

Ways to Perform Effective Blue Green Deployment in Azure Kubernetes

Author – Santanu De



Introduction

A blue green deployment model is an application release model intended to instantly transfer the traffic from the previous application version to the new version.

Azure Kubernetes Service (AKS) is a container service that simplifies the deployment, management, and operations of Kubernetes as a fully managed Kubernetes container orchestrator service.

As there is no built-in functionality in AKS to facilitate blue green deployment, it will require some planning and modifications in Helm files and object Yaml's to implement blue green deployment in AKS.

Here we are going to discuss different ways of switching between the two versions using different objects of Kubernetes.

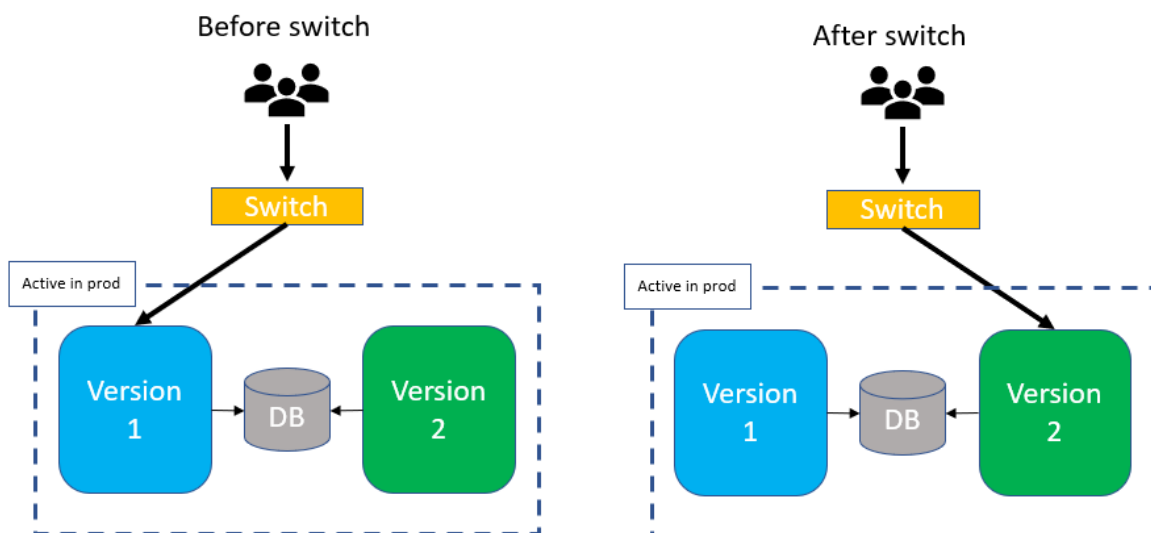


Figure 1

Objective

To achieve zero downtime during application version upgrade in production which basically means that there should be instant shift of traffic from old to new version without affecting online users.

It is important to check whether the new version is working as expected in production environment before making this shift. If required, we should also be able to roll back to the previous version in zero down time and without any hassle. Above points make blue-green application release model important for production release to the developers and operation teams compared to other models like Recreate, Rolling-update or incremental and Canary model.

So, how does the blue green deployment model work?

To understand how the blue green deployment model works let's consider an example where you need to make some minor changes in the version 1 and create version 2. Version 1 here is our blue version active in the production whereas version 2 is the green version. The green version is deployed and tested thoroughly in the production environment to ensure proper functioning of the application. In this case, both the green

and the blue could be accessed using different URLs. The blue uses the production URL, while the green uses a test URL known to the test team only. Once testing is completed and decision taken to make the green as production version a switch over of traffic is done between blue and green versions. That means now (post switch) green will have production URL whereas blue will have the test URL.

Once the green environment is up and running for a predefined time and the team is satisfied with its working, the blue version is deleted to save resources.

So, at a high level the following steps are involved:

1. The green version is made available in the production environment with test URL.
2. Test this green version thoroughly.
3. Transfer traffic to the green version.
4. If the green version develops any snag within trial period, transfer traffic back to the blue version which was kept as standby
5. Deactivate/Delete the blue version after ensuring that the new (green) version is working as expected.

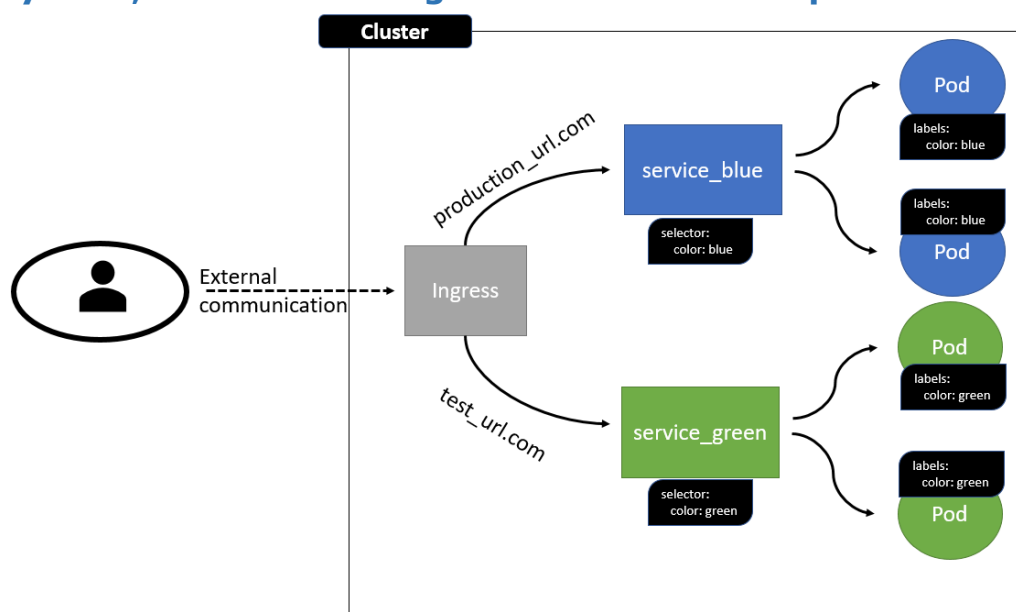
Three strategies for achieving the blue green deployment in Kubernetes

The default deployment strategy in Kubernetes is Rolling-update deployment, where pods with old version are replaced by pods of new version gradually. That means some of the old and new version pods will remain active during the deployment. To ensure a clean shift of traffic to the new version rollover deployment is generally not used in Production environment.

To enable blue green deployment in Kubernetes, three objects will be used **namely deployment, service, and ingress controller**.

Now let's try to explore different approaches for deployment and switching with minimal modifications or changes in either of the three objects mentioned above.

How the communication between Kubernetes objects i.e., deployment, services and ingress controller take place?



Whenever some request is coming for the application reaches ingress, the ingress routes the request based on the host header.

For example, in our case there are two hosts namely production_url.com and test_url.com. So, the above ingress routes traffic requested for production_url.com to service blue and for test_url.com it routes to service_green.

Now these services routes traffic across a set of pods where our application is residing. These services match with the set of pods by using labels and selectors. The labels are attached to the deployment object and these same labels are written under selector in service.

For example, in our case the service service_blue has selector "color: blue" and the pods has label "color: blue". So, whenever traffic is routed towards service_blue, this service routes the traffic to this set of pods.

In short:

To connect the deployment (the pods created) with the service:

1. The service's targetPort should match the containerPort of the deployment.
2. The service selector should match with pods label.

To connect the service with the ingress:

1. The service.port of the ingress should match with the port of the service.
2. Also, the service.name of the ingress should match the name of the service.

Approach 1 – Switch at Deployment level

Initially both the versions are deployed and running in the AKS cluster, blue version with the original URL and green version with the test URL.

The name of the deployment, service and ingress should contain the version it is referring to. For example, the service for blue version can be named service_blue, deployment as app_blue, etc.

How to make a switch?

To make a switch in the deployment we will change the version of application and database (if exists).

Model example

Pipeline 1

- Deploying a new Tenant for a new client

Deployment

name: app_blue (V1.0) # the name value to be supplied to deployment Yaml from Helm file

label:

app: demo

color: Blue

spec:

containers:

image: V1

Service

name: app_service_prod

selector:

app: demo

color: Blue

Ingress

```
HOST URL: <production_url>
backend:
  service-Name: app_service_prod
```

- Deploying new Version - if Deployment in Production is available for the Tenant then new deployment is:

Deployment

```
name: app_green (V2.0)
label:
  app: demo
  color: Green
spec:
  containers:
    image: V2
```

Service

```
name: app_service_stag
selector:
  app: demo
  color: Green
```

Ingress

```
- host: <test_url>
  backend:
    service:
      name: app_service_stag
```

Pipeline 2

- After successful testing of the Staging deployment (new Version2.0). Deploy the New Version V2.0 as app_blue and the old version (V1.0) as app_green . To minimize change in the YAMIs and Helm Files exchange the name of deployments and color values under label section.

Deployment

Name: app_blue (New Version Green -> Blue) (V2.0)

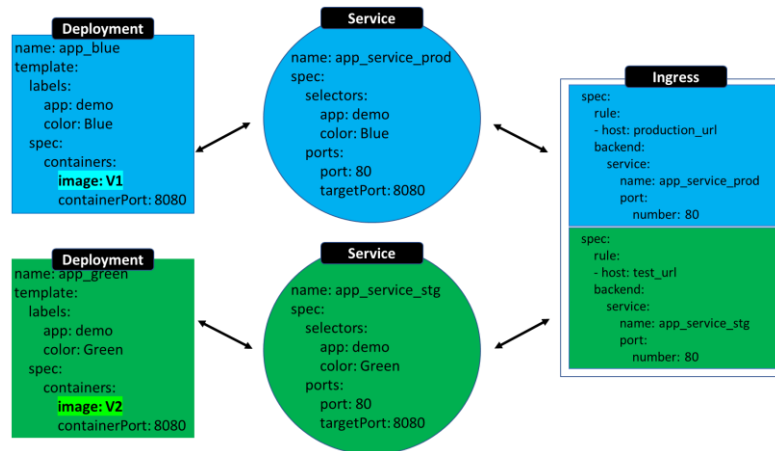
```
label:
  app: demo
  color: Blue
spec:
  containers:
    image: V2
```

Deployment

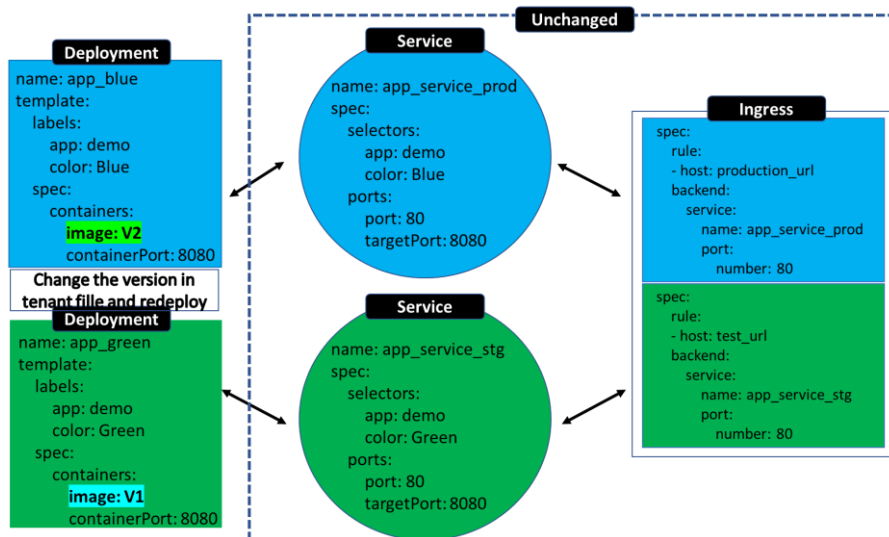
Name: app_green (Old Version Blue -> Green) (V1.0)

```
label:
  app: demo
  color: Green
spec:
  containers:
    image: V1
```

Diagrammatical representation



Pic-Deploy-1 Before switch



Pic Deploy-2 After switch

Pros- In this case, as a fresh deployment of pods are done the name of the deployments could be consistent before and after switch. In all scenarios, the production version is Blue and test version is green. Production service (app_service_prod) is always talking to app_blue. We may even name it app_ Prod. Similarly, the app_green may be termed as app_stg which is linked to ass_service_stg.

Cons- The fresh pod deployment takes few minutes to be in working condition.

To avoid fresh deployment of pods next two approaches could be adopted.

Approach 2 – Switch in Service only

Initially we have our deployments for both the version actively running in the AKS cluster, blue version on the original URL and green version on the test URL.

The name of the service and ingress should contain the color of deployment it is referring to. For example, the service for blue environment can be named `service_blue`, deployment as `app_blue`

The label of the deployment will have a 'color' variable in it to make a switch.

How to make a switch?

Service and Deployment (pods) in AKS are connected by the labels mentioned in the deployment template (template.labels.color) and those in the selector in the service (spec.selectors.color).

So, by changing the value of the label field in the service we can change the deployment it is pointing to. For example, see the figure 2, where just by changing the value of the label fruit in the service which changes the deployment it was pointing to. It is expected that the Port, targetPort, containerPort values are consistent for both the versions.

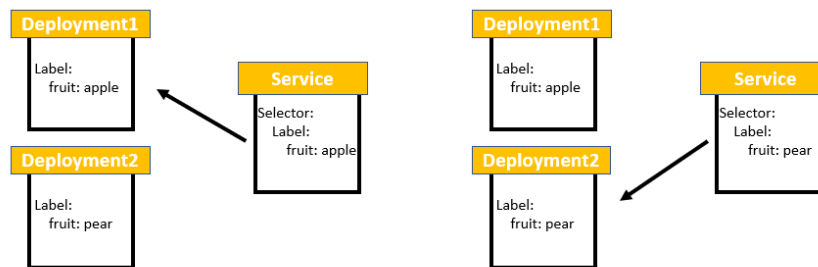


figure 2

Model example:

Pipeline 1

- Deploying a new Tenant for a new client (no existing Deployment means Production deployment will be done in the Tenant)

Deployment Name: `app_blue`

labels:

app: demo

color: Blue

Service Name: `app_service_prod`

selector:

app: demo

color: Blue

Ingress

HOST URL: `<production_url>`

backend:

Service-Name: `app_service_prod`

- Deploying new Version- if there is existing Deployment that means Production deployment is available in the Tenant, in that case, new Deployment is green deployment (all objects like Deployment, Service and Ingress etc. to be deployed for staging V2.0):

Deployment

```
name: app_green
  labels:
    app: demo
    color: Green
```

Service

```
name: app_service_stag
  selector:
    app: demo
    color: Green
```

Ingress

```
host: <test_url>
  backend:
    Service:
      name: app_service_stag
```

Pipeline 2: After successful testing of the Staging deployment (new Version2.0)

- Switch the Service **Selectors Key: value pairs** only keeping Deployment and Ingress configurations as it is:

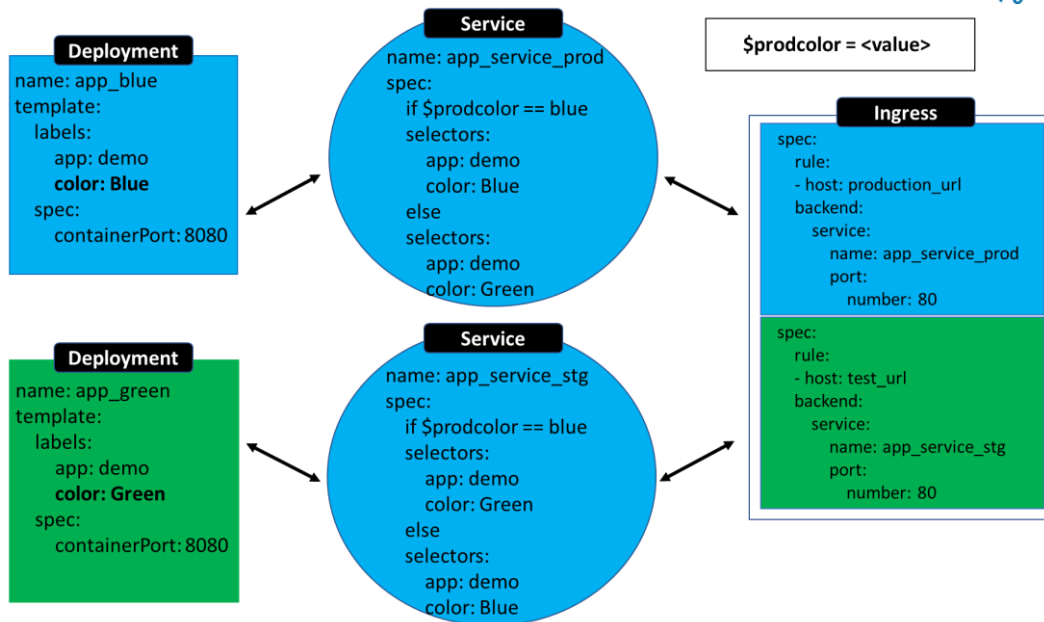
Services

```
name: app_service_prod
  if $prodcOLOR==blue
    selectors:
      app: demo
      color: Blue
  else
    selectors:
      app: demo
      color: Green
```

Service

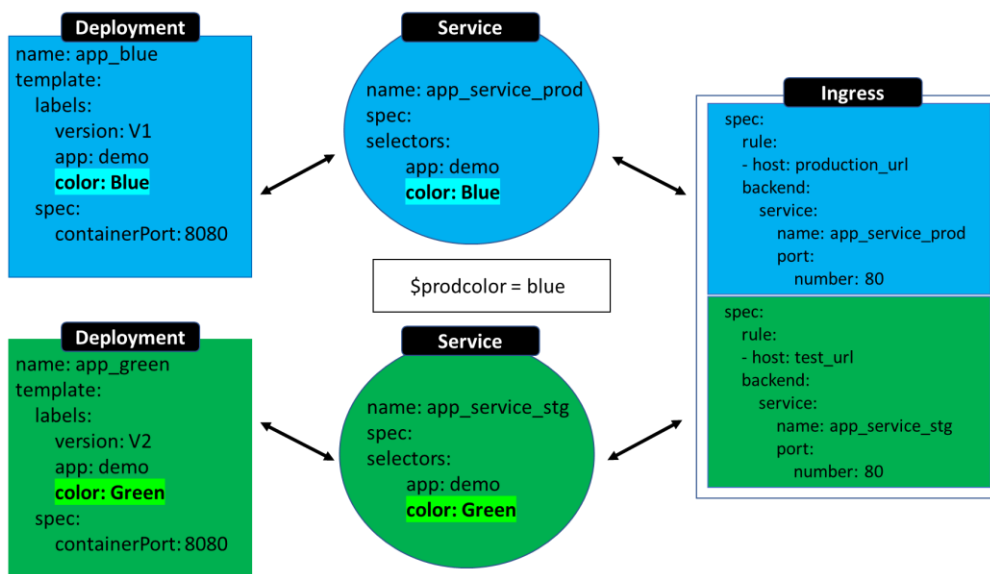
```
name: app_service_stag
  if $prodcOLOR==blue
    selectors:
      app: demo
      color: Green
  else
    selectors:
      app: demo
      color: Blue
```

Diagrammatical representation:

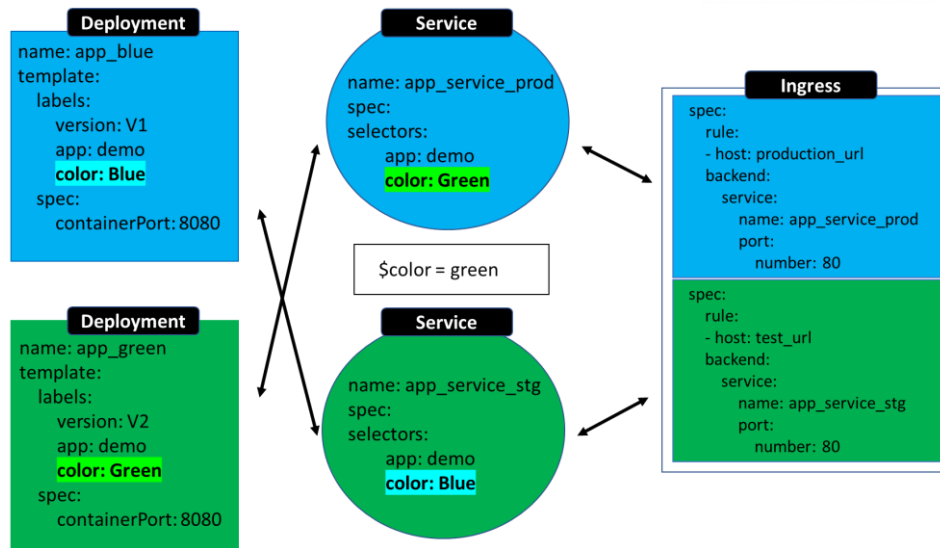


Pic- Service-1

Reference diagram for switch at service



Pic-Service-2 Before Switch



Pic- Service-3 After Switch

Pros- No fresh pod deployment, quick to be in effect. **Only spec.selectors.color values** to be changed for app_service_prod and app_service_stg to make the switchover effective.

Cons- The **name of production version of Deployment object toggles** between app_blue and app_green.

Approach 3 – Switch in Ingress only

To connect the service with the ingress:

1. The service.port of the ingress should match with the port of the service.
2. Also, the service.name of the ingress should match the name of the service.

The ingress connects with the service mentioned in the backend under Host URL of the ingress template. Now one service will be under production URL and other will be under testing URL.

How to make a switch?

So, for the switch in the ingress, we can make a new variable whose value we can provide from Jenkins or other DevOps tool. According to the value of variable we can make a switch.

Model Example:

Pipeline 1

- Deploying a new Tenant for a new client (no existing Deployment means new deployment is the Production version in the Tenant)

Deployment

```

name: app-blue
label:
  app: demo
  color: Blue
  
```

Service

```

name: app_service_blue
Selector:
  app: demo
  
```

Color: Blue

Ingress

- host: <production_url>

 backend:
 Service:
 name: app_service_blue
 port: 80

- Deploying new Version if the existing Deployment is Production deployment (say blue) in the Tenant then the new Deployment will be a staging deployment(say green)

Deployment

name: Dep-AppName-V2.0
label:
 app: demo
 color: Green

Service

name: app_service_green
selector:
 app: demo
 color: Green

Ingress

- host: <test_url>

 backend:
 Service-Name: app_service_green
 port: 80

Pipeline 2 – After successful testing of the Staging deployment (new Version2.0)

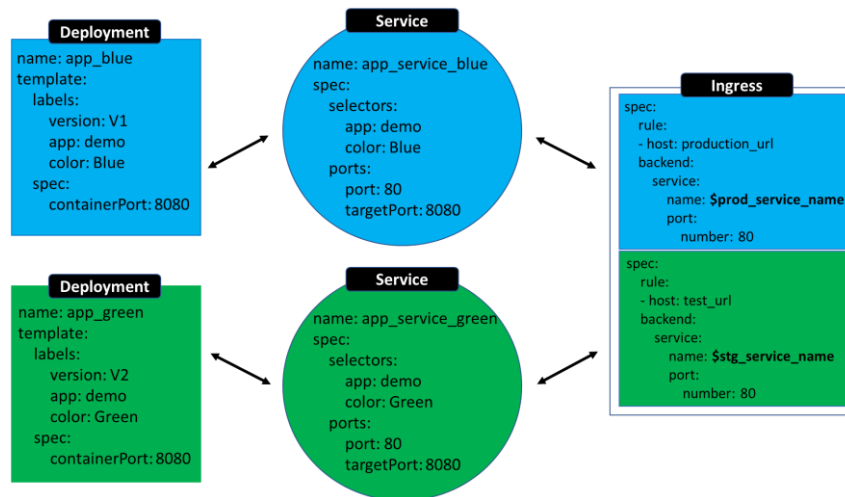
- Switch the **Service Name under Host entry in NGNEX** only keeping **Deployment and Service configurations as it is:**

```
spec:  
    - host: <production_url>  
  
    if switch==blue  
        backend:  
            service:  
                name: app_service_blue  
                port: 80  
    else  
        backend:  
            service:  
                name: app_service_green  
                port: 80  
  
    - host: <test_url>  
  
    If switch==blue  
        backend:  
            service:  
                name: app_service_green  
                port: 80
```

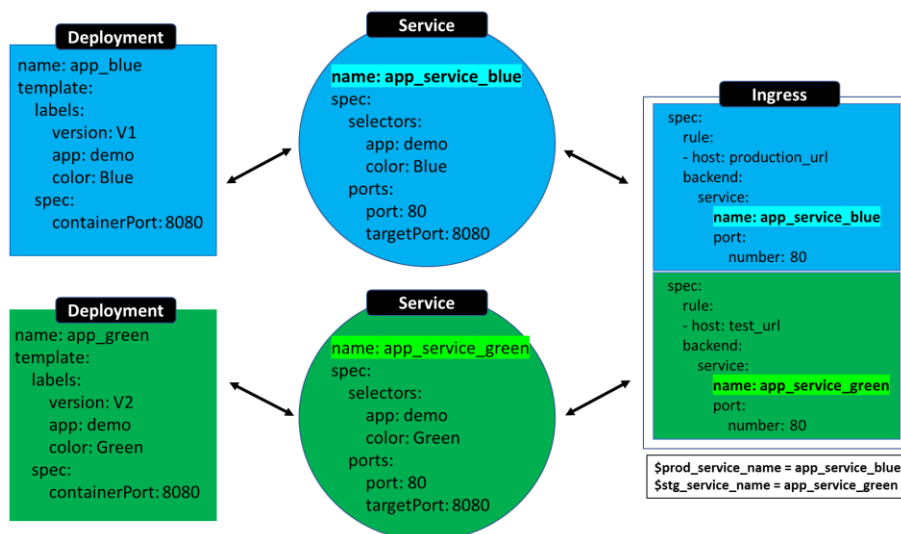
else

```
backend:
  service:
    name: app_service_blue
    port: 80
```

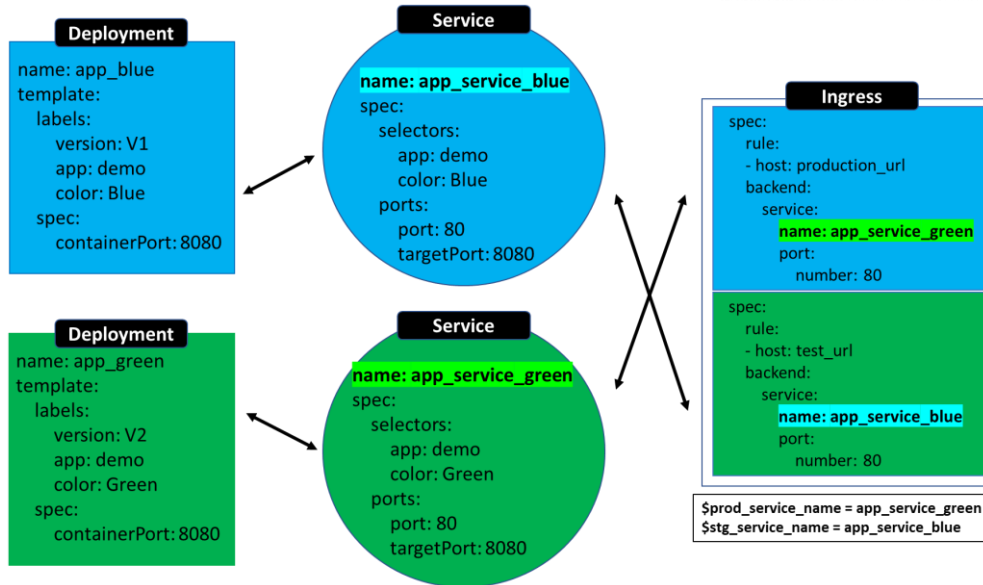
Diagrammatical representation:



Pic- Ingress-1 Reference diagram for switch at ingress



Pic-Ingress-2 Before Switch



Pic- Ingress-3 After Switch

Pros- No fresh pod deployment required, switch over is quick to be in effect. Only **backend.service.name** values to be changed for - **host: production_url** and - **host: test_url** to make the switchover effective.

Cons-

- 1) The **name of production version of Deployment object toggles** between app_blue and app_green.
- 2) The **name of service under Production URL toggles** between app_service_prod and app_service_stg. The same toggles happened under Staging URL. You must look at the Ingress configuration to find out production version.

The Pros and Cons of the three approaches are given in a table as below:

Approach #	Switch in Kubernetes Object	Pros	Cons
1	Deployment	1) Production version is always blue and test version is always green 2) Production service (app_service_prod) is always talking to Deployment app_blue and could be named app_prod	The fresh pod deployment takes few minutes to be in working condition
2	Service	1) No fresh pod deployment, quick to be in effect. 2) Only spec.selectors.color values to be changed for app_service_prod and app_service_stg to make the switchover effective.	The name of production version of Deployment object toggles between app_blue and app_green. You must look at the app_service_prod configuration to find out production version.
3	Ingress	1) No fresh pod deployment required, switch over is quick to be in effect. 2) Only backend.service.name values to be changed for - host: production_url and - host: test_url to make the switchover effective.	1) The name of production version of Deployment object toggles between app_blue and app_green. 2) The name of service under Production URL toggles between app_service_blue and app_service_green. The same toggles happened under Staging URL. You must look at the Ingress configuration to find out production version.

Table-1 Comparison of Pros and Cons of 3 approaches

Conclusion: Though there is no default mechanism for blue green deployment in AKS, it can be easily achieved by changing specific key-value pairs in Kubernetes object YAMLS of Deployment, Service, or Ingress and that can be easily achieved if Helmfile is used to deploy Helm Charts and related Yaml's. Out of the three approaches given above one may select anyone based upon their comfort level to manage releases in Kubernetes. The first two approaches could also be adopted for service type NordPort, LoadBalancer or ExternalName. Both blue and green versions are deployed independent of each other and can be deleted or redeployed without affecting the other.

If all the objects are in one namespace or cross namespace communications are allowed approach 1,2 and 3 are possible. As approach 2,3 are quick to put change over in action, one of them is recommended say approach 2 (change in service object).

If blue and green versions are deployed in separate namespaces, approach 1 is applicable solution.

The purpose of the white paper was to find different ways to tackle Blue Green deployment in Kubernetes cluster. Here three easy to manage approaches have been described with important Key- Value pairs to deal with in each of them. If you have any query, please contact me at santanu.de@capgemini.com or desantanu@gmail.com.

About Sogeti

Sogeti is a leading provider of technology and engineering services. Sogeti delivers solutions that enable digital transformation and offers cutting-edge expertise in Cloud, Cybersecurity, Digital Manufacturing, Digital Assurance & Testing, and emerging technologies. Sogeti combines agility and speed of implementation with strong technology supplier partnerships, world class methodologies and its global delivery model, Rightshore®. Sogeti brings together more than 25,000 professionals in 15 countries, based in over 100 locations in Europe, USA and India. Sogeti is a wholly-owned subsidiary of Capgemini SE, listed on the Paris Stock Exchange.