“Kubernetes for DevOps Engineers: Real-World Scenarios Explained”

1. How do you implement canary deployments in Kubernetes?

> Two deployments (stable + canary) with same service selector. Control traffic with labels or weights.

Imagine you have a website running in Kubernetes (v1 version). You want to release a new version (v2), but you’re scared that something might break.

So instead of updating all users to v2 at once, you show the new version to only 10% of users. If everything looks good, you increase it to 50%, and finally 100%. That’s called a Canary Deployment.

Like a "canary in a coal mine", if something is wrong, you find out before everyone is affected.

Example Scenario:

You have:

v1 running as the current version.

You want to release v2 to only 20% of users at first.

- Method 1: Using 2 Deployments + 1 Service Create two deployments:

myapp-v1: 4 replicas (pods) with image myapp:1.0

myapp-v2: 1 replica (pod) with image myapp:2.0

That means: 80% of users get v1, and 20% get v2.

Create a single Service:

It selects all pods with app: myapp

Since v1 has more pods, most traffic goes to it.

v1 Deployment

```
apiVersion: apps/v1
kind: Deployment
```

```yaml
metadata:
  name: myapp-v1
spec:
  replicas: 4
  selector:
    matchLabels:
      app: myapp
      version: v1
  template:
    metadata:
      labels:
        app: myapp
        version: v1
    spec:
      containers:
      - name: myapp
        image: myapp:1.0

# v2 Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myapp
      version: v2
  template:
    metadata:
      labels:
        app: myapp
        version: v2
    spec:
      containers:
      - name: myapp
        image: myapp:2.0

# Shared Service
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
spec:
  selector:
    app: myapp
  ports:
    - protocol: TCP
    port: 80
    targetPort: 8080
```

2.Your pod is stuck in CrashLoopBackOff. How do you debug it?

The container inside the pod starts, crashes, and Kubernetes tries to restart it, but it keeps failing repeatedly. So, Kubernetes waits (backs off) longer between each restart.

The CrashLoopBackOff is Kubernetes' way of protecting your system from unstable pods. It's caused by repeated crashes inside a pod and indicates an underlying issue—either in code, configuration, resources, or dependencies.

To fix it, you need to investigate the pod’s behavior, configuration, and logs, then apply the appropriate correction.

Check the Pod Status

```
kubectl get pods
```

You’ll see something like:

myapp 1/1 CrashLoopBackOff 5 (10s ago)

2. Describe the Pod (for events and reasons)

```
kubectl describe pod myapp
```

3. Check Pod Logs

```
~ kubectl logs mypod
```

4. Check Liveness/Readiness Probes

If your pod has a health check (probe) and it fails, Kubernetes kills and restarts the pod.

Check this in the output of `kubectl describe pod`

3.You updated the image in a deployment but pods still use the old version. Why?

Same Image Tag Without Change:

If I used a static tag like latest or v1, and pushed a new image with the same tag, Kubernetes doesn't detect any change because it doesn't track the internal contents of the image. It assumes the image is the same.

No Change in Pod Template:

Kubernetes only triggers a rollout if it sees a change in the Deployment’s pod template. If the image tag didn't change, Kubernetes won’t redeploy the pods.

Example:

Check if the deployment really updated:

```
kubectl get deployment <deployment-name> -o yaml | grep image
```

Re-apply your changes: `kubectl apply -f deployment.yaml`

Kubernetes Uses Cached Image

```yaml
containers:
  - name: myapp
    image: myapp:latest
    imagePullPolicy: Always
```

OR delete the pod to force a fresh pull:

```
kubectl delete pod mypod
```

Rolling Update Didn't Trigger

```
kubectl rollout restart deployment mydeployment
```

4.You want zero downtime during app updates. What to do?

Use the default RollingUpdate strategy in deployments, which ensures that new pods are gradually brought up while old ones are terminated only after the new ones become ready.

I configure a proper readinessProbe, so Kubernetes routes traffic only to healthy and fully initialized pods.

I set maxUnavailable: 0 and maxSurge: 1 in the rolling update strategy to ensure that no pod downtime occurs during the rollout.

I ensure that my application changes are backward-compatible — especially when dealing with shared services

Example:

```
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 1
    maxSurge: 1
```

4. How do you expose an internal service to external traffic securely?

To expose an internal service securely to external traffic, I use an Ingress resource with TLS/HTTPS.

I set up an Ingress Controller (like NGINX or Traefik), and use cert-manager to manage TLS certificates from Let's Encrypt.

This allows secure external access via HTTPS, while keeping internal traffic protected.

Example:

- Create a Service (ClusterIP)

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app
  ports:
    - port: 80
      targetPort: 8080
  type: ClusterIP  # Internal only
```

- Create an Ingress Resource

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-app-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    cert-manager.io/cluster-issuer: letsencrypt-prod
spec:
  tls:
    - hosts:
        - myapp
      secretName: myapp-tls
  rules:
    - host: myapp
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: my-app-service
                port:
                  number: 80
```

- TLS via cert-manager (automates HTTPS)

```
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    email: your-email@example.com
    server: https://acme-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      name: letsencrypt-prod
    solvers:
      - http01:
          ingress:
            class: nginx
```

5. How do you store database credentials in K8s securely?

In Kubernetes, I store database credentials securely using Secrets.

Secrets keep sensitive data like usernames and passwords base64-encoded and separate from application code.

I then mount the Secret as environment variables or as files into my pods. I also make sure to apply RBAC policies so that only authorized pods or users can access them.

Example:

- Store DB Credentials Using Secrets

```
kubectl create secret generic db-secret \
  --from-literal=username=admin \
  --from-literal=password=MyS3cretP@ss
```

- Create a Secret(YAML)

```
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
data:
  username: myapp=         # base64 for 'admin'
  password: myapp@123 # base64 for 'MyS3cretP@ss'
```

- Mount Secret as Environment Variables in Pod

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

```yaml
    name: db-client
spec:
  containers:
    - name: app
      image: myapp
      env:
        - name: DB_USER
          valueFrom:
            secretKeyRef:
              name: db-secret
              key: username
        - name: DB_PASS
          valueFrom:
            secretKeyRef:
              name: db-secret
              key: password
```

6. Node crashes. What happens to pods?

When a node crashes in Kubernetes, the Kubelet stops reporting to the control plane, and the node is marked as NotReady.

Kubernetes automatically waits for a short timeout (default is ~5 minutes), then evicts the pods running on that node.

If those pods are part of a Deployment, ReplicaSet, or StatefulSet, the controller automatically reschedules them onto healthy nodes.

But if the pod was created directly (not managed), it will not be recreated unless manually handled.

Example:

- You have a Deployment with 3 replicas:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
        - name: webapp
          image: nginx
```

7.Your deployment is deleted accidentally. How to recover?

If a deployment is accidentally deleted, the first step is to check if we have the deployment YAML stored in Git or as a backup. If yes, I can simply re-apply it using kubectl apply -f.

If no YAML backup is available, and the pods are still running, I can extract the current running pod's configuration and use that to recreate the deployment.

For production-grade clusters, I usually implement GitOps with tools like ArgoCD or Flux, which can automatically reapply the deleted resources from Git.

- Example:

1: Re-apply from Git or Backup YAML

```
kubectl apply -f deployment-backup.yaml
```

2: Recover from Running Pod (No YAML Available)

- Get pod name `kubectl get pods`
- ii. Extract pod config `kubectl get pod mypod -o yaml > pod.yaml`

3. Edit it to convert into a deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: recovered-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
```

```
template:
  <your pod spec here>
```