**Aim:**To learn and implement the basics of Kubernetes using Minikube on a VirtualBox-based Ubuntu environment, including deploying, accessing, exploring, and exposing a containerized application.

**Objectives:**

* Install and configure Minikube and kubectl on Ubuntu
* Start and interact with a Kubernetes cluster
* Deploy a simple app on the cluster
* Explore application internals using logs and shell access
* Expose the application externally
* Understand and utilize Kubernetes services and labels

**Tools Used:**

VirtualBox

Ubuntu (Linux OS)

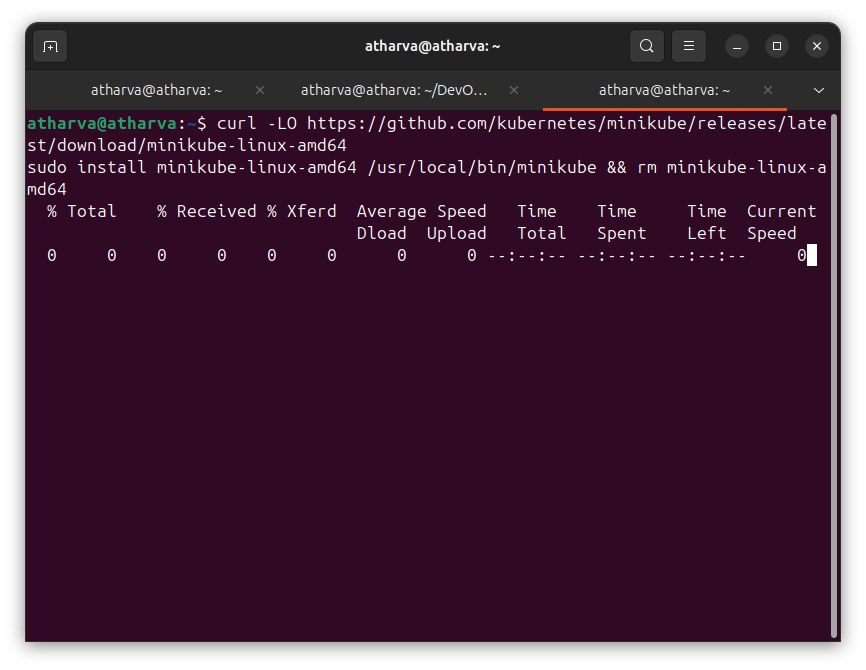
**Process:  
  
Reference Links:**<https://minikube.sigs.k8s.io/docs/start/?arch=%2Fwindows%2Fx86-64%2Fstable%2F.exe+download>

<https://minikube.sigs.k8s.io/docs/tutorials/kubernetes_101/>

## **Installation of minikube**

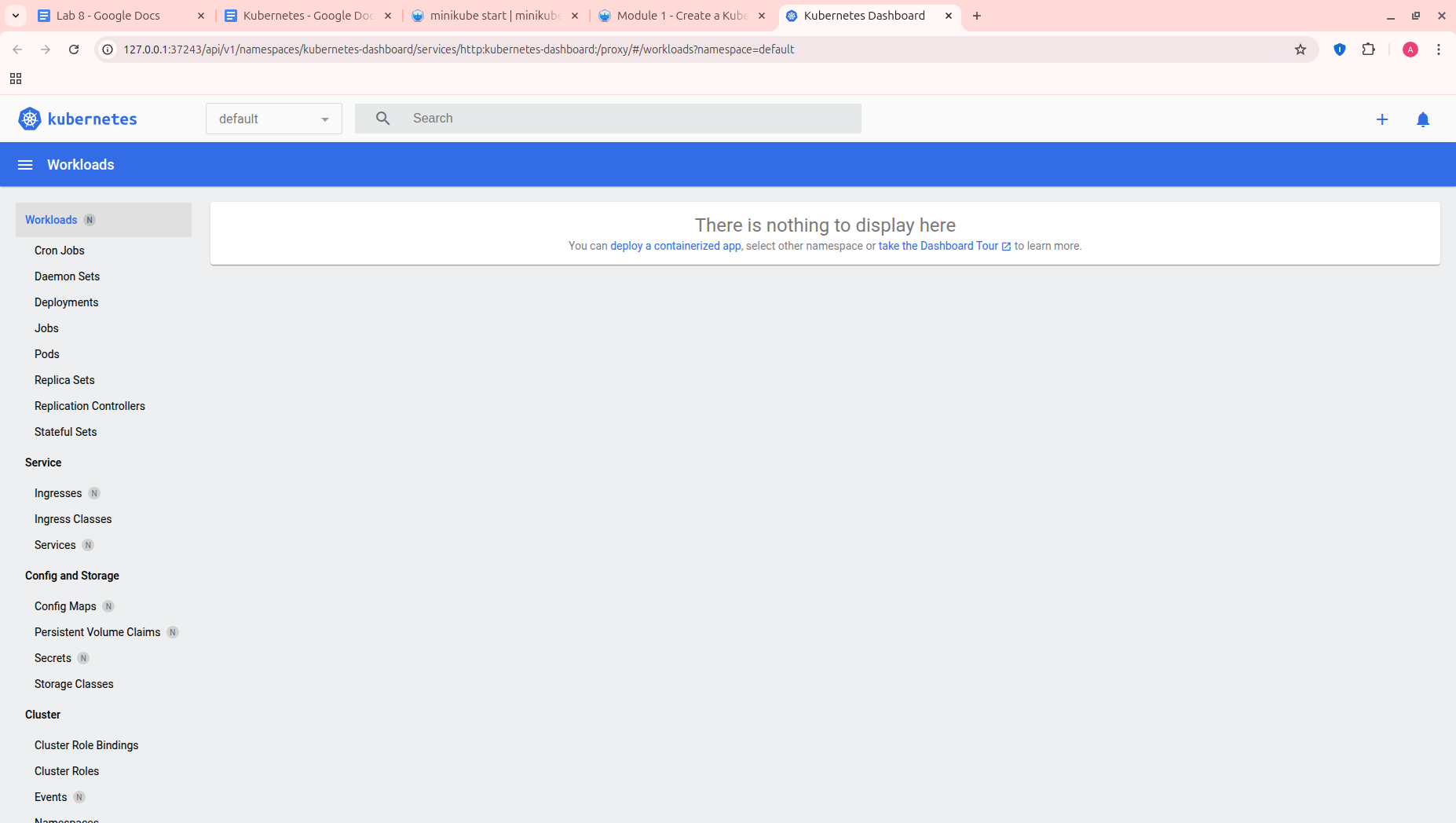
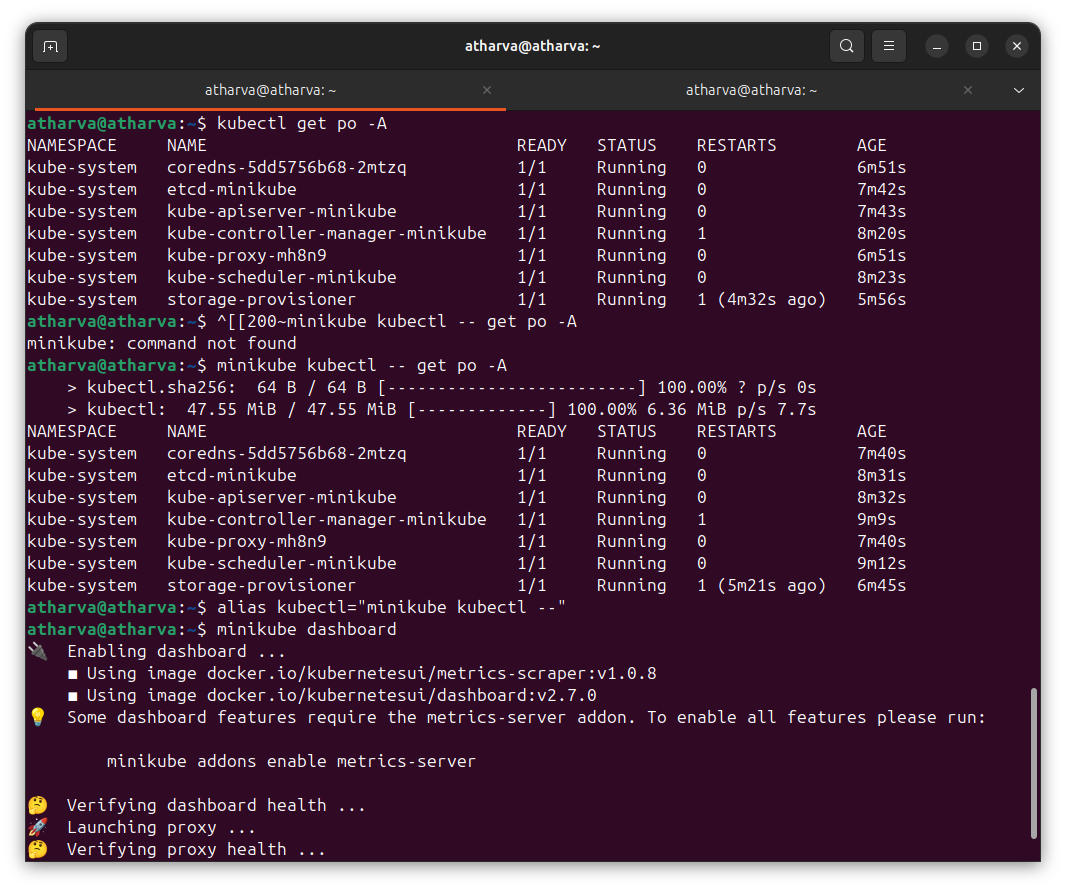
curl -LO https://github.com/kubernetes/minikube/releases/latest/download/minikube-linux-amd64

sudo install minikube-linux-amd64 /usr/local/bin/minikube && rm minikube-linux-amd64



## Start your cluster

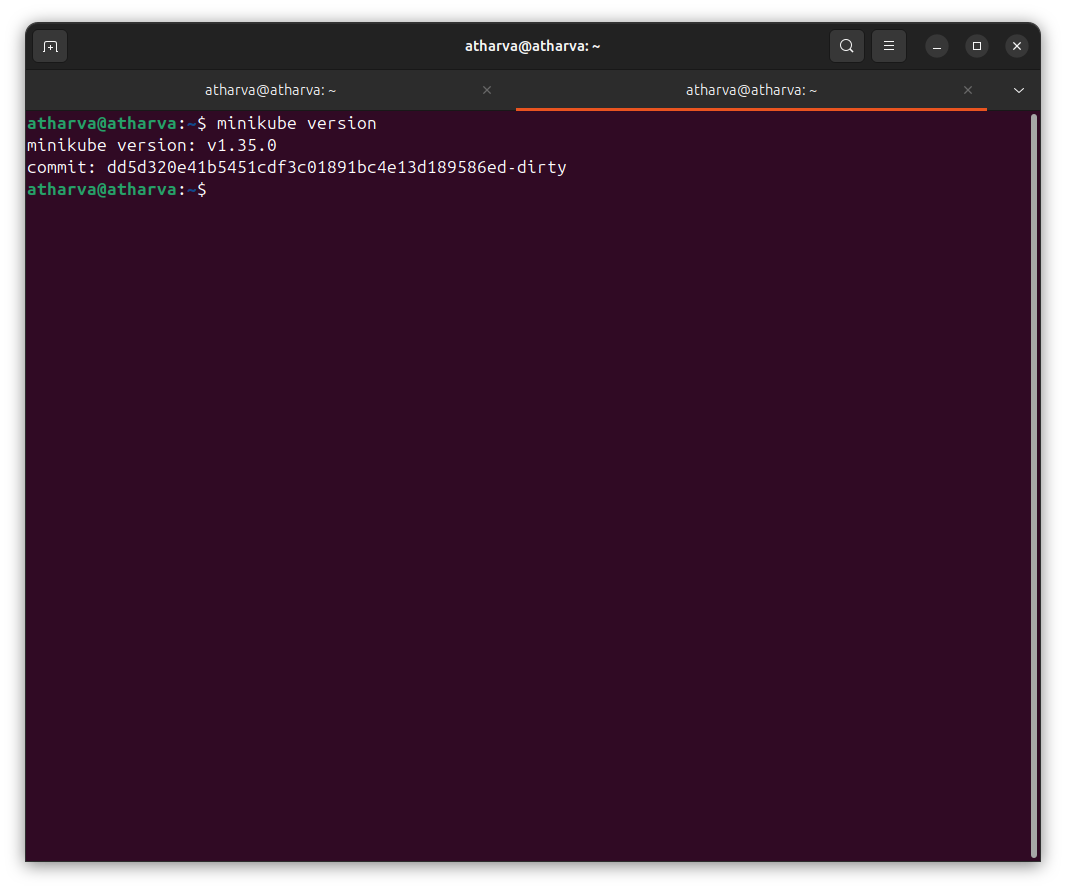
## Interact with your cluster



# **Module 1 - Create a Kubernetes Cluster**

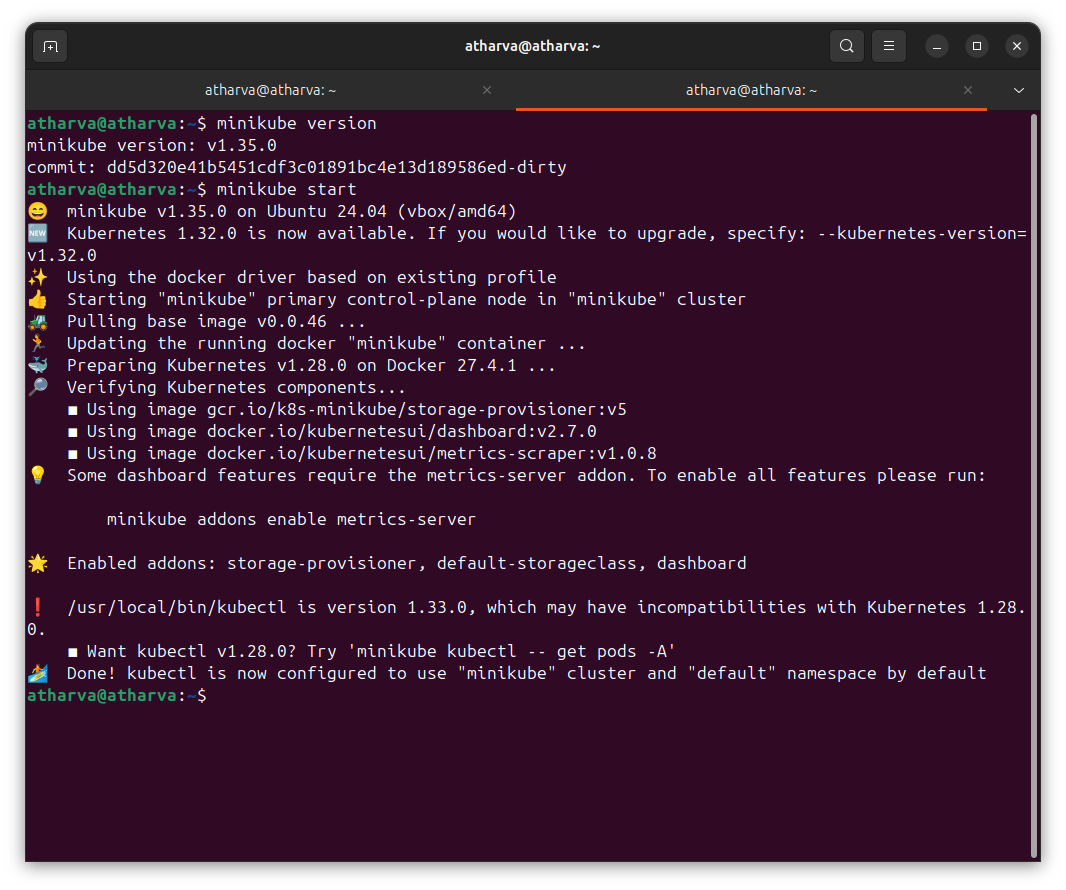
## Step 1 - Cluster up and running

minikube version



Once minikube is installed, start the cluster, by running the *minikube start* command:

minikube start

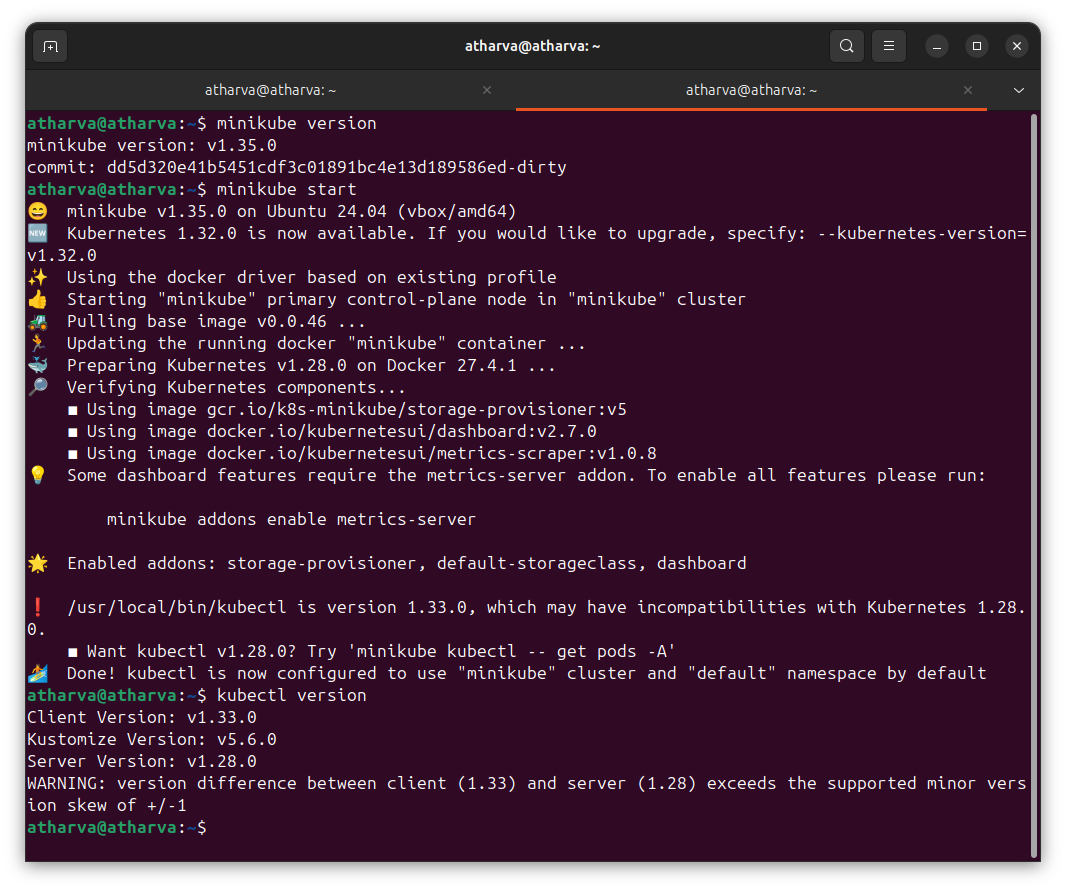


## 

## Step 2 - Cluster version

To interact with Kubernetes during this bootcamp we’ll use the command line interface, kubectl. We’ll explain kubectl in detail in the next modules, but for now, we’re just going to look at some cluster information. To check if kubectl is installed you can run the *kubectl version* command:

kubectl version

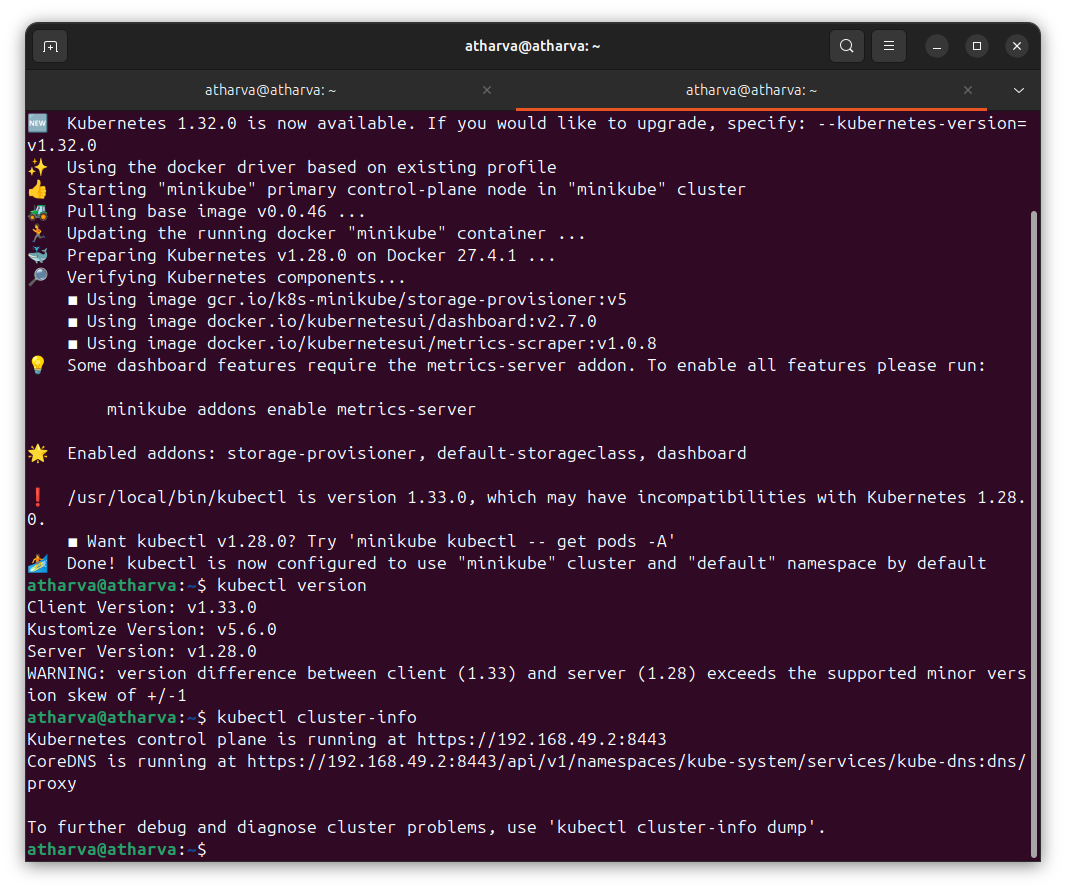


OK, kubectl is configured and we can see both the version of the client and as well as the server. The client version is the kubectl version; the server version is the Kubernetes version installed on the master. You can also see details about the build.

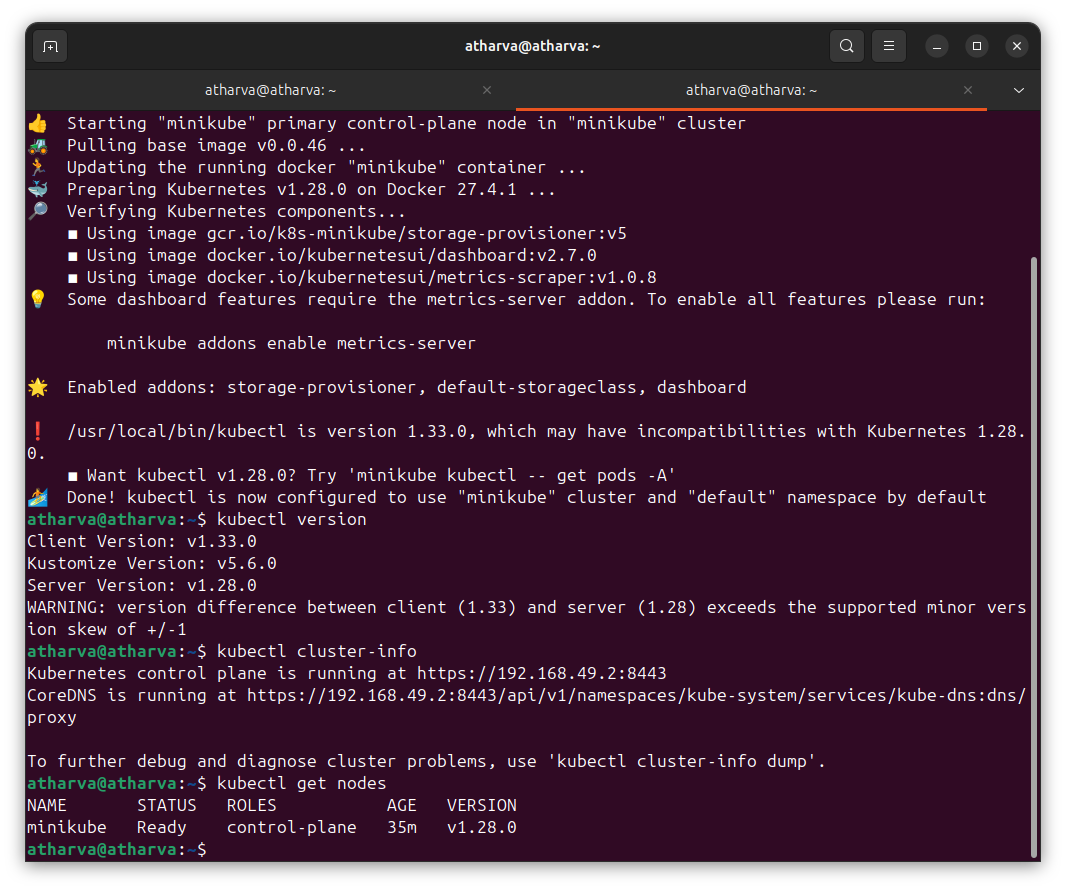
## Step 3 - Cluster details

Let’s view the cluster details. We’ll do that by running *kubectl cluster-info*:

kubectl cluster-info

  
During this tutorial, we’ll be focusing on the command line for deploying and exploring our application. To view the nodes in the cluster, run the *kubectl get nodes* command:

kubectl get nodes



# **Module 2 - Deploy an app**

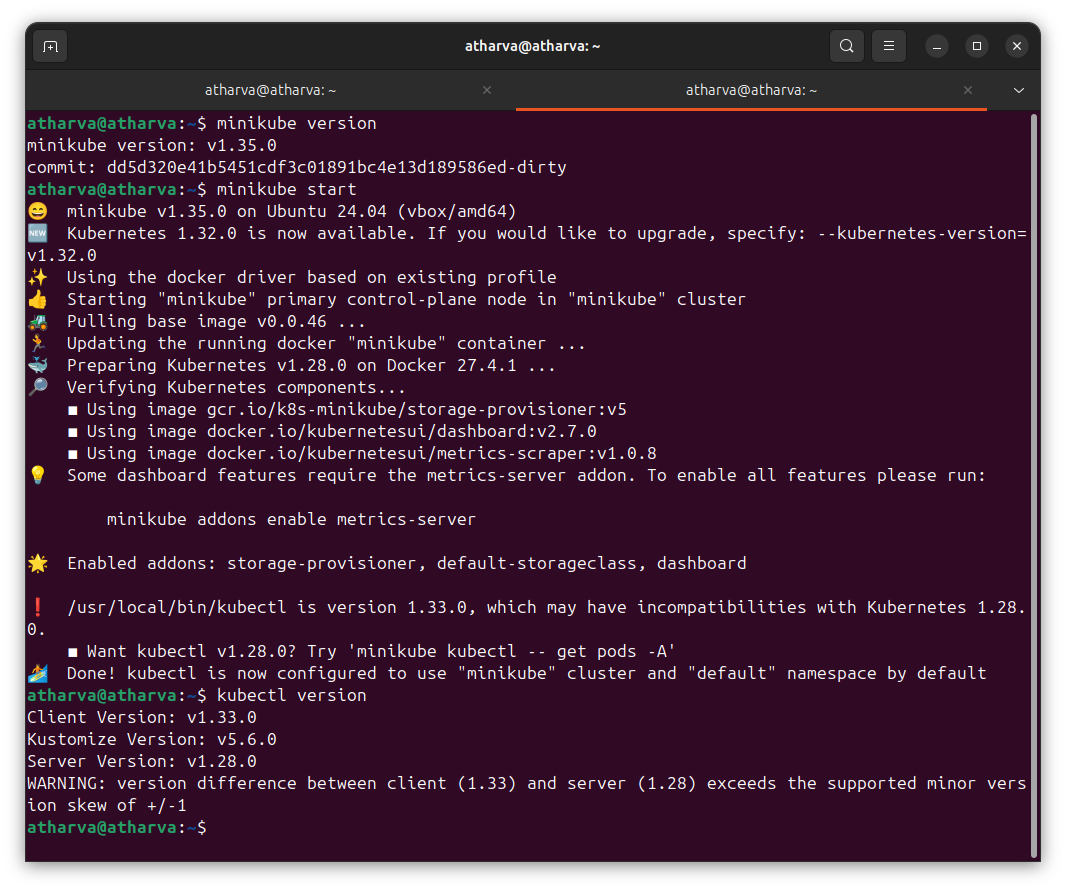
The goal of this scenario is to help you deploy your first app on Kubernetes using kubectl. You will learn the basics about kubectl cli and how to interact with your application.

## Step 1 - kubectl basics

Type kubectl in the terminal to see its usage. The common format of a kubectl command is: kubectl action resource. This performs the specified action (like create, describe) on the specified resource (like node, container). You can use --help after the command to get additional info about possible parameters (kubectl get nodes --help).

Check the kubectl is configured to talk to your cluster, by running the kubectl version command:

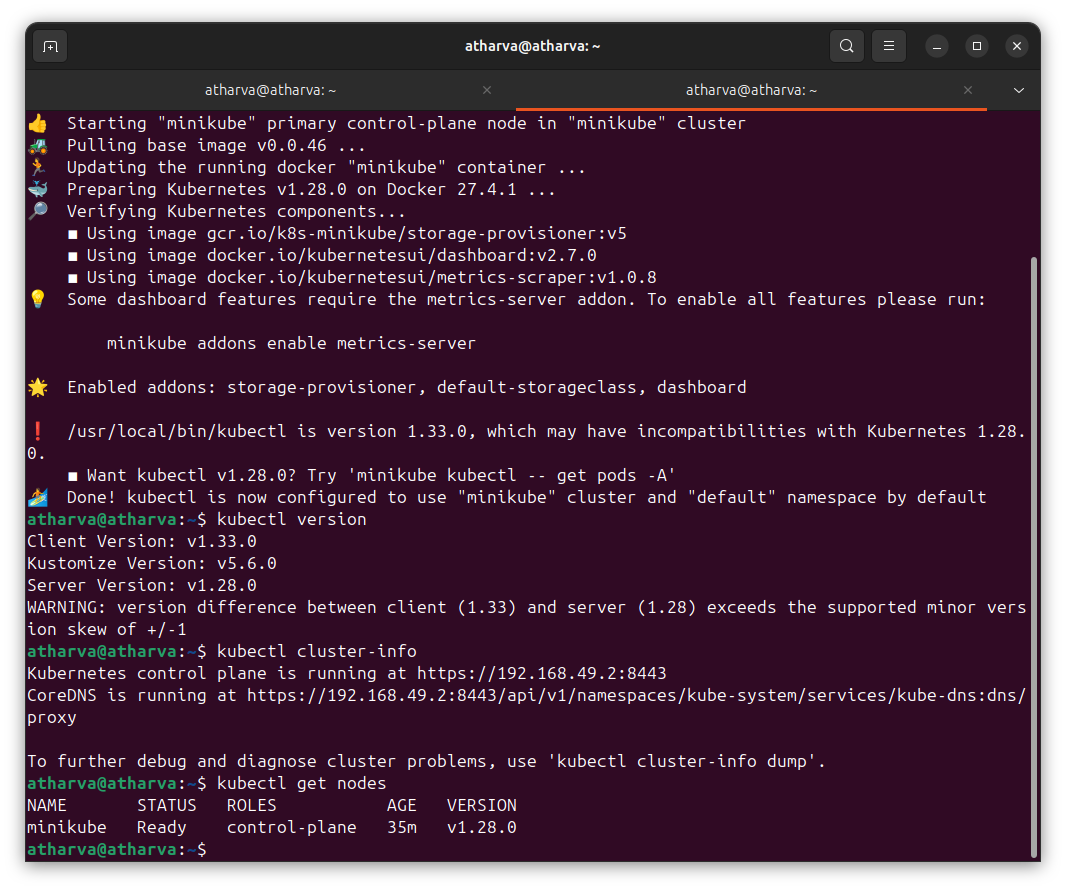
kubectl version



OK, kubectl is installed and you can see both the client and the server versions.

To view the nodes in the cluster, run the kubectl get nodes command:

kubectl get nodes

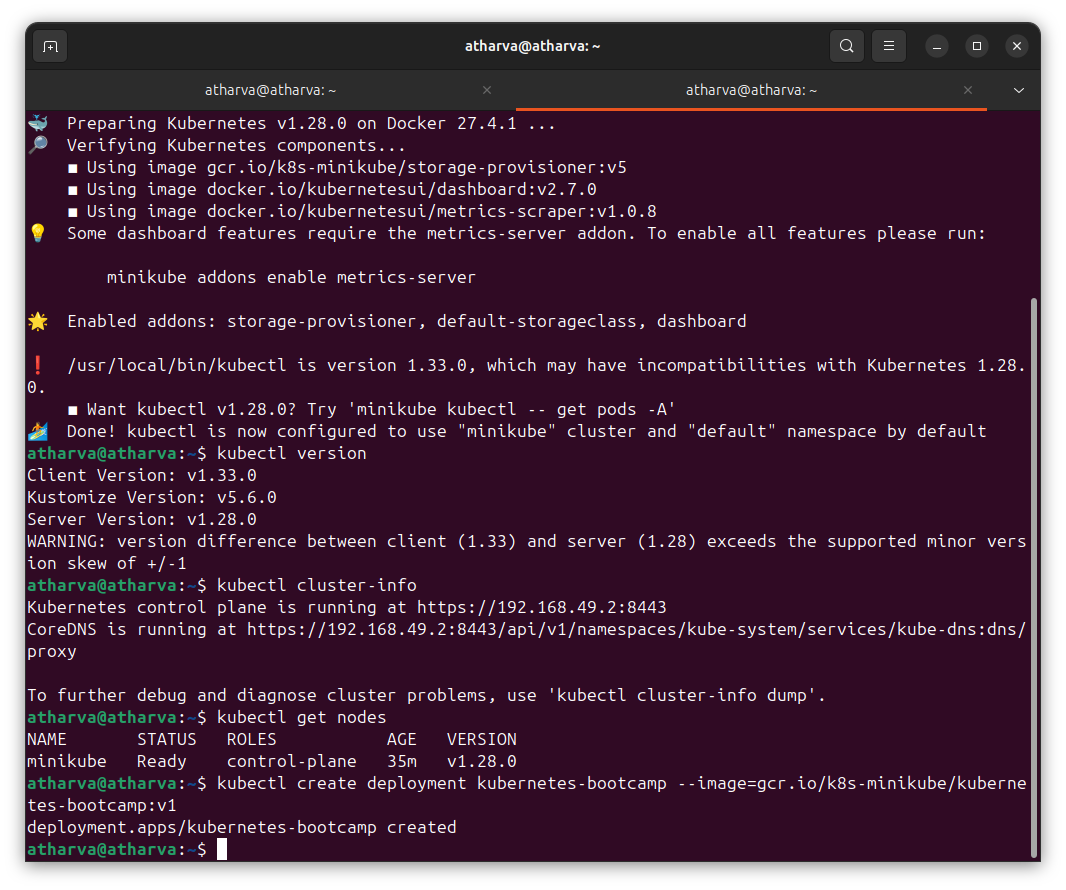


Here we see the available nodes (1 in our case). Kubernetes will choose where to deploy our application based on Node available resources.

## Step 2 - Deploy our app

Let’s deploy our first app on Kubernetes with the kubectl create deployment command. We need to provide the deployment name and app image location (include the full repository url for images hosted outside Docker Hub).

kubectl create deployment kubernetes-bootcamp --image=[gcr.io/k8s-minikube/kubernetes-bootcamp:v1](http://gcr.io/k8s-minikube/kubernetes-bootcamp:v1)

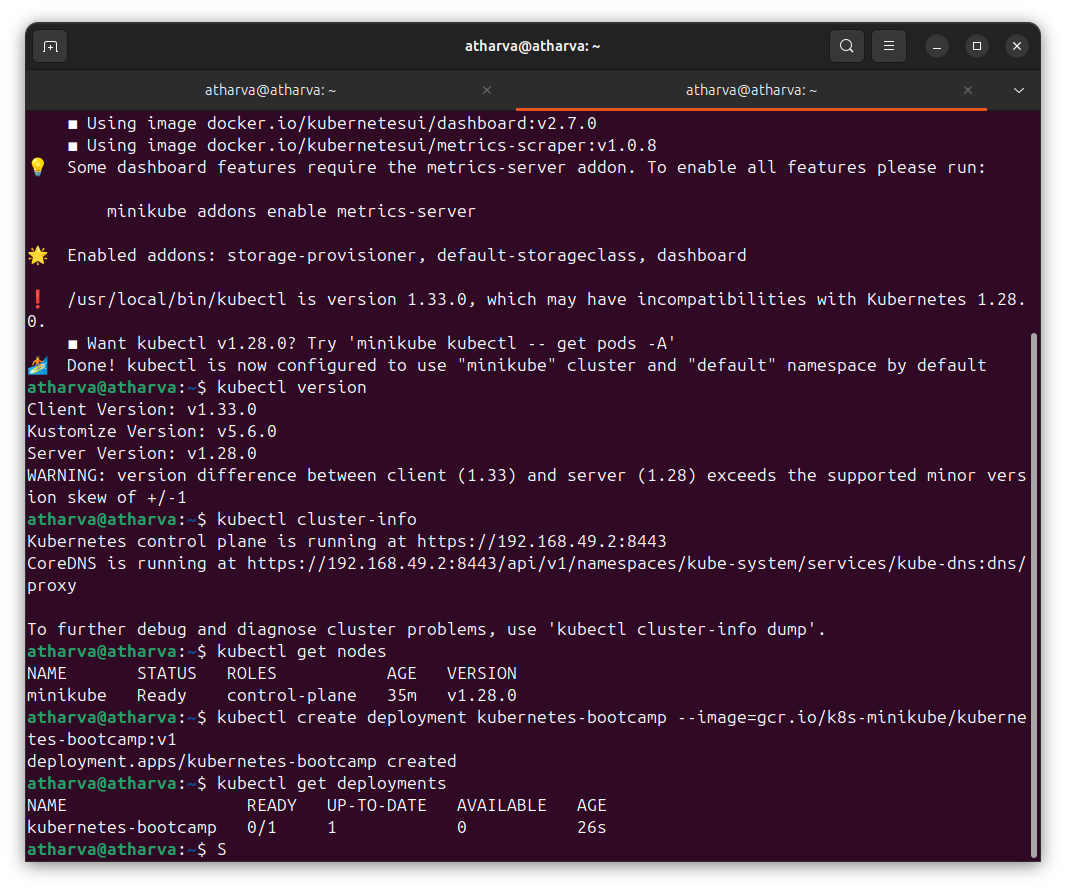


Great! You just deployed your first application by creating a deployment. This performed a few things for you:

* searched for a suitable node where an instance of the application could be run (we have only 1 available node)
* scheduled the application to run on that Node
* configured the cluster to reschedule the instance on a new Node when needed

To list your deployments use the get deployments command:

kubectl get deployments



We see that there is 1 deployment running a single instance of your app. The instance is running inside a Docker container on your node.

## Step 3 - View our app

Pods that are running inside Kubernetes are running on a private, isolated network. By default they are visible from other pods and services within the same Kubernetes cluster, but not outside that network. When we use kubectl, we’re interacting through an API endpoint to communicate with our application.

We will cover other options on how to expose your application outside the Kubernetes cluster in Module 4.

The kubectl command can create a proxy that will forward communications into the cluster-wide, private network. The proxy can be terminiated by pressing control-C and won’t show any output while its running.

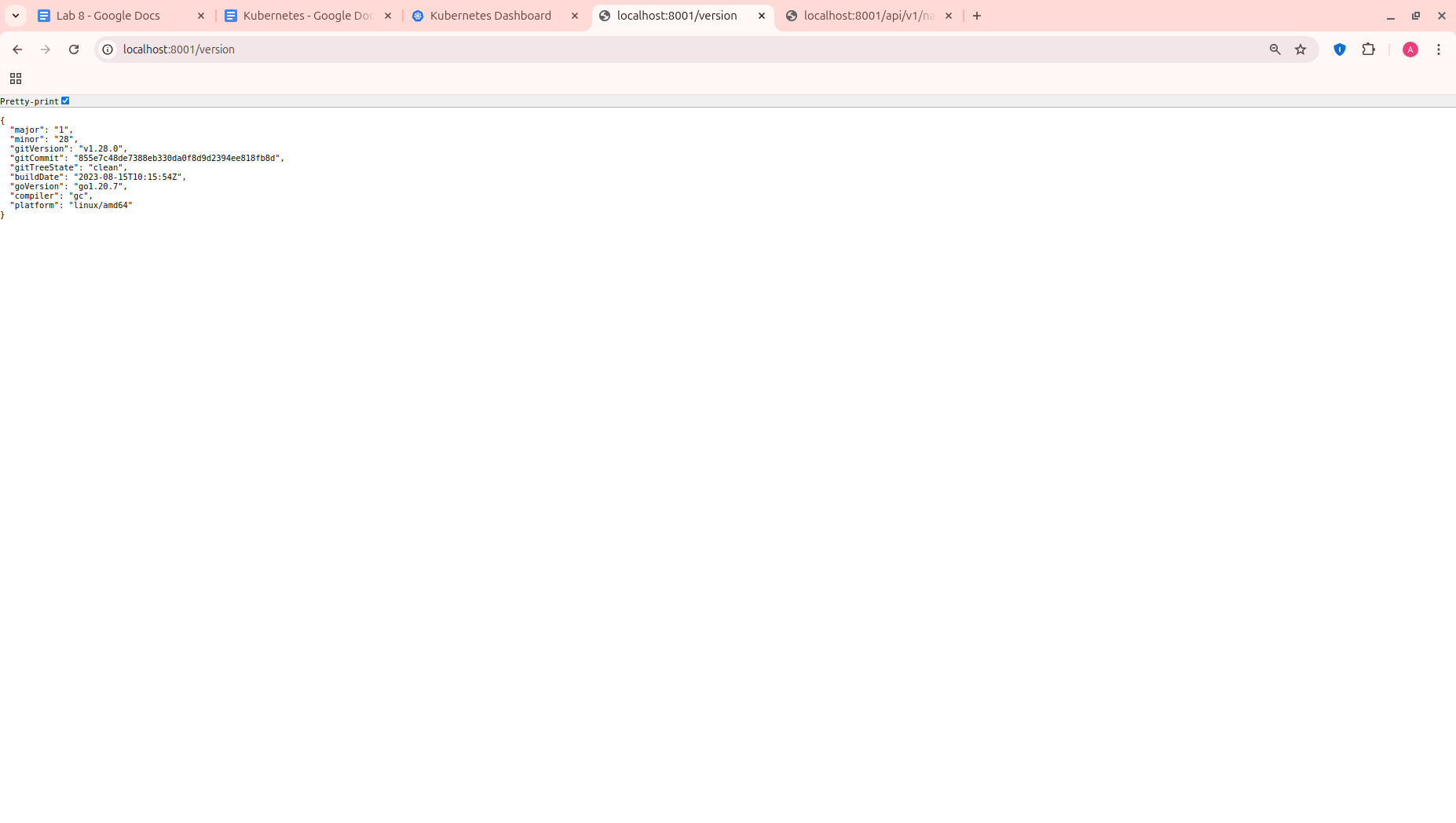
We will open a second terminal window to run the proxy.

echo -e "Starting Proxy. After starting it will not output a response. Please return to your original terminal window\n"; kubectl proxy

We now have a connection between our host (the online terminal) and the Kubernetes cluster. The proxy enabled direct access to the API from these terminals.

You can see all those APIs hosted through the proxy endpoint. For example, we can query the version directly through the API using the curl command:

curl <http://localhost:8001/version>



*Note: The proxy was run in a new tab, and the recent commands were executed in the original tab. The proxy still runs in the second tab, and this allowed our curl command to work using localhost:8001.*

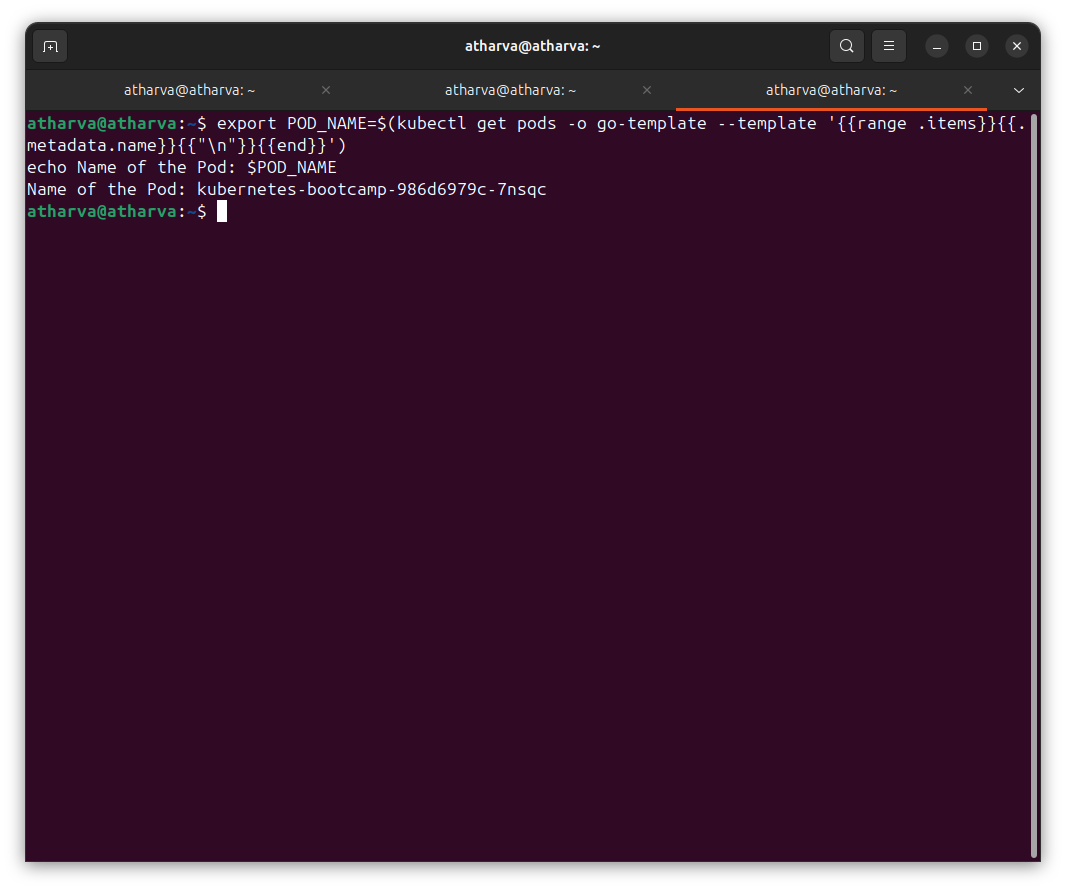
**If Port 8001 is not accessible, ensure that the kubectl proxy started above is running.**

The API server will automatically create an endpoint for each pod, based on the pod name, that is also accessible through the proxy.

First we need to get the Pod name, and we’ll store in the environment variable POD\_NAME:

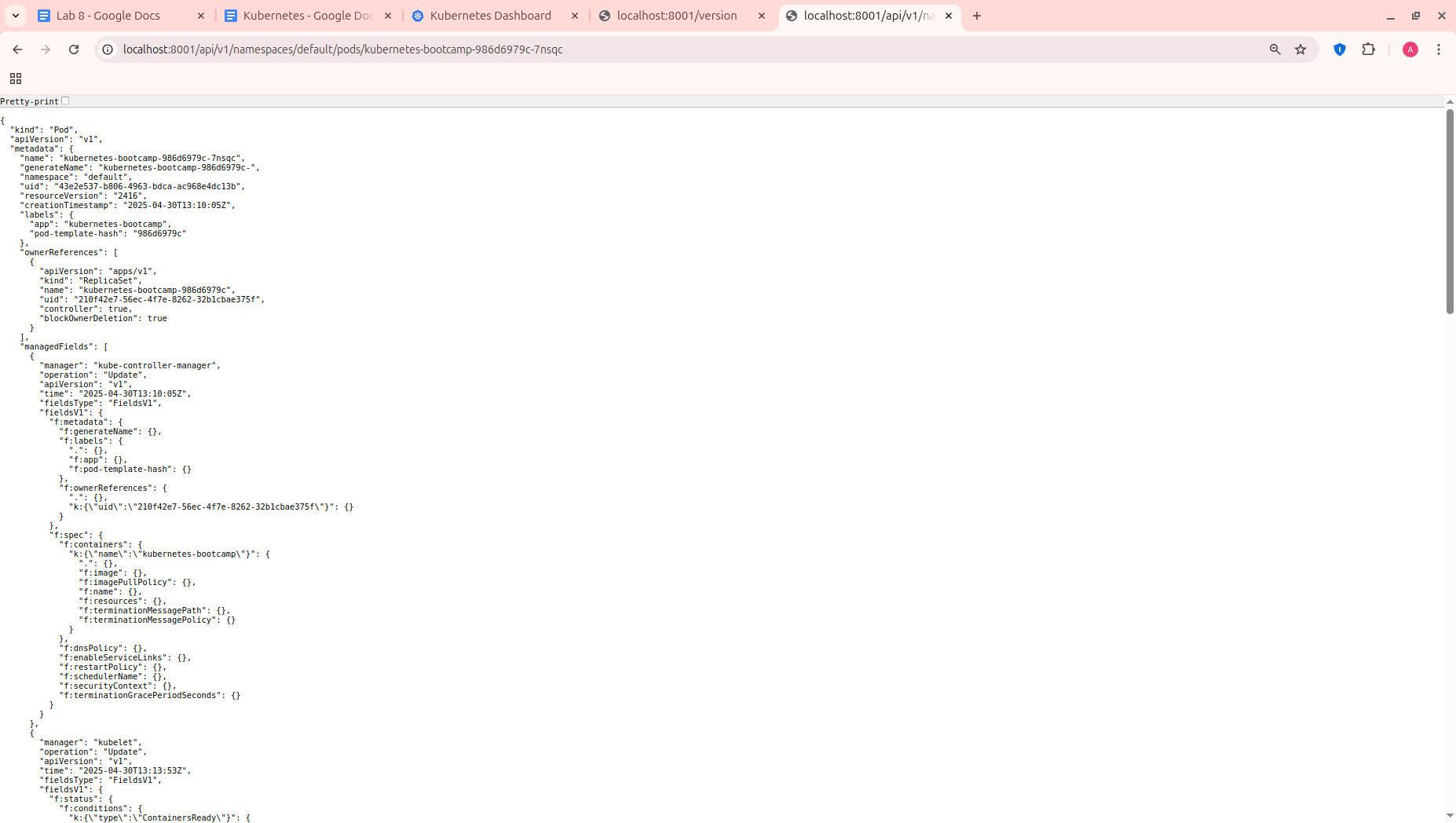
export POD\_NAME=$(kubectl get pods -o go-template --template '{{range .items}}{{.metadata.name}}{{"\n"}}{{end}}')

echo Name of the Pod: $POD\_NAME



You can access the Pod through the API by running:

curl <http://localhost:8001/api/v1/namespaces/default/pods/$POD_NAME>



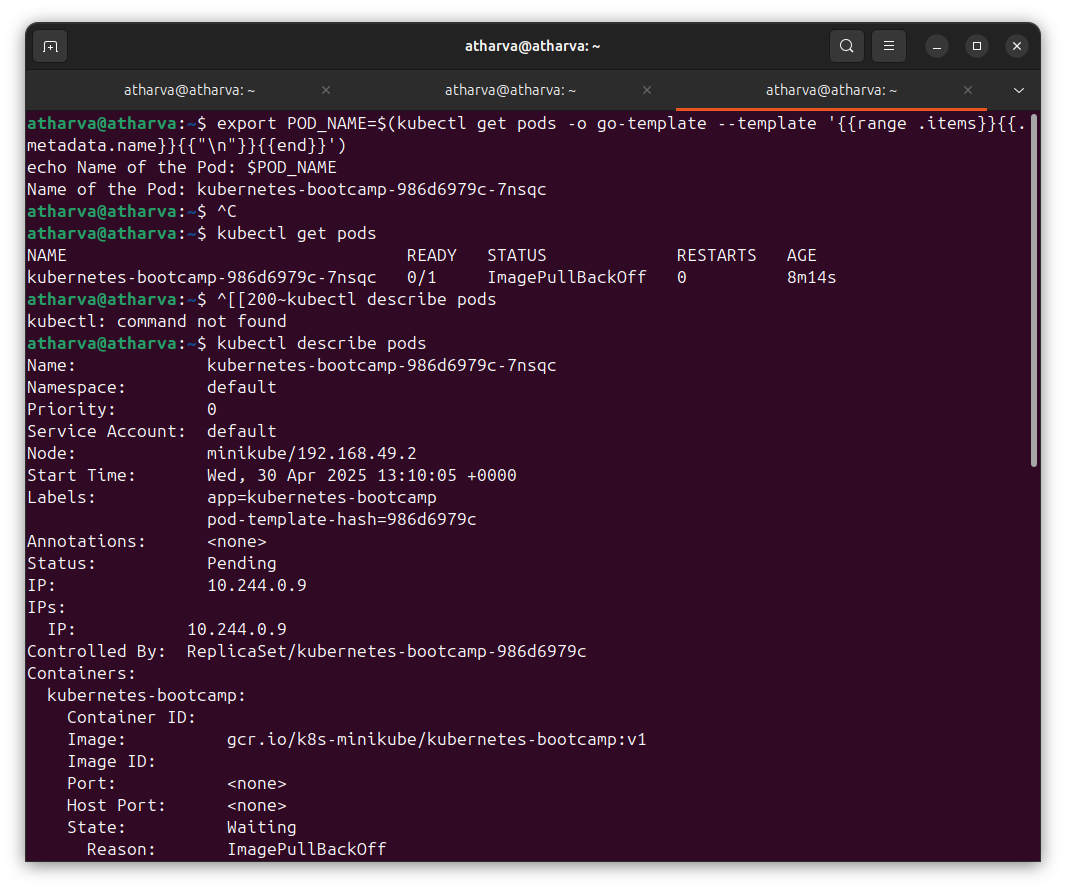
# **Module 3 - Explore your app**

In this scenario you will learn how to troubleshoot Kubernetes applications using the kubectl get, describe, logs and exec commands.

## Step 1 - Check application configuration

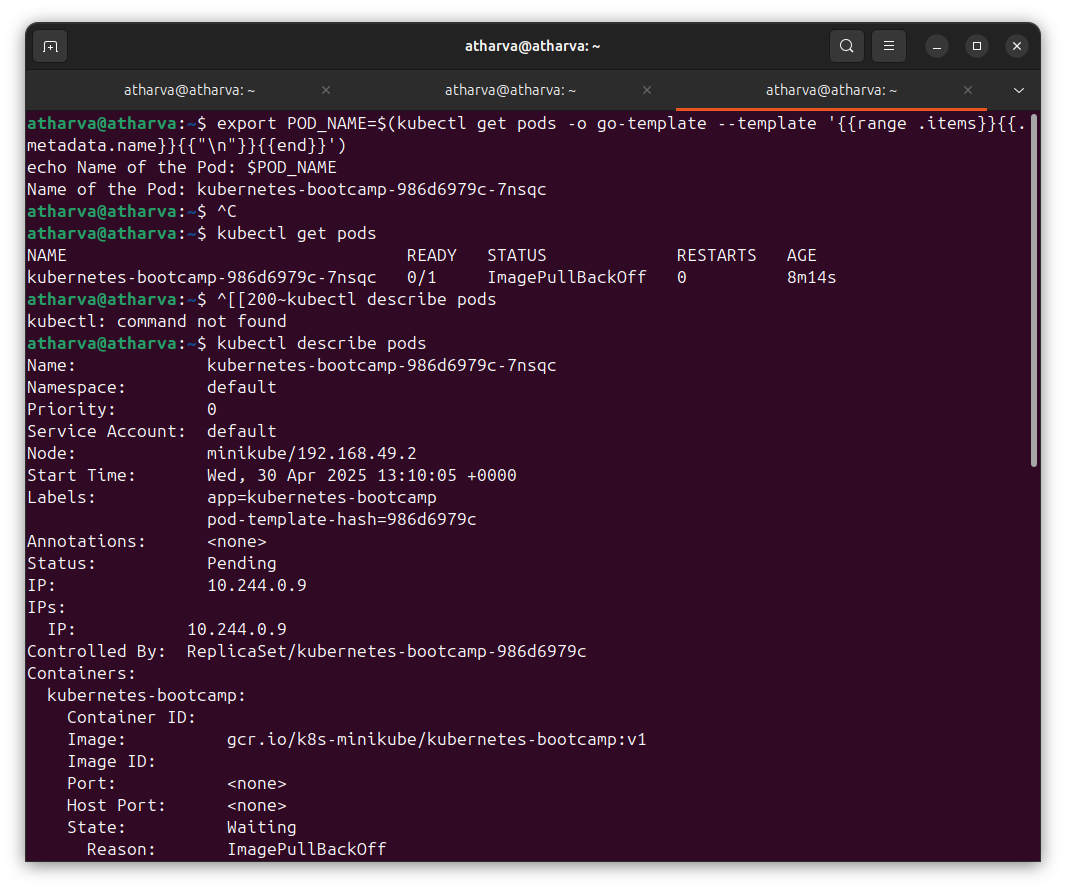
Let’s verify that the application we deployed in the previous scenario is running. We’ll use the kubectl get command and look for existing Pods:

kubectl get pods



Next, to view what containers are inside that Pod and what images are used to build those containers we run the describe pods command:

kubectl describe pods



We see here details about the Pod’s container: IP address, the ports used and a list of events related to the lifecycle of the Pod.

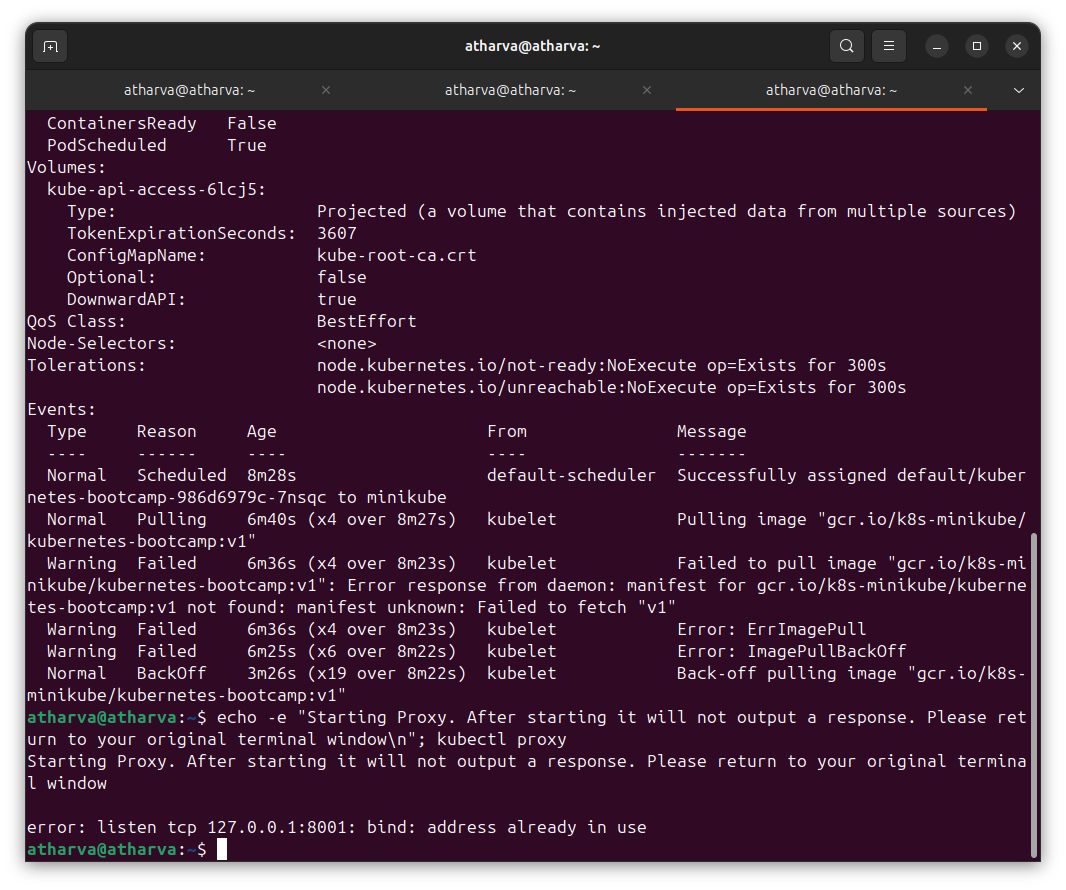
The output of the describe command is extensive and covers some concepts that we didn’t explain yet, but don’t worry, they will become familiar by the end of this bootcamp.

*Note: the describe command can be used to get detailed information about most of the Kubernetes primitives: node, pods, deployments. The describe output is designed to be human readable, not to be scripted against.*

## Step 2 - Show the app in the terminal

Recall that Pods are running in an isolated, private network - so we need to proxy access to them so we can debug and interact with them. To do this, we’ll use the kubectl proxy command to run a proxy in a second terminal window. Run the command below in a new terminal window to run the proxy:

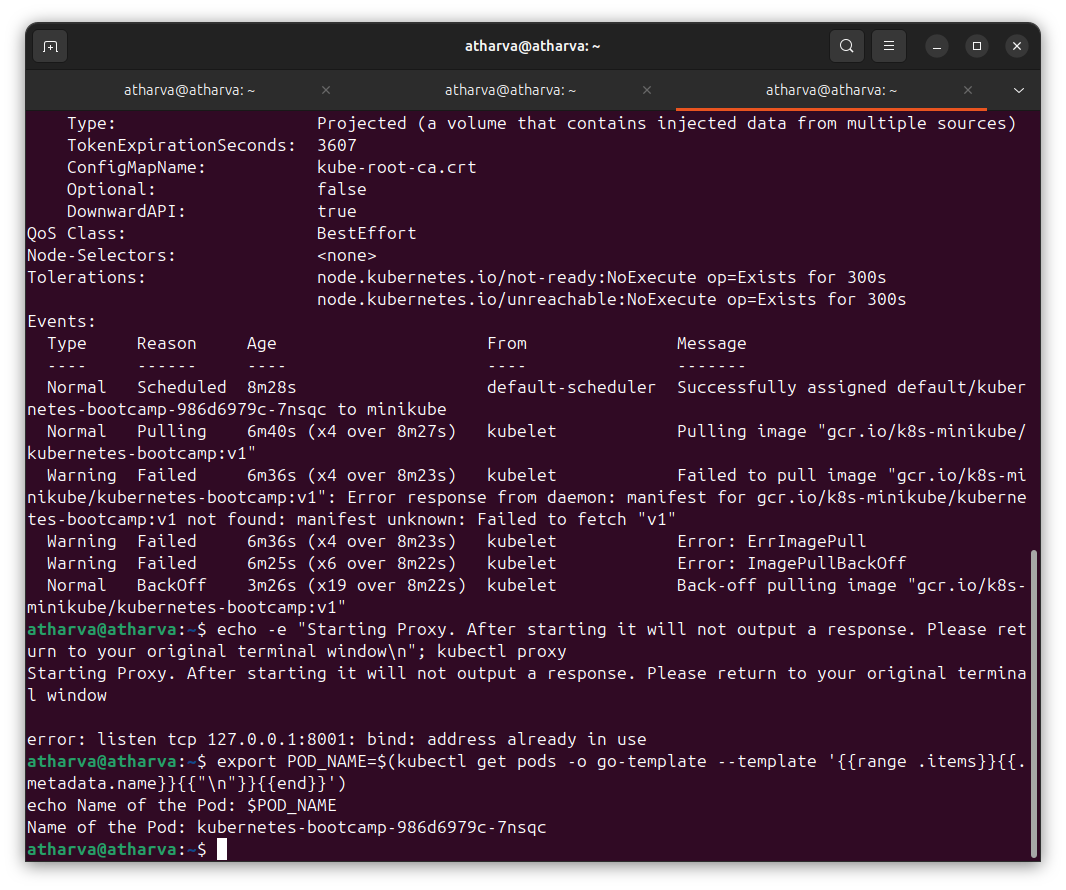
echo -e "Starting Proxy. After starting it will not output a response. Please return to your original terminal window\n"; kubectl proxy



Now again, we’ll get the Pod name and query that pod directly through the proxy. To get the Pod name and store it in the POD\_NAME ennvironment variable:

export POD\_NAME=$(kubectl get pods -o go-template --template '{{range .items}}{{.metadata.name}}{{"\n"}}{{end}}')

echo Name of the Pod: $POD\_NAME



To see the output of our application, run a curl request.

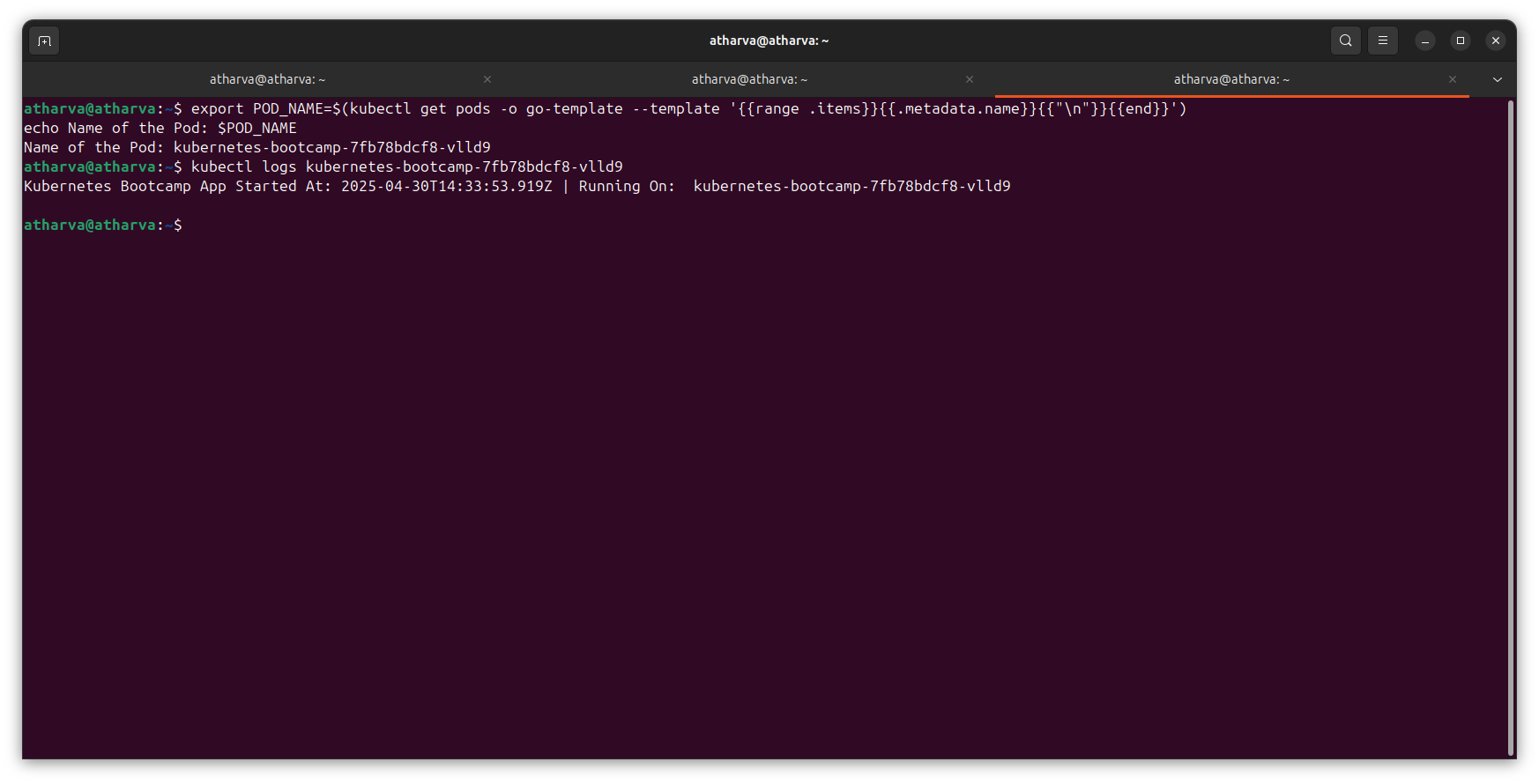
curl <http://localhost:8001/api/v1/namespaces/default/pods/$POD_NAME>

The url is the route to the API of the Pod.

## Step 3 - View the container logs

Anything that the application would normally send to STDOUT becomes logs for the container within the Pod. We can retrieve these logs using the kubectl logs command:

kubectl logs $POD\_NAME

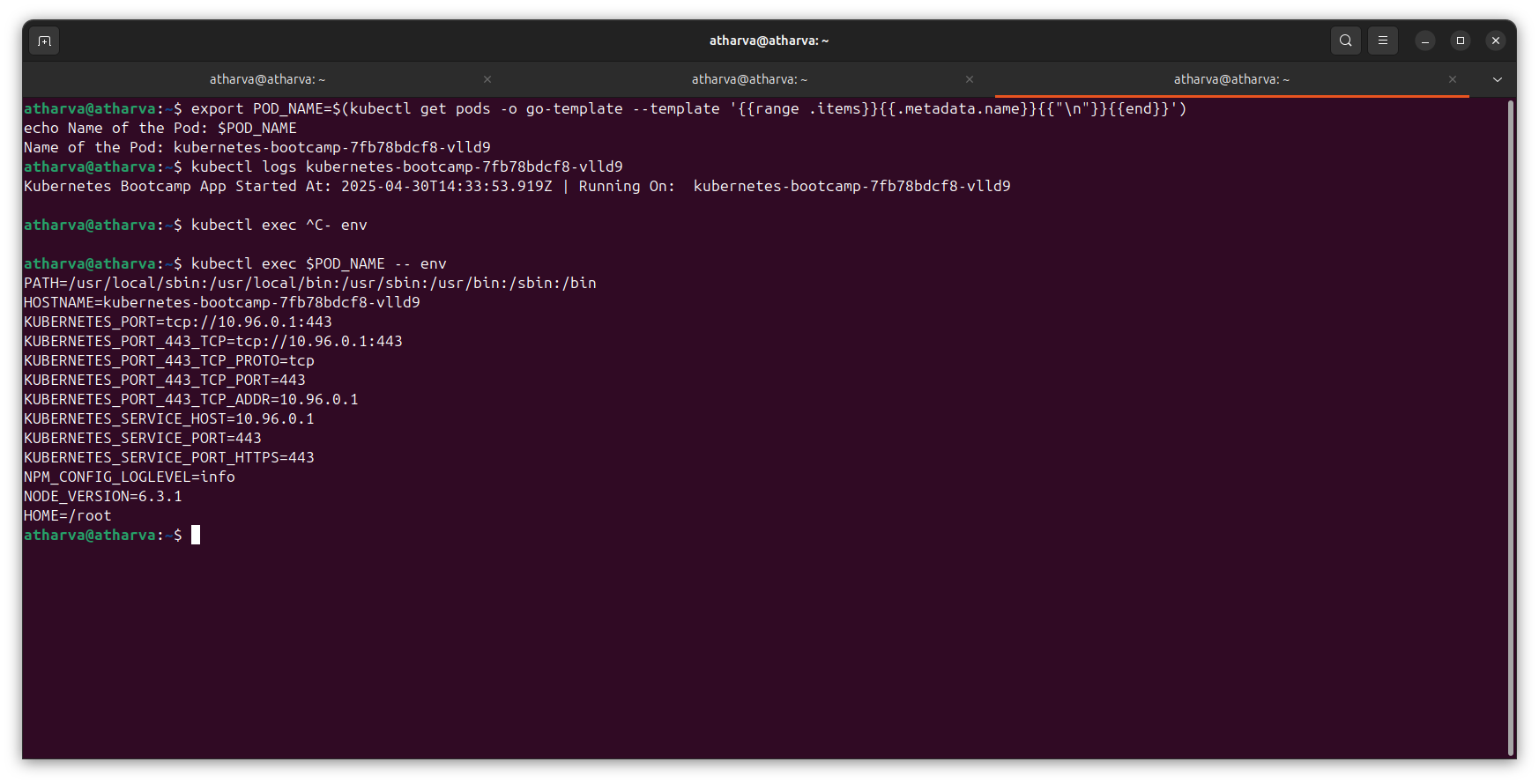


\*Note: We don’t need to specify the container name, because we only have one container inside the pod.

## Step 4 - Executing command on the container

We can execute commands directly on the container once the Pod is up and running. For this, we use the exec command and use the name of the Pod as a parameter. Let’s list the environment variables:

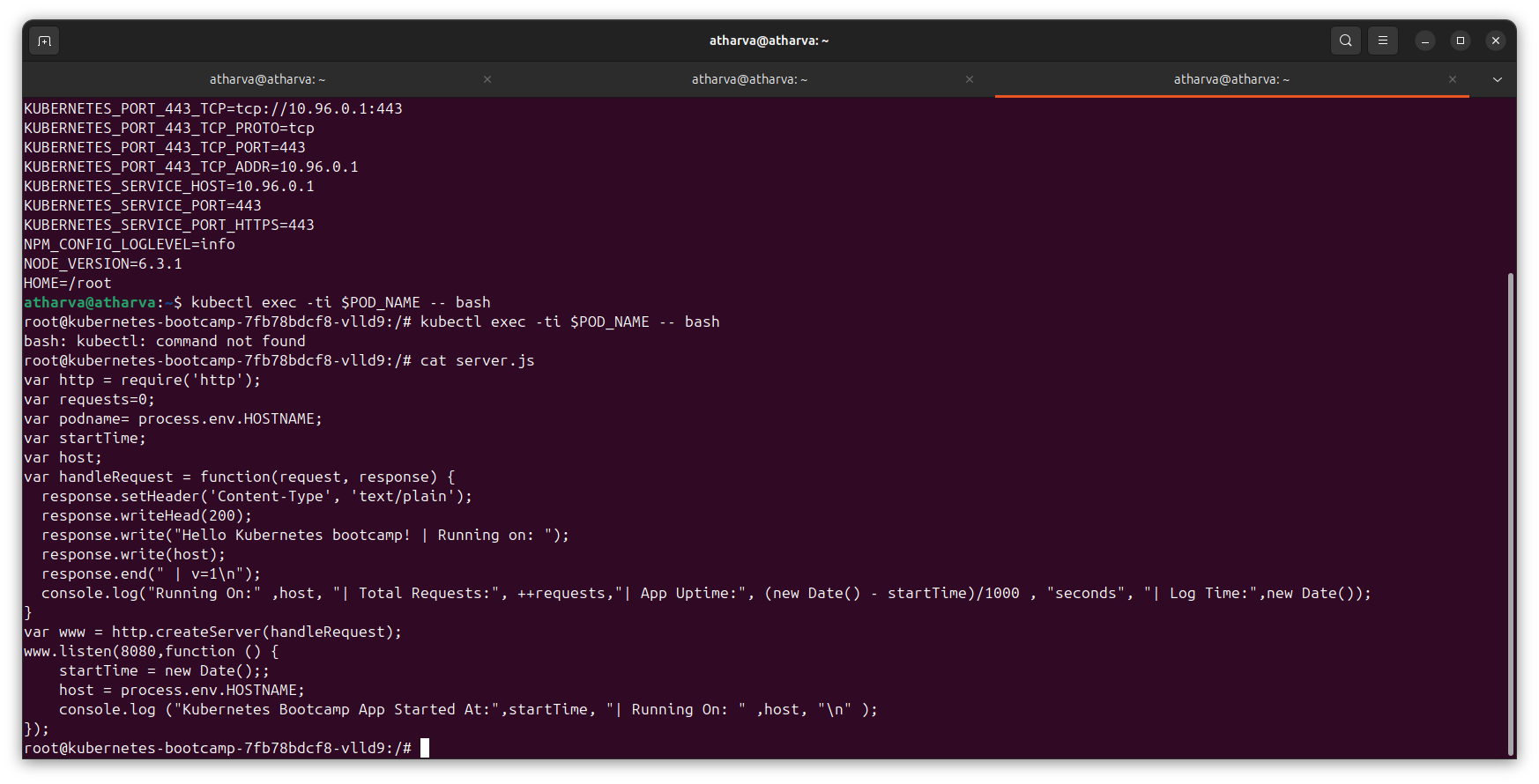
kubectl exec $POD\_NAME -- env



Again, worth mentioning that the name of the container itself can be omitted since we only have a single container in the Pod.

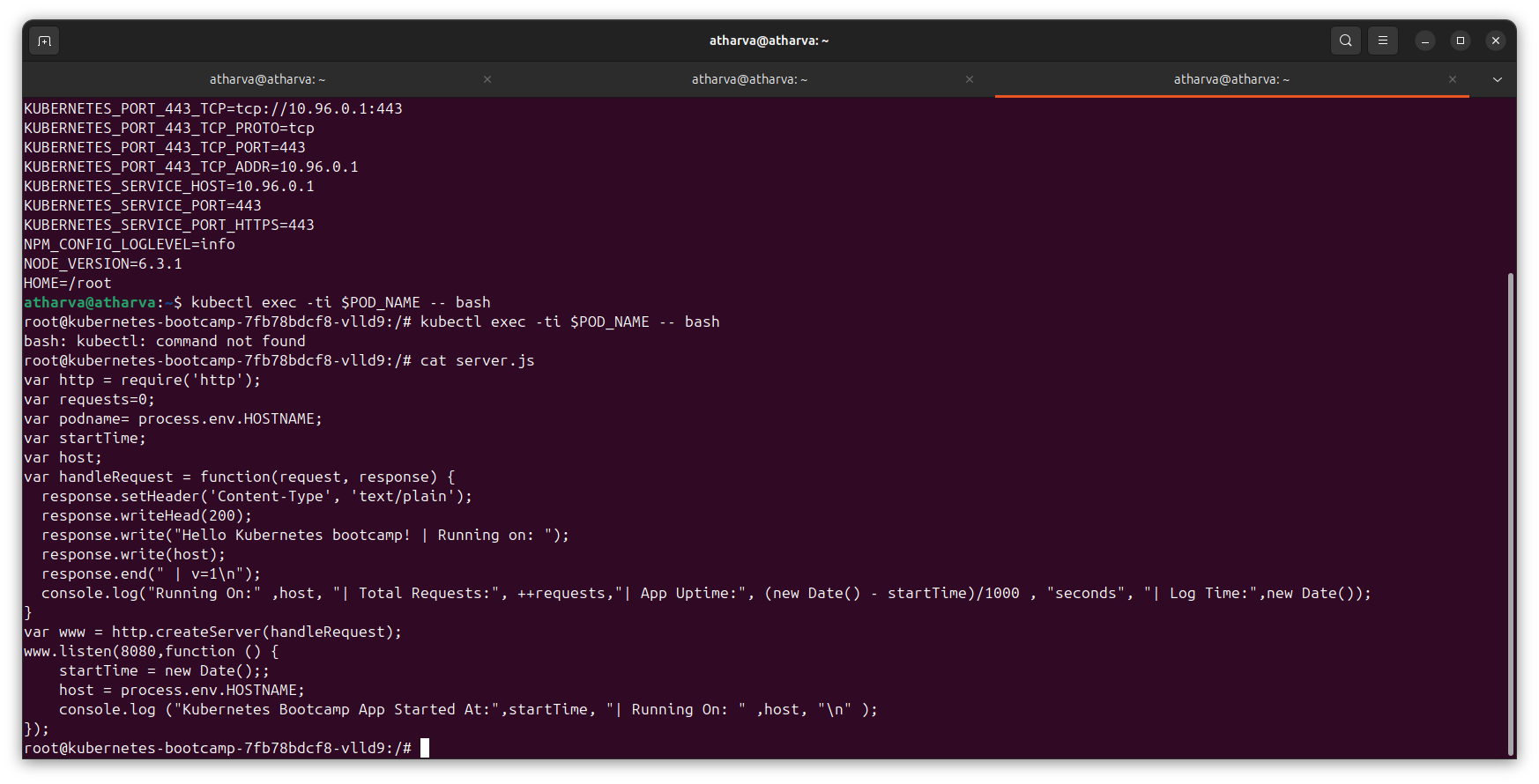
Next let’s start a bash session in the Pod’s container:

kubectl exec -ti $POD\_NAME -- bash



We have now an open console on the container where we run our NodeJS application. The source code of the app is in the server.js file:

cat server.js



You can check that the application is up by running a curl command:

curl localhost:8080

# **Module 4 - Expose your app publicly**

## Step 1 - Create a new service

Let’s verify that our application is running. We’ll use the kubectl get command and look for existing Pods:

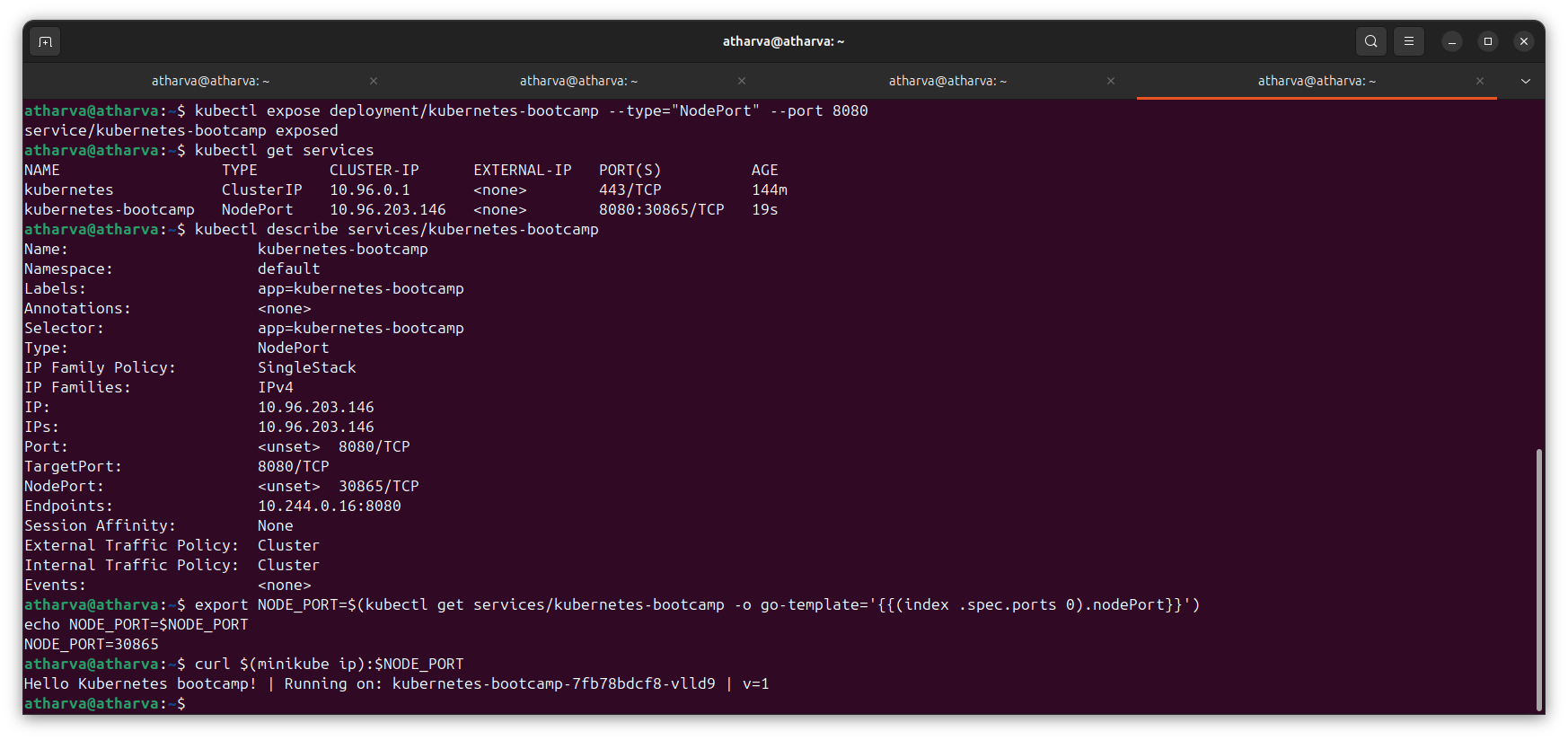
kubectl get pods

Next, let’s list the current Services from our cluster:

kubectl get services

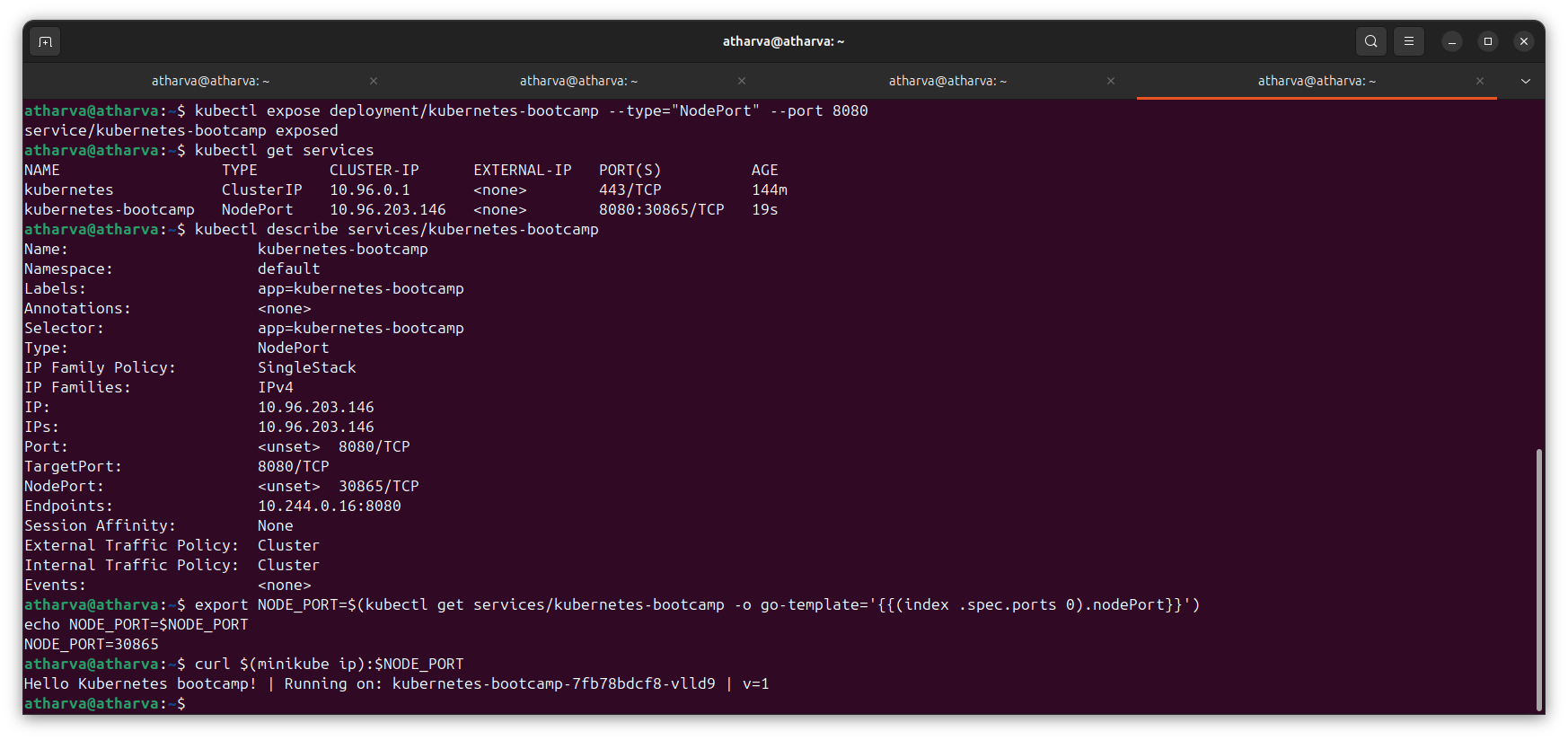
We have a Service called Kubernetes that is created by default when minikube starts the cluster. To create a new service and expose it to external traffic we’ll use the expose command with NodePort as parameter.

kubectl expose deployment/kubernetes-bootcamp --type="NodePort" --port 8080



Let’s run again the get services command:

kubectl get services

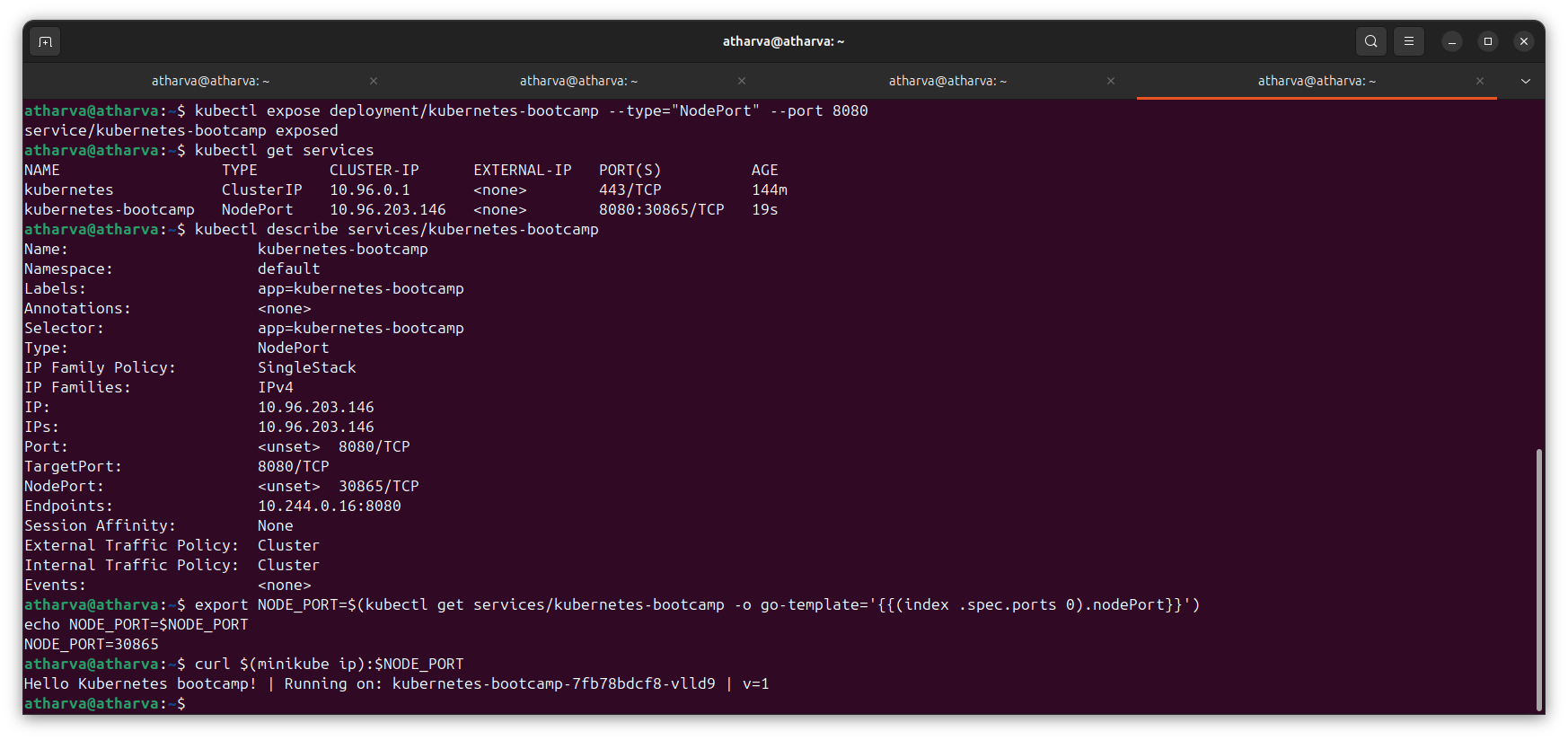


**Note for Docker Desktop users:** Due to Docker Desktop networking limitations, by default you’re unable to access pods directly from the host. Run minikube service kubernetes-bootcamp, this will create a SSH tunnel from the pod to your host and open a window in your default browser that’s connected to the service. The tunnel can be terminated by pressing control-C, then continue on to Step 2.

We have now a running Service called kubernetes-bootcamp. Here we see that the Service received a unique cluster-IP, an internal port and an external-IP (the IP of the Node).

To find out what port was opened externally (by the NodePort option) we’ll run the describe service command:

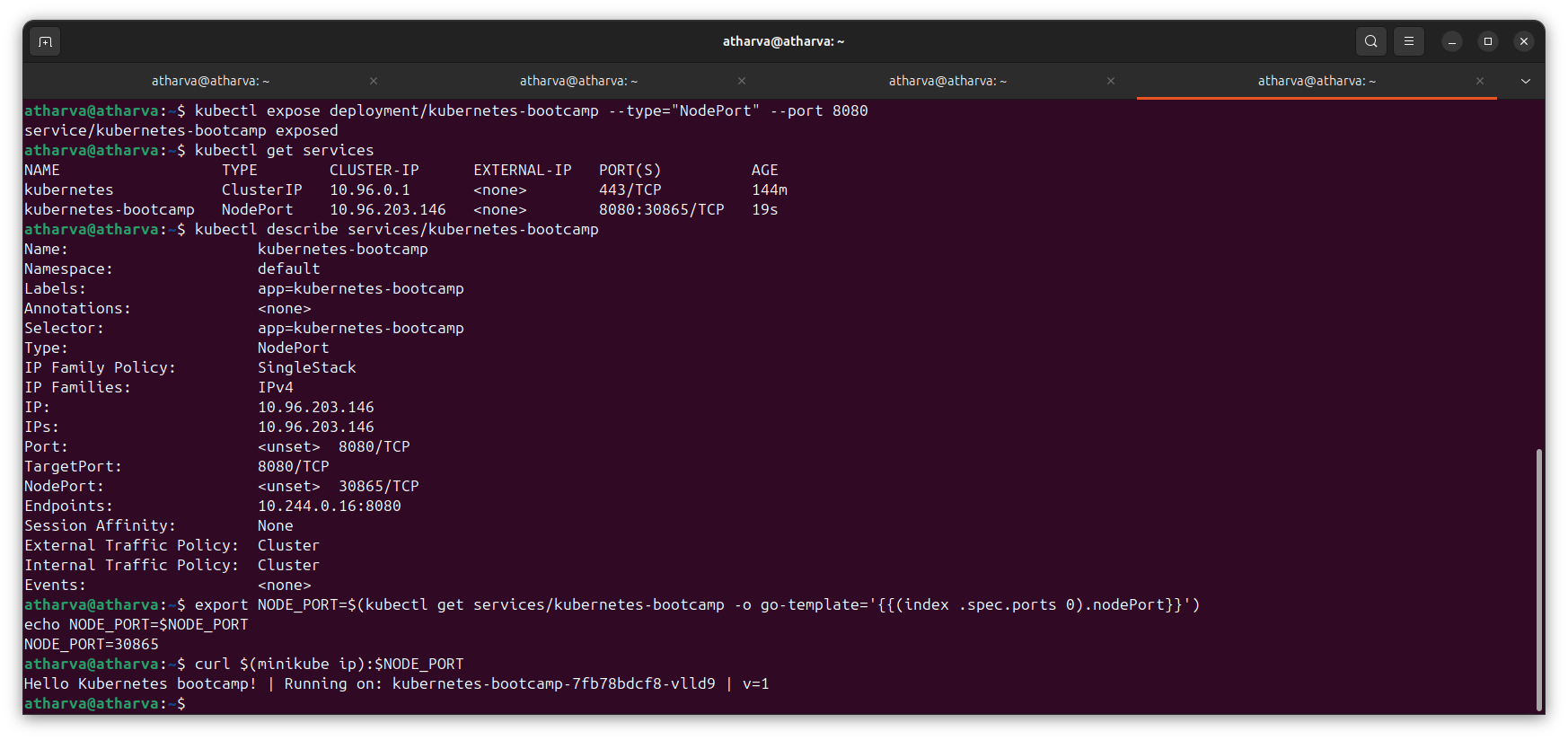
kubectl describe services/kubernetes-bootcamp



Create an environment variable called NODE\_PORT that has the value of the Node port assigned:

export NODE\_PORT=$(kubectl get services/kubernetes-bootcamp -o go-template='{{(index .spec.ports 0).nodePort}}')

echo NODE\_PORT=$NODE\_PORT



Now we can test that the app is exposed outside of the cluster using curl, the IP of the Node and the externally exposed port:

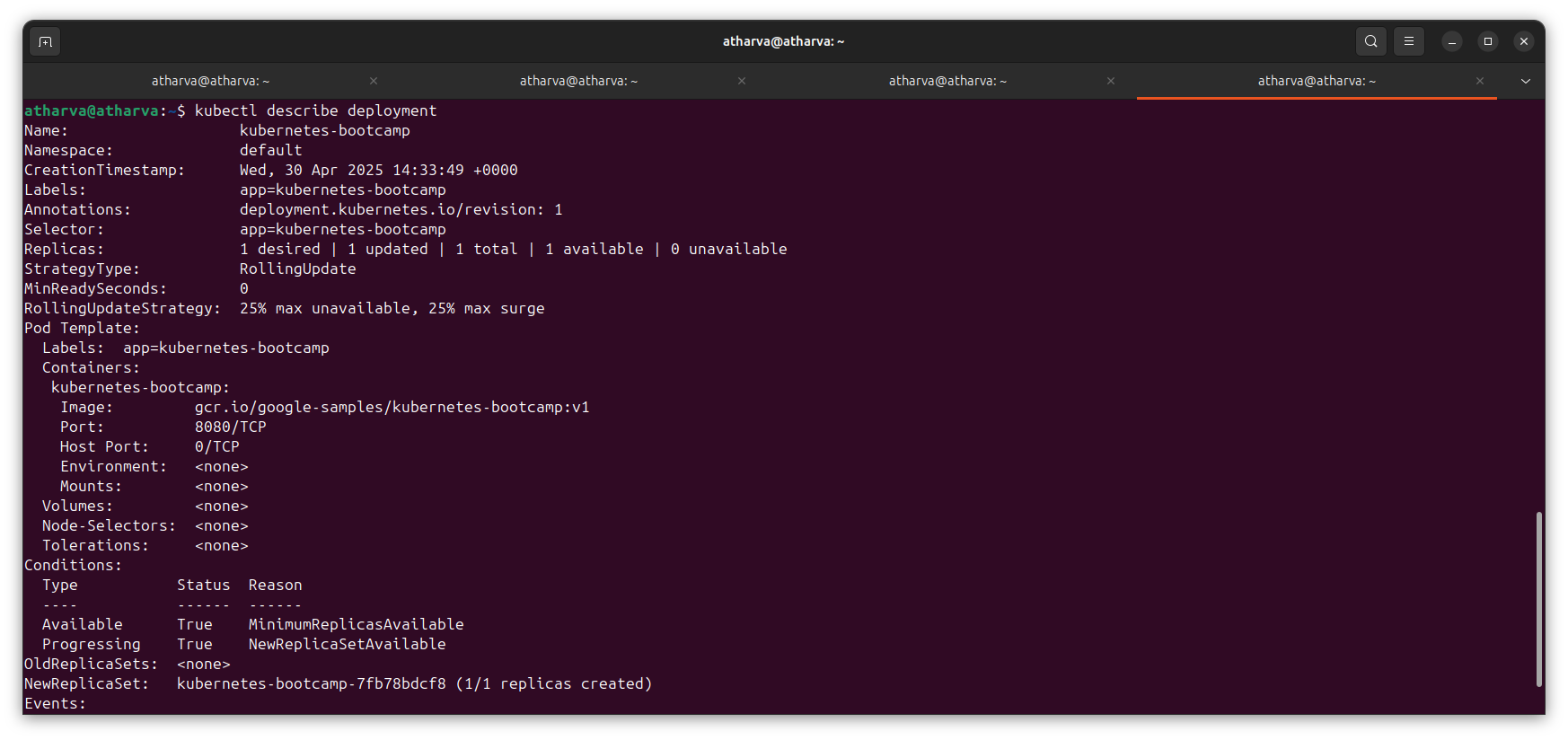
curl $(minikube ip):$NODE\_PORT

And we get a response from the server. The Service is exposed.

## Step 2 - Using labels

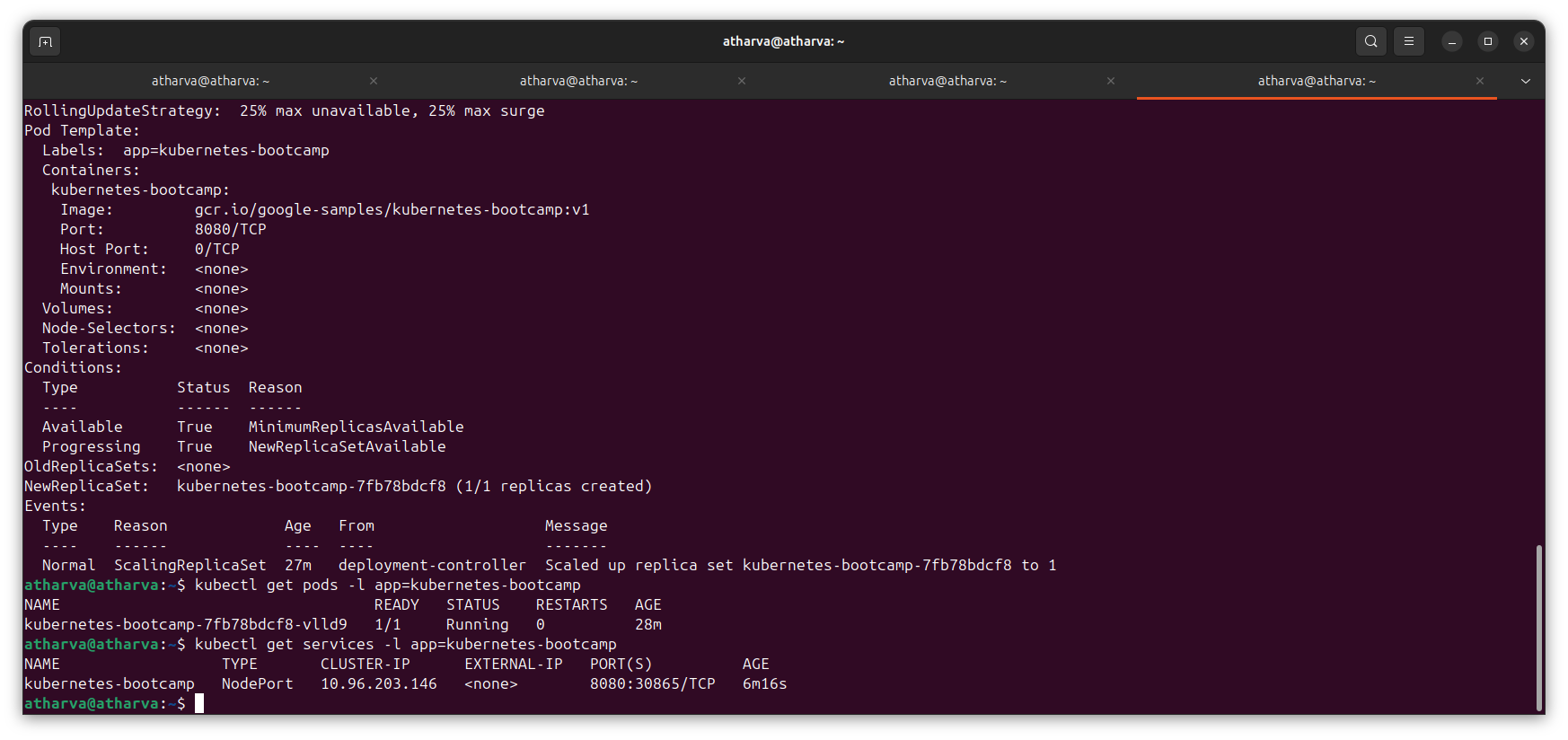
The Deployment created automatically a label for our Pod. With describe deployment command you can see the name of the label:

kubectl describe deployment



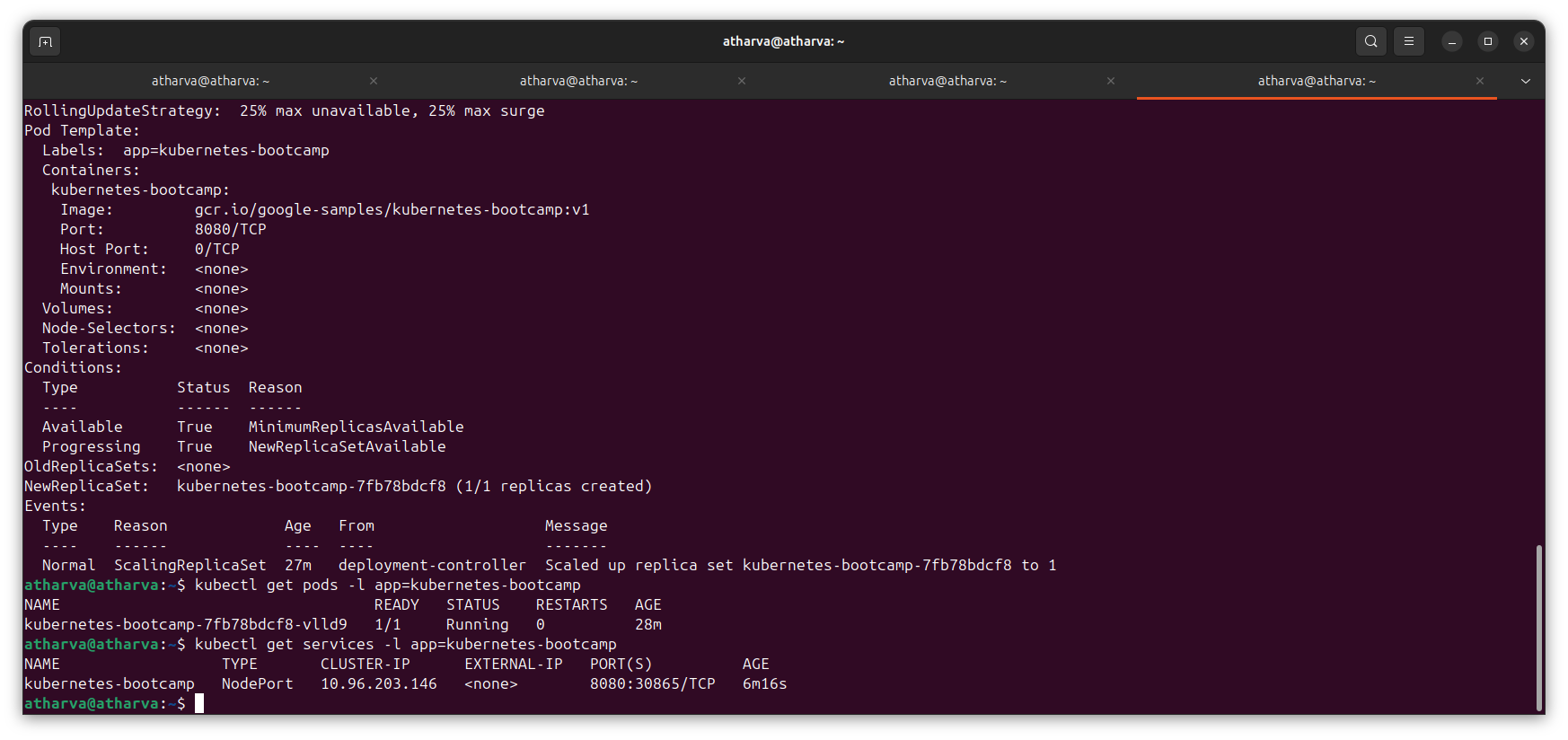
Let’s use this label to query our list of Pods. We’ll use the kubectl get pods command with -l as a parameter, followed by the label values:

kubectl get pods -l app=kubernetes-bootcamp



You can do the same to list the existing services:

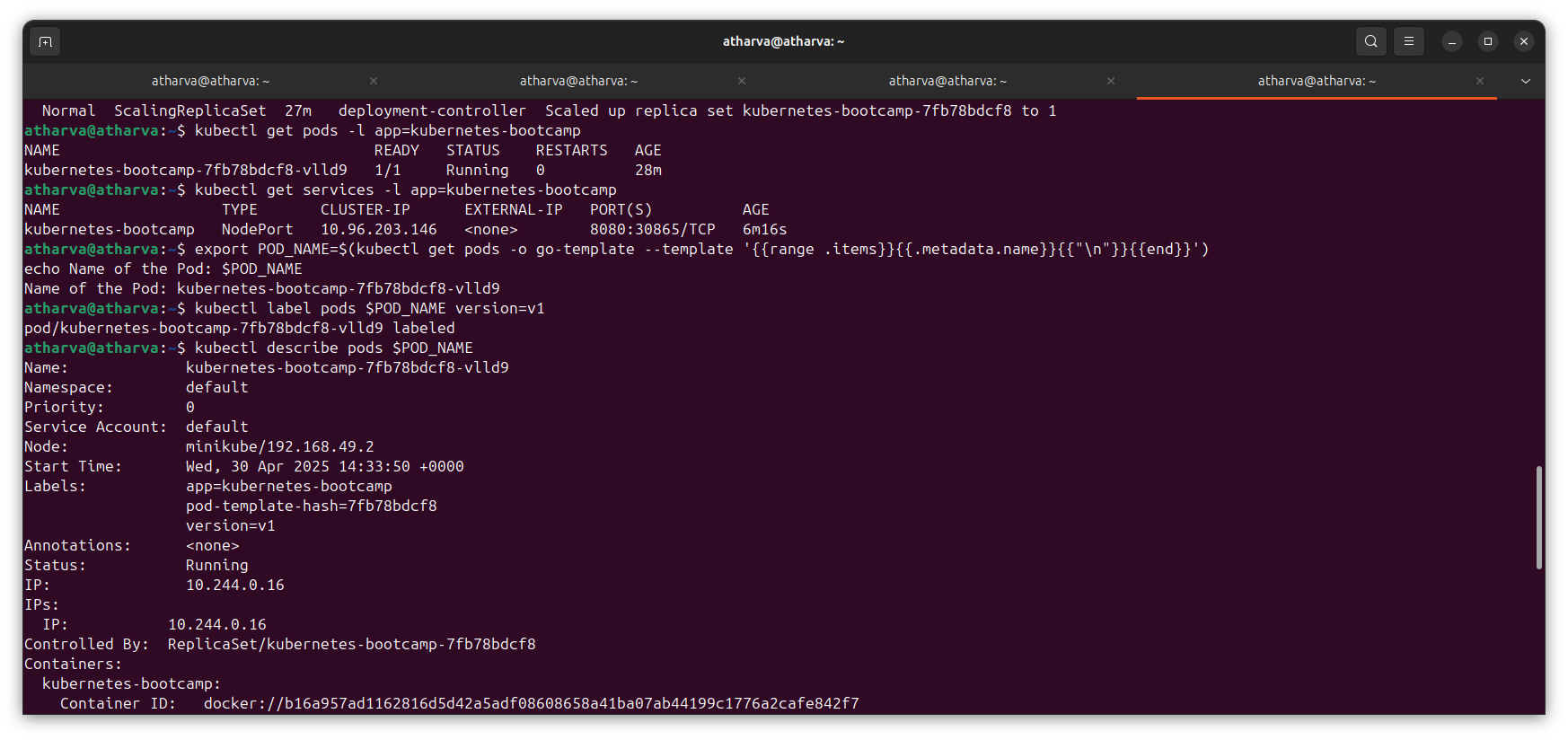
kubectl get services -l app=kubernetes-bootcamp



Get the name of the Pod and store it in the POD\_NAME environment variable:

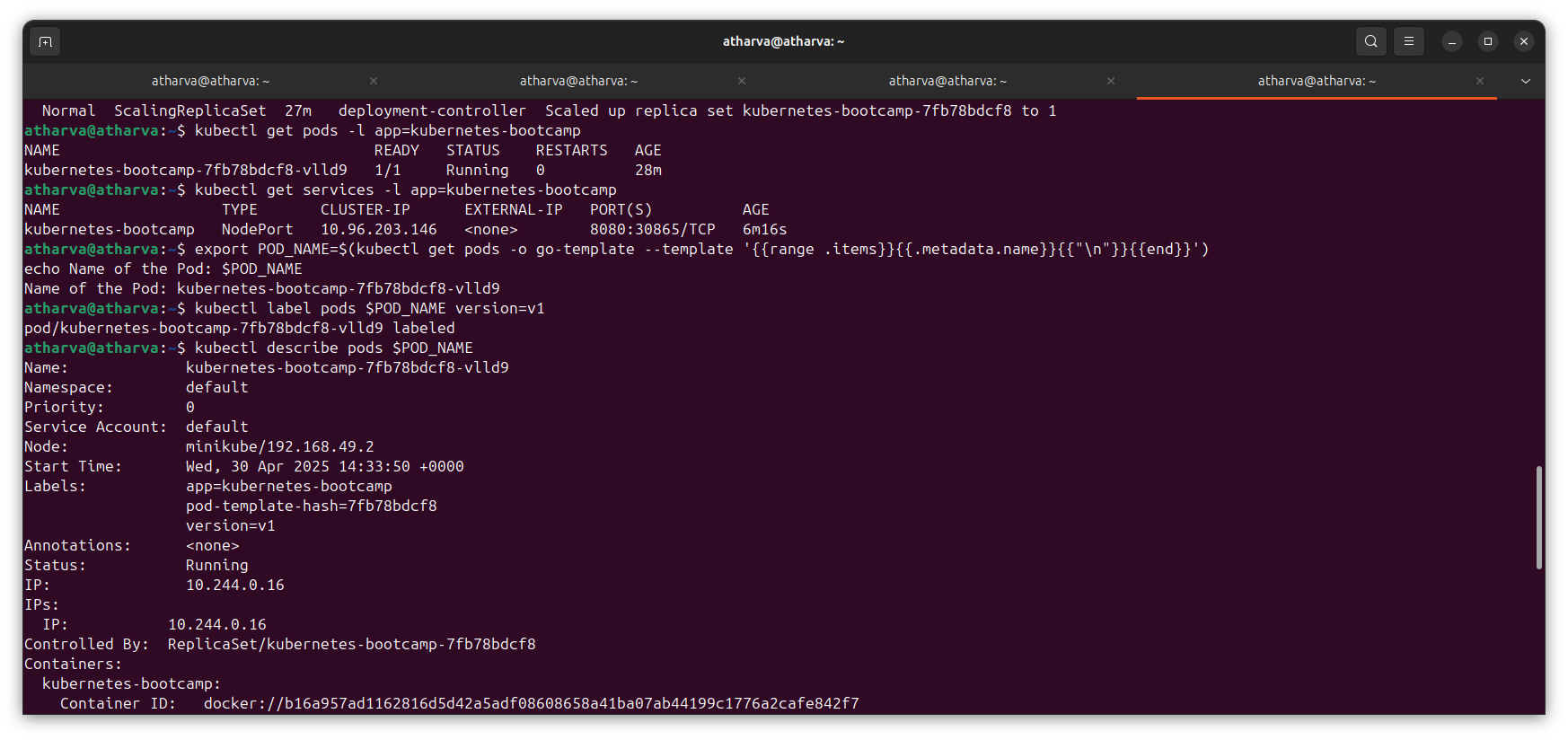
export POD\_NAME=$(kubectl get pods -o go-template --template '{{range .items}}{{.metadata.name}}{{"\n"}}{{end}}')

echo Name of the Pod: $POD\_NAME



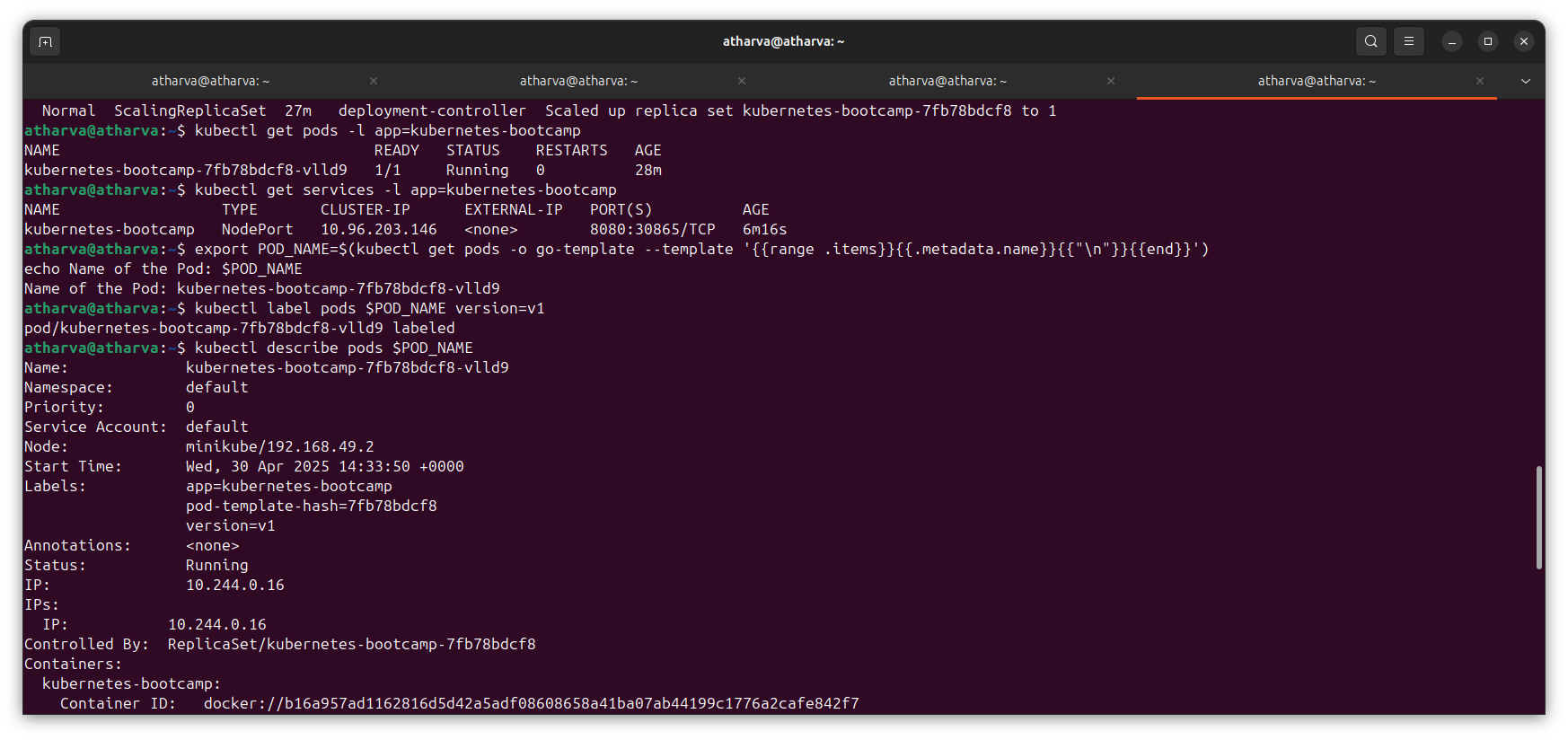
To apply a new label we use the label command followed by the object type, object name and the new label:

kubectl label pods $POD\_NAME version=v1



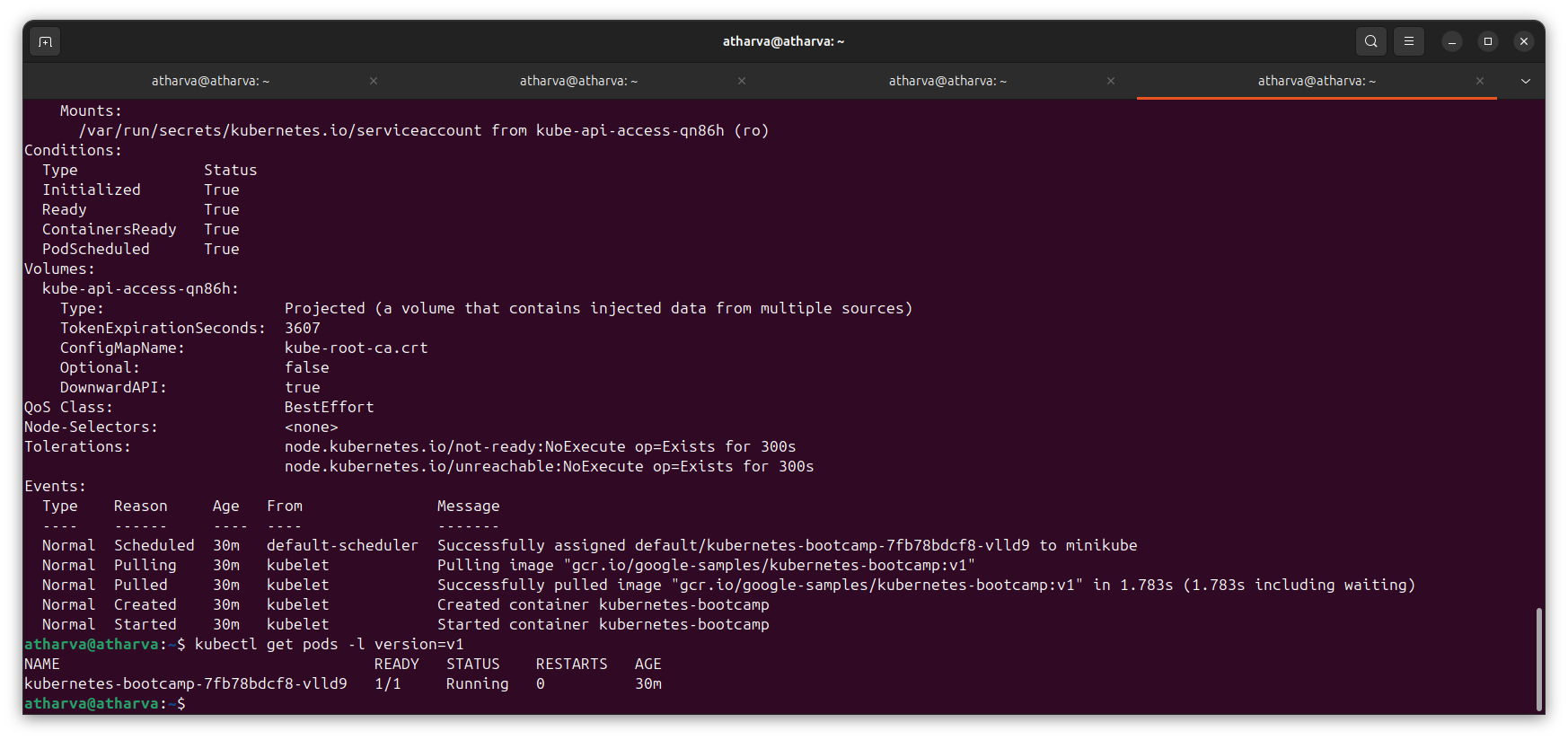
This will apply a new label to our Pod (we pinned the application version to the Pod), and we can check it with the describe pod command:

kubectl describe pods $POD\_NAME



We see here that the label is attached new to our Pod. And we can query now the list of pods using the new label:

kubectl get pods -l version=v1

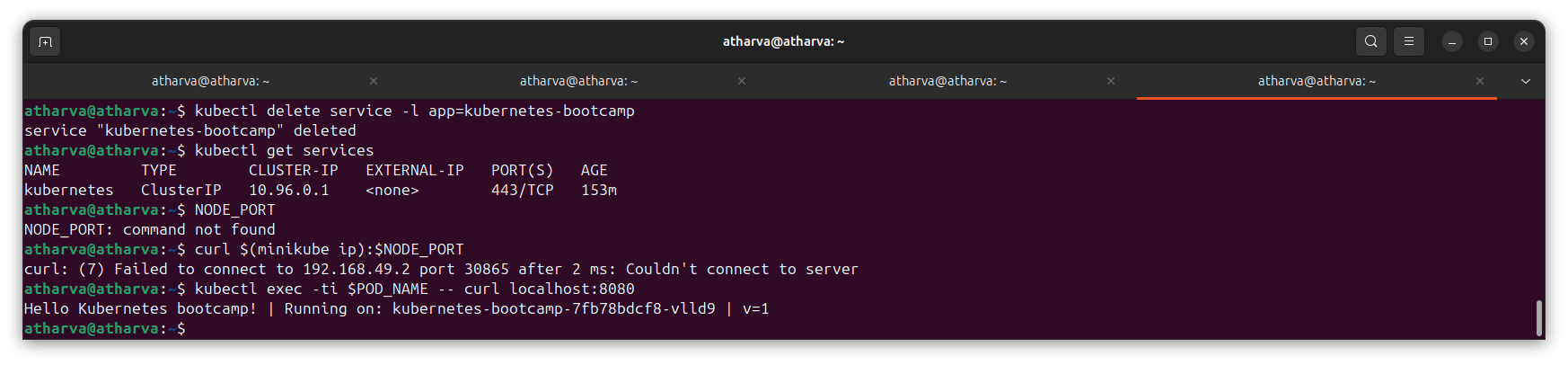


And we see the Pod.

## Step 3 - Deleting a service

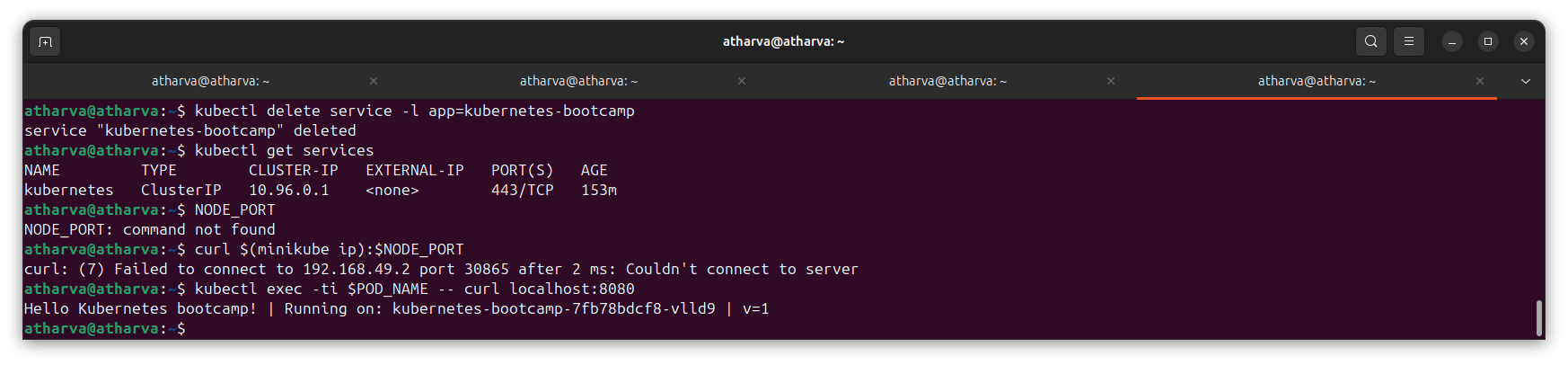
To delete Services you can use the delete service command. Labels can be used also here:

kubectl delete service -l app=kubernetes-bootcamp



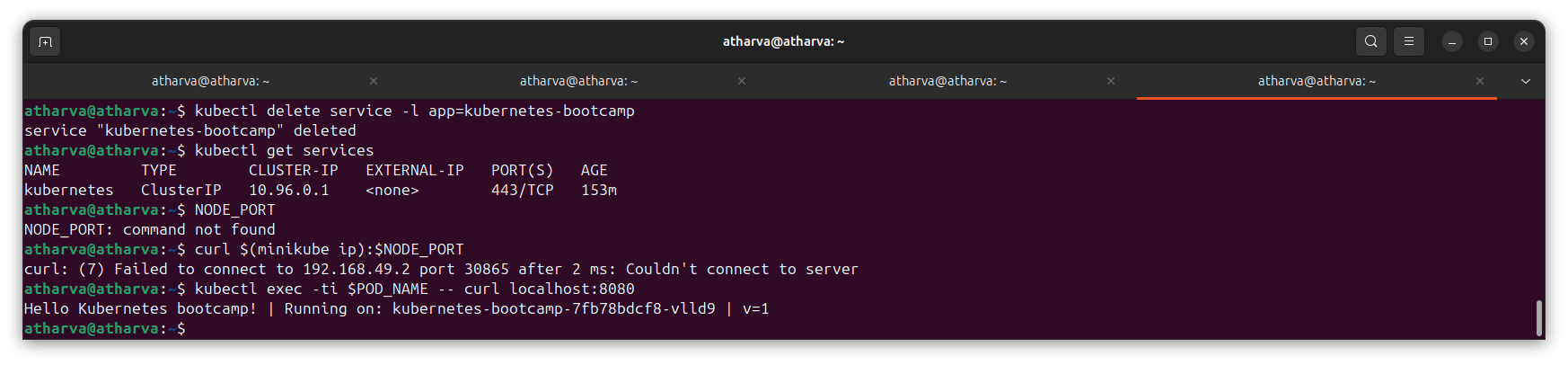
Confirm that the service is gone:

kubectl get services



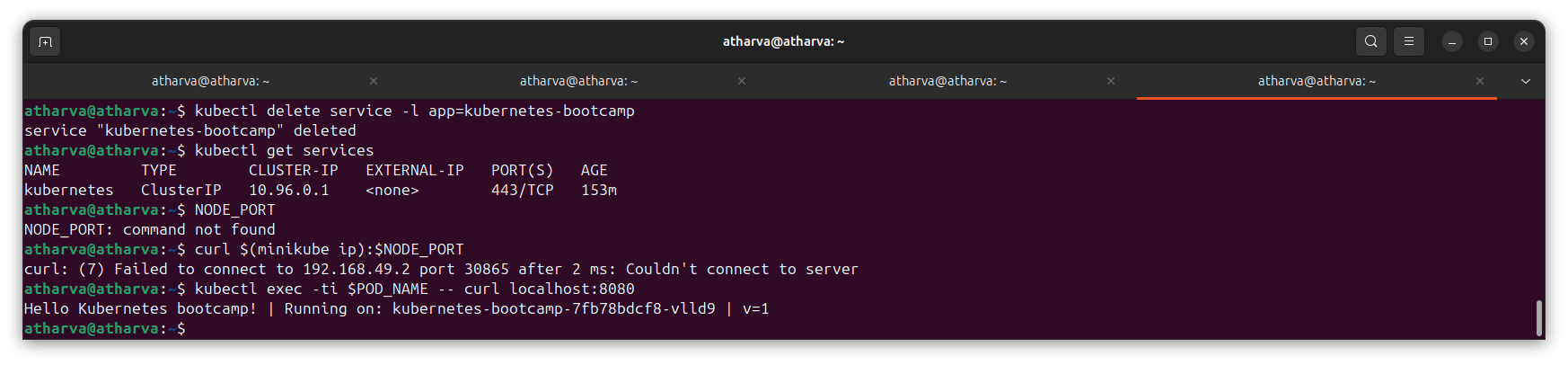
This confirms that our Service was removed. To confirm that route is not exposed anymore you can curl the previously exposed IP and port:

curl $(minikube ip):$NODE\_PORT



This proves that the app is not reachable anymore from outside of the cluster. You can confirm that the app is still running with a curl inside the pod:

kubectl exec -ti $POD\_NAME -- curl localhost:8080



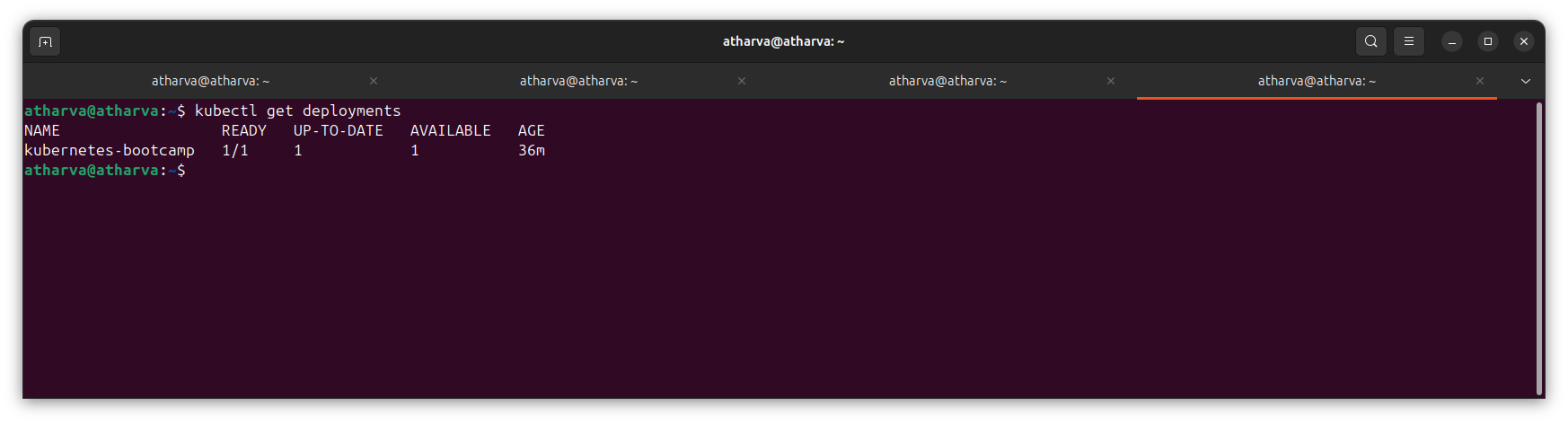
We see here that the application is up. This is because the Deployment is managing the application. To shut down the application, you would need to delete the Deployment as well.

# **Module 5 - Scale up your app**

## Step 1 - Scaling a deployment

First, let’s list the deployments using the get deployment command:

kubectl get deployments



The output should be similar to:

NAME READY UP-TO\_DATE AVAILABLE AGE

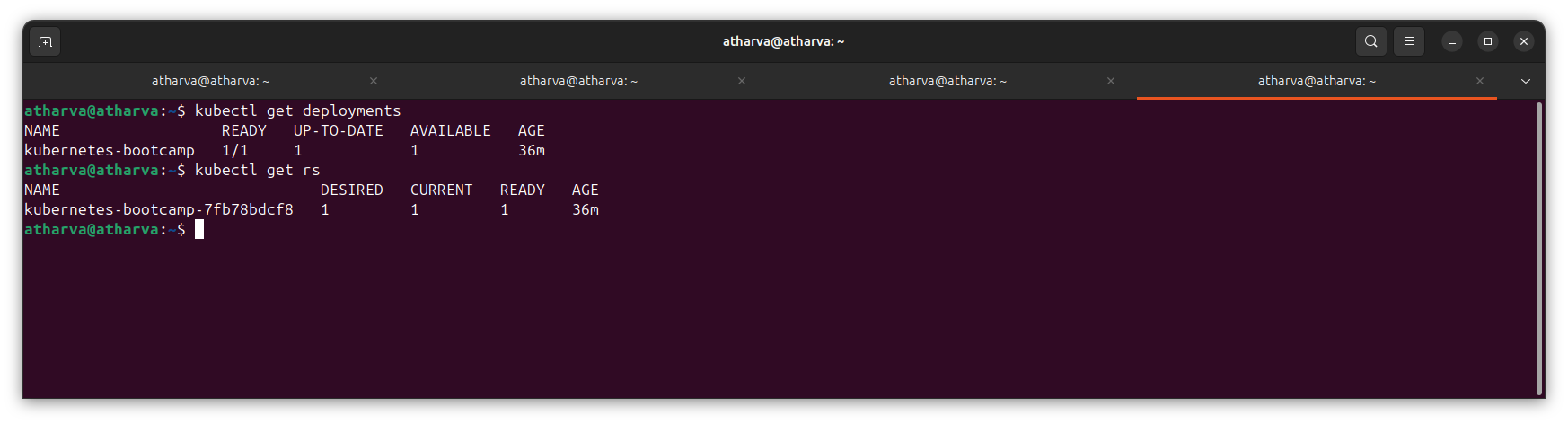
kubernetes-bootcamp 1/1 1 1 11m

We should have 1 Pod. If not, run the command again. This shows:

* *NAME* lists the names of the Deployments in the cluster.
* *READY* shows the ratio of CURRENT/DESIRED replicas
* *UP-TO-DATE* displays the number of replicas that have been updated to achieve the desired state.
* *AVAILABLE* displays how many replicas of the application are available to your users.
* *AGE* displays the amount of time that the application has been running.

To see the ReplicaSet created by the Deployment, run:

kubectl get rs



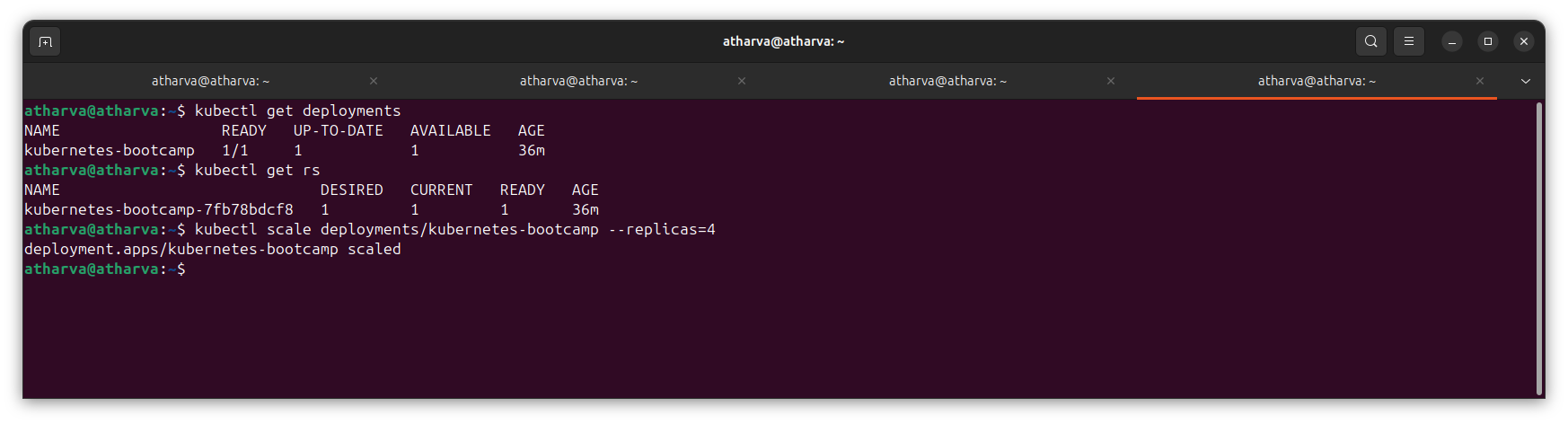
Notice that the name of the ReplicaSet is always formatted as [DEPLOYMENT-NAME]-[RANDOM-STRING]. The random string is randomly generated and uses the pod-template-hash as a seed.

Two important columns of this command are:

* *DESIRED* displays the desired number of replicas of the application, which you define when you create the Deployment. This is the desired state.
* *CURRENT* displays how many replicas are currently running.

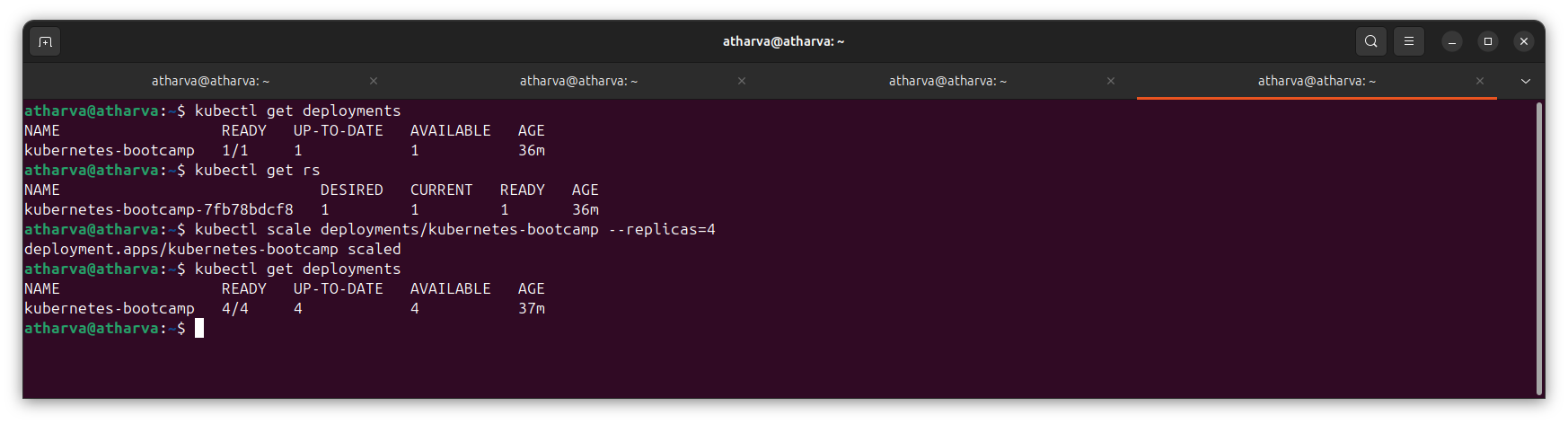
Next, let’s scale the Deployment to 4 replicas. We’ll use the kubectl scale command, following by the deployment type, name and desired number of instances:

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



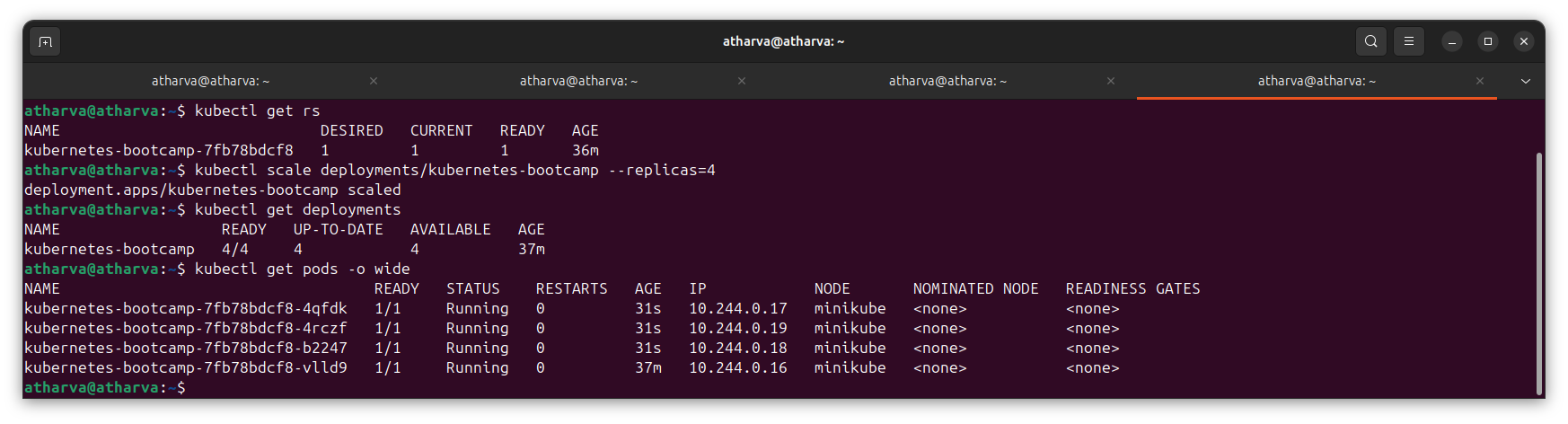
To list your Deployments once again, use get deployments:

kubectl get deployments



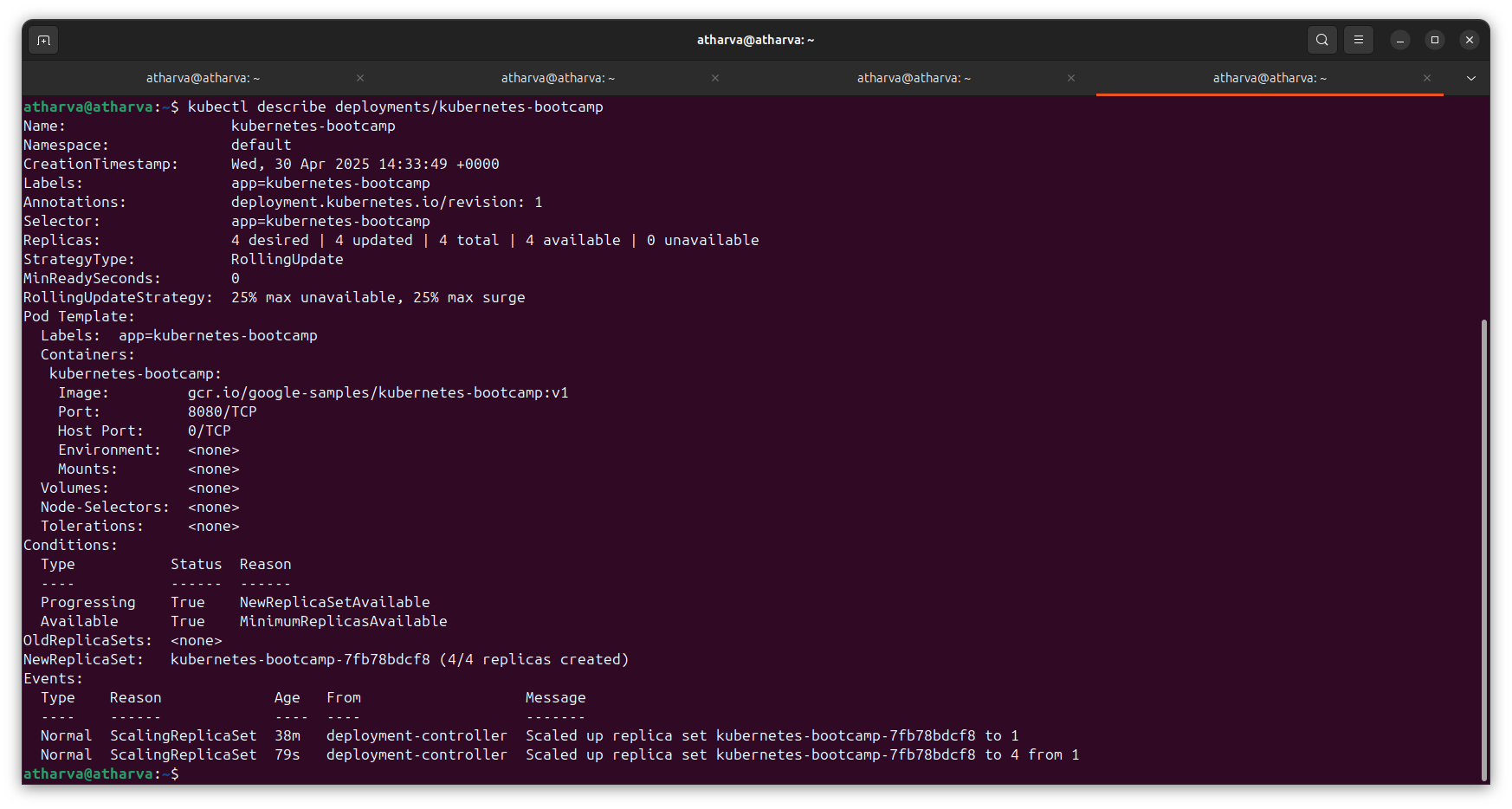
The change was applied, and we have 4 instances of the application available. Next, let’s check if the number of Pods changed:

kubectl get pods -o wide



There are 4 Pods now, with different IP addresses. The change was registered in the Deployment events log. The check that, use the describe command:

kubectl describe deployments/kubernetes-bootcamp



You can also view in the output of this command that there are 4 replicas now.

## 

## Step 2 - Load Balancing

Let’s check that the Service is load-balancing the traffic. To find out the exposed IP and Port we can use the describe service as we learned in the previous Module:

kubectl describe services/kubernetes-bootcamp

**Note for Docker Desktop users:** Due to Docker Desktop networking limitations, by default you’re unable to access pods directly from the host. Run minikube service kubernetes-bootcamp, this will create a SSH tunnel from the pod to your host and open a window in your default browser that’s connected to the service. Refresh the browser page to see the load-balancing working. The tunnel can be terminated by pressing control-C, then continue on to Step 3.

Create an environment variable called NODE\_PORT that has a value as the Node port:

export NODE\_PORT=$(kubectl get services/kubernetes-bootcamp -o go-template='{{(index .spec.ports 0).nodePort}}')

echo NODE\_PORT=$NODE\_PORT

Next, we’ll do a curl to the exposed IP and port. Execute the command multiple times:

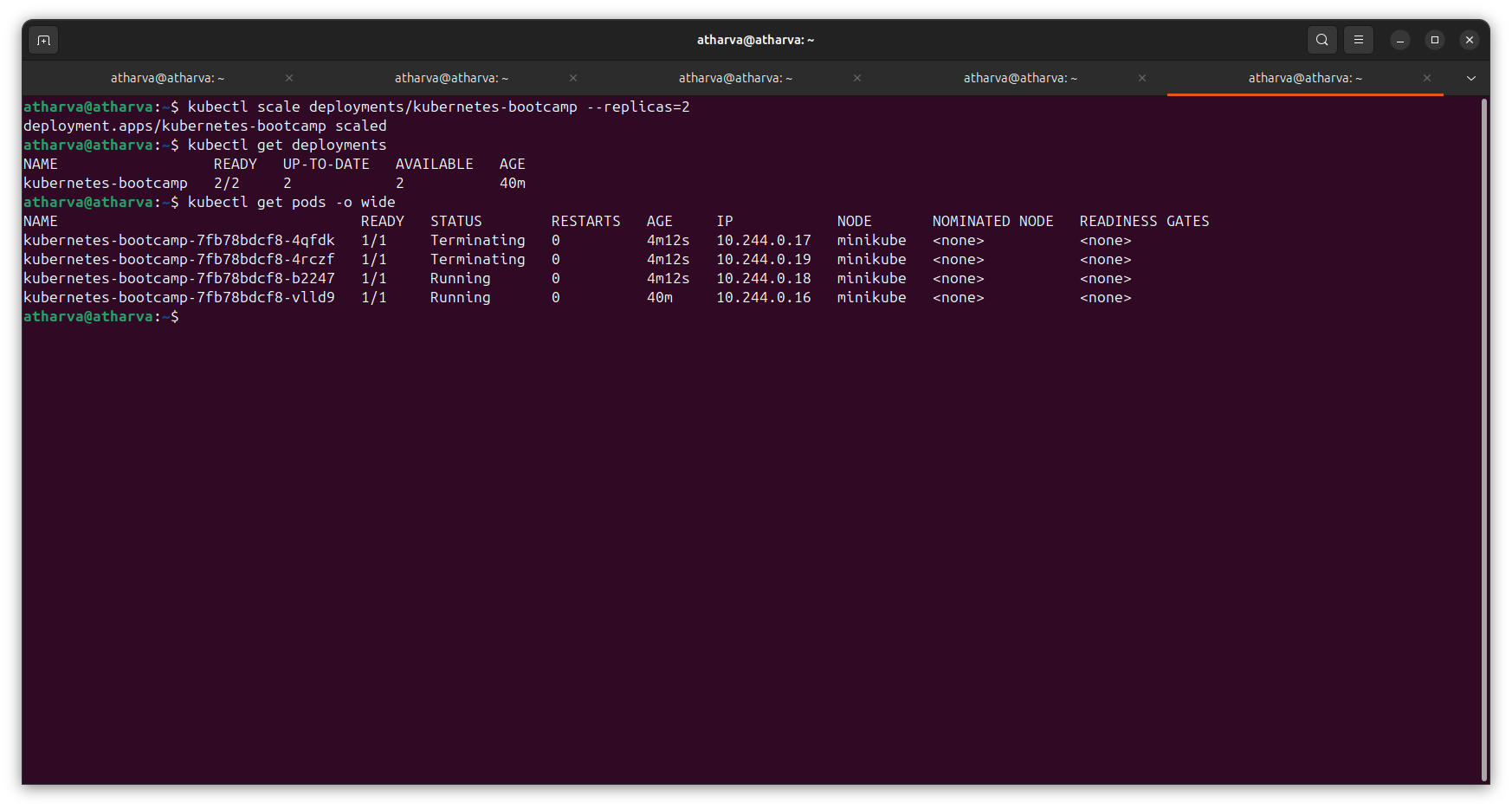
curl $(minikube ip):$NODE\_PORT

We hit a different Pod with every request. This demonstrates that the load-balancing is working.

## Step 3 - Scale Down

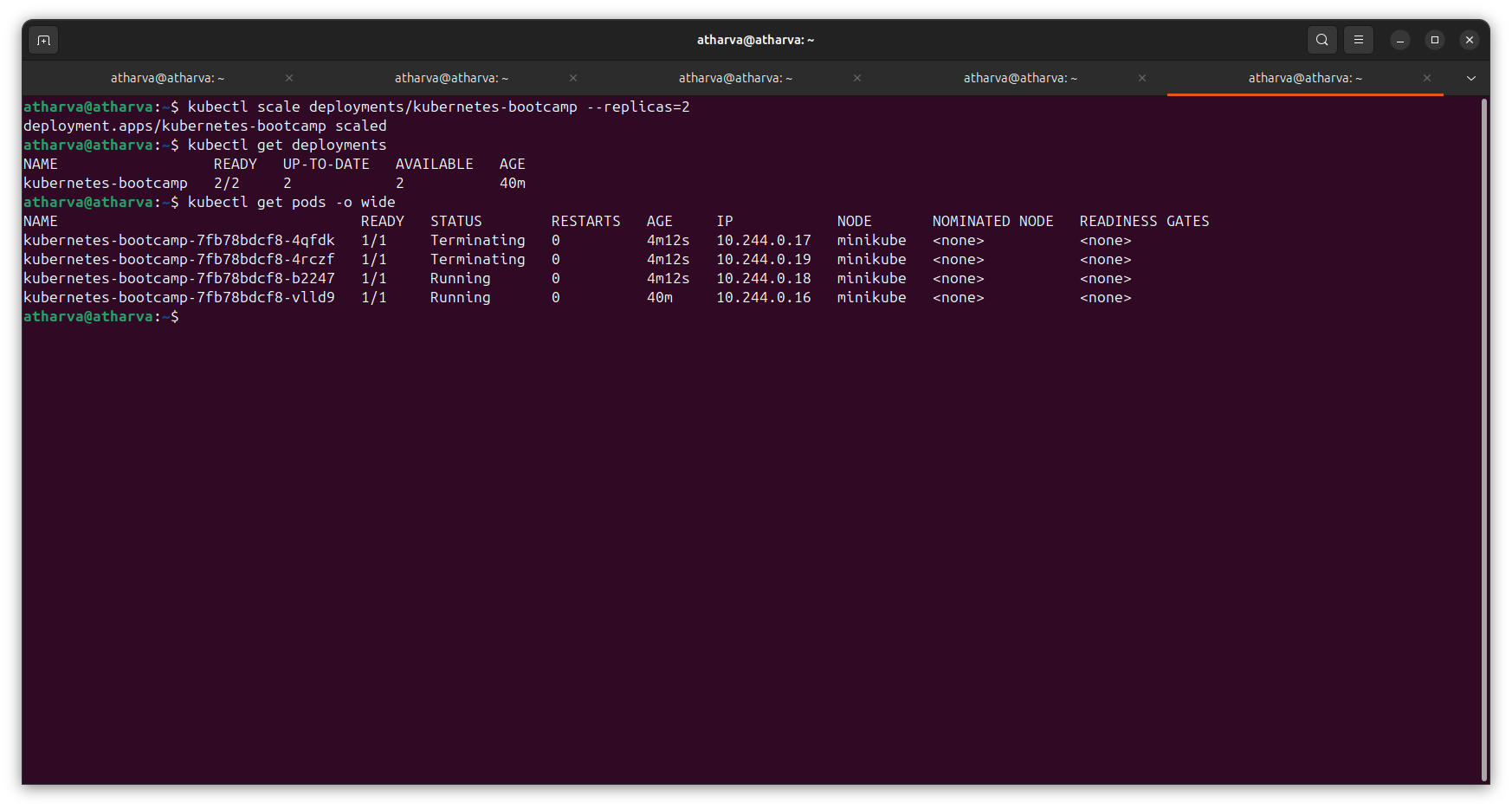
To scale down the Service to 2 replicas, run again the scale command:

kubectl scale deployments/kubernetes-bootcamp --replicas=2



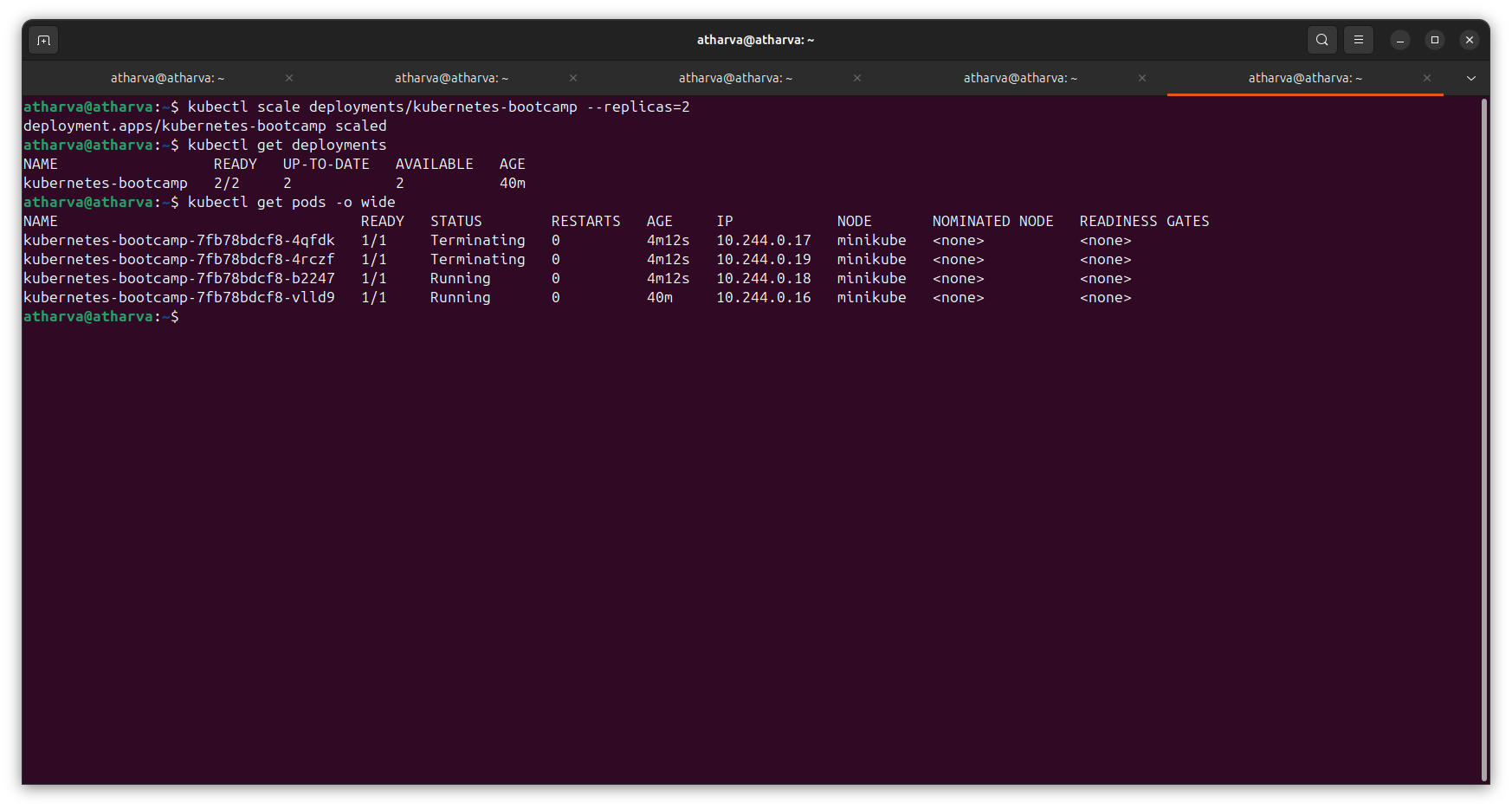
List the Deployments to check if the change was applied with the get deployments command:

kubectl get deployments



The number of replicas decreased to 2. List the number of Pods, with get pods:

kubectl get pods -o wide



This confirms that 2 Pods were terminated.

**Conclusion:** In conclusion, the Kubernetes Bootcamp with Minikube helped us understand the basics of using Kubernetes in a simple and practical way. We installed Minikube on Ubuntu, created a cluster, and used kubectl to interact with it. We learned how to deploy an app, check its status, view logs, and run commands inside it. We also exposed the app so it could be accessed from outside. Overall, this bootcamp gave us a good starting point to use Kubernetes for managing applications