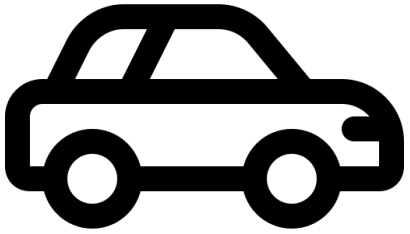
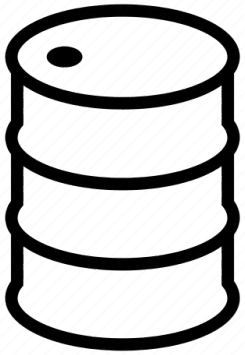
DevOps - Kubernetes

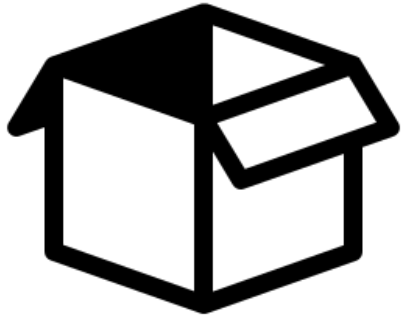
M. Ali Kahoot

What is a Container

What is a Container







What is a Container

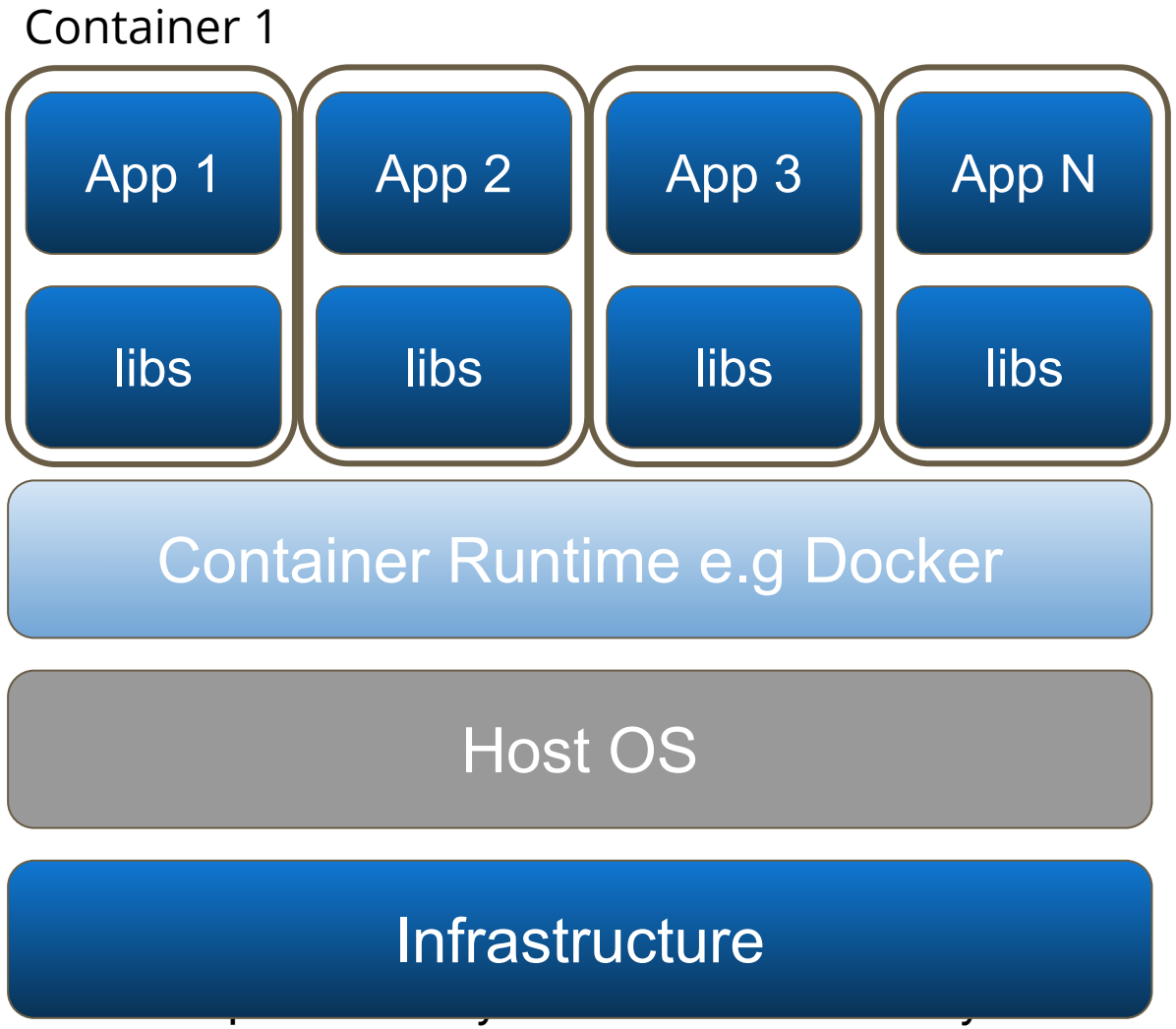
Backend App

Database

Frontend

Queues

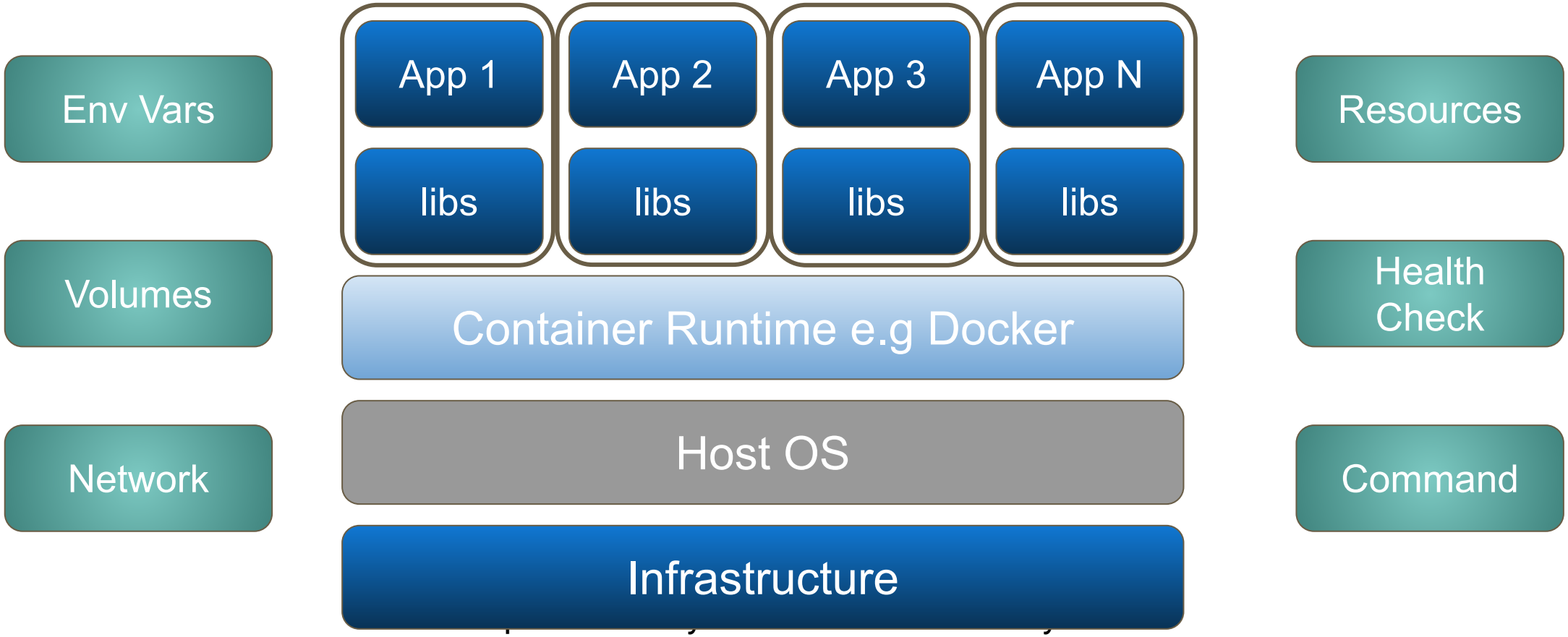
What is a Container



Container Resources

HEALTHCHECK CMD curl --fail http://localhost:5000/ || exit 1

docker run -it -e NAME="Ali Kahoot" --net dok -v DOK:/data/on/kubernetes --memory 1Gi busybox echo "The attendees are awesome"



ORCHESTRATION

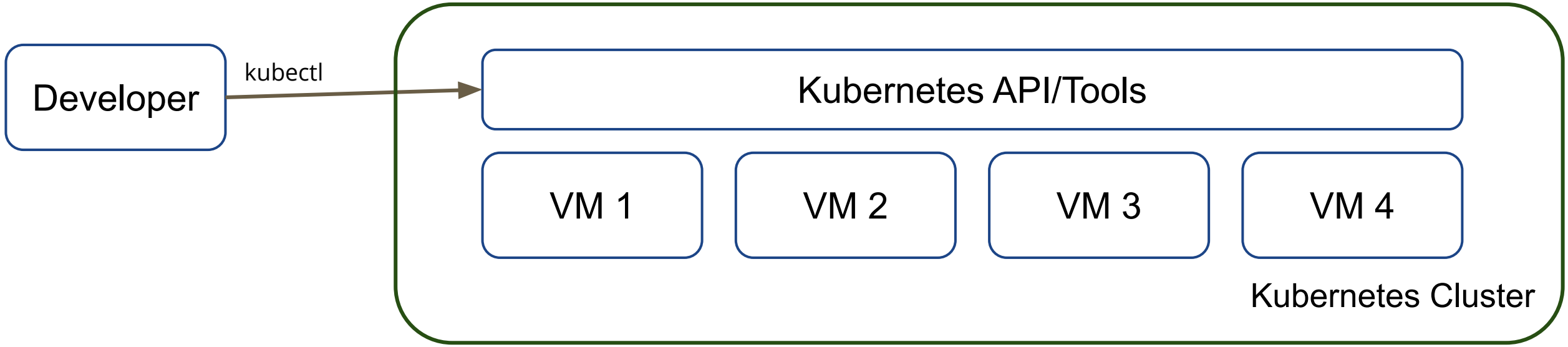
Orchestra: Every person knows what to do, but there is a single person guiding them what to do.

DevOps Course by Ali Kahoot - Dice Analytics

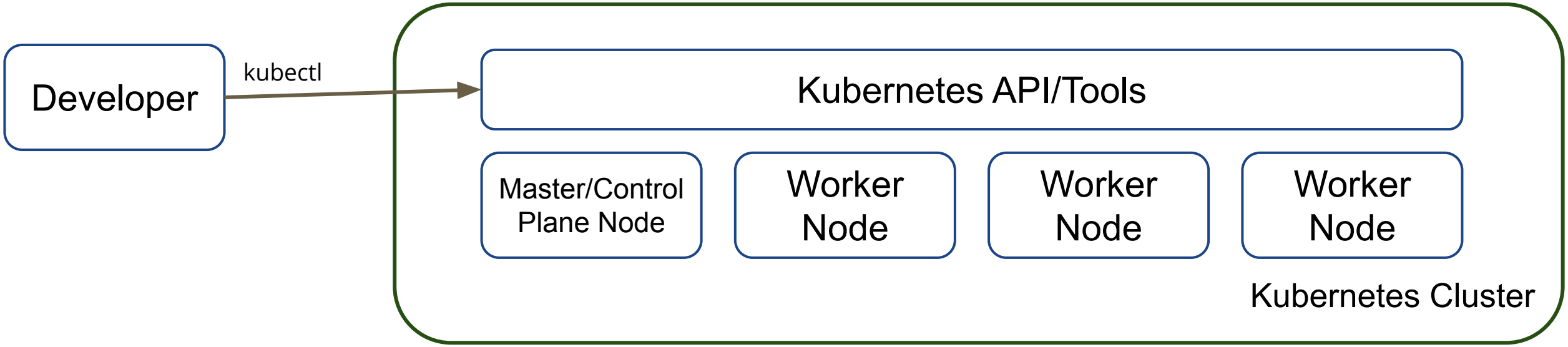
KUBERNETES & DOCKER

- K8s compliments Docker Containers, not an alternative
- Schedules Containers
- Alternate of Docker Swarm not containers

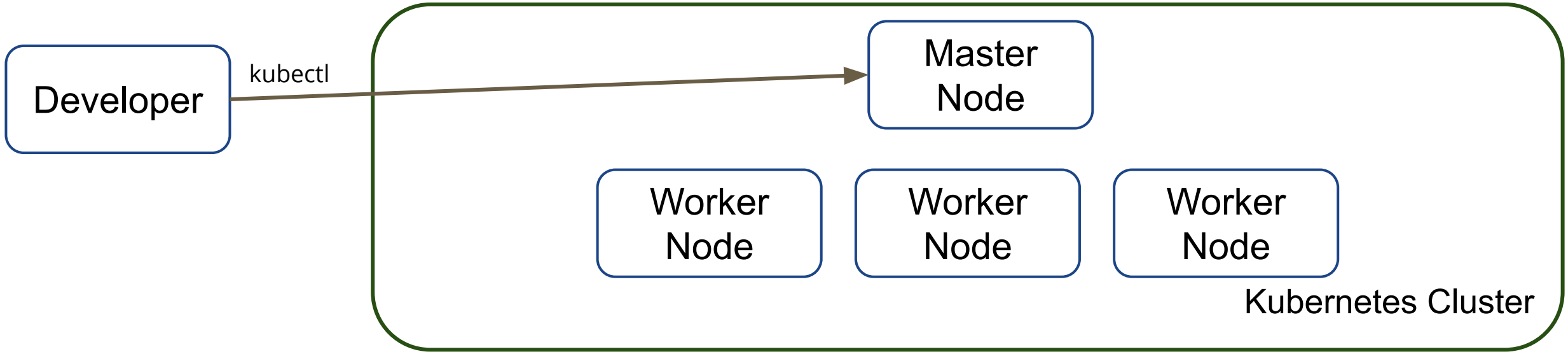
Kubernetes Architecture



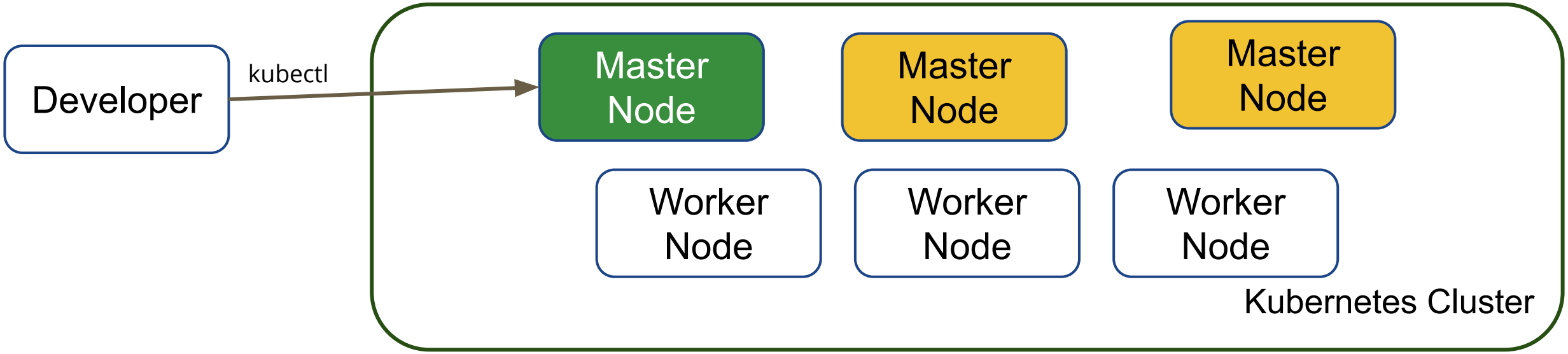
Kubernetes Architecture



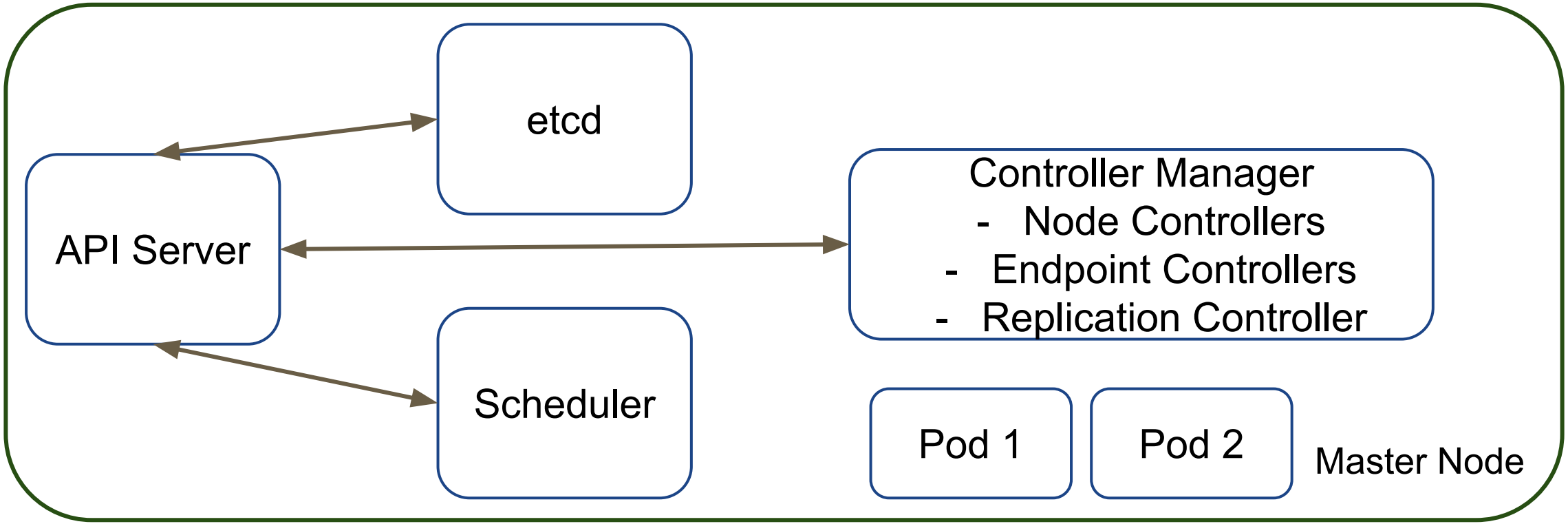
Kubernetes Architecture



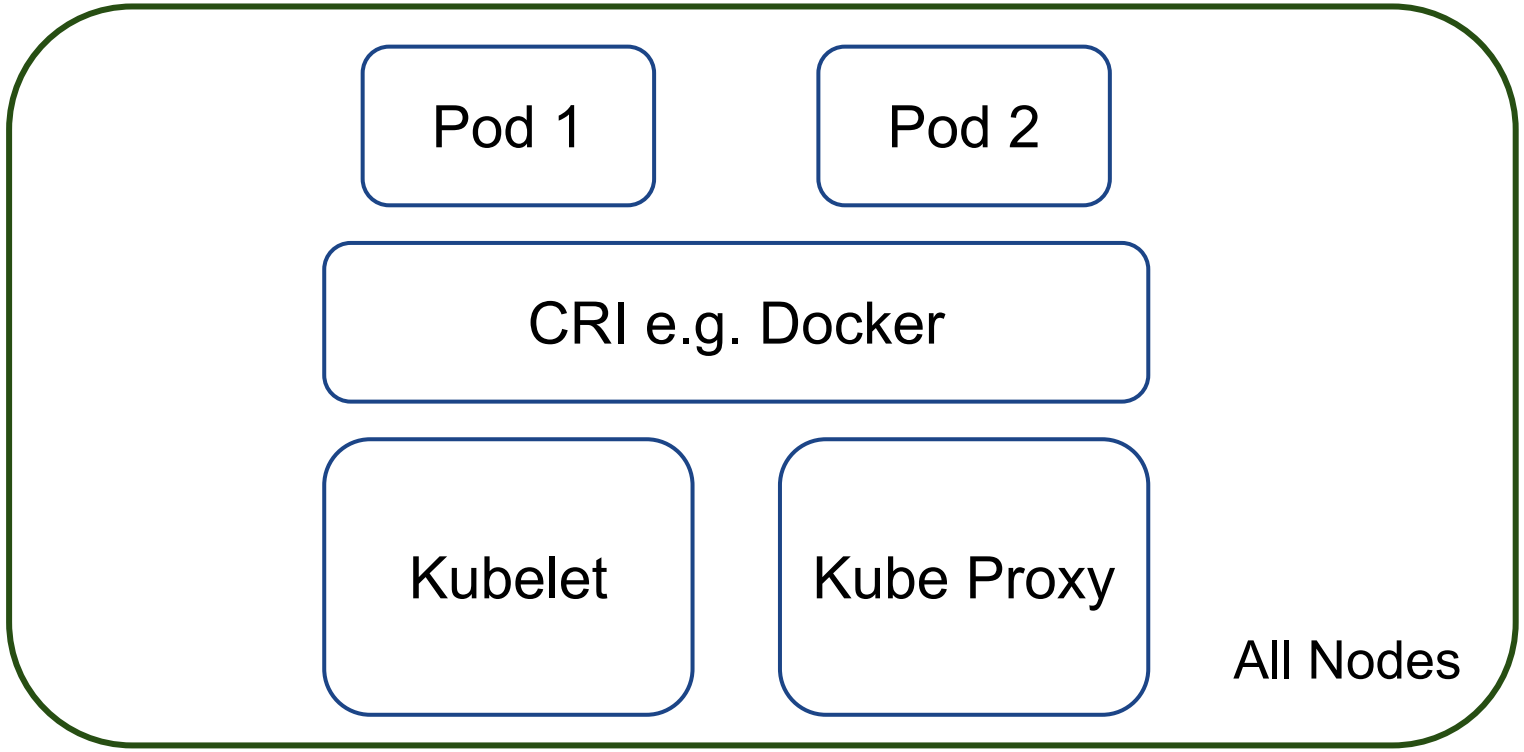
HA Kubernetes Architecture

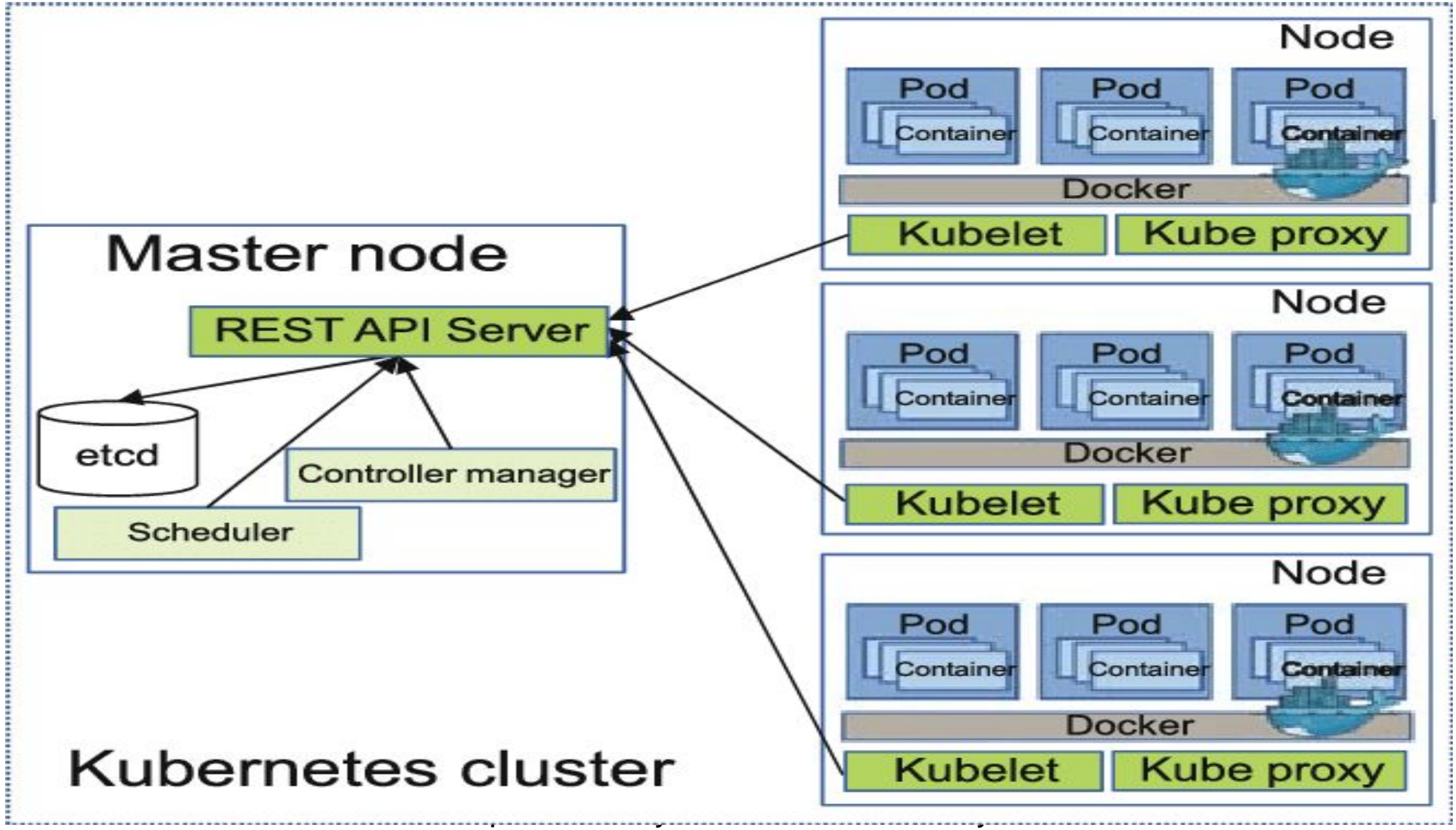


Kubernetes Architecture



Kubernetes Architecture

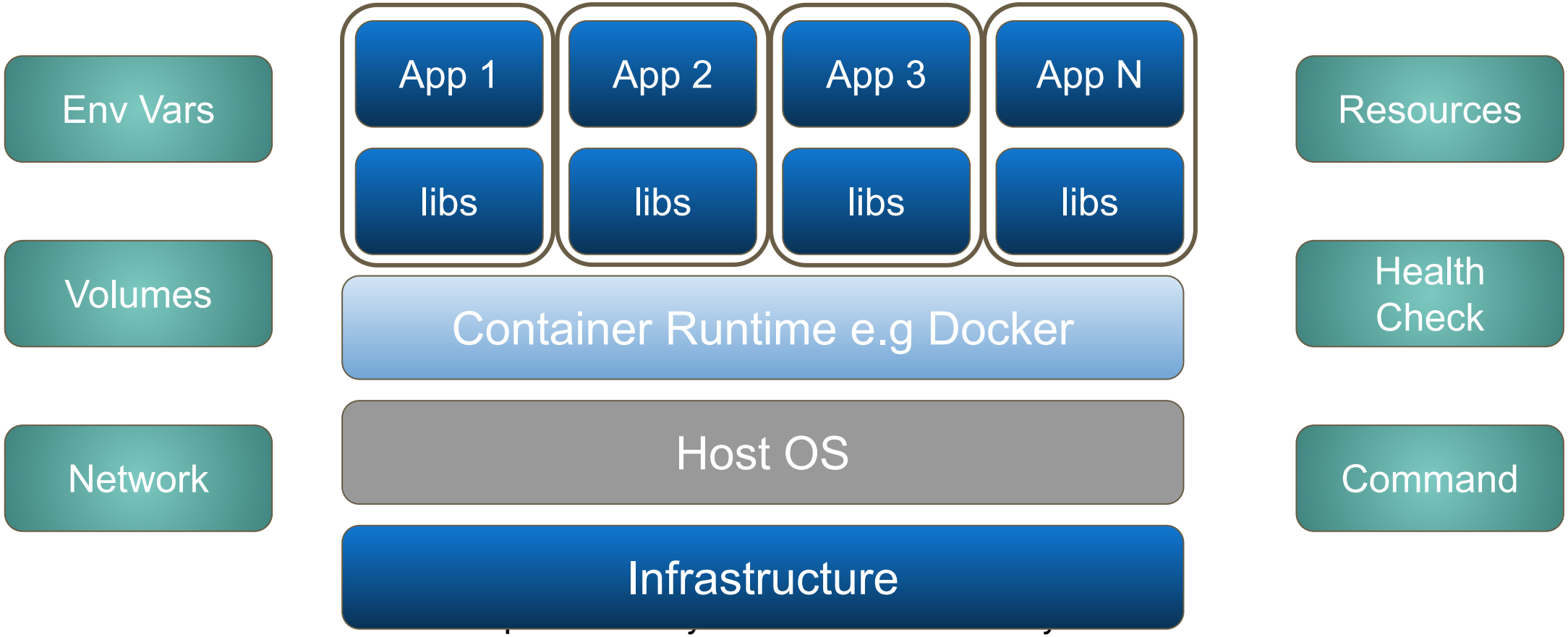



Node
Pod
Container
Pod
Container
Pod
Container
Docker
Kubelet
Kube proxy
Master node
REST API Server
etcd
Controller manager
Scheduler
Node
Pod
Container
Pod
Container
Pod
Container
Docker
Kubelet
Kube proxy
Node
Pod
Container
Pod
Container
Pod
Container
Docker
Kubelet
Kube proxy
Kubernetes cluster

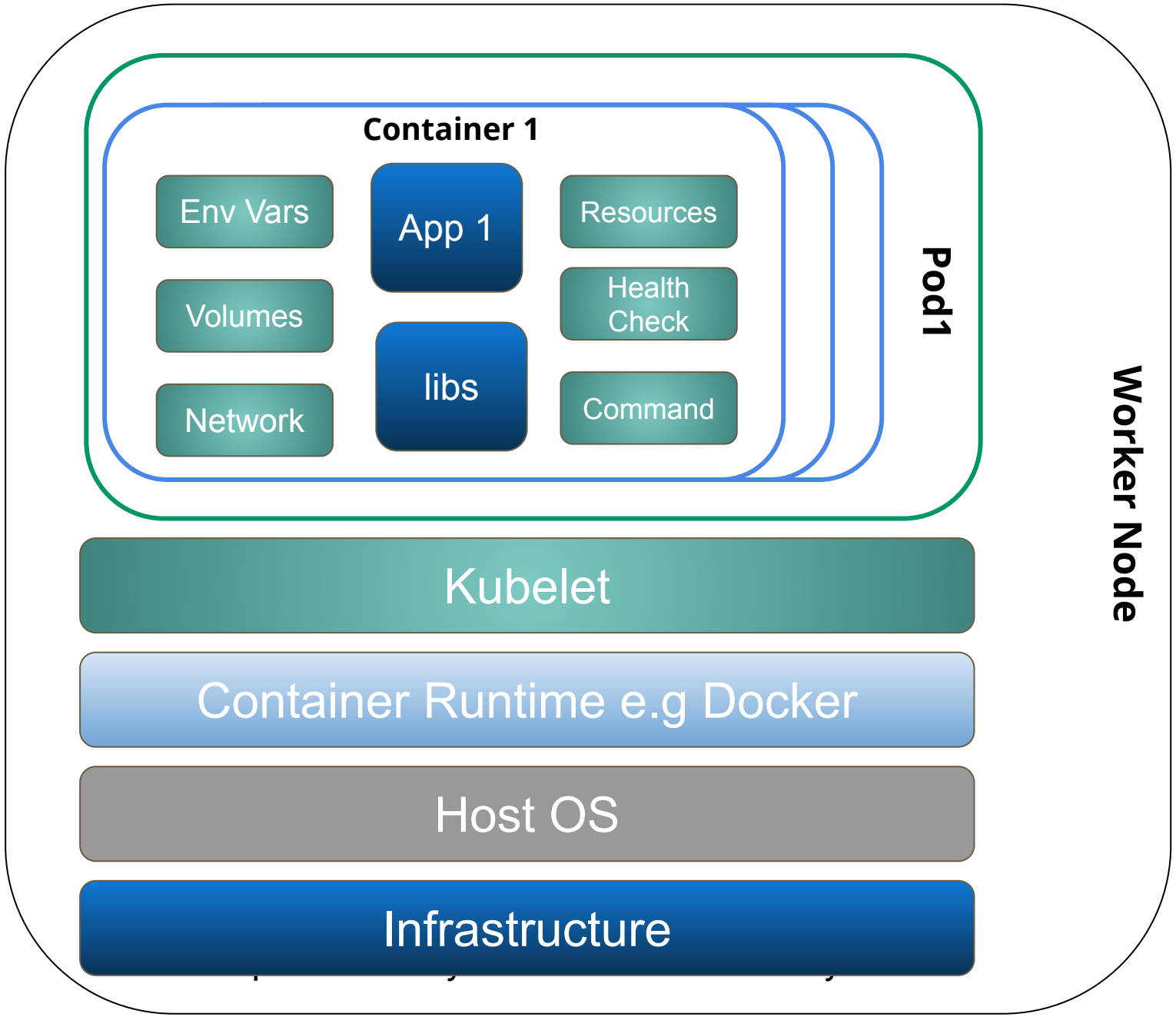
Container Resources

HEALTHCHECK CMD curl --fail http://localhost:5000/ || exit 1

docker run -it -e NAME="Ali Kahoot" --net dok -v DOK:/data/on/kubernetes --memory 1Gi busybox echo "The attendees are awesome"



Pod


Container 1
Env Vars
App 1
Resources
Volumes
Health Check
libs
Network
Command
Pod1
Kubelet
Container Runtime e.g Docker
Host OS
Infrastructure
Worker Node

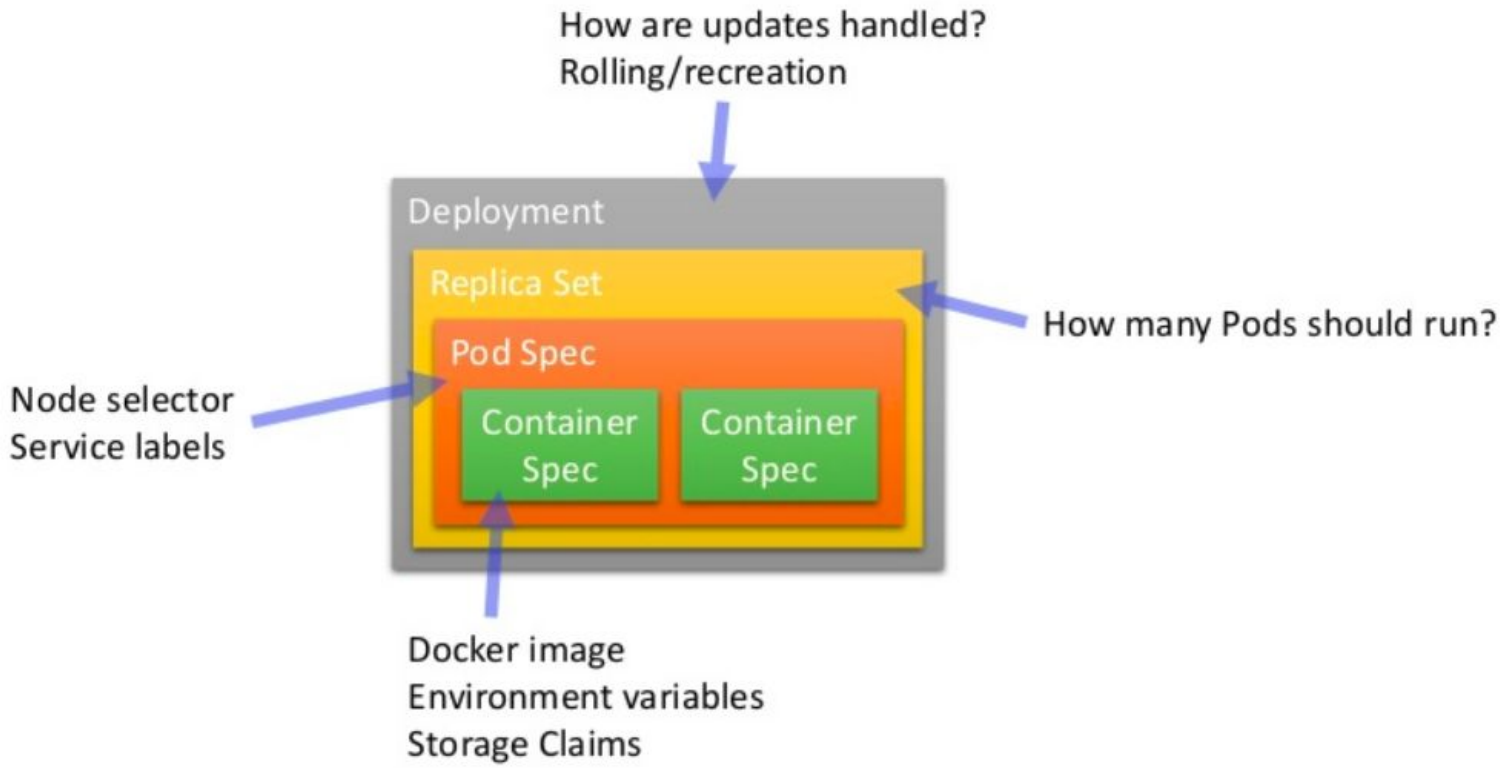
Running an App

Docker

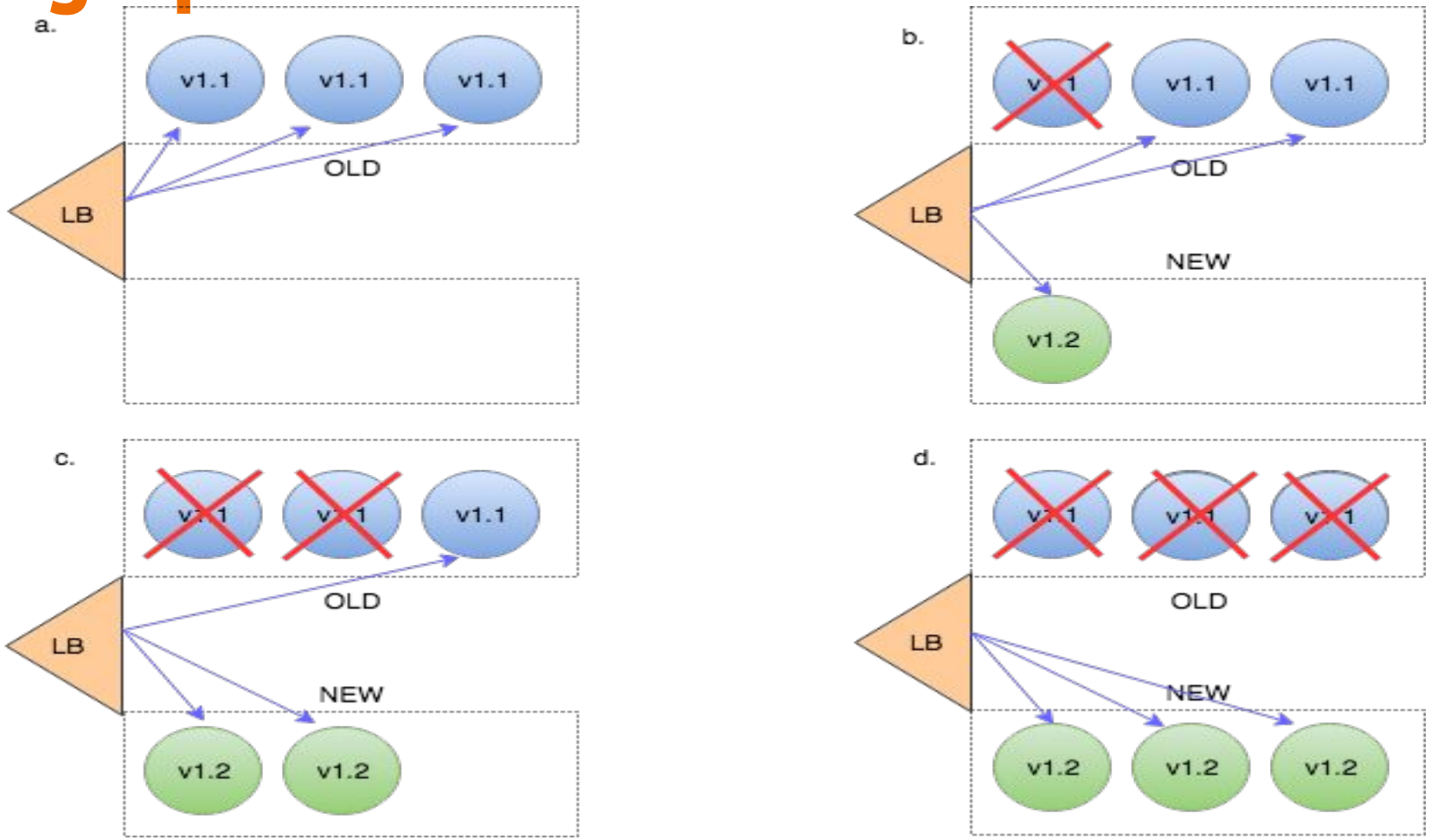
docker run -it --name dok-app -e NAME=”Ali Kahoot” --net dok -v DOK:/data/on/kubernetes --memory 1Gi busybox echo “The attendees are awesome”

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: dok-app
  labels:
    app: dok-app
spec:
  containers:
    - name: dok-app
      image: busybox
      args: ["echo","The attendees are awesome"]
      resources:
        limits:
          memory: 1Gi
      env:
        name: NAME
        value: Ali Kahoot
      volumeMounts:
        - name: dok
          mountPath: /data/on/kubernetes
      volumes:
      - name: dok
        persistentVolumeClaim:
          claimName: dok
```

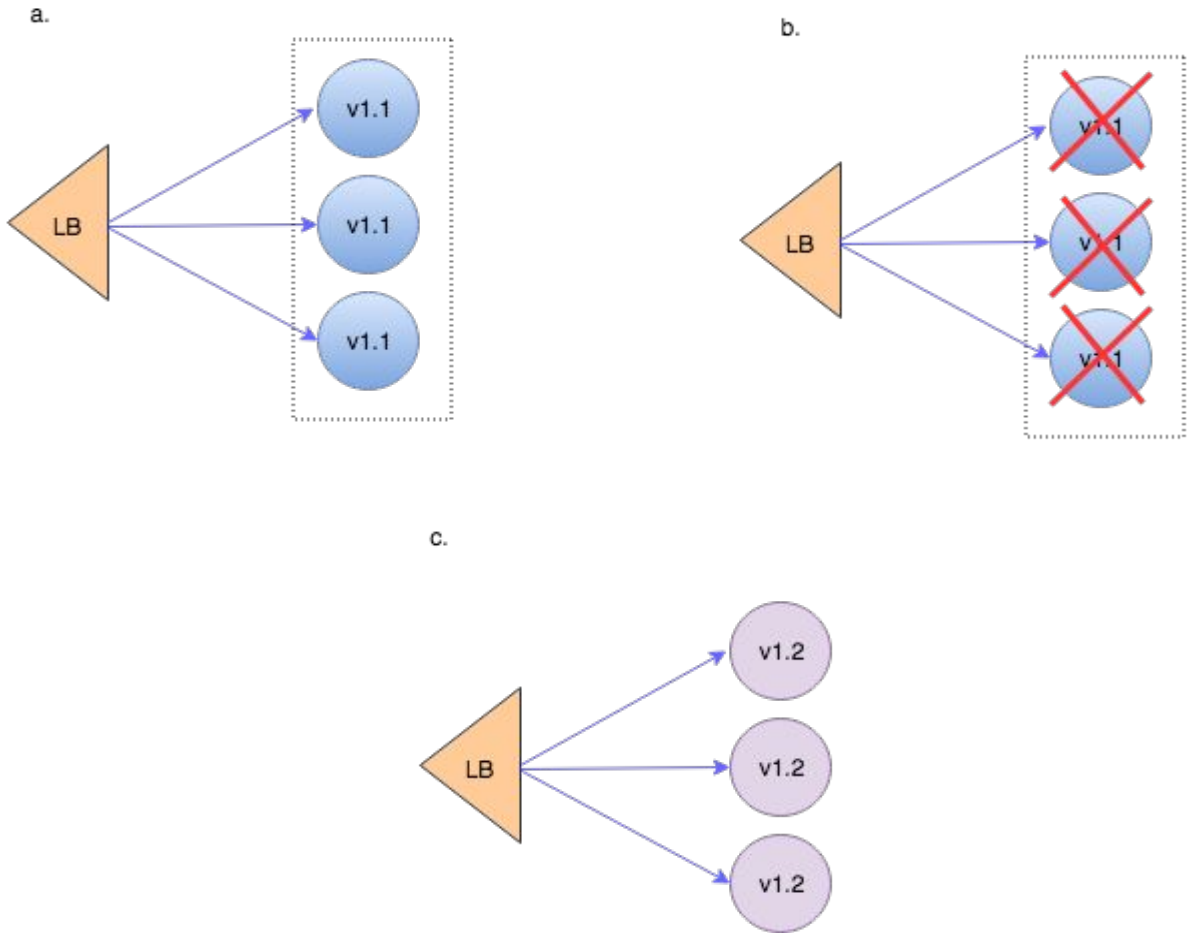
Deployment


How are updates handled?
Rolling/recreation
Deployment
Replica Set
Pod Spec
Container Spec
Container Spec
How many Pods should run?
Node selector
Service labels
Docker image
Environment variables
Storage Claims

Rolling Update



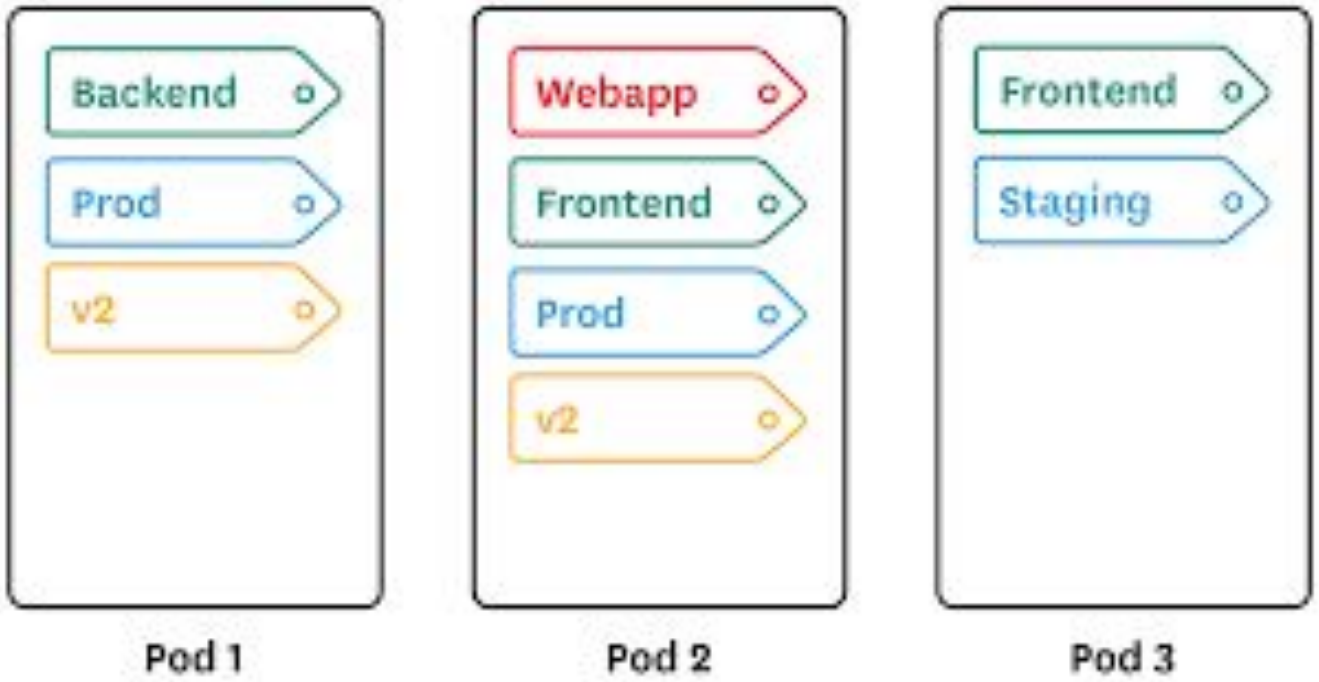
Recreate



LABELS

- a key: value pair for any of k8s resources.
- Useful for selecting objects based on labels out of 100s of resources
- Useful for querying dynamic objects e.g. pods get deleted & added so easy to query set of pods with same labels
- Map your own organizational structures onto system objects in a loosely coupled fashion

LABELS


Backend
Prod
v2
Webapp
Frontend
Prod
v2
Frontend
Staging
Pod 1
Pod 2
Pod 3

LABEL SELECTORS

- Unlike names and UIDs, labels do not provide uniqueness.
- We expect many objects to carry the same label(s)
- Via a label selector, the client/user can identify a set of objects. The label selector is the core grouping primitive in Kubernetes.
- In this way you can choose any resource i.e. NodeSelectors, Pods with specific labels, Services with Specific Labels, etc

SERVICES

- An abstract way to expose an application running on a set of Pods as a network service.
- Pods get deleted or recreated or scaled up or scaled down dynamically, so how can one expose them, and how to load balance between them? Similarly, how will a Frontend connect to Backend pod?

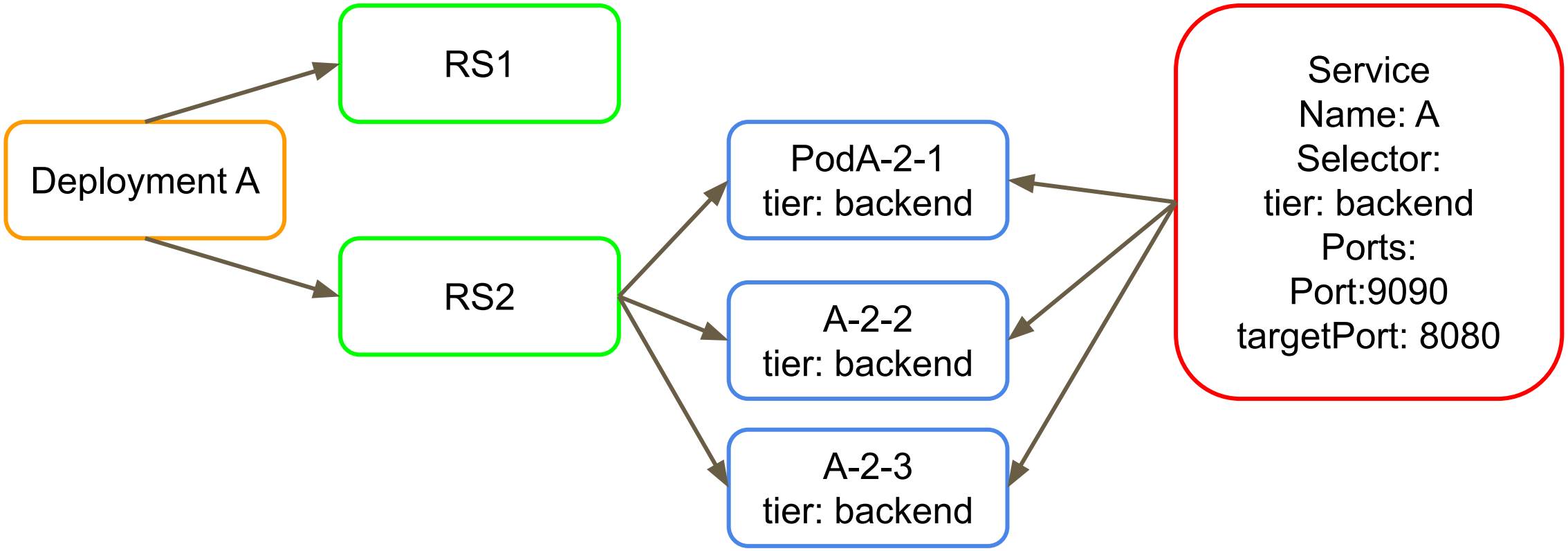
SERVICES

- Service is an abstraction which defines a logical set of Pods and a policy by which to access them
- Frontends do not care which of the backend pods they use, the frontend clients should not need to be aware of that, nor should they need to keep track of the set of backends themselves

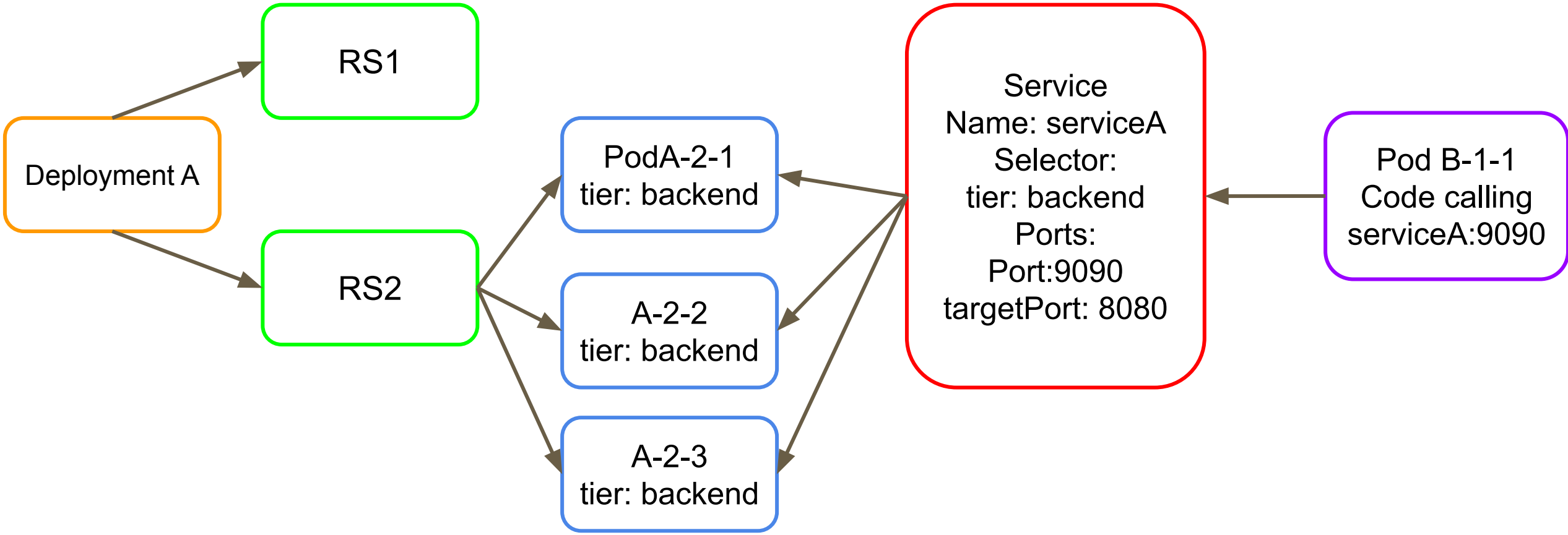
Service

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - protocol: TCP
    port: 8080
    targetPort: 9376
  - protocol: TCP
    port: 8081
    targetPort: 9377
```

Service


RS1
Deployment A
RS2
PodA-2-1
tier: backend
A-2-2
tier: backend
A-2-3
tier: backend
Service
Name: A
Selector:
tier: backend
Ports:
Port:9090
targetPort: 8080

Service

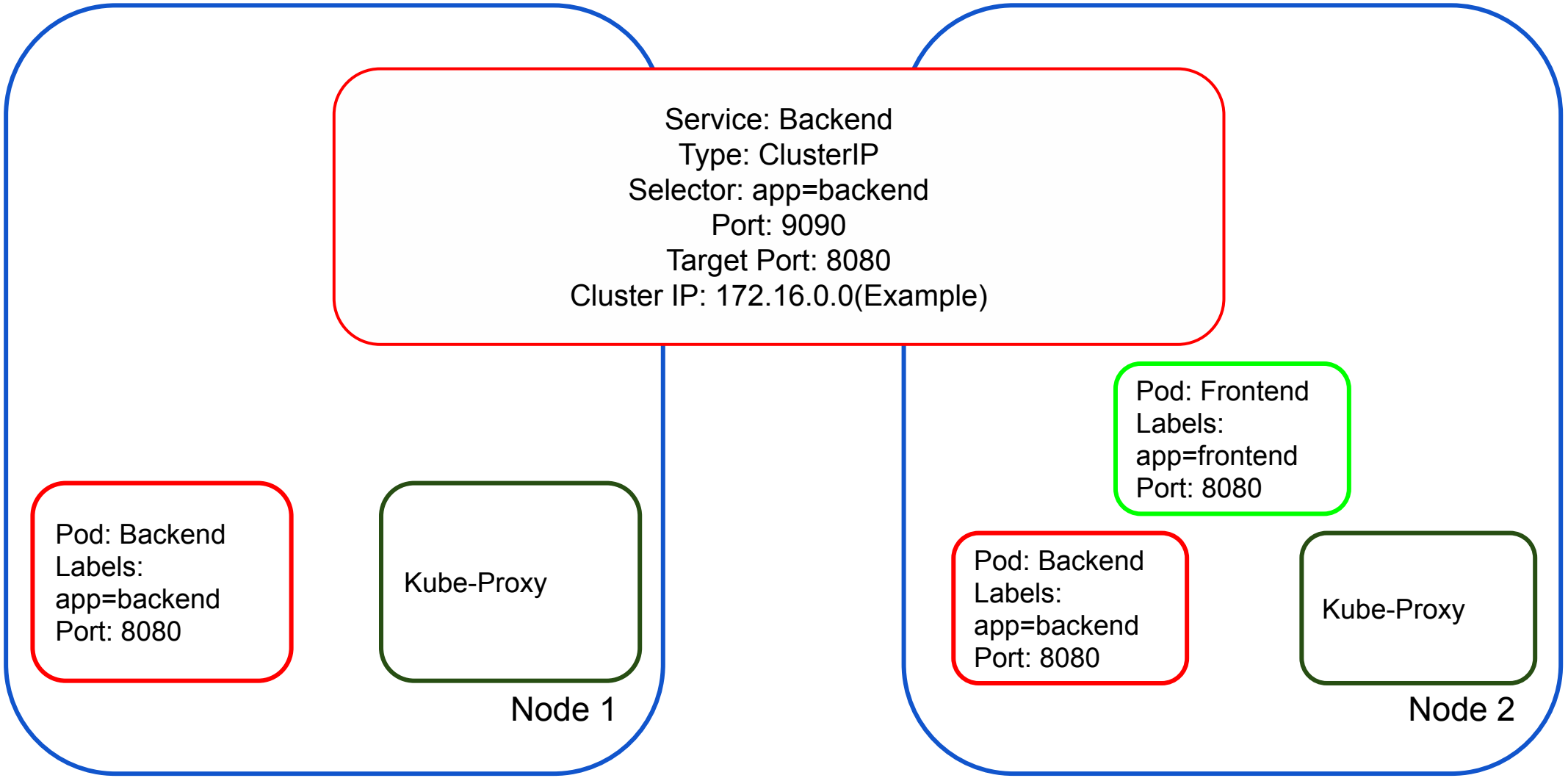

RS1
Deployment A
RS2
PodA-2-1
tier: backend
A-2-2
tier: backend
A-2-3
tier: backend
Service
Name: serviceA
Selector:
tier: backend
Ports:
Port:9090
targetPort: 8080
Pod B-1-1
Code calling
serviceA:9090

Service Types

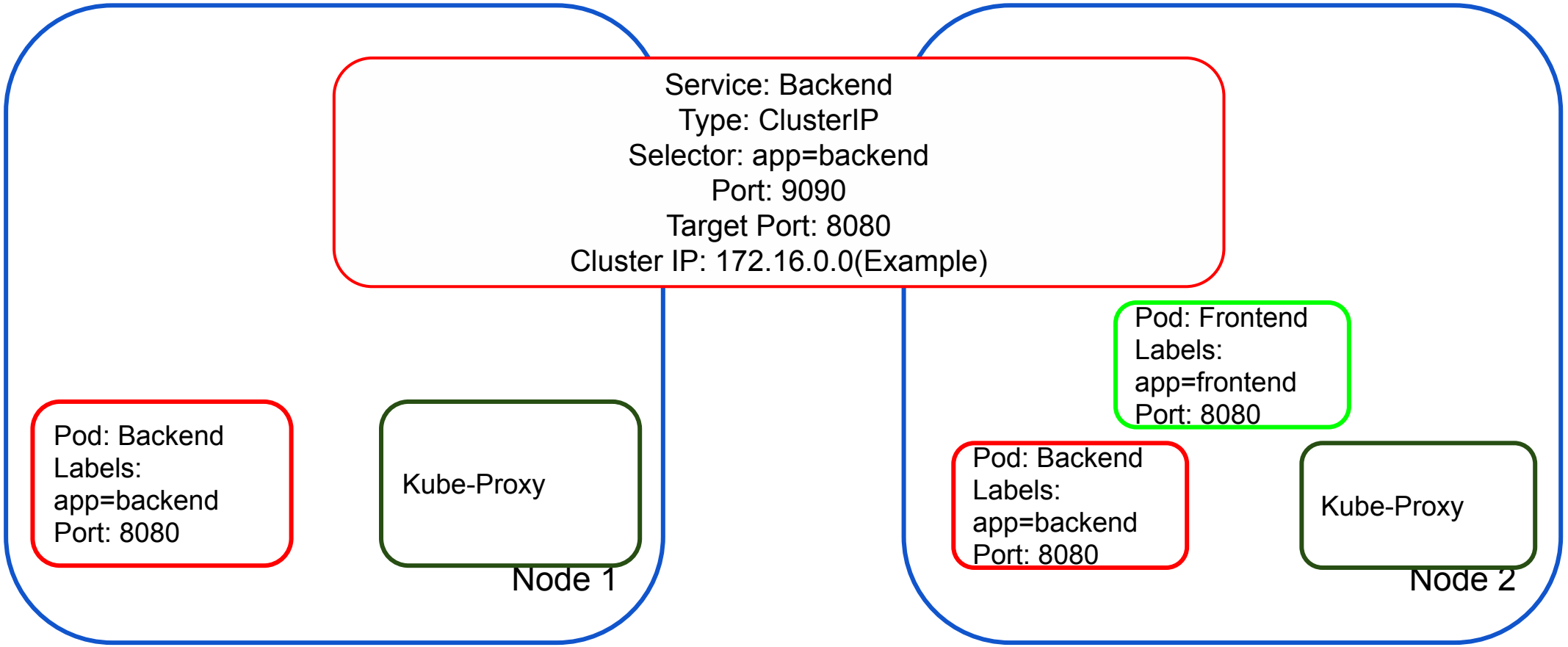
You might need to expose some of your service inside the cluster only(backends & databases) and some to the external world(frontends), so there are different type of Services for that

- ClusterIP: Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. This is the default ServiceType

- NodePort: Exposes the Service on each Node's IP at a static port (the NodePort)


Service: Backend
Type: ClusterIP
Selector: app=backend
Port: 9090
Target Port: 8080
Cluster IP: 172.16.0.0(Example)
Pod: Backend
Labels:
app=backend
Port: 8080
Kube-Proxy
Node 1
Pod: Frontend
Labels:
app=frontend
Port: 8080
Pod: Backend
Labels:
app=backend
Port: 8080
Kube-Proxy
Node 2

How can Backend Pod be accessed from any other Pod?



LABS

All the labs for this slide are available at

https://github.com/kahootali/k8s-labs/tree/main/module-2

Curl Backend Pod

```
Deploy Backend
kubectl apply -f backend.yaml
```

Now, create a new pod, with curl installed on it.

There is a file named busybox-with-curl.yaml.

Deploy that by running

```
kubectl apply -f busybox-with-curl.yaml
```

Now exec into the pod, either by dashboard or by running

```
kubectl exec -it busybox-with-curl sh
```

Curl Backend Pod

And curl the backend pod through 3 techniques.

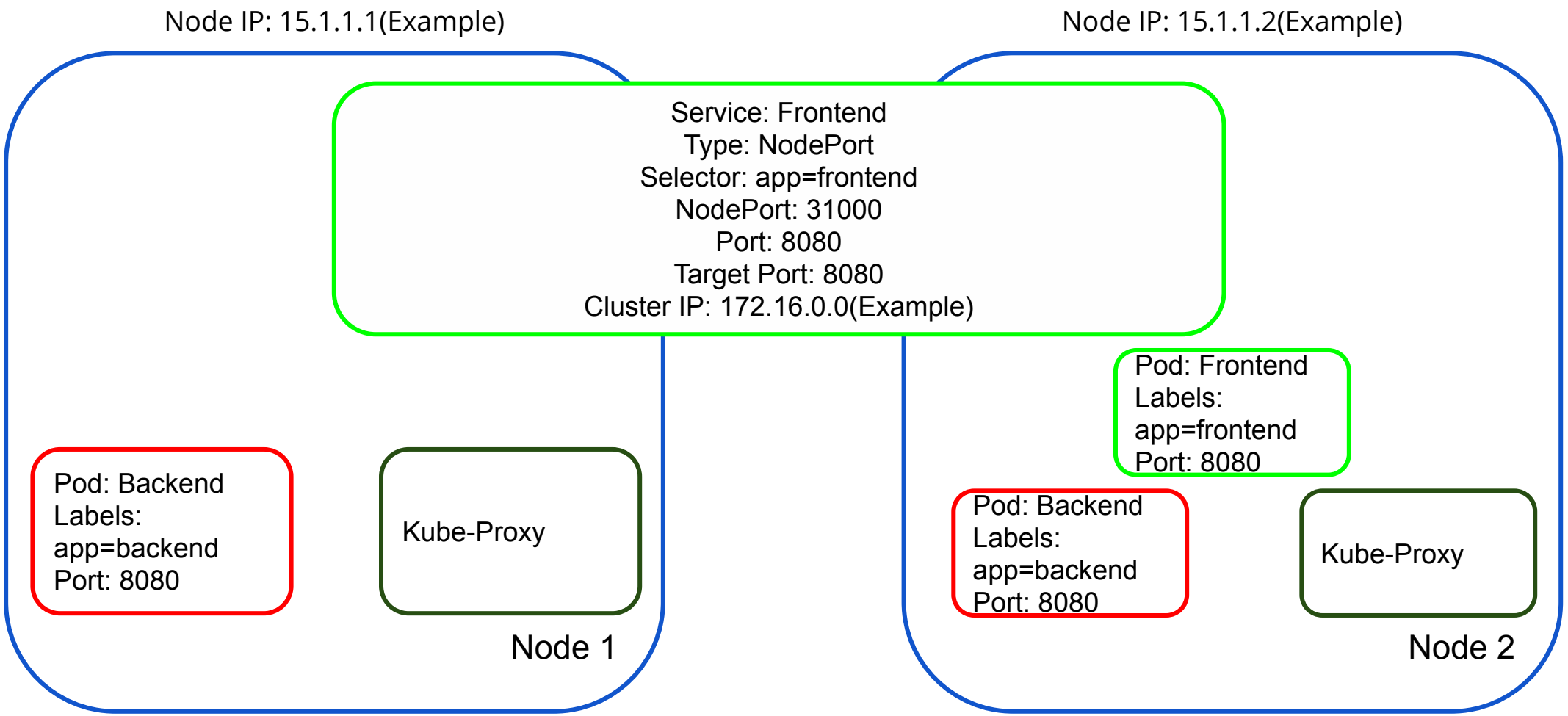
```
curl backend:9090/students
curl <backend-svc-ip>:9090/students   # Need to find this
curl <backend-pod-ip>:8080/students   # Need to find this
```

So in frontend pod, we will be giving <backend-svc-name>:port, backend:9090 in this case.

Deploying Frontend

```
Deploy Frontend
kubectl apply -f frontend.yaml
```

Now we need to access Frontend from browser, so we will expose via Nodeport



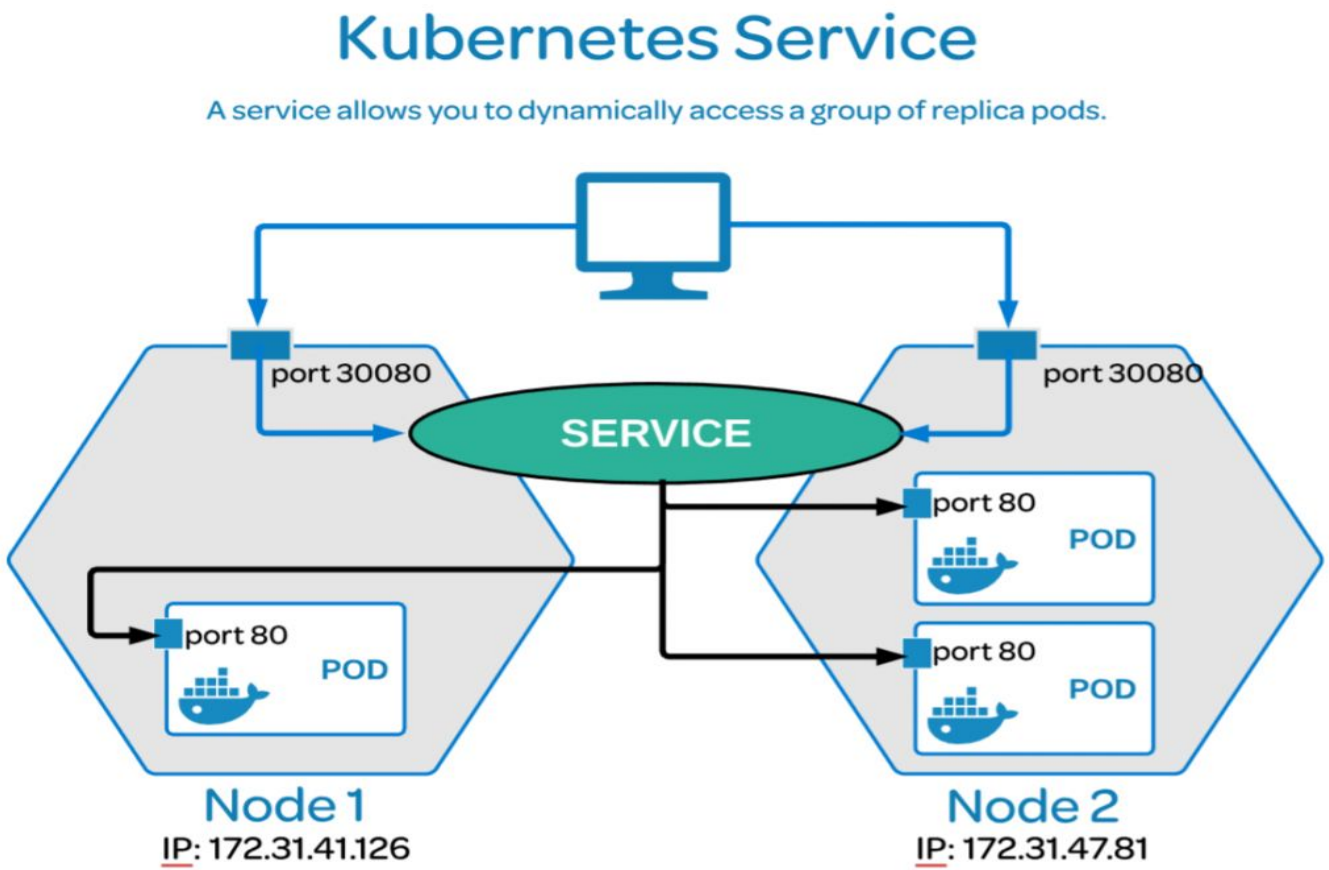
Service Type: NodePort

We will add the value in domain,

frontend-nodeport.kahootali.com ----> Point to any Node IP:NodePort

But what if Node gets deleted, and another one comes in, we will have to update the Node IP again.

NodePort

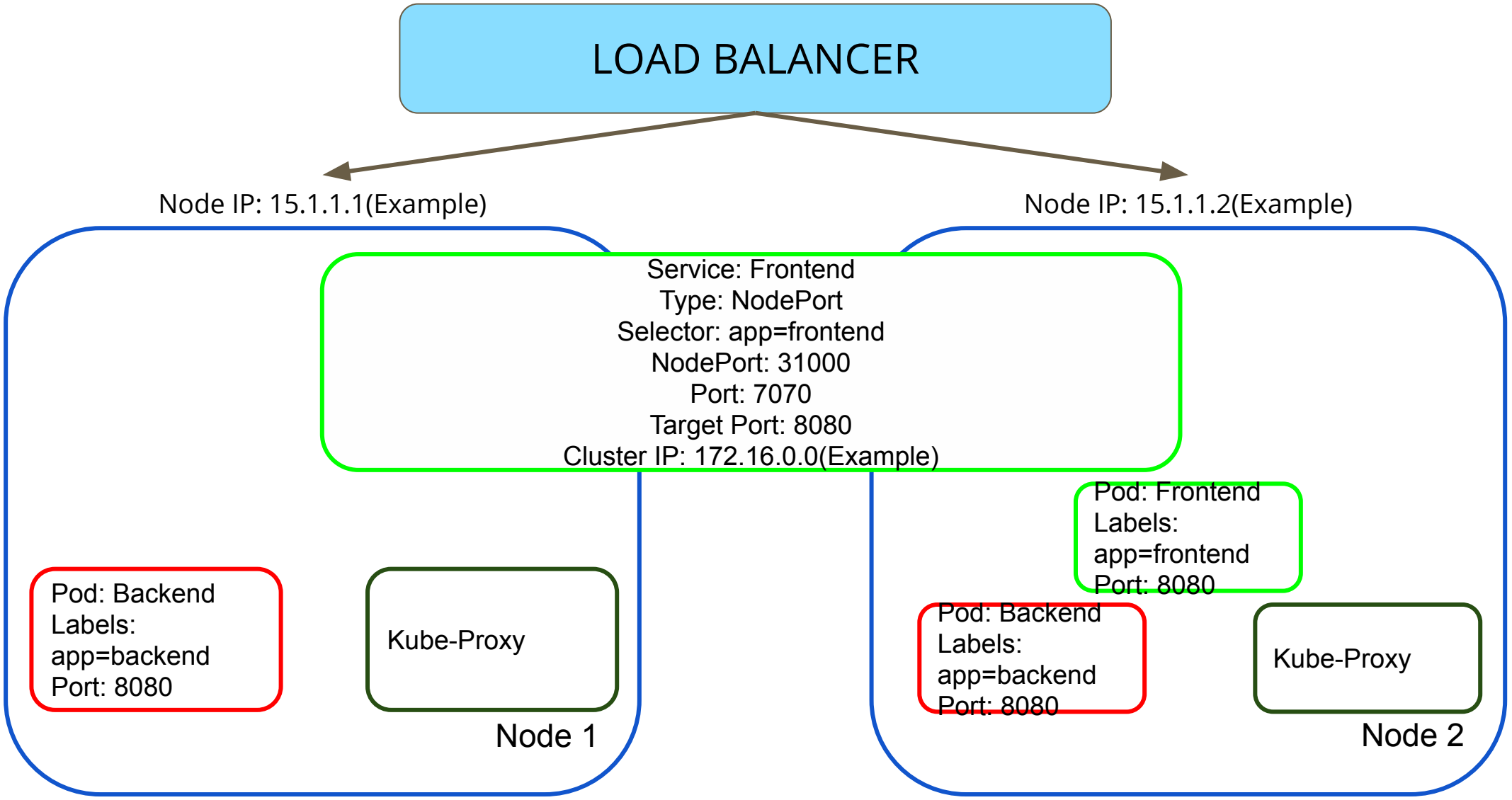

Kubernetes Service
A service allows you to dynamically access a group of replica pods.
port 30080
SERVICE
port 30080
port 80
POD
port 80
POD
port 80
POD
Node 1
IP: 172.31.41.126
Node 2
IP: 172.31.47.81

Service Types

- LoadBalancer: Exposes the Service externally using a cloud provider's load balancer.

- ExternalName: Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value. No proxying of any kind is set up.

Now we will expose Frontend via Load Balancer



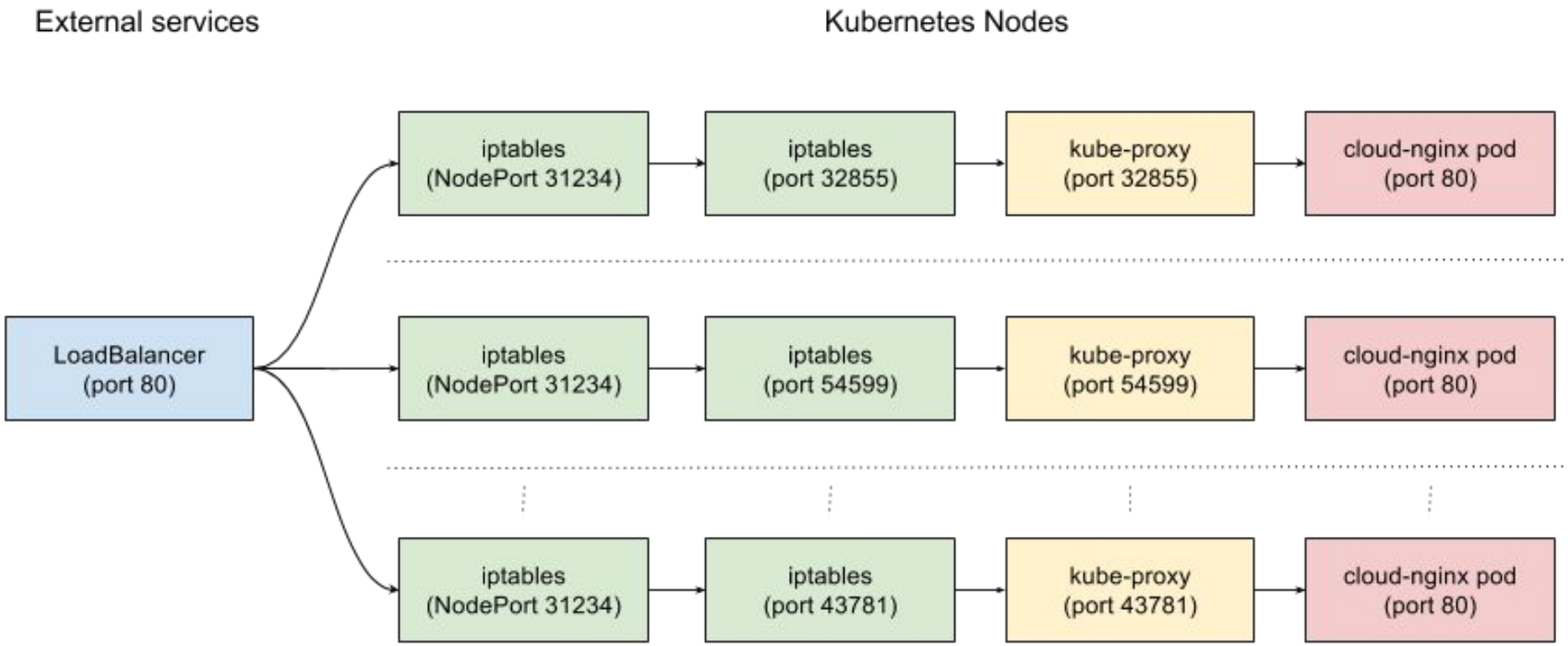
Service Type: LoadBalancer

We will add the value in domain,

frontend-loadbalancer.kahootali.com ----> Point to the LoadBalancer IP

So now even if Nodes get deleted or added, Load Balancer will handle this

Load Balancer



Another way of Exposing Pods

We can port-forward to the pod/service directly, to see whether its working fine or not.

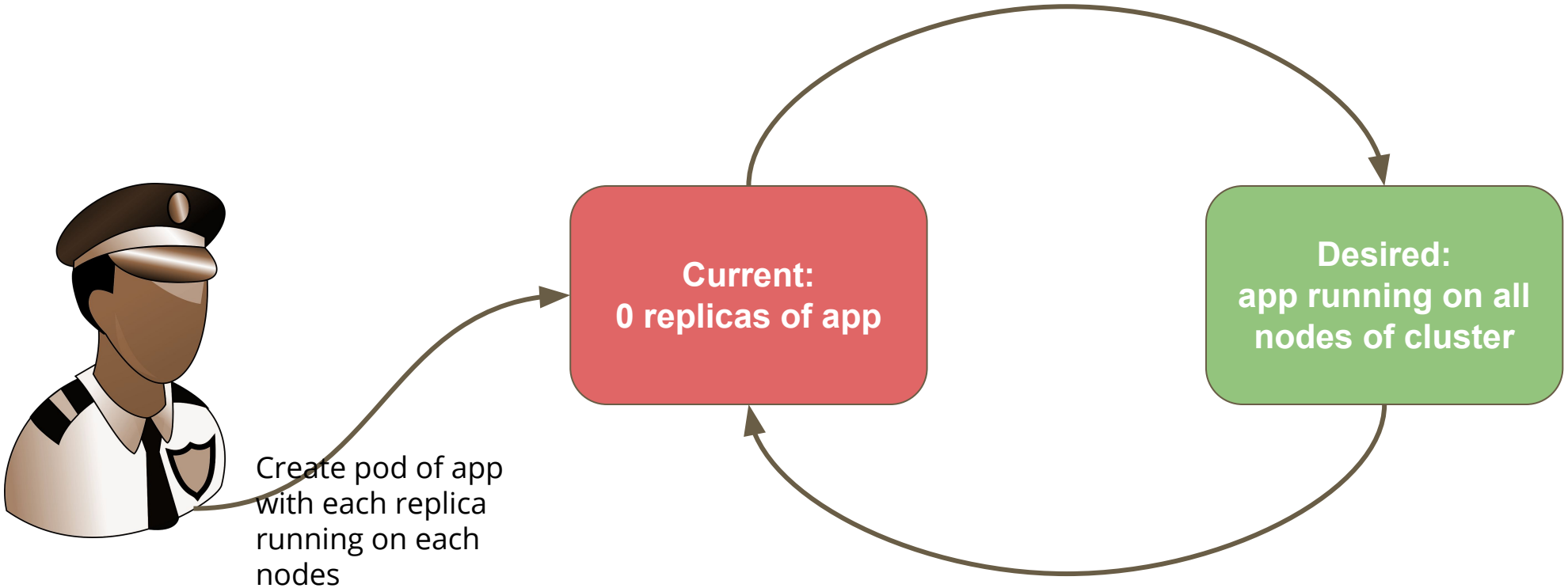
```
kubectl port-forward service/backend 9090:9090
kubectl port-forward deploy/backend 8080:8080
```

Go to localhost:9090/students or localhost:8080/students and can see the backend API running

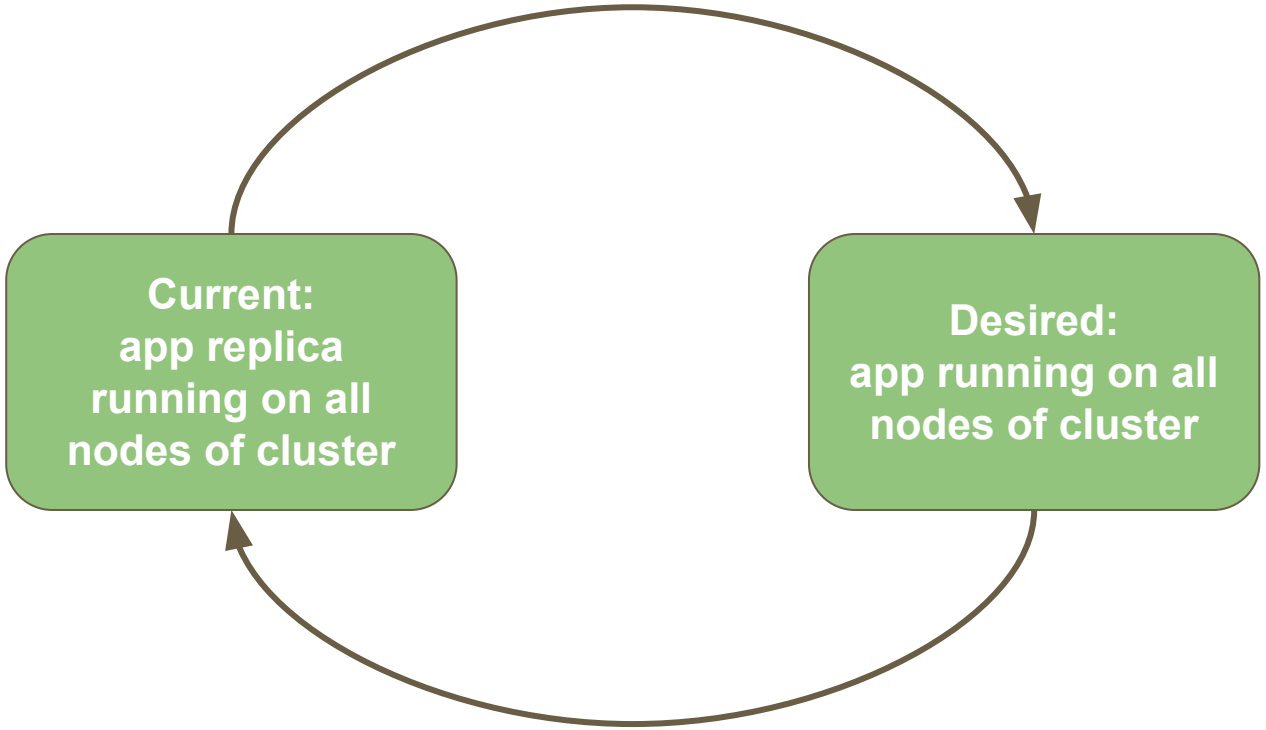
Daemonset

- Almost similar to Replicaset
- Each replica run on each node
- Provides Rolling update
- Can’t do Rollback
- Use Case: Monitoring Exporters, Logging Exporters, etc

Daemonset


Current:
0 replicas of app
Desired:
app running on all
nodes of cluster
Create pod of app
with each replica
running on each
nodes

Daemonset


Current:
app replica
running on all
nodes of cluster
Desired:
app running on all
nodes of cluster

StatefulSets

- They are only used for apps with persistence

Persistence in Kubernetes

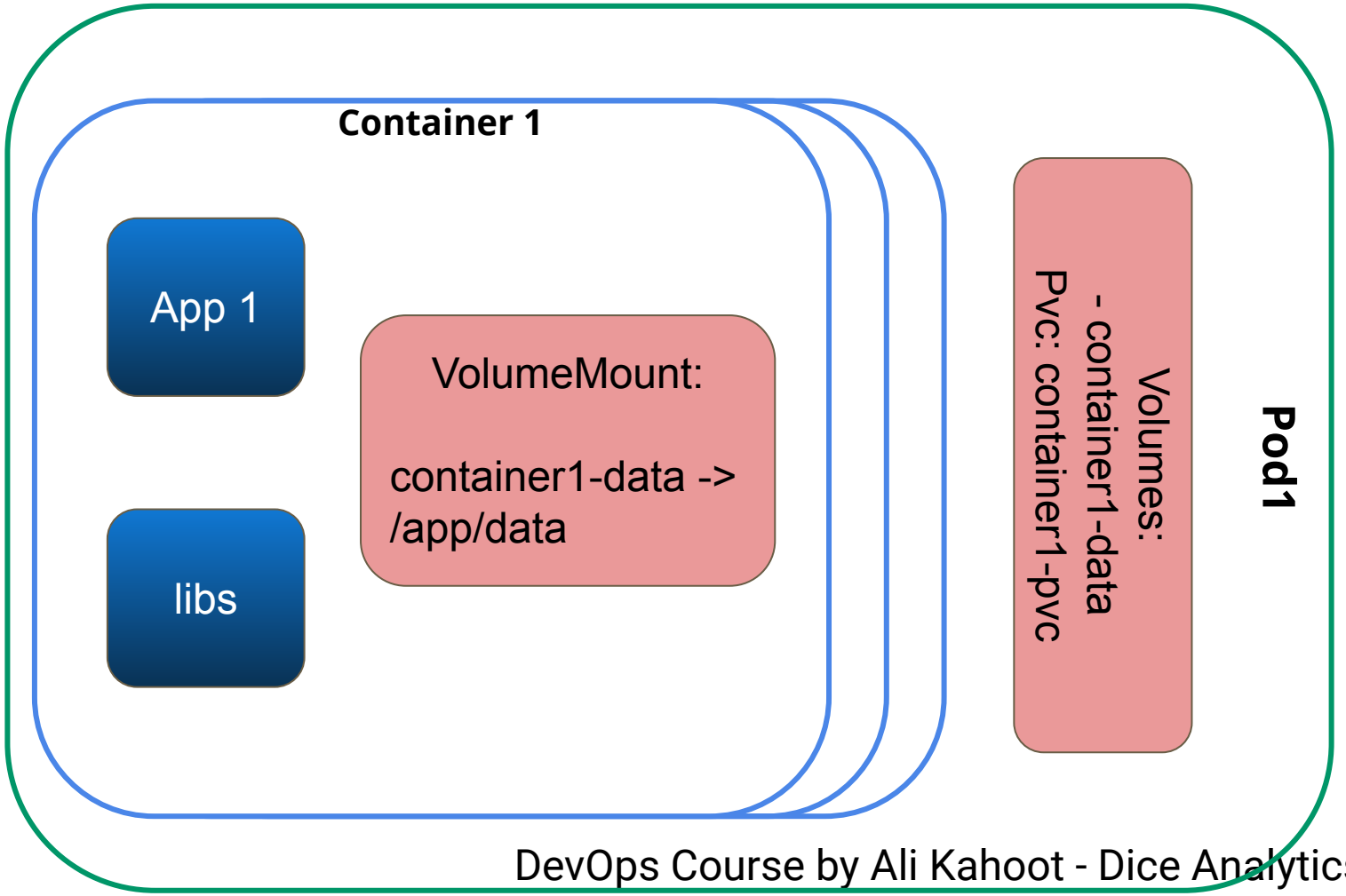
Persistence

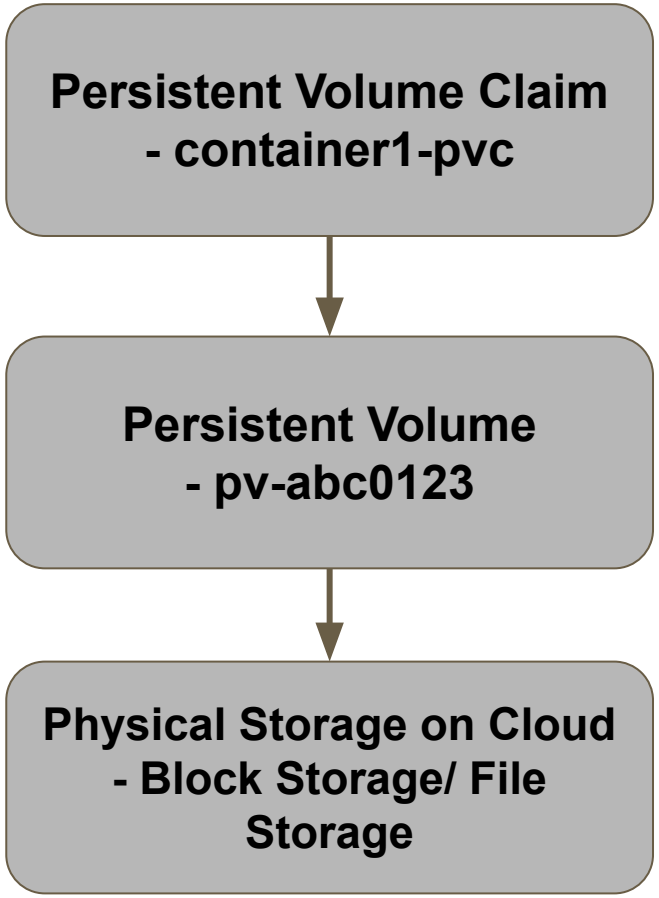
Kubernetes provides Volumes for persisting storage. There are 3 main concepts for Volumes in K8s.

- **PersistentVolume**: the low level representation of a storage volume
- **PersistentVolumeClaim**: the binding between a Pod and PersistentVolume
- **StorageClass**: allows for dynamic provisioning of PersistentVolumes

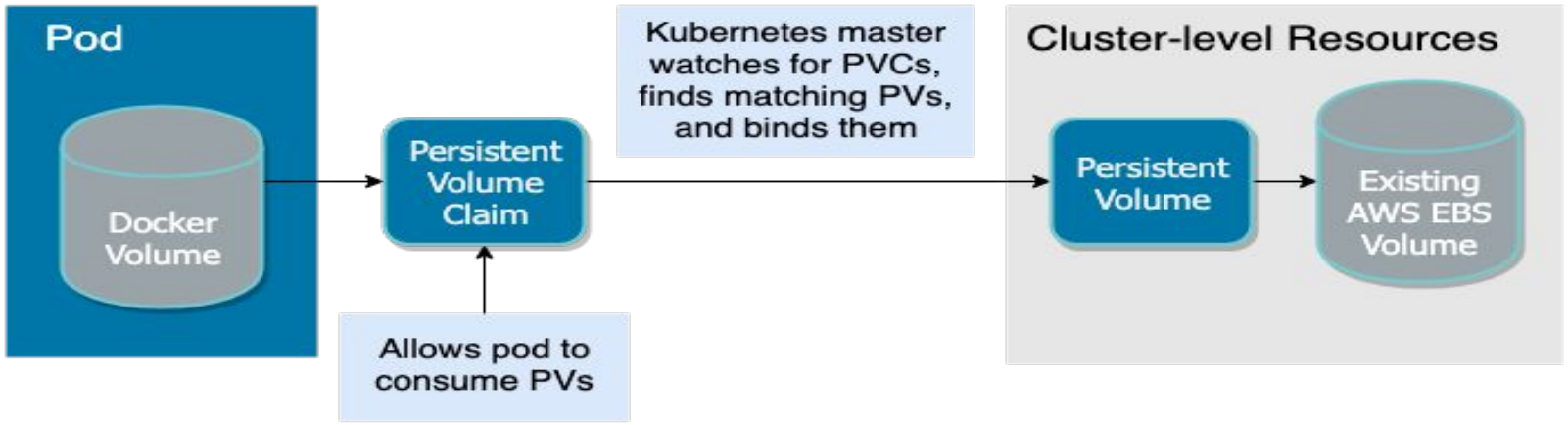
Persistent Volume is the actual storage used in the infrastructure(Cloud) whereas PersistentVolumeClaim is used to bind a part or whole PV to any pod. StorageClass is used to create new PVs dynamically.

Persistence

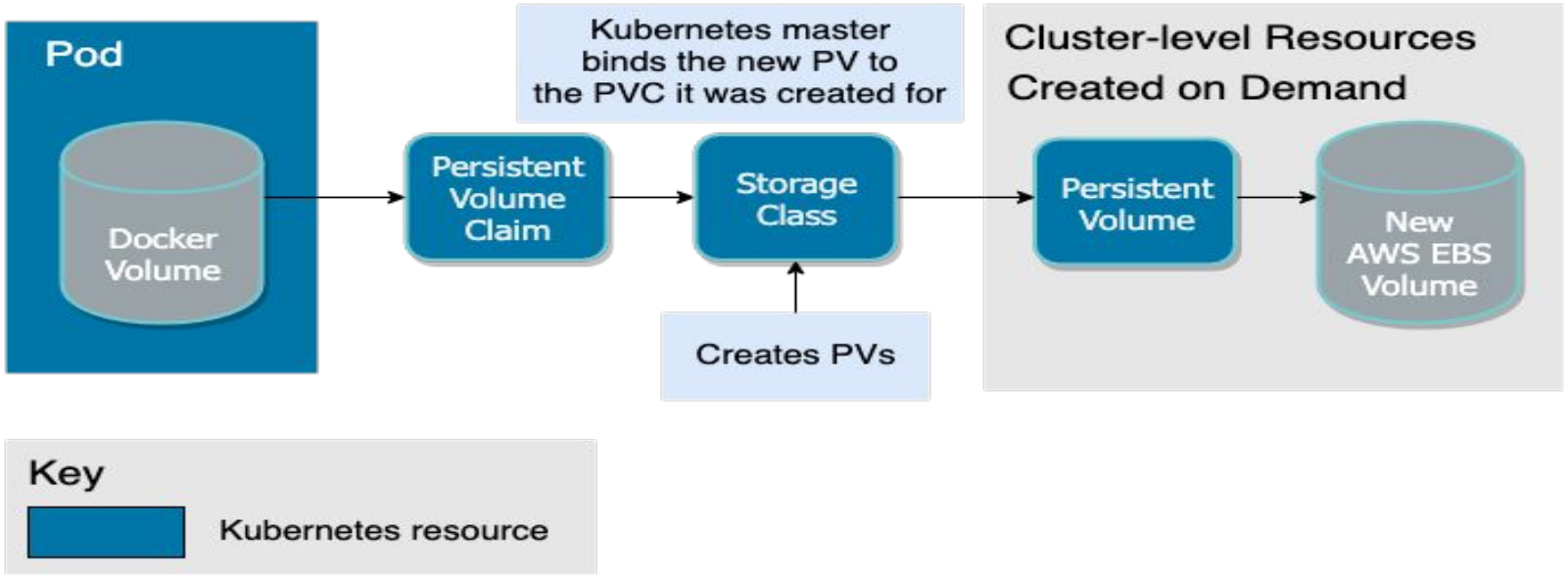




Setting Up Existing Persistent Volumes



Dynamically Provisioning New Persistent Volumes



LAB : VOLUMES - STORAGE CLASS

In minikube, a storageClass is already present with name standard, so we dont need to create PV for it, we just need to create a PVC and use that PVC in the pod.

Check the file **storage-class-app.yaml**

```
kubectl apply -f storage-class-app.yaml
```

LAB : VOLUMES - PERSISTENT VOLUMES

We will be using the Minikube VM's Storage to mount on the Pods, so we will be creating a file in minikube VM, Run following commands

```
minikube ssh

# Once sshd then run following commands

sudo mkdir /mnt/data

sudo sh -c "echo 'Hello from Kubernetes storage' > /mnt/data/index.html"
exit
```

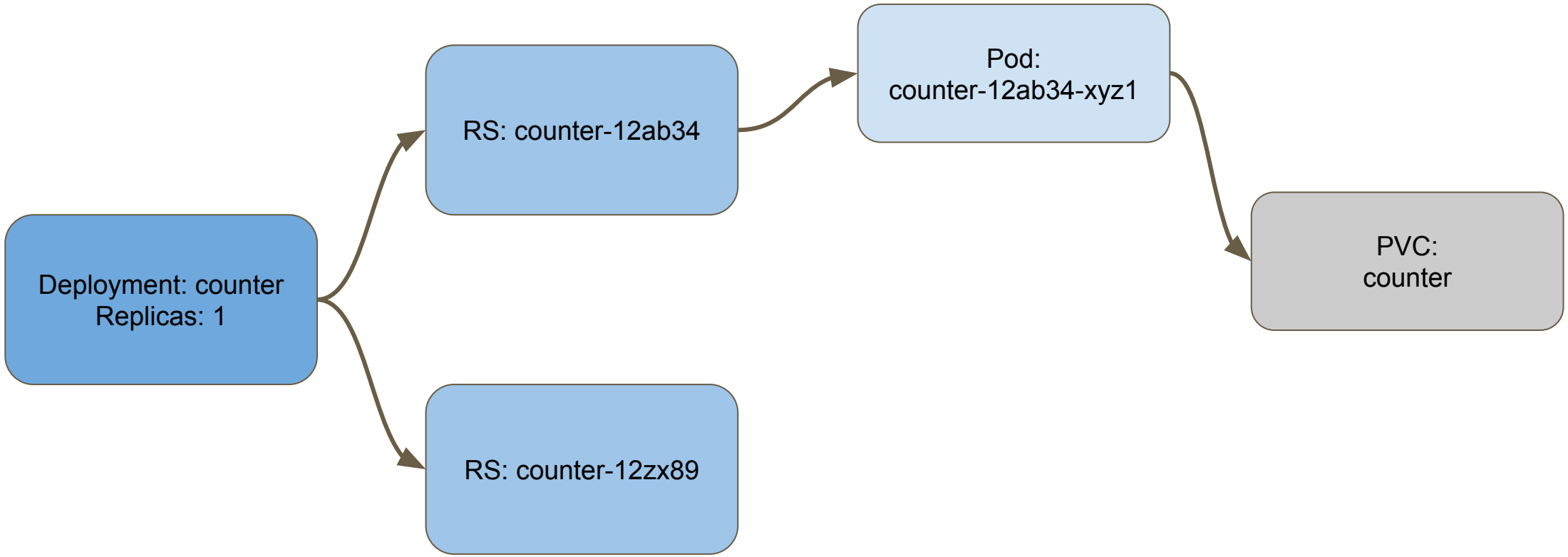
LAB : VOLUMES - PERSISTENT VOLUMES

Check the file **manual-pv.yaml**

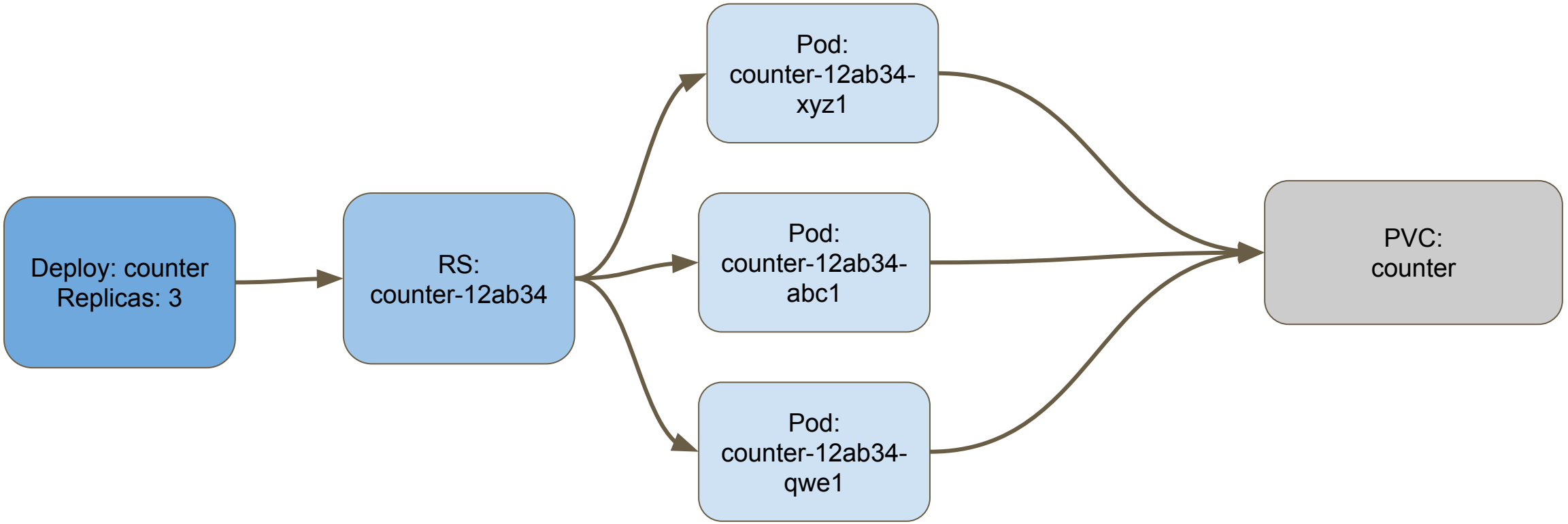
```
kubectl apply -f manual-pv.yaml
```

```
And go to browser: <minikube-ip>:31200
```

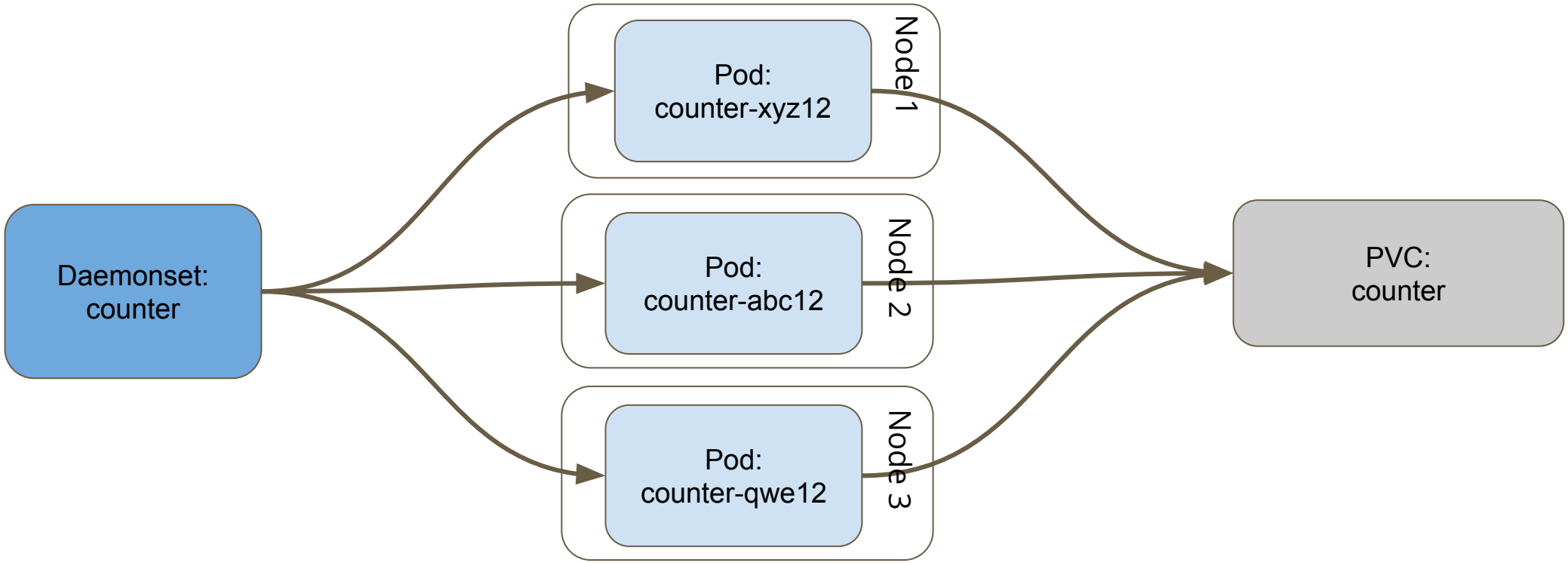
Persistence in Deployments



Persistence in Deployments



Persistence in Daemonsets

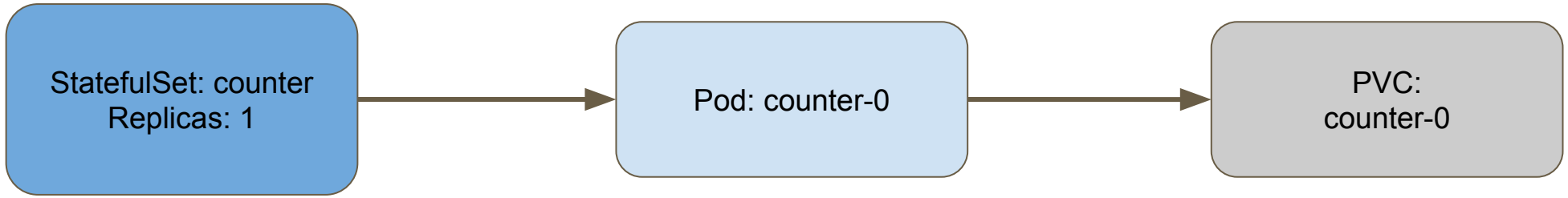


Each Replica running on each node

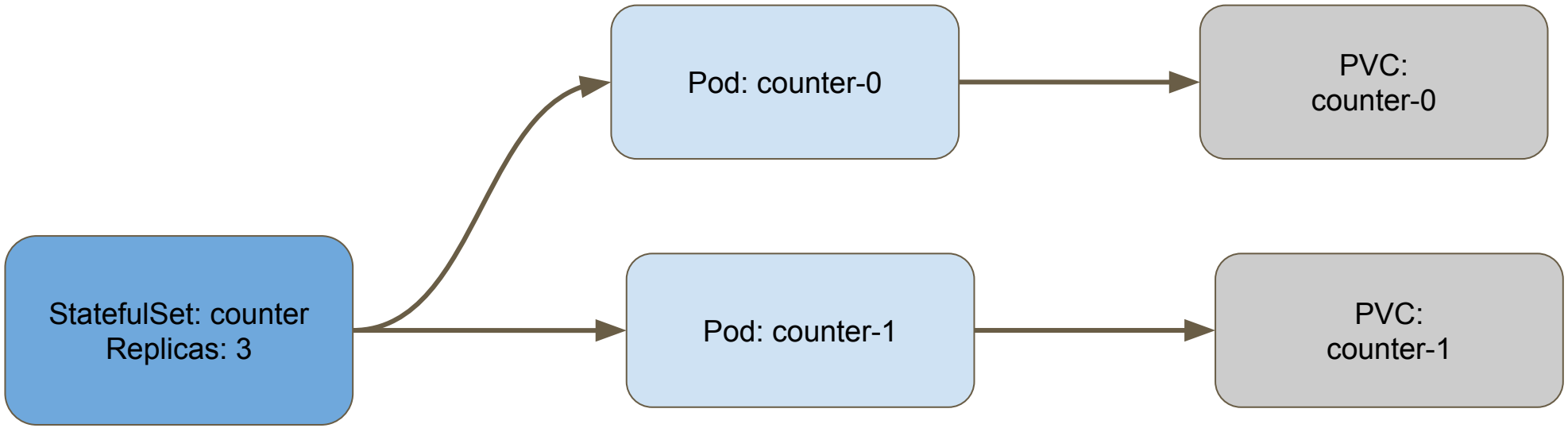
Statefulsets

- Manage Stateful Applications
- Declare PVC template inside a Statefulset manifest
- Guarantees the ordering & uniqueness of pods by having incremental naming convention
- Unlike Deployments, the replica pods are not interchangeable
- Each replica has its own state/identity
- Useful for Databases, etc

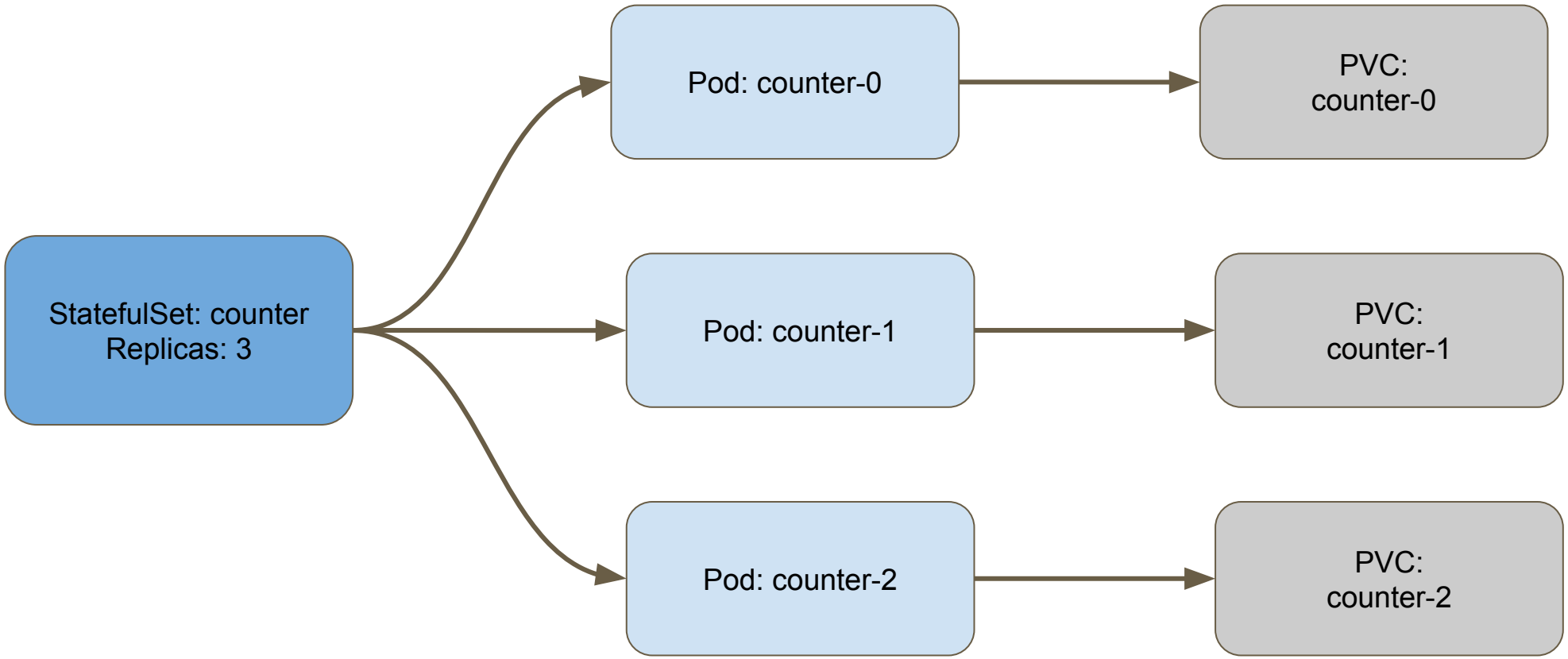
Persistence in Statefulsets



Persistence in Statefulsets



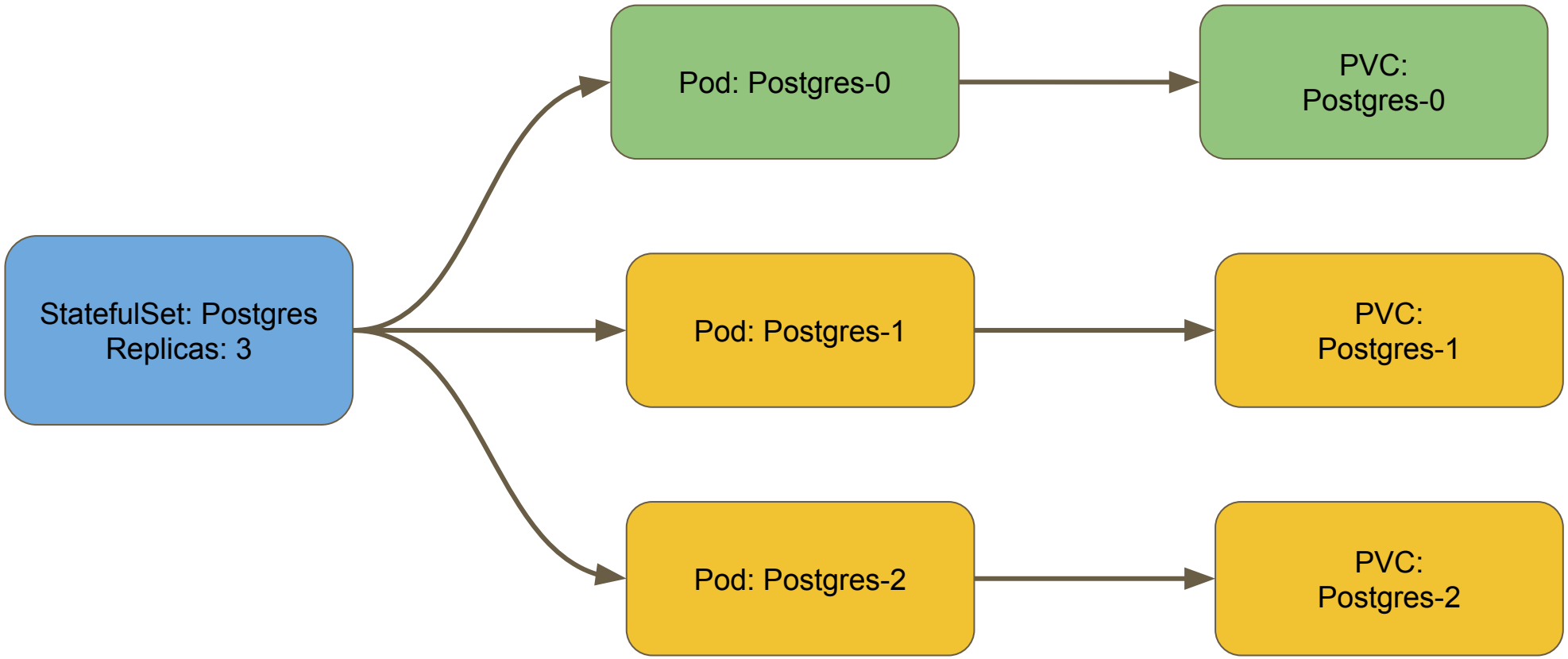
Persistence in Statefulsets



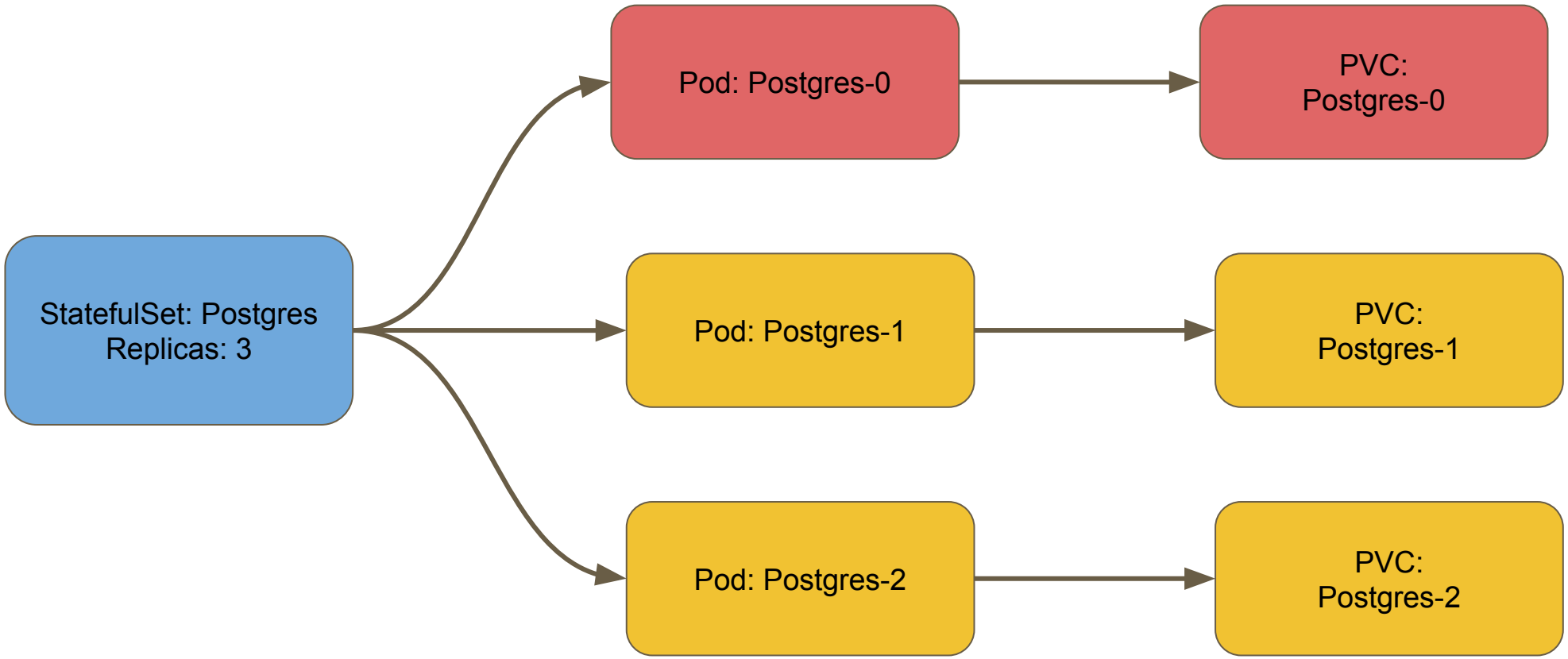
Deployments vs StatefulSet

- If we use Deployment for Databases/Stateful applications, data consistency can be compromised
- High Availability cannot be achieved, as if the PVC is deleted, the data is gone
- StatefulSets don’t have Replicasets so no Rollback option

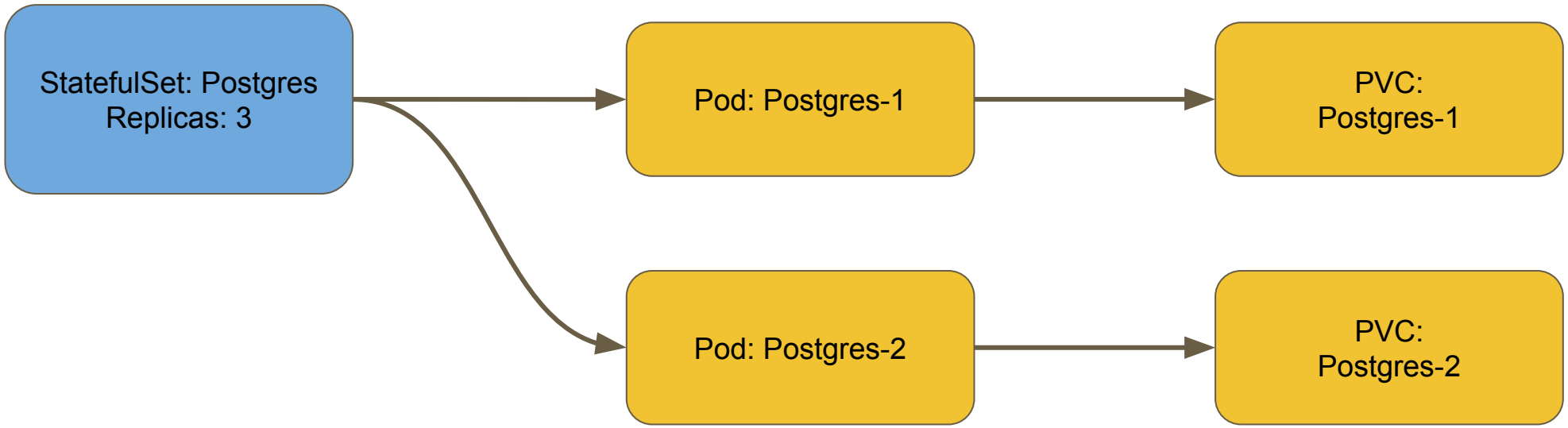
Using DBs as clusters for HA



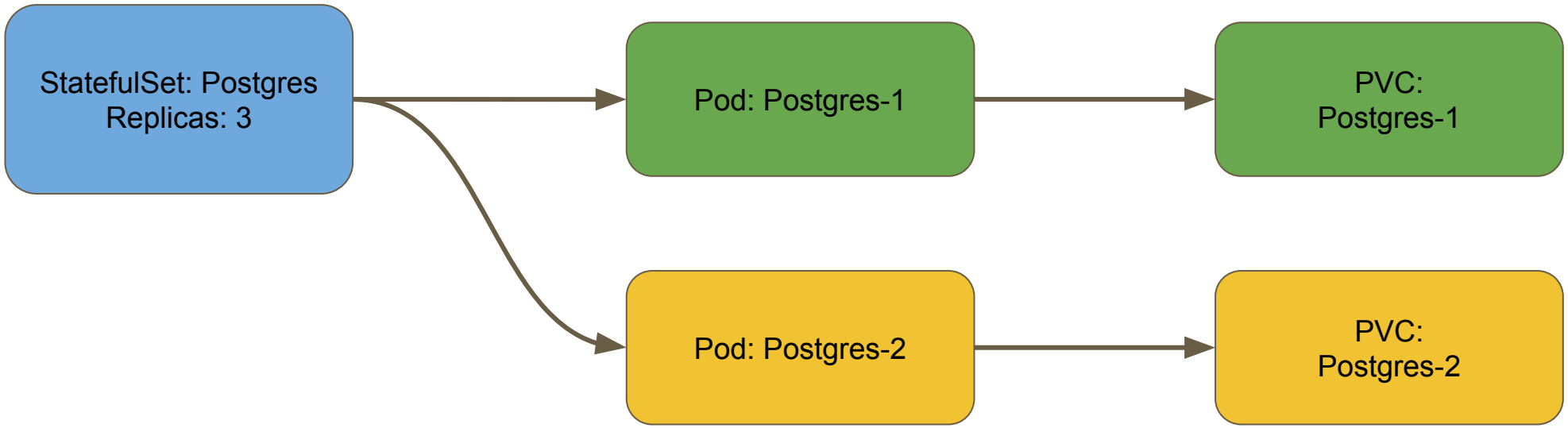
Using DBs as clusters for HA



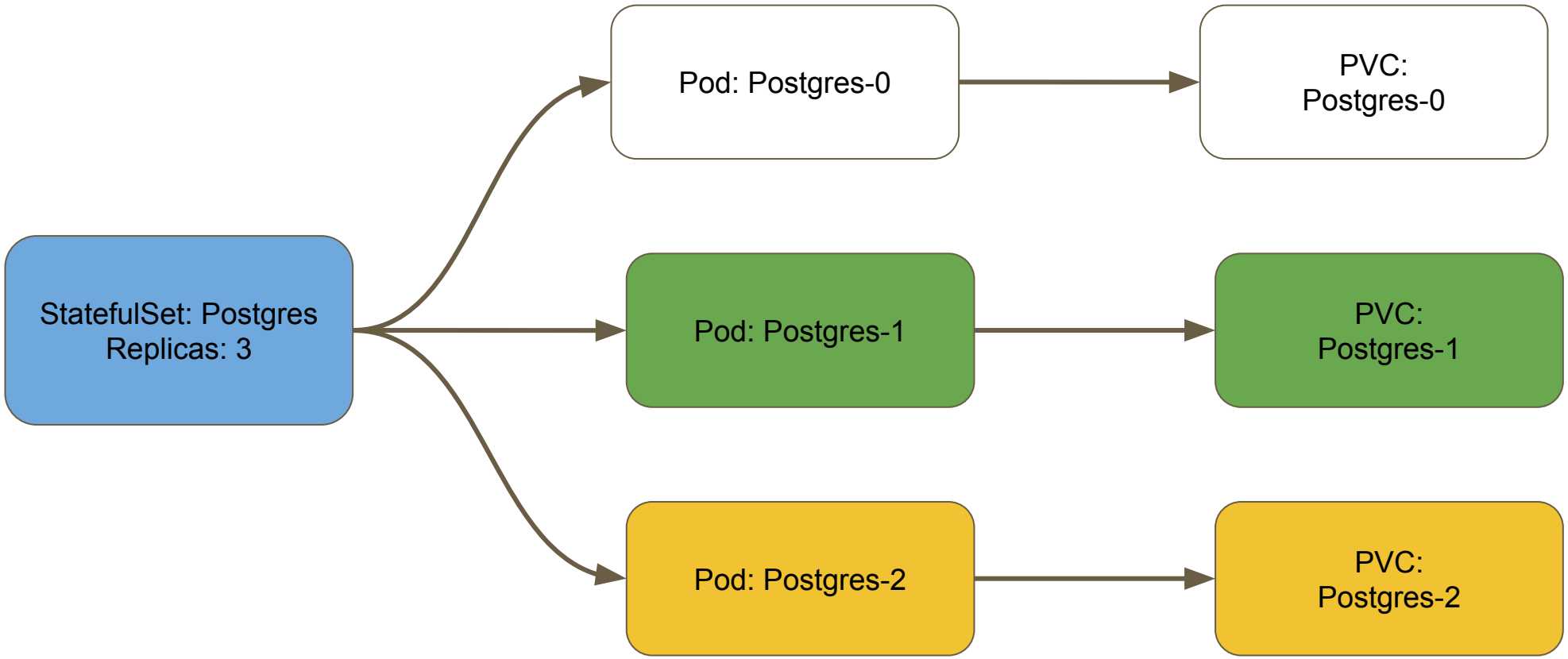
Using DBs as clusters for HA



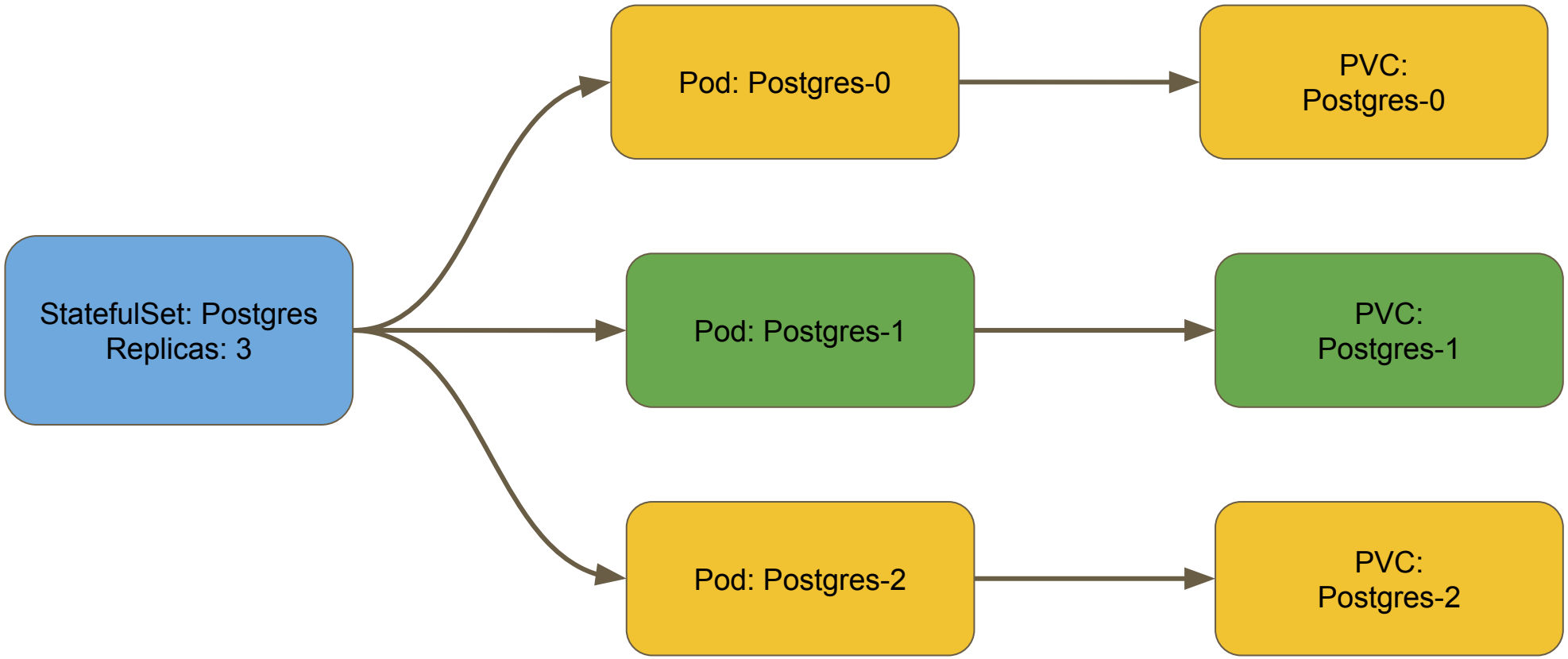
Using DBs as clusters for HA



Using DBs as clusters for HA



Using DBs as clusters for HA



Why Statefulsets for DBs

- Databases form clusters(primary/secondary replicas) to provide High Availability
- Each individual pod should have a unique identity
- As the cluster of DB communicates with each other so they should be able to predict other pod names.
- Even if one pod goes down,it will come back with the same name as before so networking will be easy
- PVC doesn’t delete automatically even if Pods/StatefulSet delete

CONFIGMAP

- Kubernetes objects for injecting containers with configuration data
- ConfigMaps hold Key-Value pairs accessible to Pods
- Designed to detach configuration from container images
- They can be used to store fine-grained or coarse-grained configuration
  - Individual properties
  - Entire configuration file
  - JSON files
- Separate configuration from the rest of your application.

LAB - CREATING CONFIGMAPS FROM FILES

If you have a lot of configuration settings, the best option is to store them in files and creating ConfigMaps from those files. First create a text file with some configuration.

```
color.primary=purple
color.brand=yellow
font.size.default=14px
```

Save this configuration in file named front-end

LAB - CREATING CONFIGMAPS FROM FILES

Now create configmap from previous file using --from- file argument

```
kubectl create configmap front-end-config --from-file=front-end
```

You can see a detailed information about a new ConfigMap by running the following commands

```
kubectl describe configmap front-end-config
```

CREATING CONFIGMAPS FROM LITERAL VALUES

You can create a ConfigMap from the literal value using --from-literal argument as shown below.

```
kubectl create configmap some-config \
--from-literal=font.size=14px \
--from-literal=color.default=green
```

You can get a detailed description of the new ConfigMap using the following command:

```
kubectl get configmap some-config -o yaml
```

CREATE A CONFIGMAPS SPEC

Check the file in configmap folder, **trading-strategy.yaml.** You can define a Configmap through a K8s manifest. CongMap manifests look similar to API resources we've already discussed.

Create a ConfigMap running the following command:

```
kubectl create -f trading-strategy.yaml
```

Check if the ConfigMap was successfully created:

```
kubectl get configmap trading-strategy -o yaml
```

USING CONFIGMAPS IN PODS

Till now we have defined ConfigMap, now we will use it in the Pod. We can use either a one field using Environment Variable or can mount the complete Configmap file.

Check the file **demo-envr.yaml** and **demo-config-volume.yaml.**

Apply these file and then we can exec into the pods, and see the values.

```
kubectl apply -f demo-envr.yaml
kubectl apply -f demo-config-volume.yaml
```

USING CONFIGMAPS AS ENVIRONMENTAL VARIABLES

Once the pod is ready and running, get a shell to the container:

```
kubectl exec -it demo-envr -- /bin/bash
```

From within the container, you can access configuration as environmental variables by using printenv command.

```
printenv STRATEGY_TYPE
printenv STRATEGY_RISK
```

INJECTING CONFIGMAPS INTO THE CONTAINER'S VOLUME

Once the pod is ready and running, get a shell to the container

```
kubectl exec -it demo-config-volume -- /bin/bash
```

and check the /etc/config folder:

```
ls /etc/config/
```

As you see, Kubernetes created files in that folder. Each file's name is derived from the key name and the file contents are the key value. You can verify this easily:

```
cat /etc/config/strategy.risk
```

SECRETS

- Similar to Configmap but a Mechanism to use such information in a safe and reliable way
- You can access them via a volume or an environment variable from a container running in a pod
- The secret data on nodes is stored in tmpfs volumes
- A per-secret size limit of 1MB exists
- The API server stores secrets as plaintext in etcd
- Secrets are registered with Kubernetes Master

POINTS TO CONSIDER

- Secrets & Configmaps needs to be created before any pods that depend on it
- Individual secrets are limited to 1MB in size
- They can only be referenced by pods in same namespace
- Once a pod is started, its secret volumes will not change, even if the secret resource is modified

LAB: CREATING SECRETS FROM FILES

You can create secrets from files. First create the files with the username and password on your local machine

```
echo -n 'admin' > username.txt
echo -n 'password' > password.txt
```

Use kubectl create secret command to package these files into a Secret and create a Secret API object on the API server

```
kubectl create secret generic db-auth --from-file=username.txt
--from-file=password.txt
```

LAB: CREATING SECRETS FROM FILES

Check if the secret was created successfully by running the following command:

```
kubectl get secrets
```

You can see the detailed information by running the following command:

```
kubectl describe db-auth
```

LAB: CREATING SECRETS FROM FILES

You can retrieve your Secret data in the base64 encoding by running the following command:

```
kubectl get secret db-auth -o yaml
```

You can easily decode the Secret

```
echo "cGFzc3dvcmQK" | base64 --decode
```

CREATING SECRETS FROM LITERAL VALUES

Another option is to create Secrets from literal values. You will manually provide kubectl with credentials as shown below

```
kubectl create secret generic literal-secret \
--from-literal=username=supergiant \
--from-literal=password=Niisdfiis
```

CREATING SECRETS FROM LITERAL VALUES

Check newly created secret:

```
kubectl get secret literal-secret -o yaml
```

Get base64 value of the password and decode by running the following command:

```
echo "Tmlpc2RmaWlz" | base64 --decode
```

CREATING A SECRET SPEC

Another option is to create Secrets using a Secret spec. In this case, we need to convert the username and password data into base64 encoding manually.

```
echo -n 'admin' | base64

YWRtaW4=

echo -n 'jiki893kdjnsd9s' | base64
amlraTg5M2tkam5zZDlz
```

CREATING A SECRET SPEC

See the file **test-secret.yaml** we can define non-base64 values as well by using stringData instead of Data

Now create the Secret by running the following command:

```
kubectl create -f test-secret.yaml
```

USING SECRETS IN PODS

Most common option is mounting Secrets as data volumes at some location within your container. You will create a pod to see how this works:

Check file **pod-with-secret.yaml** which mounts a complete secret at a path

USING SECRETS IN PODS

Create the pod

```
kubectl create -f pod-with-secret.yaml
```

Check if the volume mounted at the /etc/secrets path of our MongoDb container actually contains the Secret. First, get a shell to the container running the following command:

```
kubectl exec -it pod-with-secret -- /bin/bash
```

USING SECRETS IN PODS

When inside the container, list the /etc/secrets folder:

```
ls /etc/secrets
password username
```

Run the following command to verify secret within the container, run:

```
cat /etc/secrets/username
cat /etc/secrets/password
```

You should see the credentials you used to create the secret

USING SECRETS AS ENVIRONMENTAL VARIABLES

Environmental variables can be used for storing Secret keys as well. We can store Secrets in environmental variables using the spec.containers.env field with a set of name-values in it

Check file **secret-env.yaml** it mounts secret keys as environment variables on the pods

USING SECRETS AS ENVIRONMENTAL VARIABLES

The file above uses two environmental variables: SECRET_USERNAME and SECRET_PASSWORD , which take their values from the username and password keys of the test-secret.

Create the pod as usual:

```
kubectl create -f secret-env.yaml
```

You can access components of the Secret as simple environmental variables inside the container. Go to shell of the running container:

```
kubectl exec -it secret-env -- /bin/bash
```

USING SECRETS AS ENVIRONMENTAL VARIABLES

Once inside the container, run:

```
echo $SECRET_USERNAME
echo $SECRET_PASSWORD
```

The components of your Secret are available as environmental variables inside the container.

PROBES

- Probes are a way to tell Kubernetes that whether your application is running fine or is ready to accept requests or not.
- Kubernetes provides way for self-healing, and high availability.
- There are two types of probes:
- Readiness Probe
- Liveness Probe

LIVENESS PROBE

Through liveness probe, Kubelet keeps checking it to confirm the container is up or got down.

Suppose that a Pod is running our application inside a container, but due to some reason let’s say memory leak, cpu usage, application deadlock etc the application is not responding to our requests, and stuck in error state.

READINESS PROBE

Through readiness probe, Kubelet checks at first, and checks if the pod has started or not. We can have a probe, to check whether the application inside the container is up and only then mark it as ready.

Sometimes, applications are temporarily unable to serve traffic. For example, an application might need to load large data or configuration files during startup, or depend on external services after startup. In such cases, you don’t want to kill the application, but you don’t want to send it requests either. Kubernetes provides readiness probes to detect and mitigate these situations. A pod with containers reporting that they are not ready does not receive traffic through Kubernetes Services.

READINESS PROBE

There are two ways to check Liveness & Readiness probes

1. Executing a command
2. Through httpget

1. EXECUTING A COMMAND

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 3
      periodSeconds: 5
```

2. THROUGH HTTPGET

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/liveness
    args:
    - /server
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
      initialDelaySeconds: 3
      periodSeconds: 3
```

LAB: LIVENESS PROBE

Check the file **golang-liveness-deploy.yaml.** This will create a deployment with liveness probe enabled. Any code greater than or equal to 200 and less than 400 indicates success for `healthz` endpoint. Any other code indicates failure.

LAB: LIVENESS PROBE

For the first 10 seconds that the Container is alive, the /healthz handler returns a status of 200. After that, the handler returns a status of 500.

```
http.HandleFunc("/healthz", func(w http.ResponseWriter, r *http.Request)
{
    duration := time.Now().Sub(started)
    if duration.Seconds() > 10 {
        w.WriteHeader(500)
        w.Write([]byte(fmt.Sprintf("error: %v", duration.Seconds())))
    } else {
        w.WriteHeader(200)
        w.Write([]byte("ok"))
})
```

LAB: LIVENESS PROBE

Now check the pods

```
kubectl get pods
```

it will be ready and no restarts. And after 20 seconds, check again, it will have a restart.

LAB: READINESS PROBE

Check the file **golang-readiness-deploy.yaml.** It has readiness probe

Apply the file

```
kubectl apply -f golang-readiness-deploy.yaml
```

Now check the pods `kubectl get pods`. It will not be ready for the first 30 seconds.

LAB: READINESS PROBE

This will create a deployment with readiness probe enabled. Any code greater than or equal to 200 and less than 400 indicates success for `health` endpoint. Any other code indicates failure.

Now For the first 30 seconds, it will delay initially but then will call the http GET endpoint of `/health` and if status is 200 it will mark the pod ready. And Service will add it to its pool.

NAMESPACES

Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces.

Namespaces are intended for use in environments with many users spread across multiple teams, or projects.

Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces.

You can list the current namespaces in a cluster using:

```
kubectl get namespaces
```

LAB: NAMESPACE CREATION & DEPLOYING OBJECTS

Create 2 new namespaces by running

```
kubectl create ns dev
kubectl create ns prod
```

Check the files **golang-dev.yaml** & **golang-prod.yaml**

LAB: NAMESPACE CREATION & DEPLOYING OBJECTS

Get resources by

```
kubectl get pods -n dev
```

```
kubectl get pods -n prod
```

In this way you can divide your resources in multiple sub-environments.

Internamespace Service Calling

Till yet, we were calling service by service-name, but if we want to call a service in another namespace, we can append namespace name at end of it.

E.g Frontend running in prod wants to call backend in dev so it can call

Backend.dev:8080

Actually the url is

`<service-name>.<namespace>.svc.cluster.local:<service-port>`

LAB: Namespaces on Actual Cluster

Download the file I shared for kube-config of the cluster, save it in **.kube** folder with a file named **config-kops**

Now, to use the new cluster, you will have to use all the commands with

```
--kubeconfig ~/.kube/config-kops
```

```
kubectl get nodes --kubeconfig ~/.kube/config-kops
kubectl get pods --kubeconfig ~/.kube/config-kops
```

LAB: Namespaces on Actual Cluster

Now create namespace for yourself

```
kubectl create ns <YOUR_NAME> --kubeconfig ~/.kube/config-kops
```

Now a namespace will be created, you can run a simple app now in your namespace

Update ghost.yaml file with your NodePort, from range 31001 - 31010 and update on slack and then run

```
kubectl apply -f ghost.yaml -n <YOUR_NAME> --kubeconfig ~/.kube/config-kops
```

RBAC

- Role-based access control (RBAC) is a method of regulating access to computer or network resources based on the roles of individual users within an enterprise.
- Through RBAC you can specify rights for a user, what things are allowed for him to do.
- You can give specific roles to a Service Account/User so that user can perform only those rules and nothing else.

ROLE, CLUSTERROLES

In the RBAC API, a role contains rules that represent a set of permissions. Permissions are purely additive(there are not a denial rules). A role can be defined within a namespace with a Role, or cluster-wide with a ClusterRole.

A Role can only be used to grant access to resources within a single namespace. Here is an example Role in the default namespace that can be used to grant read access to pods:

ROLE, CLUSTERROLES

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

ROLE, CLUSTERROLES

ClusterRole:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
# "namespace" omitted since ClusterRoles are not namespaced
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

ROLEBINDING & CLUSTERROLEBINDING

A role binding grants the permissions defined in a role to a user or set of users. It holds a list of subjects (users, groups, or service accounts), and a reference to the role being granted. Permissions can be granted within a namespace with a RoleBinding, or cluster-wide with a ClusterRoleBinding.

ROLEBINDING & CLUSTERROLEBINDING

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: ServiceAccount
  name: default
  apiGroup: ""
roleRef:
  kind: Role #this must be Role or ClusterRole
  name: pod-reader # this must match the name of the Role or ClusterRole you wish to bind to
  apiGroup: "rbac.authorization.k8s.io"
```

ROLEBINDING & CLUSTERROLEBINDING

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: read-pods
subjects:
- kind: ServiceAccount
  name: default
  namespace: default
roleRef:
  kind: ClusterRole #this must be Role or ClusterRole
  name: pod-reader # this must match the name of the Role or ClusterRole you wish to bind to
  apiGroup: "rbac.authorization.k8s.io"
```

LAB: RBAC

Check rbac/ folder. We have 2 applications, one Chowkidar which reads all pods, so it needs permissions to list and get pods.

Other is kubectl pod, through which we can run all kubectl commands and give corresponding RBAC for that pod to run kubectl commands

HELM: K8S MANIFESTS ISSUES

- Vanilla Manifests: Plain kubernetes manifests like the one we used previously
- but for deploying an application, you might need multiple manifests(deployments, services, namespaces, rbac, configmaps, secrets, volumes)
- Then need multiple manifests for multiple environments, lot of code duplication
- Validate release state after deployment

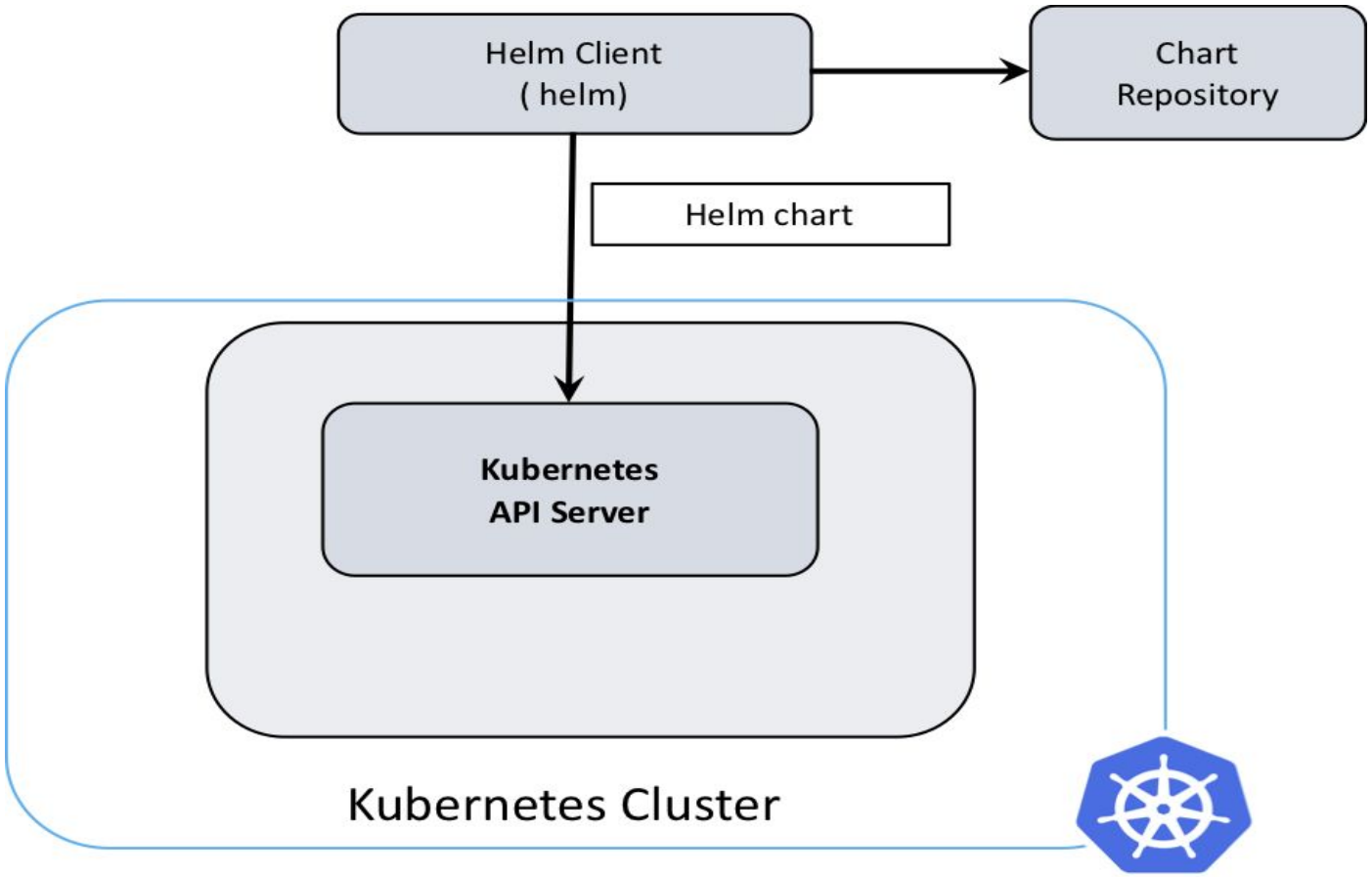
HELM

- Jointly started with Google & Deis
- Now under CNCF
- Work with any K8s cluster
- Makes application deployment easy, standard and reusable
- Improves developer productivity
- Enhances operational readiness
- Package manager for Kubernetes manifests
- Can package multiple manifests into a single chart
- Can parameterize those values for multiple environments

HELM TERMINOLOGY

- Chart: a package bundle of Kubernetes resources
- Release: Deploying a chart with some set of values, a cluster can have multiple releases of a single chart
- Repository: a repository of published Charts
- Template: a file to provide function for parameterizing

HELM ARCHITECTURE



HELM ARCHITECTURE

Helm is a Production Ready Chart manager which can be used to define, install, and upgrade even the most complex Kubernetes application.

INSTALL HELM CLIENT

Run following commands

```
curl -fsSL -o get_helm.sh
https://raw.githubusercontent.com/helm/helm/master/scripts/get-hel
m-3
chmod 700 get_helm.sh
./get_helm.sh
```

Confirm helm is installed by running `helm repo list`

LAB: CREATING A HELM CHART

Fork & Clone `https://github.com/kahootali/helm-charts`

, which contains the frontend and backend that we created previously using helm chart.

Now cd into that repo.

And run

```
helm install my-application application
```

and then go to browser and check:

- <minikube-ip>:31000/hello
- <minikube-ip>:31000/instructor/1

LAB: CREATING A HELM CHART

Now delete the release by

```
helm delete my-application
```

We will deploy again by using our custom values now.

```
helm install application --name dice-application -f
application/values-custom.yaml
```

So we can override the default values for multiple environments and can templatize things.

LAB: RUNNING A STABLE CHART

Helm used to manage a lot of charts that are commonly used, if you want to see go to
https://github.com/helm/charts

but now helm is not maintaining it rather.opensource contributors or companies who create the tool are maintaining the charts, you can see from the above link the current maintainer.

Install Wordpress

E.g. we can see that wordpress chart has been deprecated and has been moved to bitnami/charts by going to

https://github.com/helm/charts/tree/master/stable/wordpress

So now we will install wordpress using bitnami chart.

Install Wordpress

```
helm repo add bitnami https://charts.bitnami.com/bitnami

helm install wordpress bitnami/wordpress --set
service.type=NodePort --set service.nodePorts.http=31500
--set service.nodePorts.https=32500
```

Install Wordpress

Go to dashboard and wait almost 2-3 minutes to see everything comes up Green.

Then go to:

<minikube-ip>:31500

And you will see wordpress running,

You can log in by:

<minikube-ip>:31500/admin

User: user

Install Wordpress

Get the Password by running

```
echo $(kubectl get secret --namespace default wordpress -o
jsonpath="{.data.wordpress-password}" | base64 --decode)
```

Go to admin and complete the setup

In the end you can delete the release by

```
helm delete wordpress
```