DevOps – Laboratory 6

Container Orchestration

# This lab will focus on a very powerful tool called Kubernetes (k8s). Kubernetes allows you to deploy containerised applications in a highly scalable and managed way. In this lab you will learn how to use Kubernetes using a tool called Minikube which deploys a smaller version of the production version of Kubernetes and allows you to run k8s on a single VM. You will learn how to install Minukube and Kubectl (the command line interface for K8s), deploy an application on a K8s pod, scale it over multiple pods to deal with load demands and roll out updates while our application is still running.

# Getting Started

You will need to launch an AWS EC2 Instance on which you will complete this Lab. As k8s requires more processor cores (as two will be dedicated to k8s as a minimum and the other used for general processing) you will need to use a larger instance for this Lab.

Use the guidance in Lab 1 to configure an EC2 instance now but select the **t2.large** size. You do not need to open any other ports for this lab.

When you have successfully connected to your new VM via SSH you are ready to continue with this Lab.

**PLEASE REMEMBER TO STOP OR TERMINATE YOUR VM AT THE END OF THIS LAB SO THAT YOU DO NOT USE UNNECCESARY CREDIT!!**

# Important Information

This Lab documents will contain commands that you should run in the bash terminal (both locally and on your VM). This code will be provided in boxes as shown below:

|  |
| --- |
| This is a sample command that fits on one line  This is a much longer sample command that covers multiple lines but should still be copied as a single command |

As you can see, some commands will be short and clearly displayed on a single line. This can be copied directly into the Terminal. Other commands such as the second one are much longer and do not fit on one line. If you are going to copy and paste these commands make sure to copy all lines and paste as a single command. Each new command is separated by a blank line so that each command is clearly distinguished.

**If you are unfamiliar with using the Terminal or Bash it is highly recommended that you complete the following Tutorial:**

[**https://ubuntu.com/tutorials/command-line-for-beginners**](https://ubuntu.com/tutorials/command-line-for-beginners)

This tutorial is written for Ubuntu but as the Terminal we are using (GitBash for Windows or Terminal for MacOS/Linux) also uses Bash, the commands are the same and the tutorial is very relevant. The Virtual Machines you create on AWS will utilise the Ubuntu Operating system and therefore this tutorial will also be useful when interacting with your VMS.

Section 7 of this Tutorial covers SuperUser which is not applicable to Windows Systems but will be important on your VMs so reading is recommended.

The following Cheat Sheet is also helpful for remembering Bash commands and will be a good point of reference throughout this Module:

<https://cheatography.com/davechild/cheat-sheets/linux-command-line/pdf/>

**If you encounter any issued during this process, please contact the Module Tutor.**

# Installing Kubectl and Minikube

From this point in the Lab it is assumed that you have created an AWS EC2 VM and have connected to it via SSH. The remainder of this Lab you should perform all command line tasks on the VM. If you have not already done this, please see the information earlier in this Lab.

Minikube uses Docker to run a virtual environment within which k8s operates and as such you must have Docker installed before you proceed.

**If you have not already done so, install Docker using the instructions in Lab 4 and the script below.**

|  |
| --- |
| #! /bin/bash  sudo apt-get update  sudo apt-get -y remove unscd  sudo apt-get -y install apt-transport-https ca-certificates curl gnupg-agent software-properties-common  curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -  sudo apt-key fingerprint 0EBFCD88  sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb\_release -cs) stable"  sudo apt-get update  sudo apt-get -y install docker-ce docker-ce-cli containerd.io  sudo groupadd docker  sudo usermod -aG docker $USER  sudo addgroup --system docker  sudo adduser $USER docker  newgrp docker |

The first tool to install is Kubectl which is the command line interface that allows you to interact with the k8s cluster which you will learn more about later in this lab.

1. Install kubectl using the commands below:

|  |
| --- |
| sudo apt-get update  sudo apt-get install -y apt-transport-https  curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -  echo "deb https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee -a /etc/apt/sources.list.d/kubernetes.list  sudo apt-get update  sudo apt-get install -y kubectl |

1. Make sure that the installation was successful by running the ‘kubectl version’ command.

You should see information about the Client Version which tells you that the tool is installed but you will also see a message like:

**“The connection to the server localhost:8080 was refused - did you specify the right host or port?”**

This is because you do not yet have a Kubernetes cluster running for kubectl to interact with. This will be fixed during the next few installation steps.

1. You can now install Minikube, which is a tool that allows you to run Kubernetes locally, using the following commands:

|  |
| --- |
| curl -Lo minikube https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64  chmod +x minikube  sudo mv minikube /usr/local/bin/ |

1. Run the following command to start Minikube and pay attention to the terminal output to see what Minikube is doing. Some of the steps here may take some time to complete so be patient.

|  |
| --- |
| minikube start |

If you now run the ‘kubectl version’ command again you can see that the Client and Server are both configured. This concludes the installation stage and we can now focus on how we use Kubernetes.

# Kubernetes Clusters

A Kubernetes Cluster is a coordinated group of computers which is managed by K8s to work like a single unit. This means that you can run our containerized applications on a Cluster without having to worry about specifying a physical machine. K8s does all of the management for you and runs the container in the cluster where it deems most appropriate.

Clusters have two components; Master and Nodes. The Master manages the Cluster while the Nodes run the containers. See the image below for a visual description.

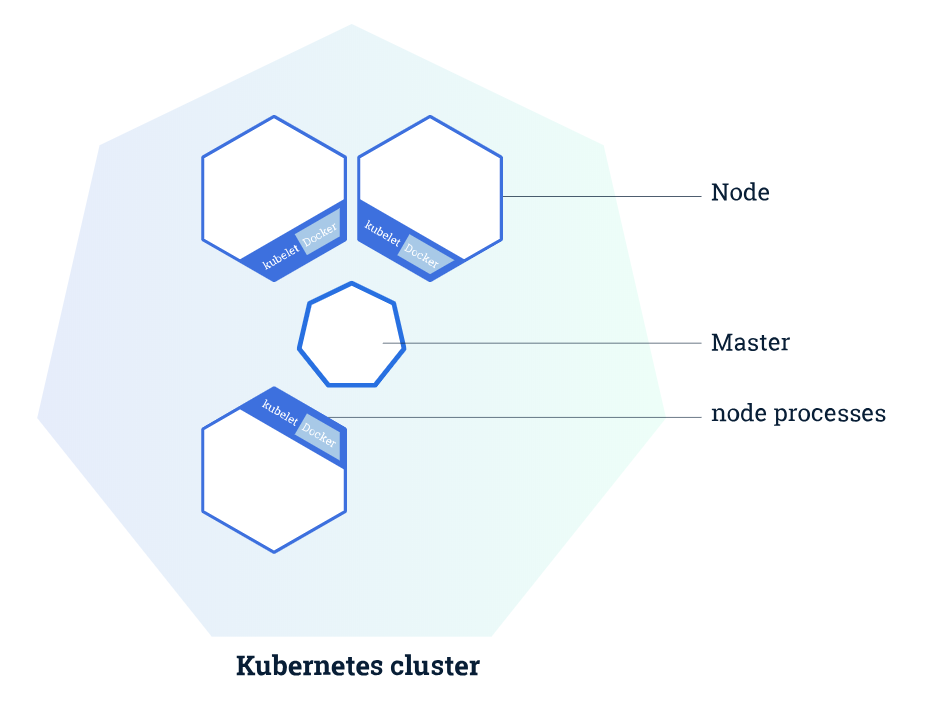


Figure : K8s Cluster

Each Node has a Kubelet which communicates with the Master and manages the Node. The Node also has a tool for managing containers which is commonly Docker (although other containerization tools can be used).

As mentioned previously, kubectl is the tool that you will use to interact with Kubernetes via the command line so the majority of the commands used from here on will be based on kubectl.

Try using the following command to see what nodes are currently running in k8s. You will come across variations of the ‘kubctl get’ command throughout this Lab:

|  |
| --- |
| kubectl get nodes |

As expected, you will only have one node running which is the Master Node. Minikube limits you to a single node as it is running on a single VM but the behavior would be identical if you were running the full version of k8s across multiple VMs.

# Kubernetes Deployments

A Kubernetes Deployment is essentially a configuration manager which tells Kubernetes how to manage the containerized application you deploy (which Node to run them on etc.) and it monitors them as they are running.

The Deployment is very powerful as it monitors all running containers and if a Node which is in use by a container goes down, the Deployment automatically relaunches the affected containers on an appropriate Node. The below image adds the Deployment and an example container to the cluster.

A close up of a logo

Description automatically generated

Figure : K8s Deployment

In this lab you will create a Deployment using a Node.js application which is packaged in a Docker container.

1. Create your first deployment as follows:

|  |
| --- |
| kubectl create deployment kubernetes-bootcamp --image=gcr.io/google-samples/kubernetes-bootcamp:v1 |

This command has pulled a sample image, created a container from that image, located a suitable node (you only have 1) and it has run the container on that node. The Deployment will also now monitor the running container and relaunch it if it goes down, or if the node goes down as discussed earlier.

This image is created from the following Node.js file and Dockerfile. You will not be working with these directly in this tutorial but it is important to understand how the application is built.

**server.js:**

|  |
| --- |
| var http = require('http');  var requests=0;  var podname= process.env.HOSTNAME;  var startTime;  var host;  var handleRequest = function(request, response) {  response.setHeader('Content-Type', 'text/plain');  response.writeHead(200);  response.write("Hello Kubernetes bootcamp! | Running on: ");  response.write(host);  response.end(" | v=1\n");  console.log("Running On:" ,host, "| Total Requests:", ++requests,"| App Uptime:", (new Date() - startTime)/1000 , "seconds", "| Log Time:",new Date());  }  var www = http.createServer(handleRequest);  www.listen(8080,function () {  startTime = new Date();;  host = process.env.HOSTNAME;  console.log ("Kubernetes Bootcamp App Started At:",startTime, "| Running On: " ,host, "\n" );  }); |

**Dockerfile:**

|  |
| --- |
| FROM node:6.14.2  EXPOSE 8080  COPY server.js .  CMD node server.js |

1. Use the kubectl get command to view your new deployment as follows:

|  |
| --- |
| kubectl get deployments |

As this is running inside your Cluster (effectively in a second virtual machine) your application is running on a private isolated network. While applications in the cluster can communicate directly with each other, your host VM does not currently have direct access to the cluster.

The kubectl tool interacts with the cluster through an API and you can use this to configure a proxy that will forward commands to the private network your Deployment is running within. This means that you can have direct access to the API from the command line without the kubectl tool.

To have this proxy running you need to open a new terminal window and connect to your VM as normal. You need to have this terminal running the proxy separately any time you want to interact with your cluster in this way. Follow the steps below to configure a proxy now:

1. Keep your original terminal open
2. Open a new terminal and ssh into your VM
3. Run the ‘kubectl proxy’ command

You should see something like ‘Starting to serve on 127.0.0.1:8001’

1. Leave this terminal running
2. Go back to your main terminal

Now you can communicate with your cluster API using the command line.

1. To access the version information for example you could use the following command:

|  |
| --- |
| curl http://localhost:8001/version |

The curl command transfers data from a server to your terminal. Here you are accessing the Cluster API version information and displaying it in the terminal.

Requiring the proxy is a bit inconvenient but you will look at another solution to this later in the lab so that you can actually see your application running.

# Kubernetes Pods and Nodes

When you deployed the application earlier in this Lab, Kubernetes didn’t just run the container on a Node; a pod was created which the container runs within. Each pod can contain one or more containers along with some shared resources such as IP addresses, storage volumes and information on running the containers within that pod.

If a pod contains more than one container, it is typical for these to be very closely related. An example would be one container running a Node.js server and another container which feeds data to that server.

When you deploy a container, the Deployment creates a pod and runs the container inside the pod. This pod is tied to the Node it is launched on and if that Node goes down then the Deployment launches an identical pod (or pods if multiple were running on the node) on another appropriate Node.

This figure should give you an idea of how pods are structured:

A close up of a logo

Description automatically generated

Figure : K8s Pods

1. Have a look at your currently running Pods using the ‘kubectl get pods’ command.

You have one pod running at the moment. The name is similar to the name of the Deployment but has a unique pod ID as well.

1. Export this pod name as an environment variable so that you can use it in future commands as follows copying your pod name after the = sign. The echo command should print your pod name if the export command has been successful.

|  |
| --- |
| export POD\_NAME=  echo $POD\_NAME |

1. You have used the ‘kubectl get’ command a few times now to get an overview of an element of the Cluster. Now look at the ‘kubectl describe’ command which will give you more information.

|  |
| --- |
| kubectl describe pods |

Here you can see much more information about the pod. Take some time to have a look at what this command has told us about the pod.

You can use kubectl to directly access a container within k8s. You should notice that these commands are extremely similar to the commands we used to directly access our Docker containers in previous labs.

1. First run a one-time command:

|  |
| --- |
| kubectl exec $POD\_NAME -- env |

This has executed the ‘env’ command inside the container you have running in the pod. You can confirm this by running the env command locally and comparing the differences.

1. You can also access a container interactively as you would in Docker as follows:

|  |
| --- |
| kubectl exec -it $POD\_NAME -- bash |

1. If you run the ‘ls’ command here, you will see the server.js file that was introduced briefly earlier in this Lab. You can view this using the ‘cat’ command which you have used before.
2. You can also now see the app running because you are inside the container. Check that the app is behaving correctly using the following:

|  |
| --- |
| curl localhost:8080 |

The output should be similar to:

**Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-69fbc6f4cf-ltmbj | v=1**

This is obviously of limited use as you do not want to have to run the app from within the container every time but it is useful for troubleshooting. You will look at a solution to this in the next section.

Exit the container by typing exit and pressing enter. You can also close the terminal window which is currently running the proxy.

# Kubernetes Services

Previously when pods were introduced it was stated that each pod gets a unique IP address that is only accessible by the Cluster. To expose this IP address to anything outside the Cluster we need to use a Service.

A Service can be used to group pods together and provide a single point of access to them. For example, imagine that you have three pods in a cluster all of which are hosting a website. Each of these pods will have a unique IP address but if we group them using a Service then all three will be accessible from the same external IP address.

This means that the user will access any of the pods using one IP so that if a pod goes down or is overloaded, the traffic is automatically directed to one of the other pods and the user has no idea that anything went wrong in the background.

Have a look at the diagram below which shows two Deployments. Deployment A has only one container running within its own Service while Deployment B has three containers running across two Nodes but still grouped by a single Service.

A close up of a logo

Description automatically generated

Figure : K8s Services

Look Deployment B in Figure 4 above. What this means is that any Pod or Node in this Deployment could go offline without having a massive impact on the functionality of the application. Worst case scenario is that the application slows down due to increased load on the remaining pods but this will be temporary as the Deployment will notice that a Pod has gone down and will restart the Pod or, in the case of a Node going down, the Deployment will relaunch the affected Pods on another suitable Node.

1. Use the ‘kubectl get service’ command to see if there are any services running at present.

You should see that you do have one service which was created when the k8s cluster was created.

1. Use the following command to create a new service for your application and have a look at what you have done:

|  |
| --- |
| kubectl expose deployment/kubernetes-bootcamp --type="NodePort" --port 8080 --name node-port-service  kubectl get services |

You can see here that a new service called node-port-service has been created with a unique IP address and it is connecting port 8080 on the pod to a port on the Service (which is probably in the 30,000s somewhere).

1. Create an environment variable called NODE\_PORT and set it to the value of the port specified by the new service you have created.
2. Using the service, you can now directly access the application from outwith the cluster as follows:

|  |
| --- |
| curl $(minikube ip):$NODE\_PORT |

The $(minikube ip) variable returns the IP address of the Node on which the pod is running. The Service has provided a link between this Node IP and the private IP of the pod so you can now access it directly without being inside the pod or using a proxy.

Note that you can currently only access this from the machine you are running the cluster on (your AWS EC2 VM) and you cannot access it from an external location. This is a limitation of the Minikube deployment which is primarily designed for development. In a full Kubernetes deployment, you could create another service which would allow you to access the application from anywhere.

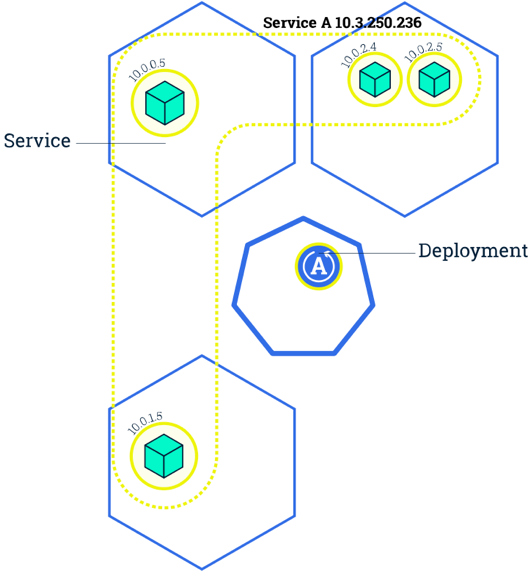
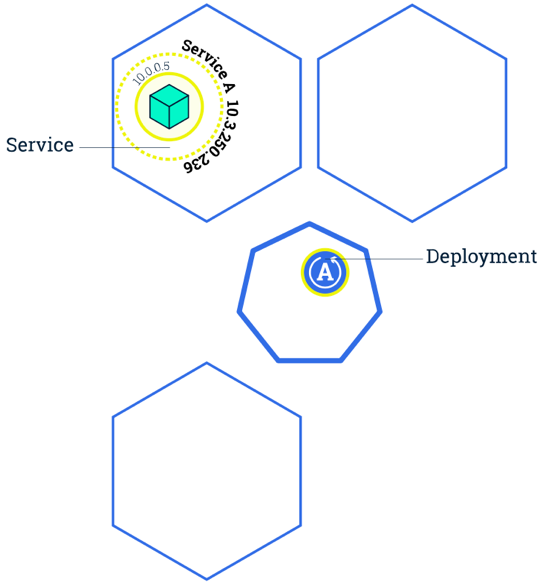
# Application Scaling, Load Balancing and Rolling Updates

So far, Kubernetes probably just looks like a complicated version of Docker, which is not an unfair assessment at this point in the Lab. In this section you will look at a few of the features that make Kubernetes so powerful.

Currently you have a single pod running your application. If a large number of users attempted to use the application at the same time you may encounter load issues. Fortunately, Kubernetes allows you to easily scale your Deployment.

Scaling a Deployment will automatically create new Pods on any available Nodes but within the same Service so that they are accessible in the same way. This is illustrated in the image below:

Figure 5: K8s Scaling



SCALE

1. Have another look at the current Deployments using the ‘kubectl get deployments’ command.

The READY section gives us the ratio of currently running pods to the desired number of pods. You have 1/1 at the moment as you have only told Kubernetes to launch one pod.

You can scale this to whatever size you like but for this example aim for 4 pods. Note that you can scale the application back down using the exact same command but specifying a lower number of replicas.

1. Run the following command to scale the application:

|  |
| --- |
| kubectl scale deployments/kubernetes-bootcamp --replicas=4 |

1. Check your pods and deployments using the appropriate ‘kubectl get’ commands to see what changes have occurred.

Hopefully you could see that there are now 4 pods in your Deployment and each pod has a unique ID. You know however, that they are all part of the same service you created for the Deployment previously.

1. If you run your application again, using the curl command, you will see that you did not need to change the IP address or the port in order to run the app.

So how does Kubernetes know which pod to use when you try to run the application?

Kubernetes provides automatic load balancing for this exact purpose. This means that when you access the pods in a Deployment through a Service, Kubernetes automatically assigns a pod to you. This means that if may users are attempting to access the application at the same time, Kubernetes will automatically distribute these users across the available pods as evenly as possible.

You can see this in action by simulating multiple users.

1. Create a file called multiple\_users.sh using nano and add the following text:

|  |
| --- |
| #! /bin/bash  for i in {1..20}  do  curl $(minikube ip):$NODE\_PORT  done |

This file will run the ‘curl $(minikube ip):$NODE\_PORT’ command 20 times for you in quick succession.

1. The following commands will make the new file executable and then run the script:

|  |
| --- |
| chmod +x multiple\_users.sh  ./multiple\_users.sh |

1. Have a look at the pod ID for each run and you should see that Kubernetes is automatically running your command on a free pod without you having to do anything differently when accessing the application.

Now that you have your application running on several pods you can take advantage of the Rolling Update feature that Kubernetes provides.

Kubernetes allows for continuous delivery of applications by allowing you to update your applications without causing any downtime for users.

This is achieved by incrementally stopping and updating pods one at a time while automatically load-balancing the traffic across the other available pods which are not being updated. Have a look at the image below which gives an overview of the process.

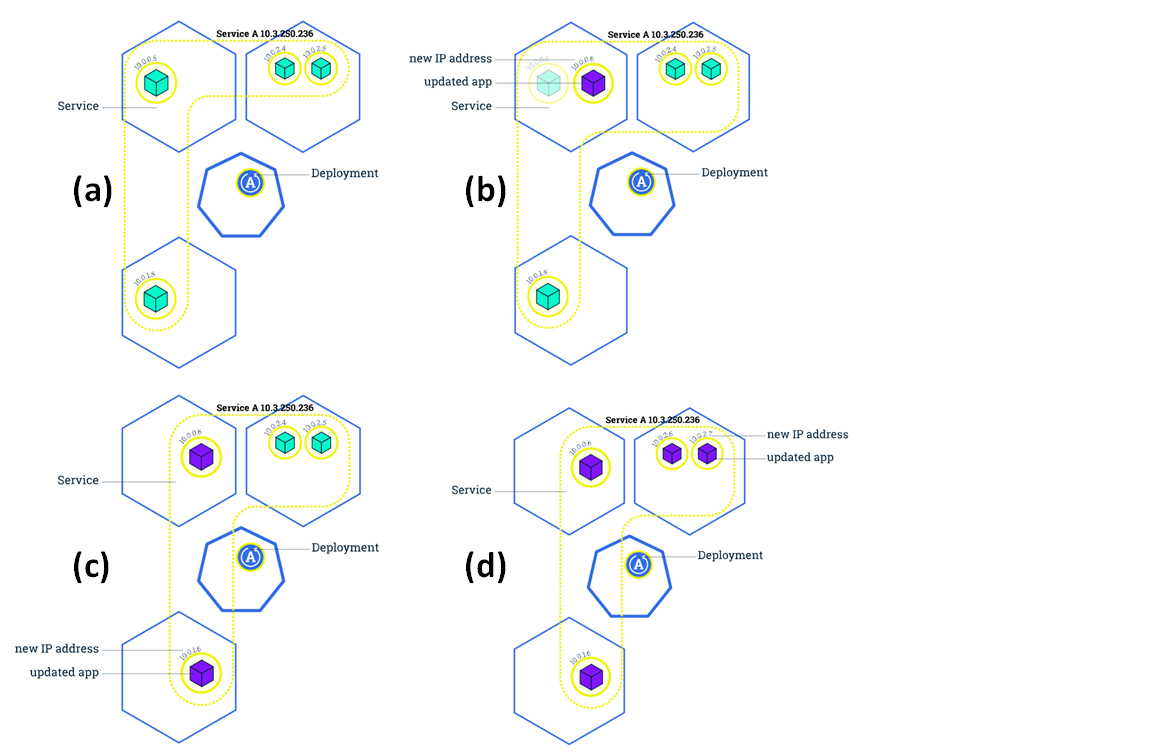


Figure : K8s Rolling Updates.

(a) V1 running on all pods, (b) One pod offline and updated,

(c) next pod updated, (d) remaining pods updated

1. Roll out a new image to your Deployment using the command below. Here you are replacing your current image, v1, with v2 and the Kubernetes Deployment automatically takes care of rolling this out across your four pods. The component ‘&& ./multiple\_users.sh’ runs the multiple\_user.sh script immediately after the update command, you should be able to see that Kubernetes is automatically running on pods with v1 and will gradually start running on pods with v2 as they are updated. The important point to note is that the application is active for the duration of this update.

|  |
| --- |
| kubectl set image deployments/kubernetes-bootcamp kubernetes-bootcamp=jocatalin/kubernetes-bootcamp:v2 && ./multiple\_users.sh |

Imagine you rolled out this new version and received a flurry of complaints from your customers that the new version wasn’t working as intended. Kubernetes allows you to roll back your changes and effectively undo the update. This means that you can quickly revert to a previously stable state quickly and hopefully minimise any disruption.

1. Roll back to v1 of your application using the below command. If you run the ‘kubectl get pods’ command a few times immediately after rolling back the changes you will see Kubernetes creating and terminating pods as v1 of your app is rolled back out.

|  |
| --- |
| kubectl rollout undo deployments/kubernetes-bootcamp |

1. Run your application again to ensure that the Deployment is once again running v1 of the app.
2. The following commands can be used to remove your deployment and your minikube vm. Note that these are irreversible:

|  |
| --- |
| kubectl delete deploy/kubernetes-bootcamp  minikube delete |

Congratulations!! You have completed the sixth lab in this module. Clearly Kubernetes is an incredibly powerful tool which has significant value in a DevOps environment, especially if you have multiple servers to manage application rollout across.

**PLEASE REMEMBER TO STOP OR TERMINATE YOUR VM AT THE END OF THIS LAB SO THAT YOU DO NOT USE UNNECCESARY CREDIT!!**