Self-Healing Application

Self-healing applications in Kubernetes are applications that can detect and recover from failures automatically without human intervention. Kubernetes provides several mechanisms to enable self-healing, including probes, replica sets, and deployments. These components together ensure that the desired state of the application is maintained, even in the face of failures, updates, or changes in the environment.

Probes play a vital role in ensuring the health and availability of pods and containers running in a Kubernetes cluster. By periodically checking the health of containers, Kubernetes can take appropriate actions such as restarting containers, marking pods as ready to receive traffic, or delaying traffic until an application inside a container has started successfully

The main idea behind ReplicationControllers and Deployments in Kubernetes is to maintain a desired number of pod replicas running at any given time. In other words, they ensure that a particular pod (or set of pods) always remains up and running.

Kubernetes provides three main types of probes to check the health of Pods

The kubelet is responsible for running probes on containers to check their health

Liveness probes

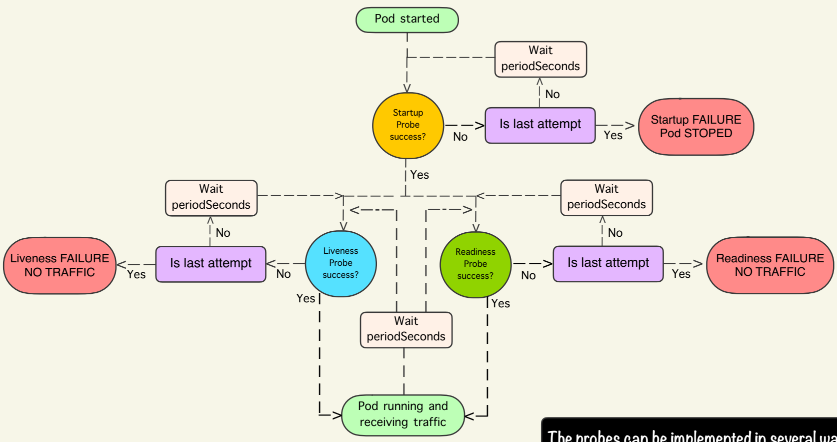
Kubernetes uses liveness probes to know when to restart a Container. For example, a liveness probe could catch a deadlock, where an application is running, but unable to make progress. Restarting a Container in such a state can help to make the application more available despite bugs.

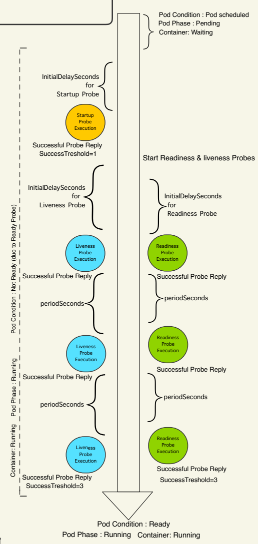
Readiness probes

Kubernetes 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.

Startup Probes

These probes let Kubernetes know when your application has started. If such a probe is configured, it disables liveness and readiness checks until it succeeds, making it useful for slow-starting containers.





The probes can be implemented in several ways

HTTP checks: Kubernetes sends an HTTP request to the specified path of your application. If the application responds with a success status code (200 - 399), the probe is successful. Otherwise, it's considered a failure

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
      initialDelaySeconds: 30
      periodSeconds: 10
```

The "initialDelaySeconds" field indicates that k8s should wait 30 seconds before checking the container's health for the first time

The "periodSeconds" field indicates that Kubernetes should check the container's health every 10 seconds thereafter

The Liveness Probe is configured to use an HTTP GET request to check the container's health. The request is sent to the path "/healthz" on port 8080, which is where the container exposes its health check endpoint

TCP checks: Kubernetes tries to establish a TCP connection to your application on the specified port. If it can establish a connection, the probe is successful. Otherwise, it's failed.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image
    ports:
    - containerPort: 8080
    livenessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 10
      failureThreshold: 3
```

We use the tcpSocket handler to check the container's health by trying to open a TCP connection to port 8080. If the connection is successful, the Liveness Probe is considered successful

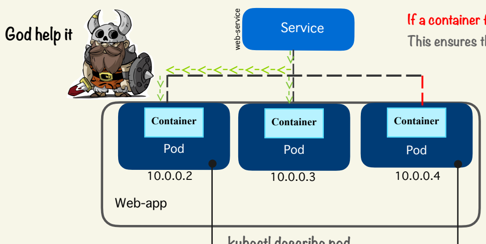
Exec checks: Kubernetes executes the specified command within your container. If the command returns an exit status of 0, the probe is successful. Otherwise, it's considered a failure.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: my-image
    livenessProbe:
      exec:
        command:
        - /bin/sh
        - -c
        - /usr/bin/custom-script.sh
      initialDelaySeconds: 30
      periodSeconds: 10
```

This probe runs a script inside the container. If the script terminates with 0 as its exit code, it means the container is running as expected

Example: a Web Application with Readiness Probe

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web-app
  template:
    metadata:
      labels:
        app: web-app
    spec:
      containers:
        - name: web-container
          image: my-web-image
          ports:
            - containerPort: 8080
          readinessProbe:
            httpGet:
              path: /healthz
              port: 8080
            initialDelaySeconds: 10
            periodSeconds: 5
            failureThreshold: 3
```



If a container fails the Readiness Probe check, it will be removed from the list of endpoints used by the service as a backend.

This ensures that the service does not send requests to the container until it becomes ready to receive them again.

```
Conditions:
  Type              Status
  ----              ------
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
```

```
Conditions:
  Type              Status
  ----              ------
  Initialized       True
  Ready             False
  ContainersReady   True
  PodScheduled      True
```

```
kubectl describe svc web-service

Name:              web-service
Namespace:         default
Labels:            <none>
Annotations:       Selector:  app=web-app
Type:              ClusterIP
IP:                10.0.0.1
Port:              http  80/TCP
TargetPort:        8080/TCP
Endpoints:         10.0.0.2:8080, 10.0.0.3:8080, 10.0.0.4:8080
Session Affinity:  None
Events:            <none>
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web-service
spec:
  selector:
    app: web-app
  ports:
    - name: http
      port: 80
      targetPort: 8080
  type: ClusterIP
```

failureThreshold is a parameter that can specify how many consecutive failures are allowed before the container is considered to have failed the probe.

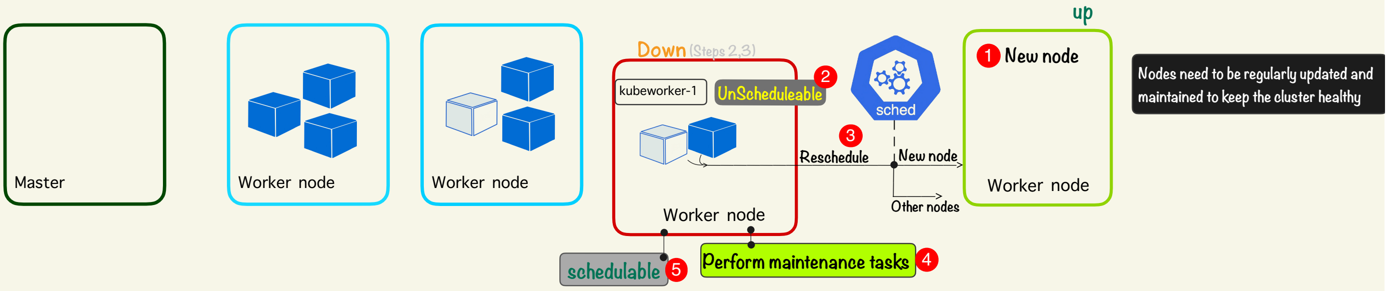
If the deployment fails the probe check three times in a row, the kubelet will restart the pod

```
K describe deploy web-app
Events
      readiness probe failed
```

Cluster maintenance

Node maintenance

Node maintenance in Kubernetes refers to the process of temporarily taking a node out of the cluster to perform maintenance tasks such as upgrading the operating system, applying security patches, replacing hardware or performing other tasks that require the node to be offline. During this time, any workloads running on the node will be evicted and rescheduled onto other nodes in the cluster to ensure high availability and minimal disruption to users.



Nodes need to be regularly updated and maintained to keep the cluster healthy

Steps to perform maintenance on a node in Kubernetes

To avoid any service disruptions during node maintenance, it's important to ensure that your Kubernetes cluster has sufficient resources and capacity to handle the workload of the evicted pods. If a node is added to the cluster, it increases the overall resources available for scheduling pods, reducing the chances of service disruptions. (optional)

Once the drain command completes and all the pods have been successfully rescheduled onto other nodes, you can perform the required maintenance tasks on the drained node. This may include updating the operating system, performing security patches, or any other necessary maintenance activities

Not adding a replacement node may cause the cluster to become unready, especially when there are a large number of pods running on the node being taken down and insufficient resources available on the remaining nodes to allocate to those pods

Node maintenance steps: Add a Node -> Cordon -> Drain -> Perform maintenance tasks -> UnCordon -> Remove added Node [after updating all nodes]

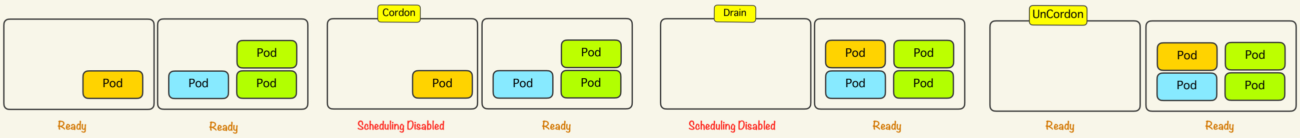
1. kubectl cordon kubeworker-1
2. kubectl drain kubeworker-1
3. Start node updating .....
4. kubectl uncordon kubeworker-1

When maintenance needs to be performed on a node, it should be cordoned as the first step

The cordon command marks a node as unschedulable. It prevents new pods from being scheduled on the node while allowing existing pods to continue running. By running kubectl cordon <node-name>, you indicate that the node is entering maintenance and should not receive any additional workload.

The drain command is used to gracefully evict pods from the node that is undergoing maintenance. It triggers the rescheduling of active pods onto other available nodes in the cluster. Running kubectl drain <node-name> initiates the process of moving pods off the node, ensuring that they are not abruptly terminated.

After the maintenance tasks are completed, the node can be brought back online and added back to the cluster. Kubernetes will automatically detect the new node and begin scheduling pods on it again. It's important to note that when a node is added back to the cluster, Kubernetes will not automatically move all the evicted pods back to the node. Instead, the scheduler will treat the node like a new node and schedule new pods onto it based on the available resources and workload requirements



Reserving resources for the operating system and the kubelet in Kubernetes is crucial for maintaining stability

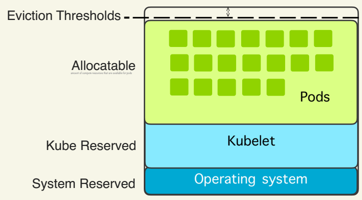
Kubernetes nodes can encounter resource starvation issues when pods consume all available capacity on a node, resulting in an insufficient allocation of resources for critical system daemons and processes that drive the functioning of the operating system and Kubernetes infrastructure. This imbalance can subsequently lead to cluster instability and performance degradation. configuring kubelet resource reserves is a good way to prevent resource starvation issues on Kubernetes nodes.

Here are some ways kube and system resource reserves can help:

kube-reserved This reserves resources for Kubernetes system daemons like kubelet, container runtime, node problem detector, etc. Prevents starvation of critical components.

system-reserved Reserves resources for the underlying node's kernel and system services. Leaves room for OS processes.

eviction-hard The kubelet will evict pods when available resources drop below this threshold to maintain reserves



To configure these reserves, you can set flags on the kubelet service like:

```
--kube-reserved=cpu=500m,memory=1Gi
--system-reserved=cpu=1,memory=2Gi
--eviction-hard=memory.available<500Mi
```

SPOF

Single Point of Failure (SPOF) refers to a component or resource that, if it fails, can cause a complete or partial outage of the entire system. This means that the failure of a single component can result in the unavailability or degraded performance of the overall Kubernetes cluster. Identifying and mitigating SPOFs is crucial for ensuring high availability and reliability in a Kubernetes environment. Here are some recommendations for ensuring the minimum amount of SPOFs for critical Kubernetes components:

- **Kubernetes Control Plane** - Need at least 3 master nodes spread across availability zones. This ensures high availability of API server and controller manager.
- **etcd** - For production, need at least 3 etcd instances, 5 for better redundancy. Should be co-located with control plane nodes.
- **Worker Nodes** - No specific minimum, but have at least 3 nodes in a cluster and spread them across zones.
- **Load Balancers** - Front load balancers with at least 2 instances or use external LB services.
- **Ingress Controllers** - Need 2+ ingress controllers like Nginx for redundancy. Configure with a load balancer.
- **Data Storage** - Use cluster-wide storage like GlusterFS, Rook, OpenEBS with replication.
- **Cluster Networking** - Should have high availability at the network level - multiple switches, routers etc. Avoid SPOF in networking.