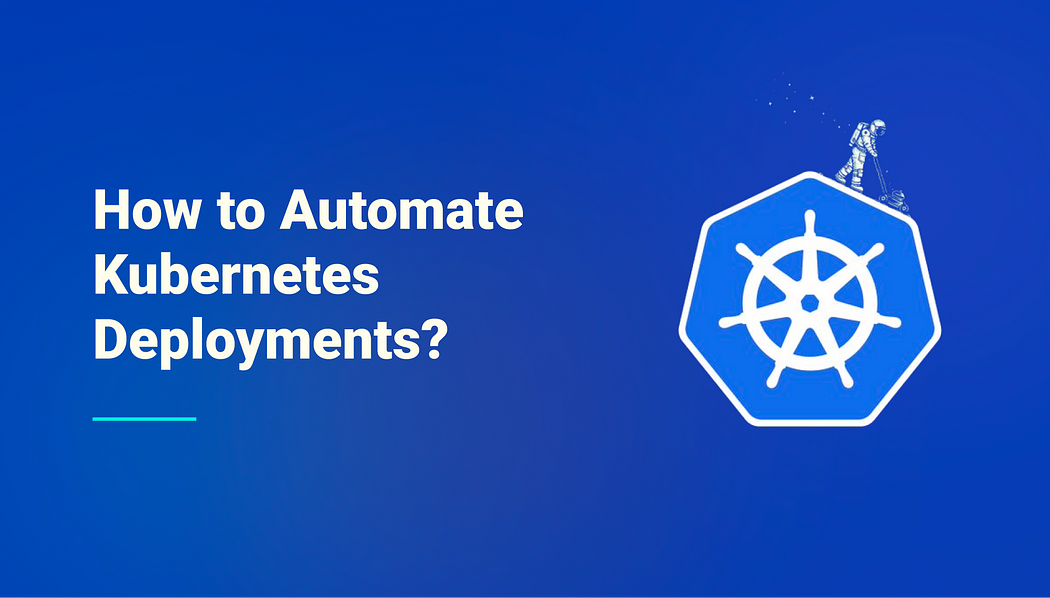
**Understanding Deployment Strategies in Kubernetes**



In a Kubernetes environment, a typical deployment in the cluster tells Kubernetes to create or modify the pods that hold a containerized application. A deployment helps to scale the number of pods using replicas and roll out with new, or updated code controllably.

Using Deployments in your Kubernetes helps to automate your work, and reduce several manual functions involved in building, scaling, updating, and rolling back applications. Since the deployment controller was developed to monitor the state of the pods in the cluster, it leads to faster deployment, with few errors.

When working with Kubernetes Deployments, there are several strategies that you can choose depending on the availability demands of your products. There are the “Recreate”, rolling update, Blue-green, and Canary deployment strategies.

Before we proceed into the article, you must know that only the Rolling deployment, and Recreate Deployment strategies are supported by Kubernetes out-of-the-box. TO perform both Blue-green strategies requires customization.

**Kubernetes Deployment Strategies.**

1. **Recreate Deployment Strategy.**

Whenever you apply the recreate strategy, all the existing pods from the previous deployments, are terminated, and replaced with a new version. Here the engineers are required to shut down the application entirely, deploy the new version, and start the entire system However, there will be some downtime within the application because new pods will only spin up, and serve user requests when all the old pods are destroyed.

This deployment strategy is better when the engineers want to push a major update of the application instead of a minor or a patch update. With this strategy, there is no shifting of traffic to another version as shown in the canary, and blue-green deployments.

The Recreate deployment strategy is suitable for environments where customers or users do not mind short but announced downtimes. With the recreate deployment strategy, planned maintenance schedules are set, and emails or push notifications are sent to the users informing them of a short downtime.

Take a look at the YAML file called recreate-deploy.yaml

apiVersion: apps/v1 #Older versions of k8s use apps/v1beta1  
kind: Deployment  
metadata:  
 name: hello-deploy  
spec:  
 replicas: 10  
 selector:  
 matchLabels:  
 app: hello-world  
 minReadySeconds: 10  
 strategy:  
 type: Recreate  
 template:  
 metadata:  
 labels:  
 app: hello-world  
 spec:  
 containers:  
 - name: hello-Pod  
 image: obusorezekiel/node-app:latest  
 ports:  
 - containerPort: 8080

Here, the .spec.strategy tells Kubernetes how to perform updates to the Pods managed by the Deployment, in this case, Recreate. To apply, simply run kubectl apply -f recreate-deploy.yaml and watch how to pods are created/updated.

**2. Rolling Update Deployment Strategy**

The rolling update is the default deployment strategy in Kubernetes that update the pods one by one without causing cluster downtime.

In the running update deployment strategy, the Kubernetes deployments update all running pods of the application to a new version one by one. Here, the deployment objects create new pods using the new pod specifications and shut down the older ones when the application starts running successfully. Engineers can safely transition from one version of the application to another without production errors.

To trigger the rolling update simply update the image in your pods using the set command, or right in your YAML file. This will automatically trigger an update called the rolling update.

For the rolling update deployment strategy, there is usually zero downtime. However, it is also difficult to roll back to a previous version because it also follows the same “one-by-one” process.

Take a look at the YAML file below.

apiVersion: apps/v1 #Older versions of k8s use apps/v1beta1  
kind: Deployment  
metadata:  
 name: hello-deploy  
spec:  
 replicas: 10  
 selector:  
 matchLabels:  
 app: hello-world  
 minReadySeconds: 10  
 strategy:  
 type: RollingUpdate  
 rollingUpdate:  
 maxUnavailable: 1  
 maxSurge: 1  
 template:  
 metadata:  
 labels:  
 app: hello-world  
 spec:  
 containers:  
 - name: hello-Pod  
 image: obusorezekiel/node-app:latest  
 ports:  
 - containerPort: 8080

In the YAML file above, .spec.strategy tells Kubernetes how to perform updates to the Pods managed by the Deployment, in this case, RollingUpdate.

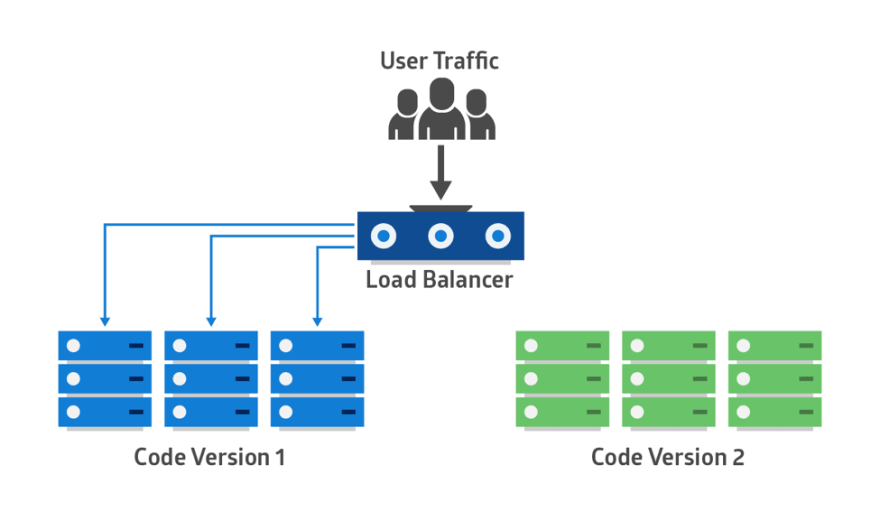
To apply, simply run kubectl apply -f recreate-deploy.yaml and watch how to pods are created/updated.

**3. Blue/Green Deployment Strategy in Kubernetes**

Although it is not an out-of-the-box Kubernetes strategy, this is also an important deployment strategy for proper software delivery into production.

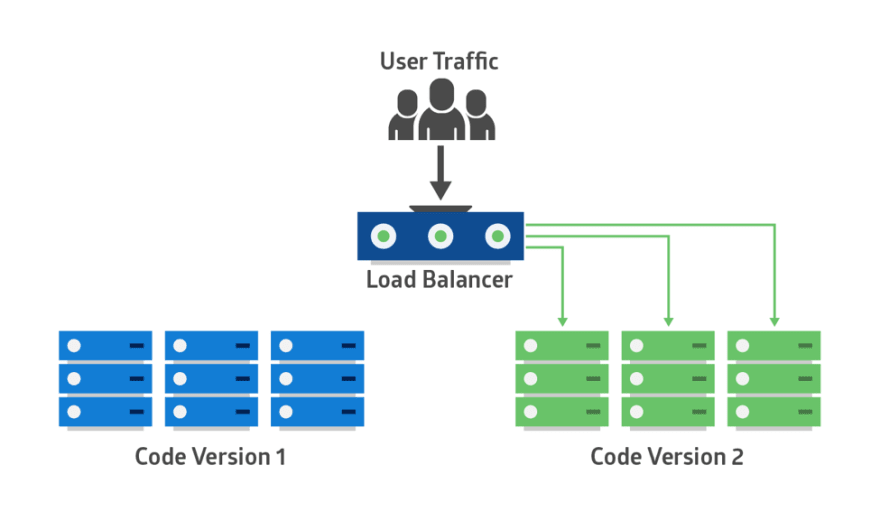
For this context, “Green” refers to the new version of the software that is deployed alongside the old version called the “Blue” version.

While the “Blue” version is running, engineers deploy the “green” version in another deployment and run basic tests to ensure that the pods within the deployments are stable, then using services, you switch to the newer version of the application.



Blue Code Version. Image Source: [Jason Skowronski](https://dev.to/mostlyjason)

One reason why a lot of engineers prefer using the Blue/Green version is because of rapid releases. If you are working within a CI/CD environment, the Blue/Green approaches will help you to move your applications into production quickly. As a DevOps engineer, you can run updates, and rollbacks without associated downtime.

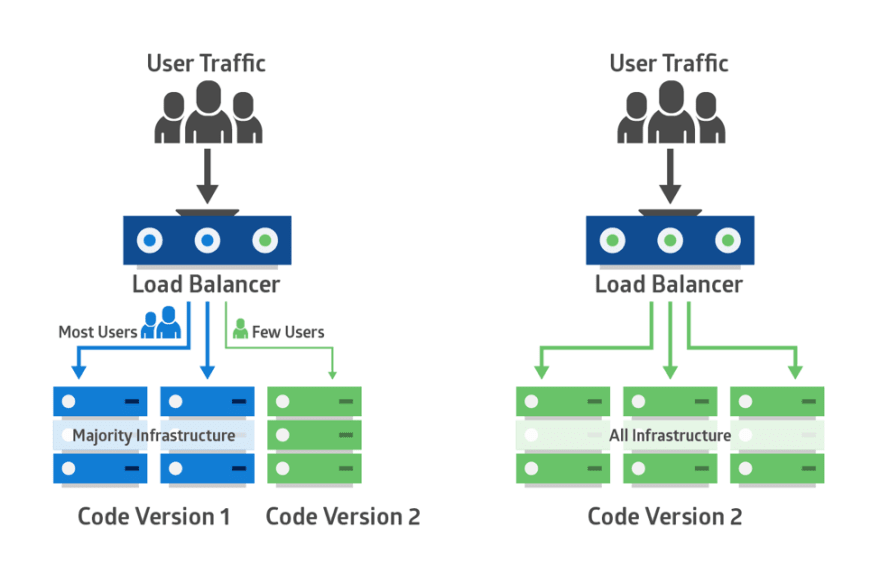


Green Code Version. Image Source: [Jason Skowronski](https://dev.to/mostlyjason)

However, there are some downtimes when it comes to working with the Blue/Green and one of them is cost and robust data migration challenges. For a typical Blue/Green deployment strategy to work, you’ll need to double up all production requirements, and if the Kubernetes infrastructure is hosted on the cloud, you’d need to pay for double.

Also, it may be difficult to keep your data evenly synchronized. That is why data migration may be difficult.

**4. Canary Deployment Strategy**



Canary Deployment. Image Source: [Jason Skowronski](https://dev.to/mostlyjason)

In a typical canary deployment, a newer version of the application is deployed to a small subset of people in the live Kubernetes cluster. A small subset of live users will be connected to the application while the rest will still be using the previous version.

The main aim of deploying to the small subset of users is for early error detection and potential problems that are present in the new code. As the software integrity improves, more canaries are created, and users will now be introduced to the new version. At the end of it all, all the live traffic will be introduced to the new version.

The main advantage of the canary deployment is that engineers can easily detect application issues very early with a small subset of users. Once any problem shows up in the canary, the production is still present, and you can easily work on the new version.

Within a Kubernetes cluster, adopting the Canary deployment may require a proper understanding of Istio. Istio is a service mesh that helps to properly route traffic and set security configurations according to your own rules. Although you can set traffic configurations without Istio, Istio makes the overall Configuration better.

**Deployment Best Practices in Kubernetes for DevOps Engineers**

1. **Enable Automated rollbacks for your deployments.**

For all deployments implemented in the Kubernetes cluster, automated rollbacks are important so that whenever a pod fails to run in a new deployment, the Kubernetes cluster will always fall over to the previous deployment for users.

**2. Implement alerting and communication channels.**

Alert managers, and communication channels like Slack help you with automated notifications for both failed and successful deployments. This will help you put your deployments in better order.

**3. Adopt Continous Integrations and Continous Delivery.**

Using CI ensures that your code is properly tested, no errors along to pipeline, and build path, and ensures a successful build.

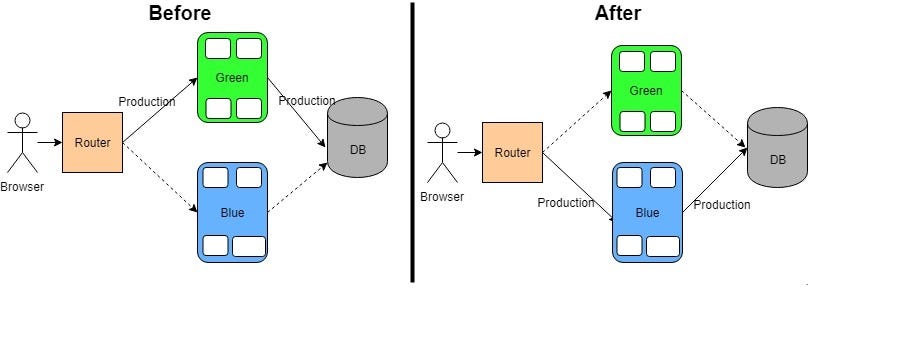
**Blue-Green Deployments with Kubernetes**



In the modern world of software development, deploying new versions of applications is a crucial part of the development cycle. However, rolling out updates to production environments can be a risky proposition, as even small issues can result in significant downtime and lost revenue. Blue-Green Deployments are a deployment strategy that mitigates this risk by ensuring that new versions of applications can be deployed with zero downtime.

In this blog, we will discuss how Blue-Green Deployments can be implemented using Kubernetes, one of the most popular container orchestration platforms. We will cover the steps involved in setting up a Blue-Green Deployment in Kubernetes, along with the benefits of using this strategy.

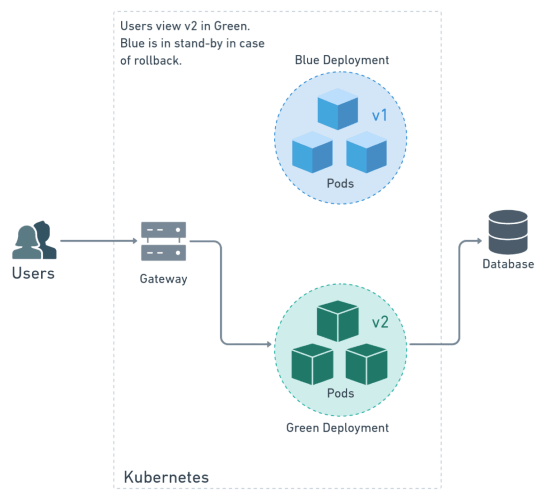
**What are Blue-Green Deployments**



A Blue-Green Deployment is a deployment strategy where two identical environments, the “blue” environment and the “green” environment, are set up. The blue environment is the production environment, where the live version of the application is currently running, and the green environment is the non-production environment, where new versions of the application are deployed.

When a new version of the application is ready to be deployed, it is deployed to the green environment. Once the new version is deployed and tested, traffic is switched to the green environment, making it the new production environment. The blue environment then becomes the non-production environment, where future versions of the application can be deployed.

**Setting up Blue-Green Deployments in Kubernetes**



The following are the steps involved in setting up Blue-Green Deployments in Kubernetes:

**Step 1: Create the Blue Deployment**

The first step in setting up a Blue-Green Deployment is to create the blue deployment. To create the blue deployment, use the following YAML code:

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: myapp:1.0  
 ports:  
 - containerPort: 80

This YAML code creates a deployment with three replicas that selects the blue environment and uses the myapp:1.0 image.

**Step 2: Create the Blue Service**

The second step is to create the blue service. To create the blue service, use the following YAML code:

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

This YAML code creates a service that selects the blue environment and maps traffic from port 80 to the target port 80 of the blue deployment.

**Step 3: Verify the Blue Deployment**

The third step is to verify that the blue deployment is working correctly. To do this, use the following command:

kubectl get pods -l app=myapp,env=blue

This command should return the three replicas of the blue deployment.

**Step 4: Create the Green Deployment**

The fourth step is to create the green deployment. To create the green deployment, use the following YAML code:

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: myapp:2.0  
 ports:  
 - containerPort: 80

This YAML code creates a deployment with zero replicas that selects the green environment and uses the myapp:2.0 image.

Note that the replicas are set to zero because we don’t want traffic to be routed to the green deployment yet.

**Step 5: Create the Green Service**

The fifth step is to create the green service. To create the green service, use the following YAML code:

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

This YAML code creates a service that selects the green environment and maps traffic from port 80 to the target port 80 of the green deployment.

**Step 6: Verify the Green Deployment**

The sixth step is to verify that the green deployment is working correctly. To do this, use the following command:

kubectl get pods -l app=myapp,env=green

This command should return zero replicas, as we have not yet scaled the green deployment.

**Step 7: Scale the Green Deployment**

The seventh step is to scale the green deployment. To scale the green deployment, use the following command:

kubectl scale deployment green --replicas=3

This command scales the green deployment to three replicas.

**Step 8: Verify Traffic Routing**

The eighth step is to verify that traffic is correctly routed to the blue deployment. To do this, use the following command:

kubectl describe service blue | grep Endpoints

This command should return the IP addresses of the three replicas of the blue deployment.

**Step 9: Switch Traffic to the Green Deployment**

The ninth step is to switch traffic to the green deployment. To do this, use the following command:

kubectl apply -f green-service.yaml

This command applies the green service YAML code, which maps traffic to the green deployment.

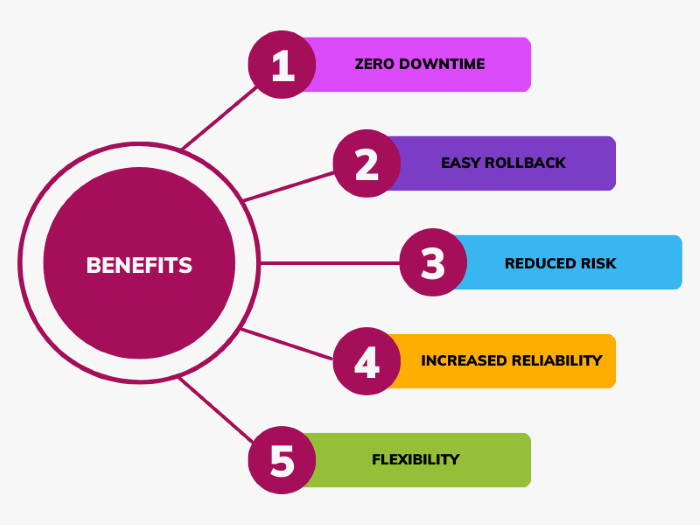
**Step 10: Verify Traffic Routing**

The final step is to verify that traffic is correctly routed to the green deployment. To do this, use the following command:

kubectl describe service green | grep Endpoints

This command should return the IP addresses of the three replicas of the green deployment.

**Benefits of Blue-Green Deployments**



1. **Zero Downtime:**Blue-Green Deployments allow new versions of applications to be deployed with zero downtime, as traffic is switched from the blue environment to the green environment seamlessly.
2. **Easy Rollback:**If a new version of the application has issues, rolling back to the previous version is easy, as the blue environment is still available.
3. **Reduced Risk:** By using Blue-Green Deployments, the risk of deploying new versions of applications is reduced significantly. This is because the new version can be deployed and tested in the green environment before traffic is switched over from the blue environment. This allows for thorough testing and reduces the chance of issues arising in production.
4. **Increased Reliability:** By using Blue-Green Deployments, the reliability of the application is increased. This is because the blue environment is always available, and any issues with the green environment can be quickly identified and resolved without affecting users.
5. **Flexibility:** Blue-Green Deployments provide flexibility in the deployment process. Multiple versions of an application can be deployed side-by-side, allowing for easy testing and experimentation.

**Conclusion**

To sum up, deployment strategies in Kubernetes are a critical component in ensuring the efficient deployment of containerized applications. The various deployment strategies available, such as recreate, rolling updates, blue-green deployment, and canary

deployments, provide a level of flexibility and control that is essential for managing complex application deployments.

Each strategy has its strengths and weaknesses, and the decision on which to use depends on the specific needs of the application and business goals. However, regardless of which deployment strategy is chosen, Kubernetes provides a reliable and scalable platform that can ensure that applications are always available and performing optimally.

Moreover, understanding deployment strategies in Kubernetes is not only essential for managing deployments but also for maintaining the overall health and resilience of the application. By employing the right deployment strategy, organizations can significantly reduce the risk of downtime and other issues that can impact business operations.

The various deployment strategies available in Kubernetes provide organizations with a range of options for managing containerized application deployments. From rolling updates to canary deployments, each strategy has its benefits and limitations. Still, they all provide a level of control and flexibility that is essential for effectively managing complex applications at scale.

By understanding these deployment strategies and selecting the appropriate one for your application, you can ensure that your applications are always available, reliable, and scalable, meeting the needs of both the business and end-users.