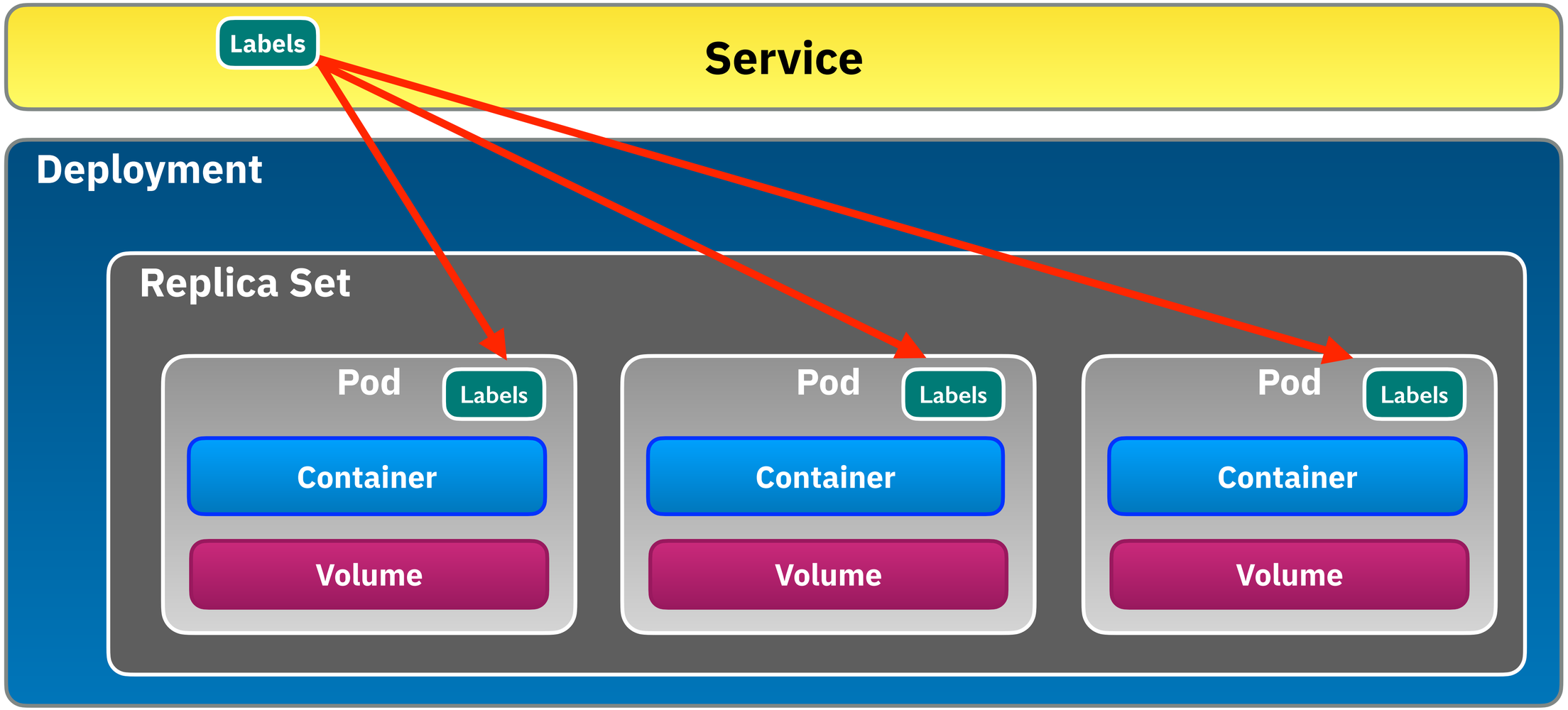
Services Map to Pods via Labels



Services Map to Pods via Labels



Services Map to Pods via Labels



Use Case for Multiple Containers in a Pod

- Pods can be used to host vertically integrated application stacks (e.g. LAMP), but their primary motivation is to support co-located, co-managed helper programs, such as:
  - Content management systems, file and data loaders, local cache managers, etc.
  - Log and checkpoint backup, compression, rotation, snapshotting, etc.
  - Data change watchers, log tailers, logging and monitoring adapters, event publishers, etc.
  - Proxies, bridges, and adapters
  - Controllers, managers, configurators, and updaters

https://kubernetes.io/docs/concepts/workloads/pods/pod/#uses-of-pods

Deployment using kubectl

- A containerized application can be deployed on Kubernetes using a deployment definition by executing a simple CLI command as follows:

```
$ kubectl create deployment <application-name> --image=<container-image>
```

Create a Service

- You can expose a deployment as a service using `kubectl expose`:

```
$ kubectl expose deployment <application-name> --type=<service-type> \
  --port=<port_number>
```

Valid <service-type>s are:

- LoadBalancer
- NodePort
- ClusterIP

Deploy using YAML

- You can create a deployment and a service using `yaml` files

```
$ kubectl create -f deployment.yaml
$ kubectl create -f service.yaml
```

- Update them

```
$ kubectl apply -f deployment.yaml
$ kubectl apply -f service.yaml
```

- And just as easily delete them

```
$ kubectl delete -f deployment.yaml
$ kubectl delete -f service.yaml
```

Deployment YAML

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hitcounter
spec:
  replicas: 3
  selector:
    matchLabels:
      app: hitcounter
  template:
    metadata:
      labels:
        app: hitcounter
    spec:
      containers:
      - image: hitcounter:1.0
        imagePullPolicy: IfNotPresent
        name: hitcounter
        ports:
        - containerPort: 8080
          protocol: TCP
      restartPolicy: Always
```

Service YAML

```yaml
apiVersion: v1
kind: Service
metadata:
  name: hitcounter-service
spec:
  type: NodePort
  selector:
    app: hitcounter
  ports:
    - name: primary
      protocol: TCP
      port: 8080
```

Linking Service to Pods

```yaml
apiVersion: v1
kind: Service
metadata:
  name: hitcounter-service
spec:
  type: NodePort
  selector:
    app: hitcounter
  ports:
    - name: primary
      protocol: TCP
      port: 8080
~
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hitcounter
spec:
  replicas: 3
  selector:
    matchLabels:
      app: hitcounter
  template:
    metadata:
      labels:
        app: hitcounter
    spec:
      containers:
      - image: hitcounter:1.0
        imagePullPolicy: IfNotPresent
        name: hitcounter
        ports:
        - containerPort: 8080
          protocol: TCP
      restartPolicy: Always
```

Linking Service to Pods

Look for Pods with this Label

```
 1 apiVersion: v1
 2 kind: Service
 3 metadata:
 4   name: hitcounter-service
 5 spec:
 6   type: NodePort
 7   selector:
 8     app: hitcounter
 9   ports:
10     - name: primary
11       protocol: TCP
12       port: 8080
~
```

```
 1 apiVersion: apps/v1
 2 kind: Deployment
 3 metadata:
 4   name: hitcounter
 5 spec:
 6   replicas: 3
 7   selector:
 8     matchLabels:
 9       app: hitcounter
10   template:
11     metadata:
12       labels:
13         app: hitcounter
14     spec:
15       containers:
16       - image: hitcounter:1.0
17         imagePullPolicy: IfNotPresent
18         name: hitcounter
19         ports:
20         - containerPort: 8080
21           protocol: TCP
22       restartPolicy: Always
```

Linking Service to Pods

Will match this Pod

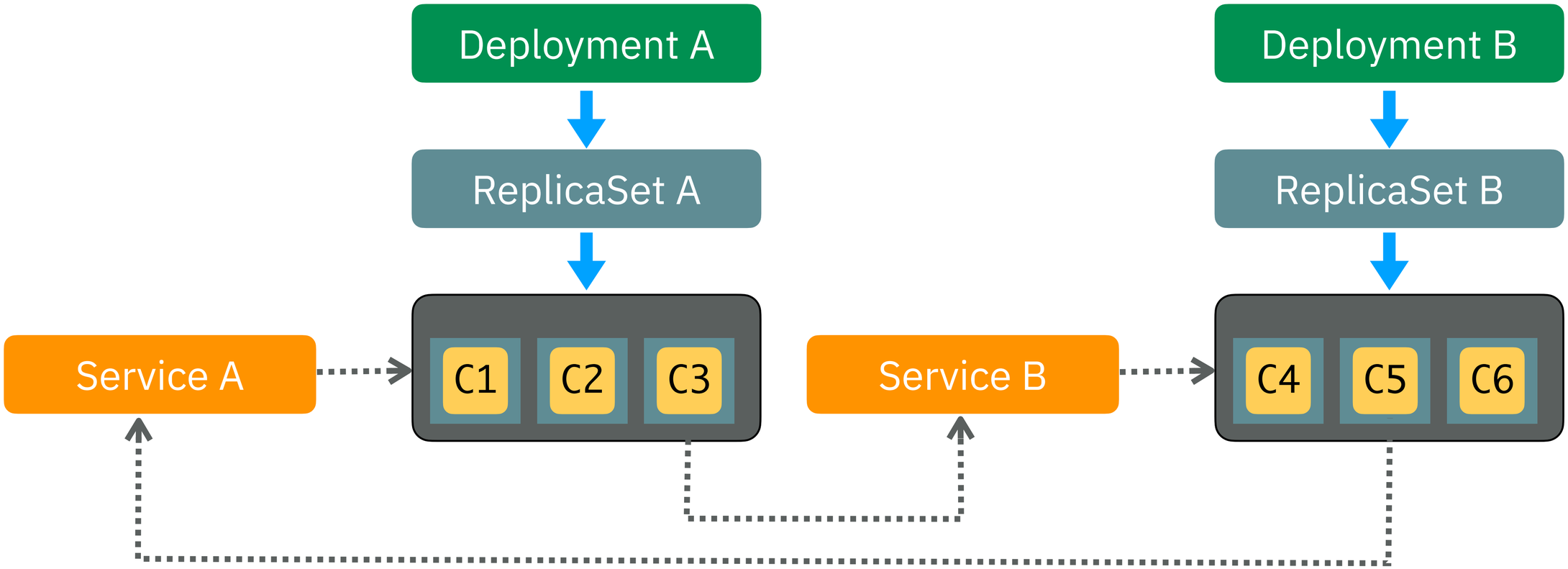
Look for Pods with this Label

```
 1 apiVersion: v1
 2
 3 metadata:
 4   name: hitcounter-service
 5 spec:
 6   type: NodePort
 7   selector:
 8     app: hitcounter
 9   ports:
10     - name: primary
11       protocol: TCP
12       port: 8080
~
```

```
 1 apiVersion: apps/v1
 2 kind: Deployment
 3 metadata:
 4   name: hitcounter
 5 spec:
 6   replicas: 3
 7   selector:
 8     matchLabels:
 9       app: hitcounter
10   template:
11     metadata:
12       labels:
13         app: hitcounter
14     spec:
15       containers:
16       - image: hitcounter:1.0
17         imagePullPolicy: IfNotPresent
18         name: hitcounter
19         ports:
20         - containerPort: 8080
21           protocol: TCP
22       restartPolicy: Always
```

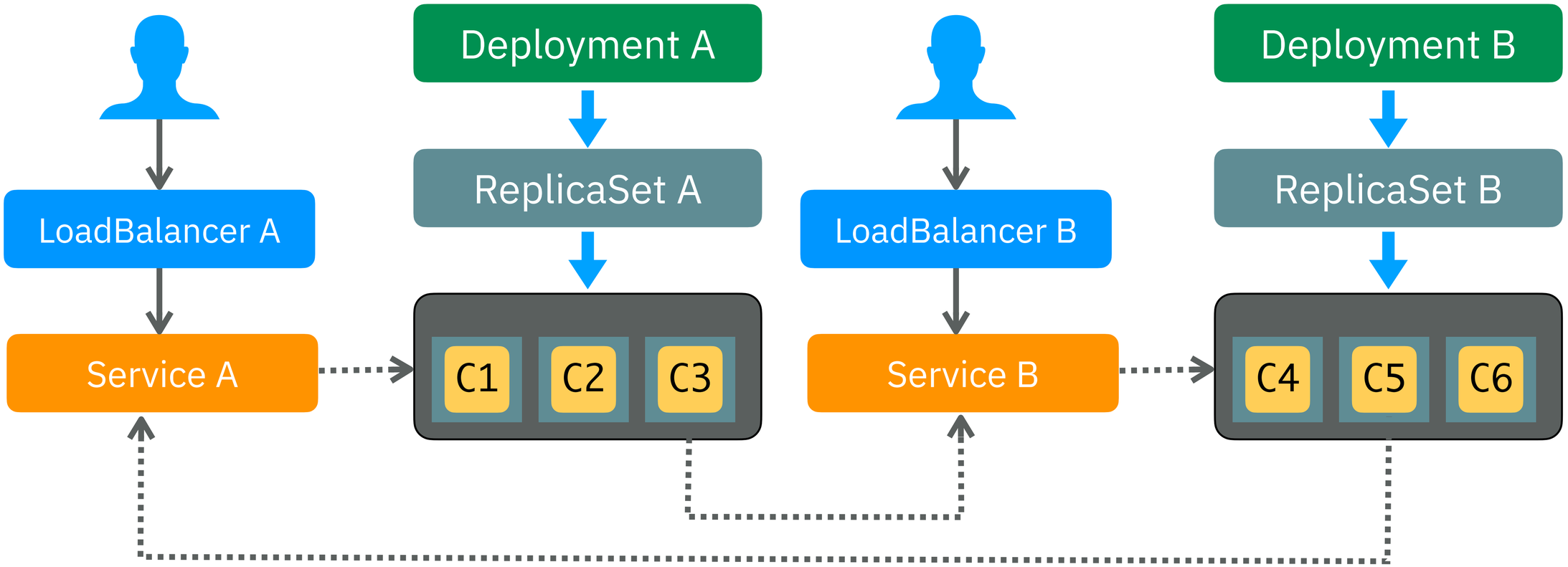
Service Discovery

- Kubernetes provides internal routing so services can find each other



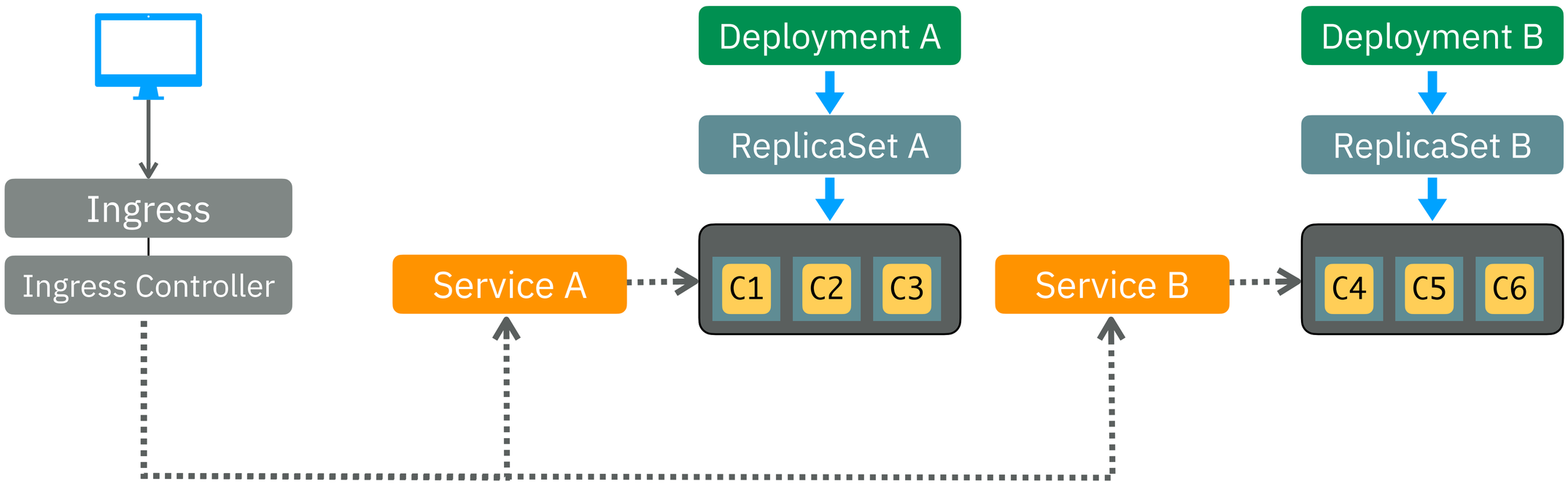
External Service Access

- LoadBalancers provide external access for a single service



External Routing

- Kubernetes provides an Ingress Controller to allow external network access or you can use NodePorts



Ingress Example

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ecommerce
spec:
  # tls:
  # - secretName: tls
  rules:
  - host: ecommerce.containers.mybluemix.net
    http:
      paths:
      - path: /shopcarts
        backend:
          serviceName: shopcart-service
          servicePort: 5000
      - path: /catalog
        backend:
          serviceName: catalog-service
          servicePort: 5000
      - path: /orders
        backend:
          serviceName: order-service
          servicePort: 5000
      - path: /recommedations
        backend:
          serviceName: recommedation-service
          servicePort: 5000
```

Ingress Example

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ecommerce
spec:
  # tls:
  # - secretName: tls
  rules:
  - host: ecommerce.containers.mybluemix.net
    http:
      paths:
      - path: /shopcarts
        backend:
          serviceName: shopcart-service
          servicePort: 5000
      - path: /catalog
        backend:
          serviceName: catalog-service
          servicePort: 5000
      - path: /orders
        backend:
          serviceName: order-service
          servicePort: 5000
      - path: /recommedations
        backend:
          serviceName: recommedation-service
          servicePort: 5000
```

Requests coming in on this URL

Ingress Example

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ecommerce
spec:
  # tls:
  # - secretName: tls
  rules:
  - host: ecommerce.containers.mybluemix.net
    http:
      paths:
      - path: /shopcarts
        backend:
          serviceName: shopcart-service
          servicePort: 5000
      - path: /catalog
        backend:
          serviceName: catalog-service
          servicePort: 5000
      - path: /orders
        backend:
          serviceName: order-service
          servicePort: 5000
      - path: /recommedations
        backend:
          serviceName: recommedation-service
          servicePort: 5000
```

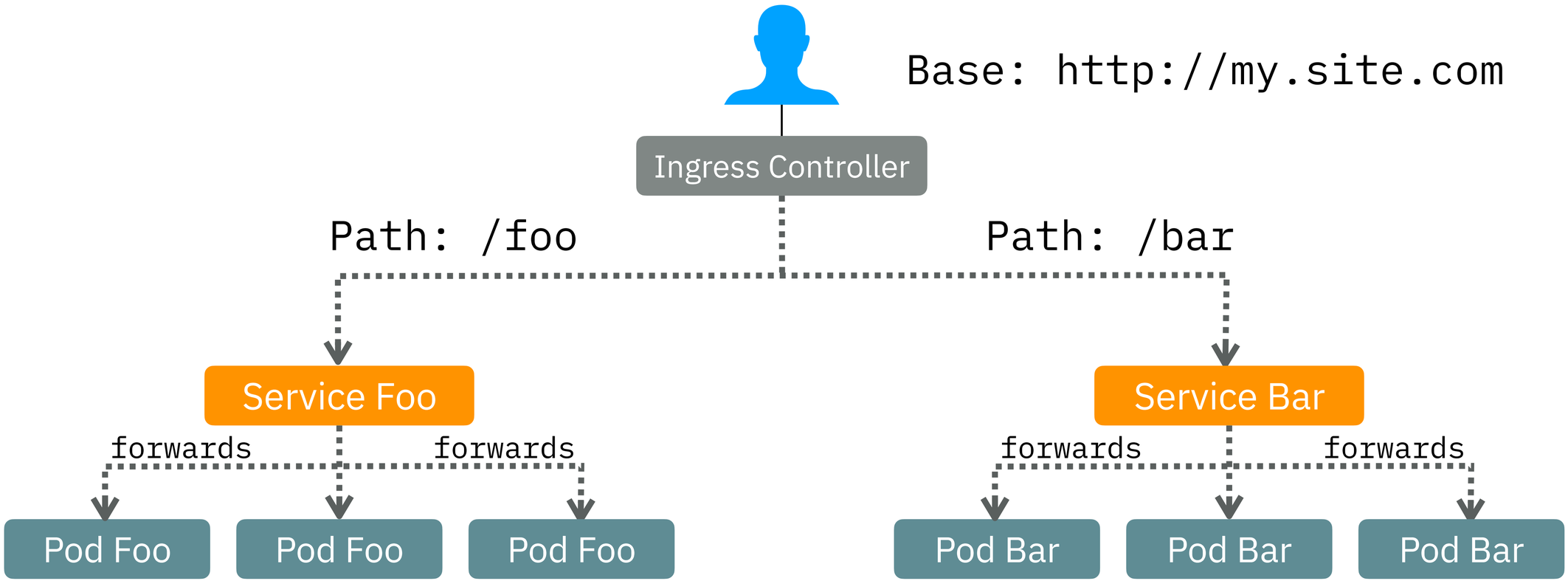
Ingress Example

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ecommerce
spec:
  # tls:
  # - secretName: tls
  rules:
  - host: ecommerce.containers.mybluemix.net
    http:
      paths:
      - path: /shopcarts
        backend:
          serviceName: shopcart-service
          servicePort: 5000
      - path: /catalog
        backend:
          serviceName: catalog-service
          servicePort: 5000
      - path: /orders
        backend:
          serviceName: order-service
          servicePort: 5000
      - path: /recommedations
        backend:
          serviceName: recommedation-service
          servicePort: 5000
```

Will be routed to these microservices

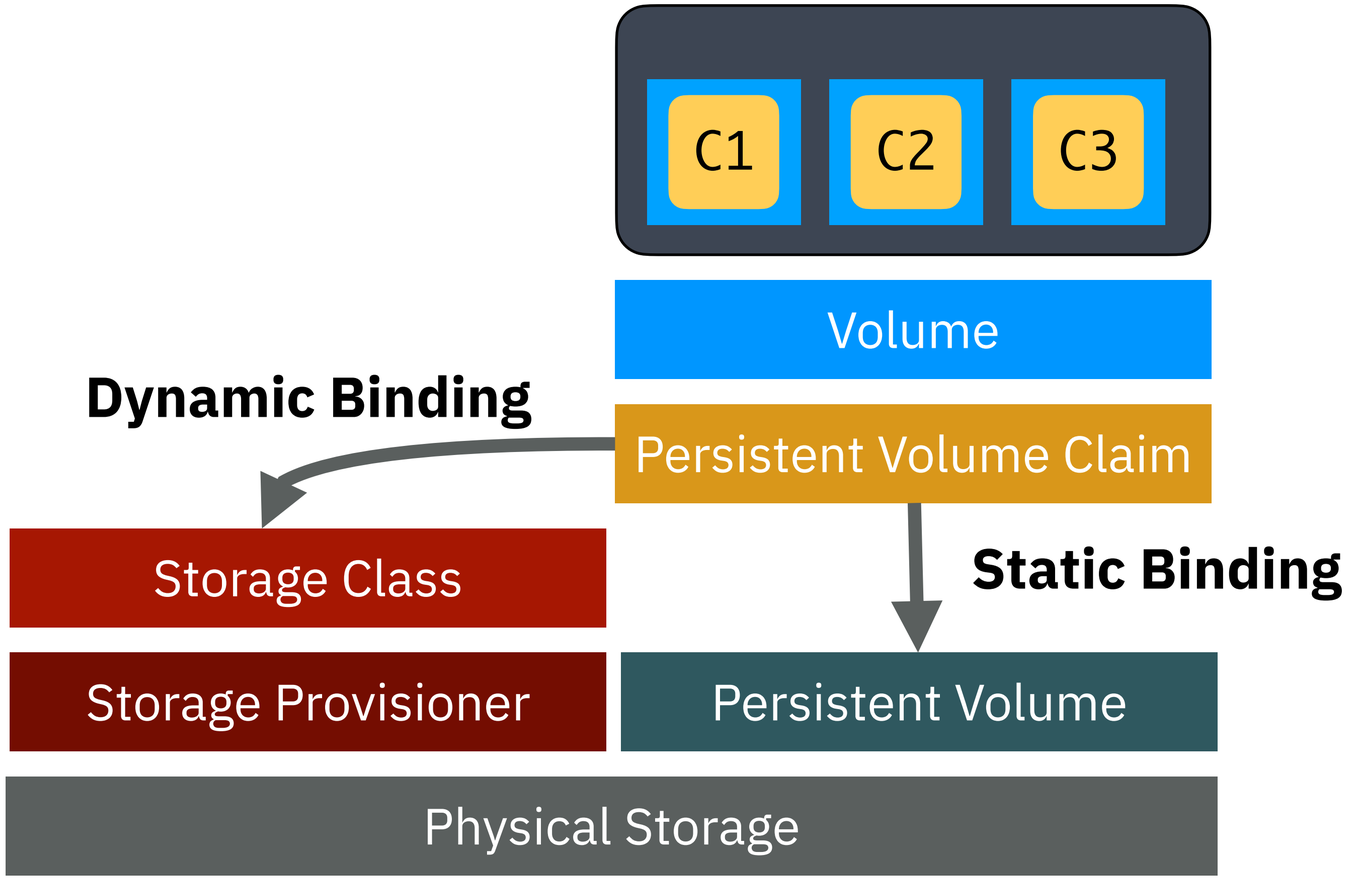
Ingress Controller

- Single entry point into multiple kubernetes services



Persistent Volumes

- Kubernetes loosely couples physical storage devices with containers by introducing an intermediate resource called persistent volume claims (PVCs).
- A PVC defines the disk size, disk type (ReadWriteOnce, ReadOnlyMany, ReadWriteMany) and dynamically links a storage device to a volume defined against a pod
- The binding process can either be done in a static way using PVs or dynamically be using a persistent storage provider



Example Volume Mount

- There is a volume named `redis-storage` and that is connected to the `redis` container via the VolumeMount: `/data/redis`

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis
    volumeMounts:
    - name: redis-storage
      mountPath: /data/redis
  volumes:
  - name: redis-storage
    emptyDir: {}
~
```

Example Volume Mount

- There is a volume named `redis-storage` and that is connected to the `redis` container via the VolumeMount: `/data/redis`

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis
    volumeMounts:
    - name: redis-storage
      mountPath: /data/redis
  volumes:
  - name: redis-storage
    emptyDir: {}
~
```

Volume named `redis-storage`

Example Volume Mount

- There is a volume named `redis-storage` and that is connected to the `redis` container via the VolumeMount: `/data/redis`

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis
    volumeMounts:
    - name: redis-storage
      mountPath: /data/redis
  volumes:
  - name: redis-storage
    emptyDir: {}
~
```

Mounted at: `/data/redis`

Volume named `redis-storage`

Configurations Management

(ConfigMap)

- Containers generally use environment variables for parameterizing their runtime configurations
- Kubernetes provides a way of managing more complex configuration files using a simple resource called ConfigMaps
- ConfigMaps can be created using directories, files or literal values using following CLI command:

```
$ kubectl create configmap <map-name> <data-source>

# map-name: name of the config map
# data-source: directory, file or literal value
```

Credentials Management

(Secrets)

- Similar to ConfigMaps, Kubernetes provides another valuable resource called Secrets for managing sensitive information such as passwords, OAuth tokens, and ssh keys.
- A secret can be created for managing basic auth credentials using the following way:

```
# write credentials to two files
$ echo -n 'admin' > ./username.txt
$ echo -n '1f2d1e2e67df' > ./password.txt

# create a secret
$ kubectl create secret generic app-credentials --from-file=./
username.txt --from-file=./password.txt
```

Credentials From Environment Variables

(Secrets)

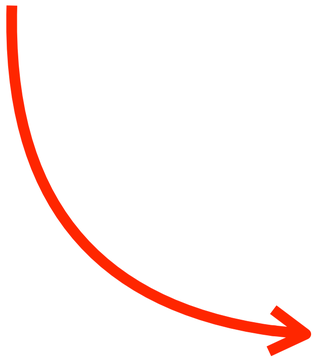
```
# export DATABASE_URI='postgres://admin:s3cr3t@postgres:5432/postgres'

# kubectl create secret generic db-creds --from-literal=database-uri=$DATABASE_URI
```

- Creates the following

```
apiVersion: v1
kind: Secret
data:
  database-uri: cG9zdGdyZXM6Ly9hZG1pbjpzM2NyM3RAcG9zdGdyZXM6NTQzMi9wb3N0Z3Jlcw==
```

Create Secrets from Literals

```
kubectl create secret generic dev-db-secret \
    --from-literal=username=devuser \
    --from-literal=password='s3cr3t'
```



```
apiVersion: v1
kind: Secret
metadata:
  name: dev-db-secret
type: Opaque
data:
  username: ZGV2dXNlcgo=
  password: czNjcjN0Cg==
```

Secret Yaml Example

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: ecommerce-apikey
  namespace: default
data:
  secret: <place base64 encoded secret here>
```

```
$ echo -n "this is my secret" | base64
dGhpcyBpcyBteSBzZWNyZXQ=
```

Secret Yaml Example

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: ecommerce-apikey
  namespace: default
data:
  secret: <place base64 encoded secret here>
```

Place the base64 encode string in secret

```
$ echo -n "this is my secret" | base64
dGhpcyBpcyBteSBzZWNyZXQ=
```

Using Secrets

```yaml
apiVersion: apps/v1
kind: Deployment
...
   spec:
      containers:
      - name: pet-demo
        image: pet-demo:v1
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 5000
          protocol: TCP
        env:
          - name: DATABASE_URI
            valueFrom:
              secretKeyRef:
                name: pet-creds
                key: binding
```

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: pet-creds
  namespace: default
data:
  binding: dGhpcyBpcyBteSBzZWNyZXQ=
```

Using Secrets

```
apiVersion: apps/v1
kind: Deployment
...
   spec:
      containers:
      - name: pet-demo
        image: pet-demo:v1
        imagePullPol
        ports:
        - containerPort: 5000
          protocol: TCP
        env:
          - name: DATABASE_URI
            valueFrom:
              secretKeyRef:
                name: pet-creds
                key: binding
```

Secrets exposed as environment variables

```
apiVersion: v1
kind: Secret
metadata:
  name: pet-creds
  namespace: default
data:
  binding: dGhpcyBpcyBteSBzZWNyZXQ=
```

Kubernetes Rolling Updates (Zero Downtime Deployments)

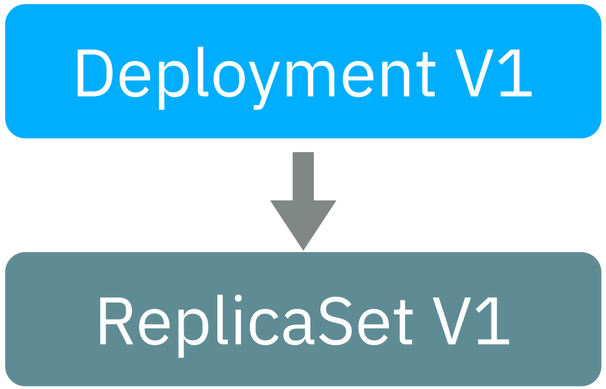
```
$ kubectl set image deployment/<application-name> <container-name>=<container-image-name>:<new-version>
```

Deployment V1

Deploying Version 1

Kubernetes Rolling Updates (Zero Downtime Deployments)

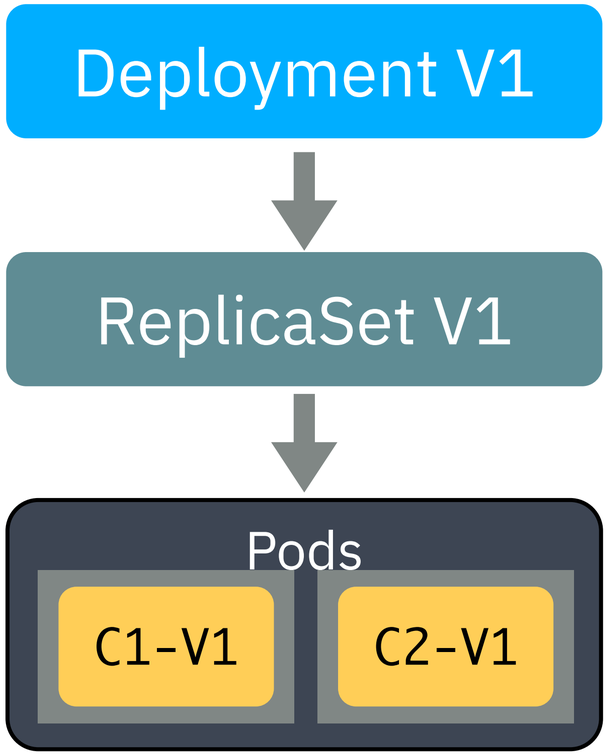
```
$ kubectl set image deployment/<application-name> <container-name>=<container-image-name>:<new-version>
```



Deploying Version 1

Kubernetes Rolling Updates (Zero Downtime Deployments)

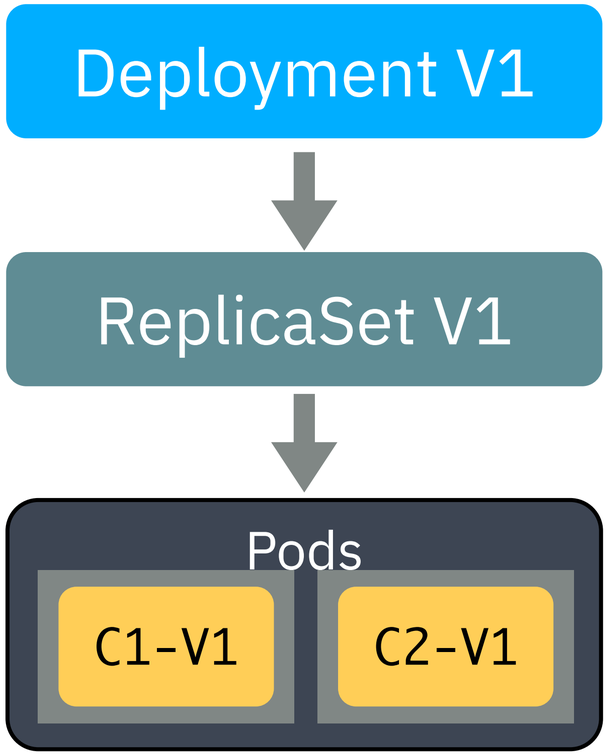
```
$ kubectl set image deployment/<application-name> <container-name>=<container-image-name>:<new-version>
```



Deploying Version 1

Kubernetes Rolling Updates (Zero Downtime Deployments)

```
$ kubectl set image deployment/<application-name> <container-name>=<container-image-name>:<new-version>
```

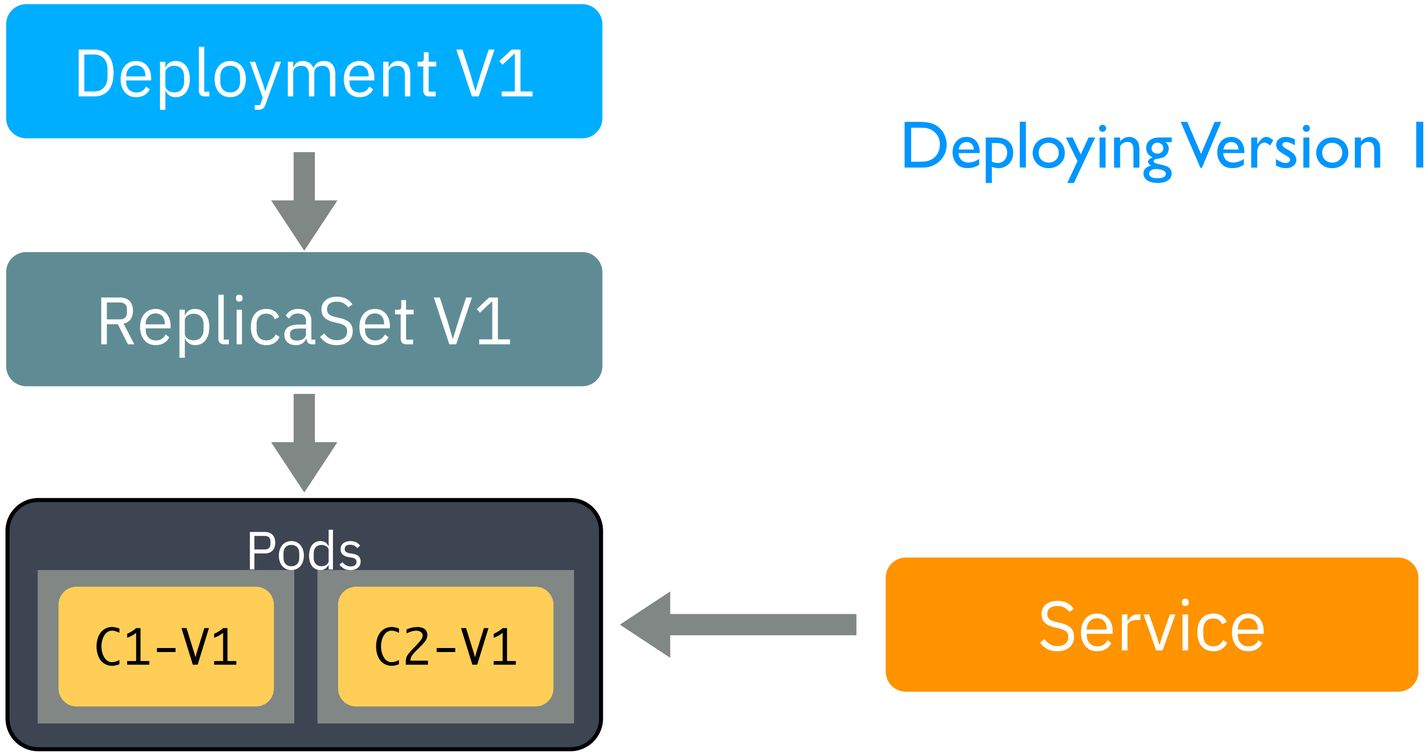


Deploying Version 1

Service

Kubernetes Rolling Updates (Zero Downtime Deployments)

```
$ kubectl set image deployment/<application-name> <container-name>=<container-image-name>:<new-version>
```



Deploying Version 1

Kubernetes Rolling Updates (Zero Downtime Deployments)

```
$ kubectl set image deployment/<application-name> <container-name>=<container-image-name>:<new-version>
```

Kubernetes Rolling Updates (Zero Downtime Deployments)

```
$ kubectl set image deployment/<application-name> <container-name>=<container-image-name>:<new-version>
```

Kubernetes Rolling Updates (Zero Downtime Deployments)

```
$ kubectl set image deployment/<application-name> <container-name>=<container-image-name>:<new-version>
```

Kubernetes Rolling Updates (Zero Downtime Deployments)

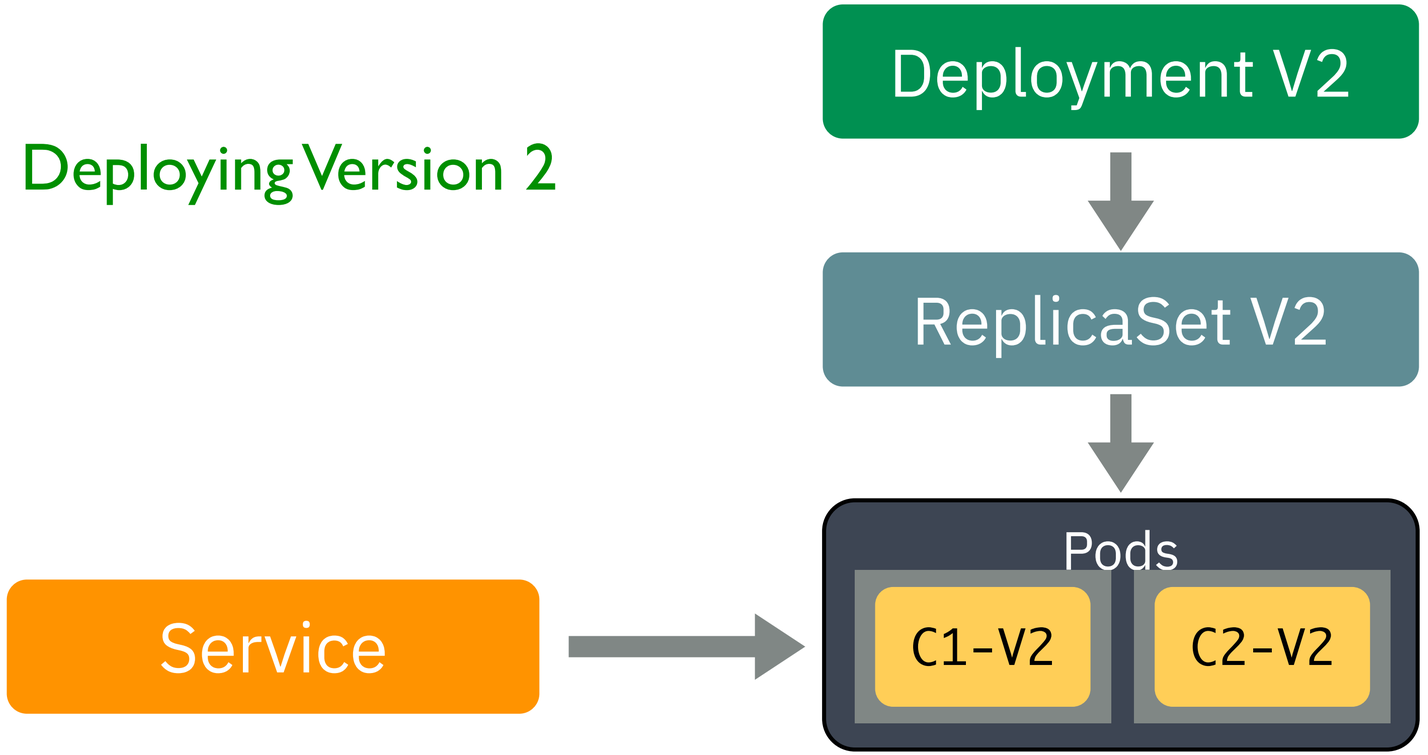
```
$ kubectl set image deployment/<application-name> <container-name>=<container-image-name>:<new-version>
```

Kubernetes Rolling Updates (Zero Downtime Deployments)

```
$ kubectl set image deployment/<application-name> <container-name>=<container-image-name>:<new-version>
```

Kubernetes Rolling Updates (Zero Downtime Deployments)

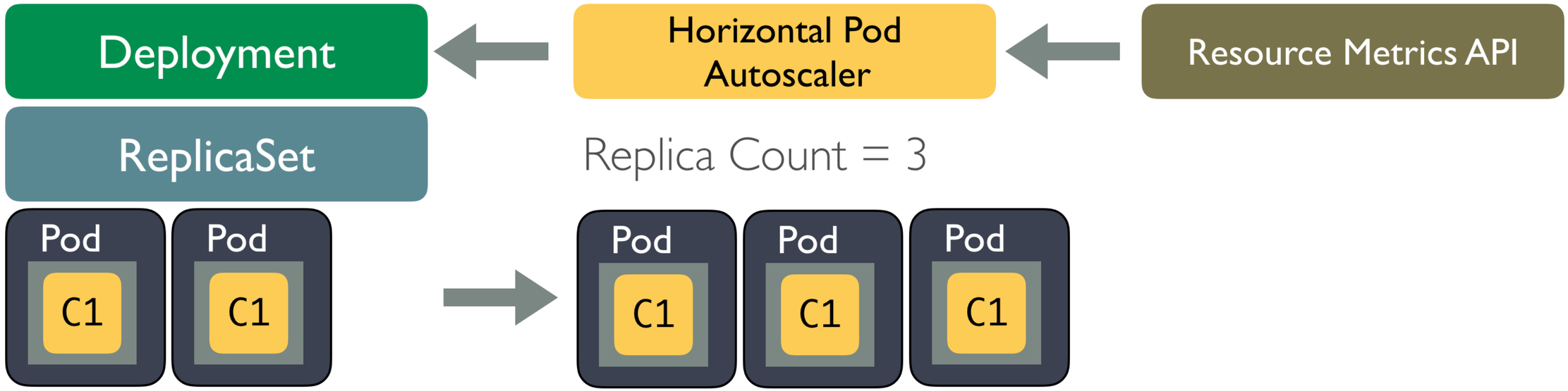
```
$ kubectl set image deployment/<application-name> <container-name>=<container-image-name>:<new-version>
```



Kubernetes Autoscaling

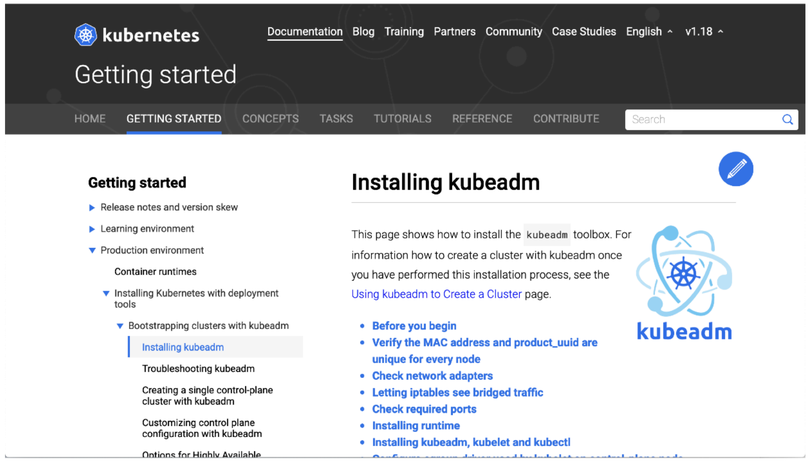
Kubernetes allows pods to be manually scaled either using ReplicaSets or Deployments. This can be achieved using the following CLI command:

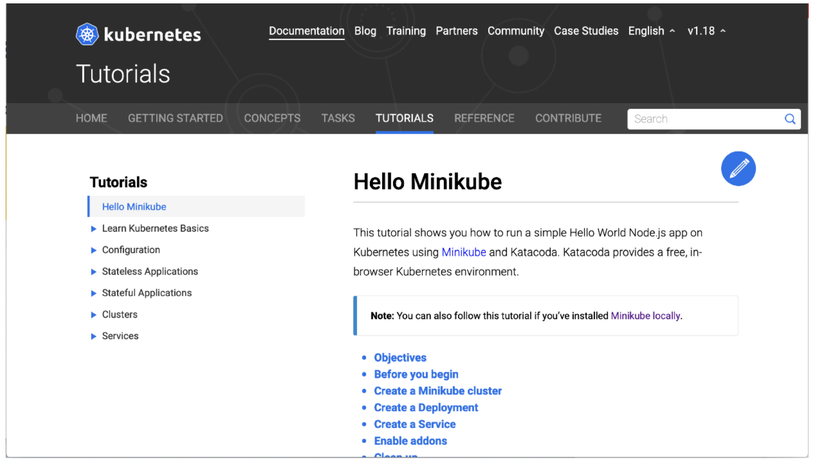
```
$ kubectl scale --replicas=<desired-instance-count> deployment/<application-name>
```

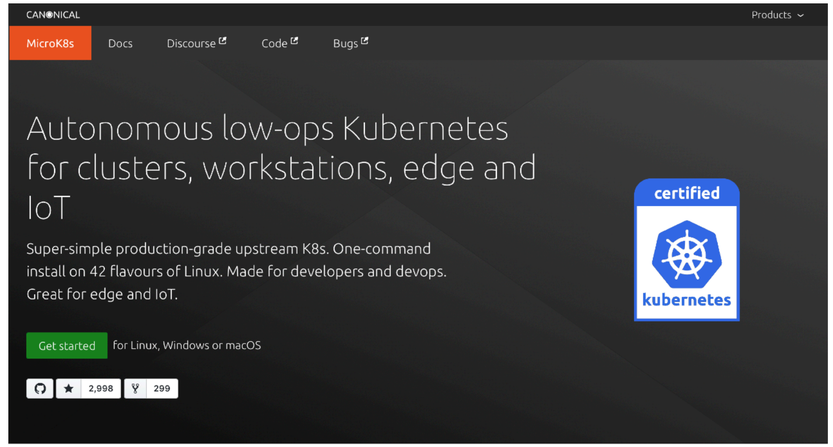


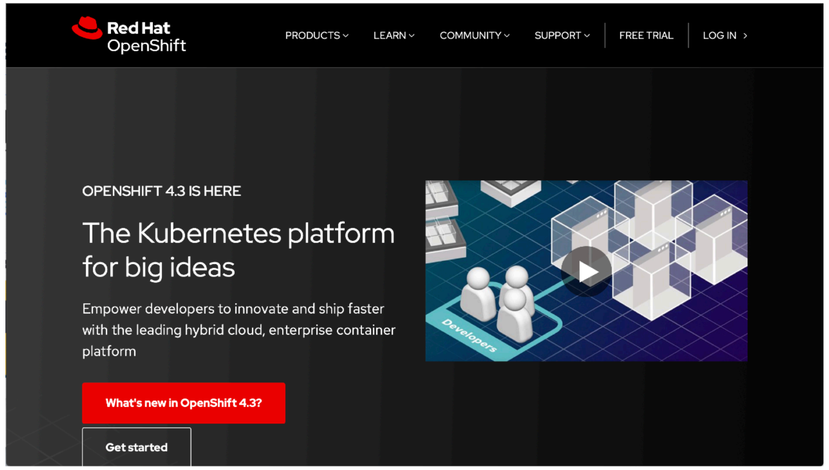
Kubernetes Distributions

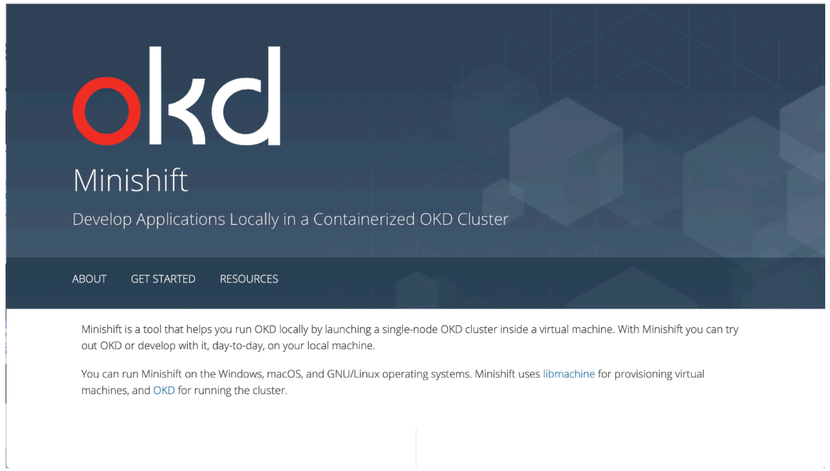
kubernetes, minikube, openshift, minishift, oco, minikube, microk8s, k3s, k3d

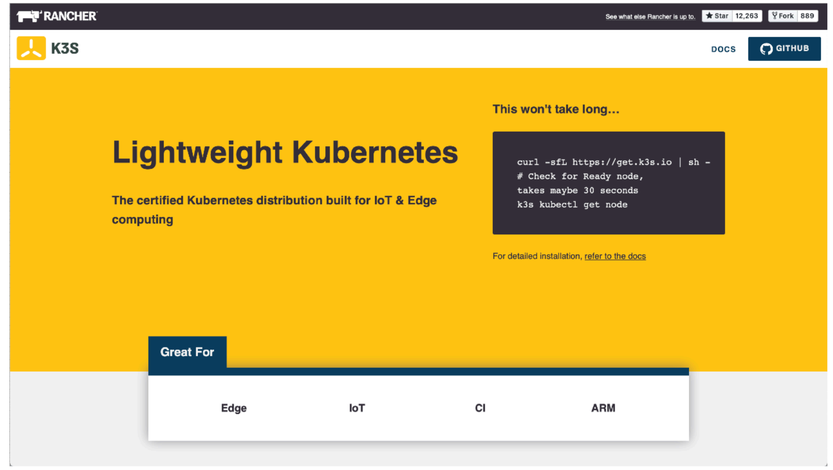












Hands-On

Confirm that you cluster Works

- Make sure that your cluster is configured correctly

```
$ kubectl get all


NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   3m44s
```

Deploy Nginx

- Create a Deployment

```
$ kubectl create deployment my-nginx --image=nginx:alpine
deployment.apps/my-nginx created
```

- Check that the deployment and pods were created

```
$ kubectl get deployments
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
my-nginx   1/1     1            1           3h42m

$ kubectl get pods
NAME                        READY   STATUS    RESTARTS   AGE
my-nginx-b49cfc867-q7mjn    1/1     Running   0          3h42m
```

Deploy Nginx

- Create a Service

```
$ kubectl expose deploy my-nginx --type=NodePort --port=80
service/my-nginx exposed
```

- Check that the service was created

```
$ kubectl get service
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP   172.21.0.1      <none>        443/TCP        317d
my-nginx     NodePort    172.21.12.226   <none>        80:30371/TCP   3h42m
```

Kubectl get all

```
vagrant@kubernetes:~$ kubectl get all

NAME                             READY   STATUS    RESTARTS   AGE
pod/my-nginx-64f47865f-2z5w8     1/1     Running   0          2m47s

NAME                 TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
service/kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP        15m
service/my-nginx     NodePort    10.104.182.40   <none>        80:30965/TCP   65s

NAME                       READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/my-nginx   1/1     1            1           2m47s

NAME                                  DESIRED   CURRENT   READY   AGE
replicaset.apps/my-nginx-64f47865f    1         1         1       2m47s
```

Naming Cnvention

- We made a deployment called `my-nginx` and it created:

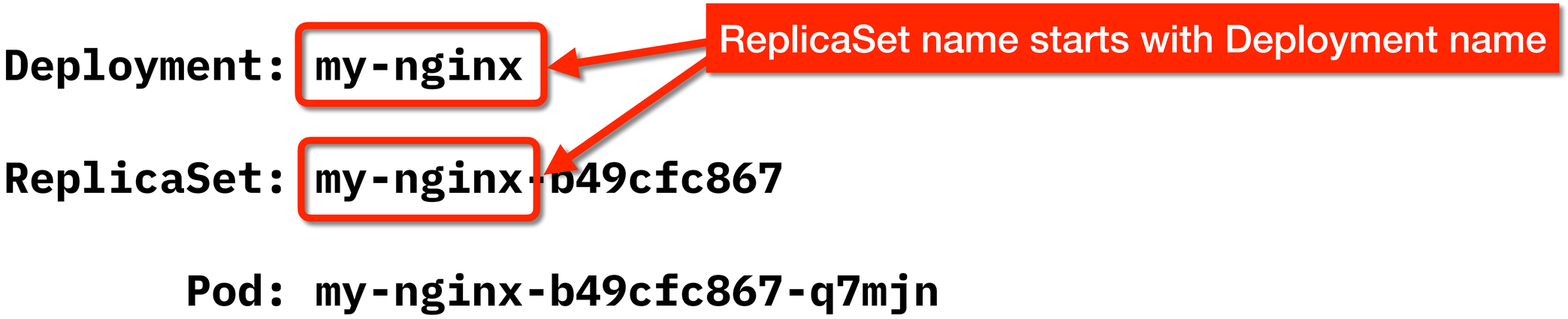
Deployment: my-nginx

ReplicaSet: my-nginx-b49cfc867

Pod: my-nginx-b49cfc867-q7mjn

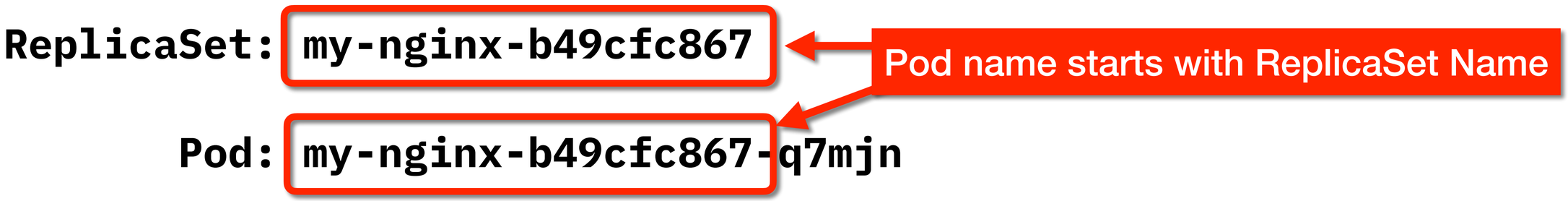
Naming Cnvention

- We made a deployment called `my-nginx` and it created:



Naming Cnvention

- We made a deployment called `my-nginx` and it created:



Get the NodePort

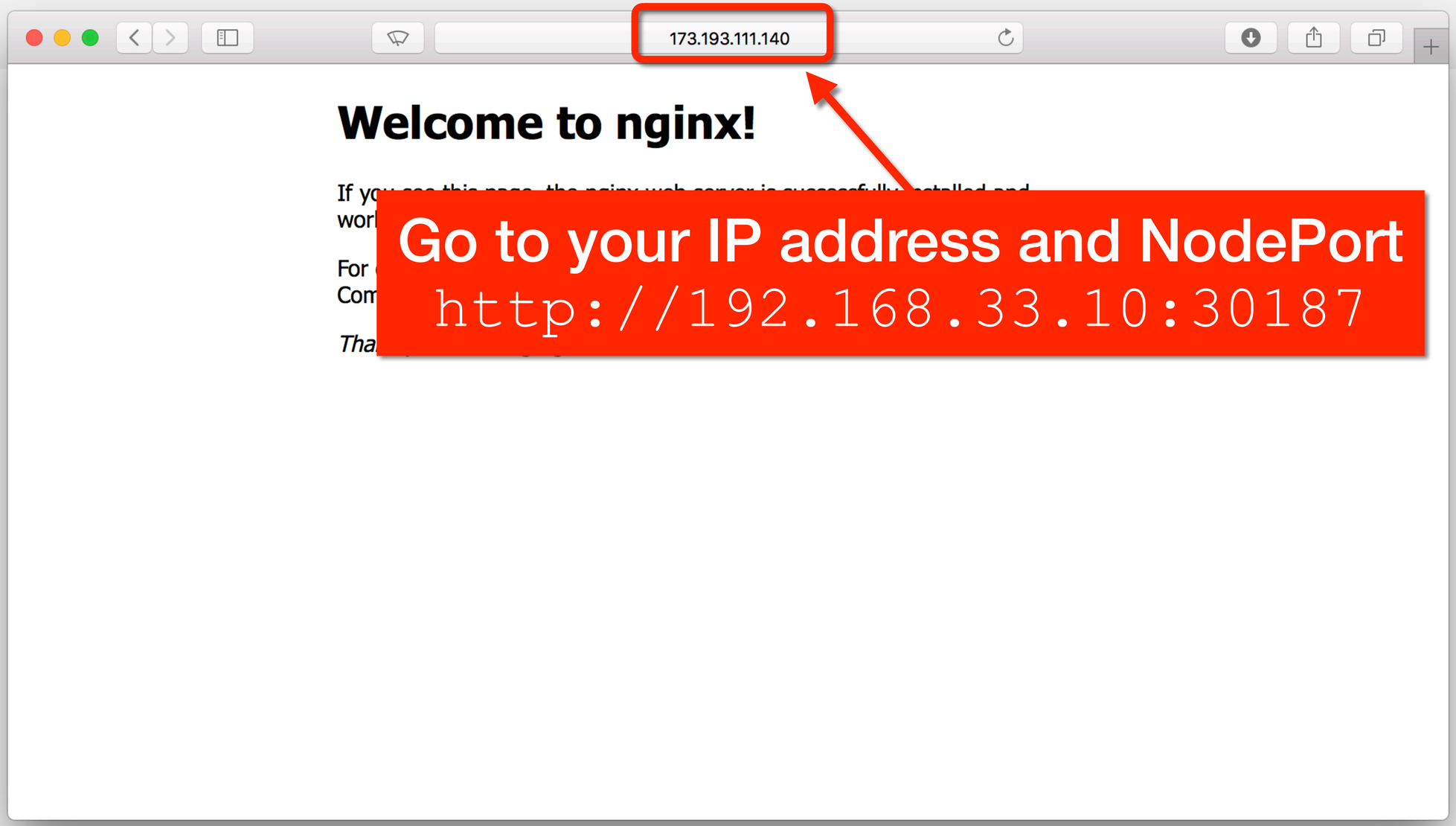
```
$ kubectl get service my-nginx
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
my-nginx      NodePort    172.21.9.54     <none>        80:30187/TCP   37d
```

Get the Public IP address

The IP address of the Vagrant Virtual Machine is in the Vagrantfile

192.168.33.10

Access NodePort



Destroy Nginx

- Delete a Service

```
$ kubectl delete svc my-nginx
service "my-nginx" deleted
```

- Delete the deployment

```
$ kubectl delete deploy my-nginx
deployment.apps "my-nginx" deleted
```

Let's Deploy our Project to Kubernetes

Build our Project as a Docker Image

- First let's build our project as a Docker image

```
$ cd /vagrant
$ docker build -t hitcounter:1.0 .
```

Push our Image to the Microk8s registry

- First let's tag the Docker image we just built

```
$ docker tag hitcounter:1.0 localhost:32000/hitcounter:1.0
```

- Then we push it to the remote registry

```
$ docker push localhost:32000/hitcounter:1.0
```

Create A Redis Service*

- The demo application requires Redis to store it's state
- We can deploy a Redis container from the `redis.yaml` file in the `./deploy` folder of the current repository

```
$ kubectl apply -f deploy/redis.yaml
deployment.apps/redis created
service/redis created
```

* Note: in a "real" deployment we would attach a volume to the Redis container to persist it's data beyond the life of the container

Redis.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis
  labels:
    app: redis
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      containers:
        - name: redis
          image: "redis:alpine"
          ports:
            - containerPort: 6379
              protocol: TCP
```

Deployment

Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: redis
  labels:
    app: redis
spec:
  ports:
  - port: 6379
    targetPort: 6379
  selector:
    app: redis
```

View the Logs from Redis

- You can view the logs of the pod to see how the service is doing

```
$ kubectl logs redis-784d69fbfd-qp5p5

1:C 01 Dec 2019 20:54:39.303 # oOOOoOOOoOOOo Redis is starting oOOOoOOOoOOOo
1:C 01 Dec 2019 20:54:39.303 # Redis version=5.0.6, bits=64, commit=00000000, modified=0, pid=1, just started
1:C 01 Dec 2019 20:54:39.303 # Warning: no config file specified, using the default config. In order to specify a
config file use redis-server /path/to/redis.conf
1:M 01 Dec 2019 20:54:39.305 * Running mode=standalone, port=6379.
1:M 01 Dec 2019 20:54:39.305 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/
somaxconn is set to the lower value of 128.
1:M 01 Dec 2019 20:54:39.305 # Server initialized
1:M 01 Dec 2019 20:54:39.305 # WARNING you have Transparent Huge Pages (THP) support enabled in your kernel. This will
create latency and memory usage issues with Redis. To fix this issue run the command 'echo never > /sys/kernel/mm/
transparent_hugepage/enabled' as root, and add it to your /etc/rc.local in order to retain the setting after a reboot.
Redis must be restarted after THP is disabled.
1:M 01 Dec 2019 20:54:39.305 * Ready to accept connections
```

Create your Credentials Secret

- Let's use the command line method of creating a secret for DATABASE_URI

```
$ export DATABASE_URI="redis://redis:6379"

$ kubectl create secret generic redis-creds \
  --from-literal=uri=$DATABASE_URI

secret/redis-creds created
```

Check your Credentials Secret

- Let's see what was created for

There is your base64 encoded secret

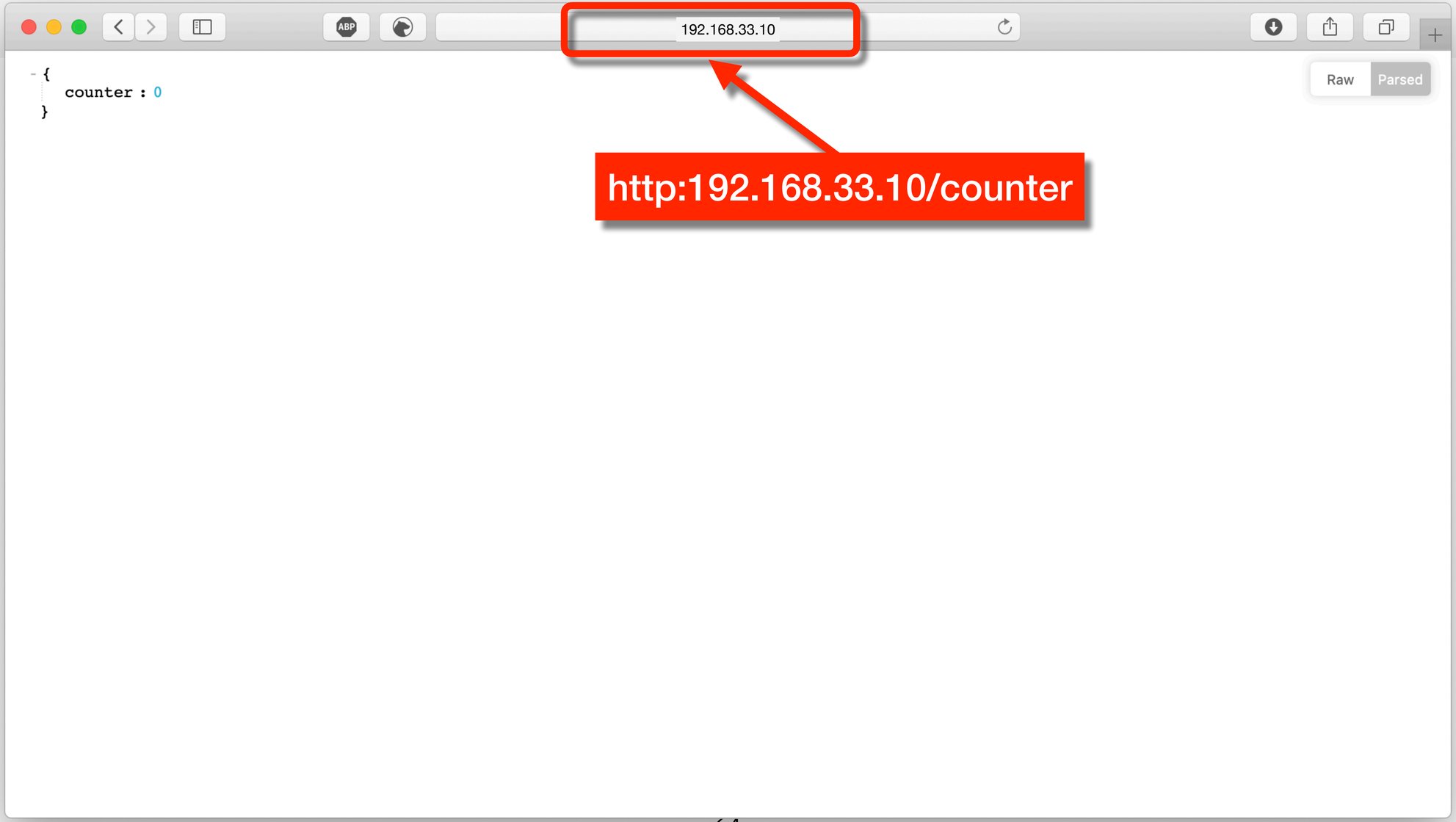
```
$ kubectl get secret redis-creds -o yaml
apiVersion: v1
data:
  uri: cmVkaXM6Ly86QHJlZGlzOjYzNzkvMA==
kind: Secret
metadata:
  creationTimestamp: "2019-11-20T03:53:22Z"
  name: redis-creds
  namespace: default
  resourceVersion: "3437"
  selfLink: /api/v1/namespaces/default/secrets/redis-creds
  uid: aee28dcf-f1f4-40ba-83fc-8b775c208264
type: Opaque
```

Now we can Deploy Our Image

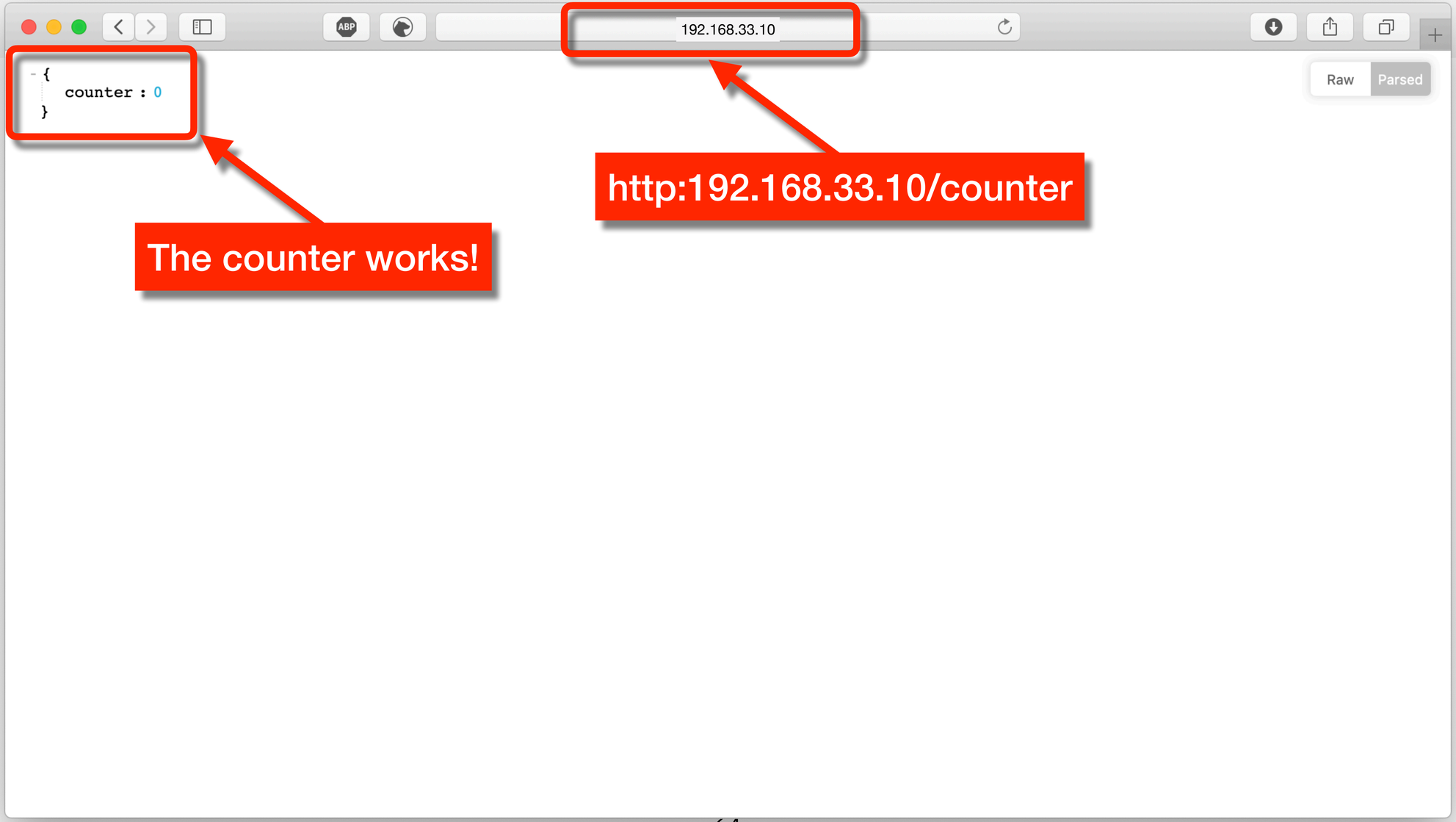
- You can use `kubectl apply` with the files under `/vagrant/deploy`

```
$ cd /vagrant/deploy

$ kubectl apply -f deployment.yaml
deployment.apps/hitcounter created

$ kubectl apply -f service.yaml
service/hitcounter-service created

$ kubectl apply -f ingress.yaml
ingress.extensions/hitcounter-ingress created
```

Your App on Kubernetes



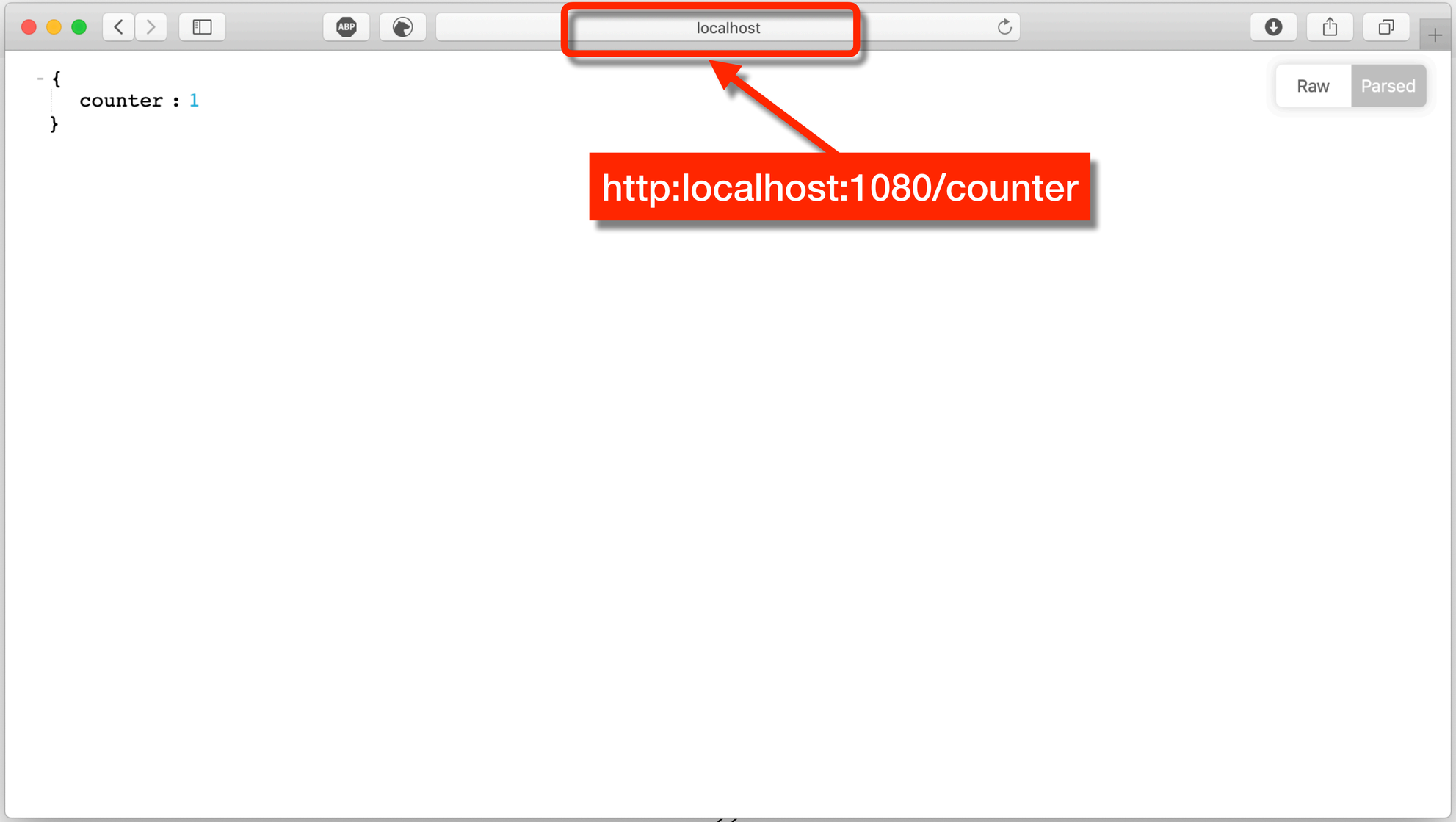
Your App on Kubernetes



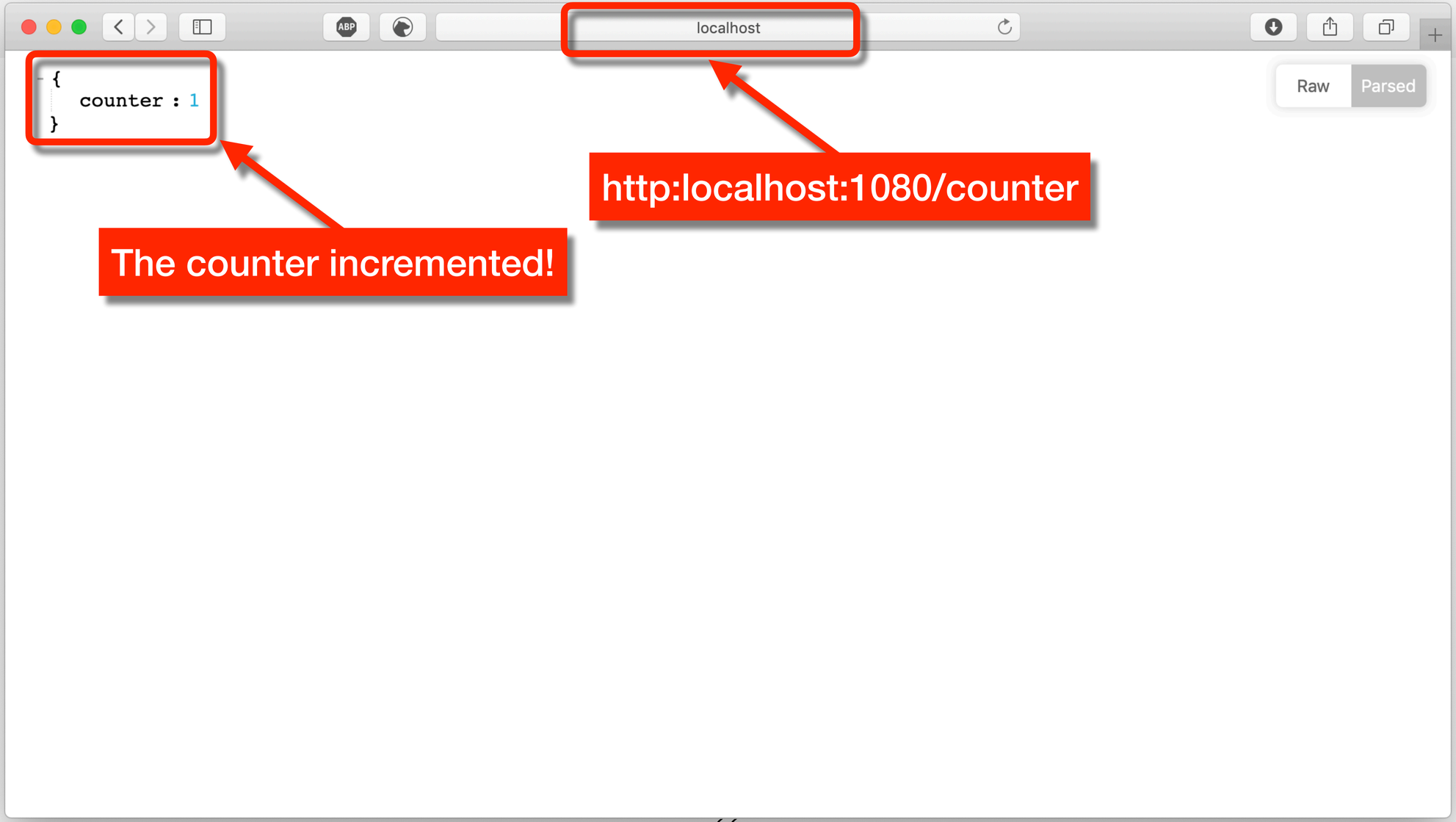
Use HTTP command to increment the counter

```
$ http post localhost/counters/foo
HTTP/1.1 201 CREATED
Connection: keep-alive
Content-Length: 14
Content-Type: application/json
Date: Wed, 20 Nov 2019 04:14:51 GMT
Server: openresty/1.15.8.2

{
    "counter": 1
    "Name": "foo"
}
```

Check again


localhost
Raw
Parsed
{
counter : 1
}
http:localhost:1080/counter

Check again


localhost
{
counter : 1
}
Raw
Parsed
http:localhost:1080/counter
The counter incremented!

Summary

You just created your first Kubernetes deployment

You learned how to create deployments

You learned how to expose deployments as services

You learned how to scale deployments

You can now deploy your microservices on any Kubernetes cloud