

**Configuration & Deployment of Program API in multiple pods in different nodes**

**Version History**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| SI | Date | Ver | Description | Changed by | Approved/Reviewed by |
| 01 | 24-Oct-21 | 1.0 | Configuration & Deployment of Program API in multiple pods in different nodes. | Maidul Islam  Software Engineer | Md. Mahedee Hasan  Head of Software Development |

Contents

[Introduction 2](#_Toc85963089)

[Configuration Requirements 2](#_Toc85963090)

[Cluster Network Ports 2](#_Toc85963091)

[Step 1: Build and Export Docker Image 4](#_Toc85963092)

[Step 2: Pod Deployment with ReplicaSet 7](#_Toc85963093)

[Step 3: Assign Label for Nodes 9](#_Toc85963094)

[Step 4: Create and Deploy Pod with Node affinity 9](#_Toc85963095)

[Step 5: Deploy a Service with Load Balancer 12](#_Toc85963096)

[Step 6: Verification of Deployments 13](#_Toc85963097)

[Step 7: Verify with Dashboard UI 16](#_Toc85963098)

[Step 8: Activity of Scheduler 18](#_Toc85963099)

[References 22](#_Toc85963100)

# Introduction

Pod and Service deployment are vital part of kubernetes cluster. Pods are the rough equivalent of a machine instance (physical or virtual) to a container. Each pod is allocated its own internal IP address, therefore owning its entire port space, and containers within pods can share their local storage and networking.

Service is an abstraction which defines a logical set of Pods and a policy by which to access them (sometimes this pattern is called a micro-service). The set of Pods targeted by a Service is usually determined by a selector.

We run our application container inside a pod and service helps to expose pod replicaset to outer world.

# Configuration Requirements

In this document, I am going to describe the procedure of Deployment and expose an application in multiple pods in different nodes in Virtual Machines. I have used VMWare Workstation but you can choose any VM according to your preference.

I have configured kubernetes cluster earlier. Please follow the instruction of document ***Kubernetes Installation and Configuration with multi-node cluster using containerd.docx*** to configure the kubernetes cluster with one Master Node and Two Worker Node in Virtual Machine.

I have also created an API in .net Core 5. Please clone the application from below URL.

[**https://github.com/Maidul-Islam-Shawon/ASAI-Program.git**](https://github.com/Maidul-Islam-Shawon/ASAI-Program.git)

# Cluster Network Ports

Network ports are important part of kubernetes cluster. Pod or Service deployment or expose depends on port. Pods or services will not work properly if we do not assign and define port properly.

Control plane node provides a collection of services such as the API server, etcd, and so on. Worker nodes need to access the API server. Specifically, the kubelet and the kube‑proxy on worker nodes will communicate to the API server over TCP/IP.

Let discuss the ports that are required and who uses those ports so that we can develop the firewall rules to help secure your Kubernetes platform. The API server, by default, runs on port 6443. Now that's configurable to any port number, but that's the default port.

Etcd runs on 2379 and 2380. Who needs to talk to etcd? The API server does because etcd is where the API server persists its data. During redundant configuration of etcd, the various replicas of etcd will need to communicate with each other over these ports. And so if we start scaling that out for redundancy purposes, these are the ports that are required for each of those additional etcd replicas.

Next is the scheduler, which runs on 10251, and it's used by itself. It's not exposed to the outside world. Controller manager, which runs on port 10252, and it also is just listening on localhost.

On the control plane node is the kubelet. This runs on port 10250, and all of the control plane components will need access to it inside of our Kubernetes cluster.

Now on the worker node side, there's a kubelet running on each worker node, and it runs on port 10250. The control plane elements will need access to this kubelet.

The NodePort service is a type of service that exposes our services and ports on each individual node in our cluster, and those port ranges are going to be allocated from 30000 to 32767. Anything that would need access to the services published on those ports.

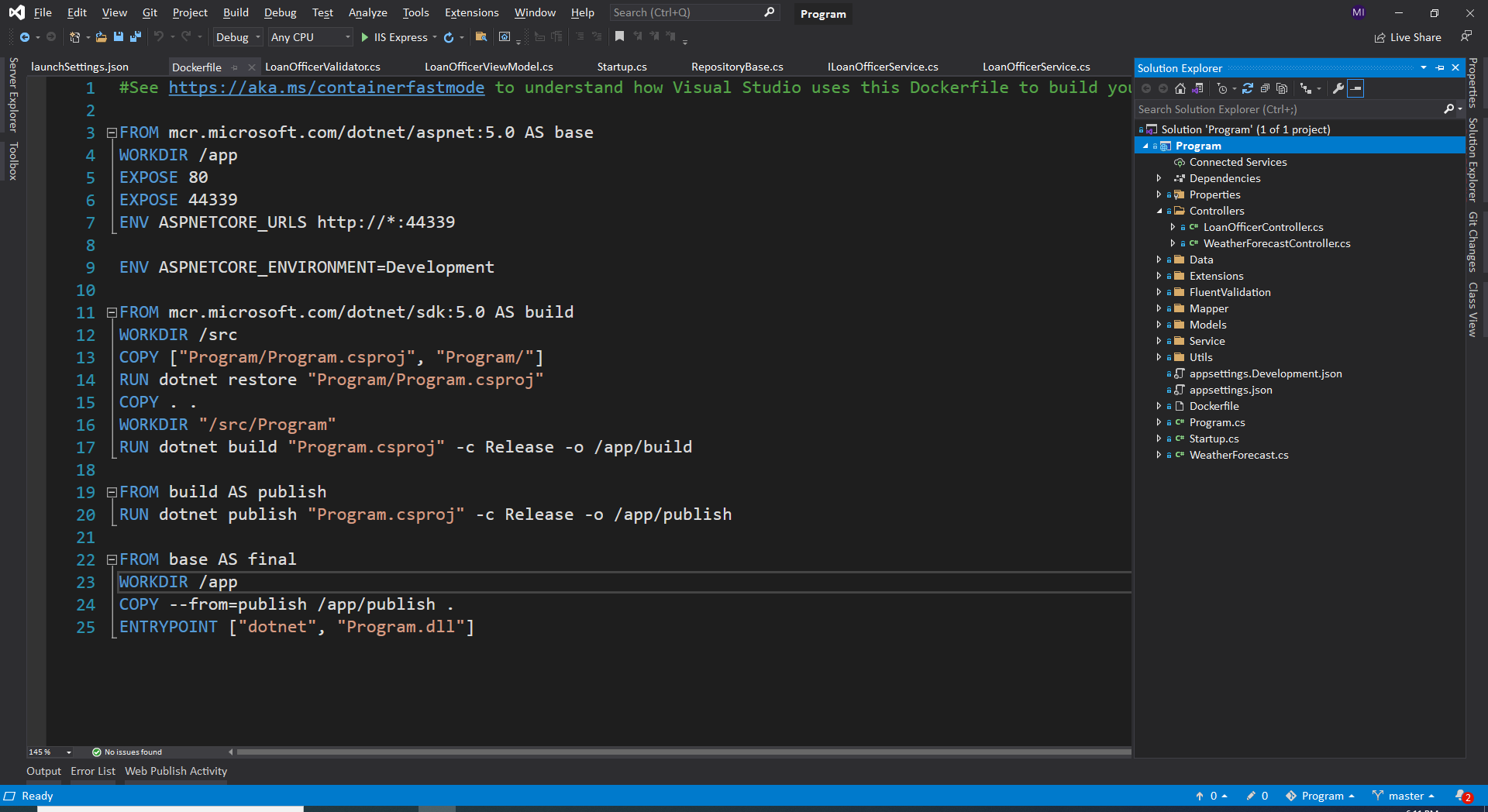
|  |  |  |
| --- | --- | --- |
| **Component** | **Ports (tcp)** | **Used By** |
| API Server | 6443 | All |
| Etcd | 2379-2380 | APi Server/etcd |
| Scheduler | 10251 | Self |
| Controller Manager | 10252 | Self |
| Kubelet | 10250 | Control Plane |
| Node Port | 30000-32767 | All |

# Step 1: Build and Export Docker Image

Please clone the application from above link and open on Visual Studio 2019. I have already configured dockerfile to build the docker image. We are going to use docker hub for our cloud registry. Please create an account in docker hub <https://hub.docker.com/>.

I have created an API with CRUD functionality and the application has been configured with InMemory Database. We are now going to build and export the application as docker image.

1. Open the application and check the Dockerfile.



If you are using your own application instead of clone, please create a dockerfile with no extension and copy and paste below code.

FROM mcr.microsoft.com/dotnet/aspnet:5.0 AS base

WORKDIR /app

EXPOSE 80

EXPOSE 44339

ENV ASPNETCORE\_URLS http://\*:44339

ENV ASPNETCORE\_ENVIRONMENT=Development

FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build

WORKDIR /src

COPY ["Program/Program.csproj", "Program/"]

RUN dotnet restore "Program/Program.csproj"

COPY . .

WORKDIR "/src/Program"

RUN dotnet build "Program.csproj" -c Release -o /app/build

FROM build AS publish

RUN dotnet publish "Program.csproj" -c Release -o /app/publish

FROM base AS final

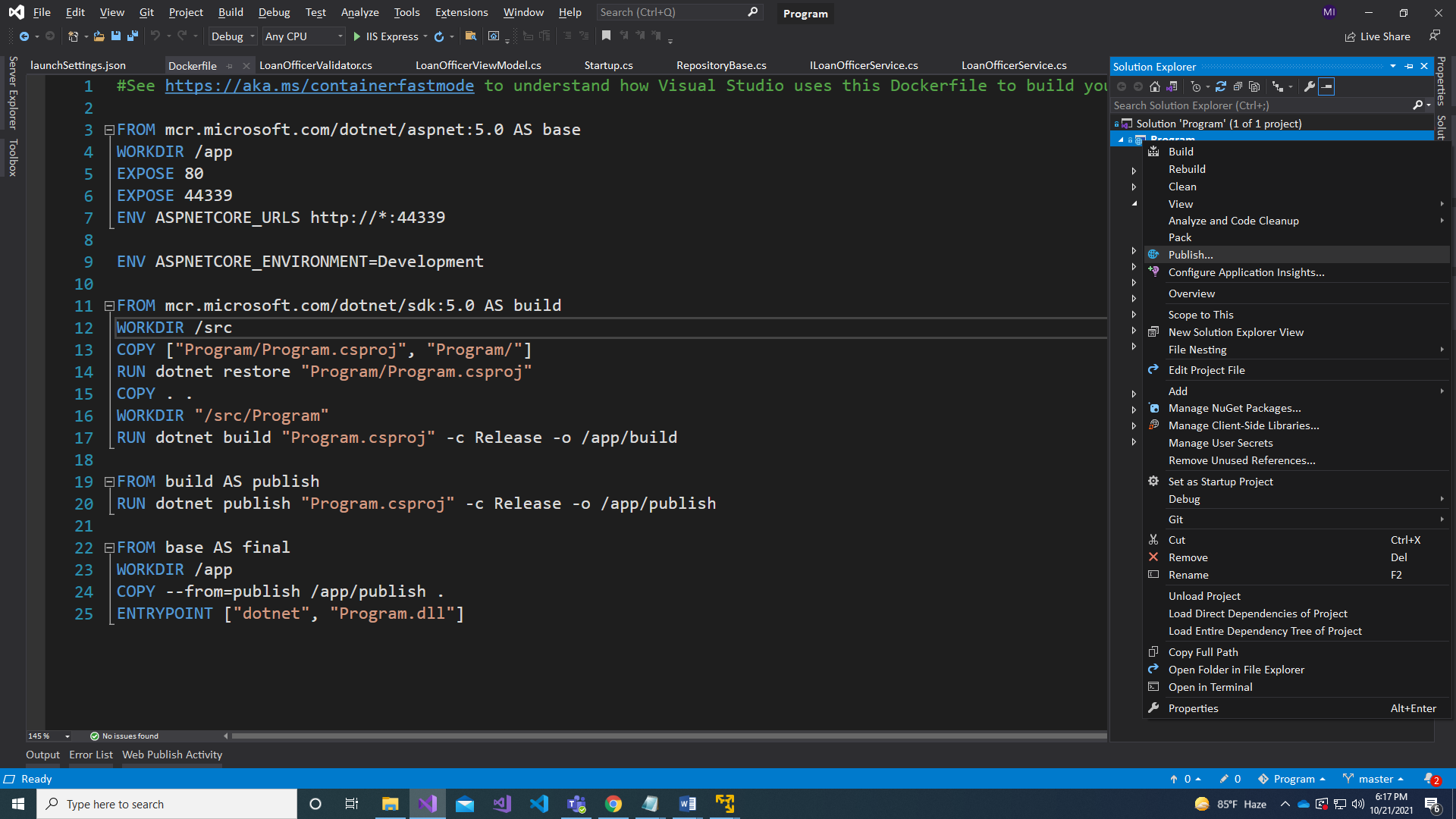
WORKDIR /app

COPY --from=publish /app/publish .

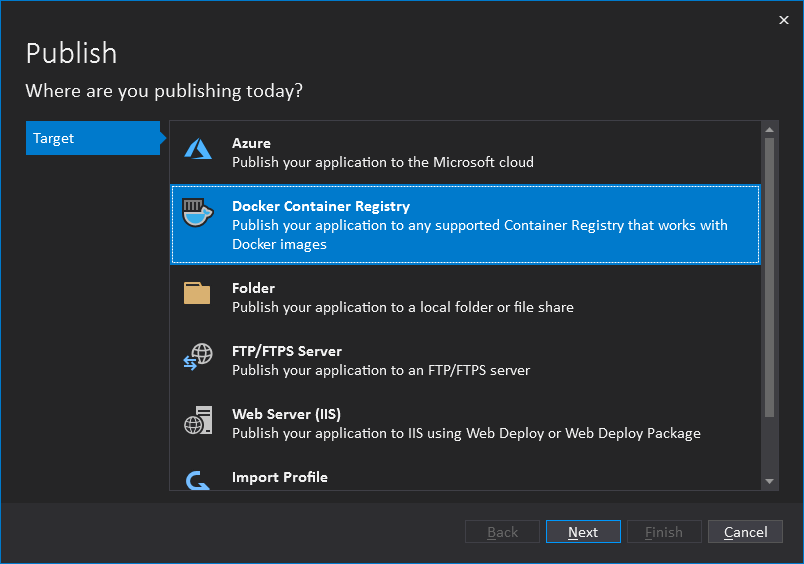
ENTRYPOINT ["dotnet", "Program.dll"]

You need to expose with your application port number instead of 44339. I have highlighted above line.

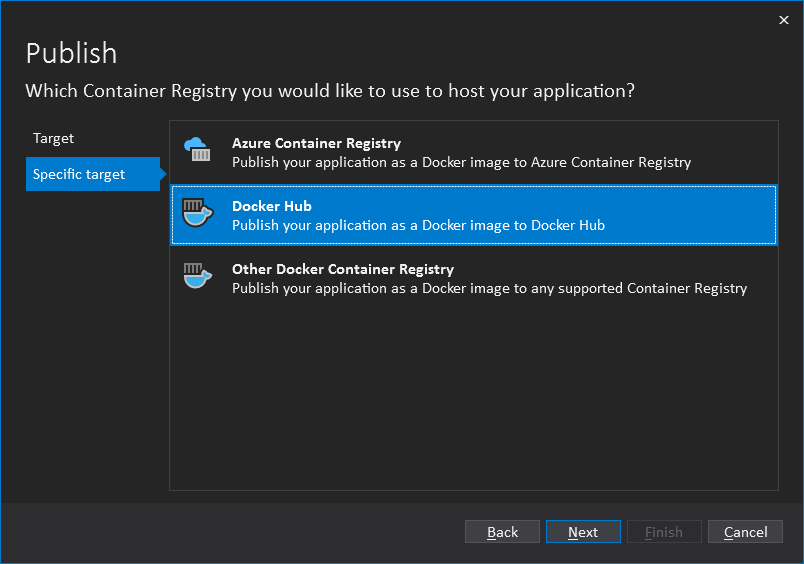
1. Click right button on your project and select publish.



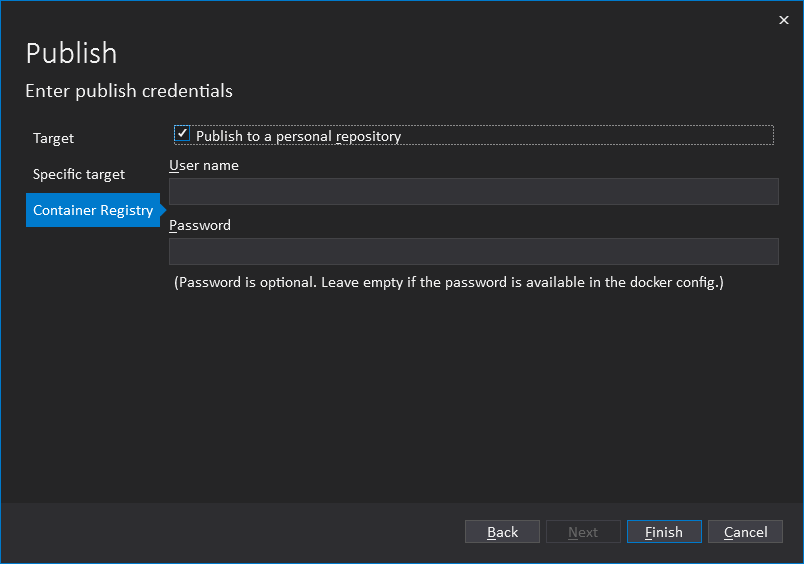
1. Select Docker Container Registry and Next.



1. Select Docker Hub and Next

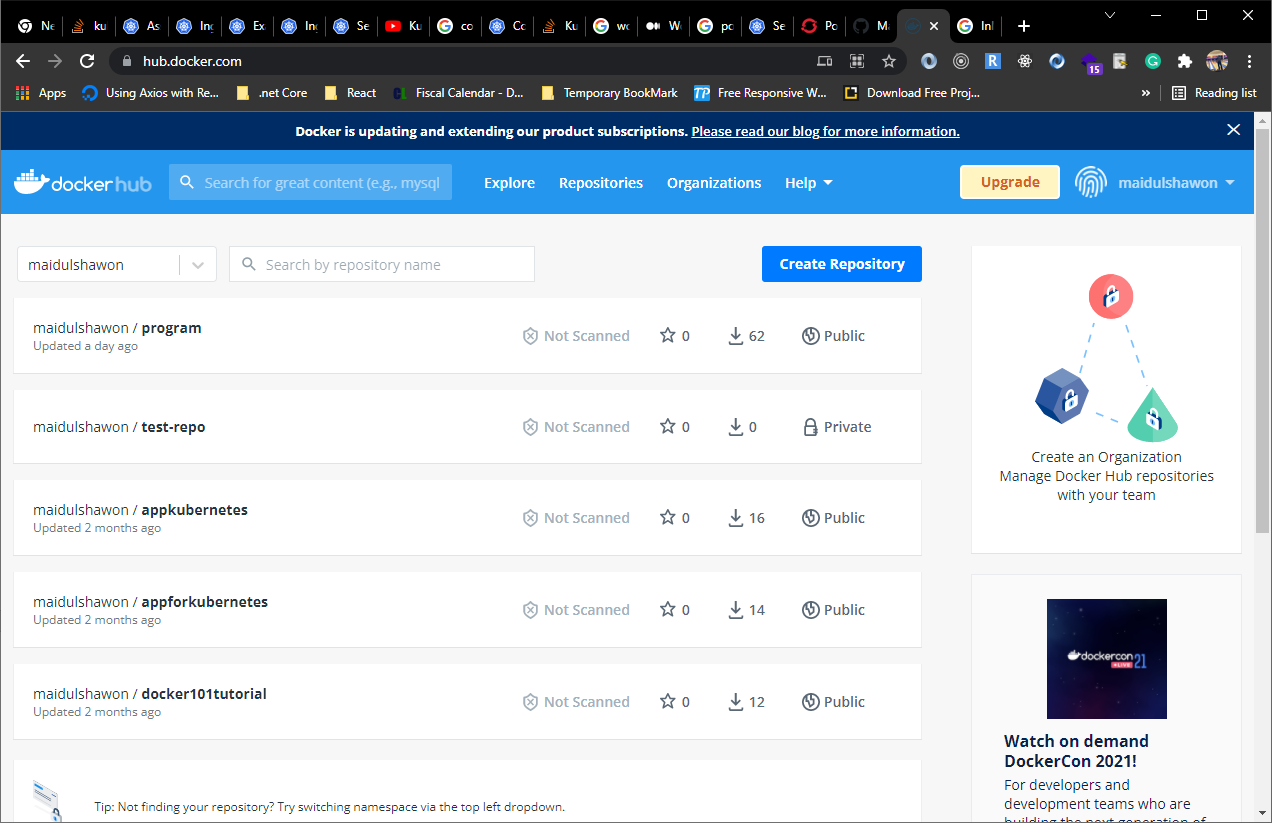


1. Provide your docker hub username and password and click .



It will take a while to build and export the image. Wait till the process completed. You will get a success message after completed the process.

1. Go to your docker hub account and you will find the image in repository list.



We are going to use above image in our cluster.

# Step 2: Pod Deployment with ReplicaSet

On my other document, I have shown to deploy a pod to a target node with NodeSelector.

NodeSelector is a good and simplest option to deploy pods but this process has some limitation. If selected node goes down or crash, pod will stop working and application will not be able to run.

To avoid that situation, we are going to approach with Node affinity.

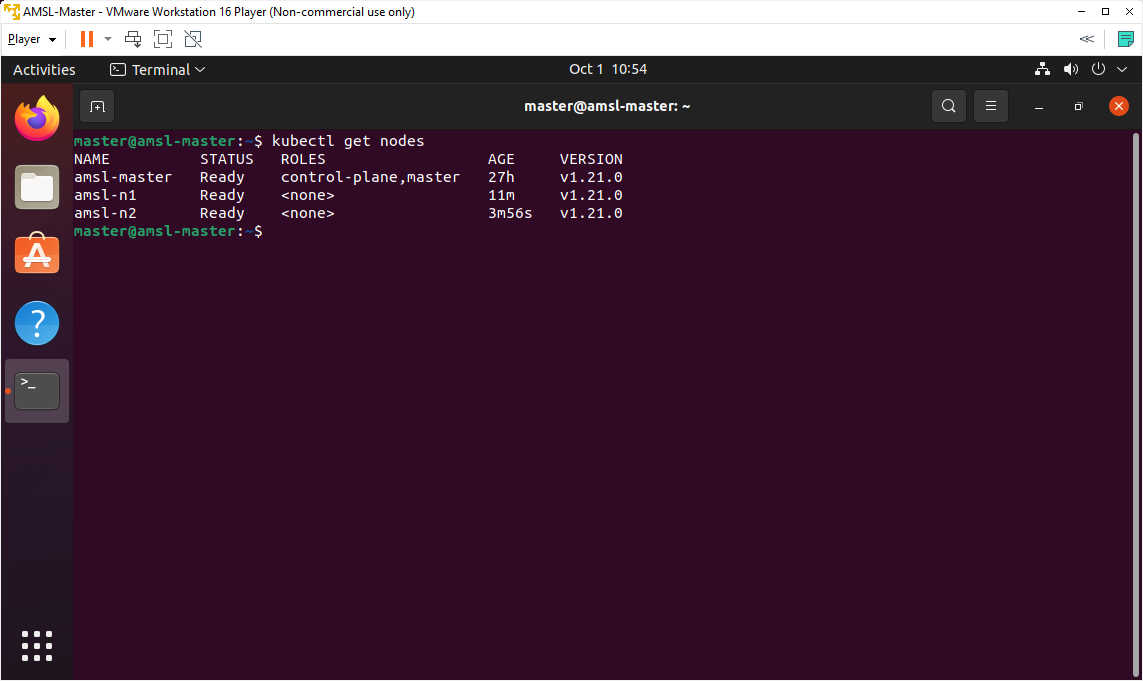
Node affinity is conceptually similar to nodeSelector -- it allows you to constrain which nodes your pod is eligible to be scheduled on, based on labels on the node.

There are currently two types of node affinity, called requiredDuringSchedulingIgnoredDuringExecution and preferredDuringSchedulingIgnoredDuringExecution. You can think of them as "hard" and "soft" respectively, in the sense that the former specifies rules that must be met for a pod to be scheduled onto a node (similar to nodeSelector but using a more expressive syntax), while the latter specifies preferences that the scheduler will try to enforce but will not guarantee. The "IgnoredDuringExecution" part of the names means that, similar to how nodeSelector works, if labels on a node change at runtime such that the affinity rules on a pod are no longer met, the pod continues to run on the node. In the future we plan to offer requiredDuringSchedulingRequiredDuringExecution which will be identical to requiredDuringSchedulingIgnoredDuringExecution except that it will evict pods from nodes that cease to satisfy the pods' node affinity requirements.

Before we proceed to deployment process, we need to check current status of cluster. Login to you Master Node.

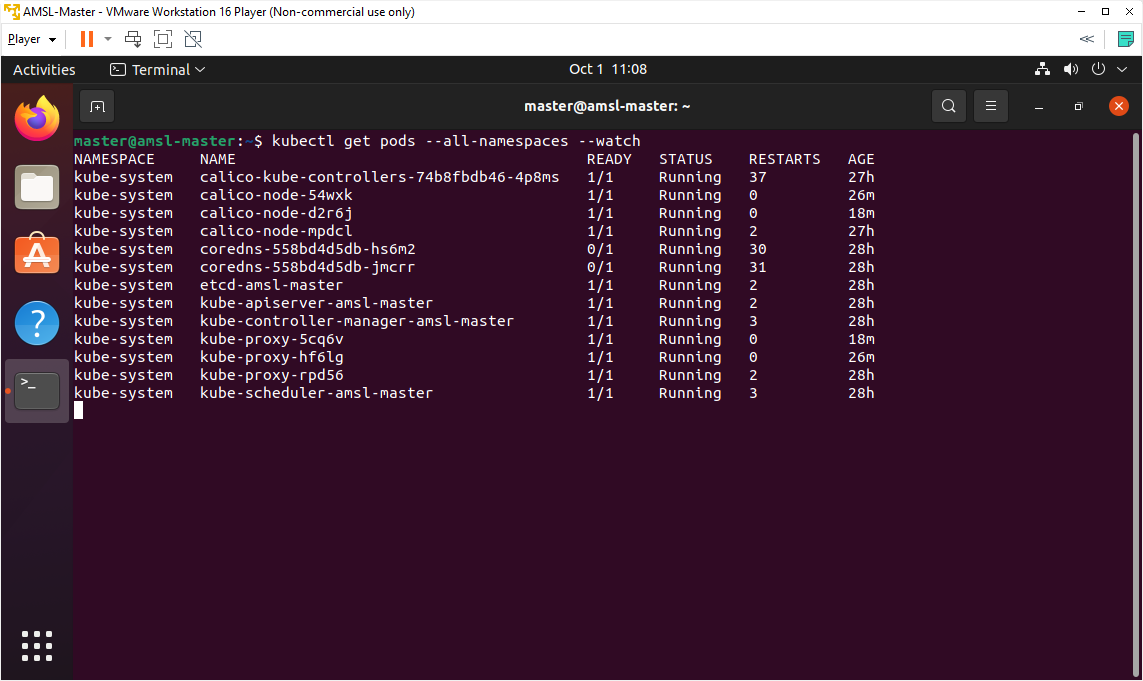
1. Write below command to verify all nodes are working properly.

kubectl get nodes



1. On the Control Plane Node, watch for the calico pod and the kube-proxy to change to Running on the newly added nodes by below command.

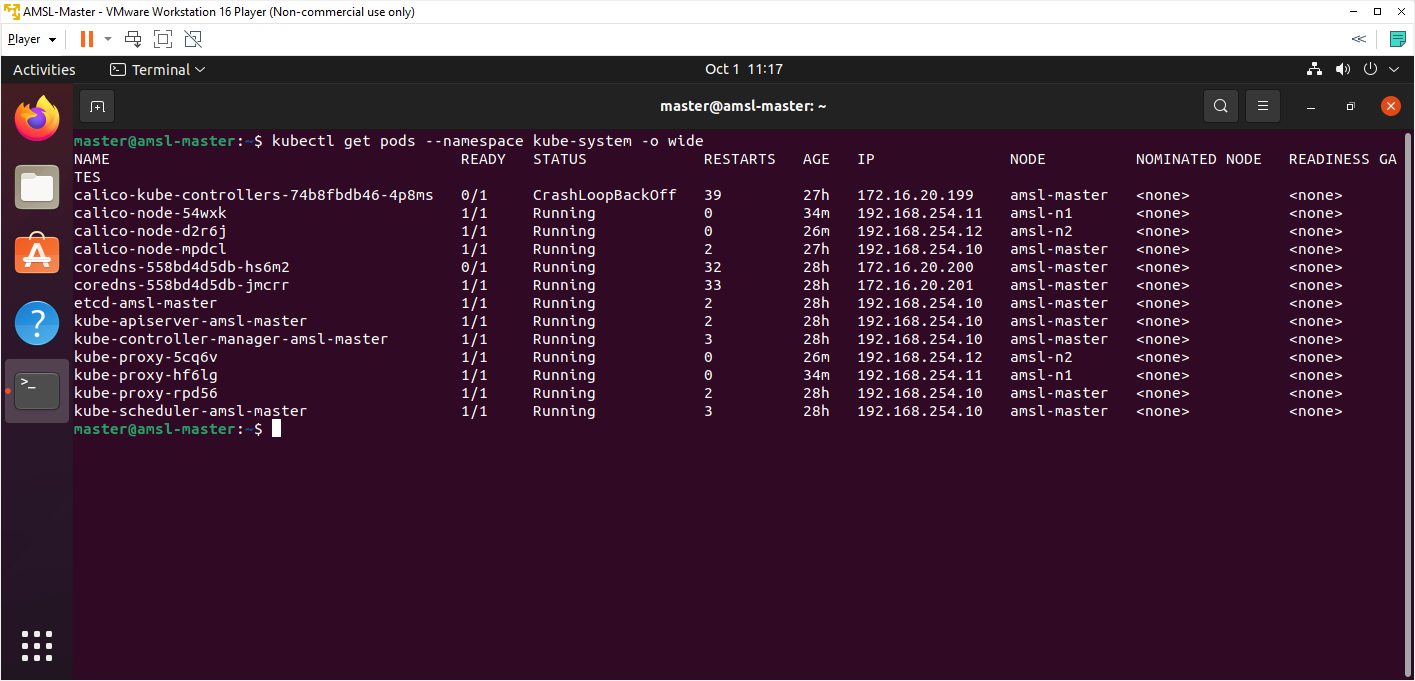
kubectl get pods --all-namespaces --watch



On the above output is showing all system pods are running.

1. Let's check system pods with extra information. A namespace is a way to group resources together.

kubectl get pods --namespace kube-system -o wide



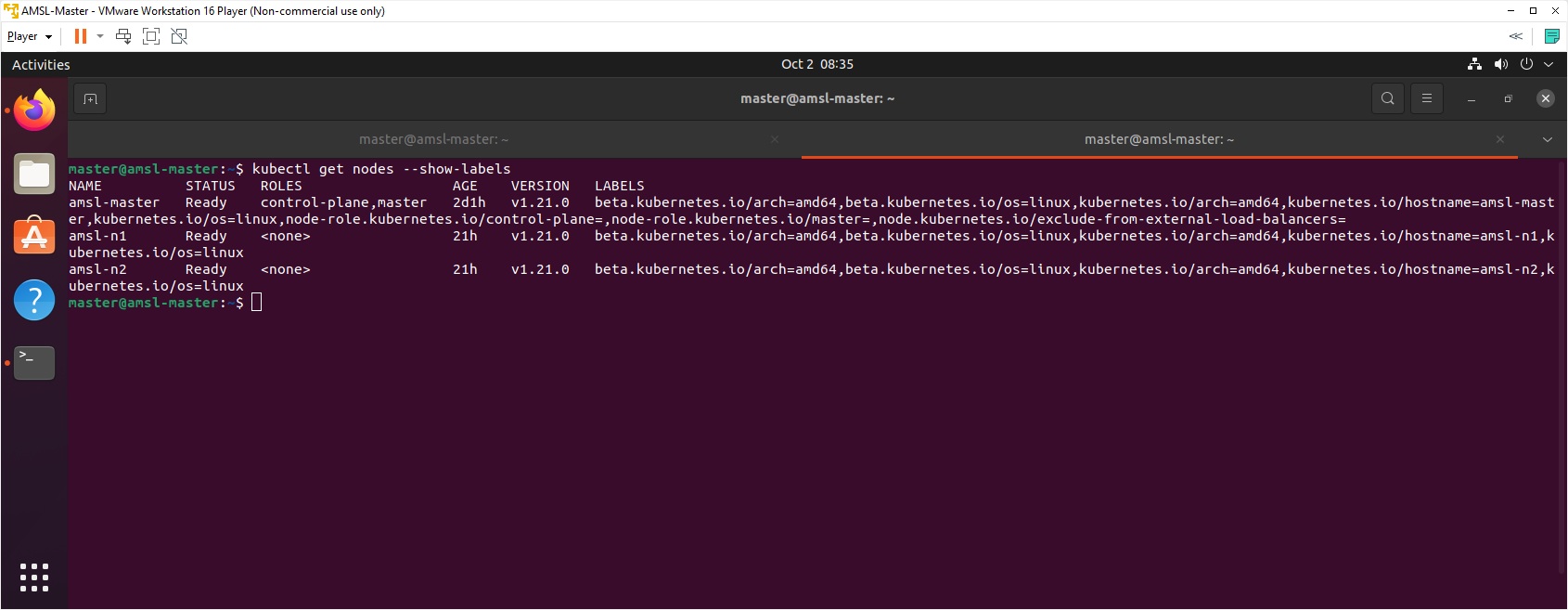
As we can see everything is running without error, we can proceed to next step.

# Step 3: Assign Label for Nodes

Now we are going to assign a label for Worker Nodes. We will deploy pods based on label name and value.

1. Check current label of nodes with below command.

kubectl get nodes --show-labels



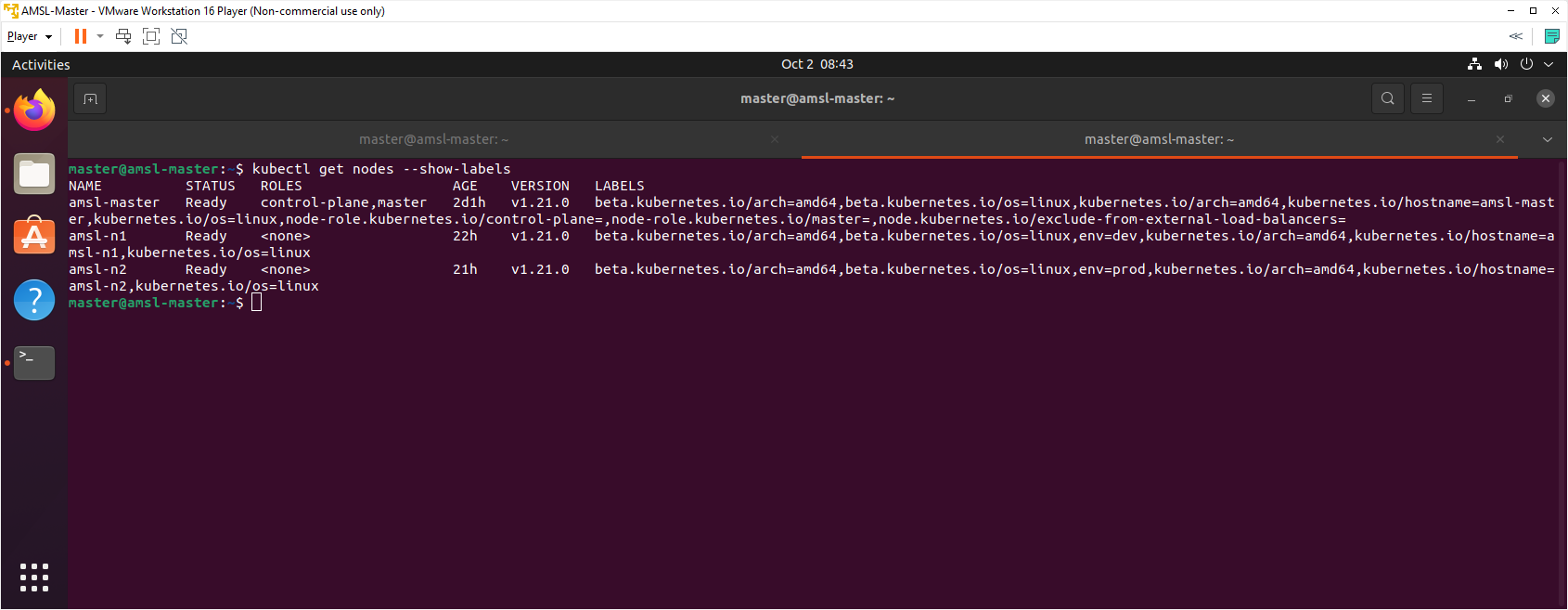
1. Now we are going to set custom label for nodes. I have decided to set the label for amsl-n1 will be env=dev and amsl-n2 will be env=prod. Use below commands to set labels for nodes.

kubectl label nodes amsl-n1 env=dev

kubectl label nodes amsl-n2 env=prod

1. We can recheck label name again with below command.

kubectl get nodes --show-labels



# Step 4: Create and Deploy Pod with Node affinity

Now we are going to deploy pod declaratively with YAML file.

1. We are going to use dry-run=client to create a yaml manifest file for deployment with a container image. We could write the yaml by hand, but we can use dry-run=client to build it for us. Write below command to generate yaml manifest file.

**kubectl create deployment amsl-program \**

**--image=** **docker.io/maidulshawon/program:latest \**

**--dry-run=client -o yaml | more**

1. Let's write this deployment yaml out to file. We are now redirecting previously generated output to deployment.yaml file.

**kubectl create deployment amsl-program \**

**--image= docker.io/maidulshawon/program:latest \**

**--dry-run=client -o yaml > program-deployment.yaml**

1. Edit program-deployment.yaml file to define node affinity with node label. Use below command to edit the file.

sudo nano program-deployment.yaml

1. Update the file with below text.

apiVersion: apps/v1

kind: Deployment

metadata:

labels:

app: amsl-program

name: amsl-program

spec:

replicas: 5

selector:

matchLabels:

app: amsl-program

strategy: {}

template:

metadata:

creationTimestamp: null

labels:

app: amsl-program

spec:

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: env

operator: In

values:

- prod

- dev

containers:

- image: docker.io/maidulshawon/program:latest

name: program

ports:

- containerPort: 44339

resources: {}

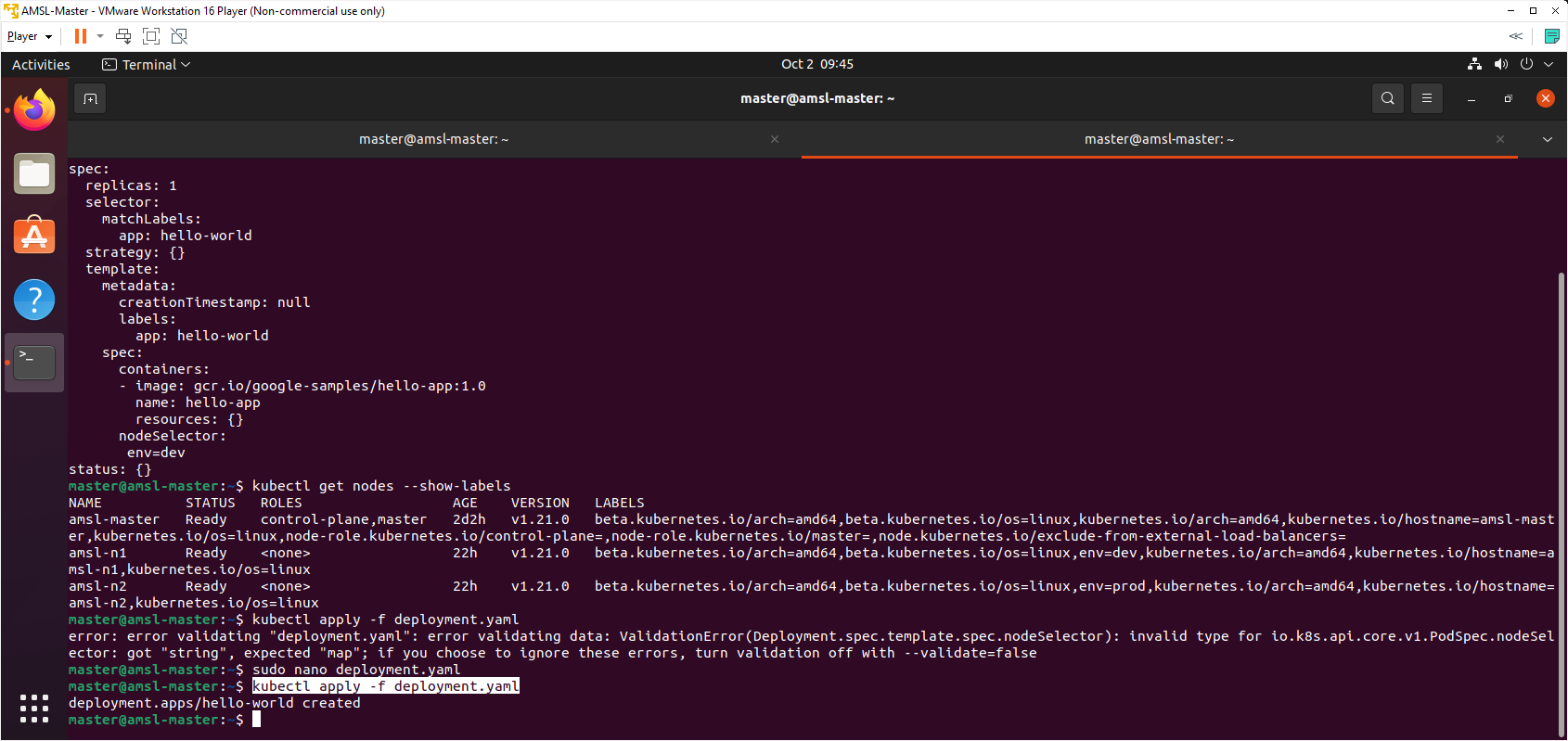
status: {}

On the above YAML file, I have instructed to create 5 replica set of pod and, container port number is same as exposed port of Dockerfile we created earlier in the application. Please replaced the green part of image URL with your image repository of docker hub.

On NodeAffinity part, I have configured as ‘requiredDuringSchedulingIgnoredDuringExecution ‘ and provided both Node label. Control Plane scheduler will configure and deploy pod replica set within defined Node.

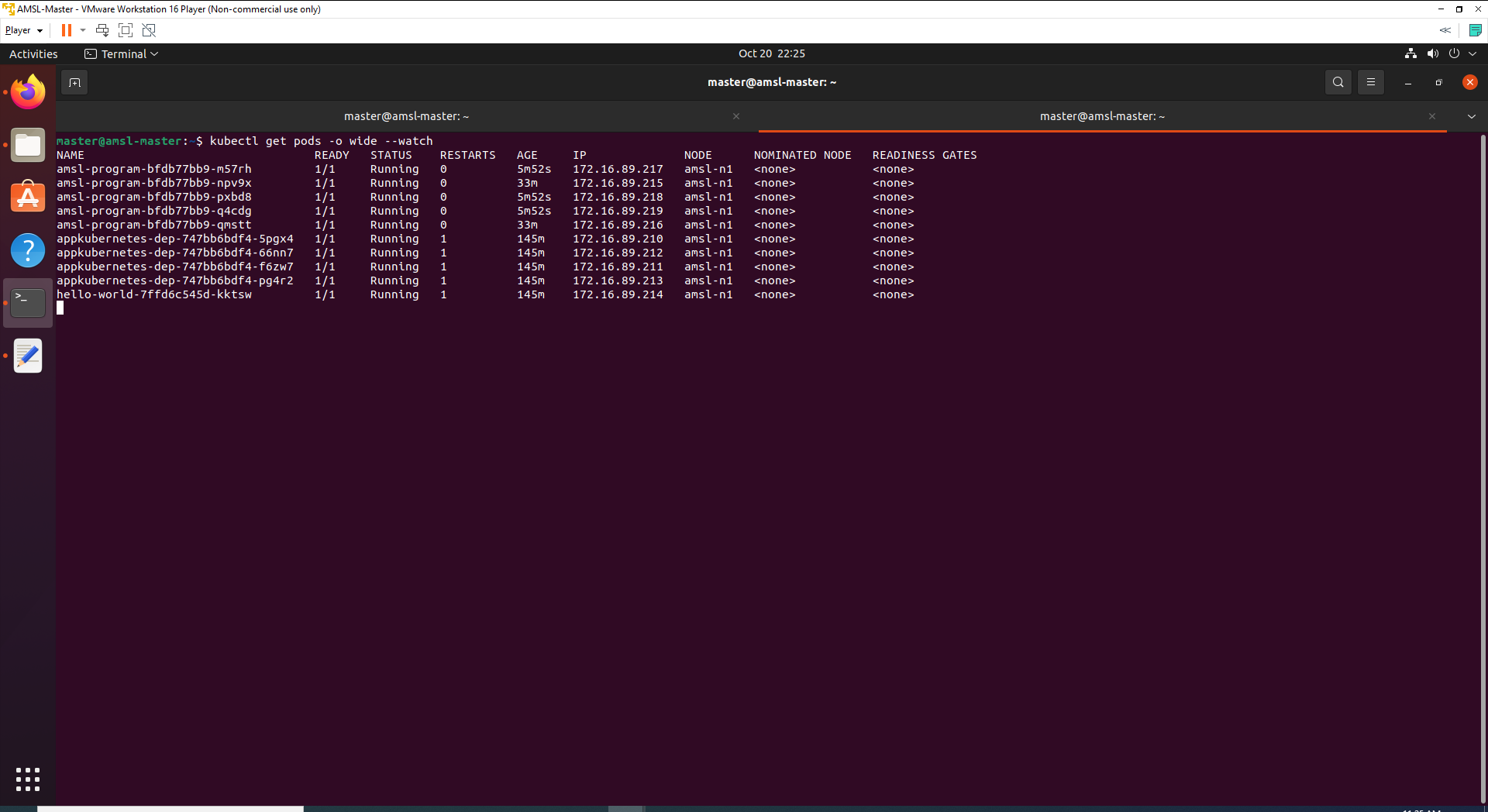
1. Use below command to execute the **program-deployment.yaml** manifest file.

kubectl apply -f program-deployment.yaml



1. Now we are going to check deployed pod with below command.

kubectl get pods -o wide



We can see that all pods are running and 5 pods has been deployed in Node1.

1. Let’s check the deployment with below command.

kubectl get deployment amsl-program



# Step 5: Deploy a Service with Load Balancer

1. Now we need to expose these set of pod. Generate the yaml file for the service with below command.

**kubectl expose deployment amsl-program \**

**--port=80 --target-port=44339 \**

**--dry-run=client -o yaml | more**

1. Write the service yaml manifest to a file.

**kubectl expose deployment hello-world \**

**--port=80 --target-port=** **44339 \**

**--dry-run=client -o yaml > program-service.yaml**

1. Edit program-service.yaml file. Use below command to edit the file.

sudo nano program-service.yaml

1. Update the file with below text.

apiVersion: v1

kind: Service

metadata:

labels:

app: amsl-program

name: amsl-program

spec:

type: LoadBalancer

externalIPs:

- 192.168.0.10

ports:

- port: 80

protocol: TCP

targetPort: 44339

selector:

app: amsl-program

status:

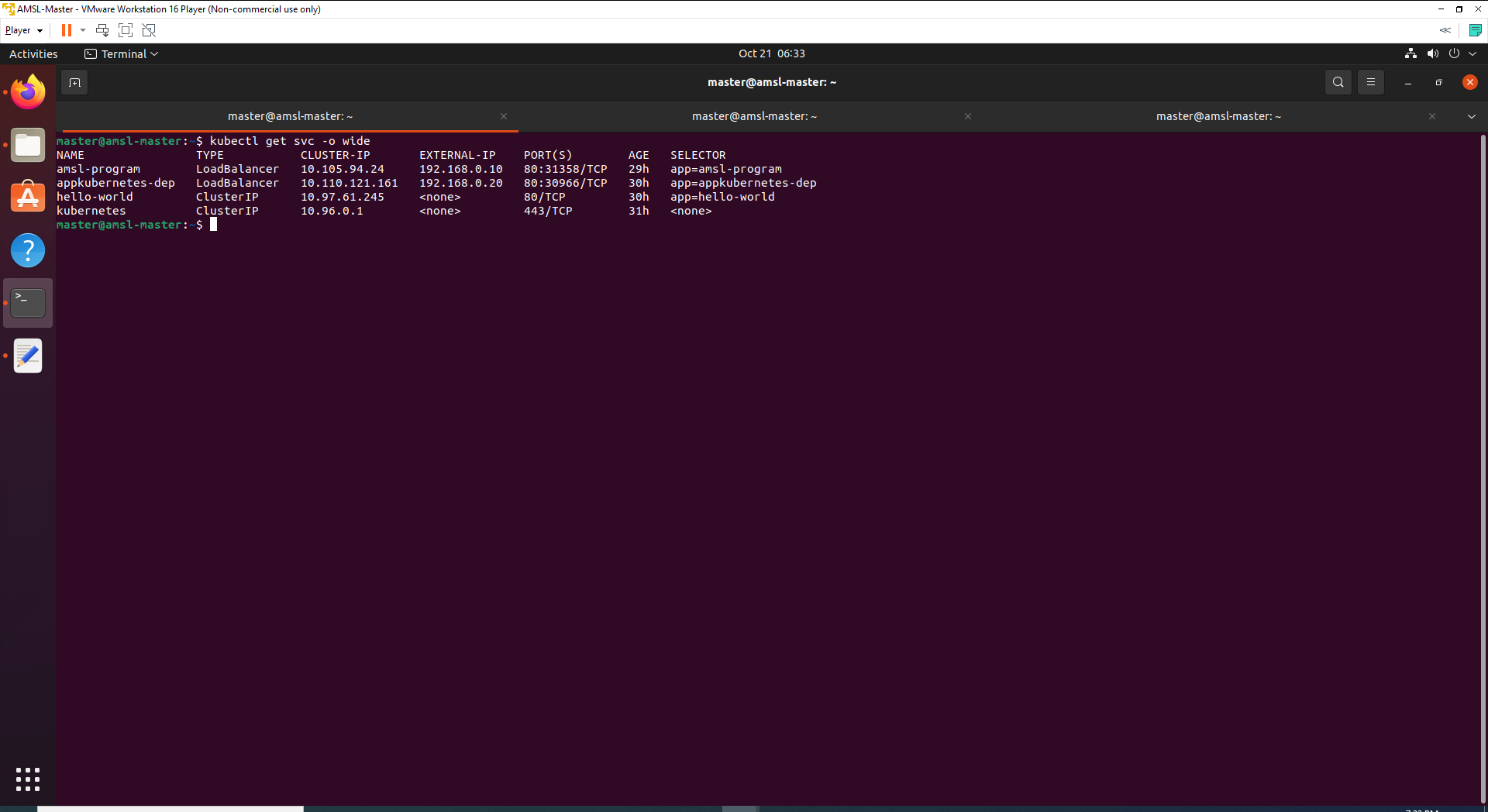
loadBalancer: {}

on the above file, we have defined the type as LoadBalancer and also definea external IP to access the application.

1. Use below command to execute the service yaml file declaratively.

kubectl apply -f program-service.yaml

1. Check the running services with below command.



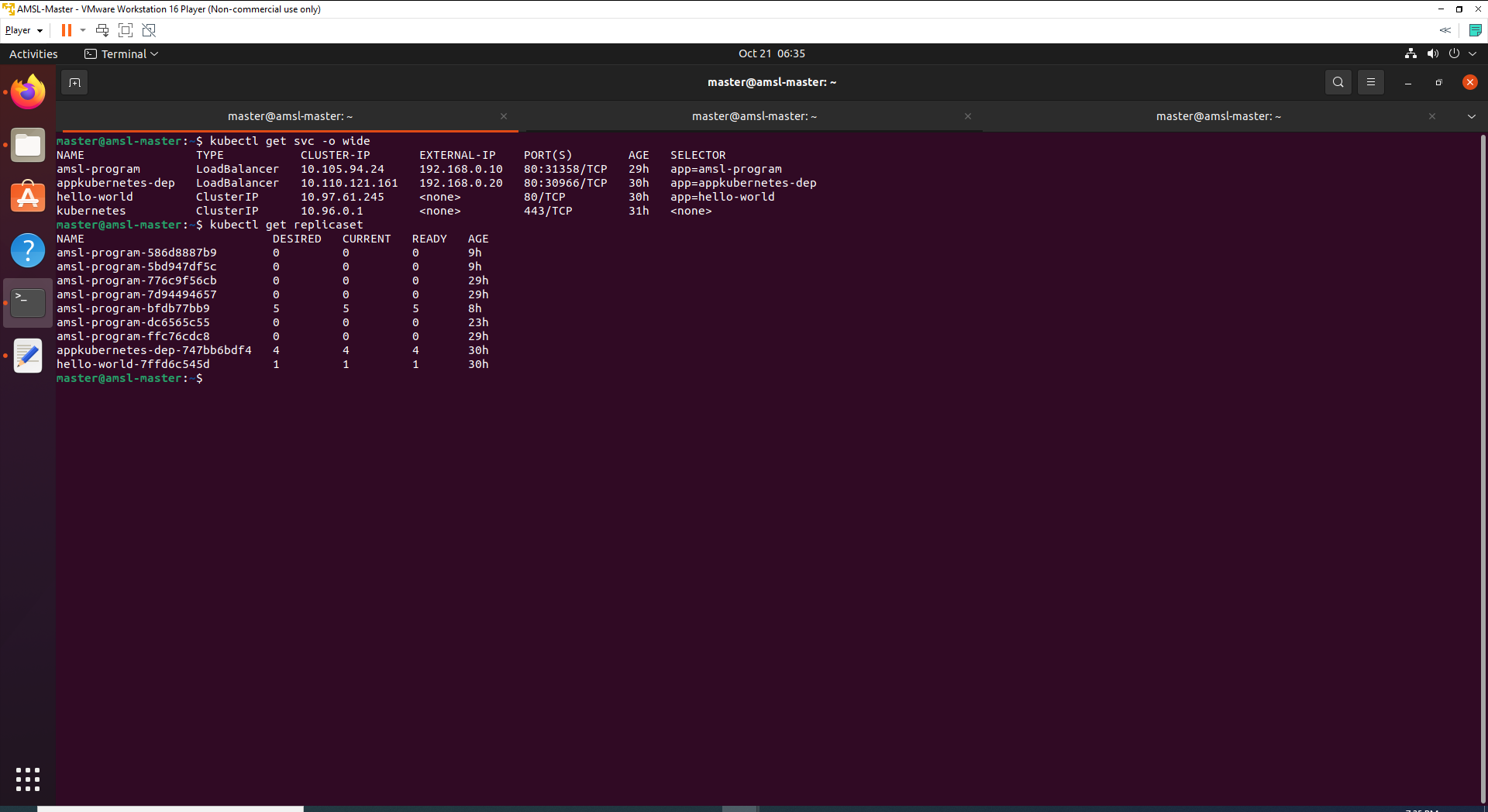
# Step 6: Verification of Deployments

1. Check out our current state, Deployment, ReplicaSet, Pod and a Service.

kubectl get all -o wide

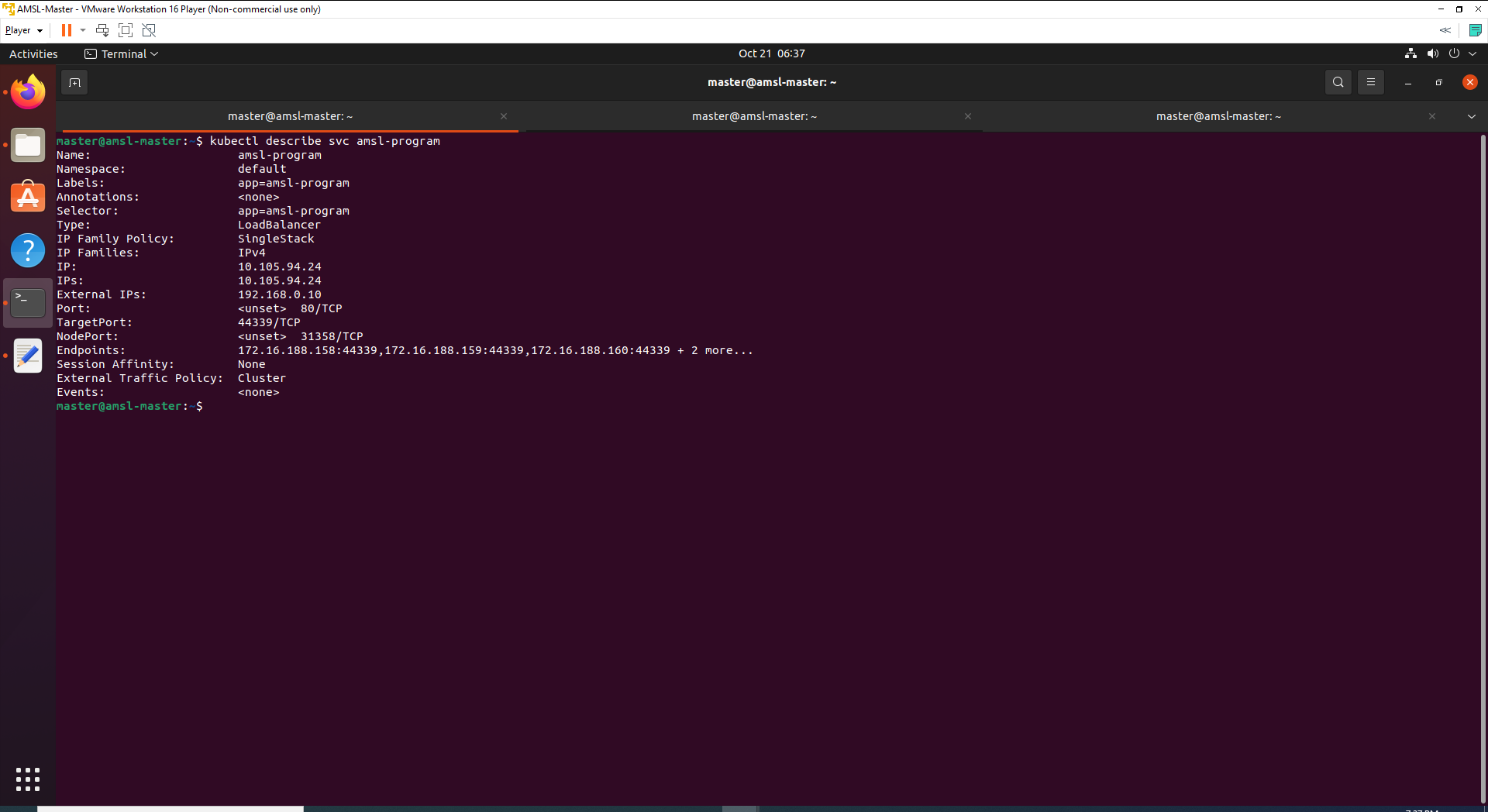
1. Deployments are made of ReplicaSets and ReplicaSets create Pods!

kubectl get replicaset



1. We can also get the detail information of service with below command.

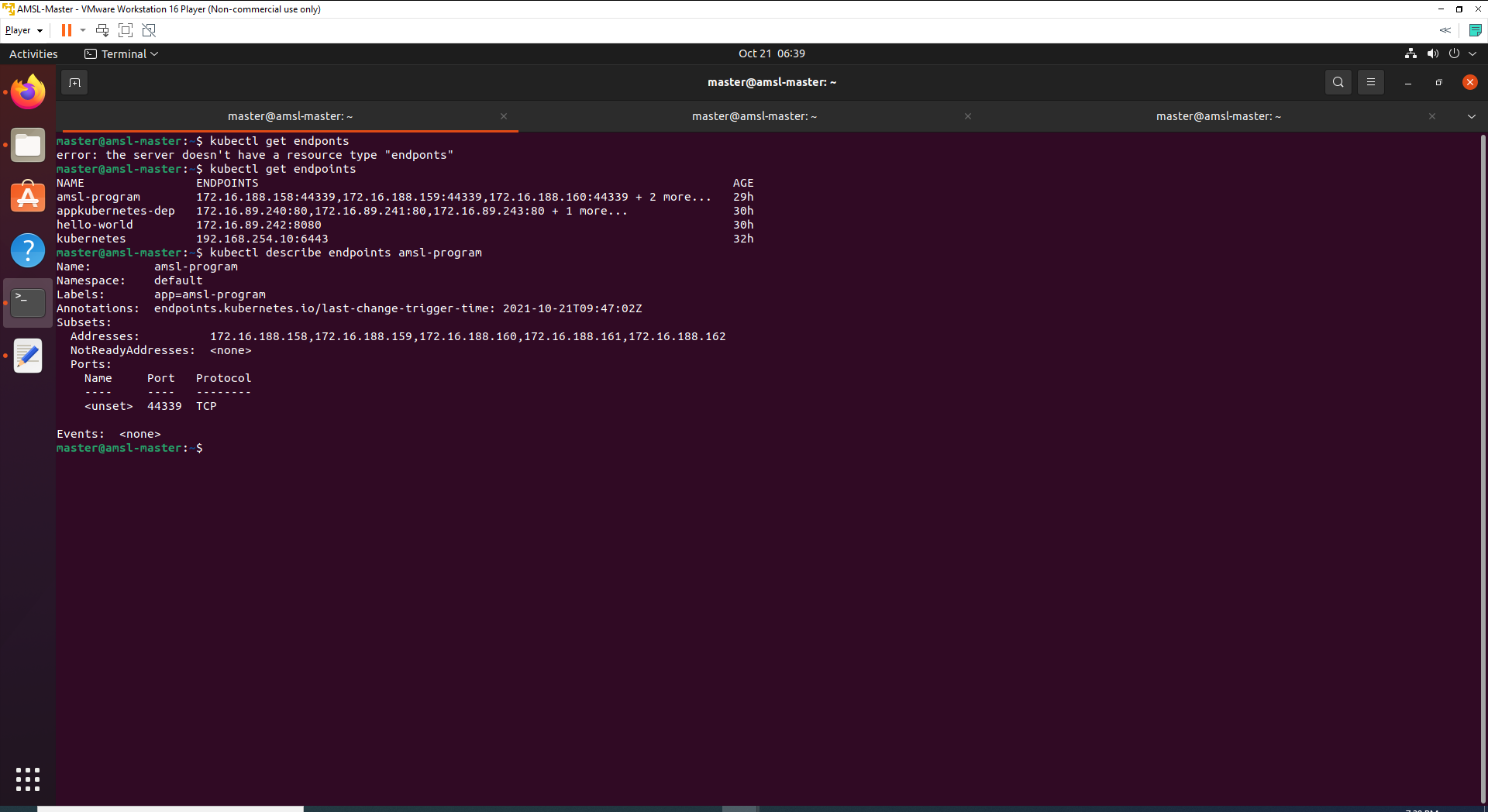
kubectl describe service hello-world



1. Use below command to check all running endpoints and full detail for specific endpoint.

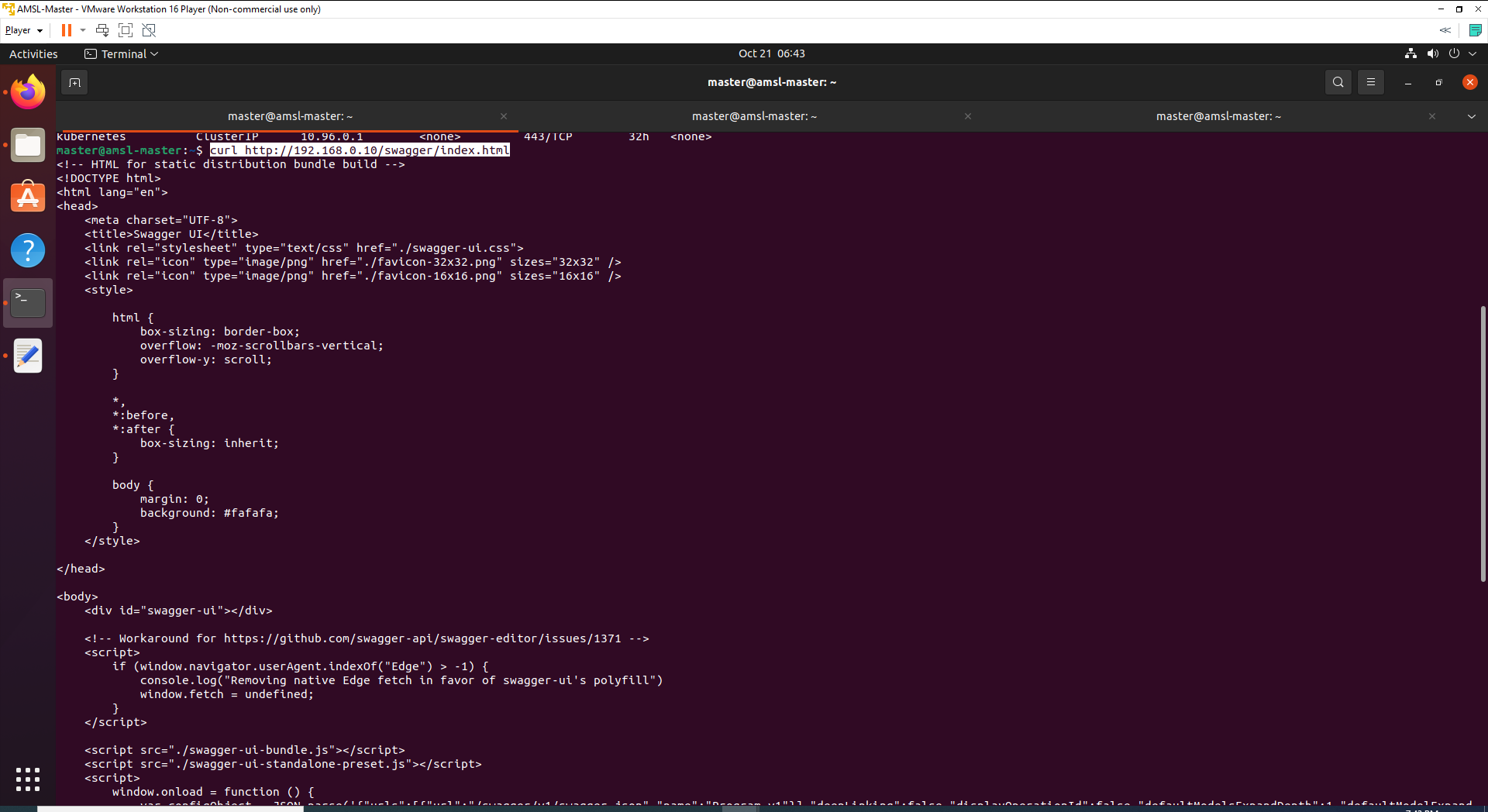
kubectl get endpoints

kubectl describe endpoints amsl-program



1. Endpoints are IP:Port pairs for each of Pods that that are a member of the Service. Check with external IP address.

curl http://192.168.0.10/swagger/index.html



# Step 7: Verify with Dashboard UI

1. Log back to Dashboard UI to verify the current state. Login procedure details are available on the other document (***Kubernetes Installation and Configuration with multi-node cluster using containerd.docx***).
2. After complete the configuration of dashboard UI, we need to find token we can use to log in. Execute following command:

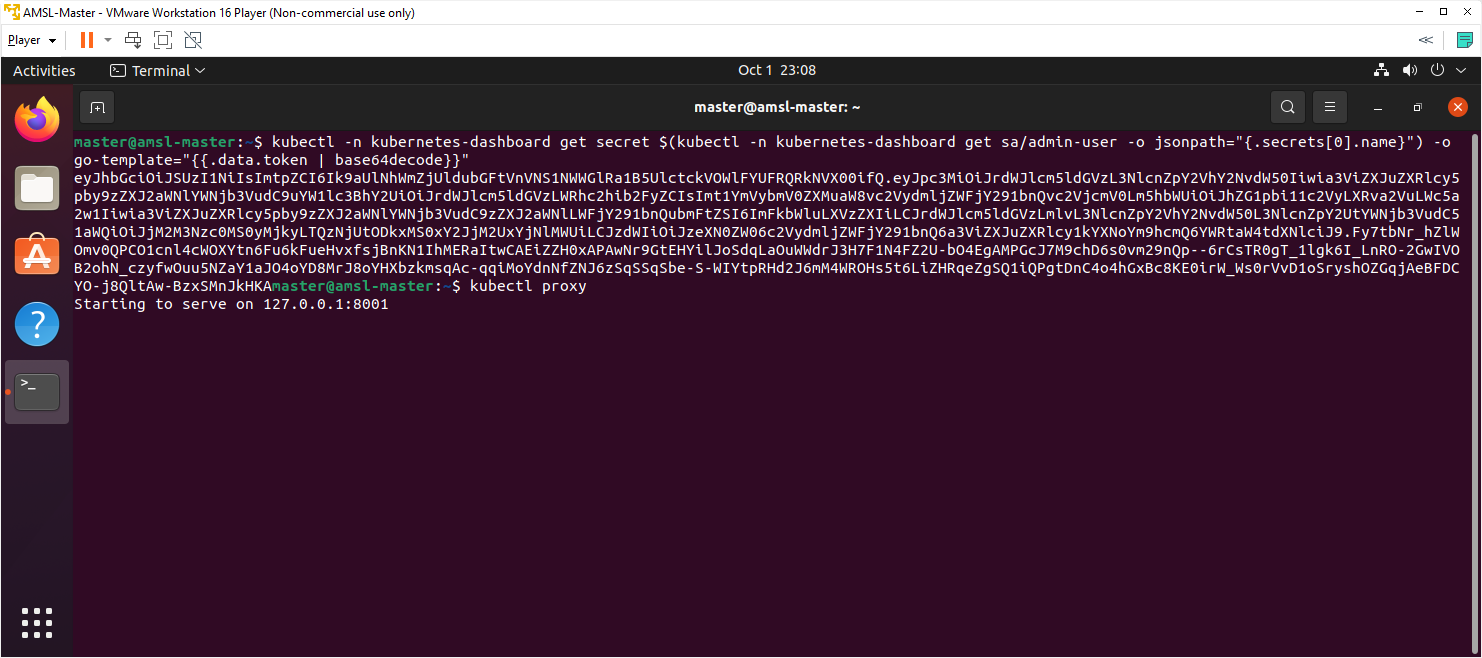
kubectl -n kubernetes-dashboard get secret $(kubectl -n kubernetes-dashboard get sa/admin-user -o jsonpath="{.secrets[0].name}") -o go-template="{{.data.token | base64decode}}"



From the above output we can see a new token has been generated, we can copy the above output token to login into dashboard.

1. Use following command to enable the access to the Dashboard using the kubectl command-line tool.

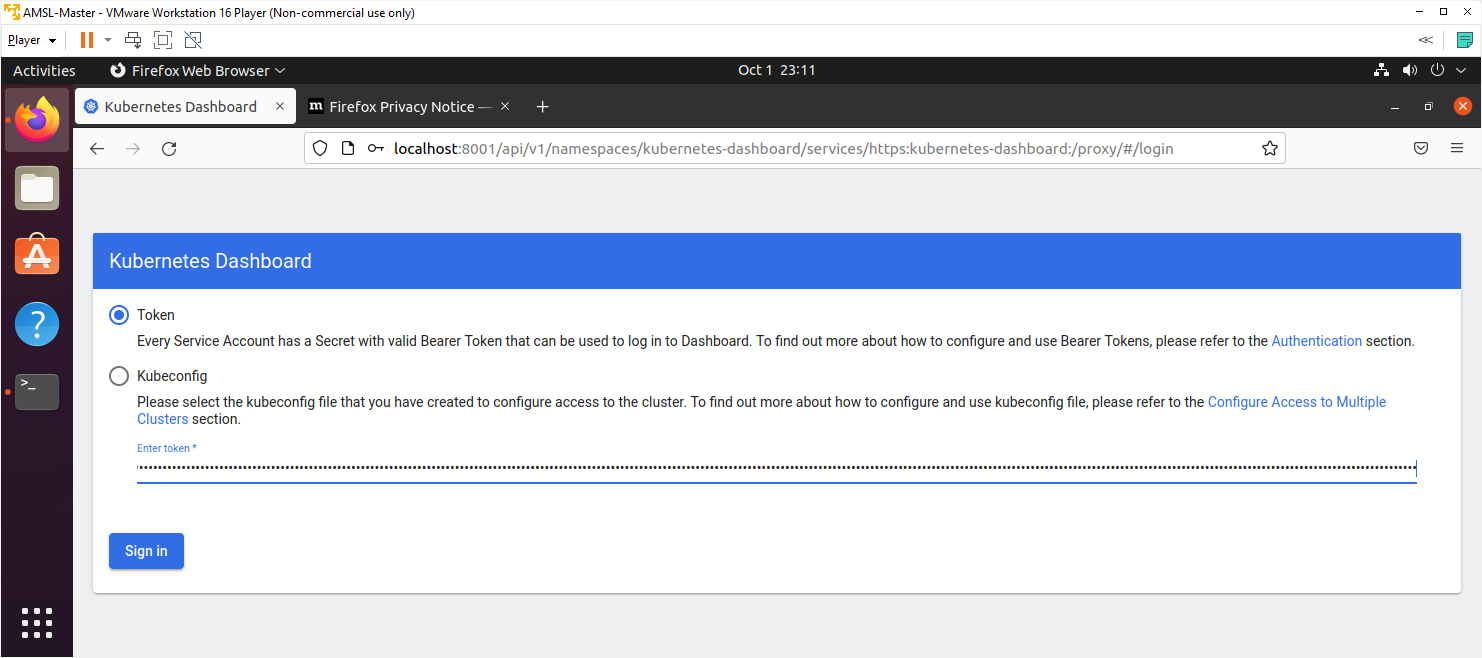
kubectl proxy

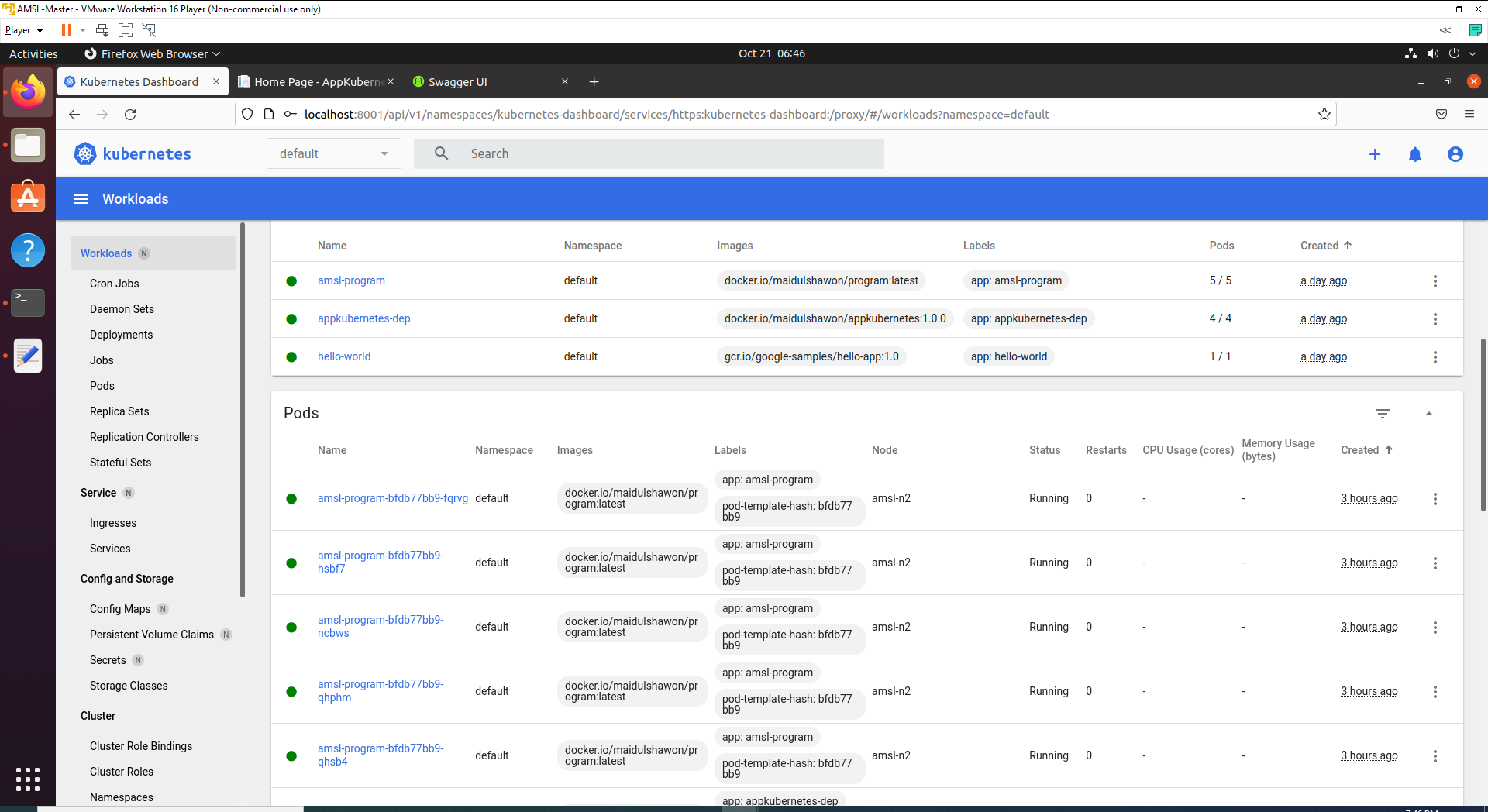


1. Use below URL to access the dashboard.

<http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/>

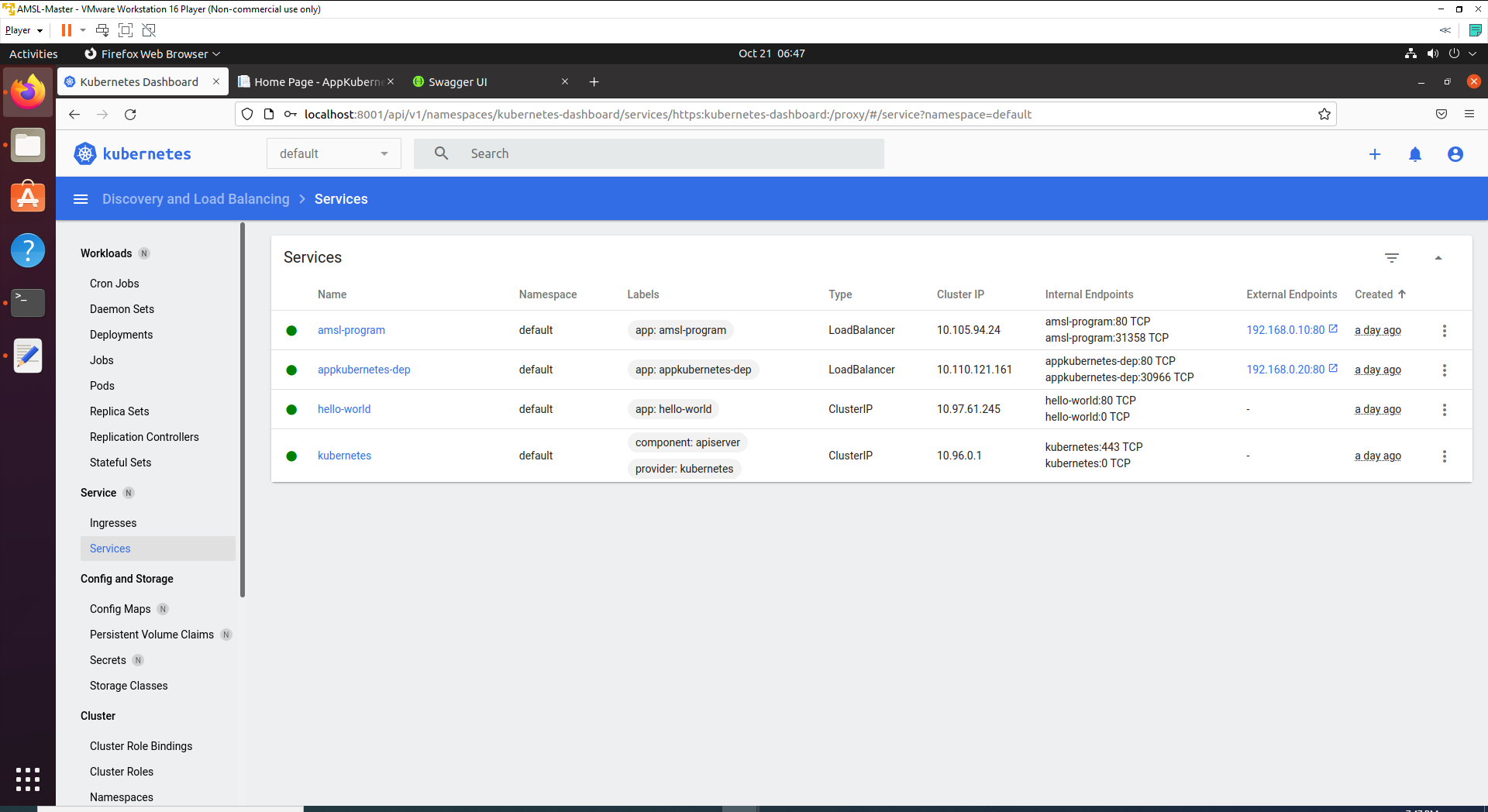
1. Use above created token and click sign in button.





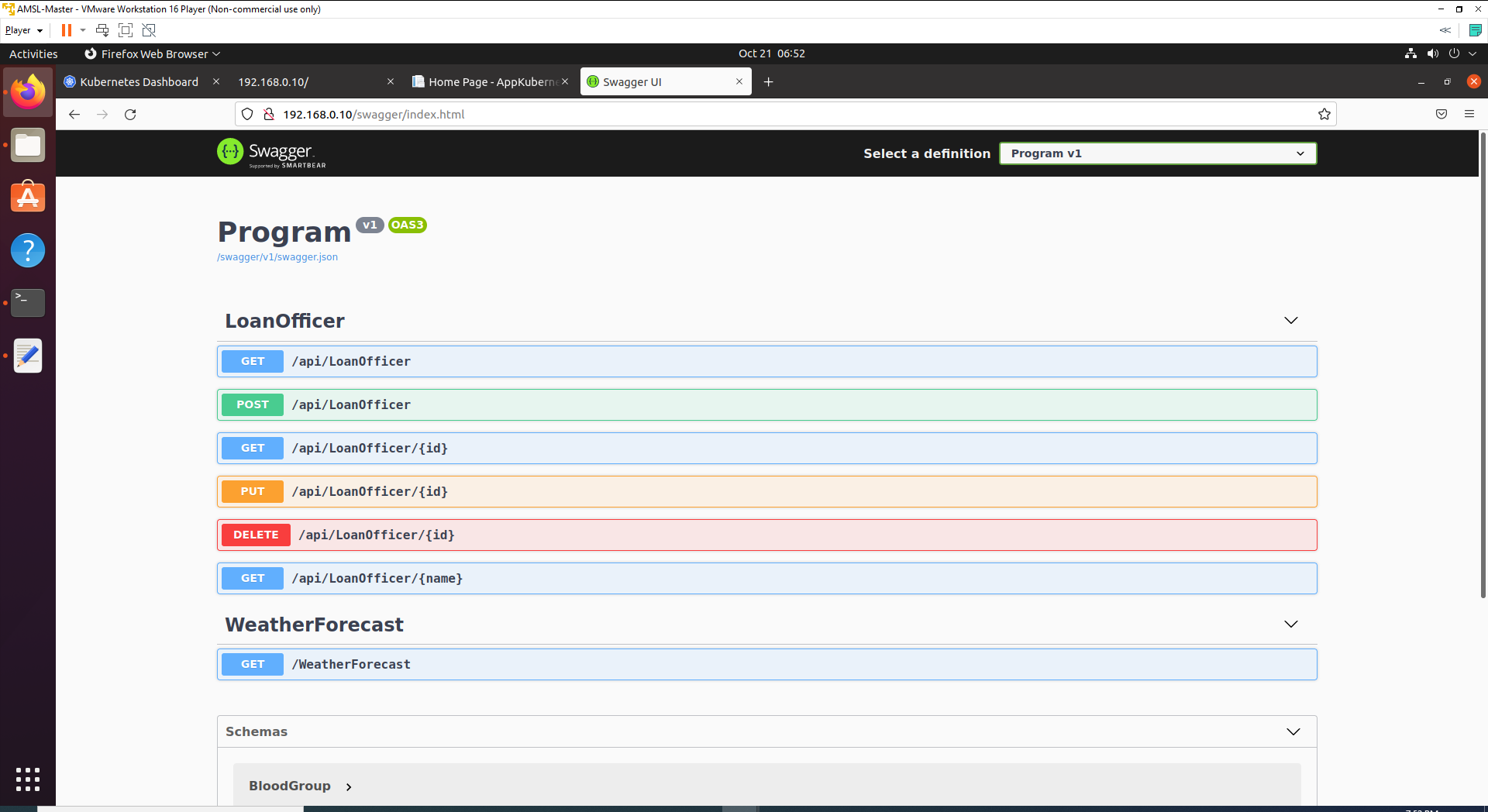
As we can see that all deployments, services and pods are running without error.

1. Now go to service tab from left nav.



You can see that external endpoint is displaying on the service list. Click on the endpoints port or visit <http://192.168.0.10/swagger/index.html>.

You will be able to see a working API like below



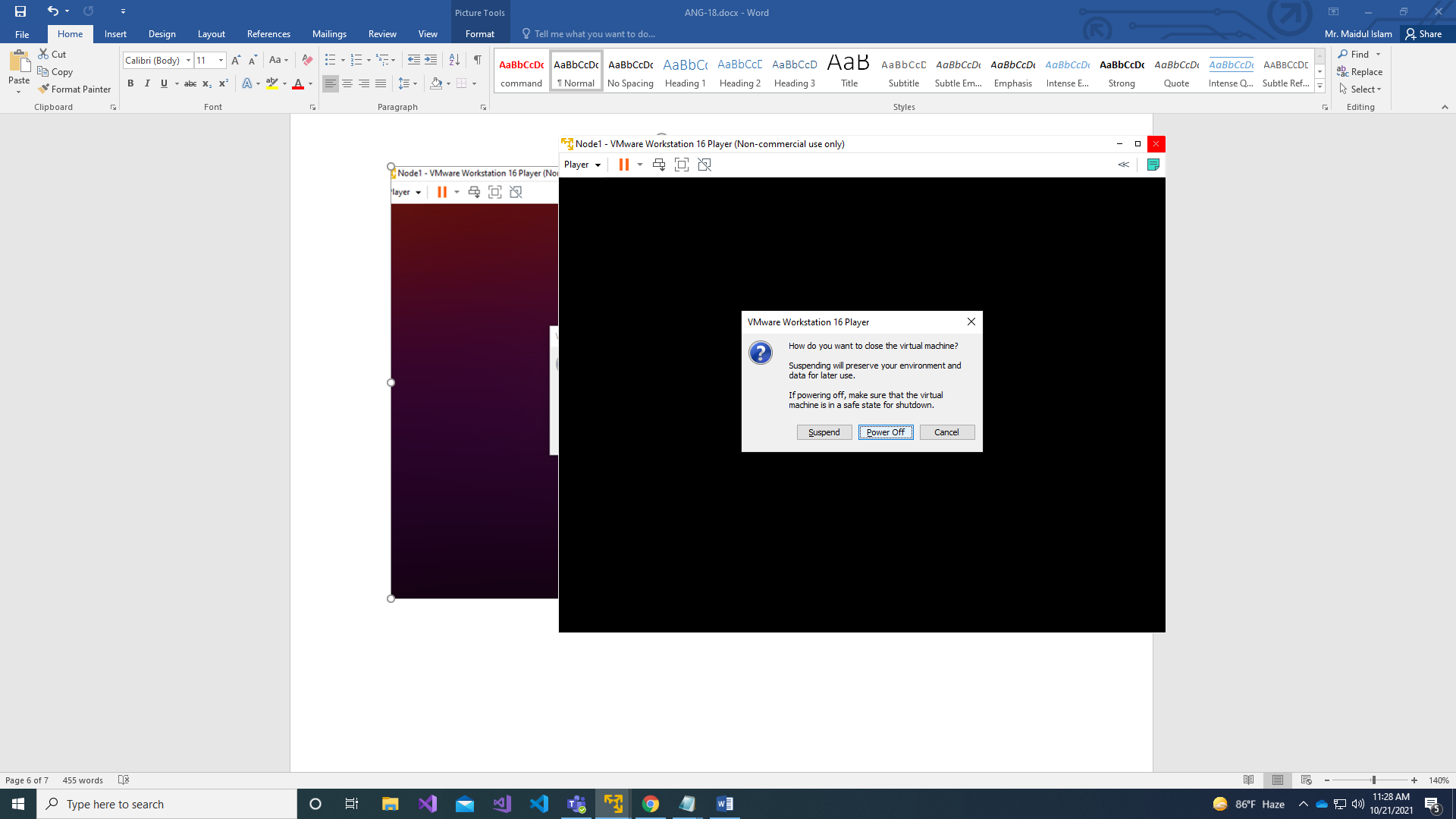
As we can see that our API is working with External IP. As we have seen that, all pods have been deployed in Node 1.

# Step 8: Activity of Scheduler

Now we are going to check, what will happen if Node 1 crash or stop working within cluster.

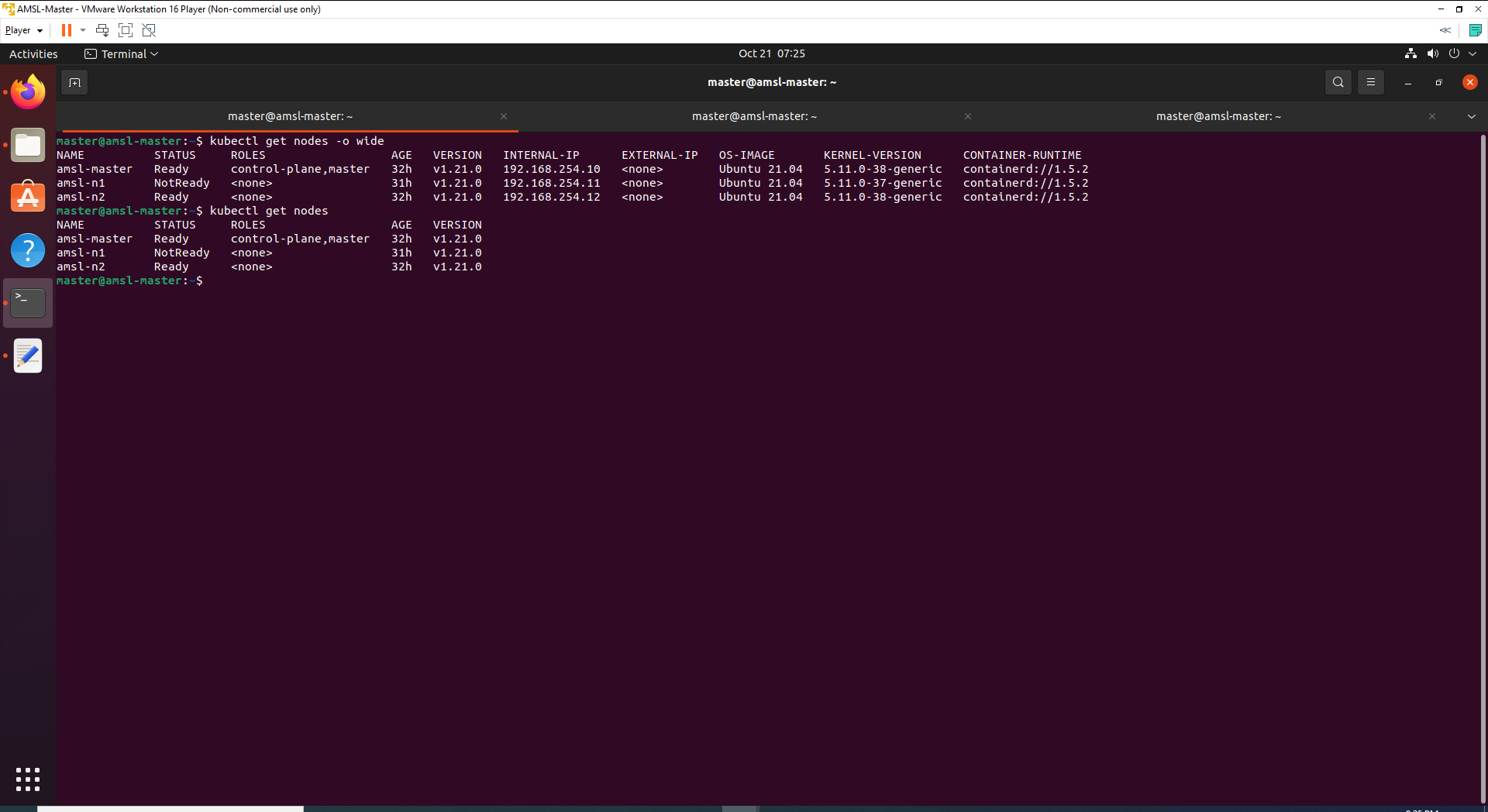
I have stopped Node 1 for the testing purpose.

1. Stop or Shut down Node 1 manually.



1. Check if Node 1 stop properly with below command.

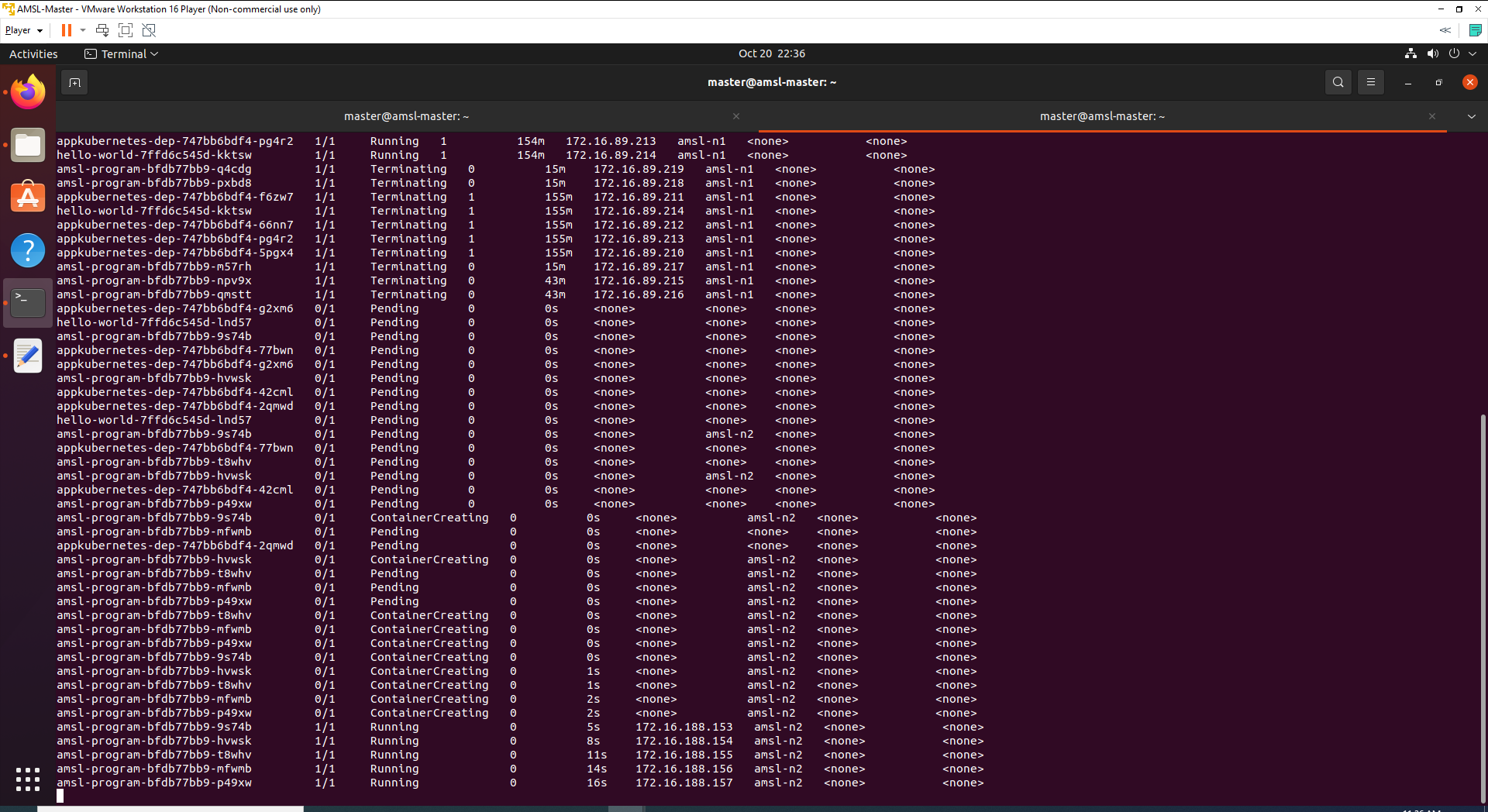
kubectl get nodes



As you can see that Node 1 or amsl-n1 status is not Ready.

1. Run below command to monitor the situation pods.

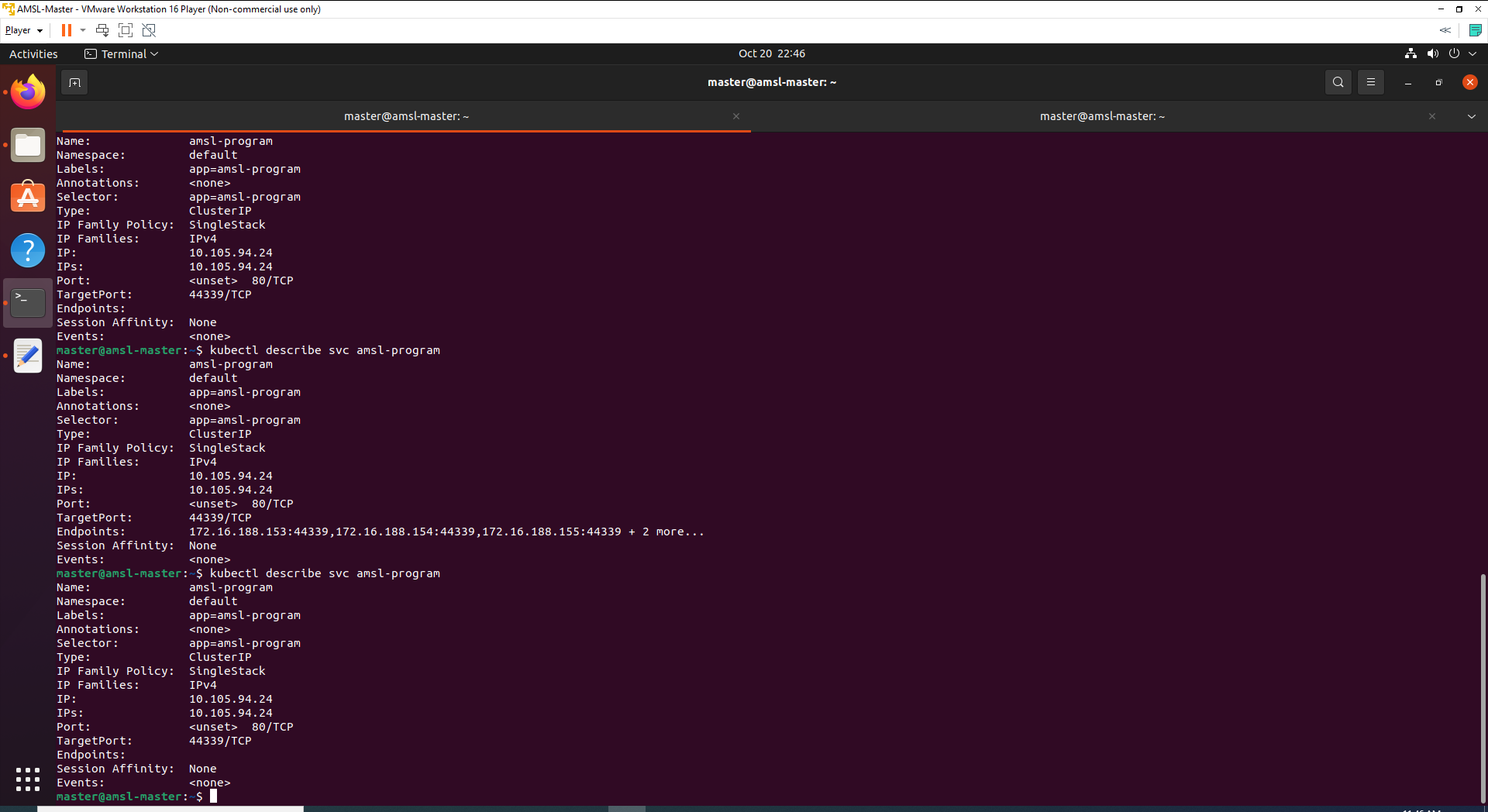
kubectl get pods -o wide –watch



You will see that pods are Terminating from Node 1 and Creating the container on Node 2. This process is automatically done by scheduler because we have set instruction earlier in program-deployment.yaml file. Exit by pressing ( ctrl+c ).

1. Check if endpoint are exist or not in the service, because earlier endpoints were created and directed at Node 1.

kubectl describe svc amsl-program

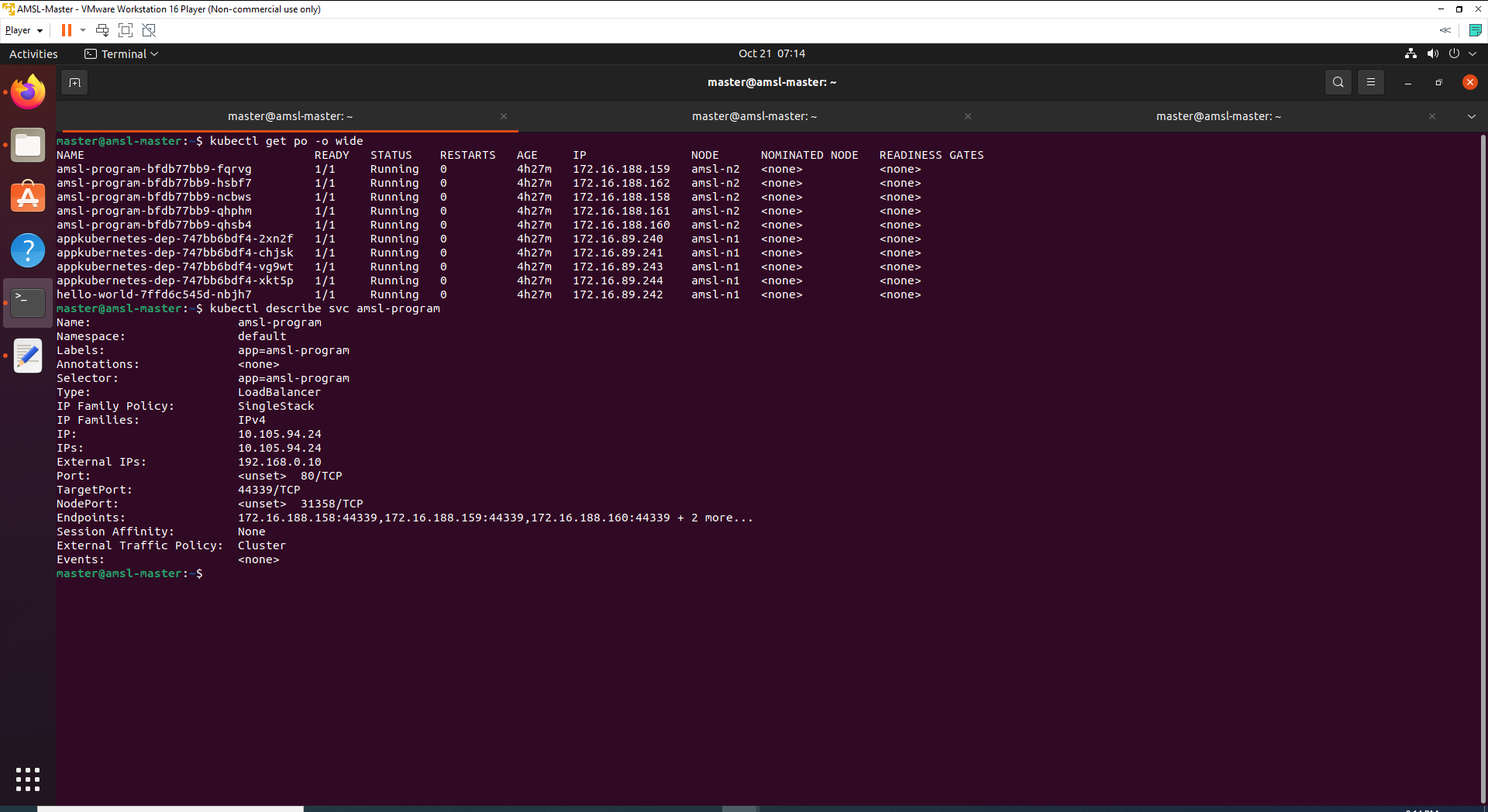


You can see that endpoints are empty.

1. As we have seen earlier pods were terminating from Node 1 and creating on Node 2, now recheck the situation of pods and services.

kubectl get pods -o wide

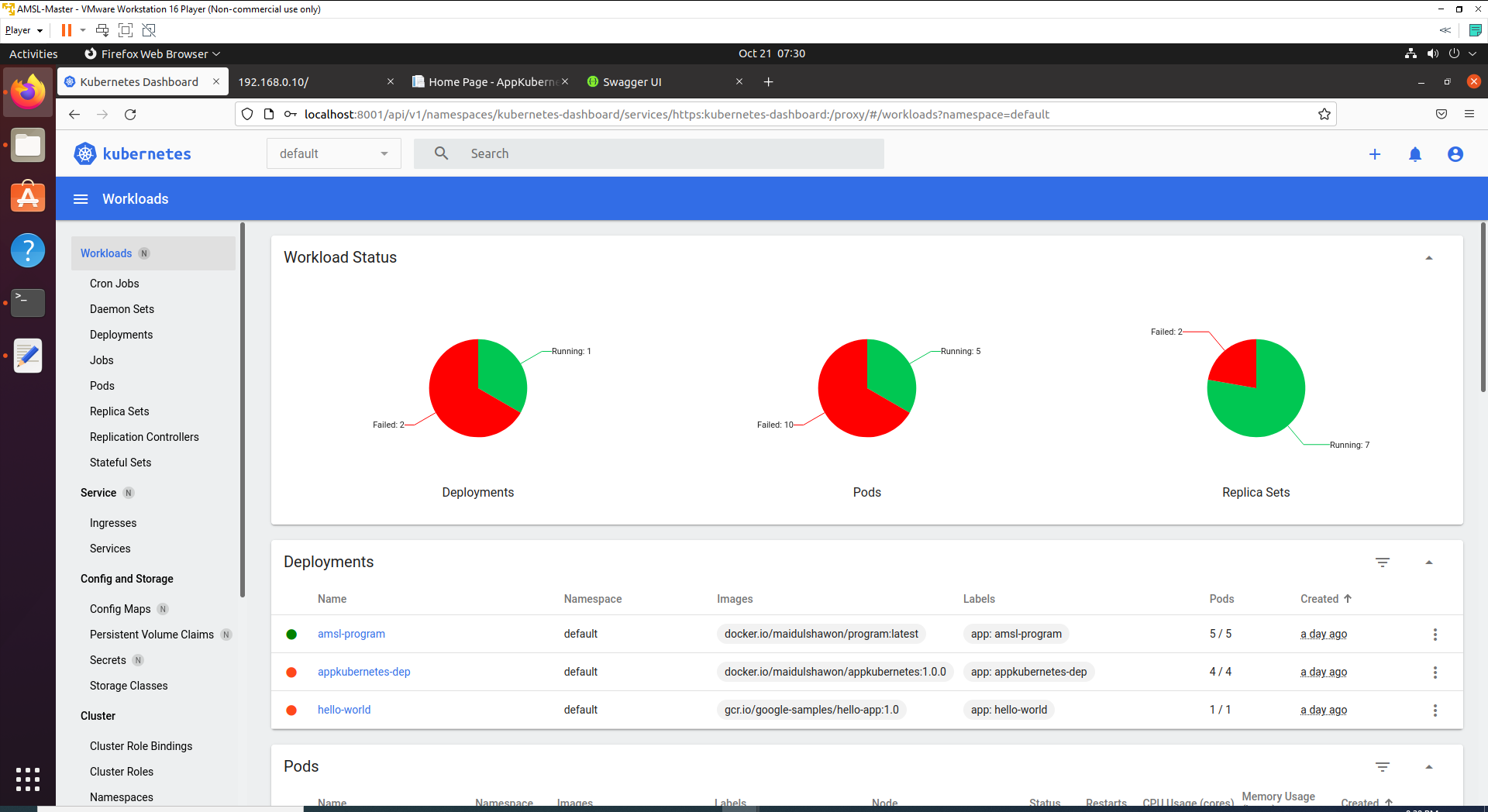
kubectl describe svc amsl-program



On the above output, you can see that endpoints addresses have been changes. As we have used load balancer with external IP, endpoint changes will not affect the application.

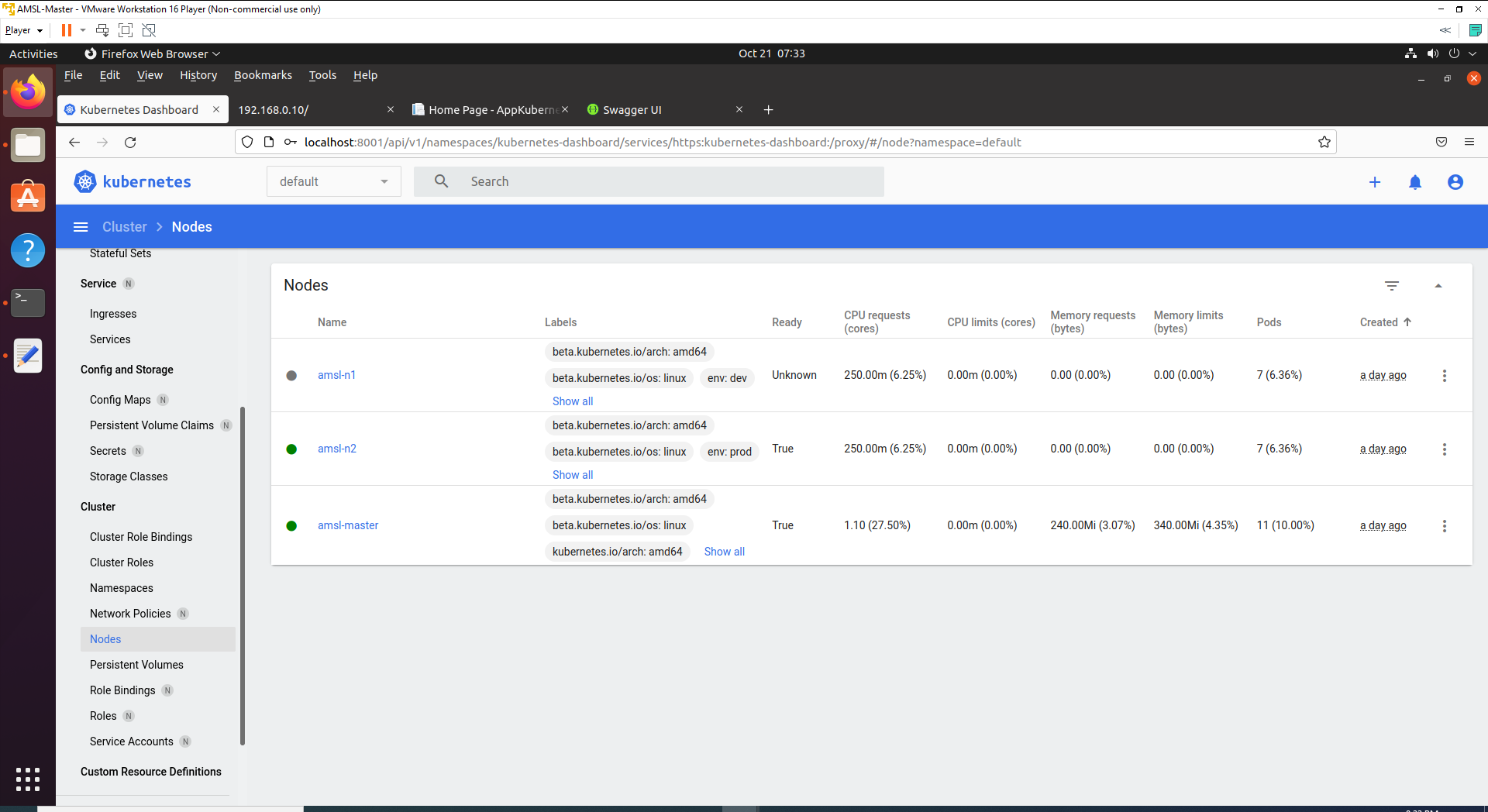
You can now re-visit <http://192.168.0.10/swagger/index.html> and will see that application is running on the same External IP we exposed on service.

If you go to kubernetes UI you will see output like below.



As you can see that, only running deployment is amsl-program and running pods are total 5 which we defined as 5 replica set. Other applications are stop working because all those pods were deployed at Node 1 with Node selector.

Node Selector is another way of deploying pod to a specific Node.



We can see that amsl-n1 or Node 1 is not in the Ready state.

# References

<https://kubernetes.io/docs/reference/>

<https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/>

<https://v1-18.docs.kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/>

<https://codebots.com/docs/ubuntu-18-04-virtual-machine-setup>

<https://kubernetesclustersetup.blogspot.com/>

<https://www.brianlinkletter.com/2016/07/how-to-use-virtualbox-to-emulate-a-network/>

<https://docs.openshift.com/dedicated/3/architecture/core_concepts/pods_and_services.html>

<https://hub.docker.com/>

<https://stackoverflow.com/questions/64884790/failed-to-update-endpoint-default-myservice-operation-cannot-be-fulfilled-on-e>