Kubernetes Observability and Pod Design Continued

Starts at 9:05 pm

# Agenda

1. Node Selector and Labels

2. Affinity and Anti Affinity

3. Probes in kubernetes

4. Metrics Server

5. Deployment Strategies

→ Node selectors and Labels

Nodes → labels

nodeselector → pod spec.

```
kubectl label nodes new-cluster1-worker disktype=ssd region=us-west

kubectl label nodes new-cluster1-worker2 disktype=hdd region=us-east
```

```
apiVersion: v1
kind: Pod
metadata:
```

```yaml
  name: multi-label-selector-pod
spec:
  nodeSelector:
    disktype: ssd
    region: us-west
  containers:
  - name: example-container
    image: nginx
    command: ["sh", "-c", "echo Hello from us-west SSD node!"]
```

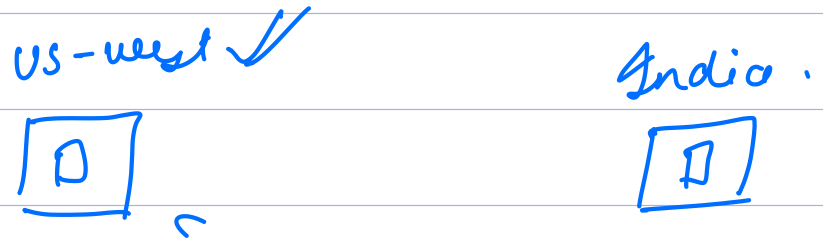
Removing Labels from the nodes

```
kubectl label nodes new-cluster1-worker2 disktype- region-
```

Removing Labels from the nodes

```
kubectl label nodes new-cluster1-worker disktype- region-
```

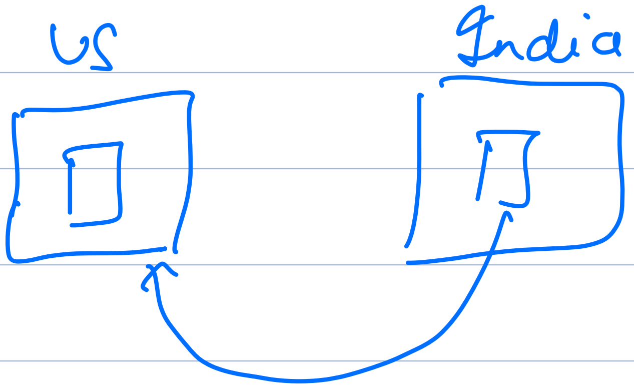
→ Affinity and Anti Affinity



① Node Affinity → Matching labels.

② Pod Affinity

③ Pod Antiaffinity.



Node Affinity: Rules to ensure pods are scheduled on nodes with specific labels.

- **Pod Affinity**: Rules to co-locate pods near other pods based on their labels.

- **Pod Anti-Affinity**: Rules to prevent pods from being scheduled on the same nodes as other pods with specific labels.

Node Affinity.

1. **RequiredDuringSchedulingIgnoredDuringExecution**:

   - The pod **can only be scheduled** on nodes that **match** the affinity rules specified.

      - If no node matches the rules, the pod will **not be scheduled**.

      - This is a **hard requirement** for scheduling.

1. **PreferredDuringSchedulingIgnoredDuringExecution**:

- The pod **prefers** to be scheduled on nodes that **match** the affinity rules specified, but it is not a hard requirement.
    - If no nodes match, the pod can still be scheduled on any available node.
    - This is a **soft preference** that Kubernetes will try to honor but will not block scheduling if no matching node is found.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
                - new-cluster1-worker
                - new-cluster1-worker2
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 1
          preference:
            matchExpressions:
              - key: environment
                operator: In
                values:
                  - production
  containers:
    - name: my-container
      image: nginx
```

Pod affinity.

Pod Affinity is a concept in Kubernetes that allows you to schedule pods onto nodes based on the **labels of other pods** running on the same or different nodes.

This is useful in scenarios where you want to keep related pods together, for example:

- Pods in the same application stack should run on the same node to reduce latency.
- Running high-throughput services together on the same node to improve network performance.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
```

```
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchLabels:
              app: my-app
          topologyKey: kubernetes.io/hostname
  containers:
    - name: my-container
      image: nginx
```

Pod Anti Affinity

Pod anti-affinity ensures that pods are **spread out** and not scheduled on the same nodes as other pods with specific labels. Useful for:

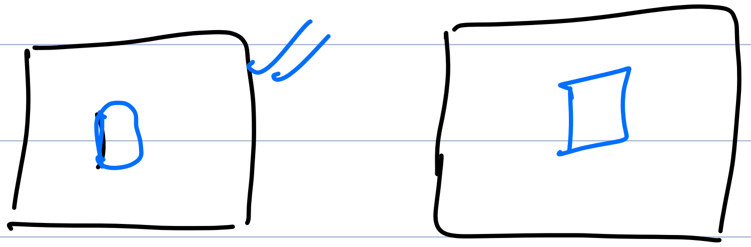
```
apiVersion: v1
kind: Pod
metadata:
  name: pod-anti-affinity-pod
  labels:
    app: my-app
```

```
spec:
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchLabels:
              app: my-app
          topologyKey: "kubernetes.io/hostname"
  containers:
    - name: nginx
      image: nginx
```

Combined use

```
apiVersion: v1
kind: Pod
metadata:
  name: combined-affinity-pod
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
```

```
        - matchExpressions:
          - key: disk-type
            operator: In
            values:
            - ssd
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        labelSelector:
          matchLabels:
            app: backend
        topologyKey: "kubernetes.io/hostname"
    podAntiAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 1
          podAffinityTerm:
            labelSelector:
              matchLabels:
                app: frontend
            topologyKey: "kubernetes.io/hostname"
  containers:
  - name: nginx
    image: nginx
```

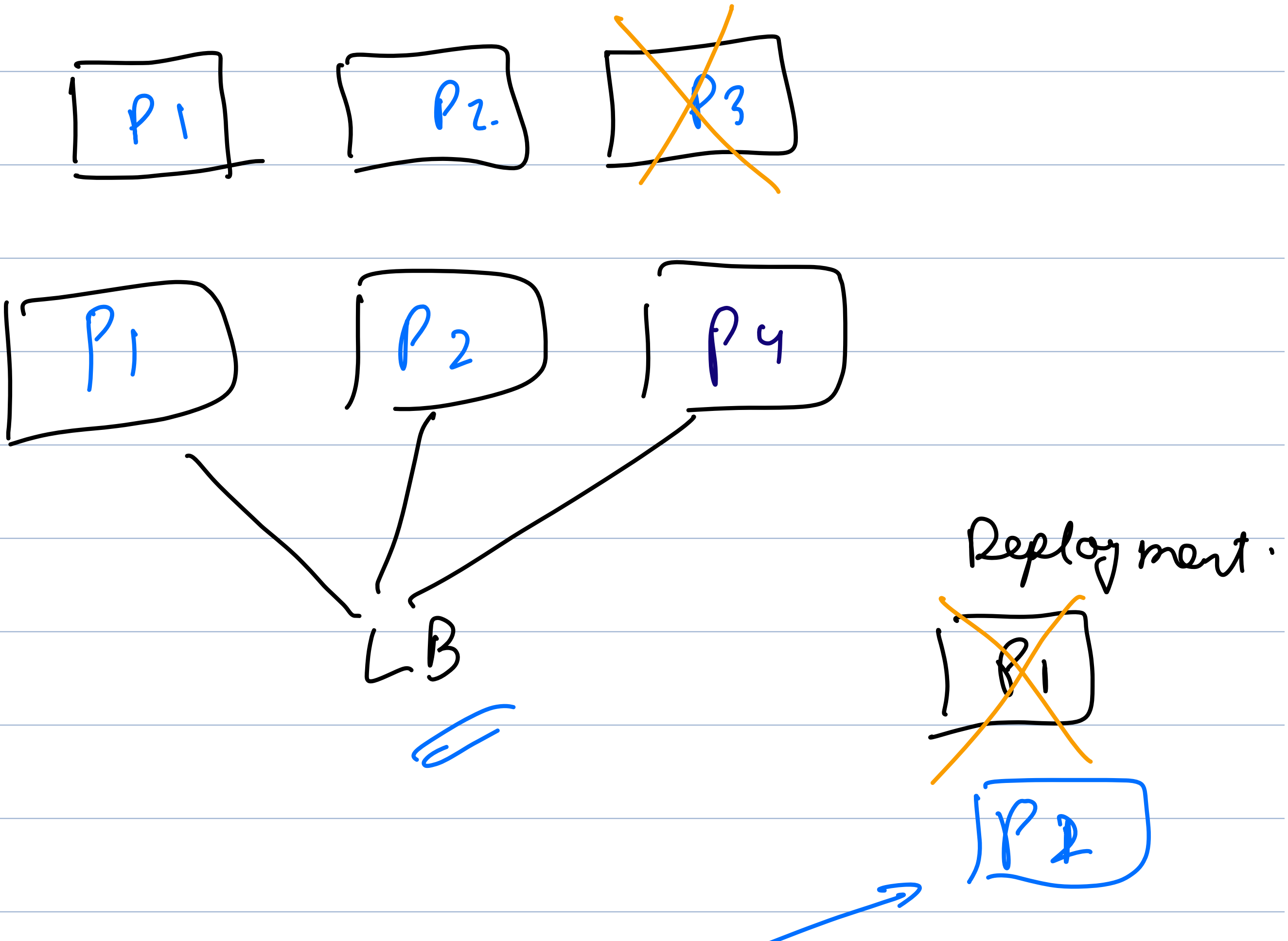


→ Probes

1. **Liveness Probe** – Checks if the container is still running. If the probe fails, Kubernetes restarts the container.

2. **Readiness Probe** – Checks if the container is ready to serve traffic. If the probe fails, the Pod is removed from the Service endpoints.

3. **Startup Probe** – Used for slow-starting applications. It prevents Kubernetes from marking a container as failed before it has fully started.



→ endpoint (backend = deployment)

↓

service.

Break → 10:21 pm

→ Liveness Probe

http://<pod-ip>:80/health

↓

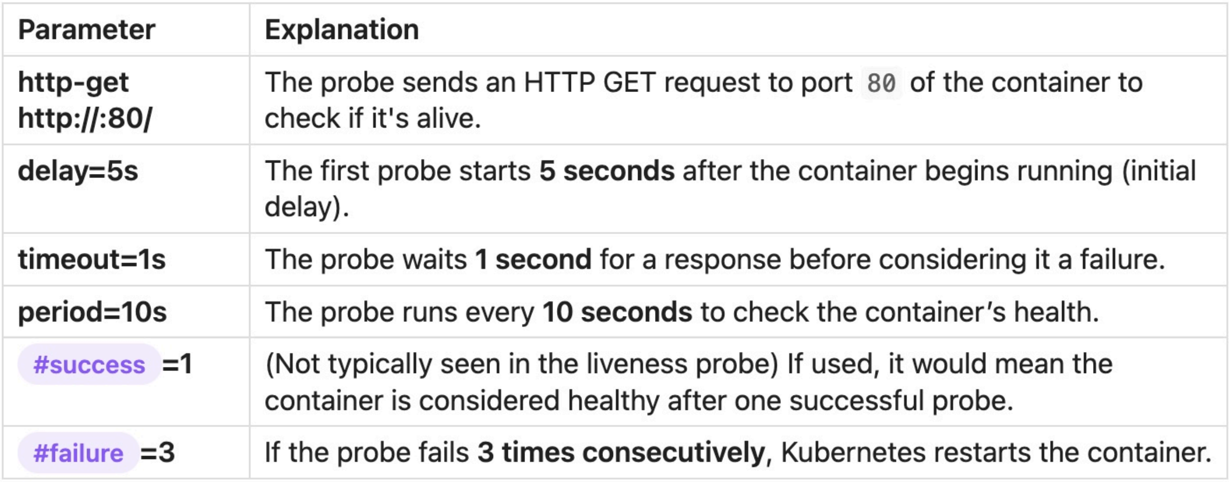
404

http://<pod-ip>:80

↓

200

Liveness: http-get http://:80/ delay=5s timeout=1s period=10s #success=1 #failure=3



Parameter	Explanation
http-get http://:80/	The probe sends an HTTP GET request to port `80` of the container to check if it's alive.
delay=5s	The first probe starts **5 seconds** after the container begins running (initial delay).
timeout=1s	The probe waits **1 second** for a response before considering it a failure.
period=10s	The probe runs every **10 seconds** to check the container's health.
#success =1	(Not typically seen in the liveness probe) If used, it would mean the container is considered healthy after one successful probe.
#failure =3	If the probe fails **3 times consecutively**, Kubernetes restarts the container.

```
    livenessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 5
      timeoutSeconds: 2
      periodSeconds: 10
      failureThreshold: 5   # Custom value, default is 3
      successThreshold: 2   # Custom value, default is 1
```

Use Cases of Liveness Probe

- **Crash Recovery**: If a container crashes due to an application error or other issue, the liveness probe can detect this and restart the container automatically.
- **Deadlocks and Stuck Processes**: If an application inside a container becomes unresponsive due to a deadlock or gets stuck, the liveness probe can restart the container to recover from this state.
- **Ensuring High Availability**: By continuously monitoring the health of containers, liveness probes help maintain the high availability and reliability of applications.

Use Cases of Readiness Probe

- **Rolling Updates**: During rolling updates, readiness probes help ensure that only the updated and ready instances of a container serve traffic. If a container fails the readiness check, it won’t receive any traffic until it passes the check.

- **Graceful Shutdowns**: Readiness probes can be used to remove a container from service endpoints before it is terminated, ensuring that no new traffic is sent to a container that is shutting down.

Readiness Probe

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: liveness-readiness-pod
spec:
  containers:
  - name: my-app
    image: nginx
    ports:
    - containerPort: 80
    livenessProbe:
      httpGet:
        path: /health
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 10
    readinessProbe:
      httpGet:
        path: /health
        port: 80
      initialDelaySeconds: 3
      periodSeconds: 5
```

Startup Probes.

Kubernetes waits for the startup probe to succeed before performing any liveness or readiness probes.

Startup Probe Fields:

- **`httpGet`**: Performs an HTTP GET request to a specified path and port.

- **`tcpSocket`**: Checks whether the container is listening on a specific port.

- **`exec`**: Runs a command inside the container to check if the application is ready.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
```

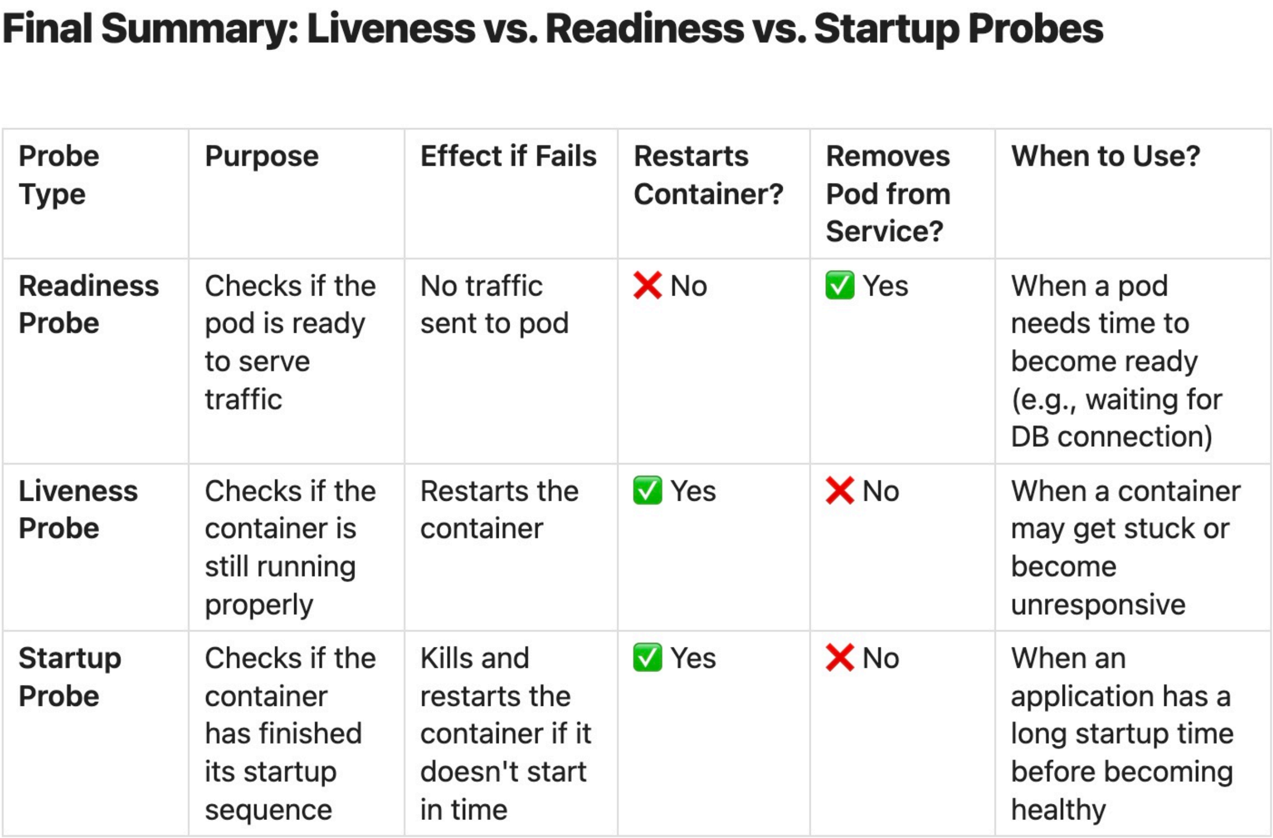
```
containers:
- name: mysql
  image: mysql:5.7
  env:
  - name: MYSQL_ROOT_PASSWORD
    value: rootpassword
  startupProbe:
    exec:
      command:
        - "mysql"
        - "-uroot"
        - "-prootpassword"
        - "-e"
        - "SELECT 1;"
    initialDelaySeconds: 60  # Allow MySQL enough time to start
    periodSeconds: 5
    failureThreshold: 10
    timeoutSeconds: 3
  ports:
  - containerPort: 3306
```

Use Cases

- **Applications with Long Startup Times**: Some applications may take a long time to initialize. Startup probes ensure that these applications are not killed or marked as unready during their initialization phase.

- **Preventing Premature Failures**: By delaying liveness and readiness checks until the application has started, startup probes prevent premature failures and restarts.

- **Ensuring Smooth Startups**: Startup probes help in managing the startup sequence smoothly, especially for applications with complex initialization processes.



Final Summary: Liveness vs. Readiness vs. Startup Probes

Probe Type	Purpose	Effect if Fails	Restarts Container?	Removes Pod from Service?	When to Use?
Readiness Probe	Checks if the pod is ready to serve traffic	No traffic sent to pod	❌ No	✅ Yes	When a pod needs time to become ready (e.g., waiting for DB connection)
Liveness Probe	Checks if the container is still running properly	Restarts the container	✅ Yes	❌ No	When a container may get stuck or become unresponsive
Startup Probe	Checks if the container has finished its startup sequence	Kills and restarts the container if it doesn't start in time	✅ Yes	❌ No	When an application has a long startup time before becoming healthy

Metrics Server.

CPU RAM nodes and pods.

```
wget https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
```

```
spec:
  containers:
  - name: metrics-server
    args:
    - --cert-dir=/tmp
    - --secure-port=4443
    - --kubelet-insecure-tls    # Add this line to bypass certificate errors
```

```
kubectl top pod --all-namespaces
```

```
kubectl top node
```

→ Deployment Strategies

① Rolling update.

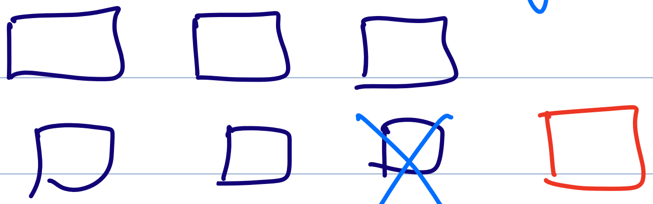
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
```

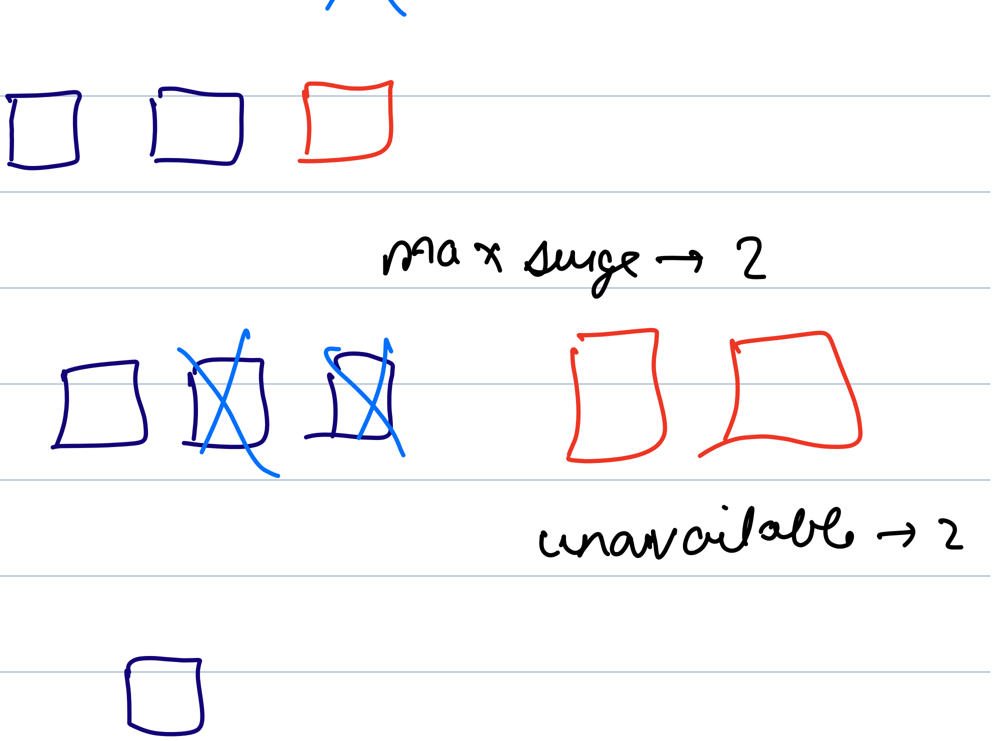
```
replicas: 3
selector:
  matchLabels:
    app: nginx
template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
    - name: nginx
      image: nginx:1.21
      ports:
      - containerPort: 80
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 2      # Allows two additional pod
    maxUnavailable: 2  # Allows two pod to be unavailable
```

default

max surge → 1

unavailable → 1

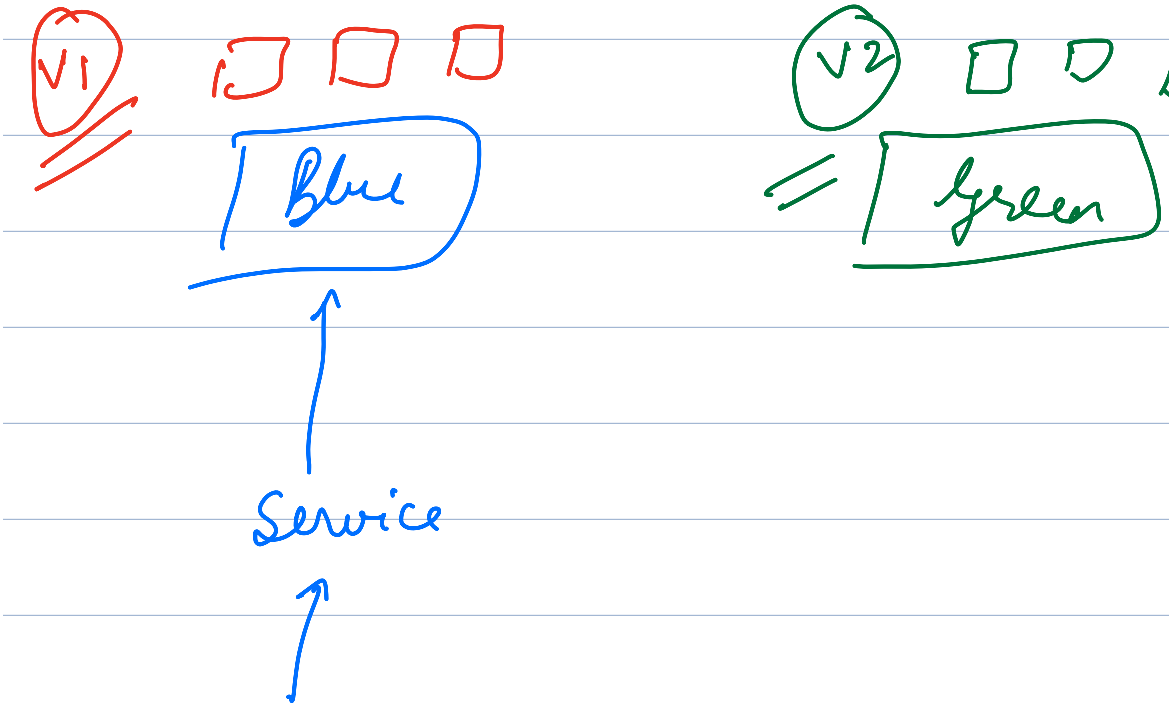


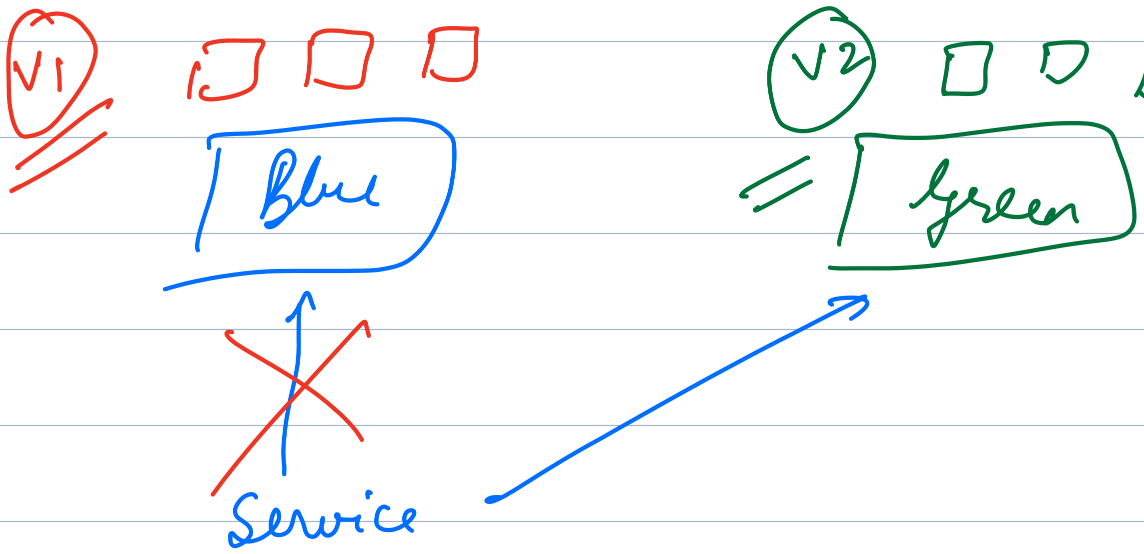


- **maxSurge**: This controls how many additional pods (above the desired replica count) can be created during an update. It allows extra pods to be spun up temporarily during the update.

- **maxUnavailable**: This controls how many pods (below the desired replica count) can be unavailable during an update. This ensures there is no sudden loss of available pods.

Blue - Green Deployment.

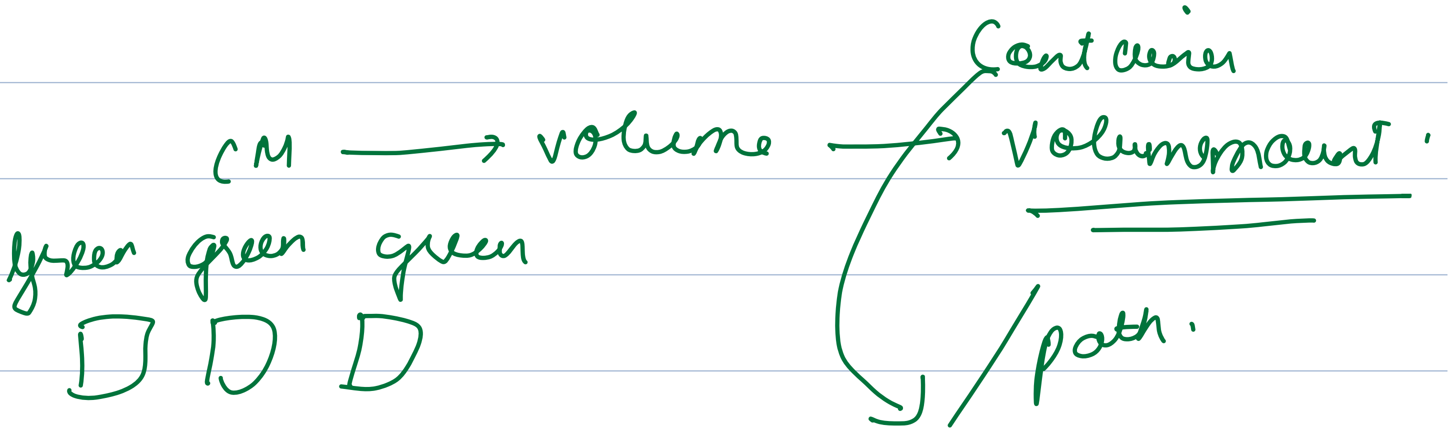


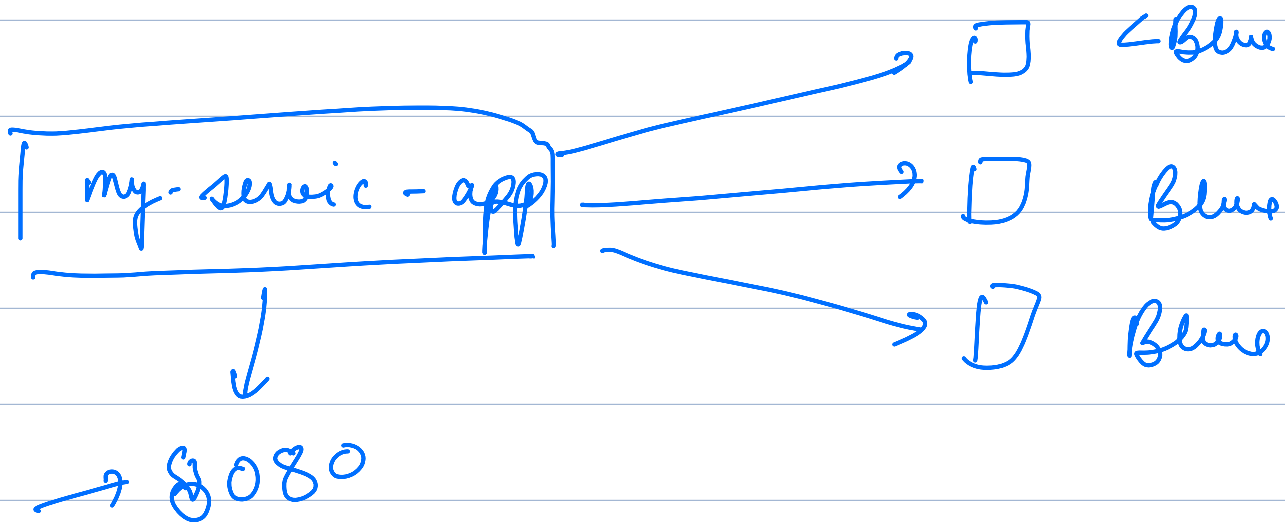


→ 2 CMS. → green.
→ Blue.

→ 2 deployments → green.
→ Blue.

→ 1 service.





curl machine :8080

Create configmaps

```
kubectl create configmap nginx-blue --from-literal=index.html='<h1>Blue</h1>'

kubectl create configmap nginx-green --from-literal=index.html='<h1>Green</h1>'
```

Blue Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  na
me: my-app-blue
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
      version: blue
  template:
    metadata:
      labels:
     app: my-app
        version: blue
    spec:
      containers:
        - name: nginx
          image: nginx:1.23
          volumeMounts:
            - name: nginx-config
              mountPath: /usr/share/nginx/html
      volumes:
        - name: nginx-config
          configMap:
            name: nginx-blue
```

```
kubectl expose deployment my-app-blue --port=80 --target-port=80 --name=my-app-service
```

```
kubectl port-forward svc/my-app-service 8080:80
```

Create Green Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-green
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
      version: green
  template:
    metadata:
      labels:
        app: my-app
        version: green
    spec:
      containers:
        - name: nginx
          image: nginx:1.24
          volumeMounts:
            - name: nginx-config
              mountPath: /usr/share/nginx/html
      volumes:

        - name: nginx-config

          configMap:

            name: nginx-green
```

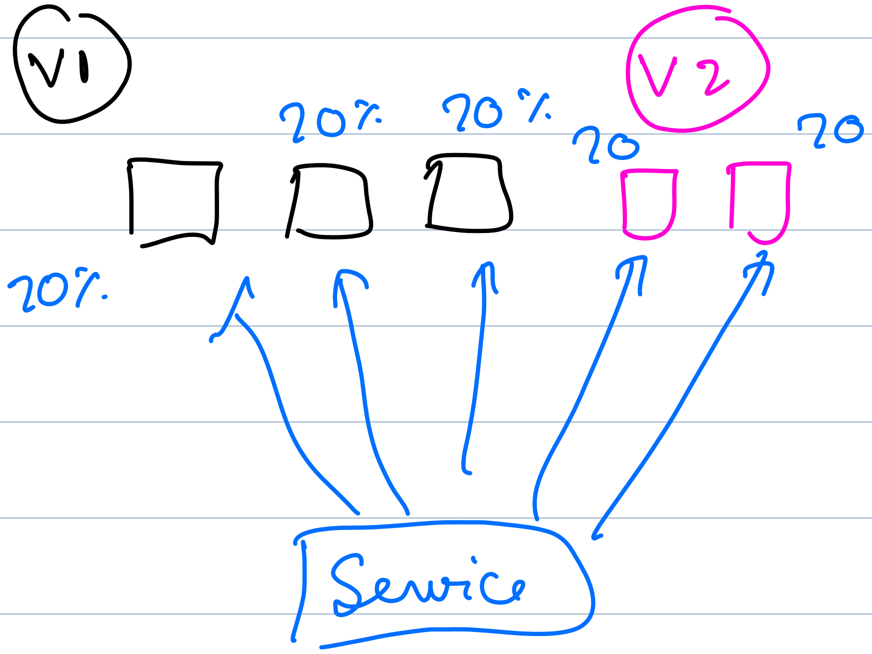
Patch the service to update the labels.

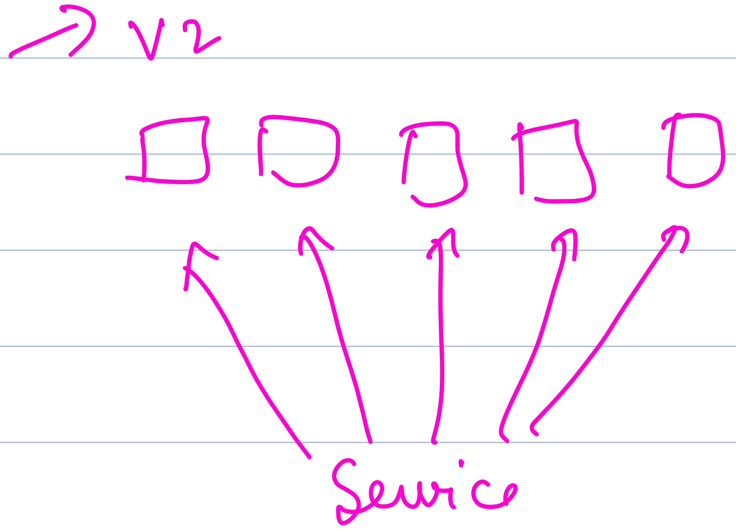
```
kubectl patch service my-app-service -p '{"spec":{"selector":{"app":"my-app","version":"green"}}}'
```

Again do port forwarding

```
kubectl port-forward svc/my-app-service 8080:80
```

Canary Deployment:



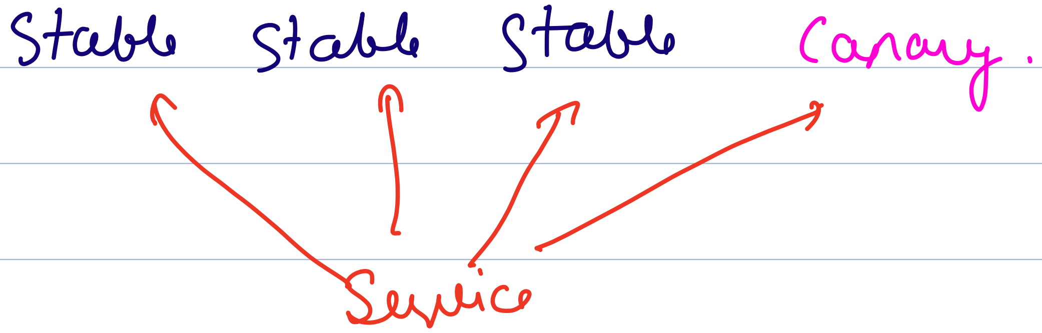


create 1 deploy. (stable)   3 replicas

create 1 deploy (canary)   1 replica.

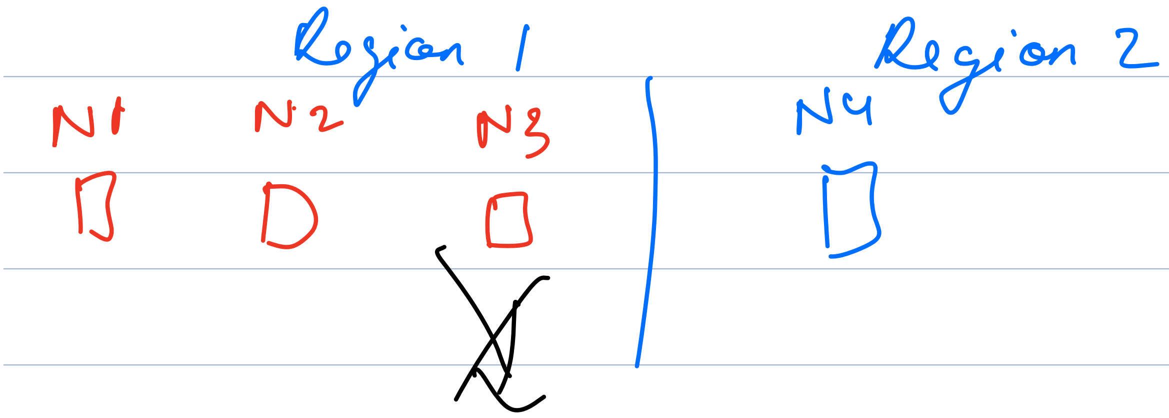
create service.

→ app = my-app.



Scale stable = 1

Scale canary = 3


Region 1
N1
N2
N3
Region 2
N4