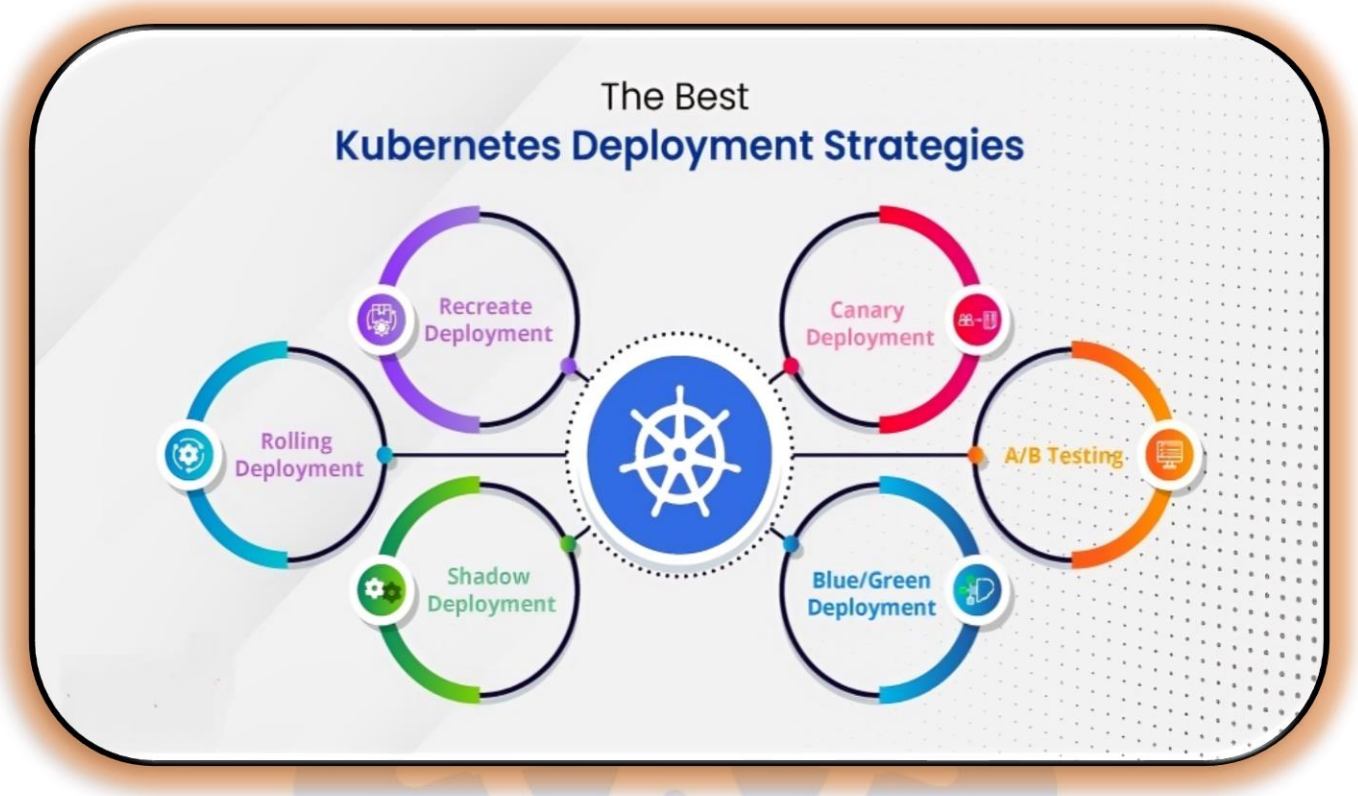
DEPLOYMENT STRATEGIES

K8S



✍️ Kubernetes Deployment strategies

Kubernetes has emerged as the go-to solution for managing containerized applications in modern software development. It offers powerful orchestration capabilities, but deploying applications seamlessly and minimizing downtime remains a challenge.

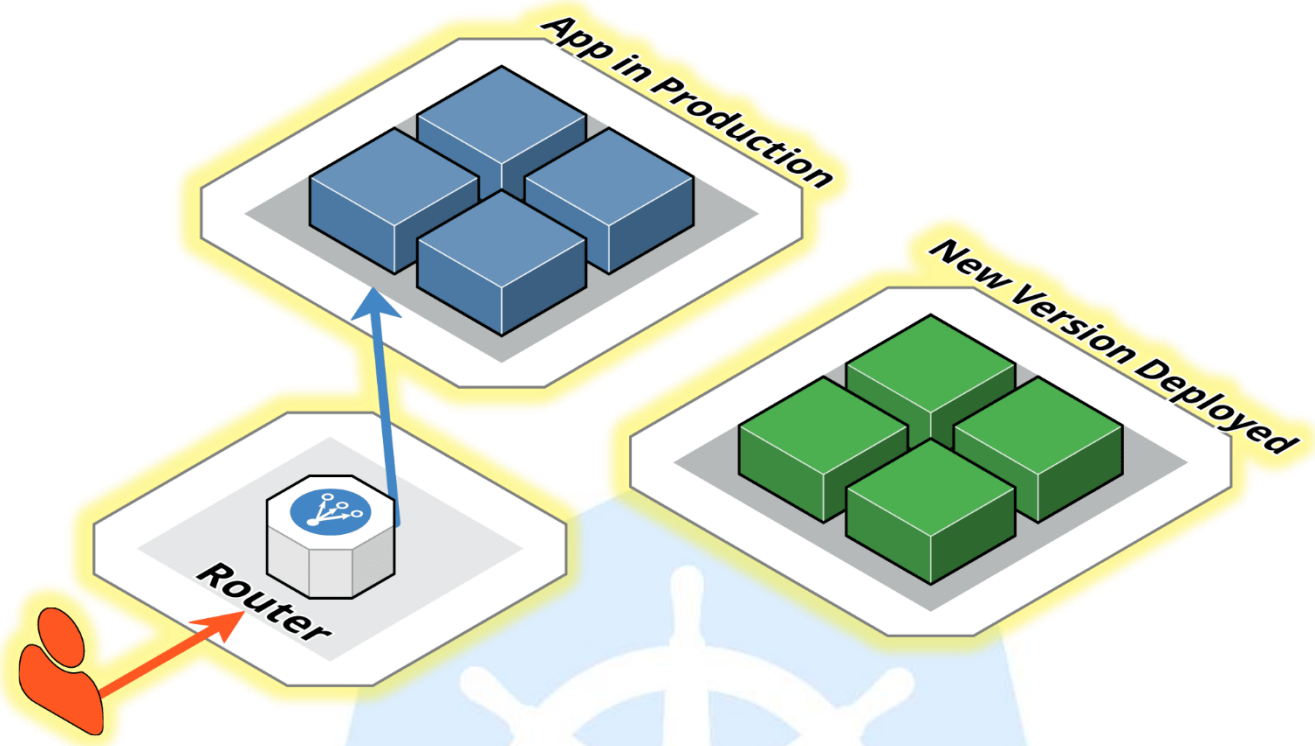
We'll explore three popular Kubernetes deployment strategies:

👉Blue-Green..

👉Rolling Deployment.

👉Canary deployments.

Blue-Green Deployments



✍️ What is Blue-Green Deployment?

Blue-Green deployment is a strategy that involves running two identical environments, one of which is active (Blue) and the other inactive (Green). The active environment serves production traffic while the inactive one remains idle.

When it's time to release a new version, the deployment team switches traffic from the Blue environment to the Green environment, making the new version live. This approach ensures zero-downtime updates.

Advantages of Blue-Green Deployments:

👉Zero Downtime

👉Easy Rollback

👉Thorough Testing

👉 Create deployment **1**.

 # kubectl create deployment blue --image=docker.io/httpd --dry-run=client -o yaml > blue.yml

```
controlplane $ kubectl create deployment blue --image=docker.io/httpd --dry-run=client -o yaml > blue.yml
controlplane $ ls
blue.yml  filesystem  snap
controlplane $
```

👉 Check the **Yaml** file.

```
  name: blue
spec:
  replicas: 1
  selector:
    matchLabels:
      app: blue
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: blue
```

👉 Apply the **Yaml** file.

 # kubectl apply -f blue.yml

```
controlplane $ kubectl apply -f blue.yml
deployment.apps/blue created
```

👉 Create deployment **2**.

 # kubectl create deployment green --image=docker.io/httpd --dry-run=client -o yaml > green.yml

```
controlplane $ kubectl create deployment green --image=docker.io/httpd --dry-run=client -o yaml > green.yml
controlplane $ ls
blue.yml  filesystem  green.yml  snap
controlplane $
```

👉 Check whether deployment is created or not

 # kubectl get deployment -o wide

```
controlplane $ kubectl get deployment -o wide
NAME    READY   UP-TO-DATE   AVAILABLE   AGE     CONTAINERS   IMAGES              SELECTOR
blue    1/1     1            1           2m26s   httpd        docker.io/httpd     app=blue
green   1/1     1            1           20s     httpd        docker.io/httpd     app=green
```

👉Create a service for deployment **1**.

 # kubetcl expose deployment blue --port=80

```
controlplane $ kubectl expose deployment blue --port=80
service/blue exposed
```

👉Check whether service is created or not.

 # kubectl get svc

```
controlplane $ kubectl get svc
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
blue         ClusterIP   10.106.208.226   <none>        80/TCP    14s
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP   3d1h
```

👉Ping **IP**

 # curl 10.106.208.226

```
controlplane $ curl  10.106.208.226
BLUE DEPLOYMENT
```

👉Attach Deployment 1 **(blue)** service to Deployment 2 **(green).**

 # Kubectl edit svc blue

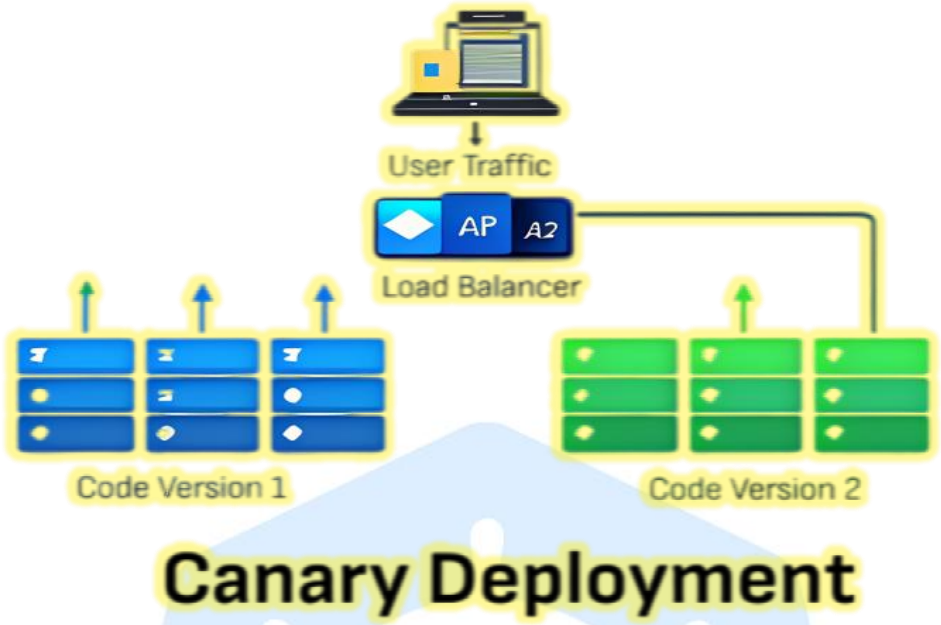
```
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: green
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

👉Ping **IP**

 # curl 10.106.208.226

```
controlplane $ curl  10.106.208.226
GREEN DEPLOYMENT
```

Canary Deployments



✍️ What is Canary Deployment?

Canary deployment is another Kubernetes deployment strategy. Unlike Blue-Green, Canary deployments don't involve running two complete environments.

Instead, they gradually roll out a new version to a subset of users while keeping the rest on the previous version. This allows for A/B testing and gradual monitoring of the new release.

Advantages of Canary Deployments:

👉**Gradual Rollout.**

👉**A/B Testing.**

👉 **Incremental Deployment.**

👉 Create deployment **1**.

 # kubectl create deployment mydep1 --image=docker.io/nginx --dry-run=client -o yaml > mydep1.yml

```
controlplane $ kubectl create deployment mydep1 --image=docker.io/nginx --dry-run=client -o yaml > mydep1.yml
controlplane $
```

👉 Give same label for deployment **1** & deployment **2.**

 # class : canary

```
    matchLabels:
      app: mydep1
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: mydep1
        class: canary
    spec:
      containers:
      - image: docker.io/nginx
        name: nginx
```

👉 Create deployment **2**.

 # kubectl create deployment mydep2 --image=docker.io/nginx --dry-run=client -o yaml > mydep2.yml

```
controlplane $ kubectl create deployment mydep2 --image=docker.io/nginx --dry-run=client -o yaml > mydep2.yml
```

👉 Apply both the **Yaml** file.

 # kubectl apply -f .

```
controlplane $ kubectl apply -f .
deployment.apps/mydep1 created
deployment.apps/mydep2 created
controlplane $
```

👉Check label of pos is same or not.

 # kubectl get pod --show-labels

```
controlplane $ kubectl get pod --show-labels
NAME                      READY   STATUS    RESTARTS   AGE   LABELS
mydep1-67c4dbb55-cns78    1/1     Running   0          22m   app=mydep1,class=canary,pod-template-hash=67c4dbb55
mydep1-67c4dbb55-l9vc6    1/1     Running   0          22m   app=mydep1,class=canary,pod-template-hash=67c4dbb55
mydep2-7dc5f44548-5jbvx   1/1     Running   0          22m   app=mydep2,class=canary,pod-template-hash=7dc5f44548
mydep2-7dc5f44548-nfrnt   1/1     Running   0          22m   app=mydep2,class=canary,pod-template-hash=7dc5f44548
```

👉Create a service for deployment **1**.

 # kubetcl expose deployment mydep1 --port=80

```
controlplane $ kubectl expose deployment mydep1 --port=80
service/mydep1 exposed
controlplane $
```

👉Give same label you have given to pod.

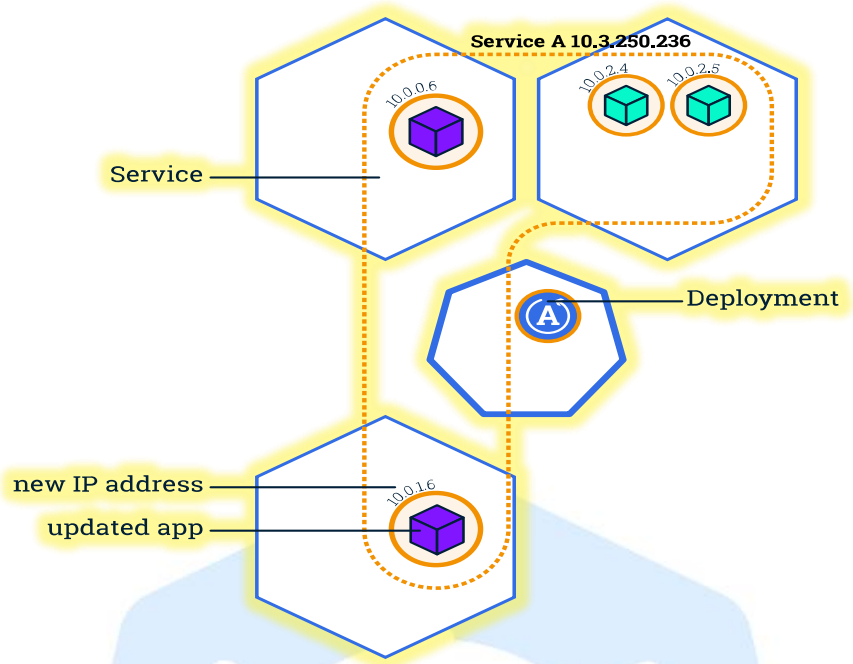
```
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    class: canary
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

👉Check whether pod is connected to svc.

 # kubectl describe svc <Service_Name>

```
controlplane $ kubectl describe svc mydep1
Name:              mydep1
Namespace:         default
Labels:            app=mydep1
Annotations:       <none>
Selector:          class=canary
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.100.163.29
IPs:               10.100.163.29
Port:              <unset>  80/TCP
TargetPort:        80/TCP
Endpoints:         192.168.1.4:80,192.168.1.5:80,192.168.1.6:80 + 1 more...
```

Rolling Deployments



✍️ What is Rolling Deployment?

A rolling deployment involves gradually replacing an older version of the application with a new one. This software deployment strategy helps gradually replace the infrastructure running the application until the rolling deployment becomes the only version.

It let you update a set of pods with no downtime, by incrementally replacing pod instances with new instances that run a new version of the application.

Advantages of Rolling Deployments:

- 👉 **Zero Downtime.**
- 👉 **Easy Rollback.**
- 👉 **Flexibility.**

👉Create deployment.

 # kubectl create deployment rollout --image=docker.io/httpd:latest

```
controlplane $ kubectl create deployment rollout --image=docker.io/httpd:latest
deployment.apps/rollout created
controlplane $
```

👉Check whether deployment is created or not

 # kubectl get deployment -o wide

```
controlplane $ kubectl get deployment -o wide
NAME      READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS   IMAGES                  SELECTOR
rollout   1/1     1            1           57s   httpd        docker.io/httpd:latest   app=rollout
controlplane $
```

👉Check the rollout history.

 # kubectl rollout history deployment <Deployment_Name>

```
controlplane $ kubectl rollout history deployment rollout
deployment.apps/rollout
REVISION  CHANGE-CAUSE
1         <none>

controlplane $
```

👉Change image version.

 # kubectl set image deployment rollout httpd=httpd:2.1 –record=true

```
controlplane $ kubectl set image deployment rollout httpd=httpd:2.1 --record=true
Flag --record has been deprecated, --record will be removed in the future
deployment.apps/rollout image updated
controlplane $
```

👉Check whether image version is change or not.

 # kubectl rollout history deployment <Deployment_Name>

```
controlplane $ kubectl rollout history deployment rollout
deployment.apps/rollout
REVISION  CHANGE-CAUSE
1         <none>
2         kubectl set image deployment rollout httpd=httpd:2.1 --record=true
3         kubectl set image deployment rollout httpd=httpd:2.3 --record=true

controlplane $
```

👉You can go one step back or directly recently used version.

#kubectl rollout undo deployment <Deployment_Name> --to-revision=1

```
controlplane $ kubectl rollout undo deployment rollout --to-revision=1
deployment.apps/rollout rolled back
```

👉Check whether image version is change or not.

 # kubectl rollout history deployment <Deployment_Name>

```
controlplane $ kubectl rollout history deployment rollout
deployment.apps/rollout
REVISION  CHANGE-CAUSE
2         kubectl set image deployment rollout httpd=httpd:2.1 --record=true
3         kubectl set image deployment rollout httpd=httpd:2.3 --record=true
4         <none>
```

Conclusion

We've discuss three Kubernetes deployment strategies:

👉 Blue-Green.

👉 Canary.

👉 Rolling deployments.

Here's a simplified conclusion for each strategy:

😃 **Blue-Green Deployment:**

This approach involves running two identical environments (Blue and Green). The Blue environment is active and serves production traffic, while the Green environment remains idle. When deploying a new version, you switch the traffic from Blue to Green, ensuring zero downtime and making it easy to roll back if needed.

😃 **Canary Deployment:**

This strategy gradually rolls out a new version to a small subset of users while the majority continue using the old version. This allows for A/B testing and reduces risks since you can monitor the new version's performance on a limited scale before a full rollout.

😃 **Rolling Deployment:**

In this method, new versions replace the old versions incrementally. It updates pods in a rolling manner, meaning that the transition happens one instance at a time, ensuring zero downtime and the ability to roll back easily if issues arise.