


# Managing Deployments Using Kubernetes Engine

## Activate Cloud Shell

Cloud Shell is a virtual machine that is loaded with development tools. It offers a persistent 5GB home directory and runs on the Google Cloud. Cloud Shell provides command-line access to your Google Cloud resources.

1. Click **Activate Cloud Shell**  at the top of the Google Cloud console.
2. Click through the following windows:
  - Continue through the Cloud Shell information window.
  - Authorize Cloud Shell to use your credentials to make Google Cloud API calls.

When you are connected, you are already authenticated, and the project is set to your **Project\_ID**, `PROJECT_ID`. The output contains a line that declares the **Project\_ID** for this session:

```
Your Cloud Platform project in this session is set to "PROJECT ID"
gcloud is the command-line tool for Google Cloud. It comes pre-installed on Cloud Shell
and supports tab-completion.
```

3. (Optional) You can list the active account name with this command:

```
gcloud auth list
```

Copied!

content\_copy

4. Click **Authorize**.

**Output:**

```
ACTIVE: *
ACCOUNT: "ACCOUNT"

To set the active account, run:
$ gcloud config set account `ACCOUNT`
```

5. (Optional) You can list the project ID with this command:

```
gcloud config list project
```

Copied!

content\_copy

### Output:

```
[core]
project = "PROJECT_ID"
```

**Note:** For full documentation of `gcloud`, in Google Cloud, refer to [the gcloud CLI overview guide](#).

## Set the zone

Set your working Google Cloud zone by running the following command, substituting the local zone as `ZONE`:

```
gcloud config set compute/zone ZONE
Copied!
```

content\_copy

## Get sample code for this lab

1. Get the sample code for creating and running containers and deployments:

```
gsutil -m cp -r gs://spls/gsp053/orchestrate-with-kubernetes .
cd orchestrate-with-kubernetes/kubernetes
Copied!
```

content\_copy

2. Create a cluster with 3 nodes (this will take a few minutes to complete):

```
gcloud container clusters create bootcamp \
  --machine-type e2-small \
  --num-nodes 3 \
  --scopes "https://www.googleapis.com/auth/projecthosting,storage-rw"
Copied!
```

content\_copy

## Task 1. Learn about the deployment object

To get started, take a look at the deployment object.

1. The `explain` command in `kubectl` can tell us about the deployment object:  
`kubectl explain deployment`

Copied!

content\_copy

2. You can also see all of the fields using the `--recursive` option:  
`kubectl explain deployment --recursive`

Copied!

content\_copy

3. You can use the `explain` command as you go through the lab to help you understand the structure of a deployment object and understand what the individual fields do:  
`kubectl explain deployment.metadata.name`

Copied!

content\_copy

## Task 2. Create a deployment

1. Update the `deployments/auth.yaml` configuration file:  
`vi deployments/auth.yaml`

Copied!

content\_copy

2. Start the editor:  
`i`

Copied!

content\_copy

3. Change the image in the containers section of the deployment to the following:

```
...
containers:
- name: auth
  image: "kelseyhightower/auth:1.0.0"
...
```

Copied!

content\_copy

4. Save the auth.yaml file: press <Esc> then type:

:wq

Copied!

content\_copy

5. Press <Enter>. Now create a simple deployment. Examine the deployment configuration file:

```
cat deployments/auth.yaml
```

Copied!

content\_copy

Output:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: auth
spec:
  replicas: 1
  selector:
    matchLabels:
      app: auth
  template:
    metadata:
      labels:
        app: auth
        track: stable
    spec:
      containers:
      - name: auth
        image: "kelseyhightower/auth:1.0.0"
        ports:
        - name: http
          containerPort: 80
        - name: health
          containerPort: 81
...
```

Notice how the deployment is creating one replica and it's using version 1.0.0 of the auth container.

When you run the `kubectl create` command to create the auth deployment, it will make one pod that conforms to the data in the deployment manifest. This means you can scale the number of Pods by changing the number specified in the `replicas` field.

6. Go ahead and create your deployment object using `kubectl create`:

```
kubectl create -f deployments/auth.yaml
```

Copied!

content\_copy

7. Once you have created the deployment, you can verify that it was created:

```
kubectl get deployments
```

Copied!

content\_copy

8. Once the deployment is created, Kubernetes will create a `ReplicaSet` for the deployment.

You can verify that a `ReplicaSet` was created for the deployment:

```
kubectl get replicaset
```

Copied!

content\_copy

You should see a `ReplicaSet` with a name like `auth-xxxxxxx`

9. View the Pods that were created as part of the deployment. The single Pod is created by the Kubernetes when the `ReplicaSet` is created:

```
kubectl get pods
```

Copied!

content\_copy

It's time to create a service for the auth deployment. You've already seen service manifest files, so the details won't be shared here.

10. Use the `kubectl create` command to create the auth service:

```
kubectl create -f services/auth.yaml
```

Copied!

content\_copy

11. Now, do the same thing to create and expose the `hello` deployment:

```
kubectl create -f deployments/hello.yaml
```

```
kubectl create -f services/hello.yaml
```

Copied!

content\_copy

12. And one more time to create and expose the `frontend` deployment:

```
kubectl create secret generic tls-certs --from-file tls/
```

```
kubectl create configmap nginx-frontend-conf --from-file=nginx/frontend.conf
```

```
kubectl create -f deployments/frontend.yaml
```

```
kubectl create -f services/frontend.yaml
```

Copied!

content\_copy

**Note:** You created a ConfigMap for the frontend.

13. Interact with the frontend by grabbing its external IP and then curling to it:

```
kubectl get services frontend
```

Copied!

content\_copy

**Note:** It may take a few seconds before the External-IP field is populated for your service. This is normal. Just re-run the above command every few seconds until the field is populated.

```
curl -ks https://<EXTERNAL-IP>
```

Copied!

content\_copy

And you get the hello response back.

14. You can also use the output templating feature of `kubectl` to use `curl` as a one-liner:

```
curl -ks https://`kubectl get svc frontend -  
o=jsonpath='{.status.loadBalancer.ingress[0].ip}'`
```

## Scale a deployment

Now that you have a deployment created, you can scale it. Do this by updating the `spec.replicas` field.

1. Look at an explanation of this field using the `kubectl explain` command again:

```
kubectl explain deployment.spec.replicas
```

Copied!

content\_copy

2. The replicas field can be most easily updated using the `kubectl scale` command:

```
kubectl scale deployment hello --replicas=5
```

Copied!

content\_copy

**Note:** It may take a minute or so for all the new pods to start up.

After the deployment is updated, Kubernetes will automatically update the associated `ReplicaSet` and start new Pods to make the total number of Pods equal 5.

3. Verify that there are now 5 `hello` Pods running:

```
kubectl get pods | grep hello- | wc -l
```

Copied!

content\_copy

4. Now scale back the application:

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

Copied!

content\_copy

5. Again, verify that you have the correct number of Pods:

```
kubectl get pods | grep hello- | wc -l
```

Copied!

content\_copy

Now you know about Kubernetes deployments and how to manage & scale a group of Pods.

## Task 3. Rolling update

Deployments support updating images to a new version through a rolling update mechanism. When a deployment is updated with a new version, it creates a new `ReplicaSet` and slowly increases the number of replicas in the new `ReplicaSet` as it decreases the replicas in the old `ReplicaSet`.

## Trigger a rolling update

1. To update your deployment, run the following command:

```
kubectl edit deployment hello
```

Copied!

content\_copy

2. Change the image in the containers section of the deployment to the following:

```
...
containers:
  image: kelseyhightower/hello:2.0.0
...
```

Copied!

content\_copy

3. **Save and exit.**

The updated deployment will be saved to your cluster and Kubernetes will begin a rolling update.

4. See the new `ReplicaSet` that Kubernetes creates.:

```
kubectl get replicaset
```

Copied!

content\_copy

5. You can also see a new entry in the rollout history:

```
kubectl rollout history deployment/hello
```

Copied!

content\_copy

## Pause a rolling update

If you detect problems with a running rollout, pause it to stop the update.

1. Run the following to pause the rollout:

```
kubectl rollout pause deployment/hello
```

Copied!

content\_copy

2. Verify the current state of the rollout:

```
kubectl rollout status deployment/hello
```

Copied!

content\_copy

3. You can also verify this on the Pods directly:

```
kubectl get pods -o jsonpath --template='{range .items[*]}{.metadata.name}{ "\t"}{ "\t"}{.spec.containers[0].image}{ "\n"}{end}'
```

Copied!

content\_copy

## Resume a rolling update

The rollout is paused which means that some pods are at the new version and some pods are at the older version.

1. Continue the rollout using the `resume` command:

```
kubectl rollout resume deployment/hello
```



Copied!

content\_copy

2. When the rollout is complete, you should see the following when running the status command:

```
kubectl rollout status deployment/hello
```

Copied!

content\_copy

Output:

```
deployment "hello" successfully rolled out
```

## Roll back an update

Assume that a bug was detected in your new version. Since the new version is presumed to have problems, any users connected to the new Pods will experience those issues.

You will want to roll back to the previous version so you can investigate and then release a version that is fixed properly.

1. Use the `rollout` command to roll back to the previous version:

```
kubectl rollout undo deployment/hello
```

Copied!

content\_copy

2. Verify the roll back in the history:

```
kubectl rollout history deployment/hello
```

Copied!

content\_copy

3. Finally, verify that all the Pods have rolled back to their previous versions:

```
kubectl get pods -o jsonpath --template='{range .items[*]}{.metadata.name}{ "\t"}{ "\t"}{.spec.containers[0].image}{ "\n"}{end} '
```

Copied!

content\_copy

Great! You learned how to do a rolling update for Kubernetes deployments and how to update applications without downtime.

# Task 4. Canary deployments

When you want to test a new deployment in production with a subset of your users, use a canary deployment. Canary deployments allow you to release a change to a small subset of your users to mitigate risk associated with new releases.

## Create a canary deployment

A canary deployment consists of a separate deployment with your new version and a service that targets both your normal, stable deployment as well as your canary deployment.

1. First, create a new canary deployment for the new version:

```
cat deployments/hello-canary.yaml
```

Copied!

content\_copy

Output:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-canary
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello
  template:
    metadata:
      labels:
        app: hello
        track: canary
        # Use ver 2.0.0 so it matches version on service selector
        version: 2.0.0
    spec:
      containers:
        - name: hello
          image: kelseyhightower/hello:2.0.0
          ports:
            - name: http
              containerPort: 80
            - name: health
              containerPort: 81
...

```

2. Now create the canary deployment:

```
kubectl create -f deployments/hello-canary.yaml
```

Copied!

content\_copy

3. After the canary deployment is created, you should have two deployments, `hello` and `hello-canary`. Verify it with this `kubectl` command:

```
kubectl get deployments
```

Copied!

content\_copy

On the `hello` service, the `app:hello` selector will match pods in **both** the prod deployment and canary deployment. However, because the canary deployment has a fewer number of pods, it will be visible to fewer users.

## Verify the canary deployment

1. You can verify the `hello` version being served by the request:

```
curl -ks https://`kubectl get svc frontend -o=jsonpath='{.status.loadBalancer.ingress[0].ip}'`/version
```

Copied!

content\_copy

2. Run this several times and you should see that some of the requests are served by `hello 1.0.0` and a small subset ( $1/4 = 25\%$ ) are served by `2.0.0`.

## Canary deployments in production - session affinity

In this lab, each request sent to the Nginx service had a chance to be served by the canary deployment. But what if you wanted to ensure that a user didn't get served by the canary deployment? A use case could be that the UI for an application changed, and you don't want to confuse the user. In a case like this, you want the user to "stick" to one deployment or the other.

You can do this by creating a service with session affinity. This way the same user will always be served from the same version. In the example below, the service is the same as before, but a new `sessionAffinity` field has been added, and set to `ClientIP`. All clients with the same IP address will have their requests sent to the same version of the `hello` application.

```
kind: Service
```

```
apiVersion: v1
metadata:
  name: "hello"
spec:
  sessionAffinity: ClientIP
  selector:
    app: "hello"
  ports:
    - protocol: "TCP"
      port: 80
      targetPort: 80
```

Due to it being difficult to set up an environment to test this, you don't need to here, but you may want to use `sessionAffinity` for canary deployments in production.

## Task 5. Blue-green deployments

Rolling updates are ideal because they allow you to deploy an application slowly with minimal overhead, minimal performance impact, and minimal downtime. There are instances where it is beneficial to modify the load balancers to point to that new version only after it has been fully deployed. In this case, blue-green deployments are the way to go.

Kubernetes achieves this by creating two separate deployments; one for the old "blue" version and one for the new "green" version. Use your existing `hello` deployment for the "blue" version. The deployments will be accessed via a service which will act as the router. Once the new "green" version is up and running, you'll switch over to using that version by updating the service.

**Note:** A major downside of blue-green deployments is that you will need to have at least 2x the resources in your cluster necessary to host your application. Make sure you have enough resources in your cluster before deploying both versions of the application at once.

## The service

Use the existing hello service, but update it so that it has a selector `app:hello,version:1.0.0`. The selector will match the existing "blue" deployment. But it will not match the "green" deployment because it will use a different version.

- First update the service:  
`kubectl apply -f services/hello-blue.yaml`  
Copied!

content\_copy

**Note:** Ignore the warning that says `resource service/hello is missing` as this is patched automatically.

## Updating using Blue-Green deployment

In order to support a blue-green deployment style, you will create a new "green" deployment for the new version. The green deployment updates the version label and the image path.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-green
spec:
  replicas: 3
  selector:
    matchLabels:
      app: hello
  template:
    metadata:
      labels:
        app: hello
        track: stable
        version: 2.0.0
    spec:
      containers:
        - name: hello
          image: kelseyhightower/hello:2.0.0
          ports:
            - name: http
              containerPort: 80
            - name: health
              containerPort: 81
          resources:
            limits:
              cpu: 0.2
              memory: 10Mi
          livenessProbe:
            httpGet:
```

```
    path: /healthz
    port: 81
    scheme: HTTP
    initialDelaySeconds: 5
    periodSeconds: 15
    timeoutSeconds: 5
  readinessProbe:
    httpGet:
      path: /readiness
      port: 81
      scheme: HTTP
    initialDelaySeconds: 5
    timeoutSeconds: 1
```

1. Create the green deployment:

```
kubectl create -f deployments/hello-green.yaml
```

Copied!

content\_copy

2. Once you have a green deployment and it has started up properly, verify that the current version of 1.0.0 is still being used:

```
curl -ks https://`kubectl get svc frontend -o=jsonpath="{.status.loadBalancer.ingress[0].ip}"`/version
```

Copied!

content\_copy

3. Now, update the service to point to the new version:

```
kubectl apply -f services/hello-green.yaml
```

Copied!

content\_copy

4. When the service is updated, the "green" deployment will be used immediately. You can now verify that the new version is always being used:

```
curl -ks https://`kubectl get svc frontend -o=jsonpath="{.status.loadBalancer.ingress[0].ip}"`/version
```

Copied!

content\_copy

## Blue-Green rollback

If necessary, you can roll back to the old version in the same way.

1. While the "blue" deployment is still running, just update the service back to the old version:

```
kubectl apply -f services/hello-blue.yaml
```

Copied!

content\_copy

2. Once you have updated the service, your rollback will have been successful. Again, verify that the right version is now being used:

```
curl -ks https://`kubectl get svc frontend -  
o=jsonpath='{.status.loadBalancer.ingress[0].ip}'`/version  
Copied!
```

content\_copy

You did it! You learned about blue-green deployments and how to deploy updates to applications that need to switch versions all at once.