# Kubernetes Deep Dive

IBM **Developer**

Sudharshan Govindan

Developer Advocate

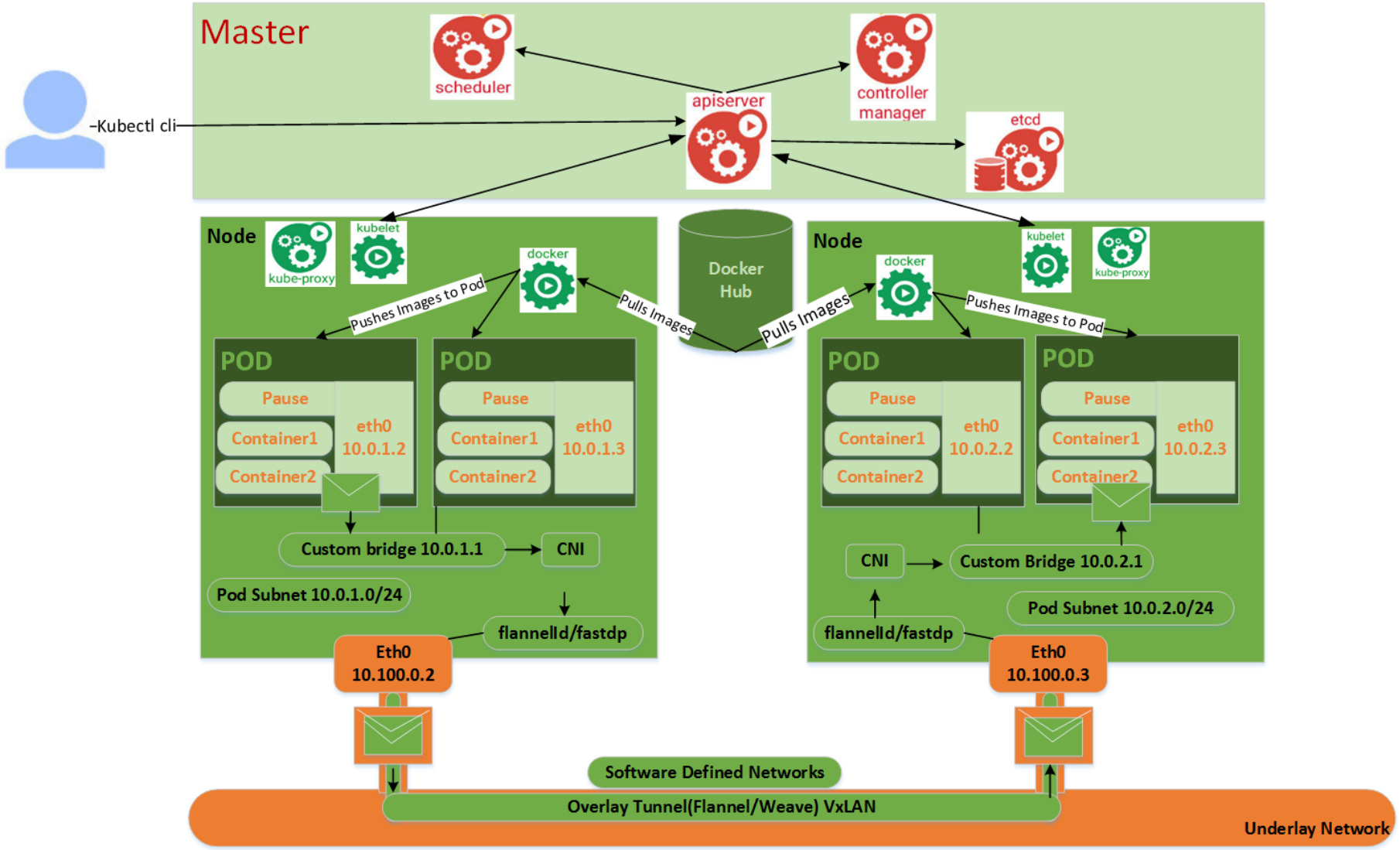
sudharshan.govindan@in.ibm.com

@sudhargovindan

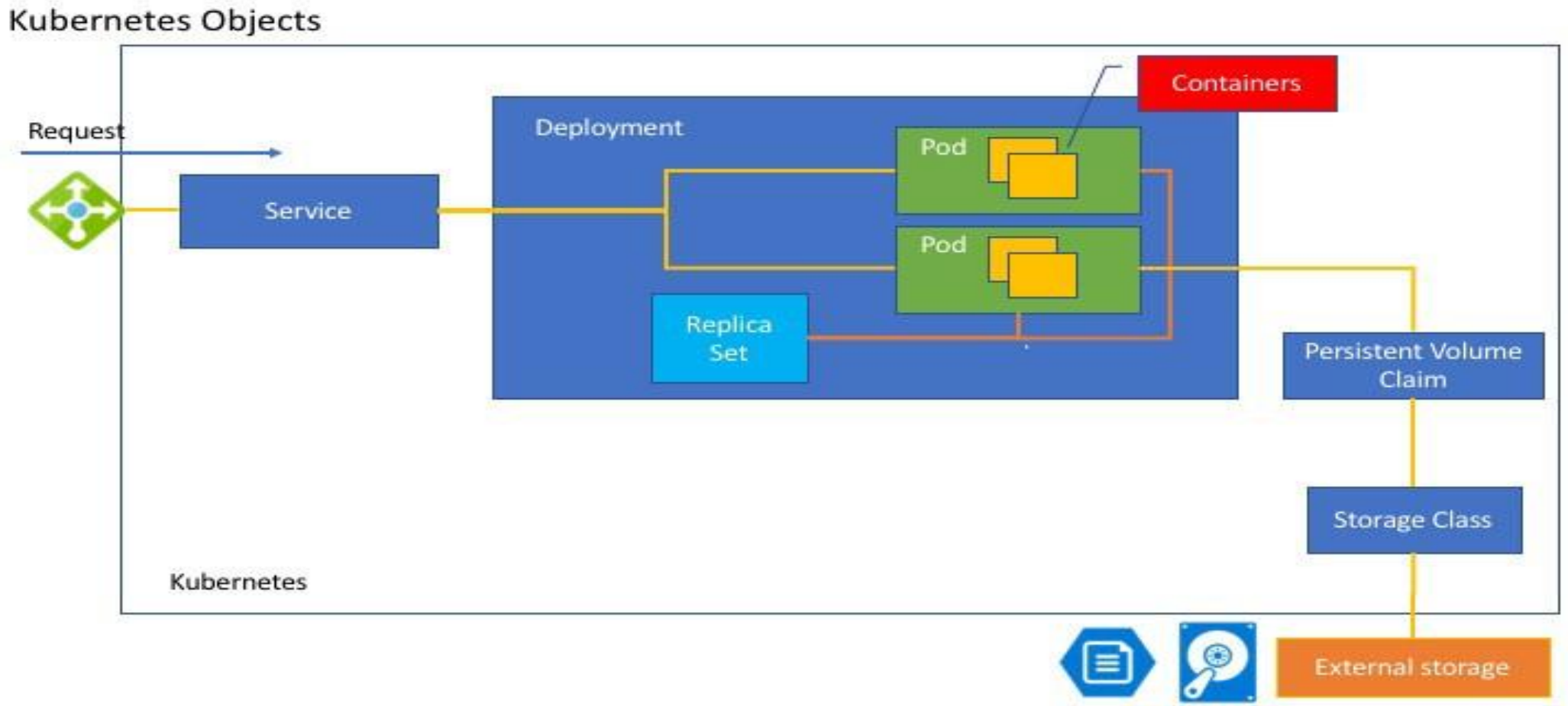
# Agenda

- Enterprise App Modernization
- Kubernetes Deep Dive

# Kubernetes Architecture



# Kubernetes Objects



# POD

```
apiVersion: v1                    1
kind: Pod                         2
metadata:
  name: kubia-manual              3
spec:
  containers:
  - image: luksa/kubia            4
    name: kubia                   5
    ports:
    - containerPort: 8080         6
      protocol: TCP
```

*1* Descriptor conforms to version v1 of Kubernetes API

*2* You're describing a pod.

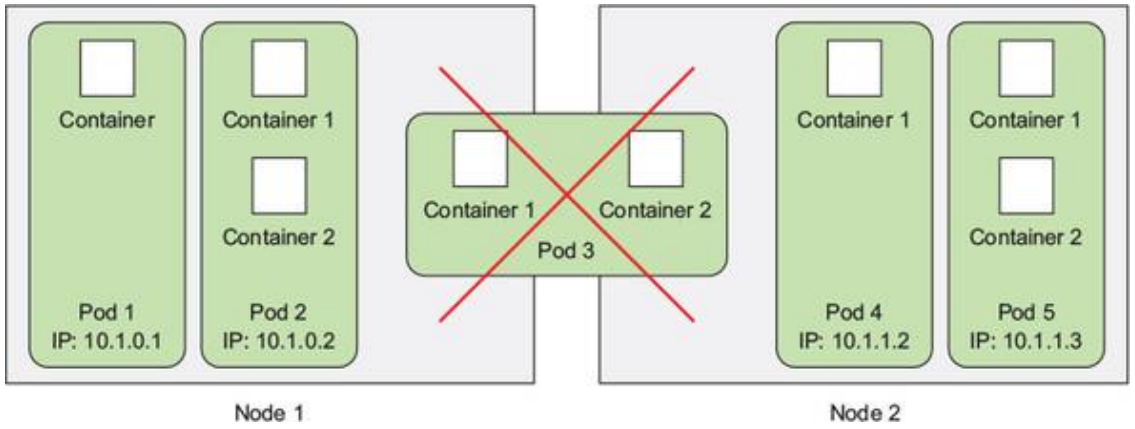
*3* The name of the pod

*4* Container image to create the container from

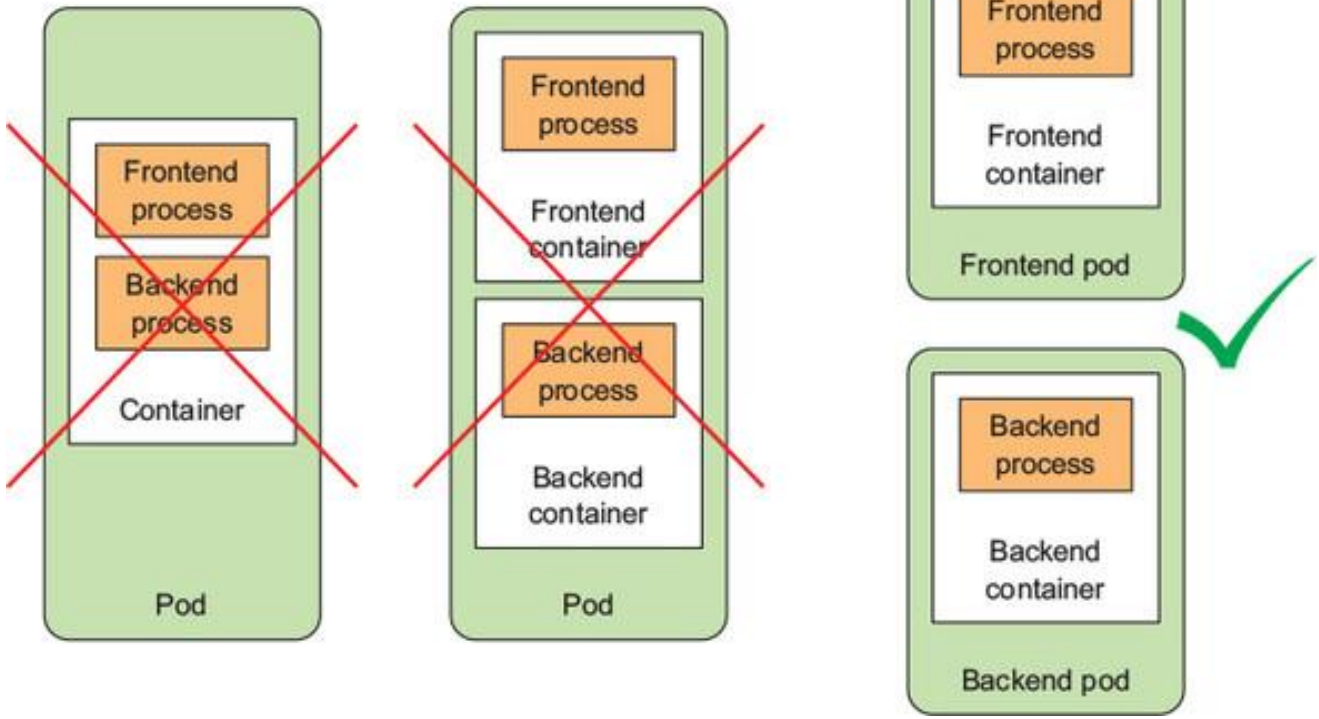
*5* Name of the container

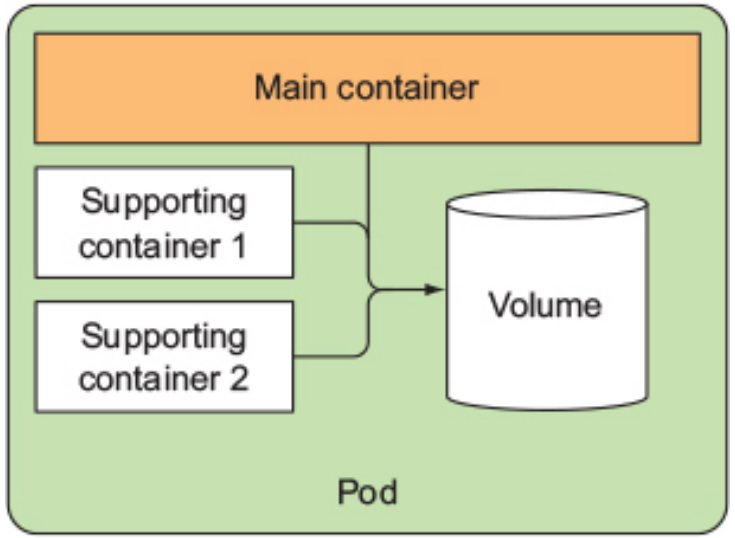
*6* The port the app is listening on

# How Pods are deployed

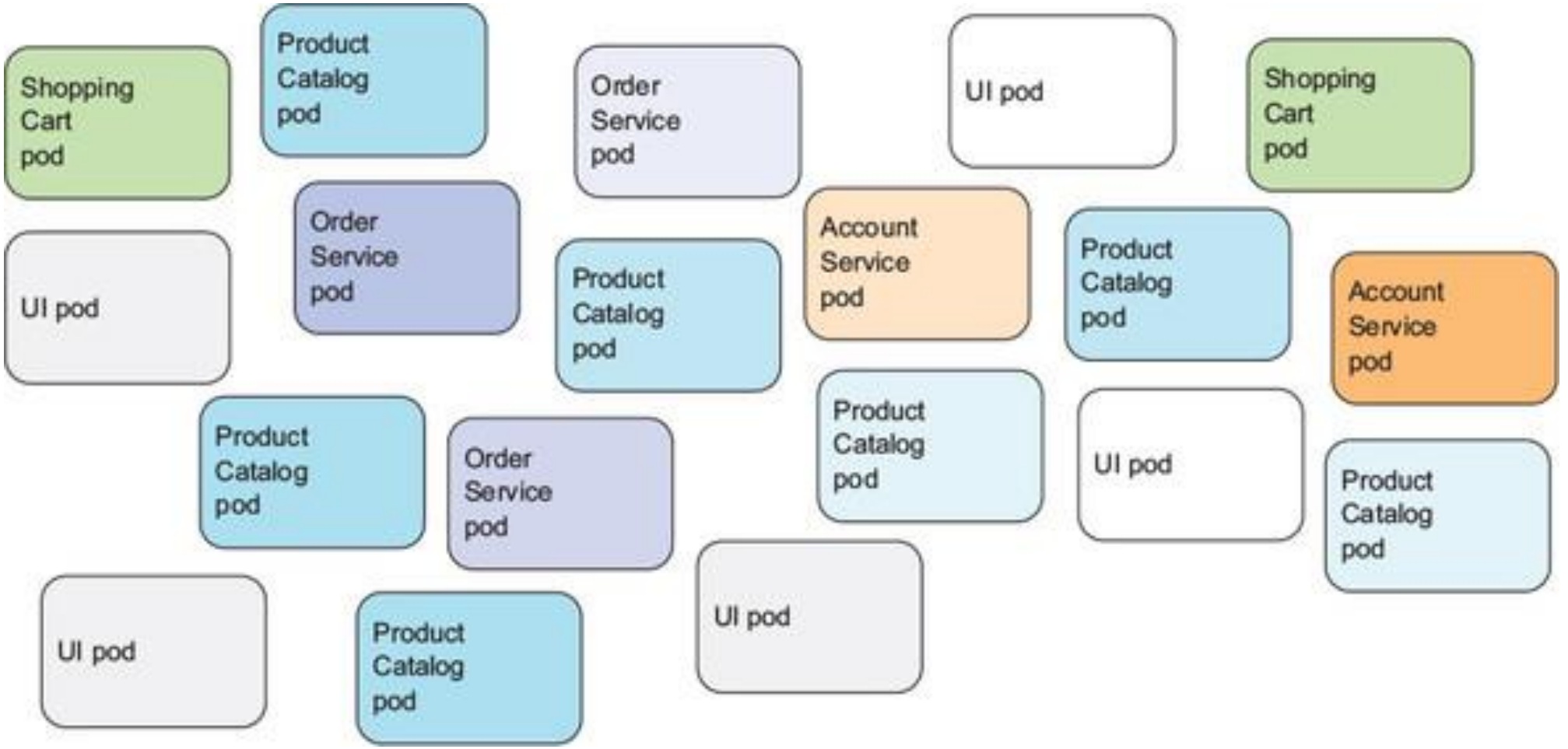



How to decide whether certain containers should be grouped together in a pod or not

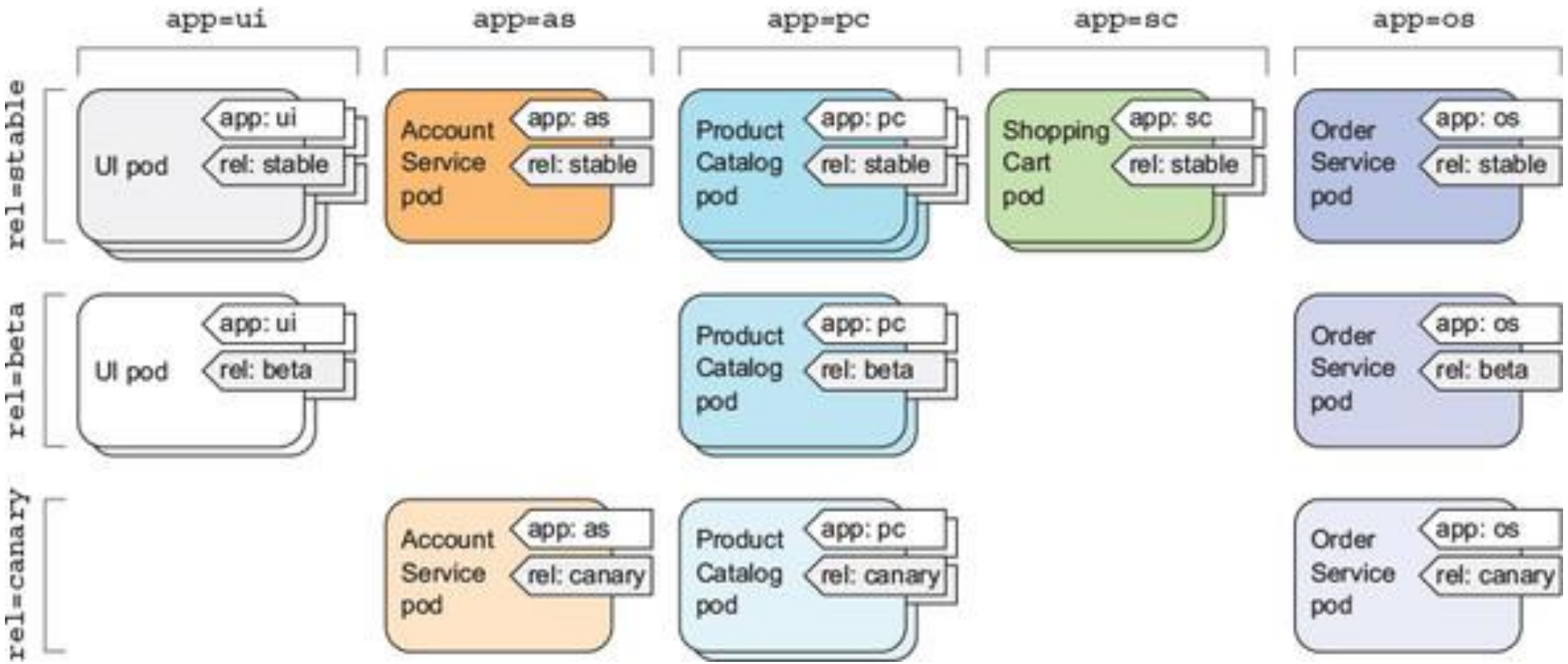


# Pods


Shopping Cart pod
Product Catalog pod
Order Service pod
UI pod
Shopping Cart pod
Order Service pod
Account Service pod
Product Catalog pod
Account Service pod
UI pod
Product Catalog pod
Product Catalog pod
Order Service pod
Product Catalog pod
UI pod
Product Catalog pod
UI pod
Product Catalog pod
UI pod

# Labels to Pod



# Add label to Pod

Labels and label selectors should be used to organize pods and easily perform operations on multiple pods at once

```
metadata:
  name: kubia-manual-v2
  labels:
    creation_method: manual          1
    env: prod                        1
```

A label selector can select resources based on whether the resource

- Contains (or doesn't contain) a label with a certain key
- Contains a label with a certain key and value
- Contains a label with a certain key, but with a value not equal to the one you specify

```
$ kubectl get po --show-labels

$ kubectl get po -l 'env'

$ kubectl get po -l creation_method=manual
```

node labels and selectors to schedule pods only to nodes that have certain features.

# Namespace

When creating pods, namespaces are used

**Deployments / Services** that need to communicate to one another may be created in the same namespace, if no communication is needed, it's recommended to use separate namespaces

A namespace is a strict isolation that occurs on Linux kernel level

- Names need to be unique within a namespace, but the same name may exist in multiple name spaces

- Every **kubectl** request uses namespaces to ensure that resources are strictly separated, and names don't have to be unique

# ReplicaSet

```
apiVersion: apps/v1beta2          1
kind: ReplicaSet                  1
metadata:
  name: kubia
spec:
  replicas: 3
  selector:
    matchLabels:                  2
      app: kubia                  2
  template:                       3
    metadata:                     3
      labels:                     3
        app: kubia                3
    spec:                         3
      containers:                 3
      - name: kubia               3
        image: luksa/kubia        3
```

*1* This manifest defines a ReplicaSet (RS)
- apps API group and version v1beta2.
The name of this ReplicationSet

The desired number of pod instances

*2* You're using the simpler matchLabels selector here

*3* The pod template for creating new pods

**expressive label selectors**

```
selector:
  matchExpressions:
    - key: app                    1
      operator: In                2
      values:                     2
        - kubia                   2
```

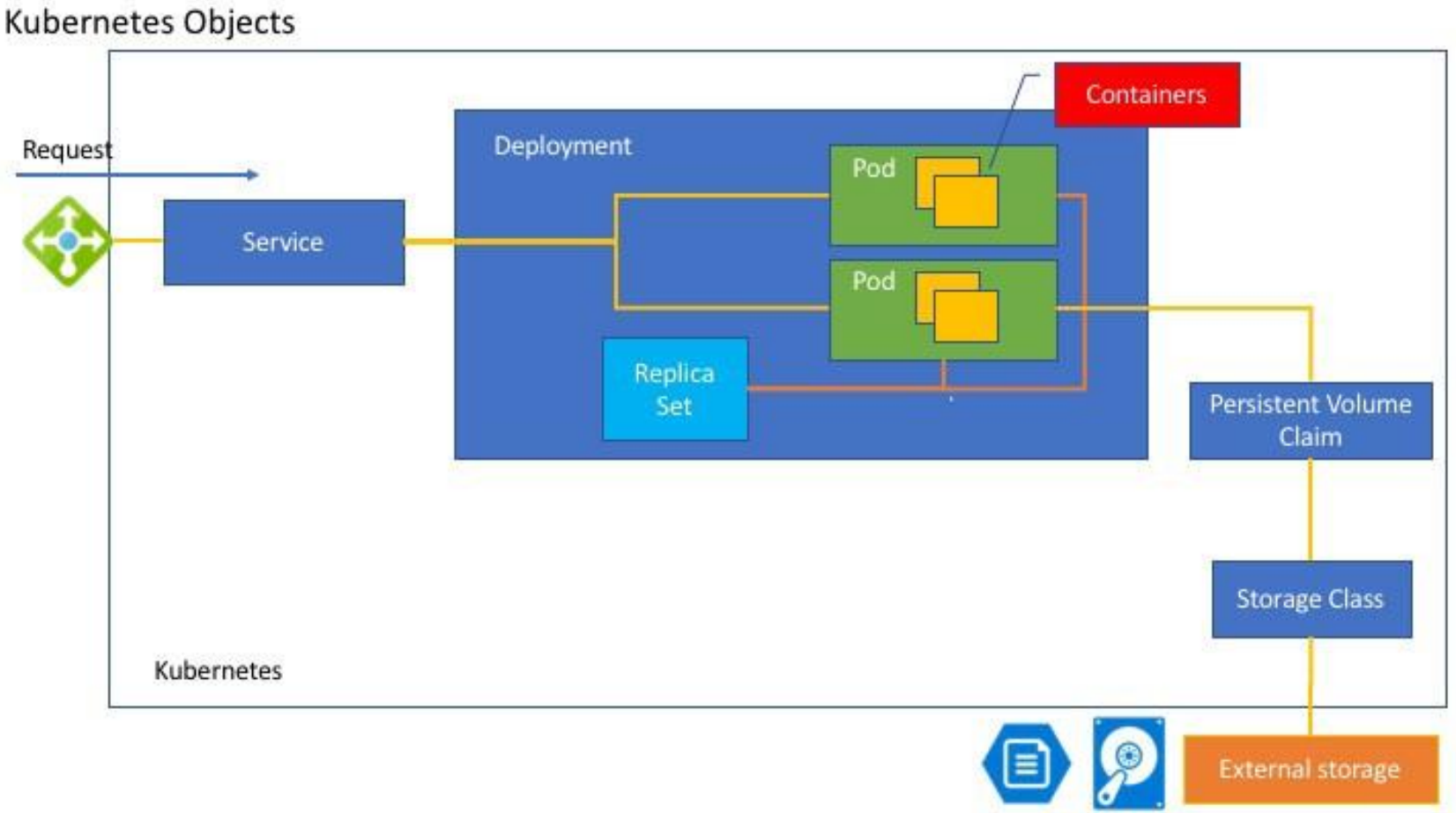
*1* This selector requires the pod to contain a label with the "app" key.
*2* The label's value must be "kubia".

# Liveness and Readiness Probes

The kubelet uses liveness probes to know when to restart a Container. For example, liveness probes could catch a deadlock, where an application is running, but unable to make progress. Restarting a Container in such a state can help to make the application more available despite bugs.

The kubelet uses readiness probes to know when a Container is ready to start accepting traffic. A Pod is considered ready when all of its Containers are ready. One use of this signal is to control which Pods are used as backends for Services. When a Pod is not ready, it is removed from Service load balancers.

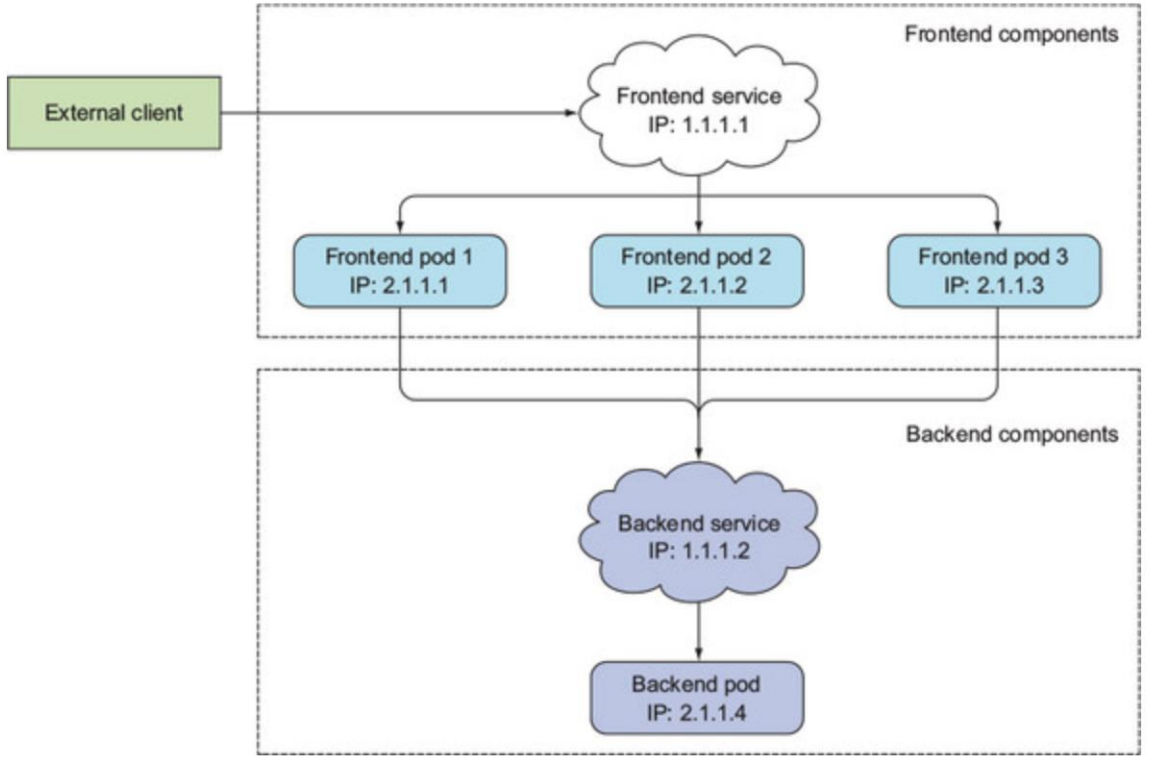
# Kubernetes Objects



# Service

Kubernetes Service is a resource you create to make a single, constant point of entry to a group of pods providing the same service

Service : IP address and port till the service exists



# Services

Different service types determine how services are accessed

- **ClusterIP** is the default and provides internal access only
- **NodePort** assigns a random port ID and exposes it on the nodes that run the service
- **LoadBalancer** is available in public cloud. May be used in private cloud, if Kubernetes provides a plugin for that cloud type
- **externalName** exposes the service using a name

- To access a cluster IP service locally (without exposing it), you may also use **kubectl proxy**
- After starting kubectl proxy, the service is accessible on the host where **kubectl proxy** was started

# Ingress

- Ingress offers service access at a specific URL
- This is useful to make sure services can be accessed in a predicted way, instead of using a random high port
- Ingress is using a proxy in the backend to provide access to the deployments

- Ingress controller must be deployed for your platform

# Persistent Volume and Persistent Volume Claim

- A pod typically is a couple of containers with one or more volumes attached
- Volumes can be mounted in a specific location in the container
- Different types of volumes are available, according to storage needs
- Volumes typically are created using YAML files while creating the VMs

**Persistent Volume** is a storage abstraction that is used to store persistent data

– Different types (NFS, iSCSI, CephFS and many more) are available

**Persistent Volume Claim**

– PVC talks to the available backend storage provider and dynamically uses volumes that are available on that storage type

Using claims with Persistent Volumes creates portable Kubernetes storage that allow you to use volumes, regardless of the specific storage provider

# VolumeTypes

Many volume types are supported

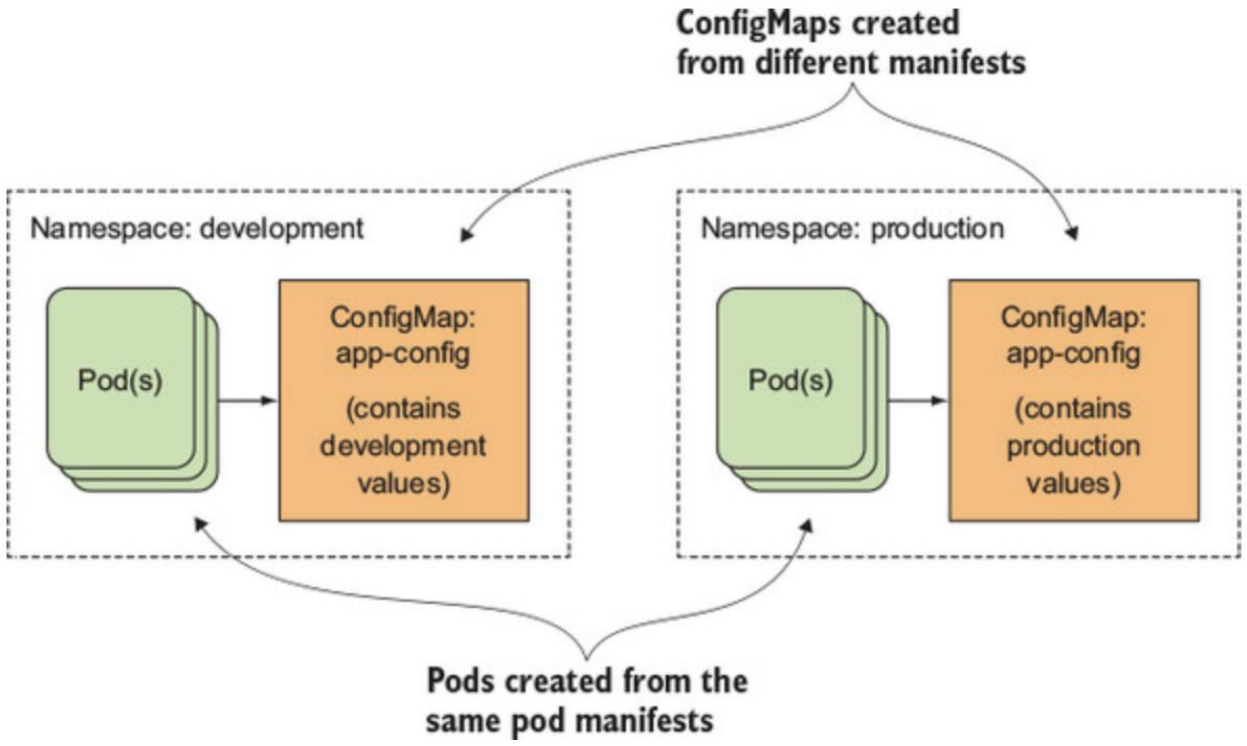
- emptyDir
- azureDisk
- cephfs
- awsElasticBlockStore
- fc: fibrechannel
- gcePersistentDisk: Google cloud
- iscsi
- nfs
- rbd
- gitrepo
- hostPath: connects to host in minikube environment

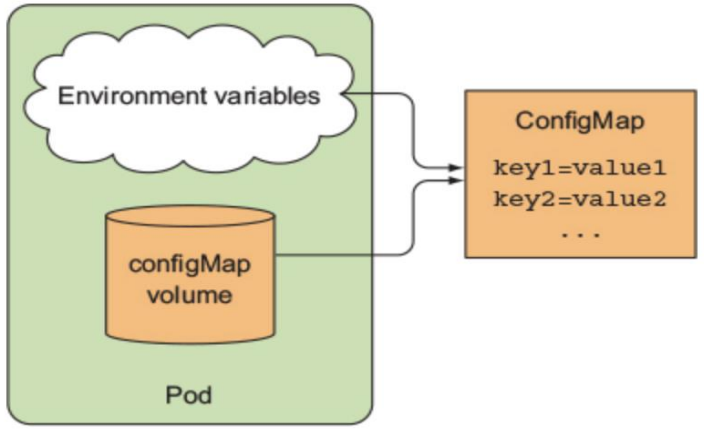
And more: choose what fits your specific needs

# ConfigMap

- Separating configuration options
- Map containing key/value pairs with the values

ranging from short literals to full config files



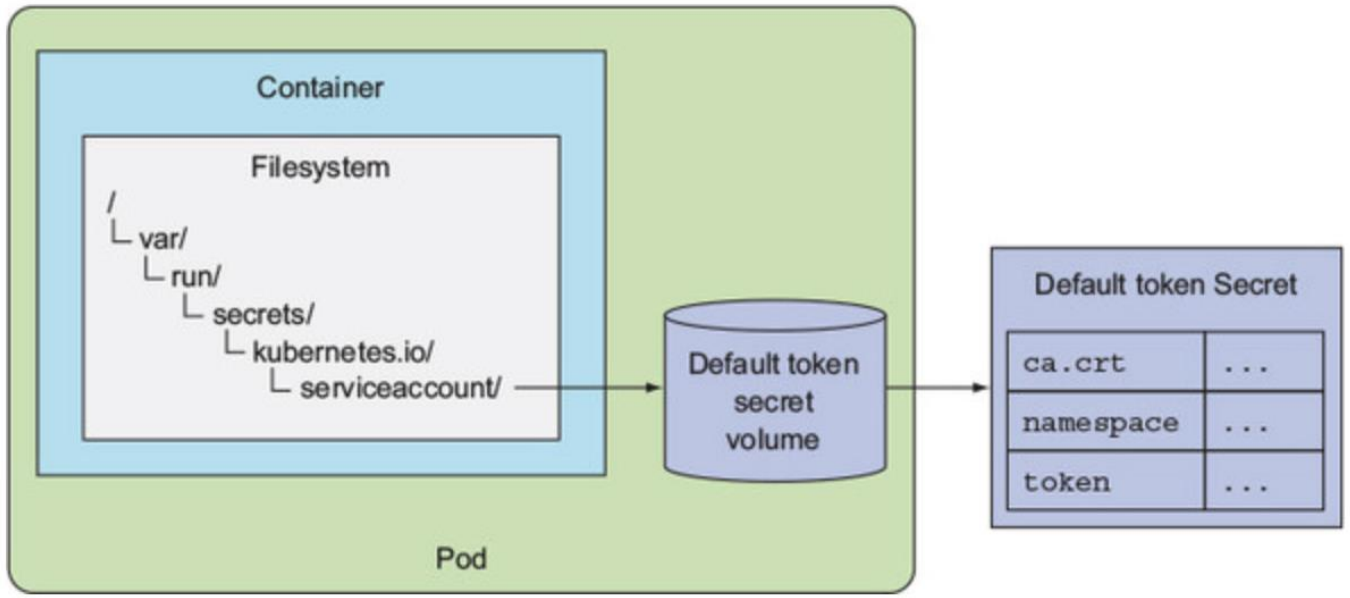

```
apiVersion: v1
kind: Pod
metadata:
  name: fortune-env-from-configmap
spec:
  containers:
  - image: luksa/fortune:env
    env:                                  1
    - name: INTERVAL                      1
      valueFrom:                          2
        configMapKeyRef:                  2
          name: fortune-config            3
          key: sleep-interval             4
...
```

# Secrets

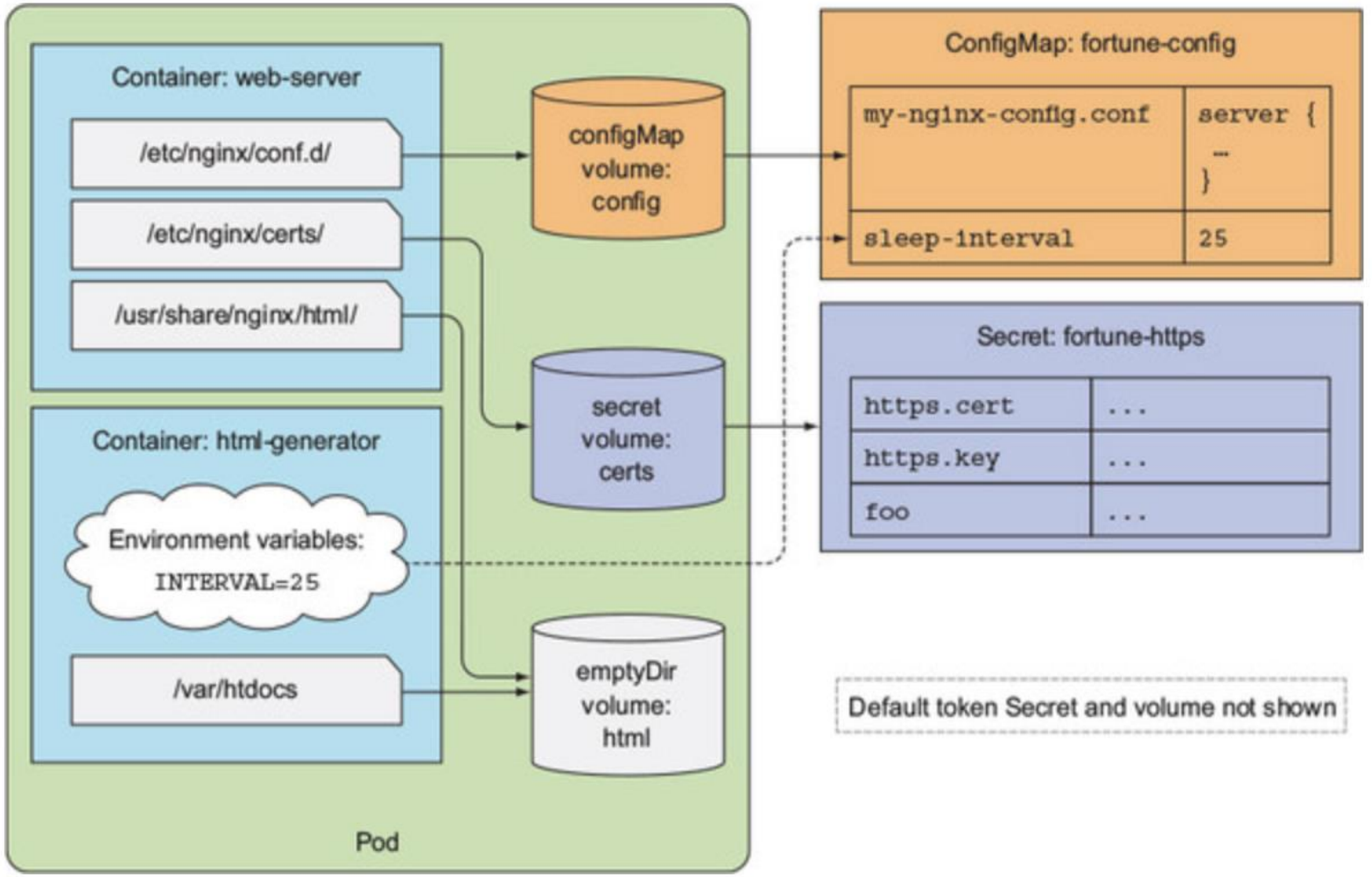
Secrets are much like ConfigMaps—they're also maps that hold key-value pairs

Used in the same way as a ConfigMap

- Pass Secret entries to the container as environment variables
- Expose Secret entries as files in a volume



# Config Map and Secrets : Combined

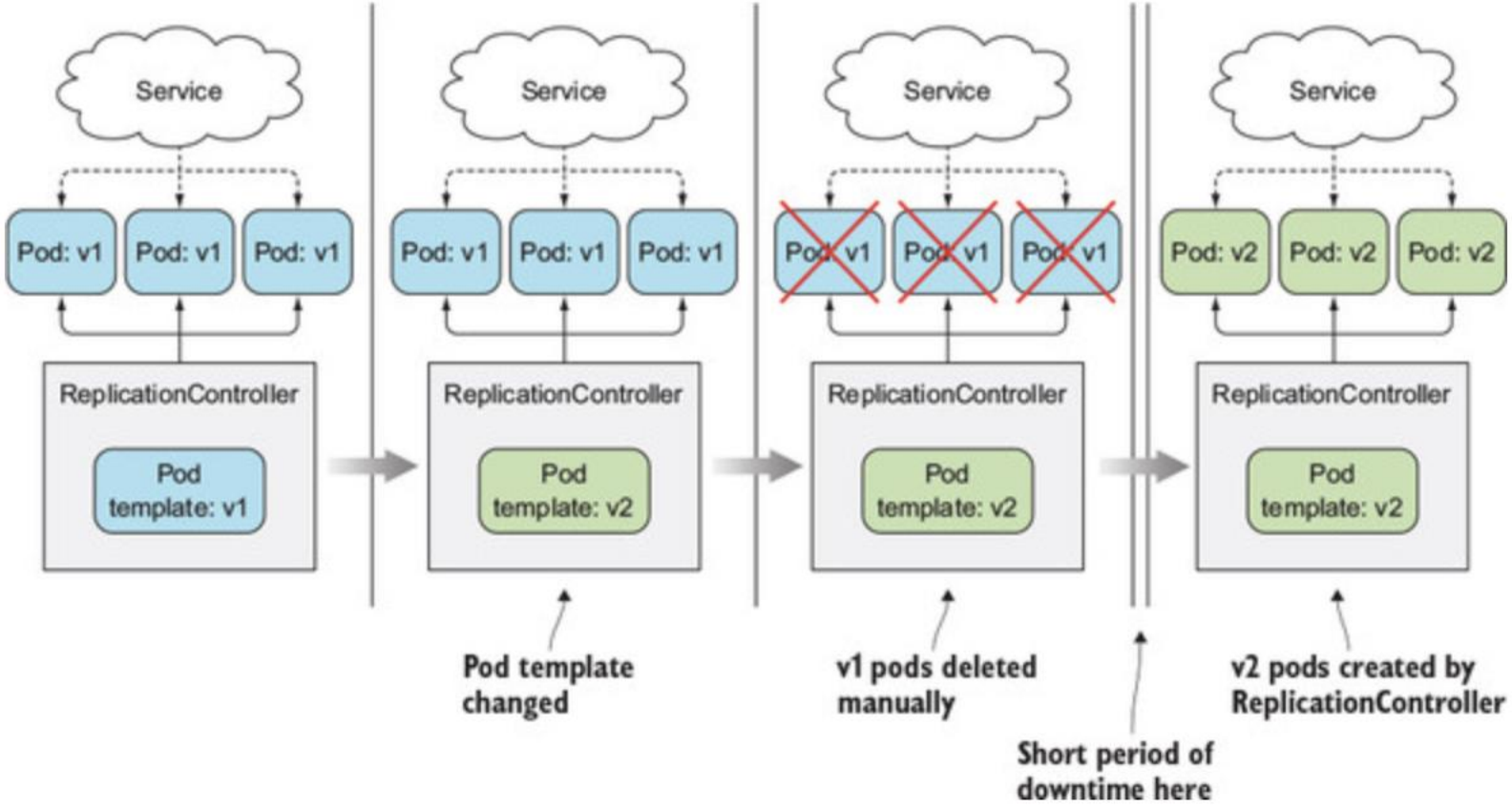


## Updating applications declaratively

- Replacing pods with newer versions
- Updating managed pods
- Updating pods declaratively using Deployment resources
- Performing rolling updates
- Automatically blocking rollouts of bad versions
- Controlling the rate of the rollout
- Reverting pods to a previous version

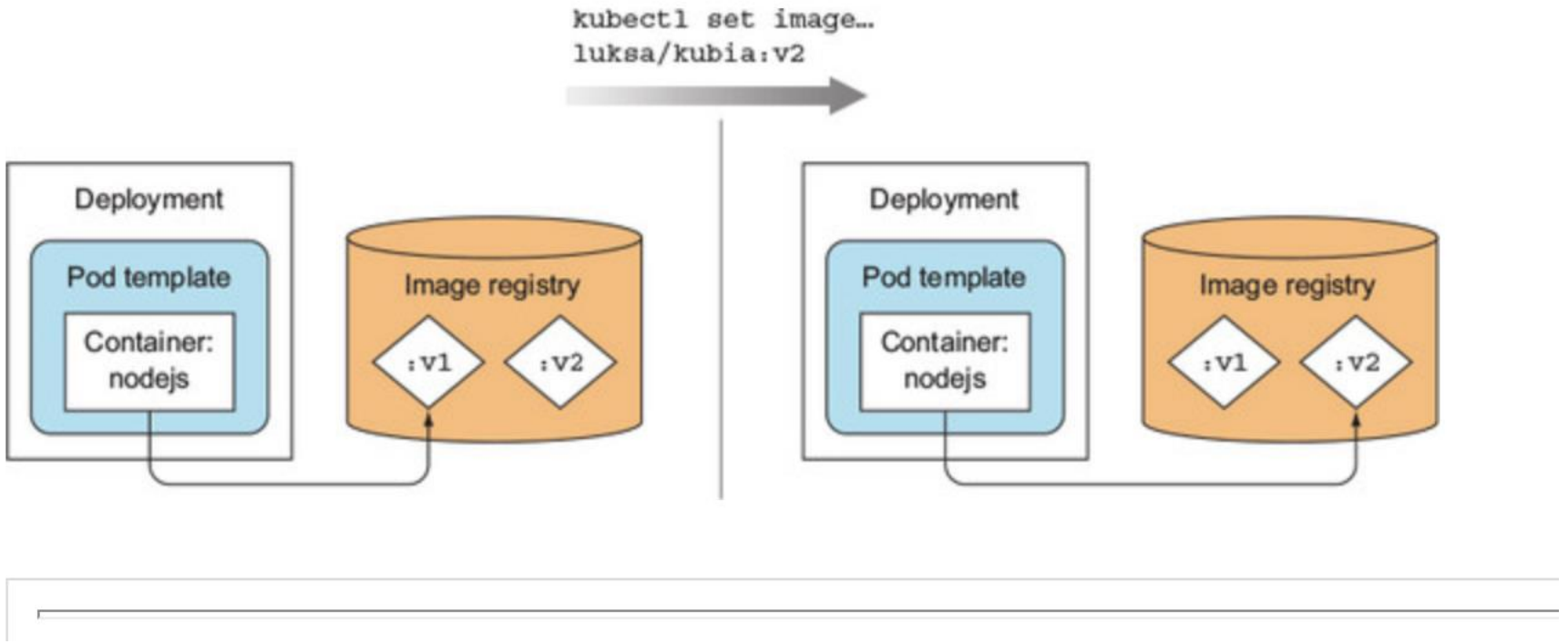
# Deployment

**Deleting old pods and replacing them with new ones**



# Deployment

**Deployment's pod template to point to a new image**



# Deployment : Rolling update