Kubernetes

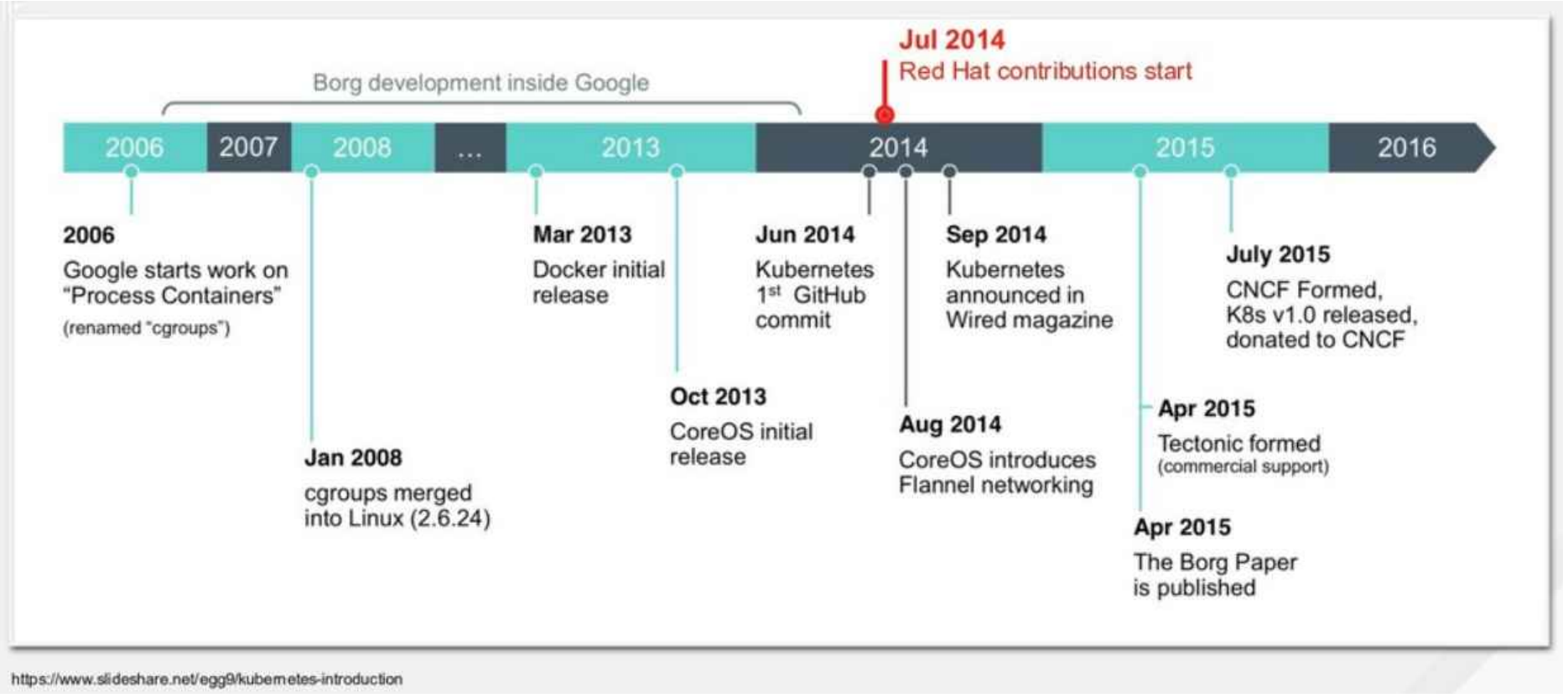
Paulo Vigne

O que é Kubernetes?

Kubernetes

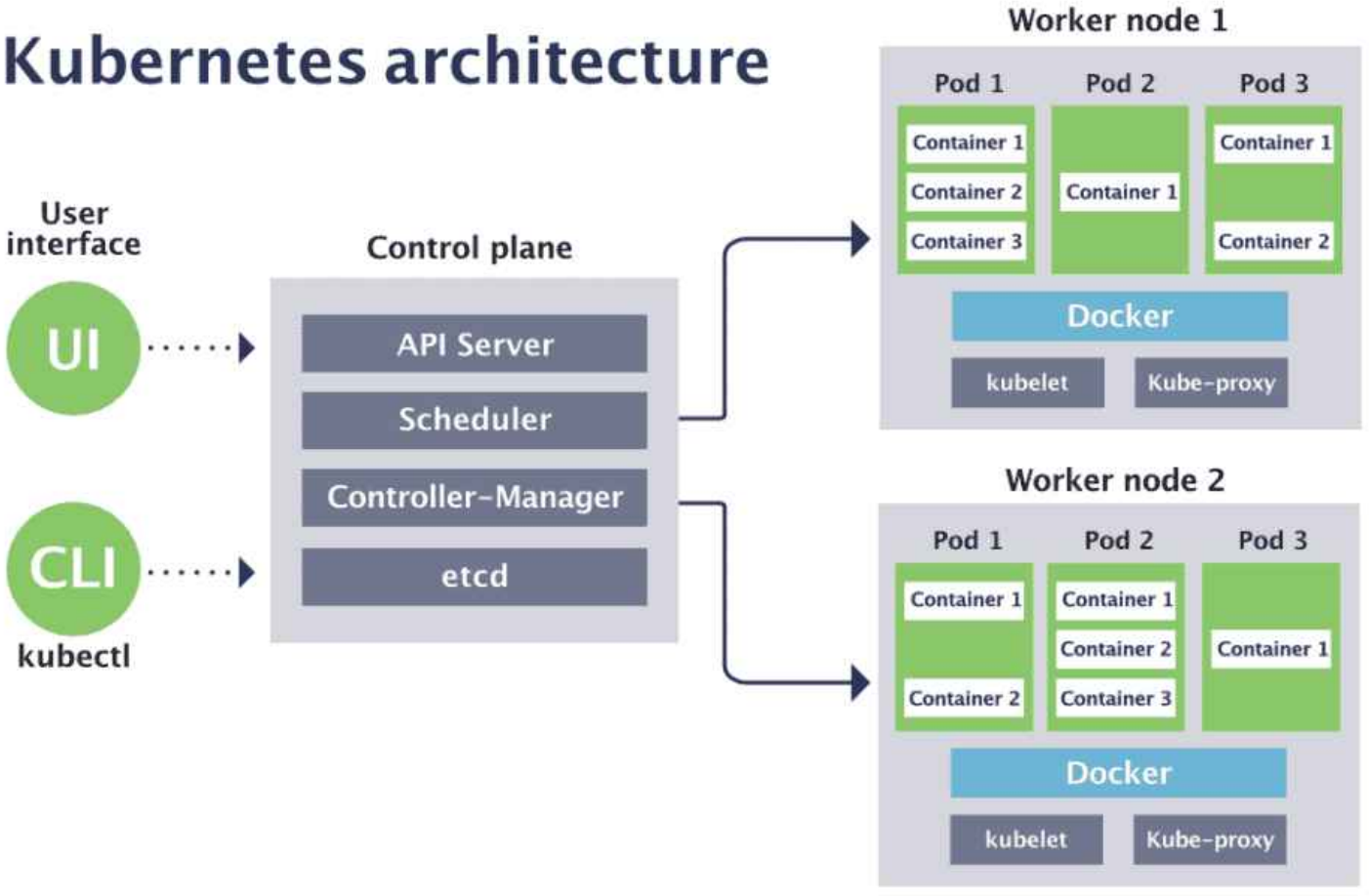
É uma plataforma open-source **portável** e **extensível** para orquestração de contêineres e serviços, que possui como facilidades sua linguagem declarativa e automação.

Kubernetes Timeline



https://www.slideshare.net/egg9/kubernetes-introduction

Kubernetes architecture



Control Plane Components

etcd
É um banco de NoSQL de Chave-Valor, sua função é guardar os dados do Cluster, apenas é acessado pelo Kube API Server.

kube-apiserver
Componente principal que expõe a API do Kubernetes. É o Fontend do Control Plane.

kube-controller-manager
Mega binário com a função de gerenciar Nós, Pods e
seus respectivos Controladores. Ainda acumula a função de gerar Service Account Tokens e Certificados de Usuários.

Control Plane Components

kube-scheduler
Observa os PODs recém criados e escalona em seus respectivos nós, verificando ativamente o melhor lugar.

cloud-controller-manager
Interage com o Cloud Provider para criação e remoção de Nós, Rotas e Seviços de LoadBalancer

Node Components

kubelet
É o controlador do Nó, que interage diretamente com o Kube-Api_Server, está presente em todos os nós worker do cluster. Ele é o responsável por instanciar os pods no Run Time de containers.

kube-proxy
Também roda em todos os Workes, é responsável por implementar a Service Network e Rules do kubernetes, fazendo um proxy entre o ambiente kubernetes e a rede do nó.

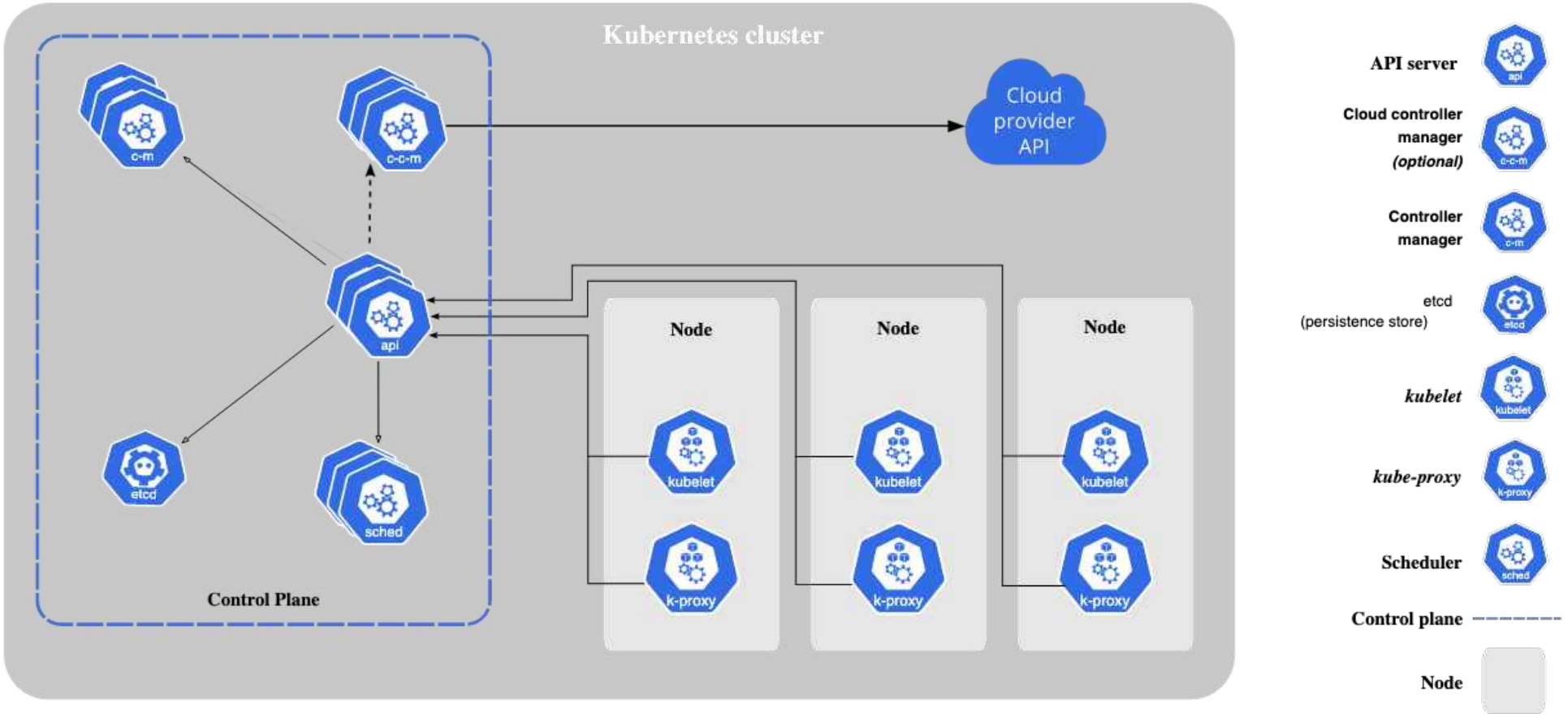
Assistants Components

CNI

Container Network Interface, é a rede interna do cluster kubernetes. Com isso, permite ao Kubernetes criar uma rede totalmente apartada do mundo físico, com o próprio IPAM, rotas e leasings.

DNS

Utilizaremos o coreDNS que em conjunto com o Kubernetes Service atuam como o Service Discovery do cluster.

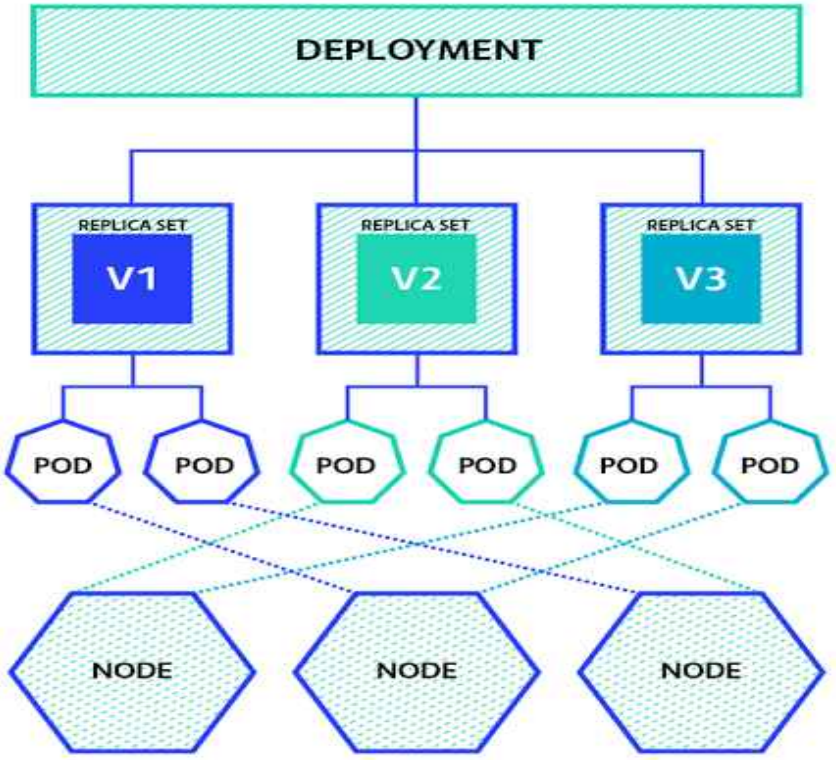


https://kubernetes.io/docs/concepts/overview/components/

Trabalhando com objetos Kubernetes

E o que é POD?

É um objeto que representa um grupo de um ou mais containers.



Tipos de Controladores de PODs

- **ReplicaSet**, o default, de maneira simples, garante a execução do POD e suas replicas conforme o desejado.
- **Daemonset** é uma maneira de garantir que cada nó irá executar uma replica do POD controlado.
- **StatefulSet** foi desenhado para PODs que precisam manter o estado.
- **Job and CronJob** rodam jobs com tarefas pré determinadas.

Trabalhando com objetos Kubernetes

Deployment

É um conceito de alto nível que acrescenta a ReplicaSets gerência de ciclo de vida das aplicações de forma transparente, como upgrades, rollbacks, application revisioning, etc.

Trabalhando com objetos Kubernetes

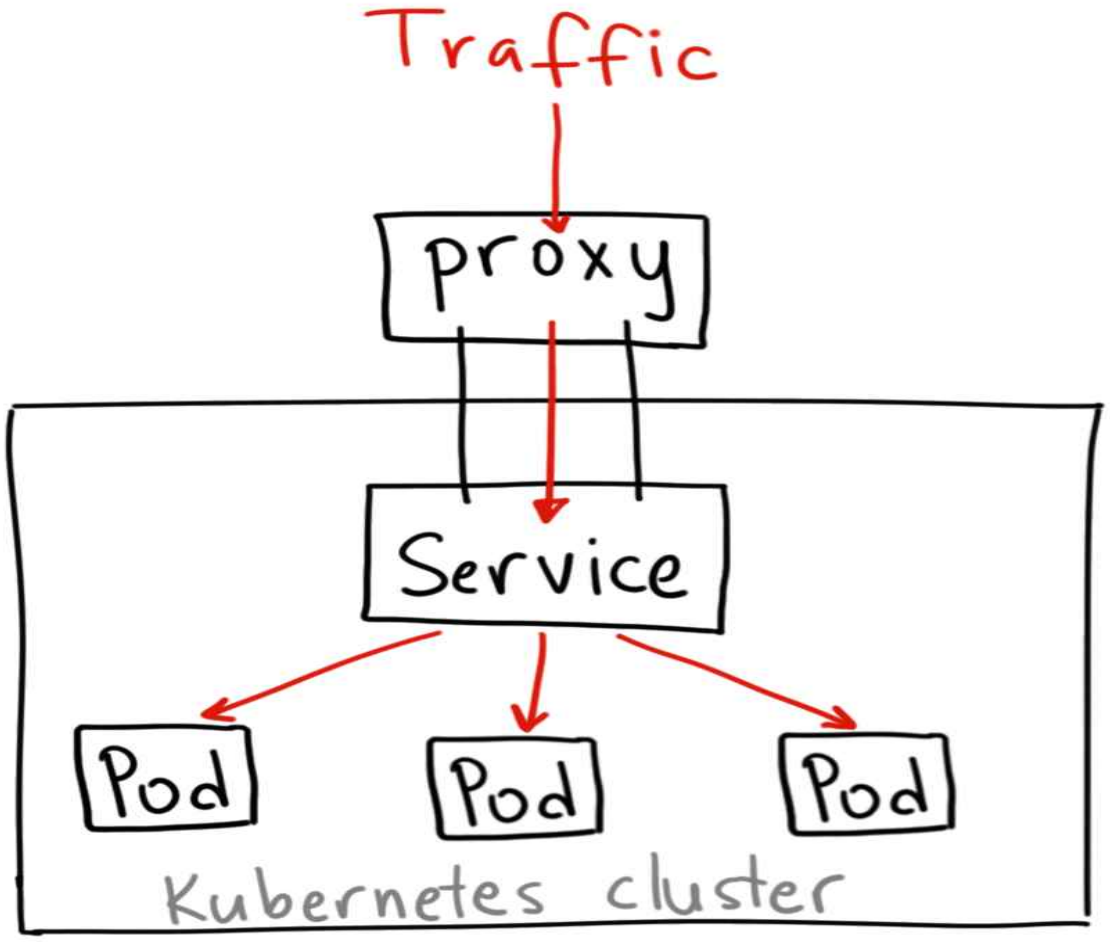
Service

Suponha que tenhamos que fazer uma conexão de rede com um POD, teríamos que descobrir seu endereço IP, para conectar diretamente a ele pela porta da aplicação. Contudo, o endereço IP pode mudar quando o POD for recriado, ou ainda podem existir várias réplicas do POD.
Felizmente neste caso temos o recurso Service, que disponibiliza um endereço IP único, imutável com um nome de DNS, que será automaticamente roteado para qualquer pod com o metadata label associado. Aqui acontece o Service Discovery, combinando o Service Registration e o CoreDNS.

Tipos de Services:

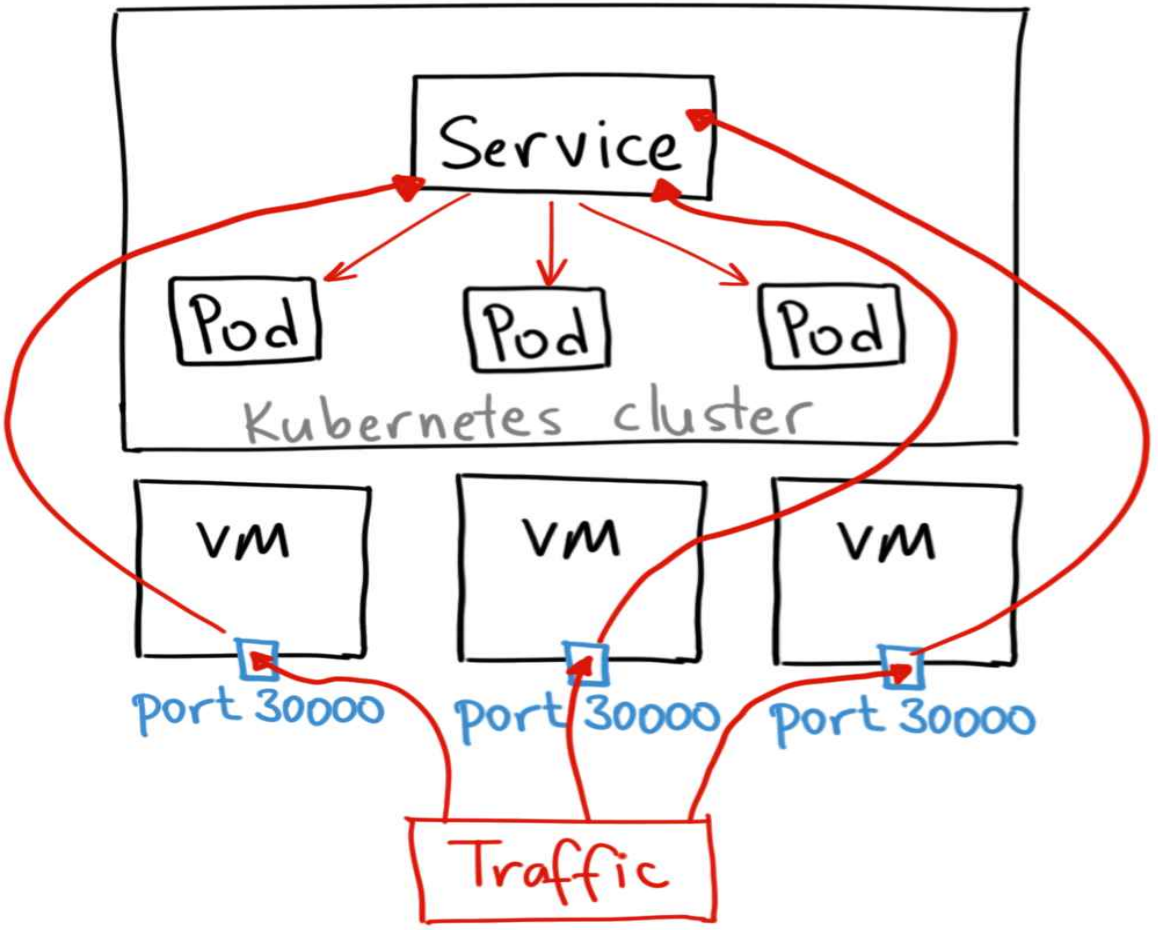
- ClusterIP
- NodePort
- LoadBalancer
- HeadLess

Trabalhando com objetos Kubernetes



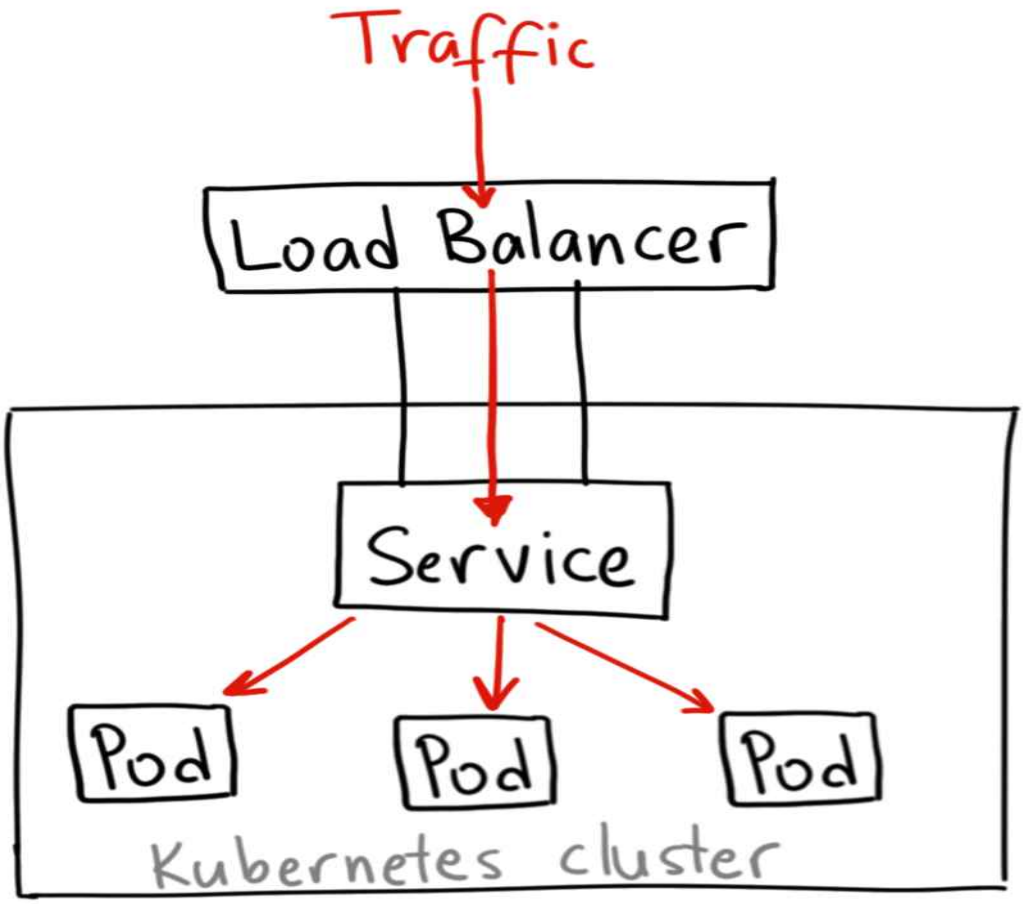

```
apiVersion: v1
kind: Service
metadata:
  name: my-internal-service
spec:
  selector:
    app: my-app
  type: ClusterIP
  ports:
  - name: http
    port: 80
    targetPort: 80
    protocol: TCP
```

Trabalhando com obietos Kubernetes



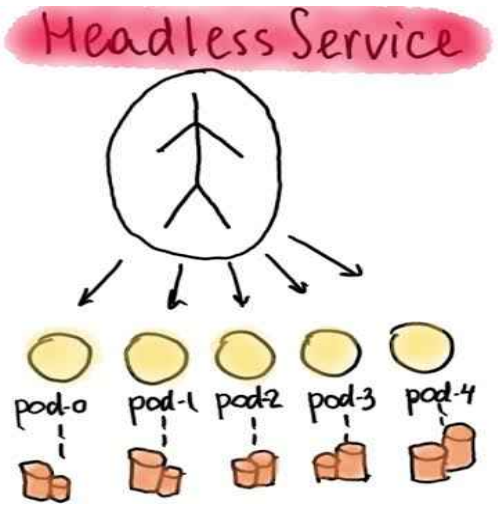

```
apiVersion: v1
kind: Service
metadata:
  name: my-nodeport-service
spec:
  selector:
    app: my-app
  type: NodePort
  ports:
  - name: http
    port: 80
    targetPort: 80
    nodePort: 30000
    protocol: TCP
```

Trabalhando com objetos Kubernetes



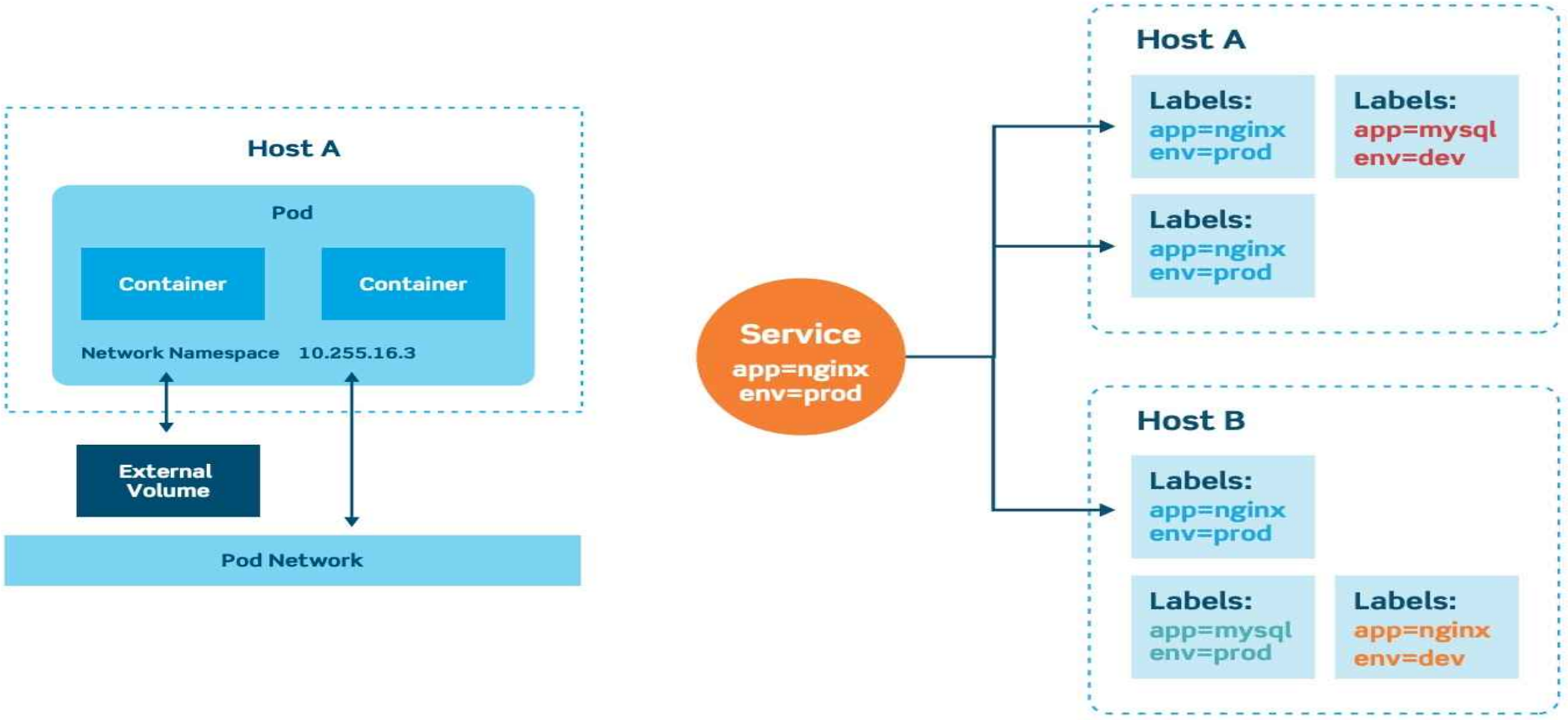

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-ingress-controller
  namespace: nginx-ingress
spec:
  clusterIP: 10.245.242.174
  externalIPs:
  - 200.197.226.18
  externalTrafficPolicy: Cluster
  ports:
  - name: http
    nodePort: 32544
    port: 80
    protocol: TCP
    targetPort: http
  selector:
    app: nginx-ingress
  type: LoadBalancer
```

Trabalhando com objetos Kubernetes




```
apiVersion: v1
kind: Service
metadata:
  name: headless-service
spec:
  clusterIP: None # <-- HeadLess
  selector:
    app: api
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
```

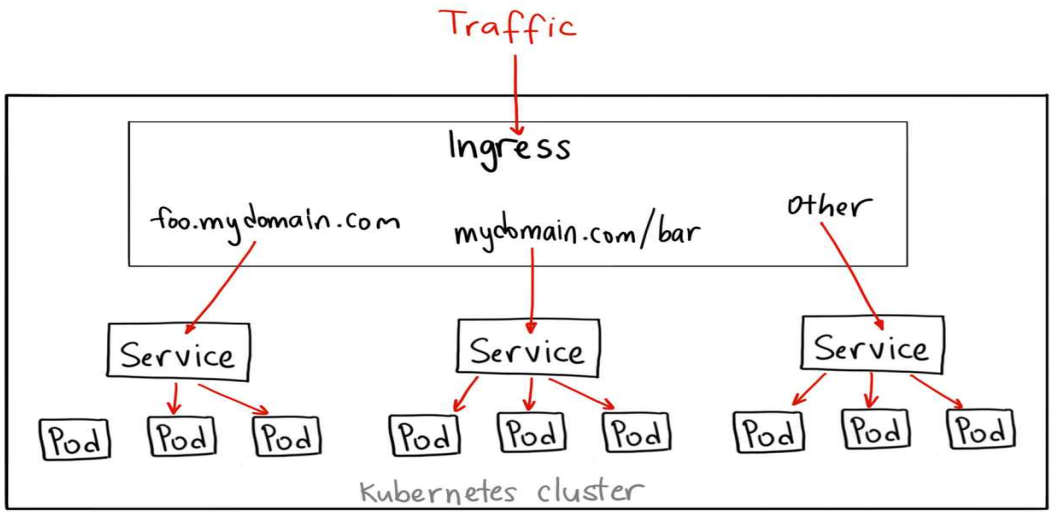
Trabalhando com objetos Kubernetes



Trabalhando com objetos Kubernetes

Ingress

Podemos pensar no ingress como um LoadBalancer que fica na frente do Service, diferente do Service LoadBalancer, que depende de um objeto externo ao Cluster para balanceamento L4, o Ingress roda dentro do Cluster e é um balanceador L7.




```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
 name: my-ingress
spec:
 backend:
   serviceName: other
   servicePort: 8080
 rules:
 - host: foo.mydomain.com
   http:
     paths:
     - backend:
         serviceName: foo
         servicePort: 8080
 - host: mydomain.com
   http:
     paths:
     - path: /bar/*
       backend:
         serviceName: bar
         servicePort: 8080
```

Trabalhando com objetos Kubernetes

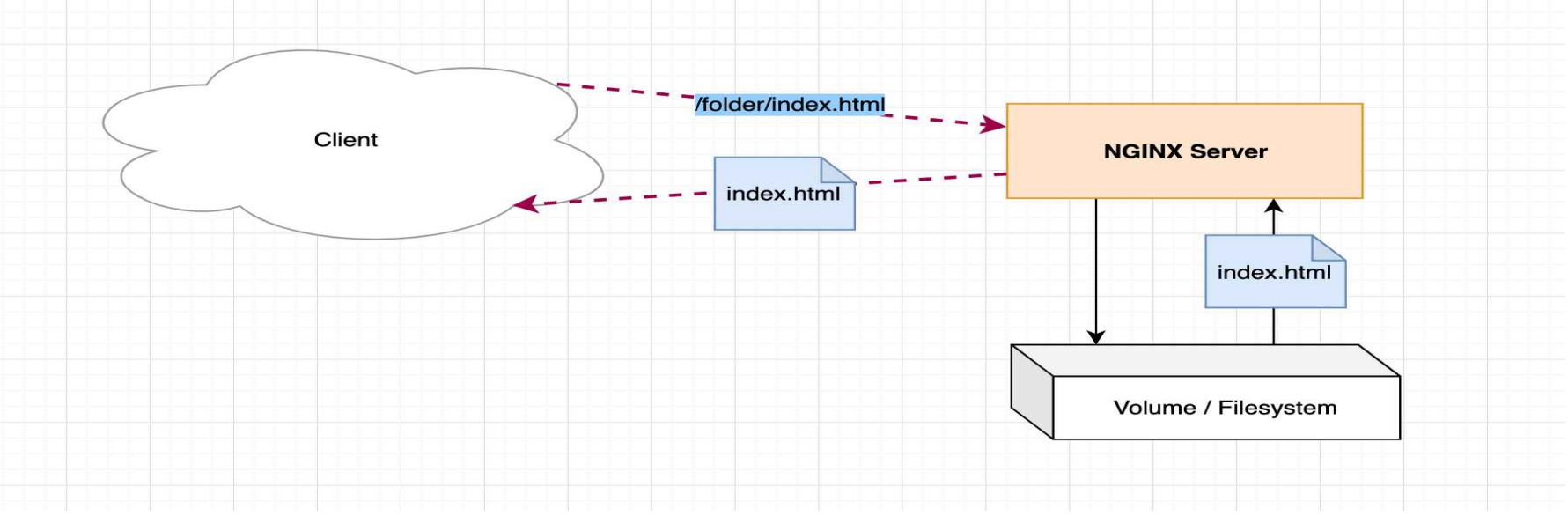
Ingress

Como já vimos, o Ingress não é um Kubernetes Service, ele simplesmente é um POD (ex: Nginx) que redireciona todas as requisições para um outro service interno (ClusterIP). Este POD geralmente é exposto por NodePort ou Service Type LoadBalancer.

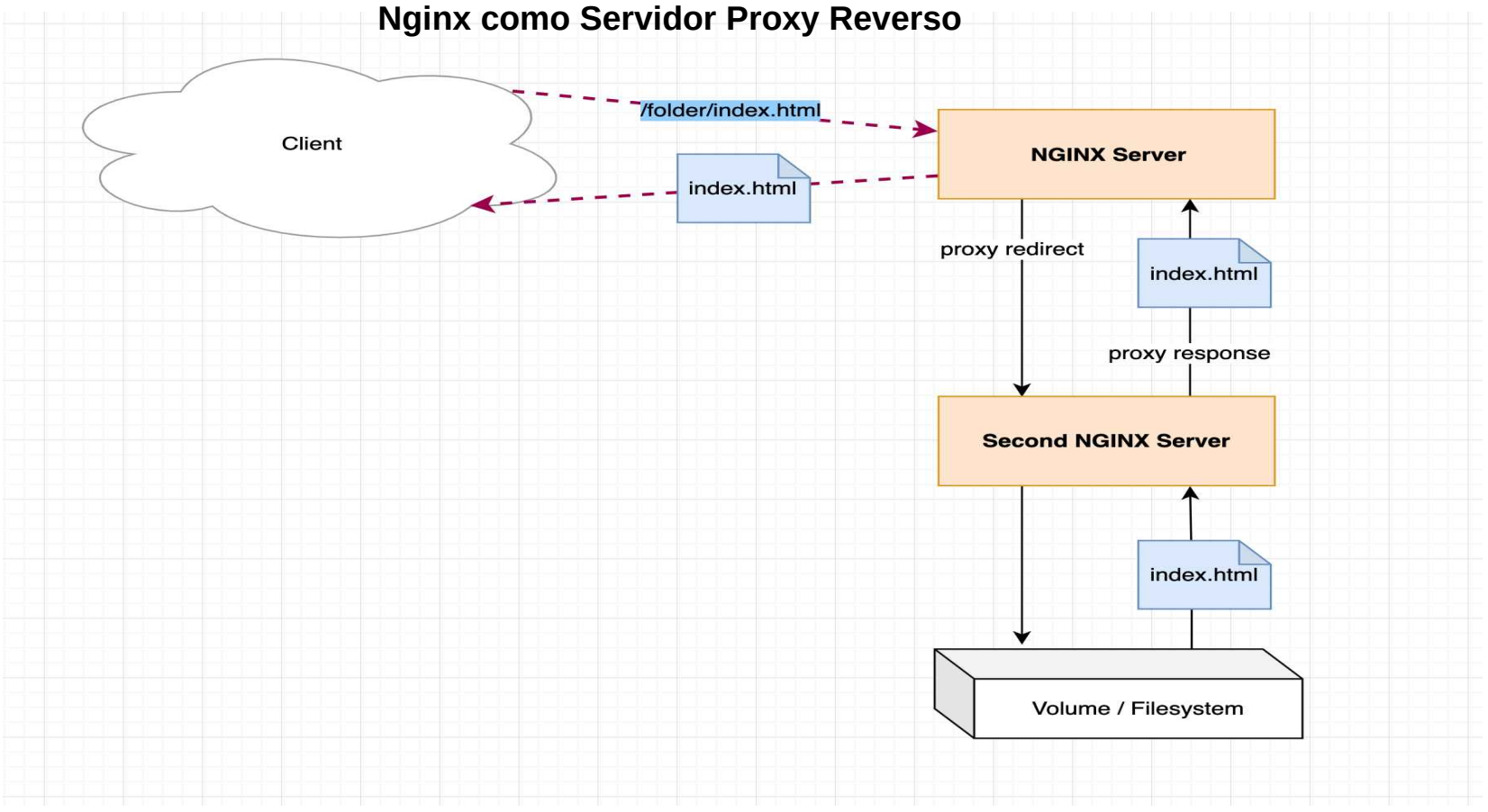
O uso mais comum do Ingress além de obviamente servir para expor um serviço interno do cluster, é para economizar LBs (L4), visto que basta expor para um único LB L4 e trabalhar com roteamento L7 utilizando Virtual Service e SNI.

Trabalhando com objetos Kubernetes

Nginx como Servidor Web

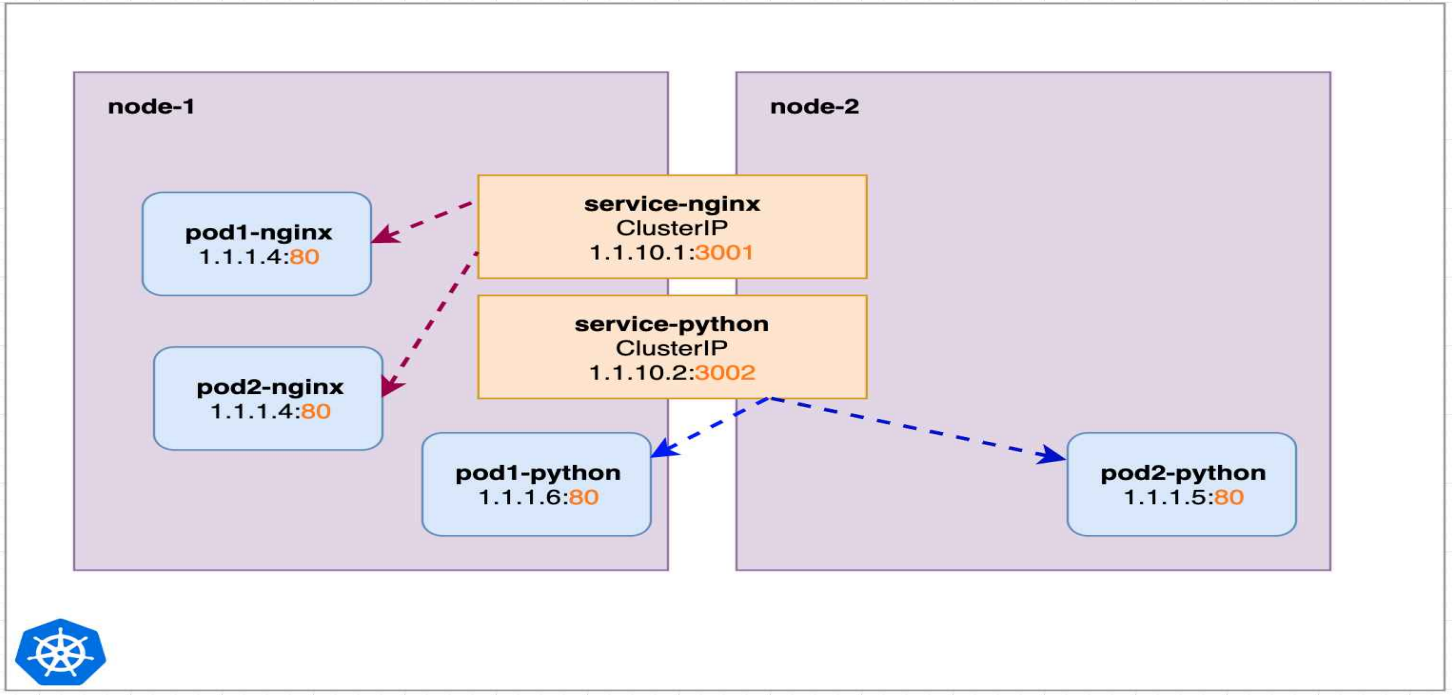


Trabalhando com objetos Kubernetes



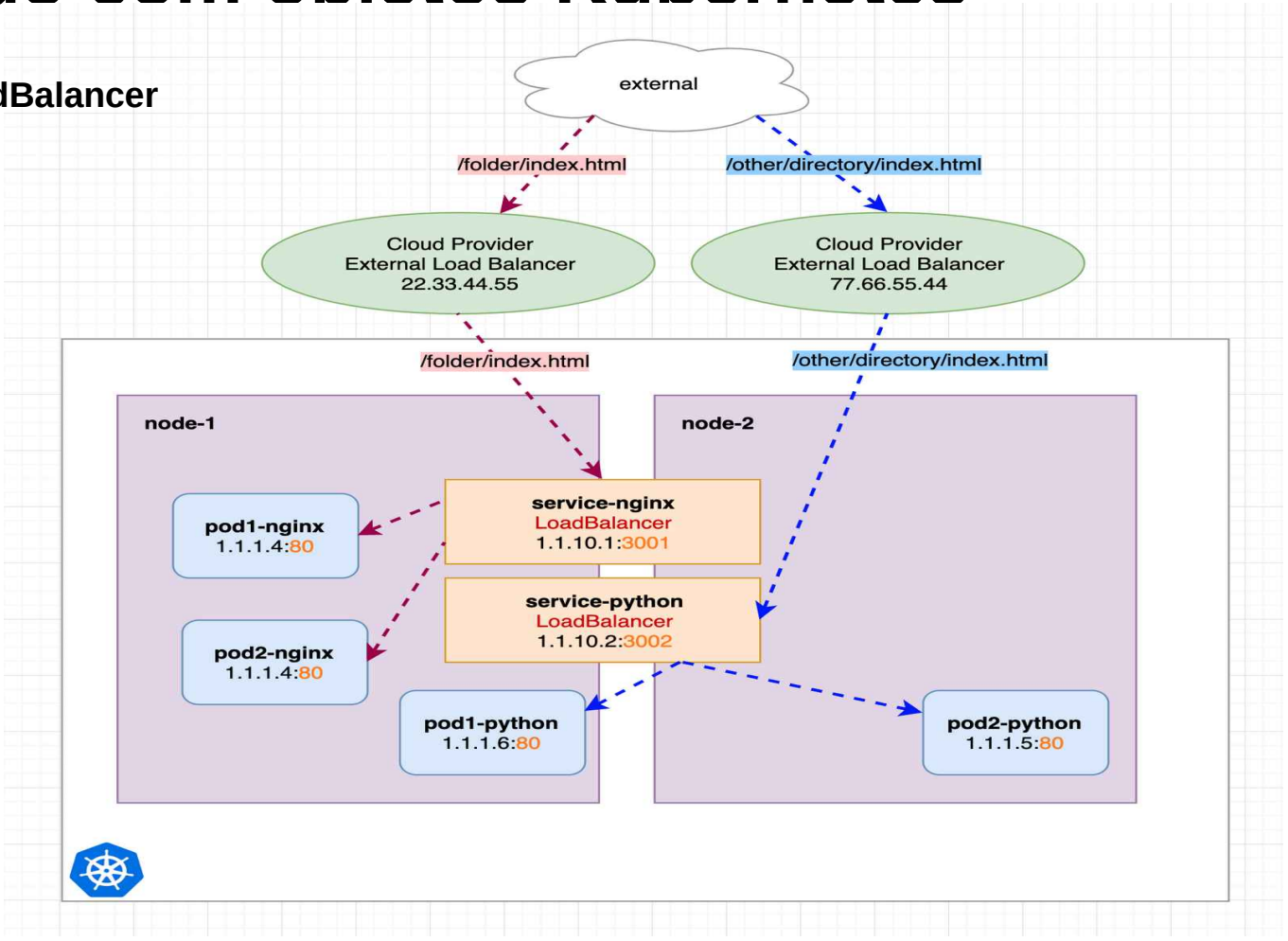
Trabalhando com objetos Kubernetes

Serviço de ClusterIP do Kubernetes



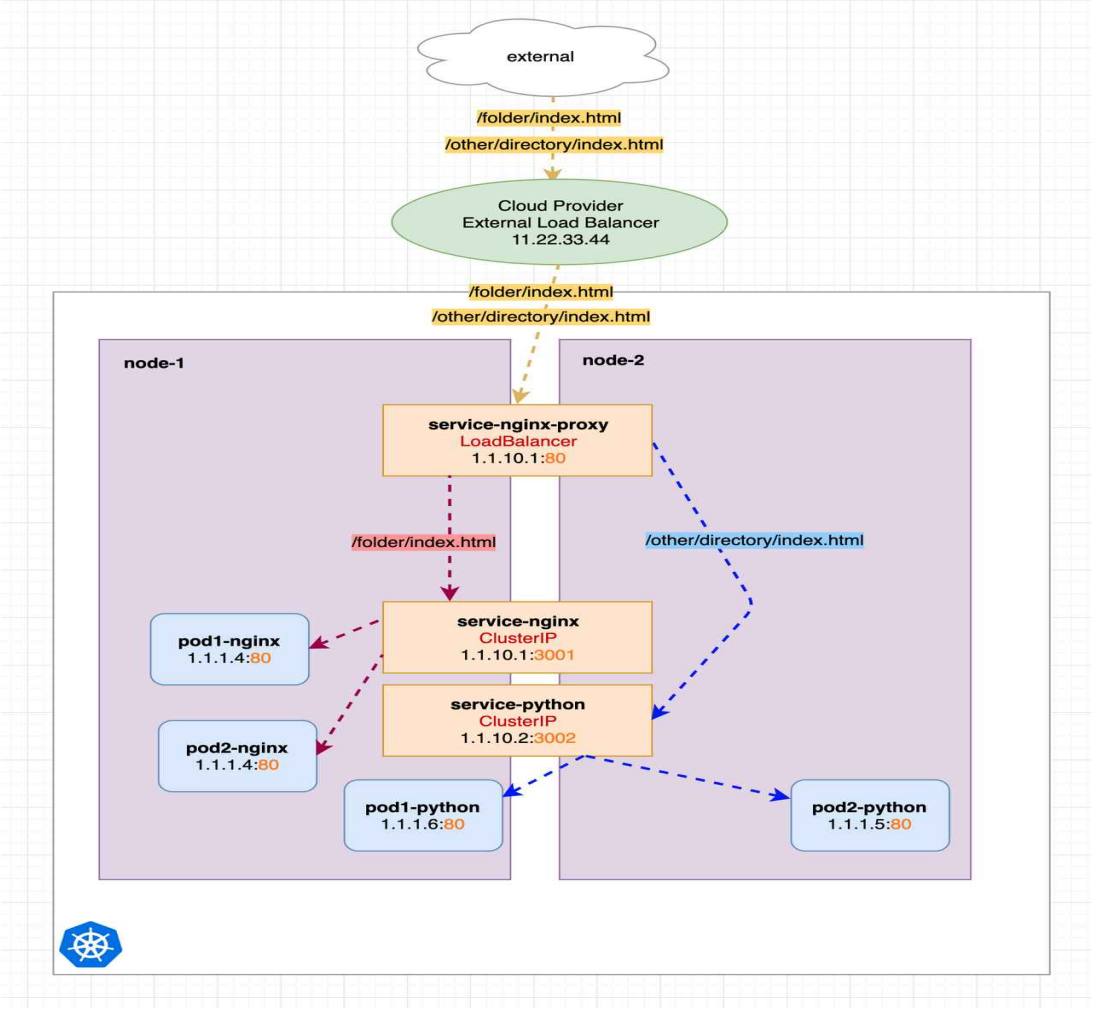
Trabalhando com obietos Kubernetes

Usando o Serviço LoadBalancer do Kubernetes



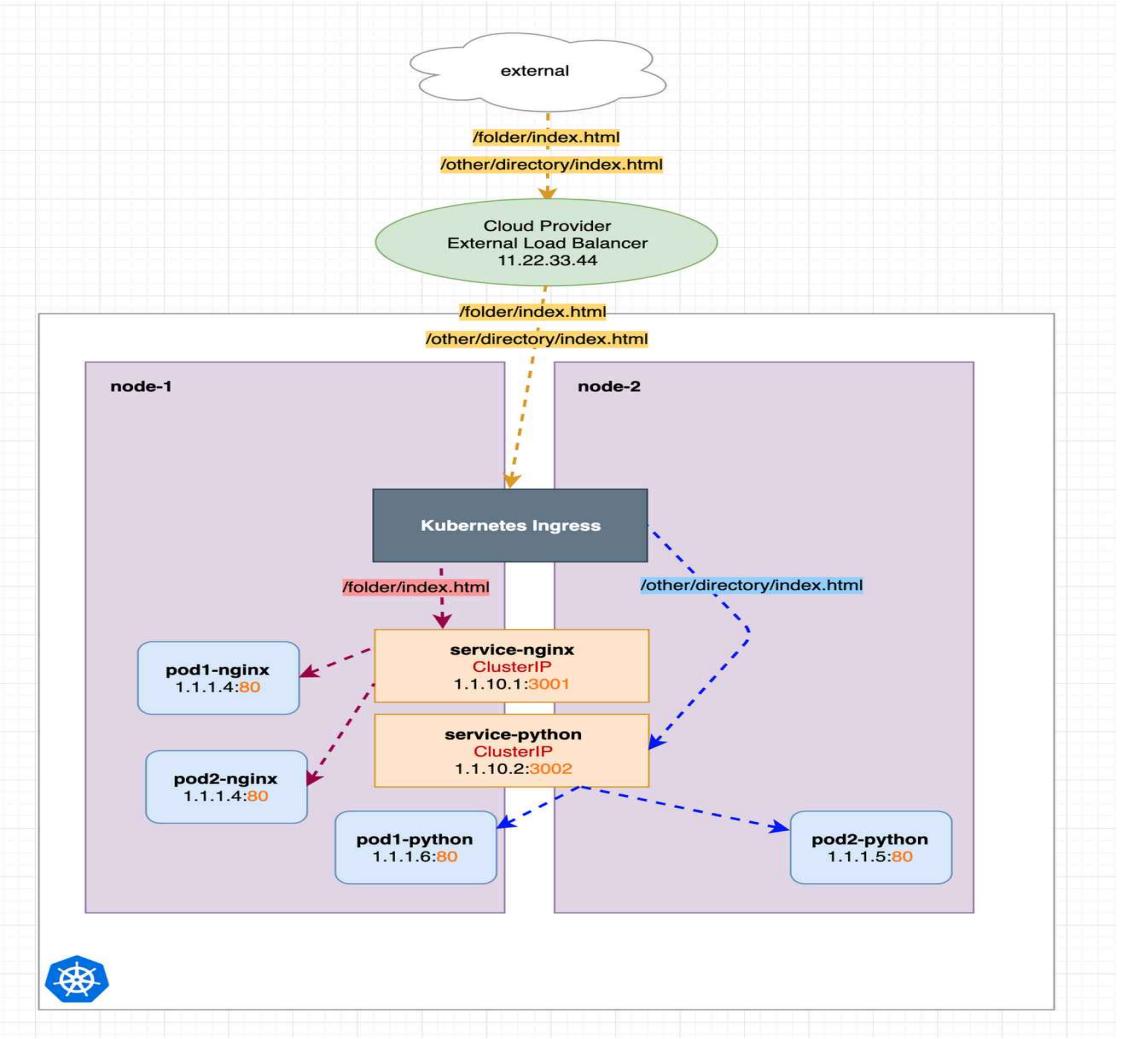
Trabalhando com objetos Kubernetes

Configurando Manualmente um Serviço Nginx para atuar como Proxy



Trabalhando com objetos Kubernetes

Kubernetes Ingress em ação



Trabalhando com objetos Kubernetes

Labels e Selectors

Labels: São pares chave/valor associados a objetos Kubernetes. Labels servem para identificarmos objetos, que a posteriori serão utilizados para por exemplo conectar recursos.

```
apiVersion: v1
kind: Deployment
metadata:
  labels:
    app: demo
```

Trabalhando com objetos Kubernetes

Labels e Selectors

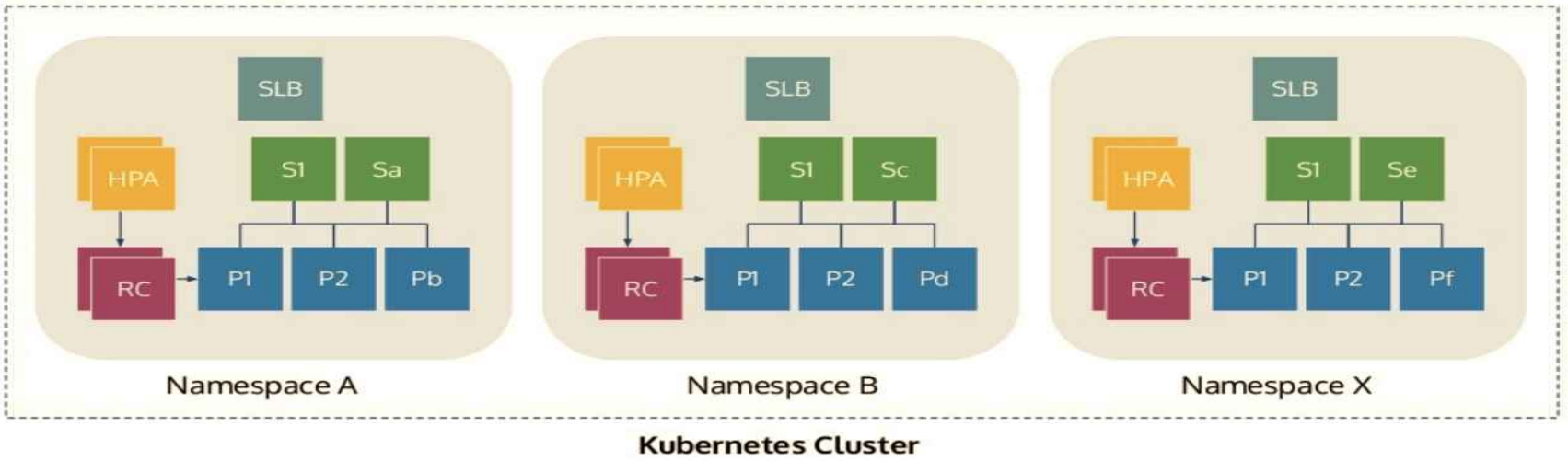
Selectors: É uma expressão que faz correspondência a um Label ou um conjunto deles.

```
apiVersion: v1
kind: Service
...
spec:
...
    selector:
        app: demo
```

Trabalhando com objetos Kubernetes

Namespaces

Os recursos do Kubernetes, como pods e Deployments, são logicamente agrupados em um namespace. Esses agrupamentos fornecem uma maneira de dividir logicamente um cluster e restringir o acesso para criar, exibir ou gerenciar recursos. Você pode criar namespaces para separar grupos de negócios, por exemplo. Os usuários podem interagir apenas com recursos dentro de seus namespaces atribuídos.



Variáveis, Configurações e dados Sigilosos

ConfigMaps

É um dicionário de configurações, que consiste em pares chave-valor de strings. O Kubernetes provê estes valores para seus containers.
São usados principalmente para manter o código da aplicação separado da configuração, permitindo mudar a configuração durante o Run-Time.

Secrets

Funcionam de forma semelhante ao ConfigMaps, porém são objetos utilizados para guardar dados sensíveis, como senhas e access keys.
Secrets são ofuscados, que significa que não são exibidos pelo comando kubectl describe, ou em mensagens de log ou terminal.

Variáveis, Configurações e dados Sigilosos

Variáveis de Ambiente

```
env:
    - name: APP_COLLOR
      value: pink

env:
    - name: APP_COLLOR
      valueFrom:
        configMapKeyRef:
          key:
          name:

env:
    - name: APP_COLLOR
      valueFrom:
        secretKeyRef:
          key:
          name:
```

Afinidade de Nós

Utilizamos o recurso de node affinity, quando temos a necessidade de fazer um pod ser escalonado em um determinado nó, temos dois tipos de afinidades, hard, onde o POD ficará aguardando o Nó disponível para escalar, ou soft onde o POD dará a preferência pelo nó caso disponível. O nodeAffinity é um atributo sofisticado com possibilidade de combinações de labels e restrições lógicas.

1) requiredDuringSchedulingIgnoredDuringExecution (hard)
2) preferredDuringSchedulingIgnoredDuringExecution (soft)

Vale lembrar que possuímos atributos mais simples como o nodeName, onde o “spec” do Pod recebe diretamente nome do nó onde o pod será instanciado e o nodeSelector, onde podemos utilizar uma simples label para localizar o nó desejado.

Pod Affinities e Anti-Affinities

É exatamente a mesma idéia da Afinidade de Nós, porém agora é no âmbito de PODs, servem para quando necessitamos deixar PODs juntos no mesmo Nó ou separados Nós diferentes.

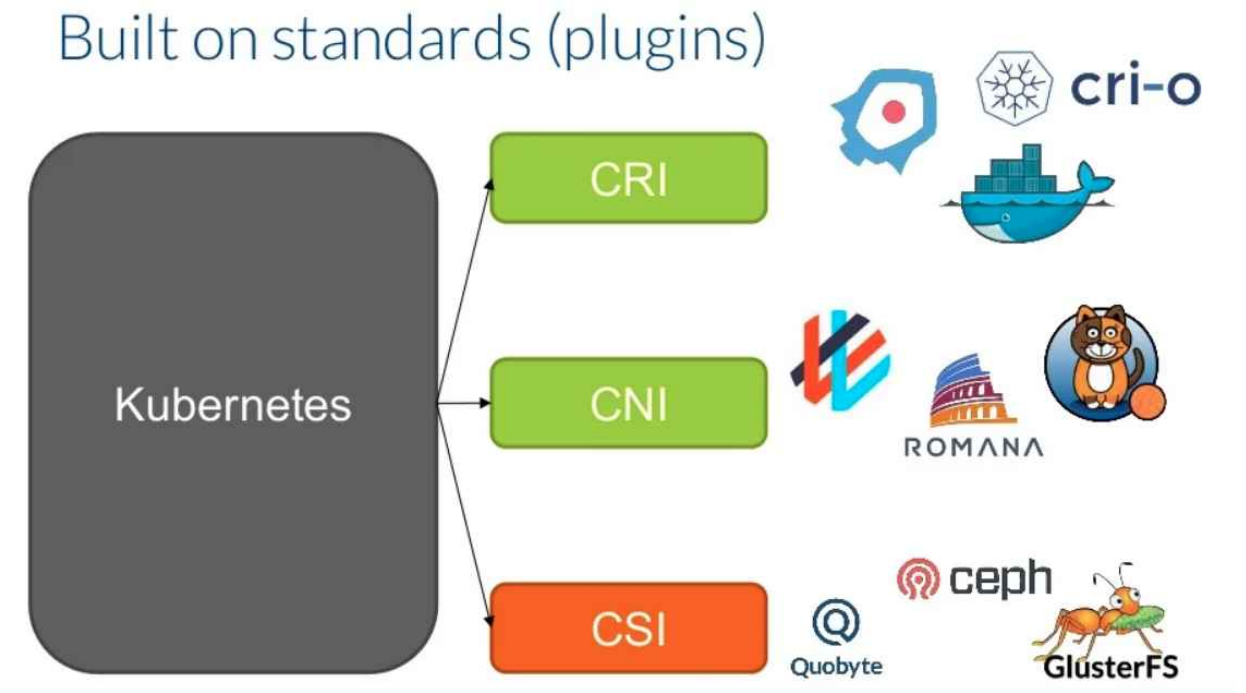
1) requiredDuringSchedulingIgnoredDuringExecution (hard)
2) preferredDuringSchedulingIgnoredDuringExecution (soft)

Taints and Tolerations

Quando colocamos um taint em um nó, **removemos a capacidade** de escalonamento do mesmo, onde apenas os pods que possuem **tolerations** poderão ser escalados neste nó.

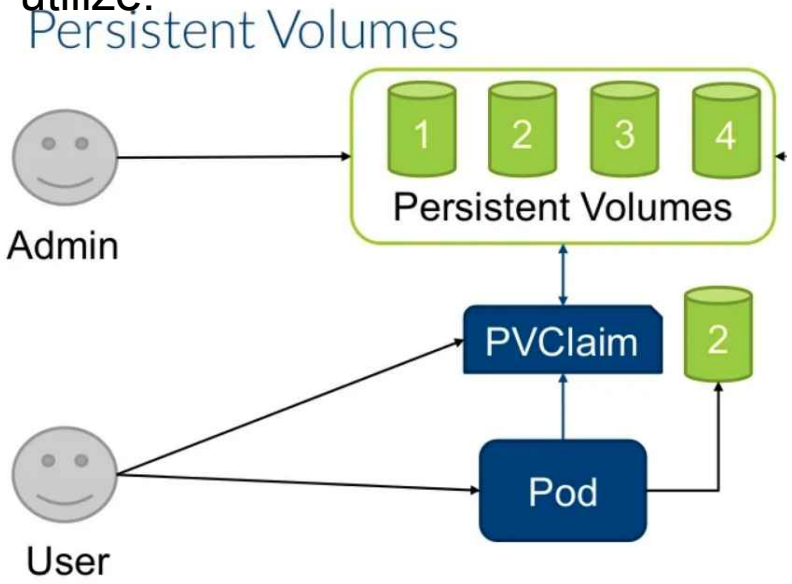
Se combinarmos o Node Affinity / Selector com Taints e Tolerations, podemos ter Nós exclusivos para estes PODs.

Persistência de Volumes - CSI

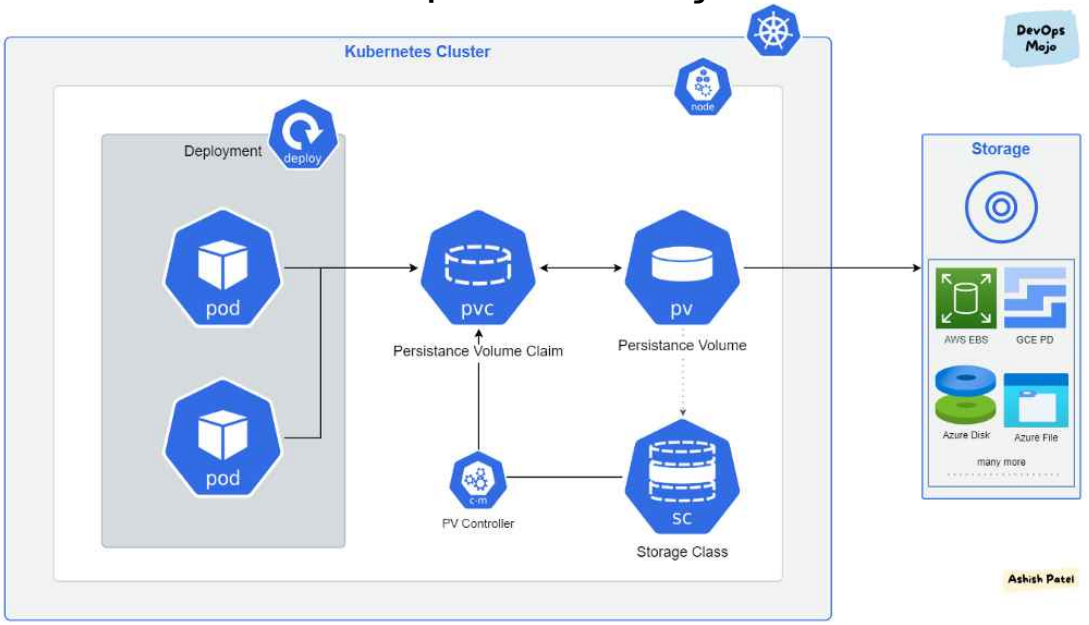


Persistência de Volumes – PV, PVC e SC

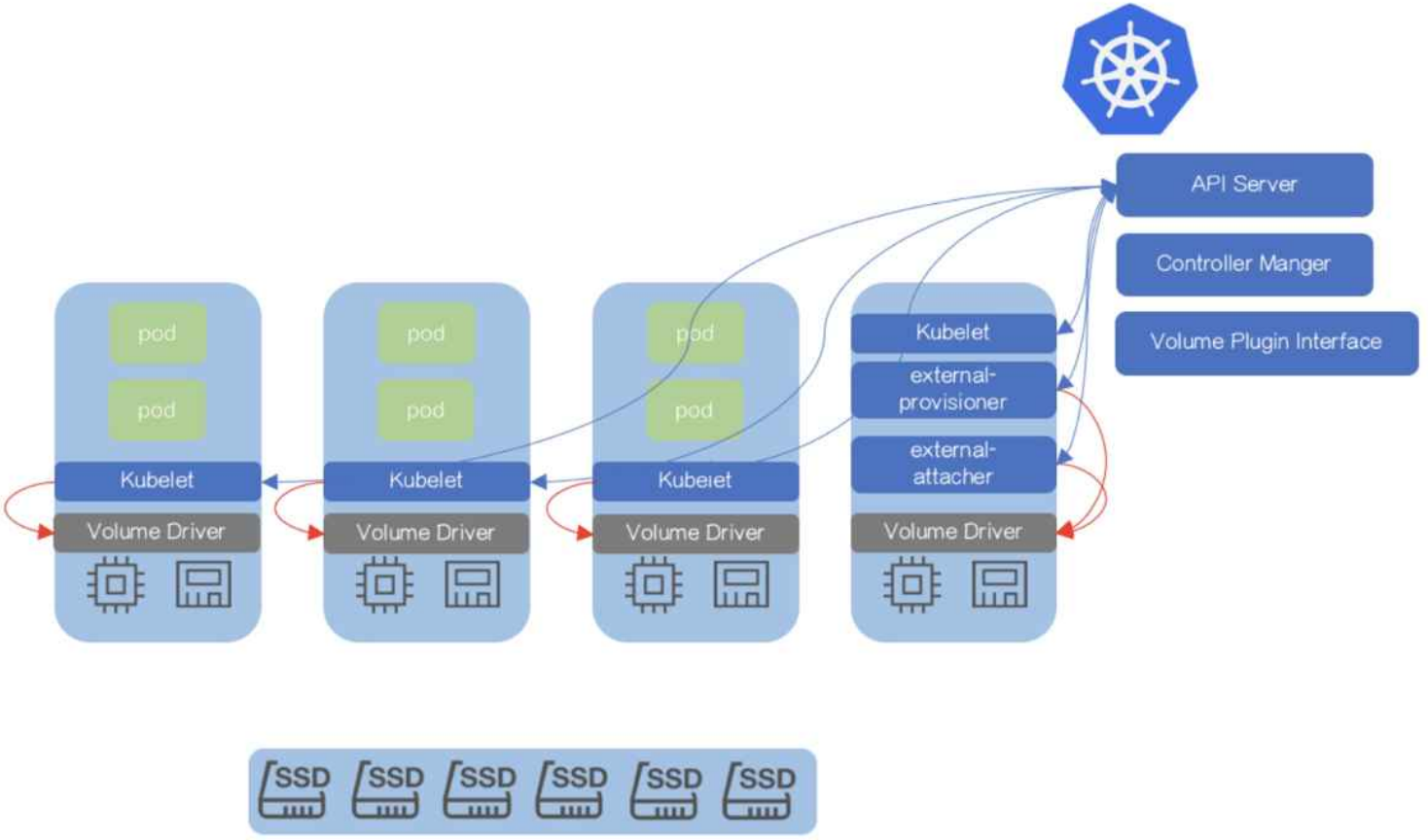
PVs são Estáticos, ou seja, precisam estar alocados para que um PVC o utilize.



SCs são Dinâmicos, ou seja criam os
PVs em tempo de execução



Persistência de Volumes – Fluxo de um SC



Persistência de Volumes

PersitentVolumes (PV)
São usados para gerenciar o armazenamento **durável** em um cluster. Este objeto pertence ao **cluster**.

kubectl get pv

Exemplos de PVs:

PersitentVolumeClaims (PVC)
É uma solicitação e uma declaração de um **PersistentVolume**. Este objeto pertence ao **namespace**.

kubectl get pvc -n <NAME SPACE>

StorageClass
É o provisionamento dinâmico de **PersitentVolumes.**

kubectl get sc

Persistência de Volumes

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /tmp
    server: 172.17.0.2
```

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 8Gi
  storageClassName: slow
  selector:
    matchLabels:
      release: "stable"
    matchExpressions:
      - {key: environment, operator: In, values: [dev]}
```

Persistência de Volumes

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: block-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
  persistentVolumeReclaimPolicy: Retain
  fc:
    targetWWNs: ["50060e801049cfd1"]
    lun: 0
    readOnly: false
```

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: block-pvc
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
  resources:
    requests:
      storage: 10Gi
```

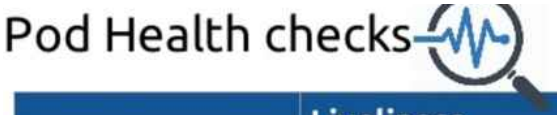
Health Check e Probes

Temos 3 Tipos de Probes no Kubernetes:

1. **Liveness**: Garantir que o POD esteja sempre rodando. A ação é reiniciar o pod.

2. **Readiness**: Garantir que o tráfego seja apenas direcionado para PODs saudáveis. A ação é redirecionar o tráfego para outro lugar.

3. **Startup**: Garantir que o POD tenha o tempo necessário para inicializar. A ação é aguardar o POD subir, útil para aplicações legadas.



	Liveliness	Readiness
On failure	Kill container	Stop sending traffic to pod
Check types	Http , exec , tcpSocket	Http , exec , tcpSocket
Declaration example (Pod.yaml)	livenessProbe: failureThreshold: 3 httpGet: path: /healthz port: 8080	readinessProbe: httpGet: path: /status port: 8080

```yaml
spec:
  containers:
  - name: goproxy
    image: k8s.gcr.io/goproxy:0.1
    ports:
    - containerPort: 8080
    readinessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 10
    livenessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 20
```

Health Check e Probes

Importante I: Liveness probes não esperam por readiness probes terem sucesso. Se você quer que liveness probe espere a inicialização da aplicação, utilize o parâmetro ***initialDelaySeconds*** ou ***startupProbe***.

Importante II : A implementação incorreta de readiness probes, pode resultar num crescente número de processos no container, causando starvation.

Fonte: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/