ackstorm

Kubernetes Training

2019 / Barcelona

kubernetes

kubernetes

Kubernetes, or k8s, is an open-source platform for managing containerized workloads and services. Kubernetes 1.0 was released on July 21, 2015; shortly thereafter Google partnered with the Linux Foundation to create the Cloud Native Computing Foundation (CNCF), and donated Kubernetes as a seed technology to the organization. It can run on most cloud providers, as well as on bare metal.

Kubernetes provides a container-centric management environment. It can orchestrate computing, networking, and storage infrastructure on behalf of user workloads, which will result in much simpler application development, deployment and maintenance.

Kubernetes Components

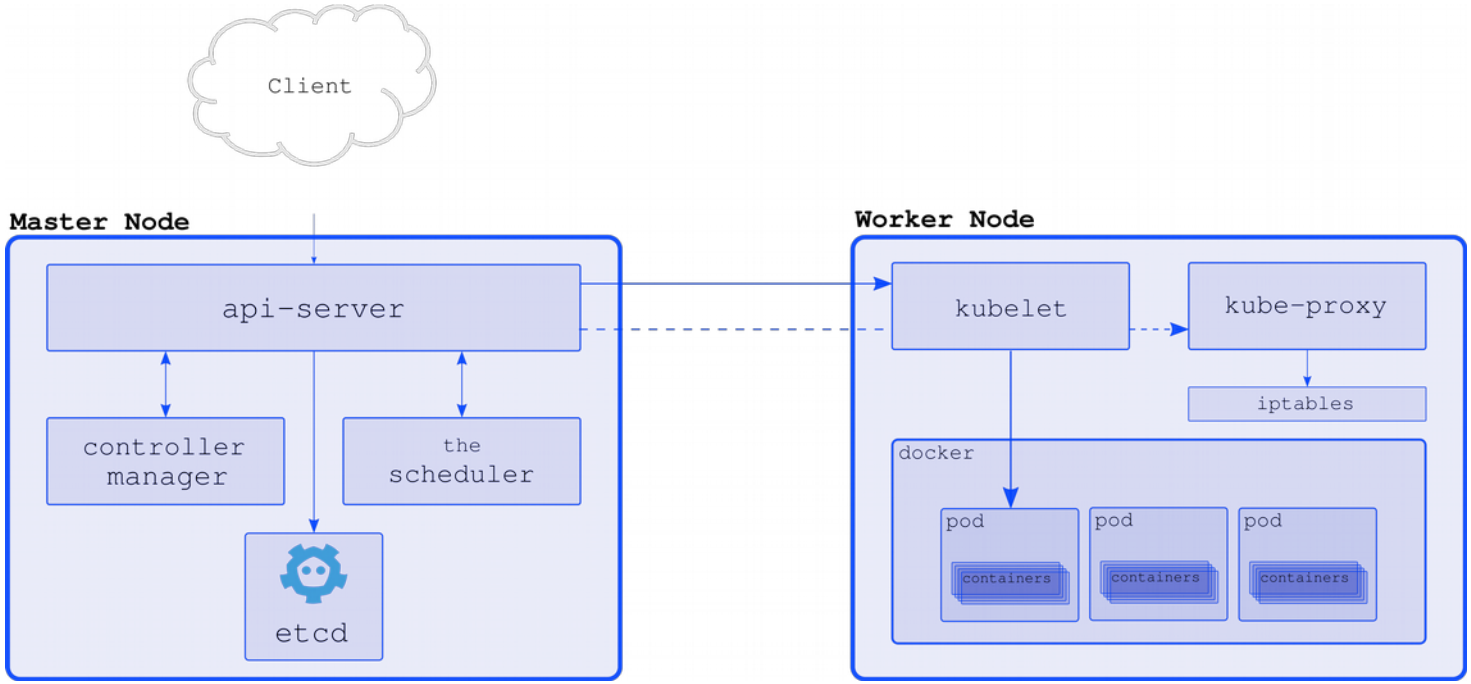
Kubernetes Objects are persistent entities in the system. Kubernetes uses these entities, as an abstraction, to represent the state of the cluster. Some of the basic kubernetes objects are:

- Pod
- Service
- Volume
- Namespace

In addition, Kubernetes contains a number of higher-level abstractions called Controllers. Controllers build upon the basic objects, and provide additional functionality and convenience features. Some of these controllers are:

- ReplicaSet
- Deployment
- StatefulSet
- DaemonSet
- Job/Cronjob

Kubernetes Architecture



Kubernetes Objects

Kubernetes Objects are persistent entities in the system. Kubernetes uses these entities, as an abstraction, to represent the state of the cluster. Some of the basic kubernetes objects are:

- Pod
- Service
- Volume
- Namespace

In addition, Kubernetes contains a number of higher-level abstractions called Controllers. Controllers build upon the basic objects, and provide additional functionality and convenience features. Some of these controllers are:

- ReplicaSet
- Deployment
- StatefulSet
- DaemonSet
- Job/Cronjob

Pod

A pod is a group of one or more containers, with shared storage and network, and specifications for how to run the containers.

it contains one or more application containers which are tightly coupled; being executed on the same physical or virtual machine.

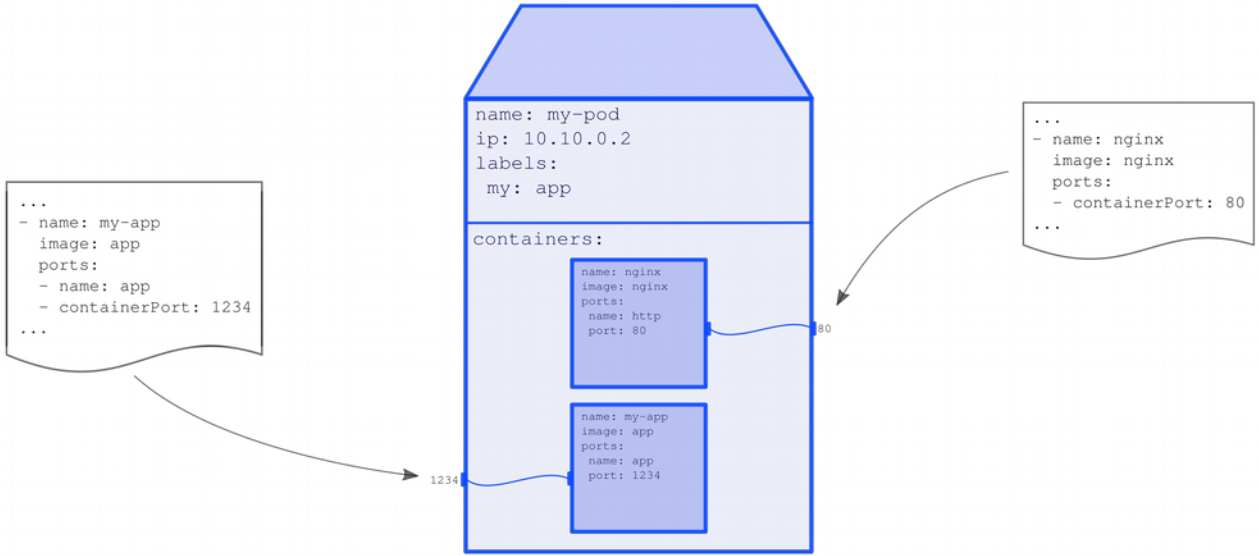
Containers within a pod share IP an address and port space, and can communicate with each other via localhost.

Pods are not intended to be treated as durable entities. They won't survive scheduling failure, node failure, or other evictions, such as due to lack of resources, or node maintenance.

Pod – Ports

Container ports get directly mapped on the same port of pod, independently if these have been declared in the yaml file.

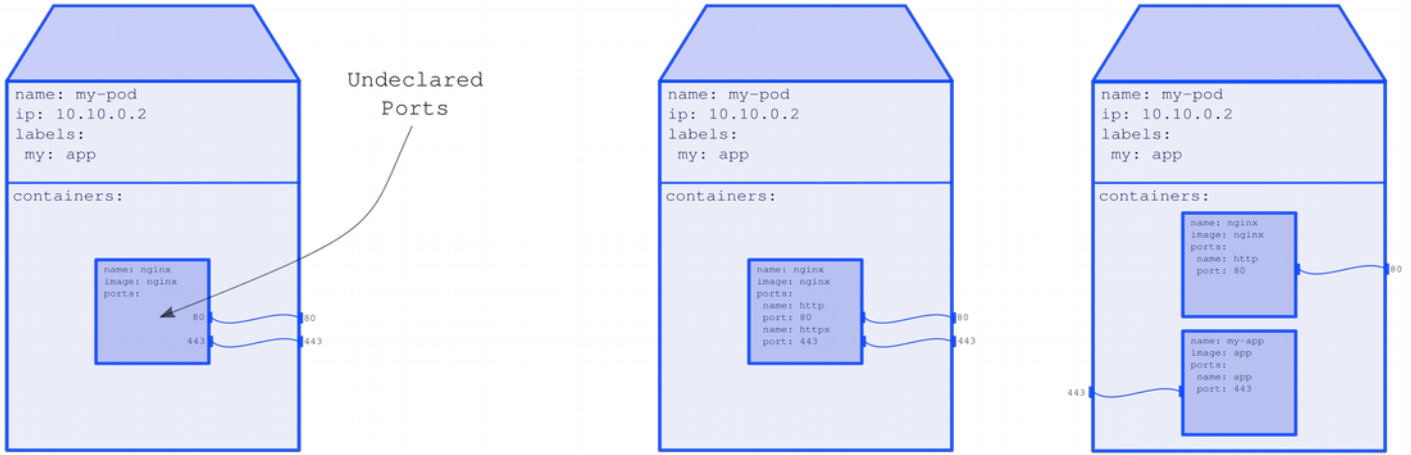
Not declaring the port in the yaml file will avoid Kubernetes properly managing the pod. In other words, Kubernetes will treat the pod as it is declared in the yaml file, not as it behaves by default.



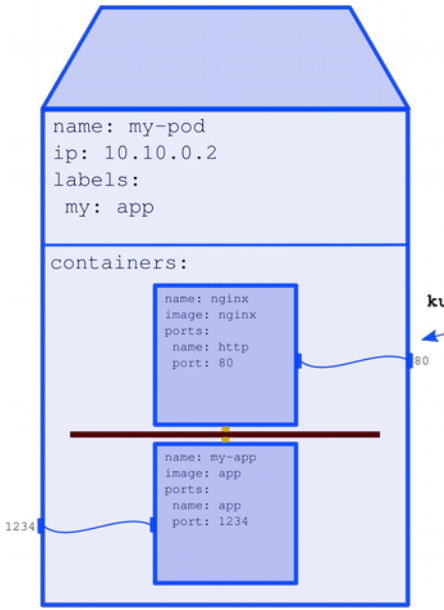
Pod – Ports

For example all these three pods are serving on port 80 and 443, even though they have been configured slightly different.

If we would expose the first pod (create a service), we would need to explicitly declare to which port to forward the traffic, while for the second and third pods, Kubernetes will retrieve this information from the metadata server (etcd)



Pod - yaml

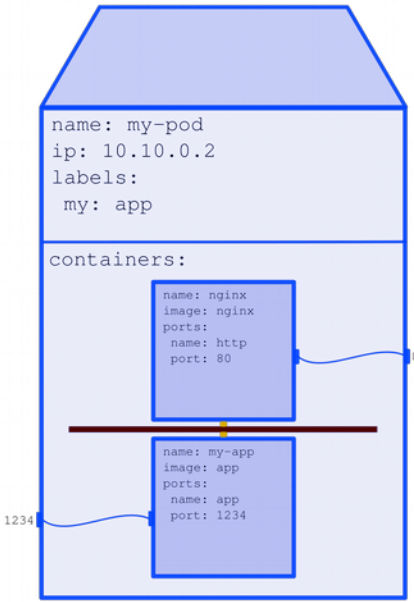


kubectl create -f pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    my: app
spec:
  volumes:
  - name: app-nginx
    emptyDir: {}
  containers:
    - name: app
      image: app
      volumeMounts:
      - name: app-nginx
        mountPath: /data/app
    - name: nginx
      image: nginx
      ports:
      - containerPort: 80
      volumeMounts:
      - name: app-nginx
        mountPath: ../nginx/html
```

The metadata of the pod.
(name, labels and annotations)

Pod – yaml



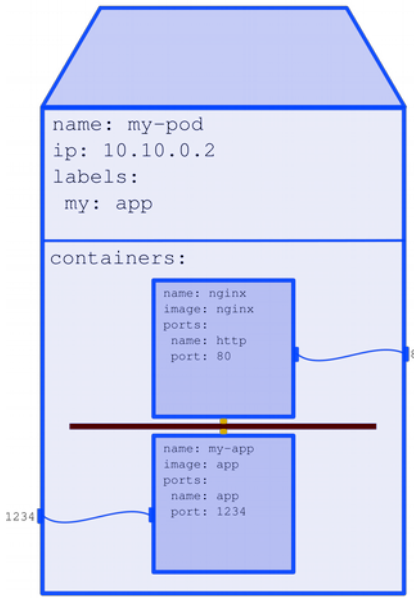
kubectl create -f pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    my: app
spec:
  volumes:
  - name: app-nginx
    emptyDir: {}
  containers:
    - name: app
      image: app
      volumeMounts:
      - name: app-nginx
        mountPath: /data/app
    - name: nginx
      image: nginx
      ports:
      - containerPort: 80
      volumeMounts:
      - name: app-nginx
        mountPath: ../nginx/html
```

Volumes, in the pod

Volumes, in the containers

Pod – yaml



kubectl create -f pod.yaml

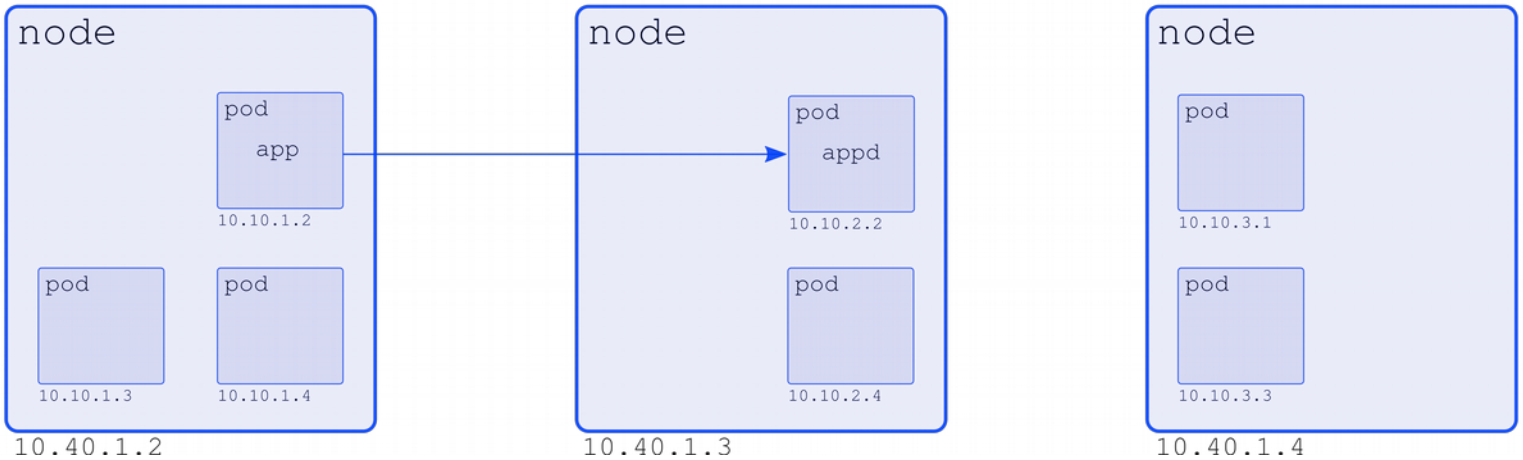
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
    my: app
spec:
  volumes:
  - name: app-nginx
    emptyDir: {}
  containers:
    - name: app
      image: app
      volumeMounts:
      - name: app-nginx
        mountPath: /data/app
    - name: nginx
      image: nginx
      ports:
      - containerPort: 80
      volumeMounts:
      - name: app-nginx
        mountPath: ../nginx/html
```

Container specifications

Pod-to-Pod Communication

In the cluster, most of the pods communicate with each other. But being a pod such a volatile object, and given the nature of Kubernetes, pod-to-pod direct communication might result in a disaster.

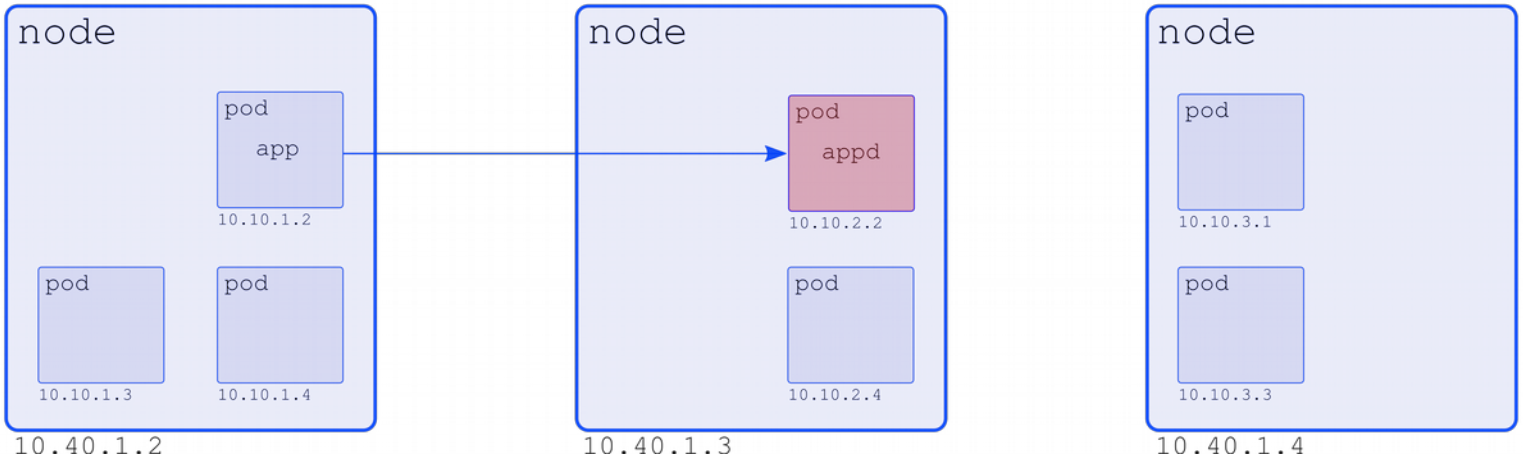
In the scenario below, pods 'app' and 'appd' are communicating. If we would have configured the pod to run always, and it suddenly would crash, Kubernetes would re-create it, with no guarantees it is going to be re-created on the same node, or is going to have the same IP.



Pod-to-Pod Communication

In the cluster, most of the pods communicate with each other. But being a pod such a volatile object, and given the nature of Kubernetes, pod-to-pod direct communication might result in a disaster.

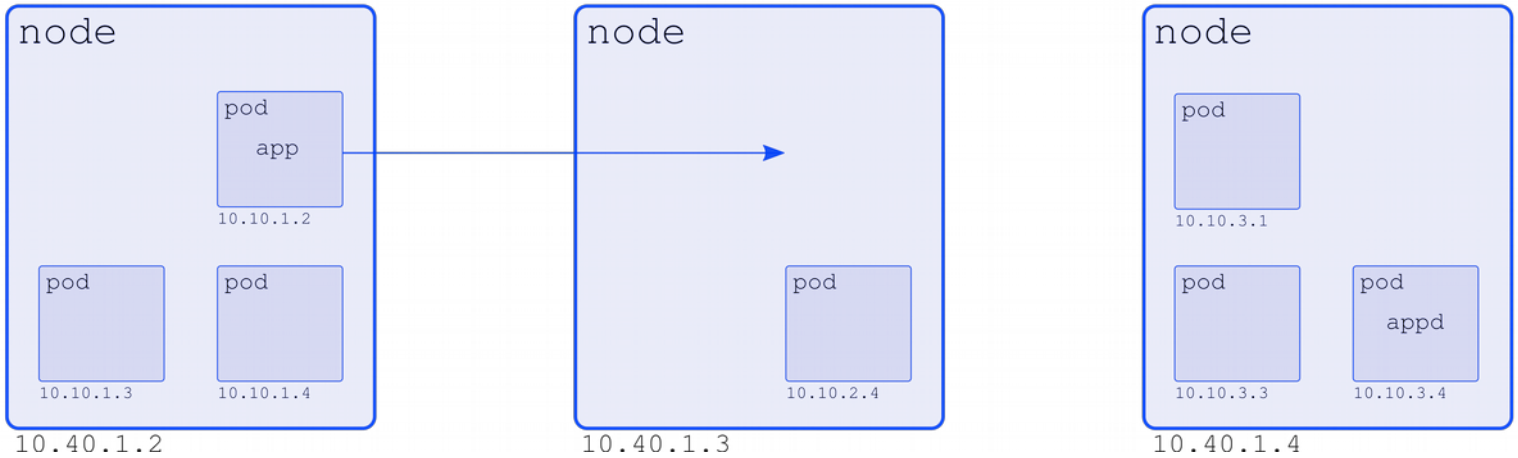
In the scenario below, pods 'app' and 'appd' are communicating. If we would have configured the pod to run always, and it suddenly would crash, Kubernetes would re-create it, with no guarantees it is going to be re-created on the same node, or is going to have the same IP.



Pod-to-Pod Communication

In the cluster, most of the pods communicate with each other. But being a pod such a volatile object, and given the nature of Kubernetes, pod-to-pod direct communication might result in a disaster.

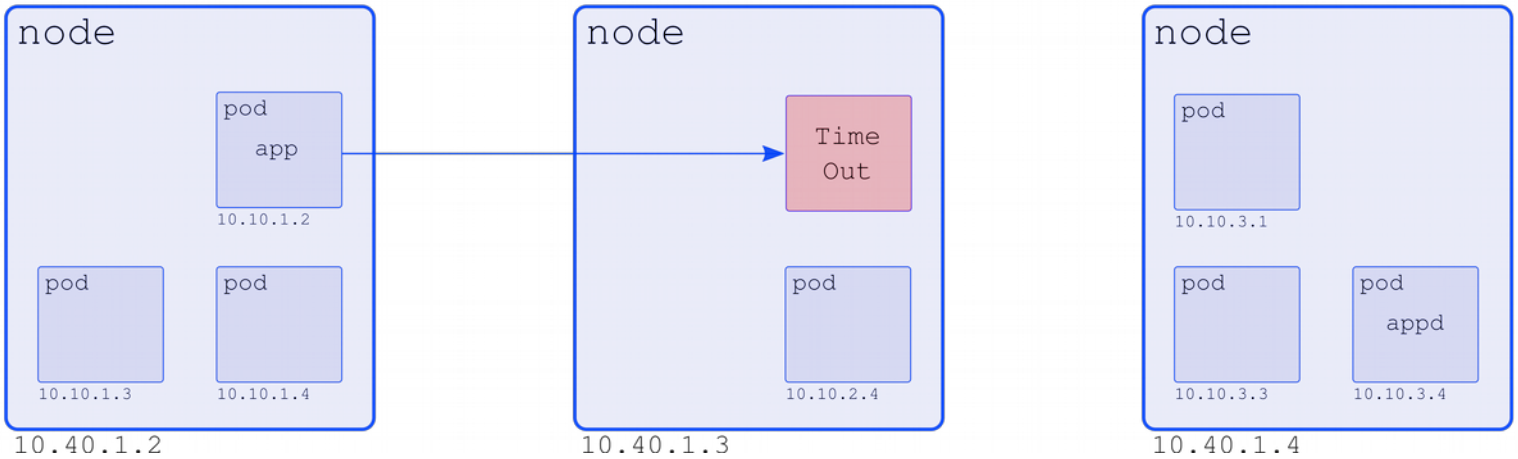
In the scenario below, pods 'app' and 'appd' are communicating. If we would have configured the pod to run always, and it suddenly would crash, Kubernetes would re-create it, with no guarantees it is going to be re-created on the same node, or is going to have the same IP.



Pod-to-Pod Communication

In the cluster, most of the pods communicate with each other. But being a pod such a volatile object, and given the nature of Kubernetes, pod-to-pod direct communication might result in a disaster.

In the scenario below, pods 'app' and 'appd' are communicating. If we would have configured the pod to run always, and it suddenly would crash, Kubernetes would re-create it, with no guarantees it is going to be re-created on the same node, or is going to have the same IP.



Namespace

Namespace definition: a class of elements in which each element has a name unique to that class, although it may be shared with elements in other classes.

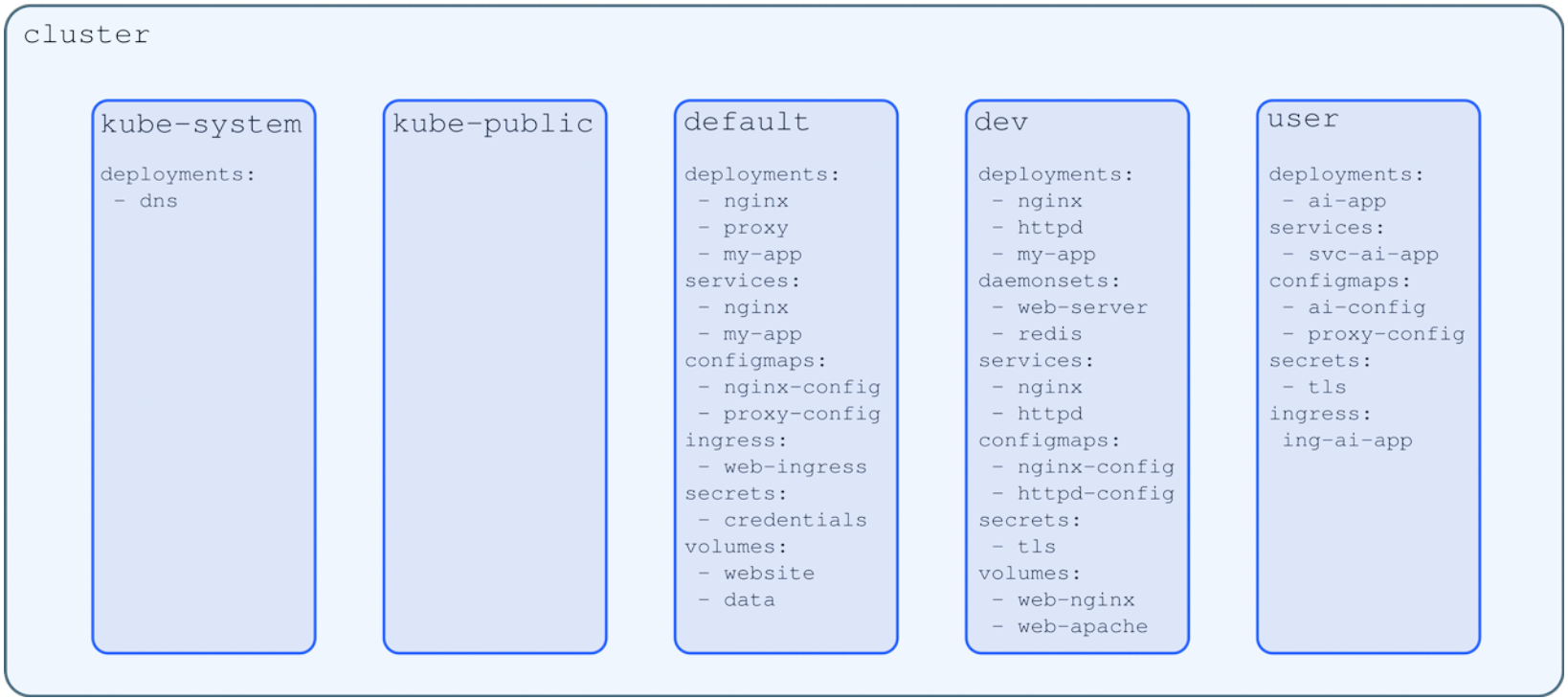
Namespaces "divide" a cluster in sub-clusters. This way different projects, teams, or customers can share a Kubernetes cluster. In fact, Namespaces are intended for use in environments with many users spread across multiple teams, or projects.

Namespaces provide a scope for names, and a mechanism to attach authorization and policy to a subsection of the cluster.

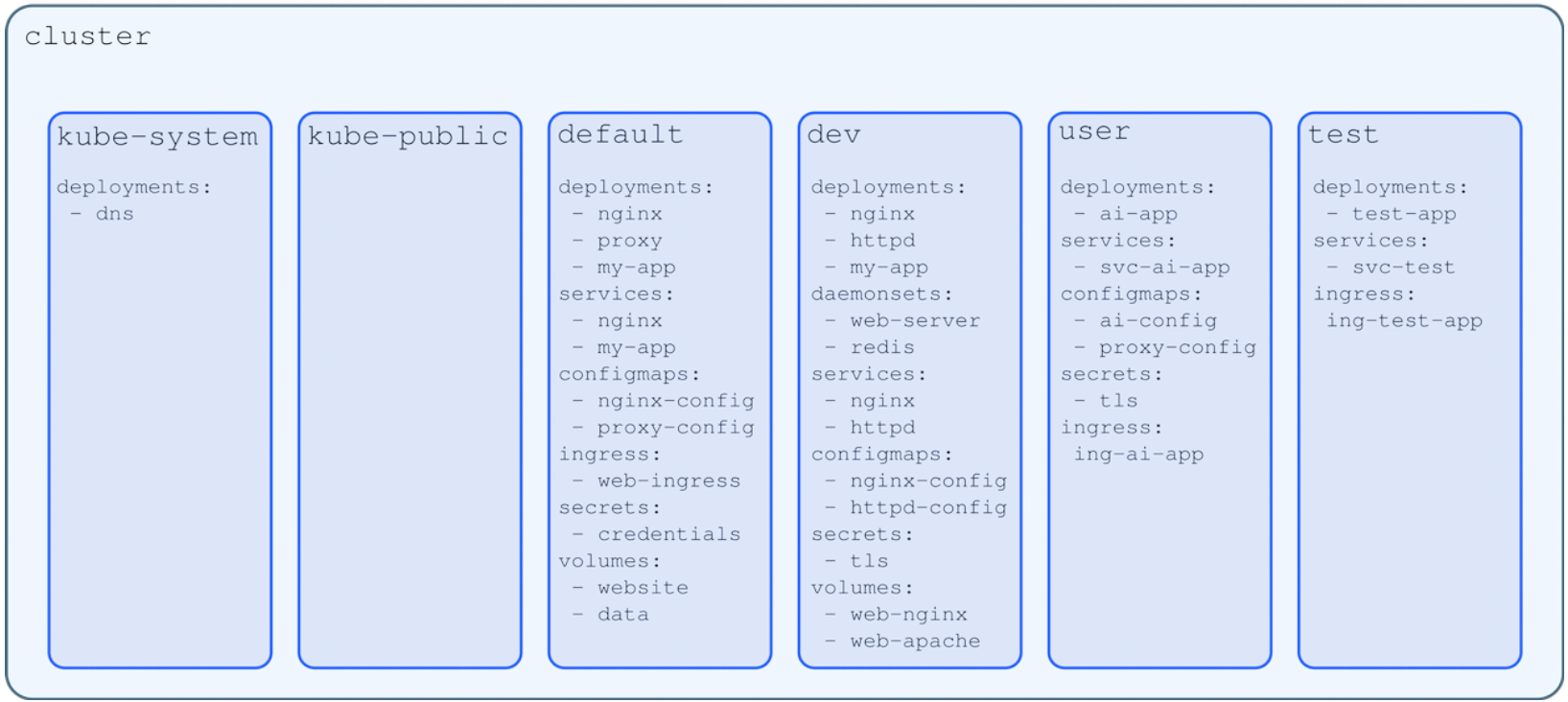
Kubernetes starts with three initial namespaces:

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

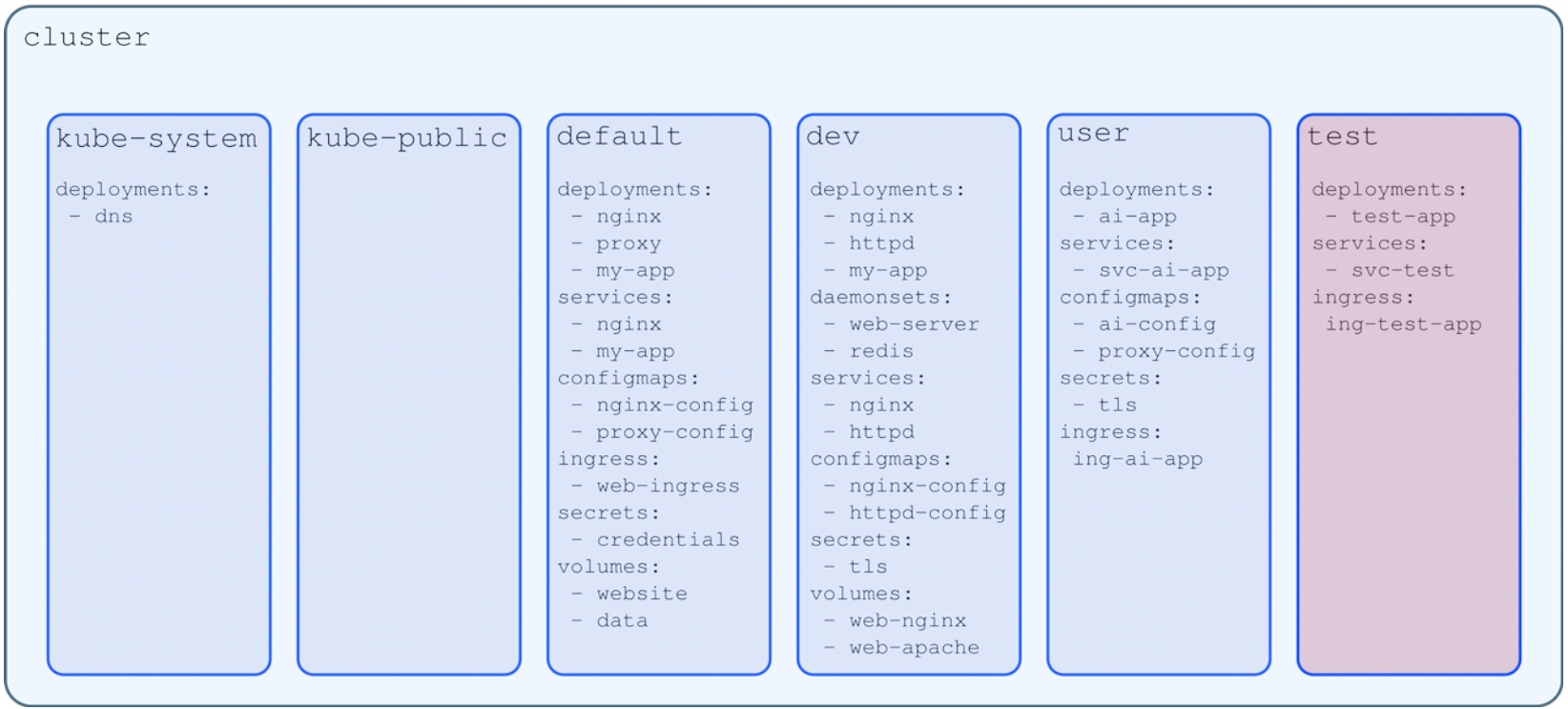
Namespace


cluster
kube-system
deployments:
- dns
kube-public
default
deployments:
- nginx
- proxy
- my-app
services:
- nginx
- my-app
configmaps:
- nginx-config
- proxy-config
ingress:
- web-ingress
secrets:
- credentials
volumes:
- website
- data
dev
deployments:
- nginx
- httpd
- my-app
daemonsets:
- web-server
- redis
services:
- nginx
- httpd
configmaps:
- nginx-config
- httpd-config
secrets:
- tls
volumes:
- web-nginx
- web-apache
user
deployments:
- ai-app
services:
- svc-ai-app
configmaps:
- ai-config
- proxy-config
secrets:
- tls
ingress:
ing-ai-app

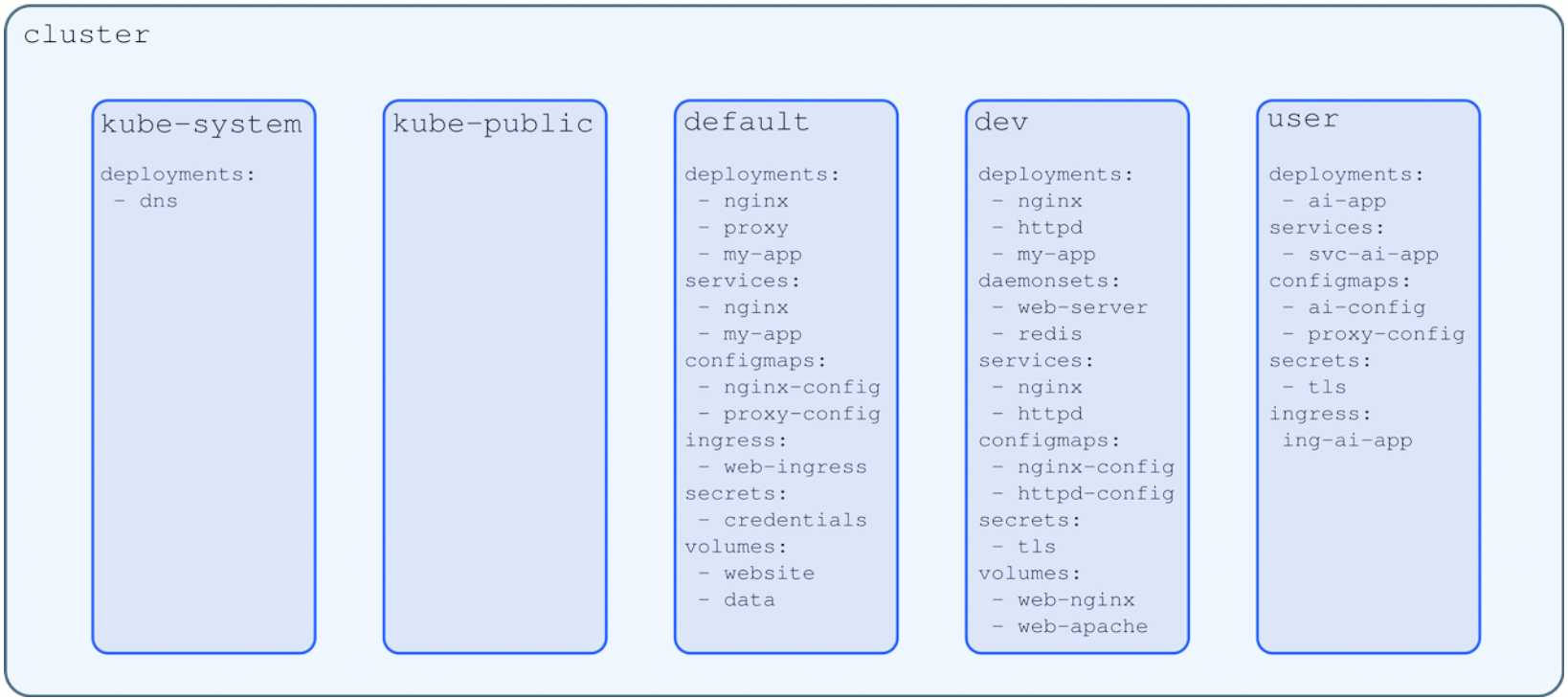
Namespace


cluster
kube-system
deployments:
- dns
kube-public
default
deployments:
- nginx
- proxy
- my-app
services:
- nginx
- my-app
configmaps:
- nginx-config
- proxy-config
ingress:
- web-ingress
secrets:
- credentials
volumes:
- website
- data
dev
deployments:
- nginx
- httpd
- my-app
daemonsets:
- web-server
- redis
services:
- nginx
- httpd
configmaps:
- nginx-config
- httpd-config
secrets:
- tls
volumes:
- web-nginx
- web-apache
user
deployments:
- ai-app
services:
- svc-ai-app
configmaps:
- ai-config
- proxy-config
secrets:
- tls
ingress:
ing-ai-app
test
deployments:
- test-app
services:
- svc-test
ingress:
ing-test-app

Namespace


cluster
kube-system
deployments:
- dns
kube-public
default
deployments:
- nginx
- proxy
- my-app
services:
- nginx
- my-app
configmaps:
- nginx-config
- proxy-config
ingress:
- web-ingress
secrets:
- credentials
volumes:
- website
- data
dev
deployments:
- nginx
- httpd
- my-app
daemonsets:
- web-server
- redis
services:
- nginx
- httpd
configmaps:
- nginx-config
- httpd-config
secrets:
- tls
volumes:
- web-nginx
- web-apache
user
deployments:
- ai-app
services:
- svc-ai-app
configmaps:
- ai-config
- proxy-config
secrets:
- tls
ingress:
ing-ai-app
test
deployments:
- test-app
services:
- svc-test
ingress:
ing-test-app

Namespace


cluster
kube-system
deployments:
- dns
kube-public
default
deployments:
- nginx
- proxy
- my-app
services:
- nginx
- my-app
configmaps:
- nginx-config
- proxy-config
ingress:
- web-ingress
secrets:
- credentials
volumes:
- website
- data
dev
deployments:
- nginx
- httpd
- my-app
daemonsets:
- web-server
- redis
services:
- nginx
- httpd
configmaps:
- nginx-config
- httpd-config
secrets:
- tls
volumes:
- web-nginx
- web-apache
user
deployments:
- ai-app
services:
- svc-ai-app
configmaps:
- ai-config
- proxy-config
secrets:
- tls
ingress:
ing-ai-app

Lab 1

```
1.- Create a namespace with your name
   $ kubectl create namespace <YOUR-NAME>
2.- Configure your context
   $kubectl config set-context $(kubectl config current-context) -n NAMESPACE)
3.- Create a pod and inspect it
   $ kubectl create -f pod.yaml
   $ kubectl describe pod nginx
4.- Identify other pod IP adresses
   $ kubectl get pods -o wide --all-namespaces | grep nginx
5.- Access the pod you created previously
   $ kubectl exec -it nginx bash
6.- Install curl in the container
   $ apt-get update
   $ apt-get install -y curl
7.- curl the address of any other nginx pod
   $ curl IP_ADDR
8.- Delete and re-create the pod couple of times, and observe the IP address change
   $ kubectl delete pod nginx
   $ kubectl create -f pod.yaml
   $ kubectl get pod -o wide
```

Labels

Labels are key/value pairs that are attached to objects, such as pods.

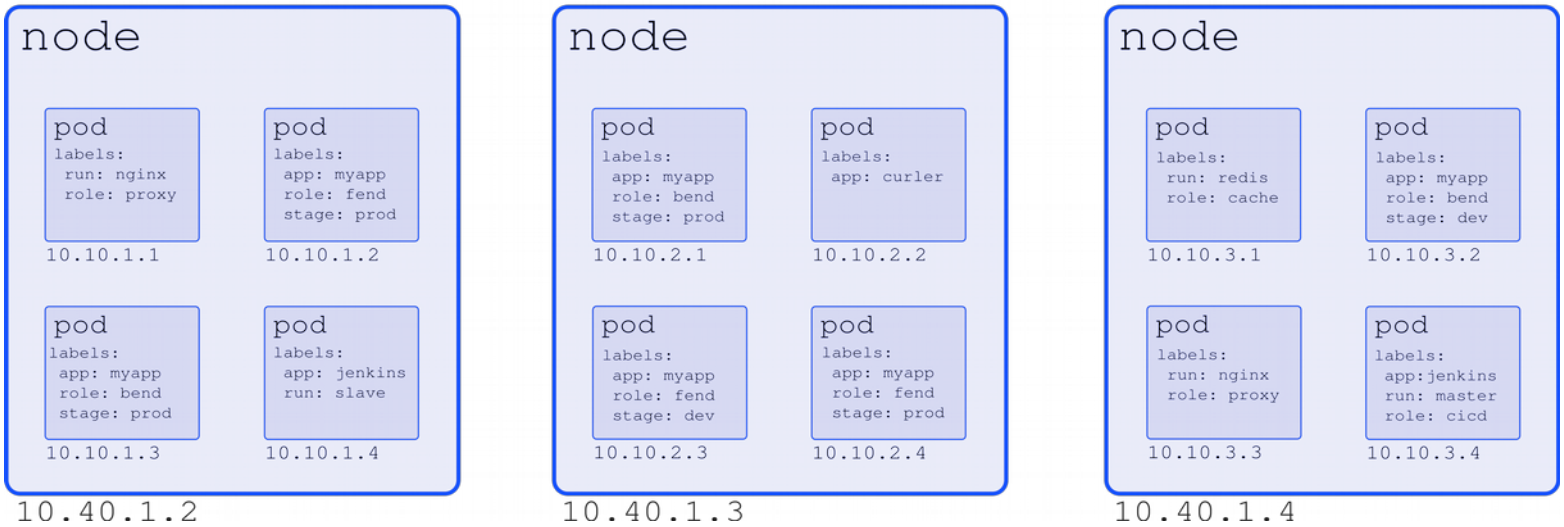
Labels are intended to be used to specify attributes of objects that are meaningful and relevant to users, but not necessarily the core system.

Labels can be used to organize and to select subsets of objects.

The user can identify a set of objects (e.g. pods) via a label selector. Label selector is the core grouping primitive in Kubernetes.

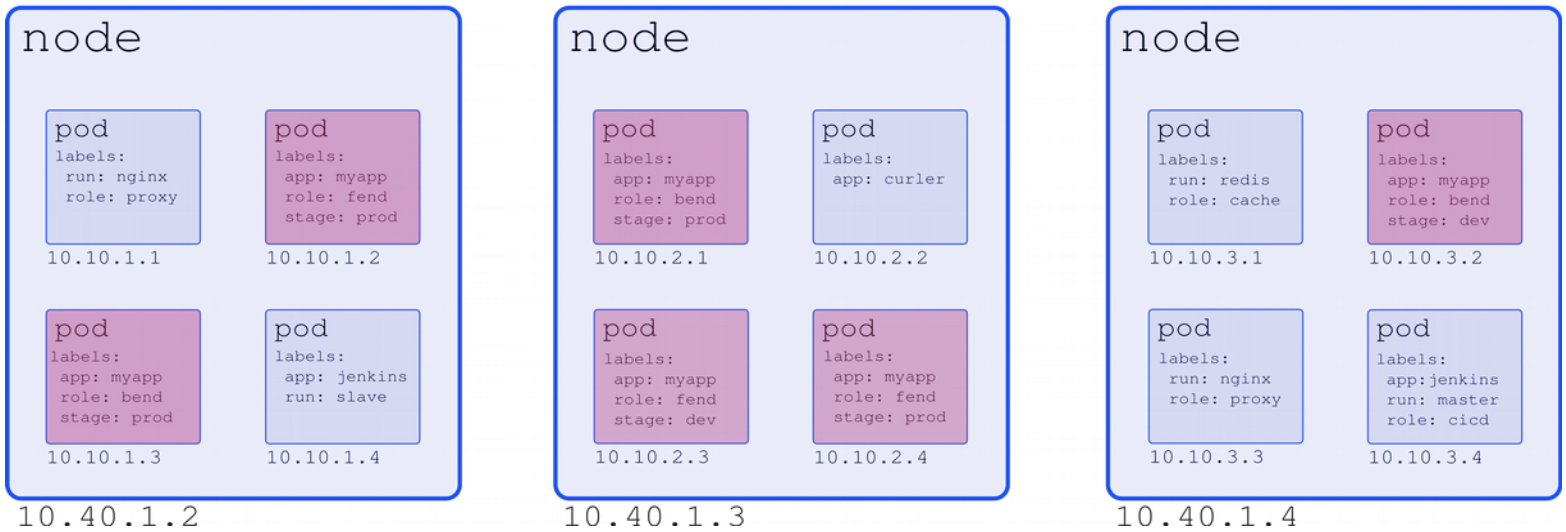
Each Key must be unique for a given object.

Labels


node
pod
labels:
run: nginx
role: proxy
10.10.1.1
pod
labels:
app: myapp
role: fend
stage: prod
10.10.1.2
pod
labels:
app: myapp
role: bend
stage: prod
10.10.1.3
pod
labels:
app: jenkins
run: slave
10.10.1.4
10.40.1.2
node
pod
labels:
app: myapp
role: bend
stage: prod
10.10.2.1
pod
labels:
app: curler
10.10.2.2
pod
labels:
app: myapp
role: fend
stage: dev
10.10.2.3
pod
labels:
app: myapp
role: fend
stage: prod
10.10.2.4
10.40.1.3
node
pod
labels:
run: redis
role: cache
10.10.3.1
pod
labels:
app: myapp
role: bend
stage: dev
10.10.3.2
pod
labels:
run: nginx
role: proxy
10.10.3.3
pod
labels:
app:jenkins
run: master
role: cicd
10.10.3.4
10.40.1.4

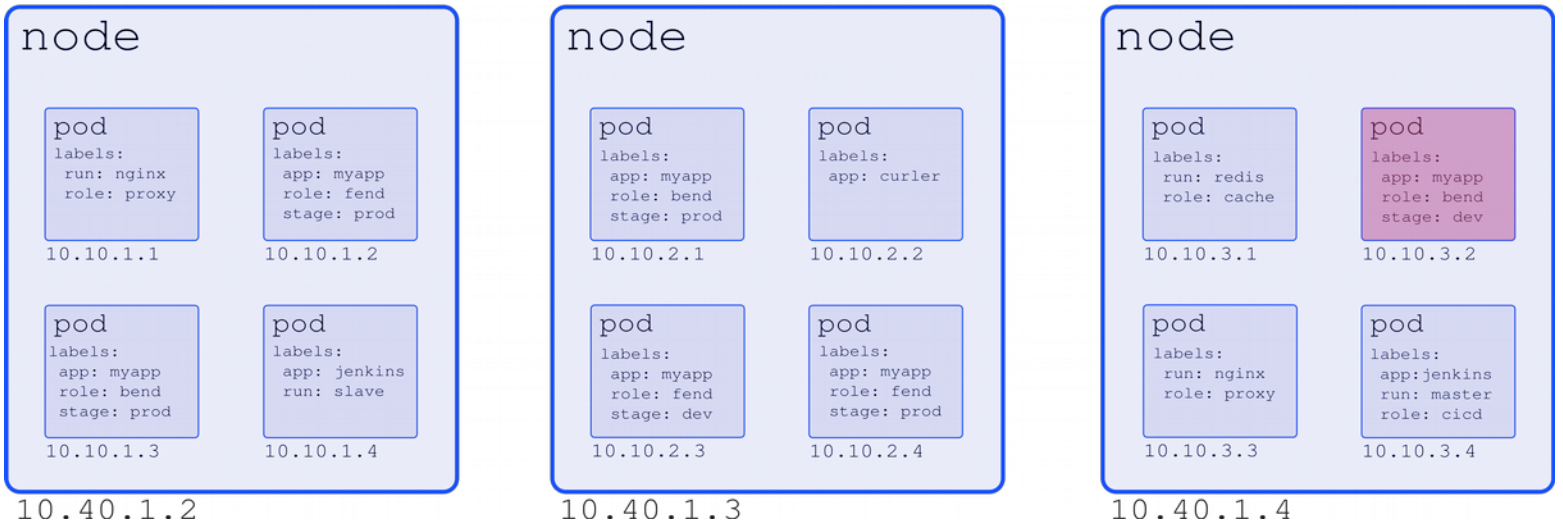
Labels

```
label selector:
  app: myapp
```



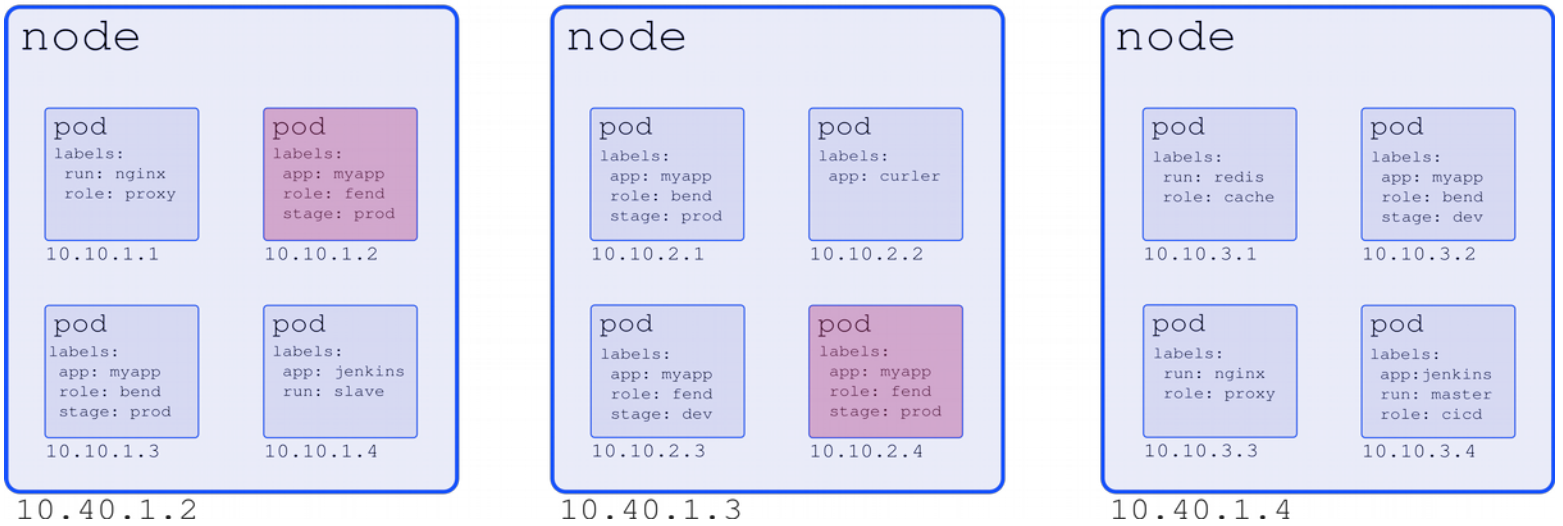
Labels

```
label selector:
  app: myapp
  role: bend
  stage: dev
```



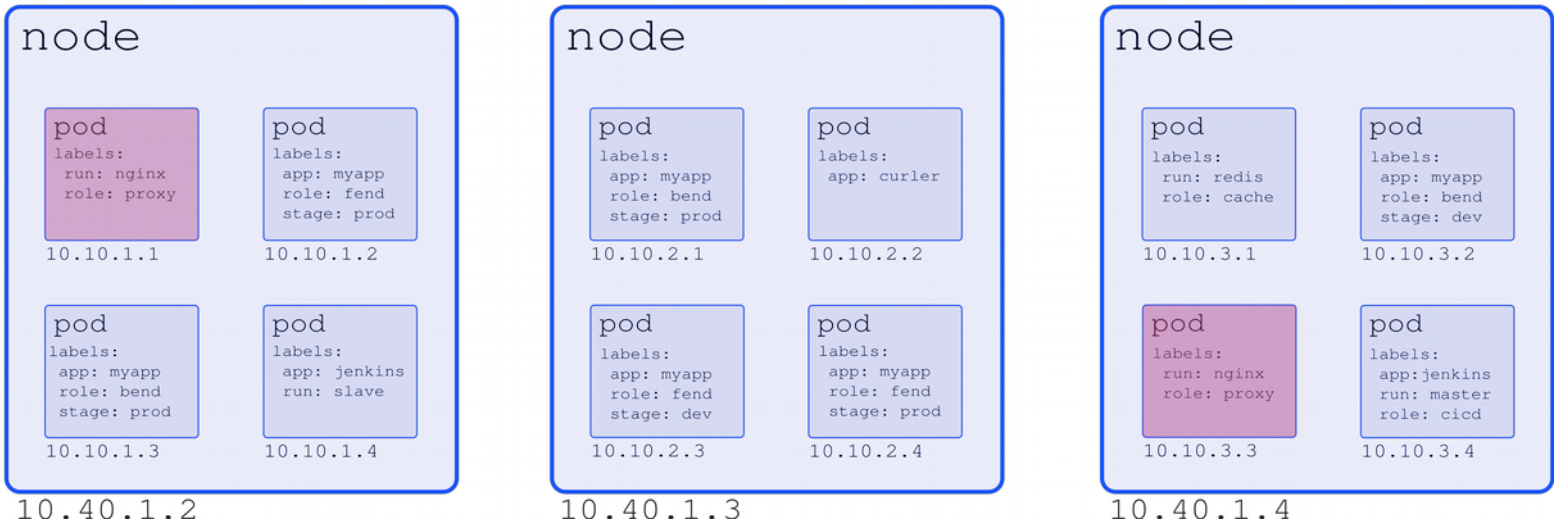
Labels

```
label selector:
  app: myapp
  role: fend
  stage: prod
```



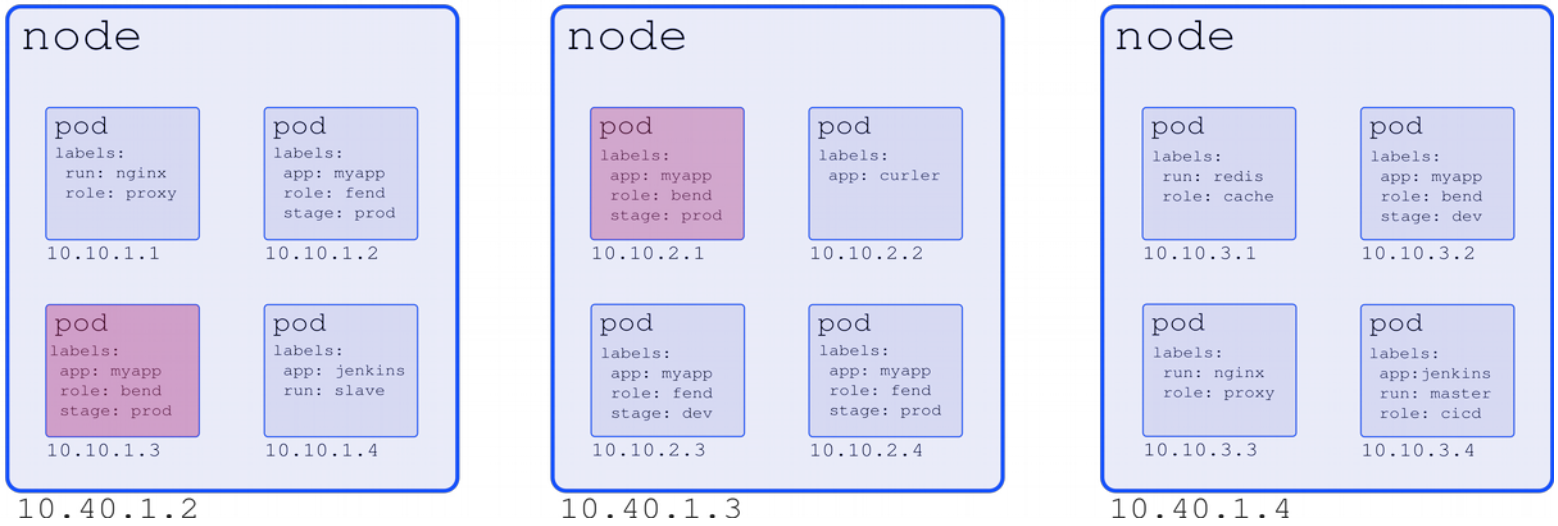
Labels

```
label selector:
  run: nginx
  role: proxy
```



Labels

```
selector:
  matchLabels:
    app: myapp
  matchExpressions:
    - {key: role, operator: In, values: [bend]}
    - {key: stage, operator: NotIn, values: [dev]}
```



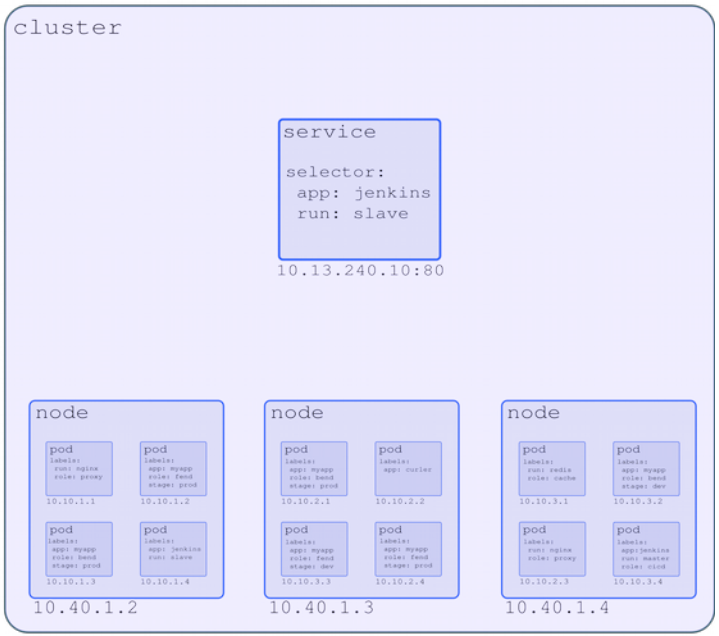
Services

A Service is an abstraction which defines a logical set of Pods, through their labels.

A service has an IP address, and all requests to this IP address are forwarded to the pods with the matched label.

There are four type of services:

- ClusterIP: the service is only reachable from within the cluster.
- NodePort: the service is reachable from outside cluster; on a static port (30000 - 32767), on each node.
- LoadBalancer(*): the service is reachable from outside cluster, through TCP load balancer, on a standard port.
- ExternalName: Maps the service to a CNAME record with its value, from the parameter .spec.externalName



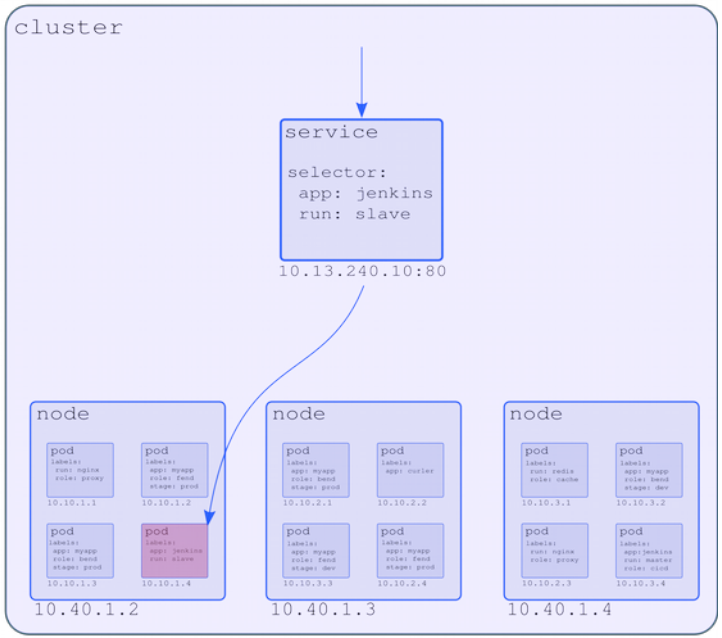
Services

A Service is an abstraction which defines a logical set of Pods, through their labels.

A service has an IP address, and all requests to this IP address are forwarded to the pods with the matched label.

There are four type of services:

- ClusterIP: the service is only reachable from within the cluster.
- NodePort: the service is reachable from outside cluster; on a static port (30000 - 32767), on each node.
- LoadBalancer(*): the service is reachable from outside cluster, through TCP load balancer, on a standard port.
- ExternalName: Maps the service to a CNAME record with its value, from the parameter .spec.externalName



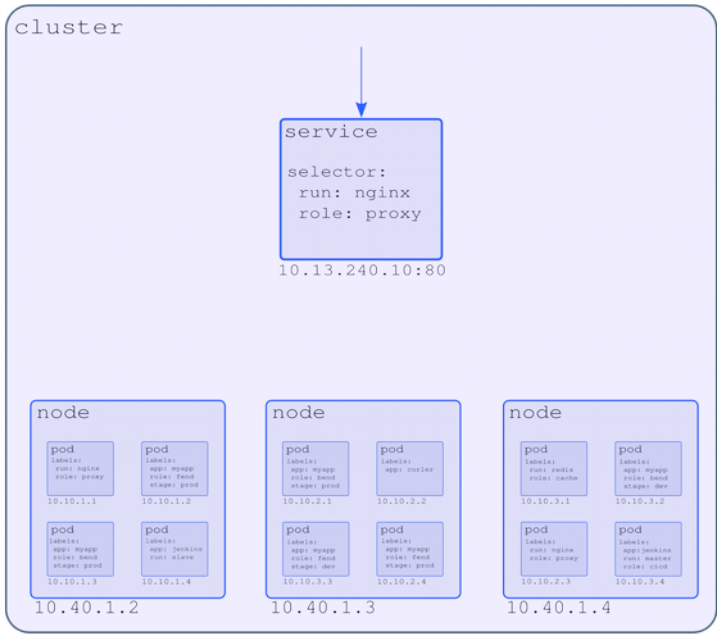
Services

A Service is an abstraction which defines a logical set of Pods, through their labels.

A service has an IP address, and all requests to this IP address are forwarded to the pods with the matched label.

There are four type of services:

- ClusterIP: the service is only reachable from within the cluster.
- NodePort: the service is reachable from outside cluster; on a static port (30000 - 32767), on each node.
- LoadBalancer(*): the service is reachable from outside cluster, through TCP load balancer, on a standard port.
- ExternalName: Maps the service to a CNAME record with its value, from the parameter .spec.externalName



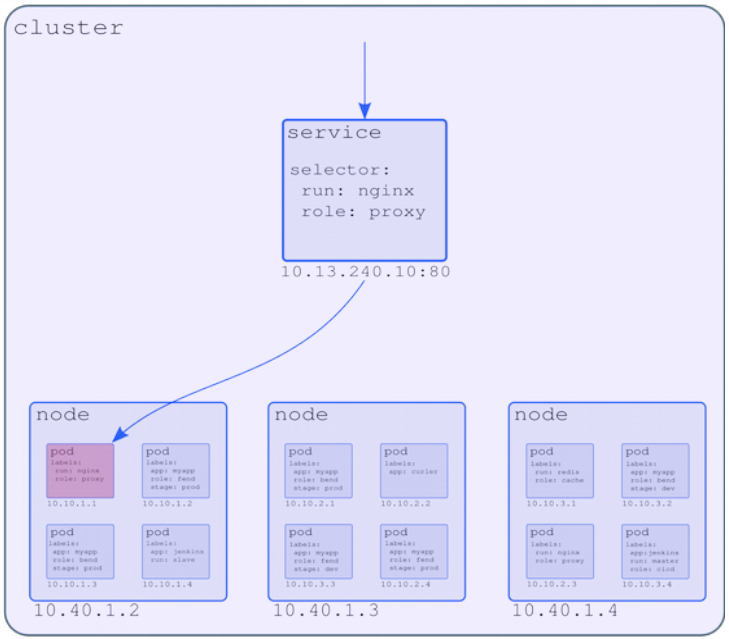
Services

A Service is an abstraction which defines a logical set of Pods, through their labels.

A service has an IP address, and all requests to this IP address are forwarded to the pods with the matched label.

There are four type of services:

- ClusterIP: the service is only reachable from within the cluster.
- NodePort: the service is reachable from outside cluster; on a static port (30000 - 32767), on each node.
- LoadBalancer(*): the service is reachable from outside cluster, through TCP load balancer, on a standard port.
- ExternalName: Maps the service to a CNAME record with its value, from the parameter .spec.externalName



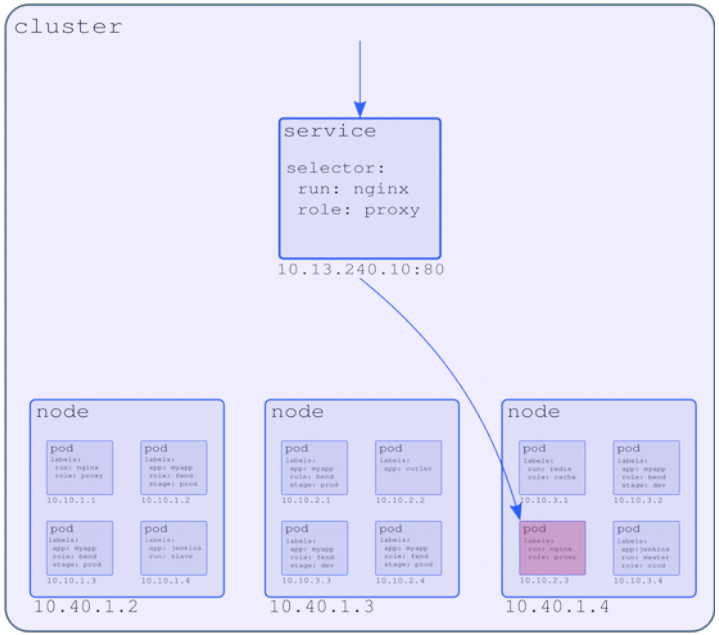
Services

A Service is an abstraction which defines a logical set of Pods, through their labels.

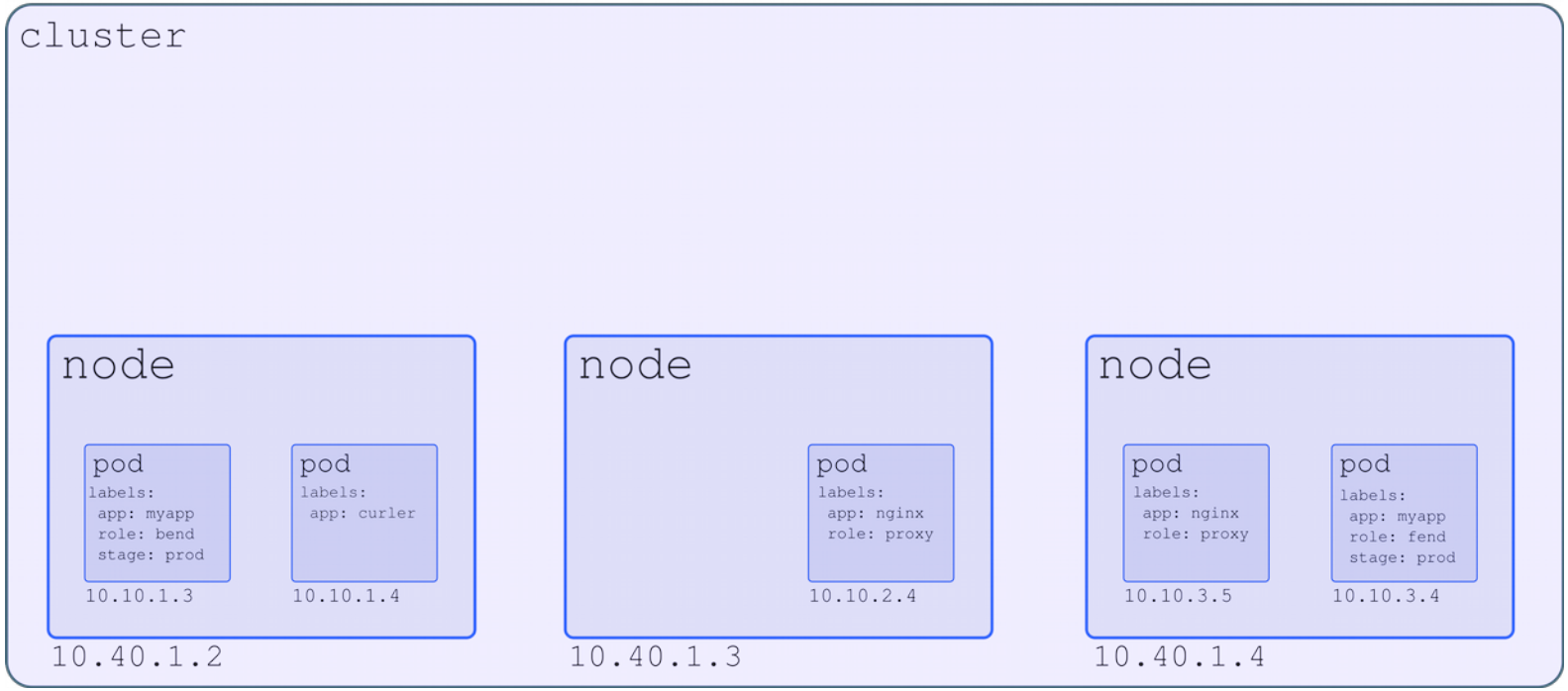
A service has an IP address, and all requests to this IP address are forwarded to the pods with the matched label.

There are four type of services:

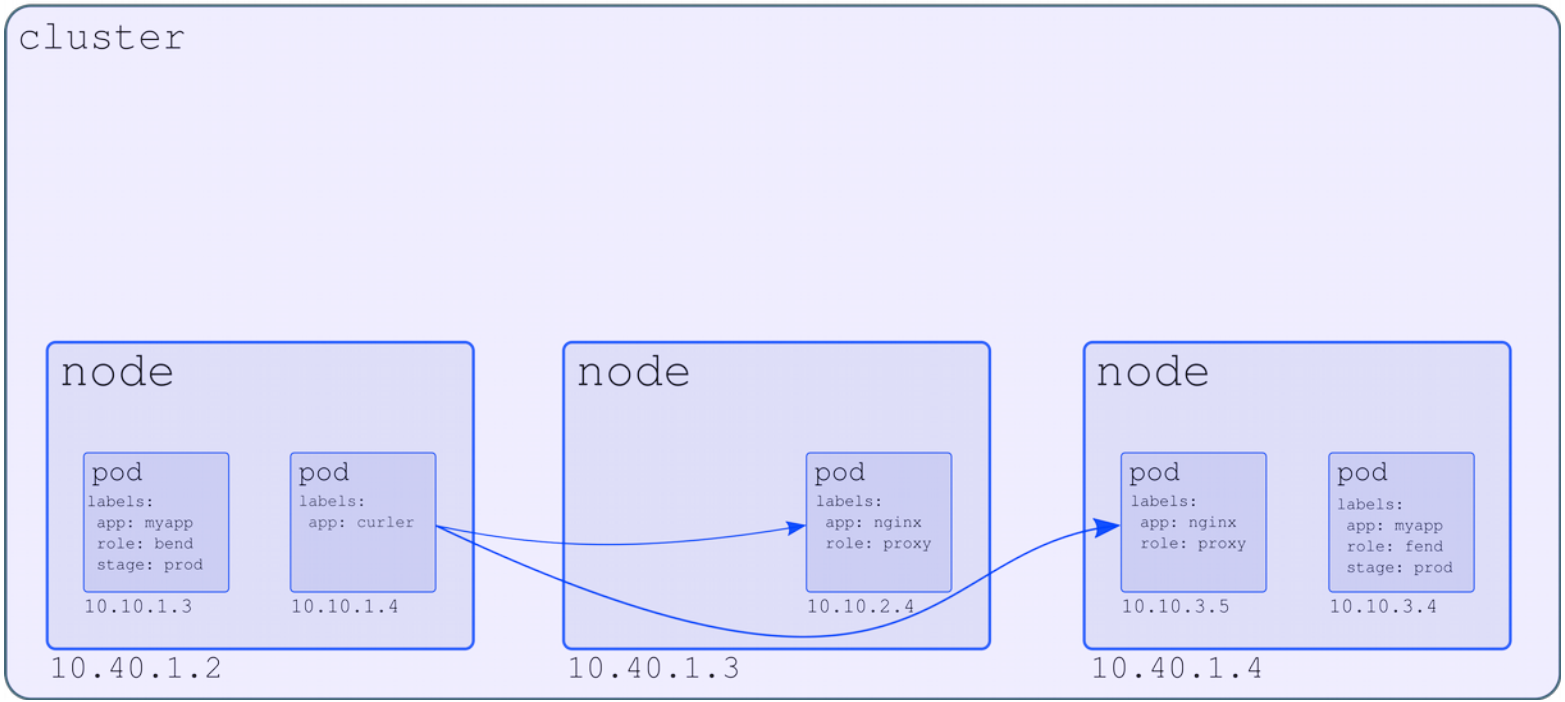
- ClusterIP: the service is only reachable from within the cluster.
- NodePort: the service is reachable from outside cluster; on a static port (30000 - 32767), on each node.
- LoadBalancer(*): the service is reachable from outside cluster, through TCP load balancer, on a standard port.
- ExternalName: Maps the service to a CNAME record with its value, from the parameter .spec.externalName



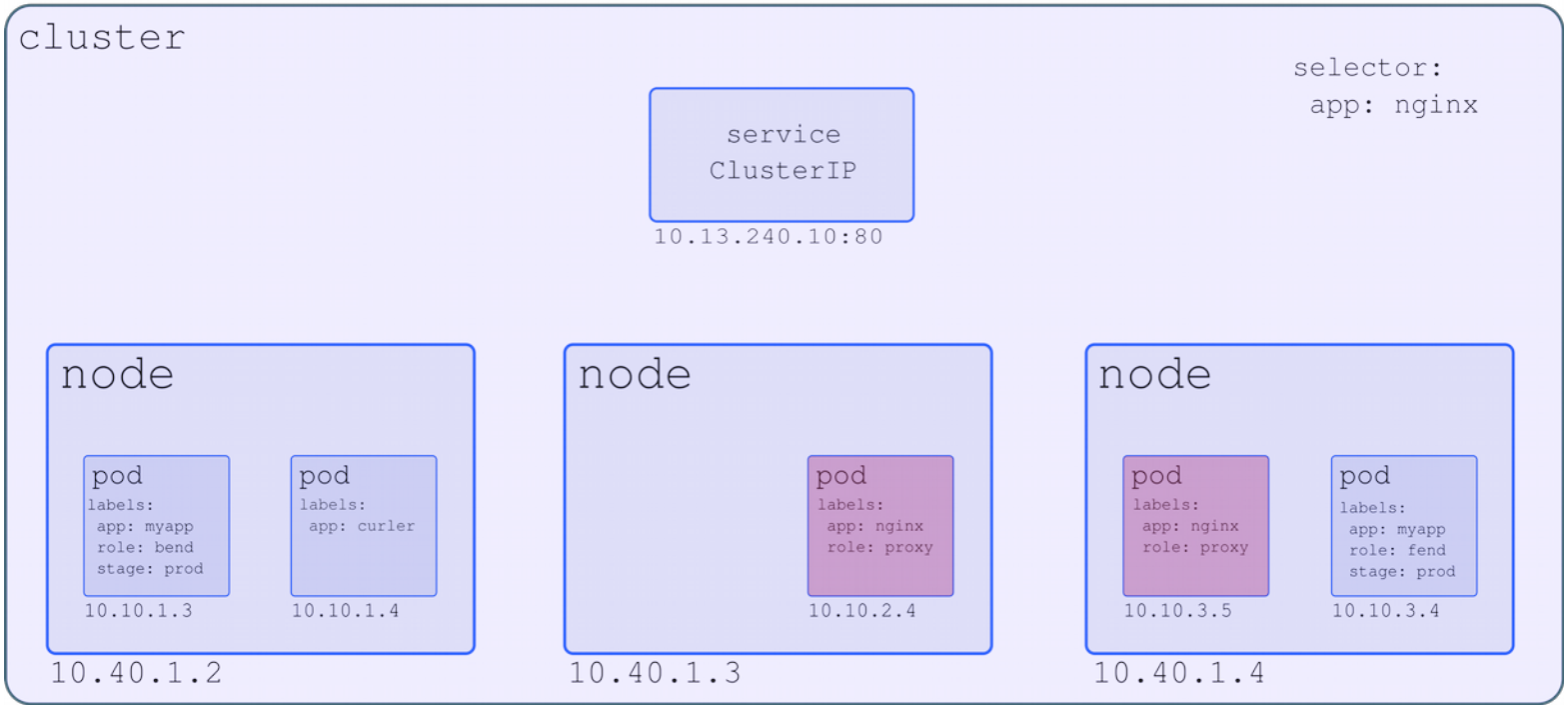
Services – ClusterIP



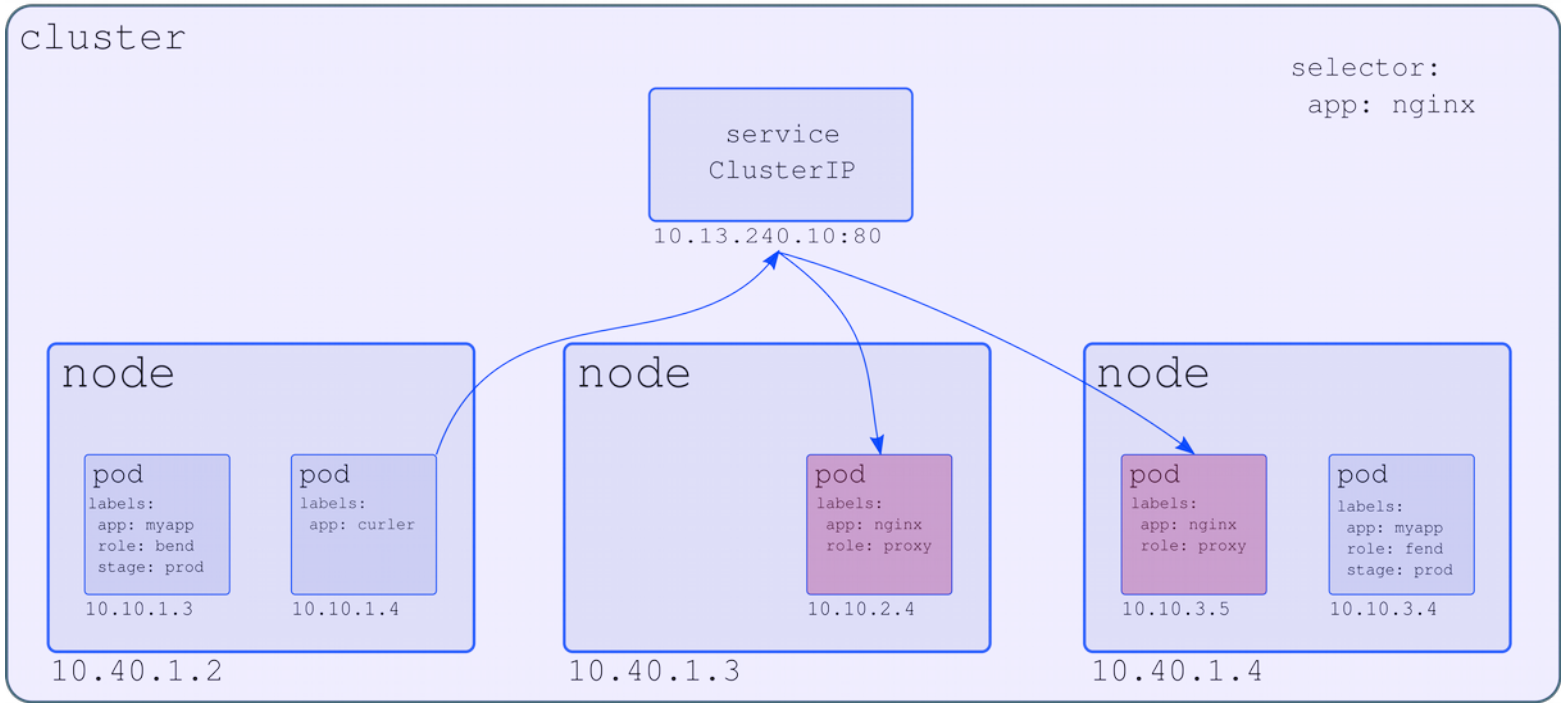
Services – ClusterIP



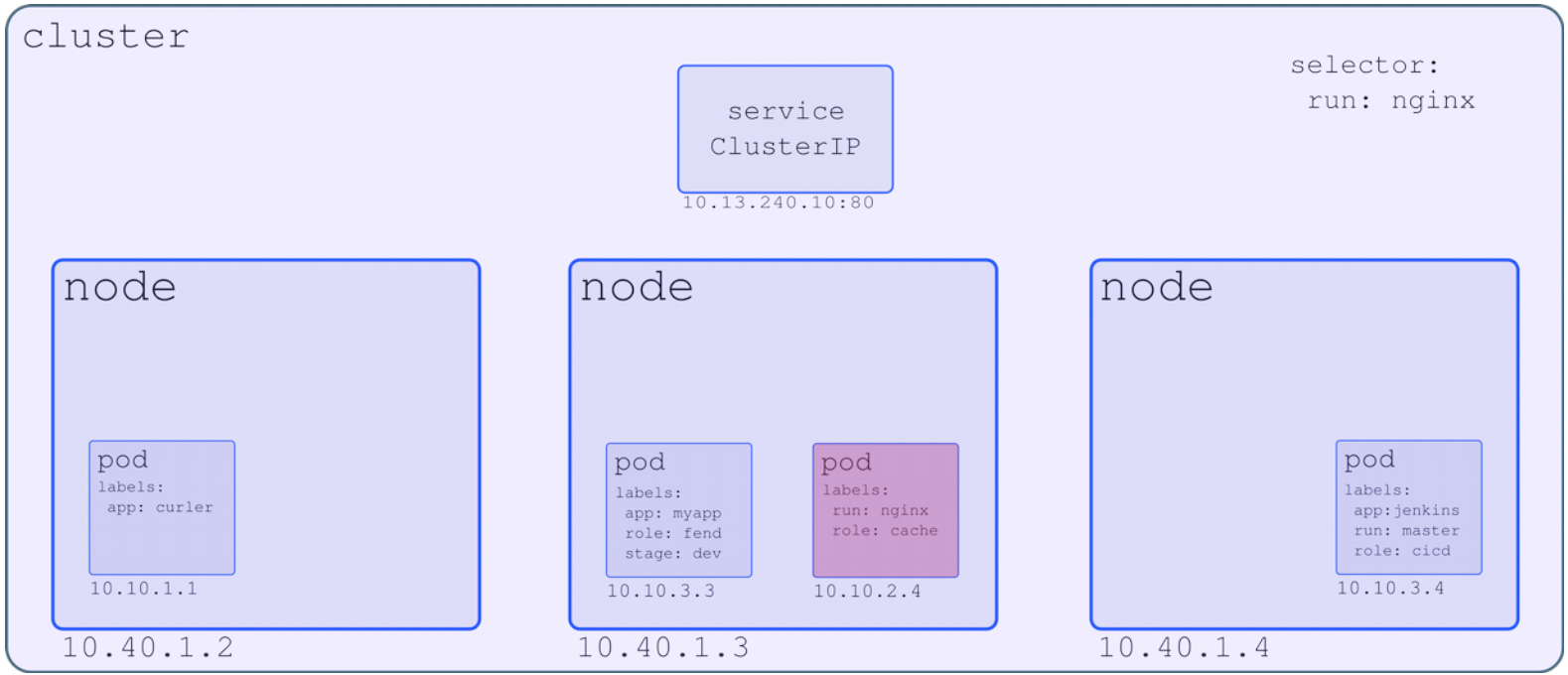
Services – ClusterIP



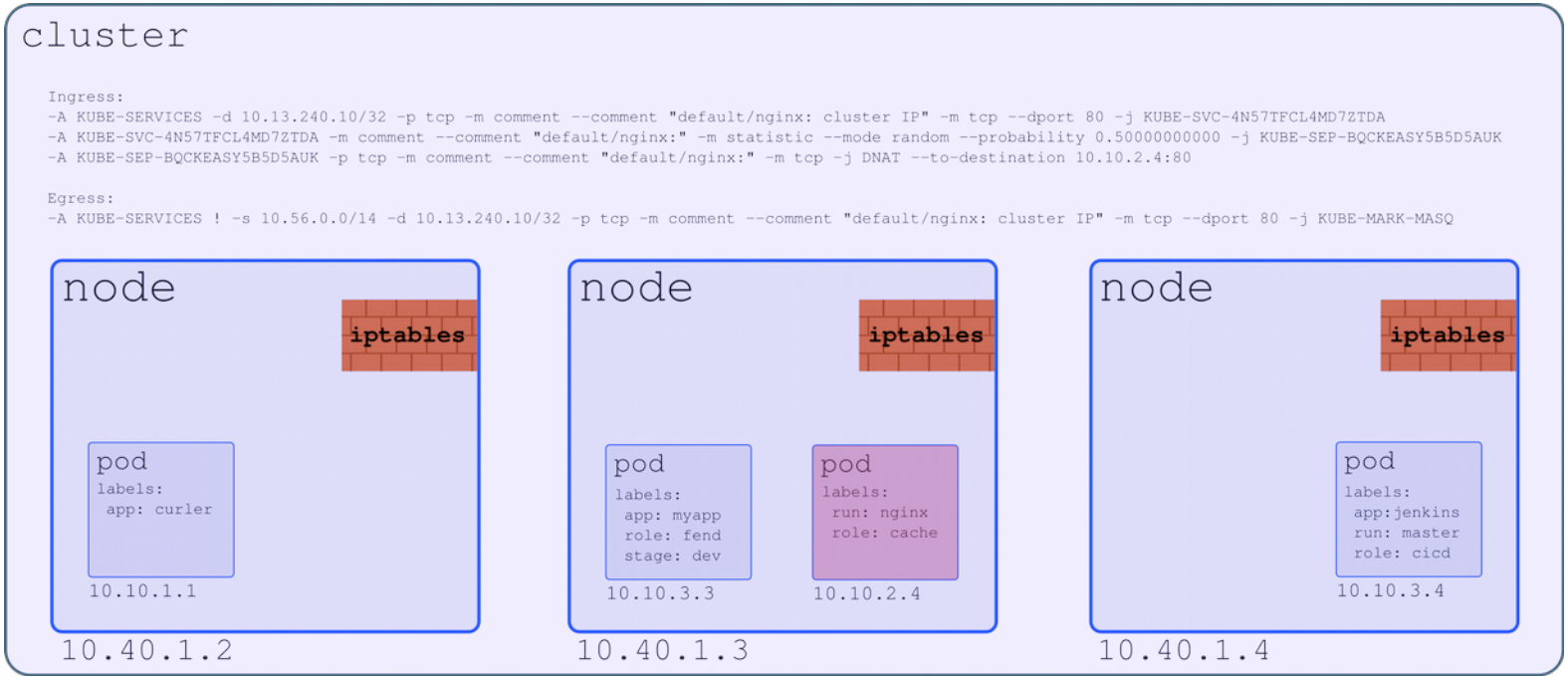
Services – ClusterIP



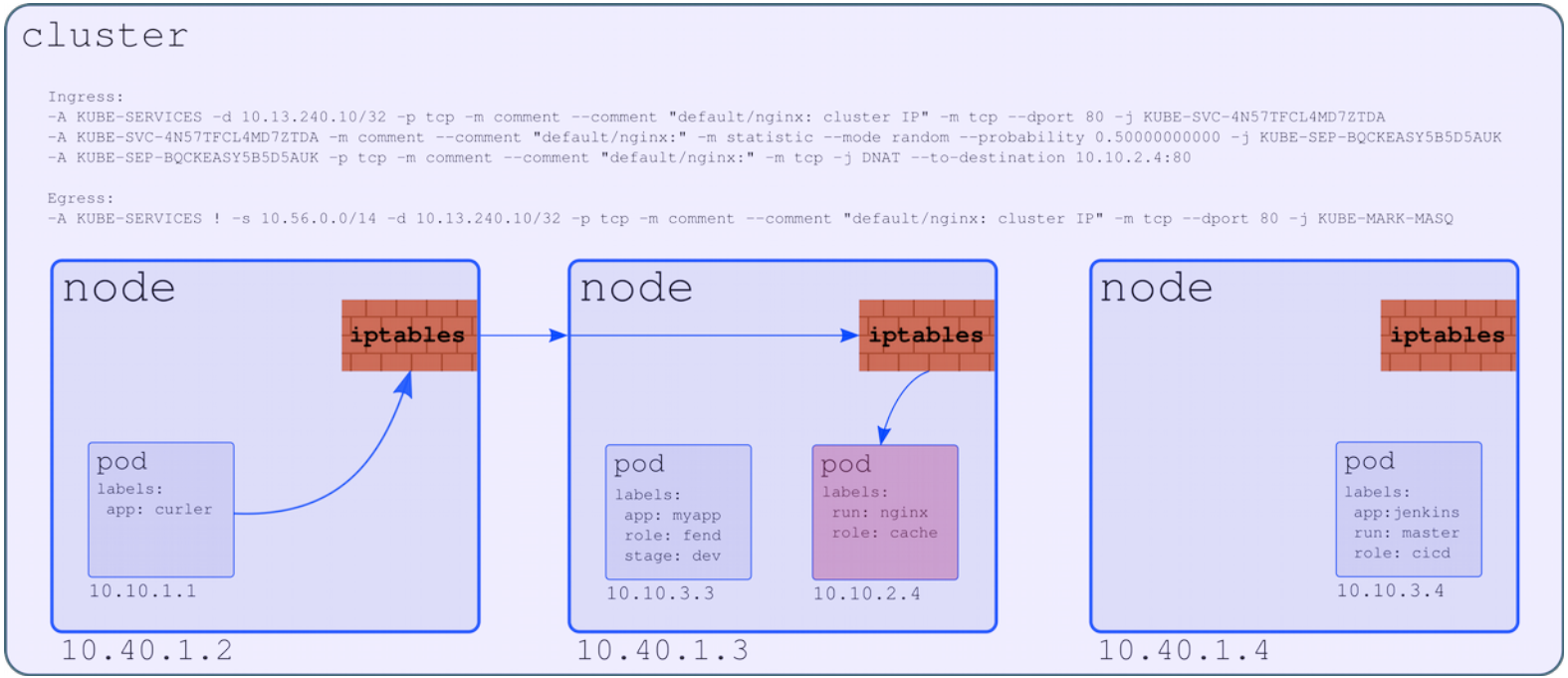
Services – ClusterIP



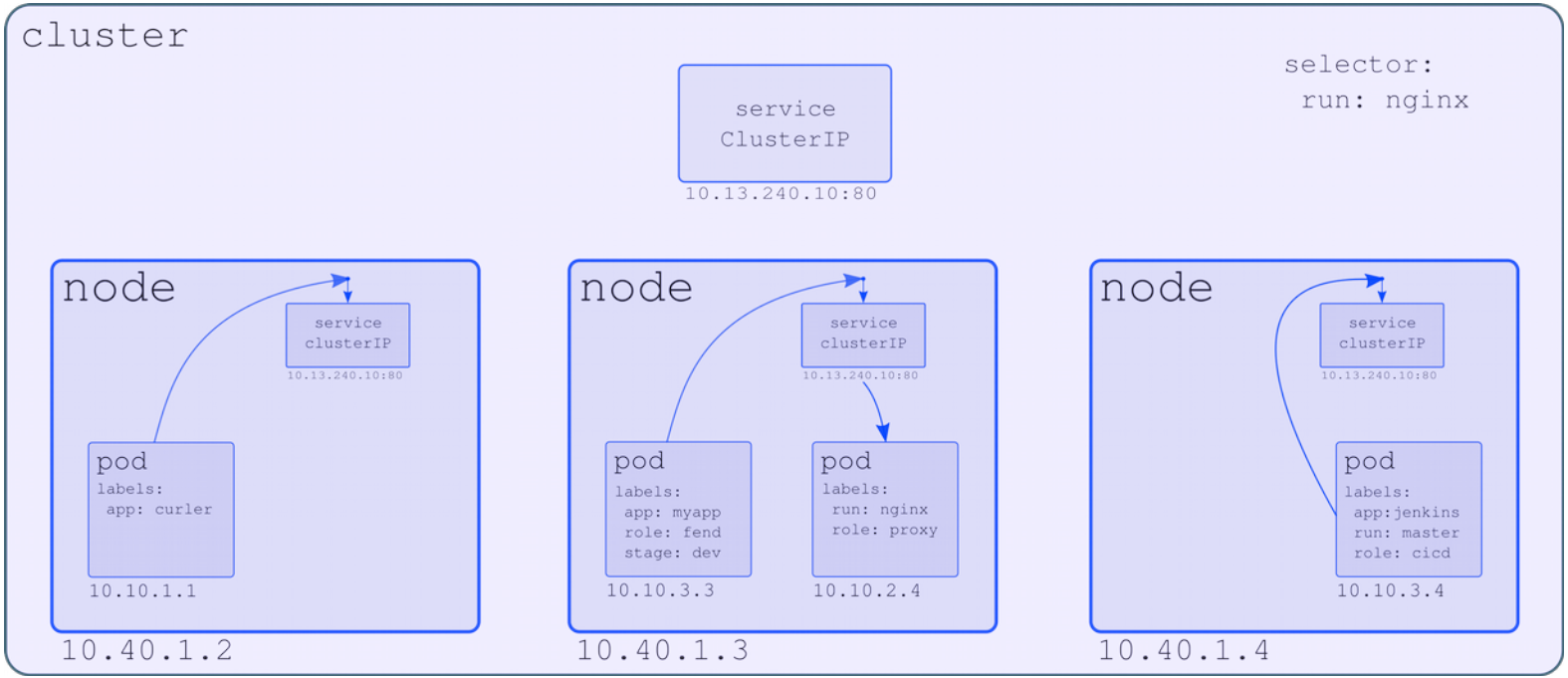
Services – ClusterIP



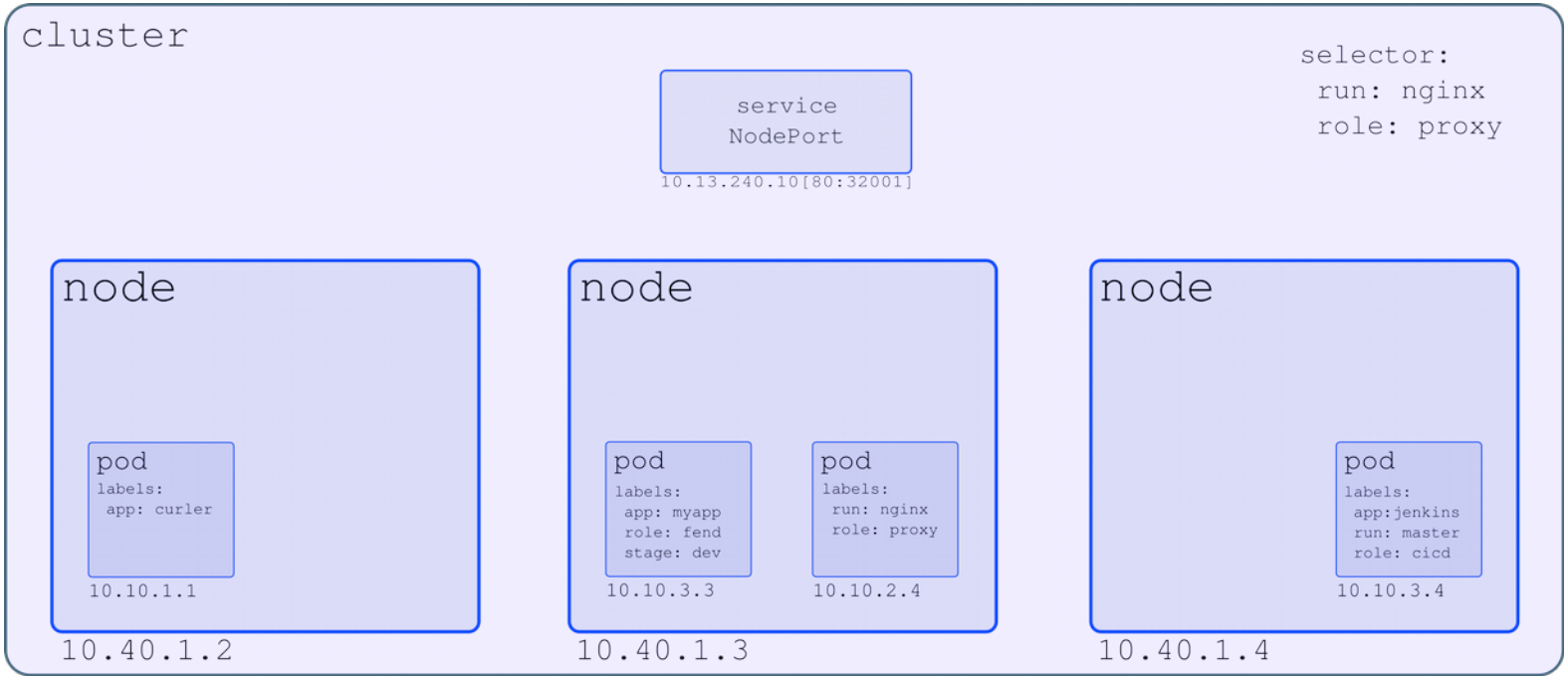
Services – ClusterIP



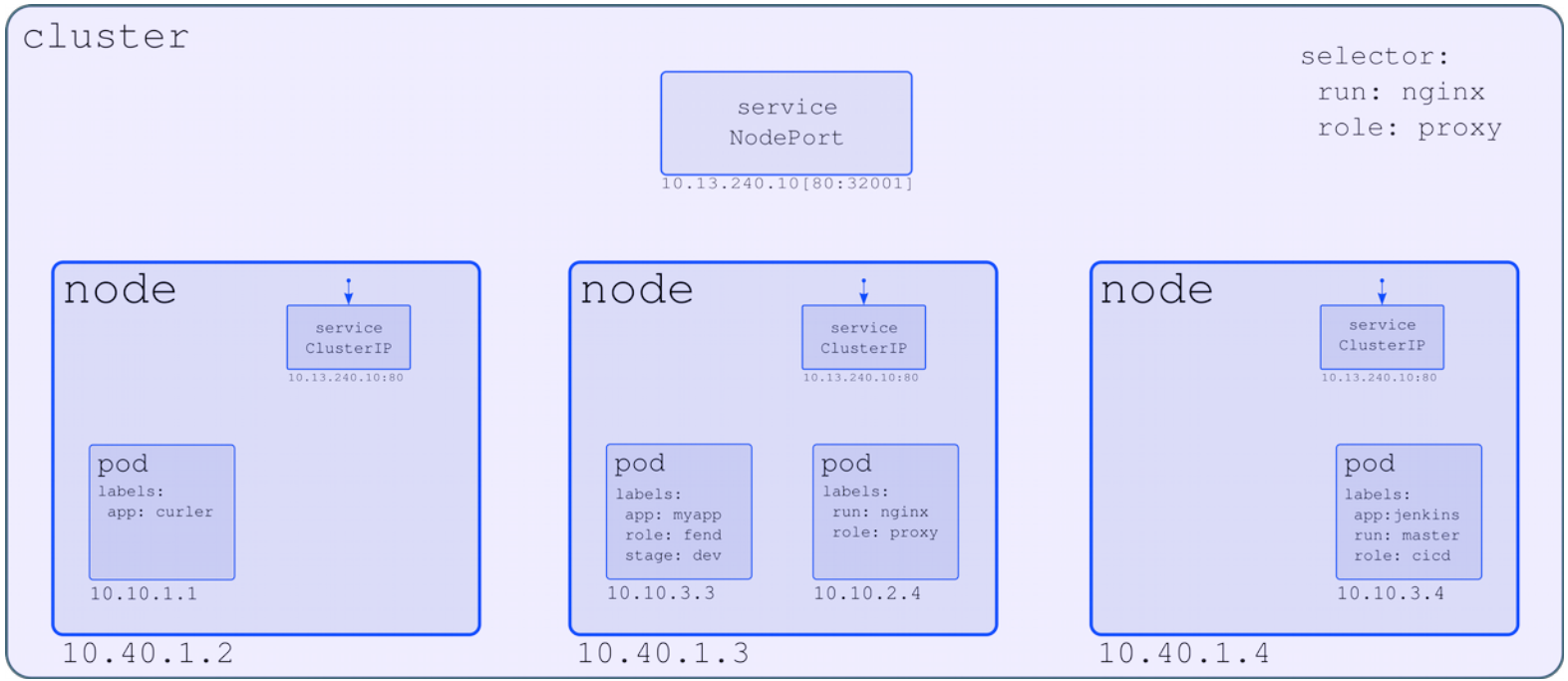
Services – ClusterIP



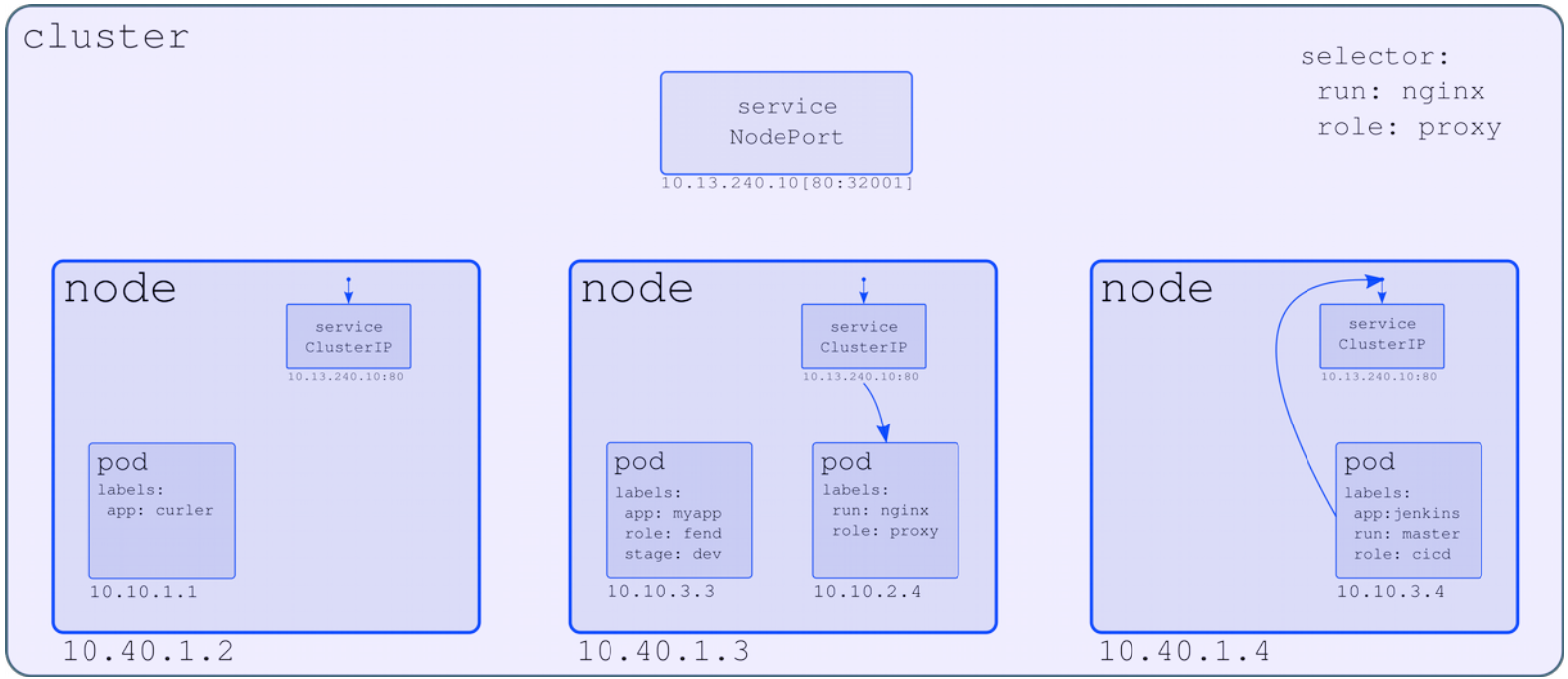
Services – NodePort



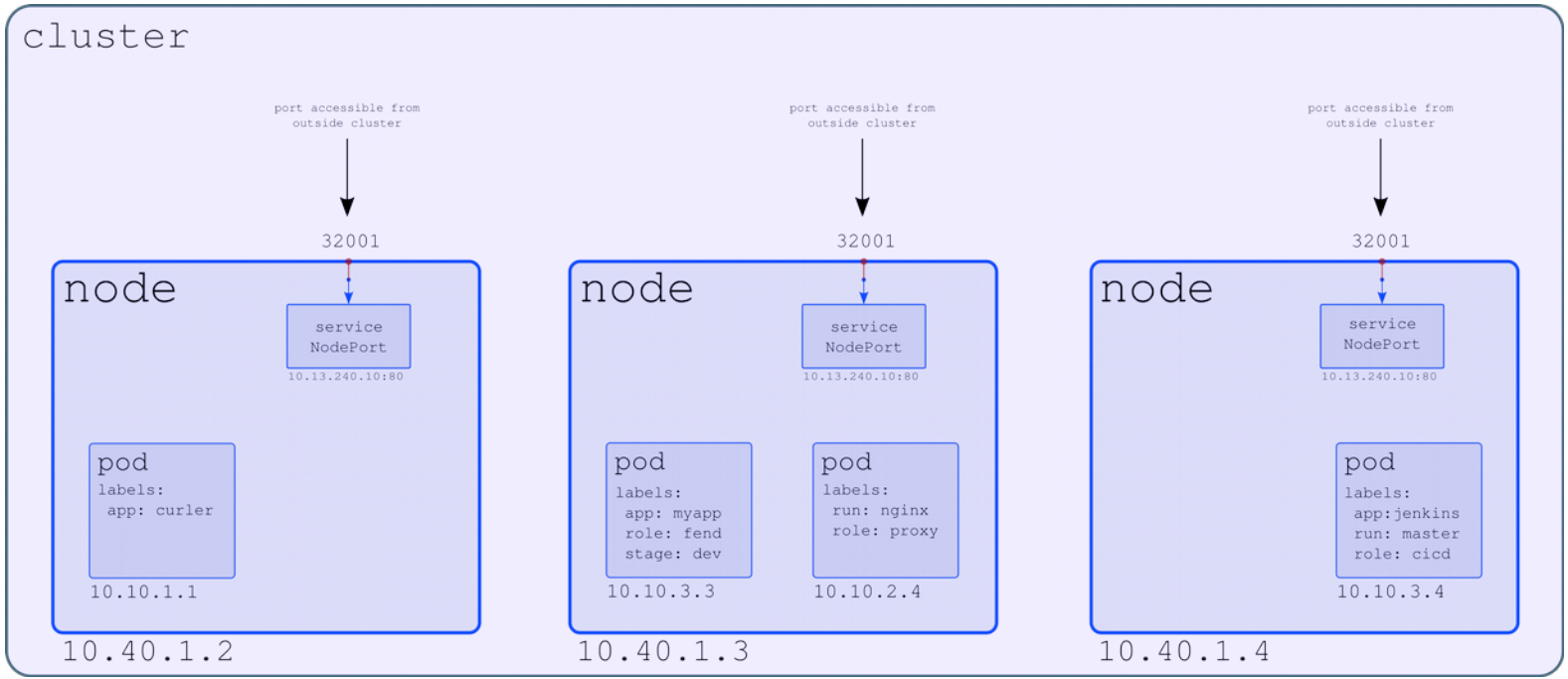
Services – NodePort



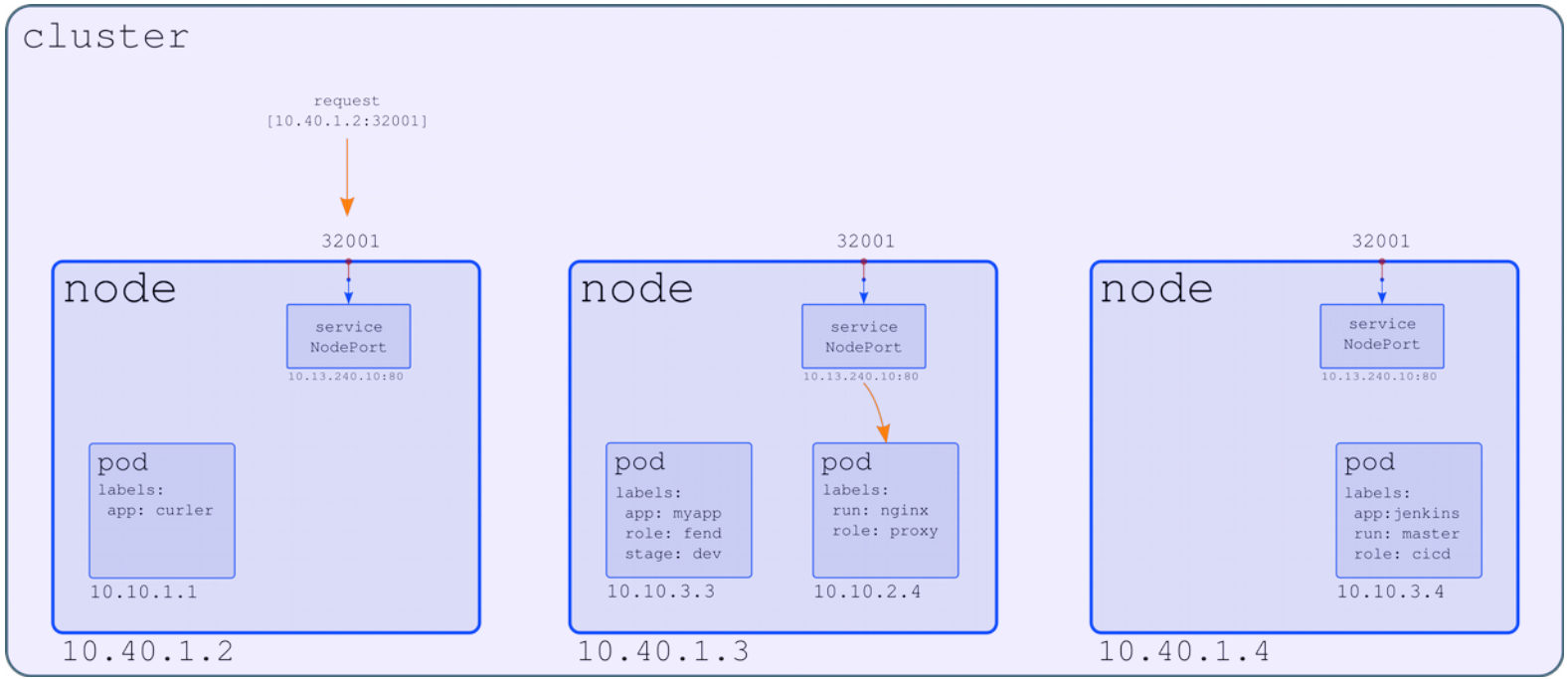
Services – NodePort



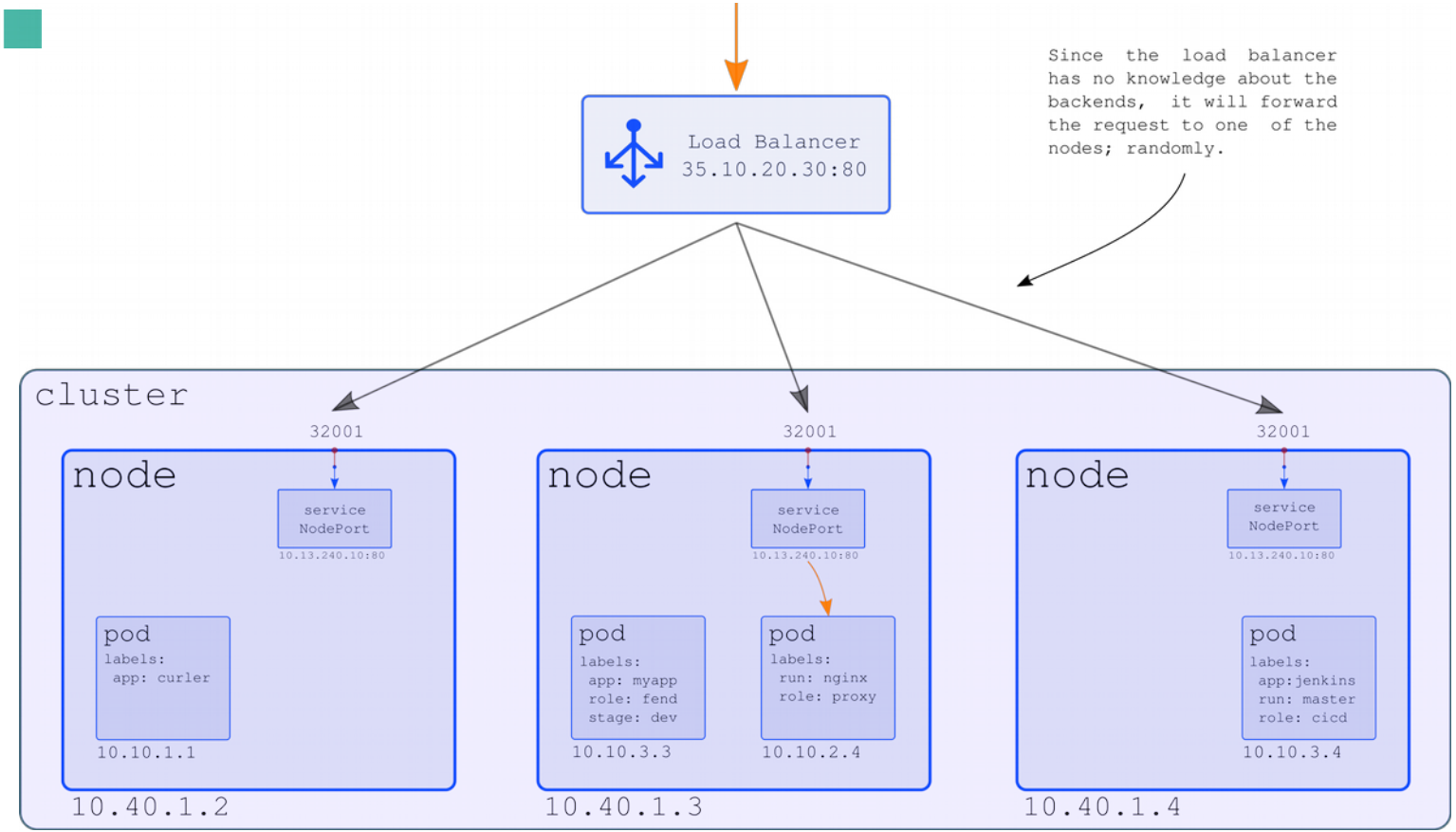
Services – NodePort



Services – NodePort



Services – LoadBalancer



Lab 2

```
1.- Create ClusterIP type service with svc-ip.yaml file.
    $ kubectl create -f svc-ip.yaml
2.- Describe the service, and verify the previously created pod is targeted, under `endpoints`.
    $ kubectl describe service svc-nginx
3.- Get the IP address of the service.
4.- Create and access a helper pod, with curler.yaml
    $ kubectl create -f curler.yaml
    $ kubectl exec -it curler sh
5.- curl the service
    $ curl SVC_IP_ADDR

6.- Edit the service, and change the type to be NodePort.
    $ kubectl edit svc svc-nginx
7.- Identify on which port of the node the service is running.
    $ kubectl get svc
8.- Create a firewall rule in GCP to open the node port.
    $ gcloud compute firewall-rules create pXXXX --allow tcp:XXXX
9.- Identify the IP addresses of the nodes
    $ kubectl describe node | grep -E 'InternalIP|ExternalIP'
10.- Access curler pod again (or from anywhere else), and curl any node; on that port.

11.- Modify again the service and change the service type to be LoadBalancer.
12.- Once the external address has been assigned to the service, curl it from anywhere.
```

Volumes

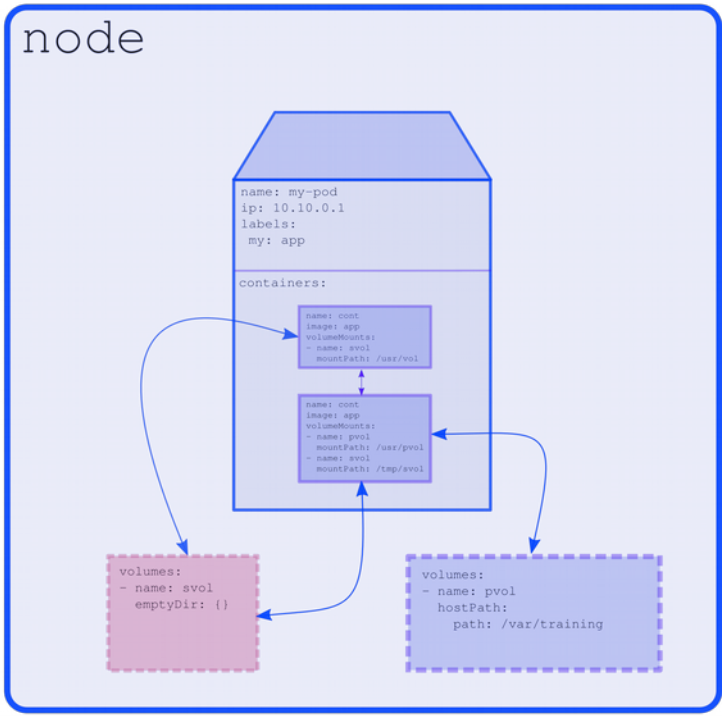
Kubernetes volume abstraction solves two problems that containers present:

- The data in a container is ephemeral. If a container crashes, kubelet will restart it, but the data will be lost.
- When running more than one container in a pod, sometimes it is necessary to share files among them.

There are several types of volumes. K8s supports most cloud providers physical storages/volumes, and many third-party filesystems.

Other Kubernetes objects, like configMaps, secrets and DownwardAPI, can be mounted as volume in a container.

Each Container in the pod must specify independently where to mount each volume.



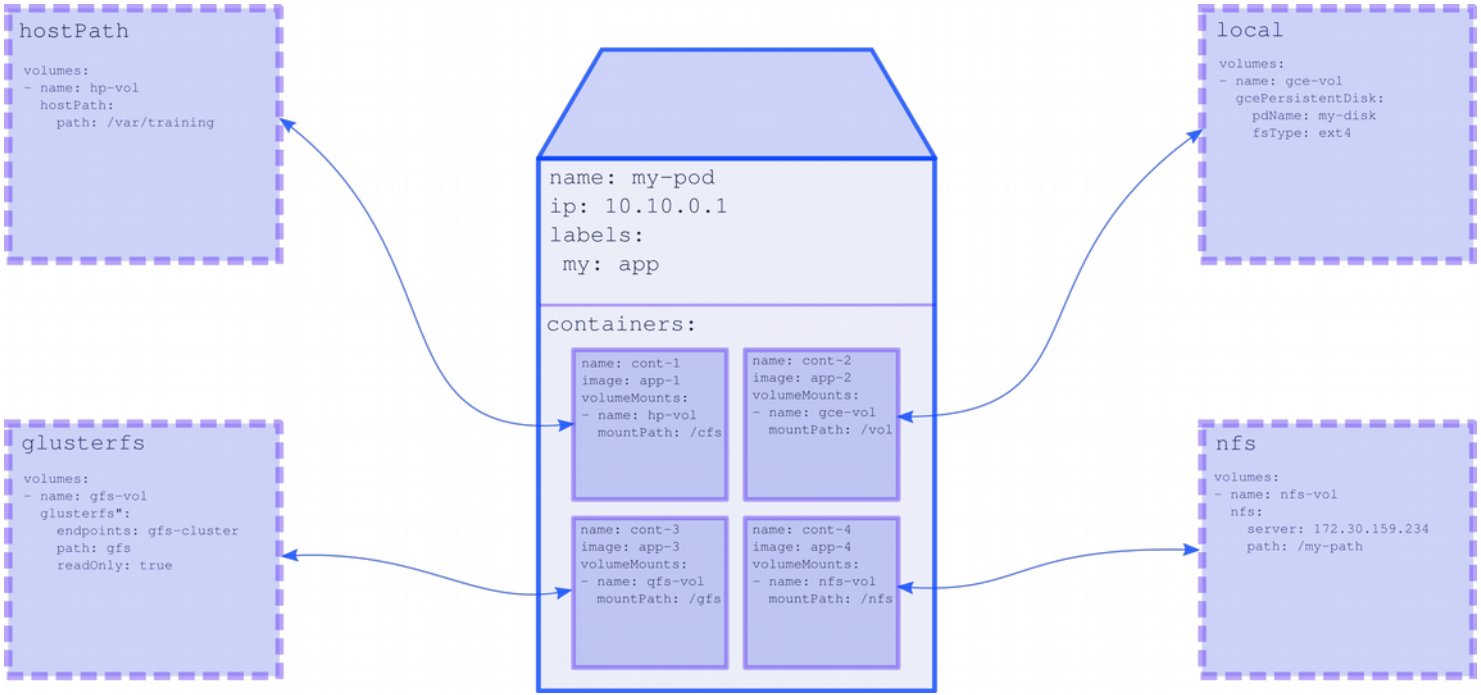
Volumes – provisioning

The PersistentVolume subsystem provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed. Kubernetes offers two API resources for this purpose: PersistentVolume and PersistentVolumeClaim

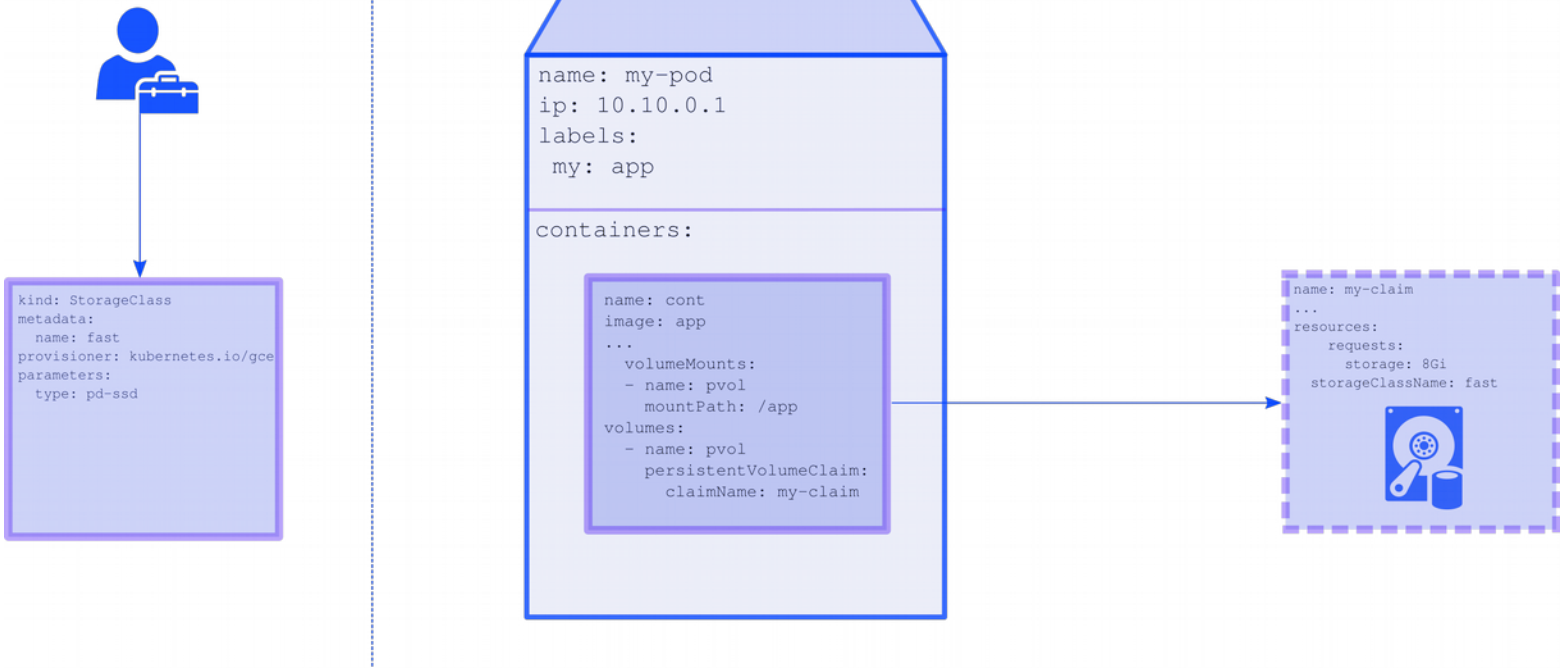
A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator, as a resource. PVs have independent lifecycle of any individual pod that uses the PV. The administrator will explicitly specify the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system. Since PVs are to be provisioned beforehand, will be used storage statically

A PersistentVolumeClaim (PVC) is a request for storage by a user. The request will need to be fulfilled, for a pod to be provisioned with a persistent storage. Similar to pods that consume node resources in terms of CPU and memory, PVCs consume PV resources, in terms of storage size and access mode (e.g. can be mounted once read/write or many times read-only). PVCs are to provision storage, dynamically.

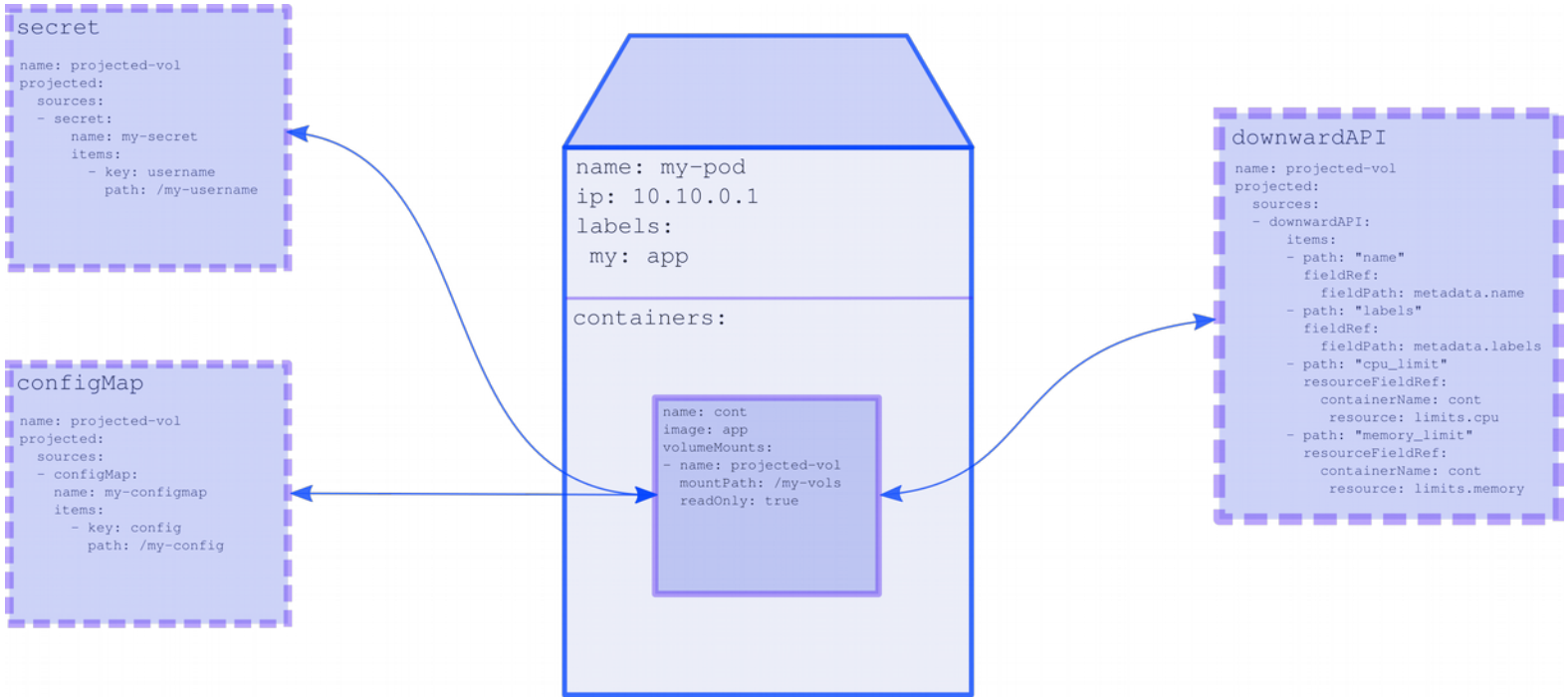
Volumes – PersistentVolume



Volumes – PersistentVolumeClaim



Volumes – Projected Volumes



Lab 3

1.- Create a PersistentVolume and a Pod with pv-n-pod.yaml file.
    $ kubectl create -f pv-n-pod.yaml
2.- Access the pod and go to the path mapped in the container.
    $ kubectl exec -it POD bash
3.- Create a file called <YOUR-NAME>-file
    $ touch <YOUR-NAME>-file
4.- Exit from the container.
5.- Identify the node, there the pod is running
    $ kubectl get pod -owide
6.- SSH into the node and go to the path mapped in the host, and confirm the file is there.
    $ gcloud compute ssh NODE

7.- Create a PersistentVolumeClaim and a Pod with pvc-n-pod-yaml.
    $ kubectl create -f pvc-n-pod.yaml
8.- Observe the relationship between PV and PVC.
    $ kubectl get pv
    $ kubectl get pvc
9.- Once the block has been created, access the conainer and confirm it has been attached correctly.
    $ kubectl exec -it POD bash
    $ df -h

10.- Create a PVC with a StatefulSet to see dynamic volume provisioning (instructor).
    $ kubectl create -f pvc-template.yaml

end