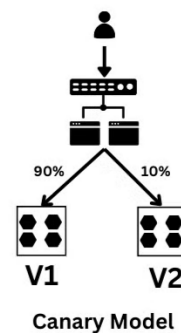
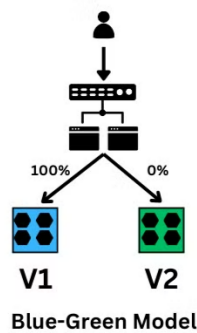
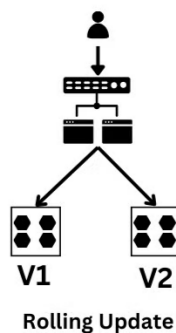



Deployment Strategies in Kubernetes: A Hands-on Guide

Introduction: Deployment strategy in kubernetes is used to update application running in kubernetes environment. These strategies play a crucial role in application lifecycle management, directly influencing application availability, service continuity and the capacity to adapt to new requirements.

3 Widely Used Kubernetes Deployment Strategies



 Balavignesh-manoharan

In this article, we will look into three main kubernetes deployment strategies and how to deploy each of these strategies.

The 3 Main Kubernetes Deployment Strategies:

1. **Rolling Update**
2. **Canary Model Deployment**
3. **Blue/Green Deployment**

Rolling Update

Rolling update are the default strategy for deployment in kubernetes. This process allows you to update your application in kubernetes with minimal downtime, as it ensures that some instances of your application are always running during the update process.

The rolling update uses a readiness probe to check if the pod is ready, before starting to scale down the pods of the older version. If there is a problem, you can stop an update and roll it back, without stopping the entire cluster. To perform a rolling update, simply update the image of your pods using `kubectl set image`. This will automatically trigger a rolling update.

There are two optional parameters:

- **maxSurge**: Specifies the number of pods of the deployment that is allowed to create at one time.
- **maxUnavailable**: Maximum number of pods that are allowed to be unavailable during the rollout.

Rolling update Hands-on:

Pre-requisites:

- minikube

Once your minikube is up and running, we will create a deployment with an nginx image and try to check the rolling update.

1. Start the minikube

```
minikube start
```

```

bala_pc@Bala:~/deployment$ minikube start
minikube v1.33.1 on Ubuntu 22.04 (amd64)
Using the docker driver based on existing profile
Starting "minikube" primary control-plane node in "minikube" cluster
Pulling base image v0.0.44 ...
Restarting existing docker container for "minikube" ...
Preparing Kubernetes v1.30.0 on Docker 26.1.1 ...
Verifying Kubernetes components...
  ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
  ▪ Using image registry.k8s.io/ingress-nginx/controller:v1.10.1
  ▪ Using image registry.k8s.io/ingress-nginx/kube-webhook-certgen:v1.4.1
  ▪ Using image registry.k8s.io/ingress-nginx/kube-webhook-certgen:v1.4.1
  ▪ Using image registry.k8s.io/metrics-server/metrics-server:v0.7.1
  ▪ Using image docker.io/kubernetes/metrics-scraper:v1.0.8
  ▪ Using image docker.io/kubernetes/dashboard:v2.7.0
Verifying ingress addon...
Some dashboard features require the metrics-server addon. To enable all features please run:

    minikube addons enable metrics-server

Enabled addons: metrics-server, storage-provisioner, dashboard, ingress, default-storageclass
Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default

```

2. Once the minikube is started, create a deployment using the below command.

```
kubectl create deploy nginx --image=nginx
```

This command creates a deployment with 1 pod running.

3. Check if the pods are running.

```
kubectl get pods
```

4. Let us increase the replica to 4 as we need 4 replicas to try the rolling update in the example, so run the below command to edit.

```
kubectl edit deploy nginx
```

Change the replica count to **4**, save it and exit the editor.

5. Again check with pods to see the deployment has scaled its replica.

6. Open a new tab and run the below command.

```
kubectl get pods -w
```

This command will watch for the pods and we can see the rolling update in this tab.

7. Come back to the old tab and change the image in the deployment file using the below command.

```
kubectl set image deployment/nginx nginx=nginx:v2
```

nginx:v2 image does not exist, so the pod will go to crashbackoff as soon as we apply the changes.

8. You can see that it will try to change only **one pod** of the **old version** and try to create a new pod with new version. But since we have given a non-existent image it will go into the imagePullBackOff state.

```
bala_pc@Bala:~/deployment$ kubectl get pods -w
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-bf5d5cf98-6xzkv	1/1	Running	0	110s
nginx-bf5d5cf98-7nz4a	1/1	Running	0	3m40s
nginx-bf5d5cf98-c54dn	1/1	Running	0	110s
nginx-bf5d5cf98-km5tc	1/1	Running	0	110s
nginx-7d774f94d-zhaz7	0/1	Pending	0	0s
nginx-bf5d5cf98-c54dn	1/1	Terminating	0	7m33s
nginx-7d774f94d-zhgz7	0/1	Pending	0	0s
nginx-7d774f94d-d5rn5	0/1	Pending	0	0s
nginx-7d774f94d-zhgz7	0/1	ContainerCreating	0	0s
nginx-7d774f94d-d5rn5	0/1	Pending	0	0s
nginx-7d774f94d-d5rn5	0/1	ContainerCreating	0	0s
nginx-bf5d5cf98-c54dn	0/1	Terminating	0	7m35s
nginx-bf5d5cf98-c54dn	0/1	Terminating	0	7m36s
nginx-bf5d5cf98-c54dn	0/1	Terminating	0	7m36s
nginx-bf5d5cf98-c54dn	0/1	Terminating	0	7m36s
nginx-7d774f94d-zhgz7	0/1	ErrImagePull	0	7s
nginx-7d774f94d-d5rn5	0/1	ErrImagePull	0	10s
nginx-7d774f94d-zhgz7	0/1	ImagePullBackOff	0	60s
nginx-7d774f94d-d5rn5	0/1	ImagePullBackOff	0	62s

9. Now when you try to run the `kubectl get pods` command, you can see that instead of 4 replicas 3 of them are running.

```

bala_pc@Bala:~/deployment$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
nginx-7d774f94d-sg9kf               0/1    ErrImagePull  0          16s
nginx-7d774f94d-t6r88               0/1    ErrImagePull  0          16s
nginx-bf5d5cf98-6xzkv               1/1    Running      0          19m
nginx-bf5d5cf98-7nz4q               1/1    Running      0          21m
nginx-bf5d5cf98-km5lc               1/1    Running      0          19m

```

10. When you create a service then kubernetes will see that only **3** pods are running, so it will forward to the request to only these pods since the other pod is not healthy.
11. So in this way if you're using a deployment strategy using rolling update, even if something goes wrong we still have **old copies running**. Hence user will not see any **downtime**. If you have provided a existing image, then one by one the pods will get updated to the new version and the application is deployed successfully.

Canary Model Deployment

A canary strategy is something when the new version is rolled out to a **small subset of users** before being rolled out to the entire user base. Based on the feedback and response, it is either rolled out to the rest or rolled back. This approach limits the impact of any new bugs and allows for real-world testing on a small scale before broad deployment.

Once confidence increases in the new version, you can gradually roll it out to the entire infrastructure. This can be achieved by using certain parameters in your load balancer spec section. As in when a user is trying to hit the load balancer of your application, then **90%** user should reach the version 1 and **10%** user should reach the version 2. Let us see the hands-on demo on how this is achieved in real time.

Canary Model Hands-on:

1. First delete your previous deployment if it is running.

```
kubectl delete deploy nginx
```

2. For the load balancer to be created, we need ingress controller. Since we are using minikube let us enable ingress controller. (nginx ingress controller)

```
minikube addons enable ingress
```

The reason for using a load balancer here is to manage traffic distribution across multiple versions and also rollback smoothly in case if the new version isn't stable.

3. For example we will use the default nginx ingress controller canary which is available in github. Visit the below page for your example.

```
https://kubernetes.github.io/ingress-nginx/examples/canary/
```

Ingress Nginx Has the ability to handle canary routing by setting specific annotations, the following is an example of how to configure a canary deployment with weighted canary routing.

4. Copy the deployment and service example, and run it in your terminal.

```
echo "  
---  
# Deployment  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: production  
  labels:  
    app: production  
spec:  
  replicas: 1  
  selector:  
    matchLabels:  
      app: production  
  template:  
    metadata:  
      labels:  
        app: production
```

```

spec:
  containers:
  - name: production
    image: registry.k8s.io/ingress-nginx/e2e-test-echo@sha256:
    ports:
    - containerPort: 80
    env:
      - name: NODE_NAME
        valueFrom:
          fieldRef:
            fieldPath: spec.nodeName
      - name: POD_NAME
        valueFrom:
          fieldRef:
            fieldPath: metadata.name
      - name: POD_NAMESPACE
        valueFrom:
          fieldRef:
            fieldPath: metadata.namespace
      - name: POD_IP
        valueFrom:
          fieldRef:
            fieldPath: status.podIP
  ---
# Service
apiVersion: v1
kind: Service
metadata:
  name: production
  labels:
    app: production
spec:
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP

```

```
    name: http
  selector:
    app: production
" | kubectl apply -f -
```

5. Check if the deployment and service is created in the name of production.

```
kubectl get all
```

6. Now let us create the deployment and service for canary (**version 2**). Copy the below code.

```
echo "
---
# Deployment
apiVersion: apps/v1
kind: Deployment
metadata:
  name: canary
  labels:
    app: canary
spec:
  replicas: 1
  selector:
    matchLabels:
      app: canary
  template:
    metadata:
      labels:
        app: canary
    spec:
      containers:
        - name: canary
          image: registry.k8s.io/ingress-nginx/e2e-test-echo@sha256:
          ports:
```



```

    - containerPort: 80
  env:
    - name: NODE_NAME
      valueFrom:
        fieldRef:
          fieldPath: spec.nodeName
    - name: POD_NAME
      valueFrom:
        fieldRef:
          fieldPath: metadata.name
    - name: POD_NAMESPACE
      valueFrom:
        fieldRef:
          fieldPath: metadata.namespace
    - name: POD_IP
      valueFrom:
        fieldRef:
          fieldPath: status.podIP
  ---
# Service
apiVersion: v1
kind: Service
metadata:
  name: canary
  labels:
    app: canary
spec:
  ports:
    - port: 80
      targetPort: 80
      protocol: TCP
      name: http
  selector:
    app: canary
" | kubectl apply -f -

```

7. Create an ingress for the production deployment (version 1) using the below code.

```
echo "
---
# Ingress
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: production
  annotations:
spec:
  ingressClassName: nginx
  rules:
  - host: echo.prod.mydomain.com
    http:
      paths:
      - pathType: Prefix
        path: /
        backend:
          service:
            name: production
            port:
              number: 80
" | kubectl apply -f -
```

8. Now let us create an ingress for the canary deployment (**version 2**) in which you will notice that there is an additional field called **annotation**. This annotation tells the controller that the traffic percentage is **10%**.

```
echo "
---
# Ingress
apiVersion: networking.k8s.io/v1
kind: Ingress
```

```

metadata:
  name: canary
  annotations:
    nginx.ingress.kubernetes.io/canary: \"true\"
    nginx.ingress.kubernetes.io/canary-weight: \"10\"
spec:
  ingressClassName: nginx
  rules:
  - host: echo.prod.mydomain.com
    http:
      paths:
      - pathType: Prefix
        path: /
        backend:
          service:
            name: canary
            port:
              number: 80
" | kubectl apply -f -

```

9. So as per the above ingress, we can see that the traffic is distributed and only **10%** of the users will be directed to the new version and the rest **90%** will be directed to the old version.

10. We can check this using the curl command. First let us copy the minikube IP

```
minikube ip
```

11. Copy the ip and keep it aside. To test our setup, run **minikube ssh** and once inside the cluster run the below command

```
for i in $(seq 1 10); do curl -s --resolve echo.prod.mydomain.com
```

In the above command, we're displaying up to 10 requests. The "resolve" option is used for DNS mapping our domain to echo.prod.mydomain.com. Replace **"INGRESS_CONTROLLER_IP"** with your minikube IP.

12. You will get the following output.

```
docker@minikube:~$ for i in $(seq 1 10); do curl -s --resolve echo.prod.mydomain.com:80:192.168.49.2 echo.prod.mydomain.com | grep "Hostname"; done
Hostname: production-59555fb857-bd1lk
Hostname: canary-5fdf8c5579-l2q8d
Hostname: production-59555fb857-bd1lk
Hostname: production-59555fb857-bd1lk
Hostname: production-59555fb857-bd1lk
Hostname: production-59555fb857-bd1lk
Hostname: production-59555fb857-bd1lk
Hostname: production-59555fb857-bd1lk
Hostname: production-59555fb857-bd1lk
Hostname: production-59555fb857-bd1lk
```

13. You can see that the load balancer is forwarding **90%** of request to the old version (version 1) and **10%** of request to the new version (version 2).

14. Once the canary model is tested and all the features are working fine, then we can slowly implement more traffic using the annotations in the ingress file by increasing the canary weight.

15. Lastly we will use the canary weight as **100** and see how it results.

```
kubectl edit ingress canary
```

and change the canary-weight to 100 and save the file.

15. Run the curl command again to check the output.

```
for i in $(seq 1 10); do curl -s --resolve echo.prod.mydomain.com:80:192.168.49.2 echo.prod.mydomain.com | grep "Hostname"; done
```

```
docker@minikube:~$ for i in $(seq 1 10); do curl -s --resolve echo.prod.mydomain.com:80:192.168.49.2 echo.prod.mydomain.com | grep "Hostname"; done
Hostname: canary-5fdf8c5579-l2q8d
Hostname: canary-5fdf8c5579-l2q8d
Hostname: canary-5fdf8c5579-l2q8d
Hostname: canary-5fdf8c5579-l2q8d
Hostname: canary-5fdf8c5579-l2q8d
Hostname: canary-5fdf8c5579-l2q8d
Hostname: canary-5fdf8c5579-l2q8d
Hostname: canary-5fdf8c5579-l2q8d
Hostname: canary-5fdf8c5579-l2q8d
Hostname: canary-5fdf8c5579-l2q8d
```

Finally all the requests is sent to canary. This is how canary deployment strategy is achieved.

Blue/Green Deployment

This method involves running two identical environments, a new version(green) alongside a old version(blue). The blue/green strategy keeps only one version live at any given time. It involves routing traffic to a blue deployment while creating

and testing a green deployment. After the testing phase is concluded you start routing traffic to the new version. Then you can keep the blue deployment for a future rollback.

However this strategy requires double resources for both deployment and can incur high costs. Also it requires a way to switch over traffic rapidly from blue to green version and back.

Blue/Green Deployment Hands-on:

1. The first step is to create a blue deployment. Use the below YAML code to create the blue environment.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: blue
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
      env: blue
  template:
    metadata:
      labels:
        app: myapp
        env: blue
    spec:
      containers:
        - name: myapp
          image: balav8/blue-env
          ports:
            - containerPort: 3000
```

This code creates a deployment with 3 replicas and the env set is blue and the image used is blue-env.

2. Apply the file to create the deployment.

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

3. Next step is to create the service, use the below code for the service file.

```
apiVersion: v1
kind: Service
metadata:
  name: blue
spec:
  selector:
    app: myapp
    env: blue
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
  type: NodePort
```

This code creates a service of env blue and we have set the target Port as **3000** in which our application was running. Also the type of service we are using here is **NodePort**.

4. Apply the service file to expose the application, so that we can access our app with Nodeport IP.

```
kubectl apply -f svc.yaml
```

5. To check if our application is running, we will access our application in the browser.

6. But before that, since we are using Minikube in WSL we need to use minikube tunnel to expose the service and then we will be able to access from host machine.

7. Just open a new terminal and run the below command.

```
minikube service blue
```

Here blue represents the service name. Now keep this terminal running and copy the URL which is provided by the tunnel for service blue.

```
bala_pc@Bala:~/deployment$ minikube service blue
```

NAMESPACE	NAME	TARGET PORT	URL
default	blue	80	http://192.168.49.2:30876

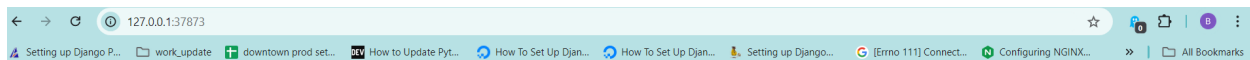
```
🏃 Starting tunnel for service blue.
```

NAMESPACE	NAME	TARGET PORT	URL
default	blue		http://127.0.0.1:37873

```
🌈 Opening service default/blue in default browser...  
👉 http://127.0.0.1:37873  
! Because you are using a Docker driver on linux, the terminal needs to be open to run it.
```

Copy the above URL and open a browser and run it.

8. You can see the blue environment running.



BLUE ENVIRONMENT

9. The next step is to create the green-deployment. Use the below code for the green-deployment.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: green
spec:
  replicas: 0
  selector:
    matchLabels:
      app: myapp
      env: green
  template:
    metadata:
      labels:
        app: myapp
        env: green
    spec:
      containers:
      - name: myapp
        image: balav8/green-env
        ports:
        - containerPort: 3000
```

This code creates a deployment with 0 replicas and the env set is green and the image used is green-env.

You can see that the green deployment replica count is set to 0. This means that the green deployment is defined and ready to be activated, but it is not consuming resources until needed. We will scale the green environment only when we are ready to route the traffic to it. This avoids the overhead of running additional pods and instances unnecessarily optimizing resources utilization.

10. Now suppose if our new version of application is tested and ready to be deployed, then we can scale the green deployment to three replicas.

```
kubectl scale deployment green --replicas=3
```

or just edit the deployment file and change the replica count to **3**.

11. Apply the above deployment again with the changes.

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

12. Check the pods if they are running.

```
bala_pc@Bala:~/deployment$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
blue-54cb6578d8-7hqhg	1/1	Running	0	43m
blue-54cb6578d8-wp4mp	1/1	Running	0	43m
blue-54cb6578d8-zq6gb	1/1	Running	0	43m
green-56484d7f6b-8qphr	1/1	Running	0	99s
green-56484d7f6b-8whkv	1/1	Running	0	99s
green-56484d7f6b-xjxrx	1/1	Running	0	99s

As you can see both the blue and the green deployment pods are running.

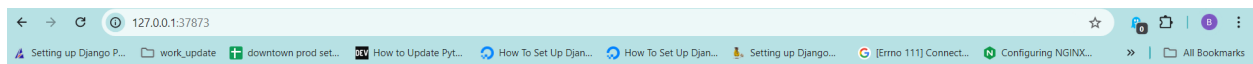
13. Now switch the traffic to green deployment. To do this, just edit your service file which you had created earlier.

```
apiVersion: v1
kind: Service
metadata:
  name: blue
spec:
  selector:
    app: myapp
    env: green
  ports:
  - protocol: TCP
    port: 80
    targetPort: 3000
  type: NodePort
```

Just change the env from blue to green, and apply the changes.

```
bala_pc@Bala:~/deployment$ kubectl apply -f blue-svc.yaml
service/blue configured
```

14. Once you have run the above command, go to your browser and refresh the page to see the new version deployed.



GREEN ENVIRONMENT

15. We have successfully deployed the green environment. This is how blue/green deployment strategy works.

No single strategy fits all scenarios. The decision should come from thoroughly analyzing the application requirements, organizational context and capabilities.

Conclusion: To sum up, there are different ways to deploy an application. Each strategy - Rolling Update, Canary Deployment, and Blue/Green Deployment - offers unique benefits and challenges. Rolling Update provides a smooth transition with minimal downtime, Canary Deployment allows for controlled testing with real users, and Blue/Green Deployment offers quick rollback capabilities. The key is to choose the strategy that best aligns with your application's needs, your team's expertise, and your organization's goals.

If you found this post useful, give it a like 👍

Repost 🔄

Follow @Bala Vignesh for more such posts 🚀