Kubernetes Theory and Complete Project

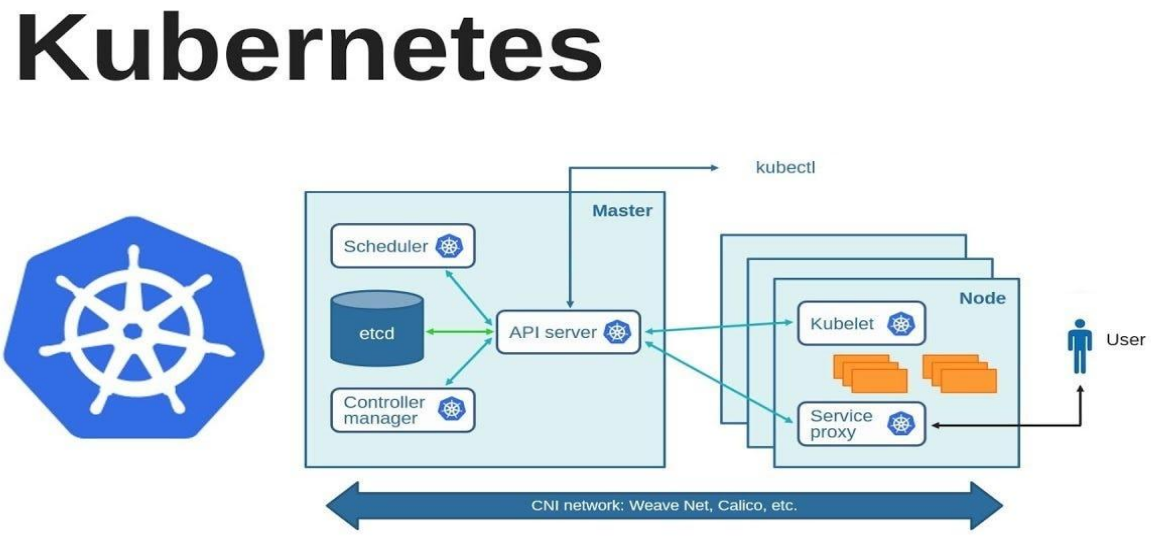
What is Kubernetes:

- An open source container orchestration tool used to manage multi container deployment.
- Why kubernetes if we already have Docker Compose, Docker Swarm-
- For production ready deployment of microservices.
- Having less failures and downtimes.
- Good backup and restores strategy.

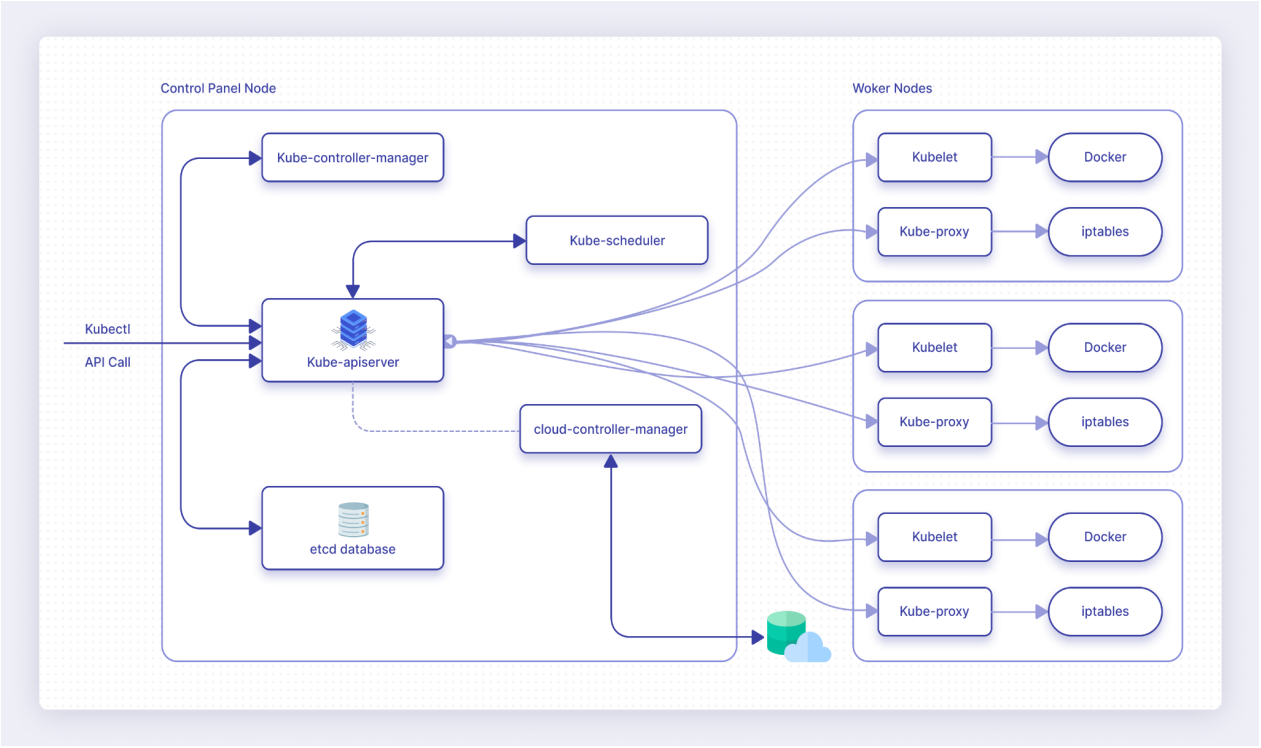
Microservices vs. Monolithic Architecture:

- If all the functionalities of a project exist in a single codebase, then that application is known as a monolithic application. We design our application in various layers like presentation, service, and persistence and then deploy that codebase as a single jar/war file. This is nothing but a monolithic application, where **“mono”** represents the single codebase containing all the required functionalities.
- It is an architectural development style in which the application is made up of smaller services that handle a small portion of the functionality and data by communicating with each other directly using lightweight protocols like HTTP. According to Sam Newman, “Microservices are the small services that work together.”

Architecture Diagram:



- Based on master-slave architecture where master runs on different server than slaves.
- Actual application code runs on slaves nodes while master consist only configuration files, secret files, etcd, scheduling etc.
- Kubernetes API server validates and configures data for the api objects which include pods, services, replicationcontrollers, and others. It manages the slave nodes.
- **Minikube** - k8s cluster with single node. used for test purposes. If you want to quickly spin up a k8s environment on your laptop.
- **kubectl** - command line tool for k8s cluster. you can create, delete, update components and debug stuff in k8s using *kubectl*.
- **kubelet** - k8s process that runs on each node to manage containers: starting, communicating with them etc.
- So in combination, you may have a **minikube** cluster with 1 node that has **kubelet** process running on it, and you can create/update/delete things in the clusterusing **kubectl.**



Kubernetes Cluster Components: Two major components!

1. **Control Plane Components:**
   The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new pod when a deployment's `replicas` field is unsatisfied).

   Control plane components can be run on any machine in the cluster. However, for simplicity, set up scripts typically start all control plane components on the same machine, and do not run user containers on this machine.

   1. **API Server:**

      The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane.

      The main implementation of a Kubernetes API server is kube-apiserver. kube-apiserver is designed to scale horizontally—that is, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances.

   2. **Etcd:**

      Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

      If your Kubernetes cluster uses etcd as its backing store, make sure you have a back up plan for those data.

   3. **kube-scheduler:**

      Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on.

   4. **kube-controller-manager:**

      Control plane component that runs controller processes. Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

      Some types of these controllers are:

- Node controller: Responsible for noticing and responding when nodes go down.
- Job controller: Watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.
- EndpointSlice controller: Populates EndpointSlice objects (to provide a link between Services and Pods).
- ServiceAccount controller: Create default ServiceAccounts for new namespaces.

5. **cloud-controller-manager:**

   A Kubernetes control plane component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with your cluster. The cloud-controller-manager only runs controllers that are specific to your cloud provider.

   The following controllers can have cloud provider dependencies:
   - Node controller: For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding
   - Route controller: For setting up routes in the underlying cloud infrastructure
   - Service controller: For creating, updating and deleting cloud provider load balancers

2. Node Components:

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

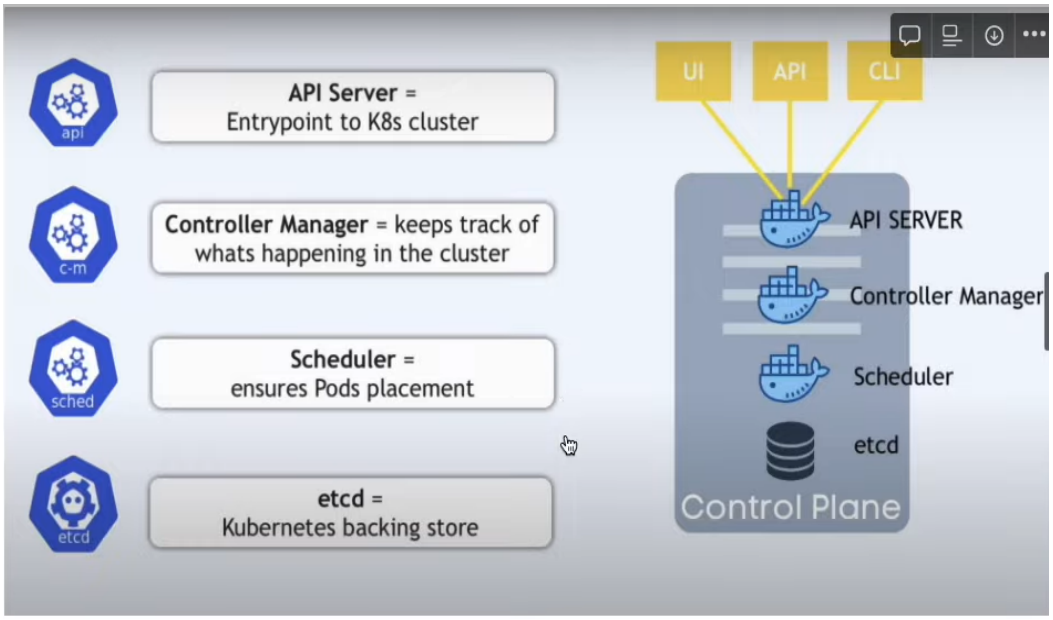
1. **kubelet:**

   An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod. The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

Each node runs two main components - *a kubelet and a container runtime*. The kubelet is in charge of facilitating communication between the control plane and the node.

2. **kube-proxy:**

   kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept. kube-proxy maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.



Node vs Pod vs Kubernetes Container:

- Node is a server or virtual machine. Nodes in Kubernetes can be described as the smallest units of computing power. They are a collection of CPU and memory resources used by the tool to run processes. Nodes are usually available in clusters — intelligent groups of nodes that can distribute workload among their component nodes to increase efficiency.
- Pods are the smallest independent deployable units in Kubernetes. Kubernetes pods are collections of containers that share the same resources and local network. This enables easy communication between containers in a pod. Pods

replicate a logical host for containers that are tightly coupled with each other. Although they can host multiple containers together, limiting the number of containers in one pod to a minimum is advised. This is because pods are meant to be scaled up and down quickly, and each container in the pod is scaled with it irrespective of its requirements.

- In a nutshell, containers are packages of applications or services bundled together with their execution environments. Containers are a powerful CI/CD asset since they can be created and modified programmatically.

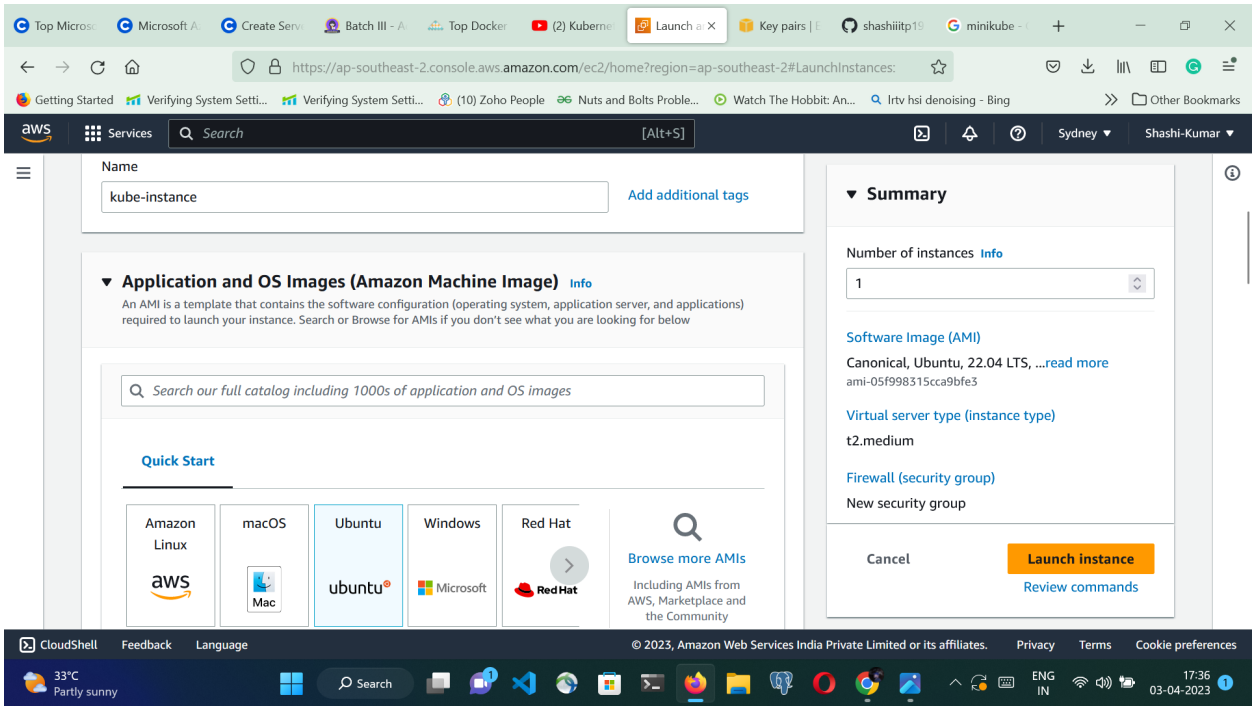
How Do Nodes, Pods, and Containers Work With Each Other?

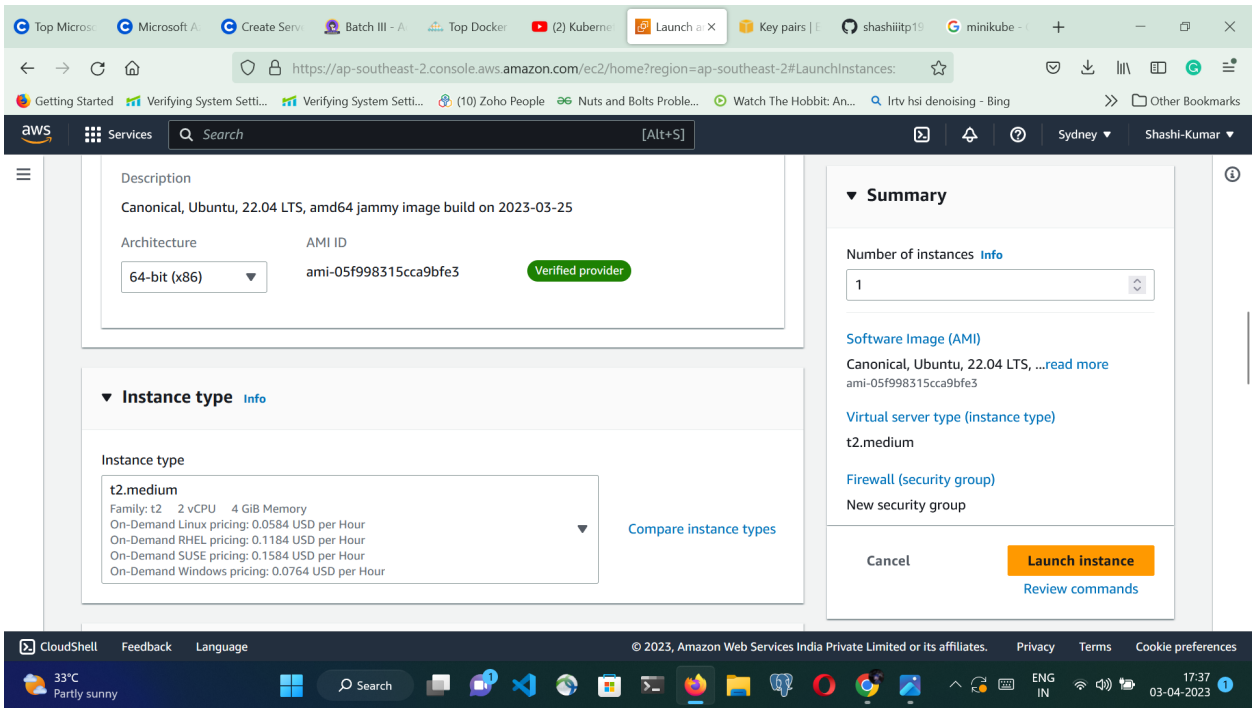
1. Containers are packages of applications and execution environments.
2. Pods are collections of closely-related or tightly coupled containers.
3. Nodes are computing resources that house pods to execute workloads.

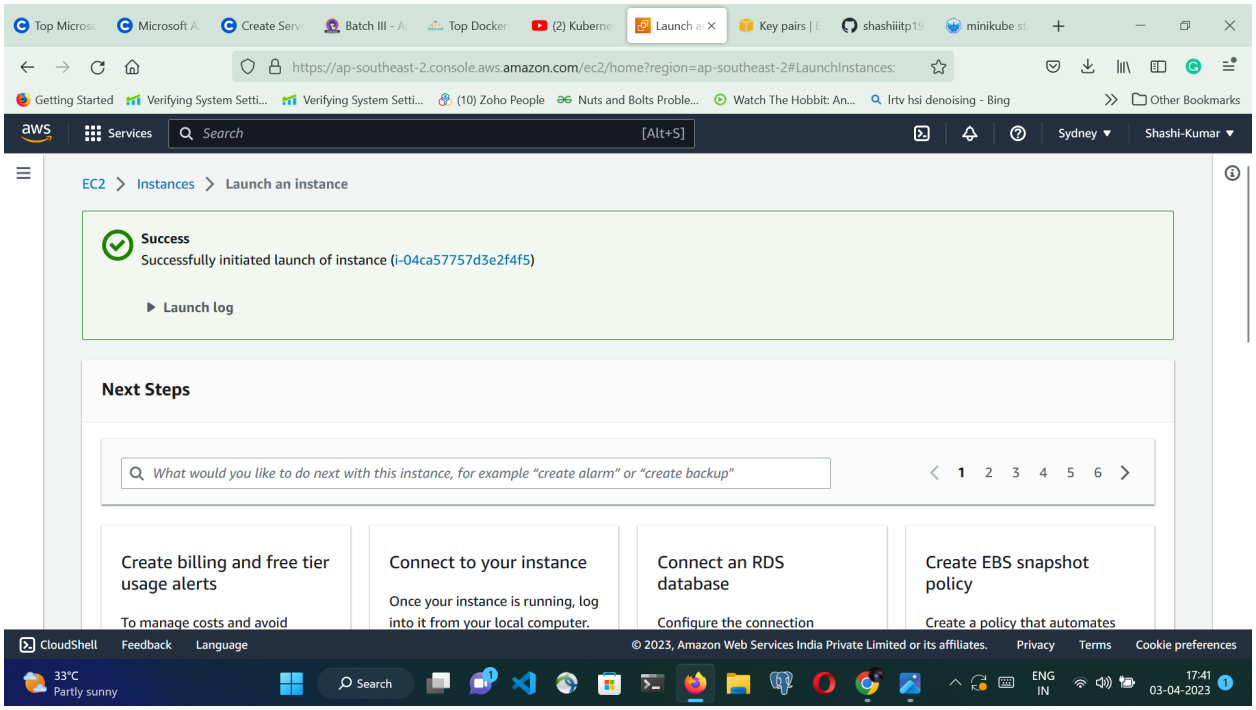
Kubernetes Project

As we know we need to different server for master and slave which might be costly. So there is a tool **Minikube** used to run kubernetes locally.

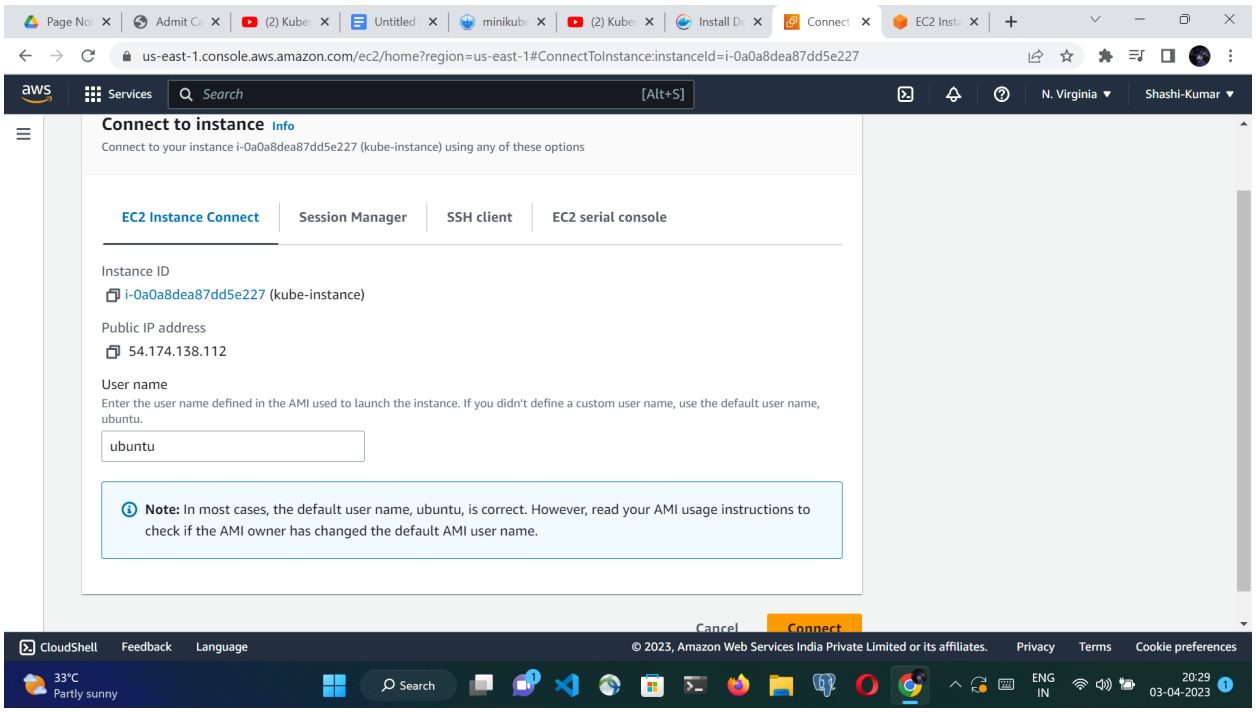
Step-1. Create an EC2 ubuntu instance with name “kube-instance” and machine type t2.medium.
Important: t2.medium is not free tier, we are using because it is as per the require ment of minikube. So delete it after completion of this project asap.


Name
kube-instance
Add additional tags
Application and OS Images (Amazon Machine Image) Info
An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. Search or Browse for AMIs if you don't see what you are looking for below
Search our full catalog including 1000s of application and OS images
Quick Start
Amazon Linux
macOS
Ubuntu
Windows
Red Hat
Browse more AMIs
Including AMIs from AWS, Marketplace and the Community
Summary
Number of instances Info
1
Software Image (AMI)
Canonical, Ubuntu, 22.04 LTS, ...read more
ami-05f998315cca9bfe3
Virtual server type (instance type)
t2.medium
Firewall (security group)
New security group
Cancel
Launch instance
Review commands


Description
Canonical, Ubuntu, 22.04 LTS, amd64 jammy image build on 2023-03-25
Architecture
64-bit (x86)
AMI ID
ami-05f998315cca9bfe3
Verified provider
Instance type Info
Instance type
t2.medium
Family: t2 2 vCPU 4 GiB Memory
On-Demand Linux pricing: 0.0584 USD per Hour
On-Demand RHEL pricing: 0.1184 USD per Hour
On-Demand SUSE pricing: 0.1584 USD per Hour
On-Demand Windows pricing: 0.0764 USD per Hour
Compare instance types
Summary
Number of instances Info
1
Software Image (AMI)
Canonical, Ubuntu, 22.04 LTS, ...read more
ami-05f998315cca9bfe3
Virtual server type (instance type)
t2.medium
Firewall (security group)
New security group
Cancel
Launch instance
Review commands

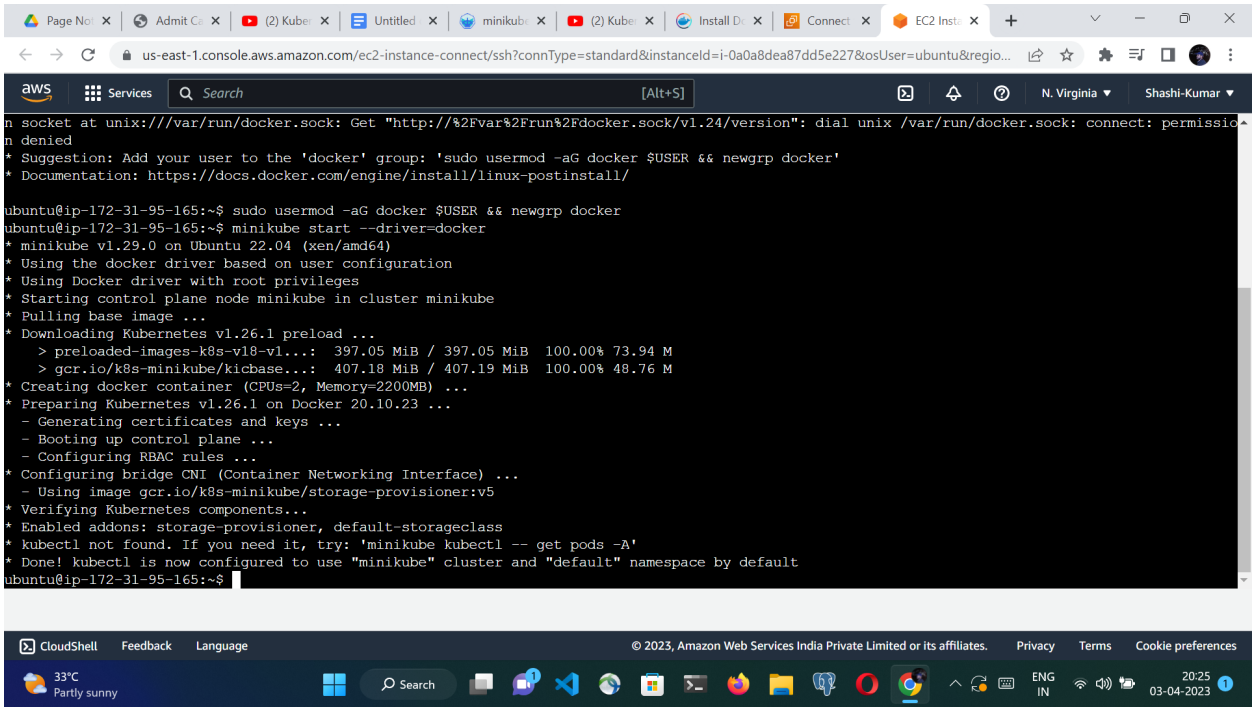


Connect to instance-



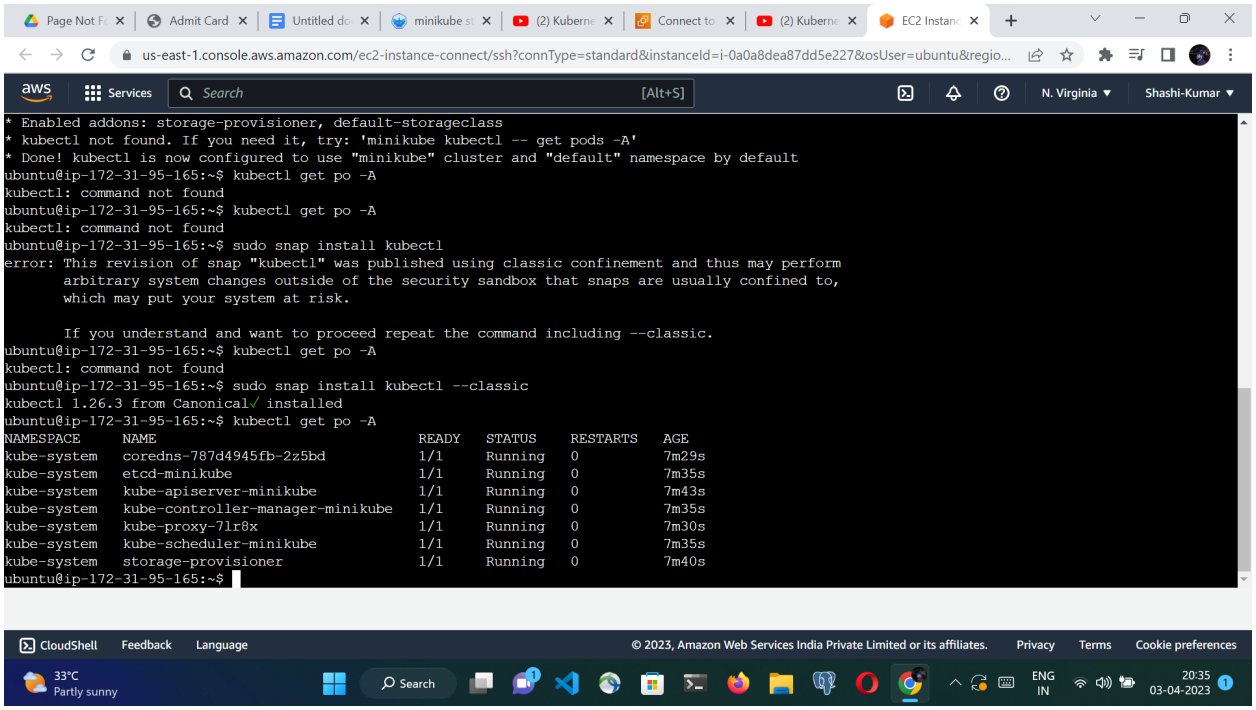
Step-2.

Install docker: https://docs.docker.com/engine/install/ubuntu/
Install Minikube: https://minikube.sigs.k8s.io/docs/start/
Minikube installed successfully.



After minikube installation:

- sudo snap install kubectl --classic (this will install kubectl CLI)
- kubectl get po -A

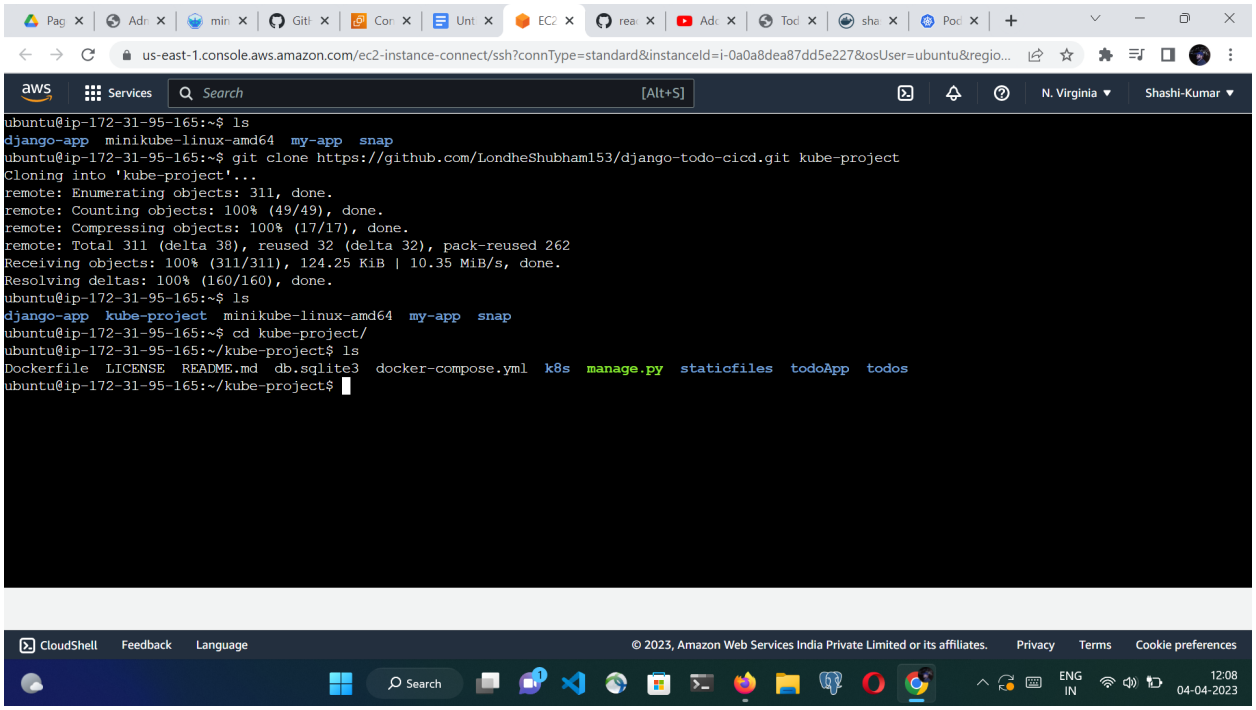


“Learn How to Manage Docker Containers”

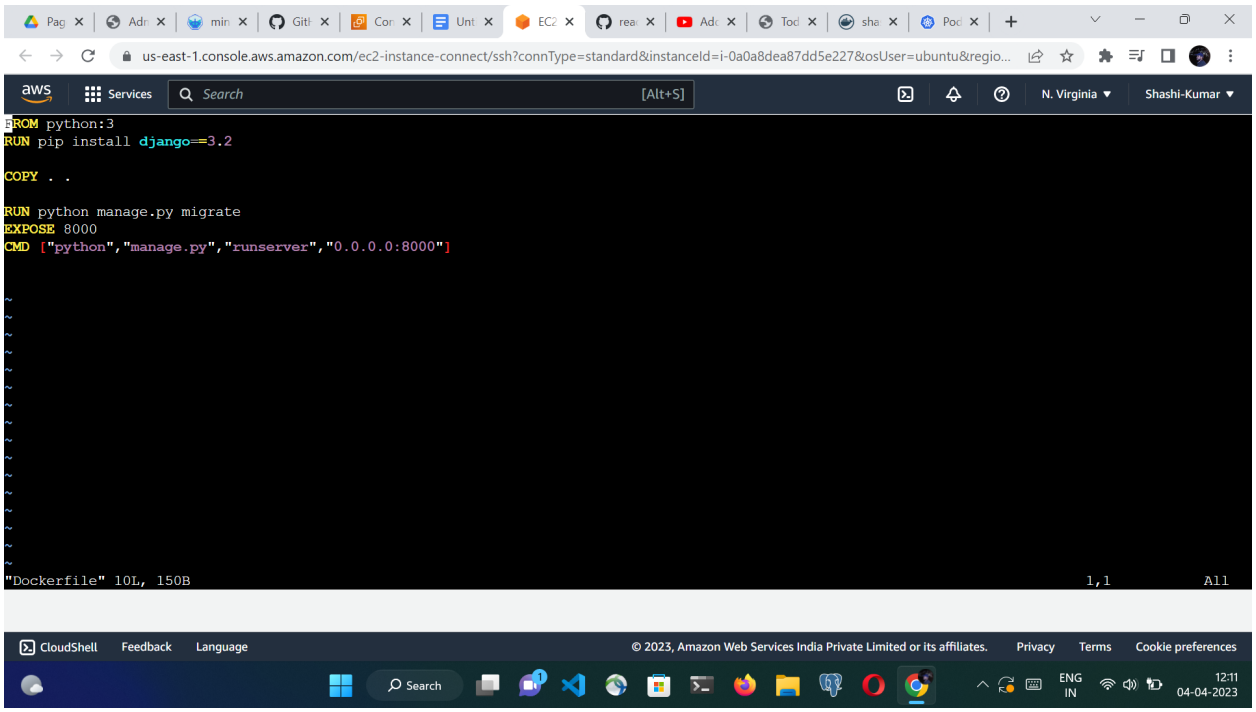
Step-3. Now first create a docker image of our app and run it to verify if app is running. After this just delete that image.

- git clone https://github.com/LondheShubham153/django-todo-cicd.git kube-project
- cd kube-project
- ls

Verify if you can see all the files.

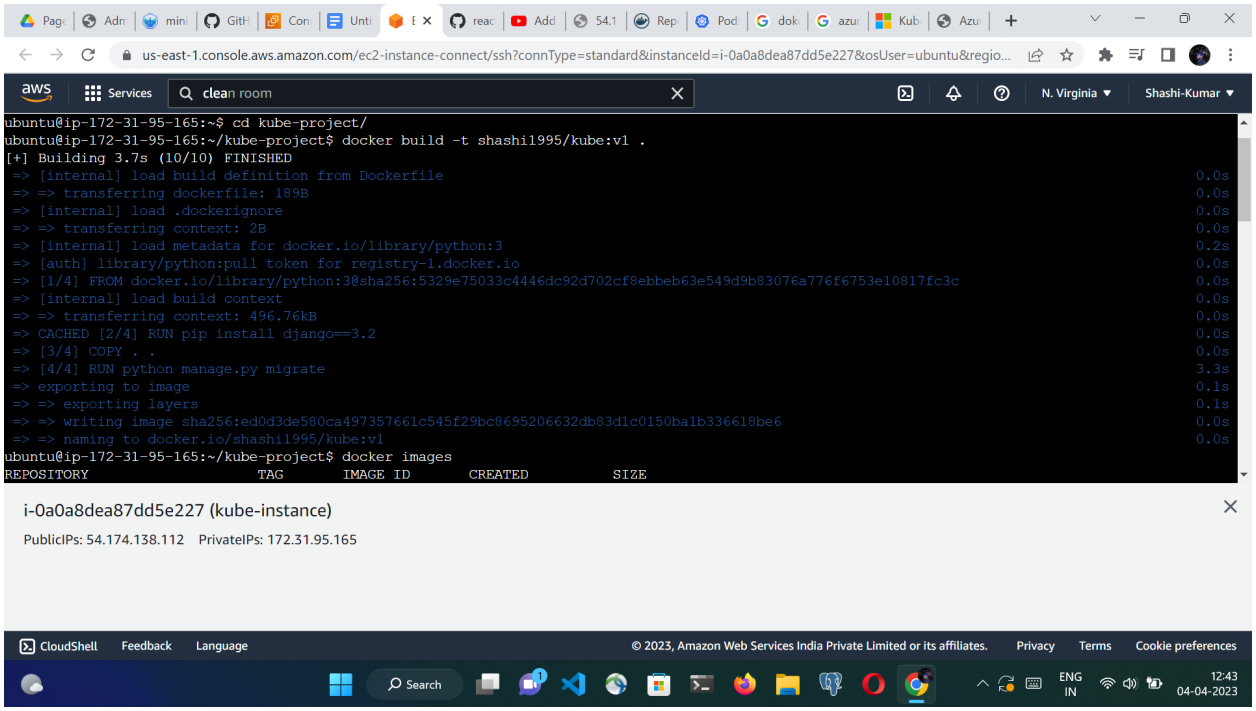


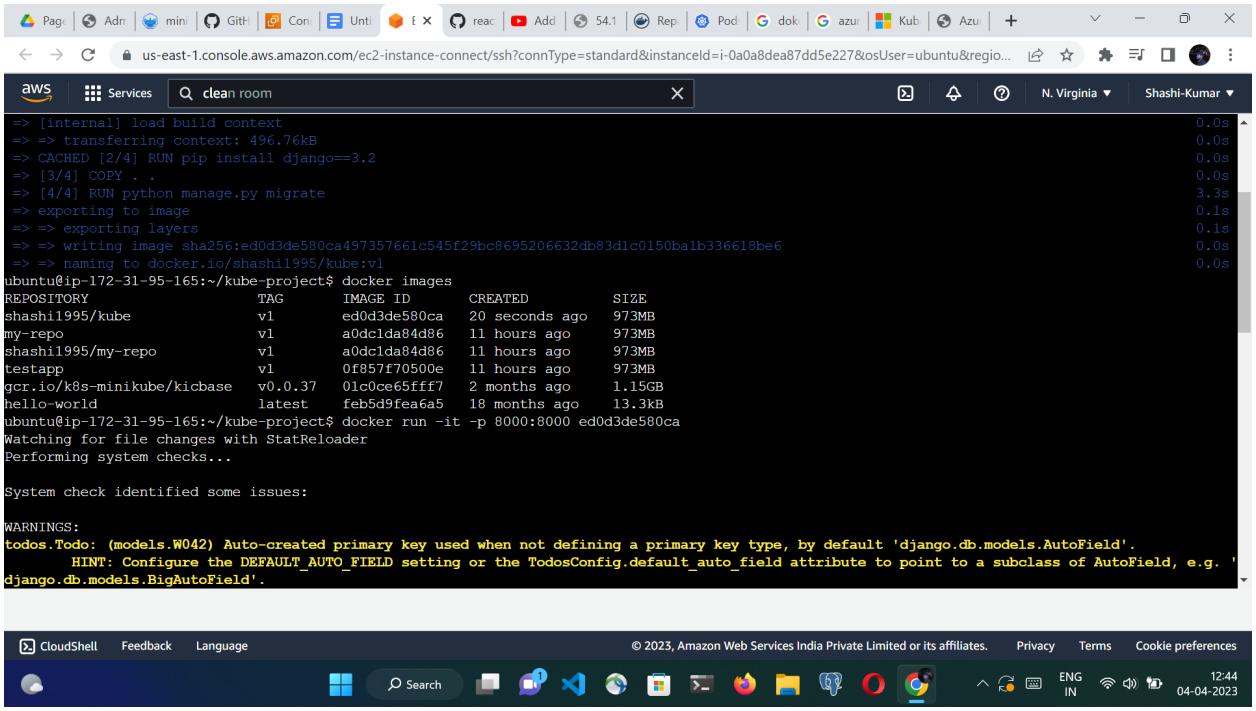
- vim Dockerfile and verify if it is correct and save.

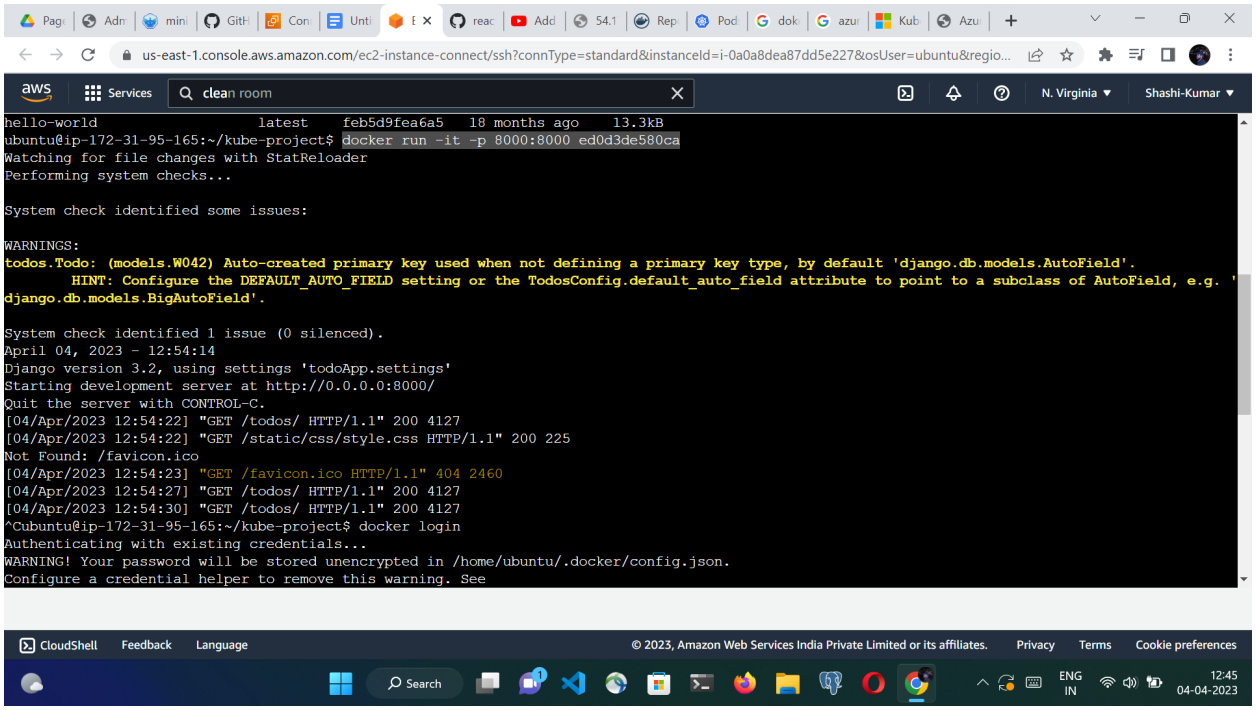


Step-4. Build and run docker image and push it to Docker Hub repository for centralized management.

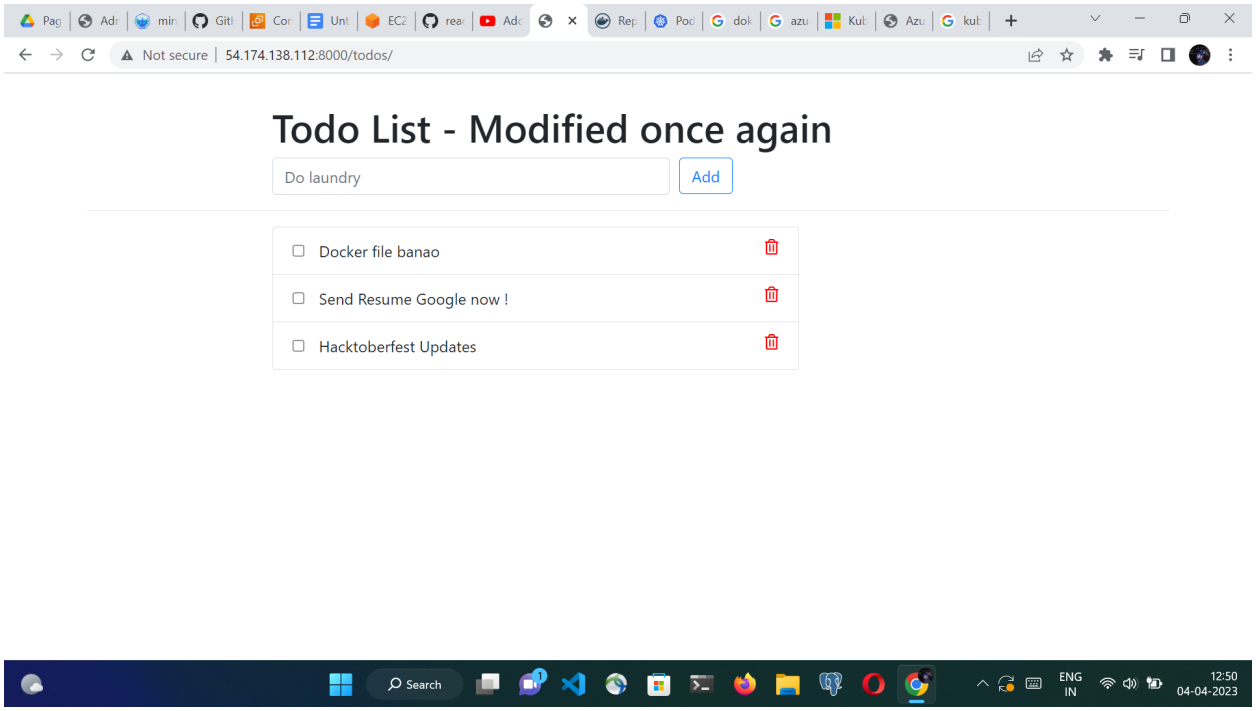
- docker build -t shashi1995/kube:v1 .
- docker images
- docker run -it -p 8000:8000 your-image-id





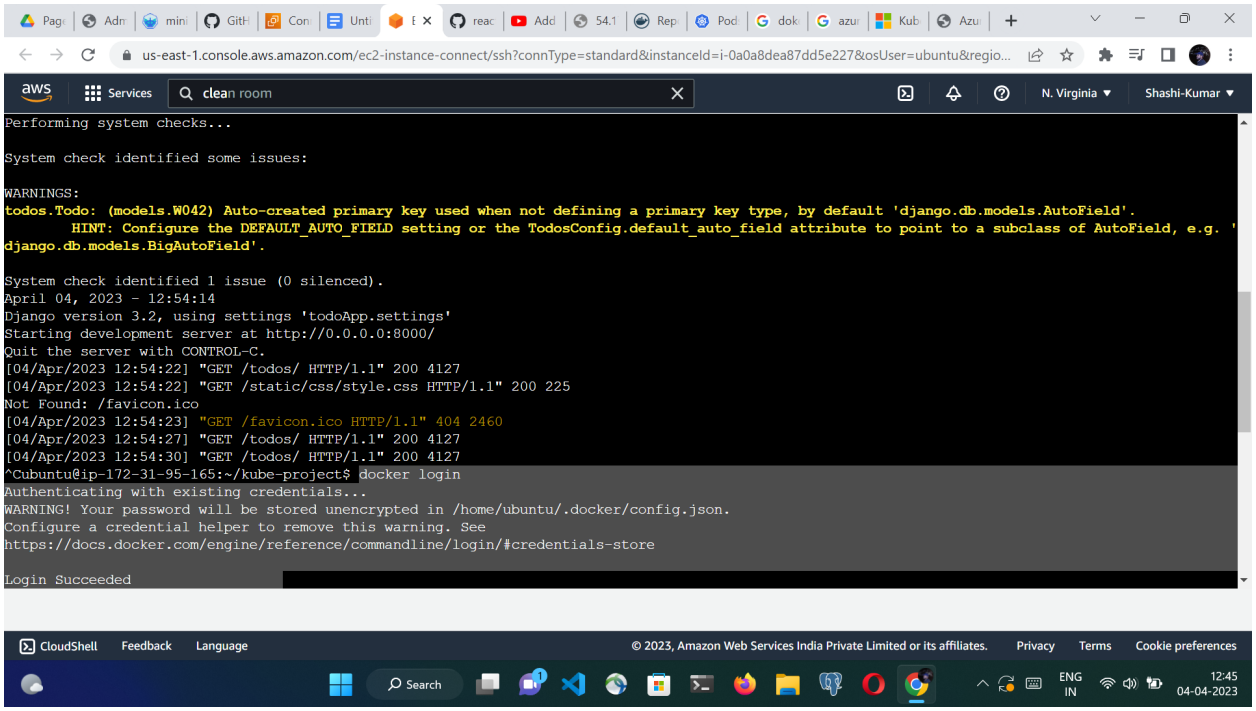


After running the container, our app is running on 8000-



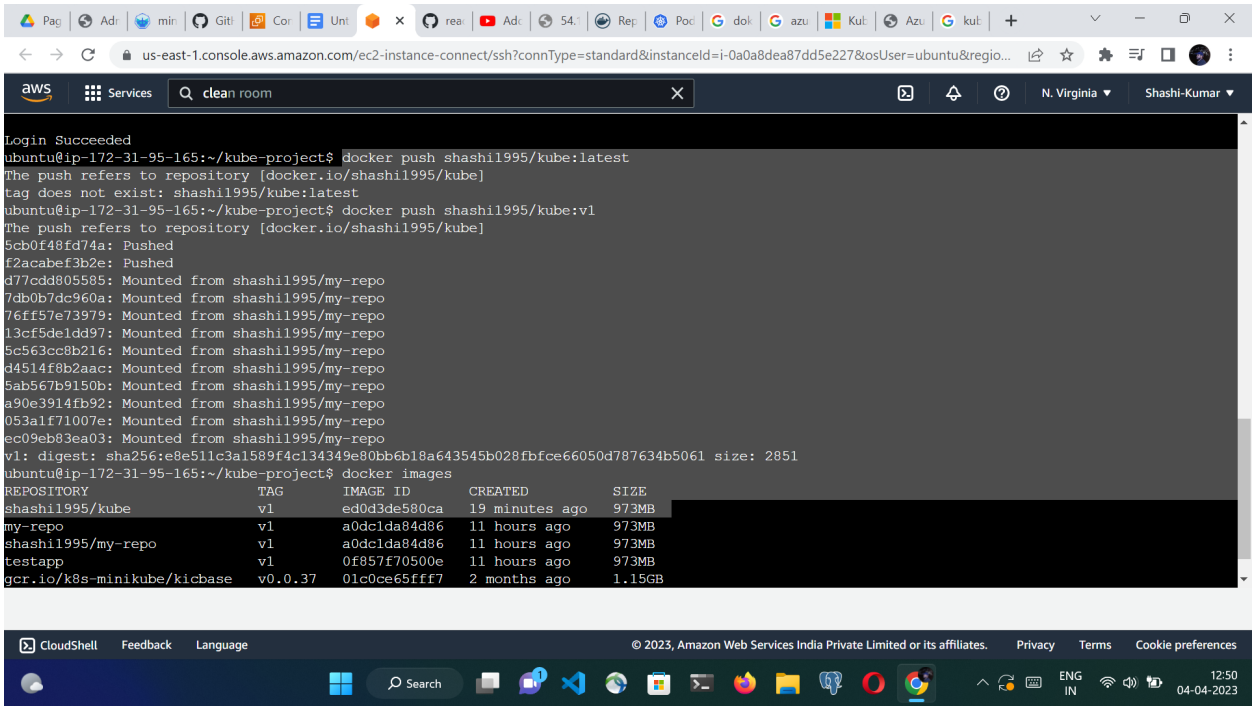
Step-5. Login to DockerHub and push you image there. So that whenever required you can pull it.

- docker login

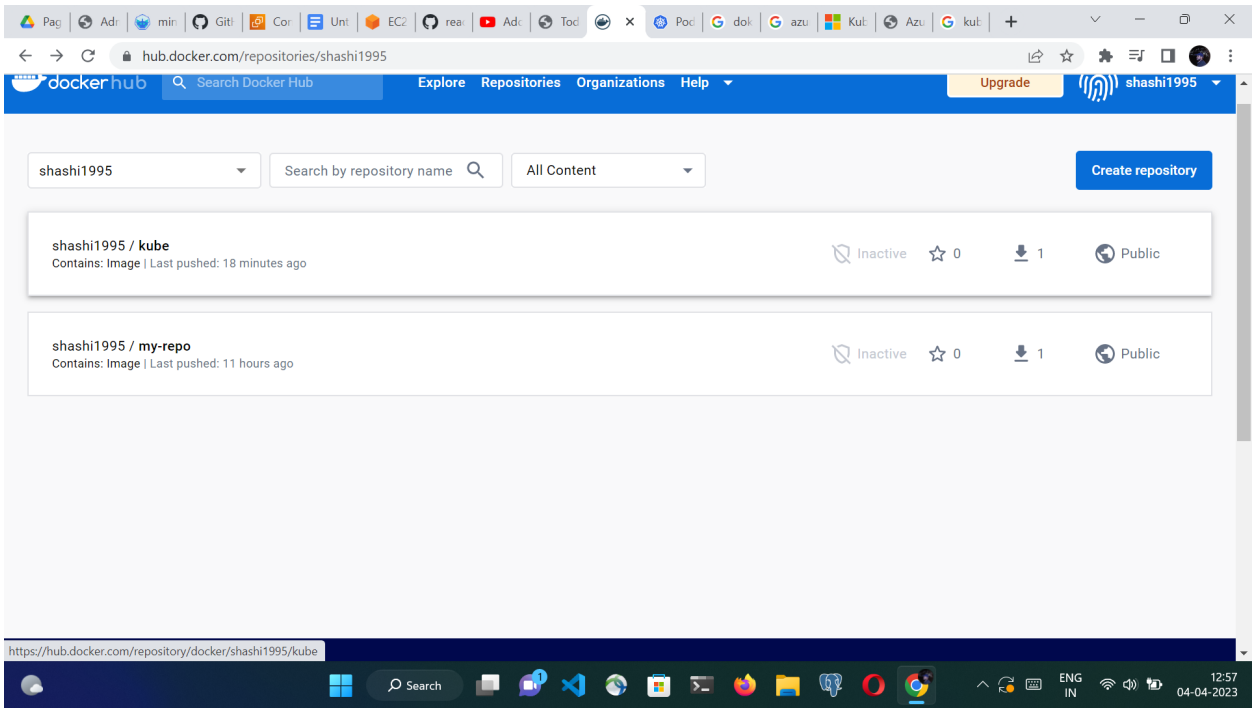


Push the images to your docker hub:

- docker push shashi1995/kube:v1



Verify from dockerhub-



“Learn How to Manage Deployment, Replication, Auto-Healing and Auto-Scaling.”

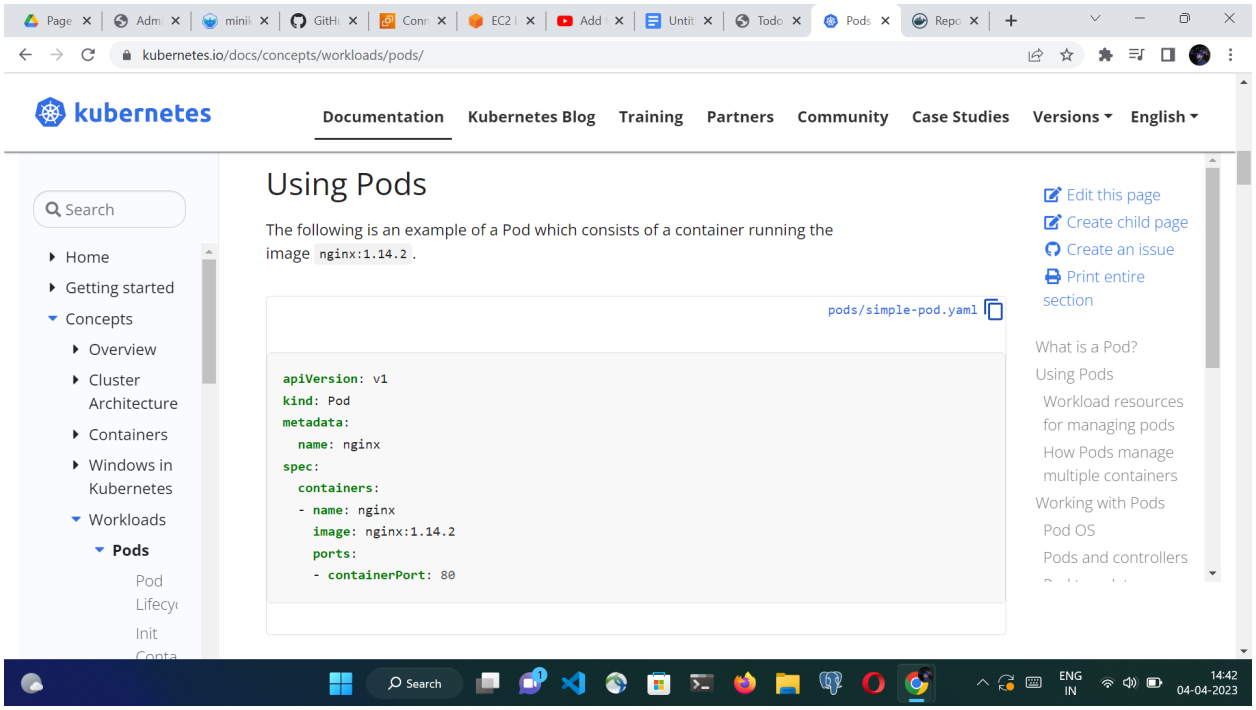
Q- Now why we are using kubernetes when the app can be run through docker only?

- Docker is not production ready approach, which means there are several issues that may occur like-
    1. Containers are short term, means it may be crashed any time.
    2. In containers we are running processes and processes are prune to stop. E.g - some how our ec2 instance is down, due to heavy load ec2 is not able to provide required resources.
    3. Some by mistake pess Ctrl + C key, which will terminate the running container.
    4. Someone may kill our containers
    5. No autohealing and autoscaling

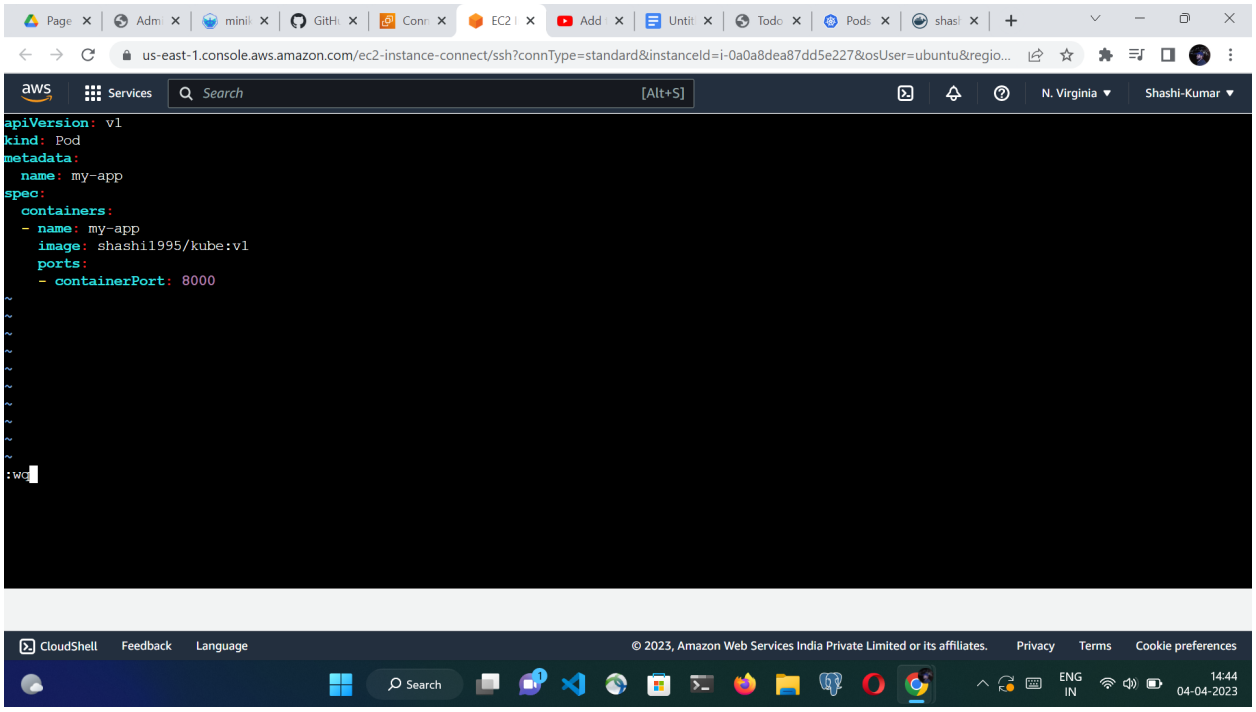
Start with Kubernetes:

1. Create a pod and run it with kubectl. It will pull the image from dockerhub.

   Create a new directory for all kubernetes files.
   - mkdir kube-files
   - cd kube-files
   - vim pod.yaml (Create a pod.yaml file and save it)



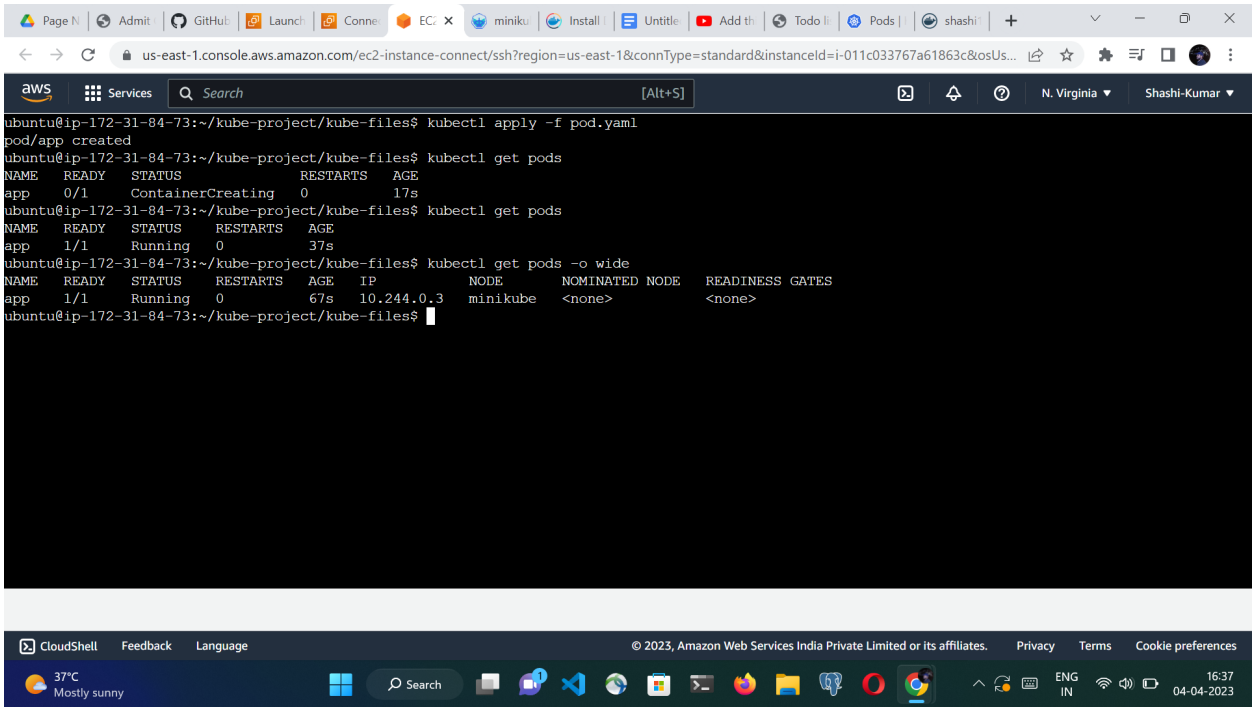
2. Edit it with your image and name-



3. Create our pod:
   kubectl apply -f pod.yaml (pod created)

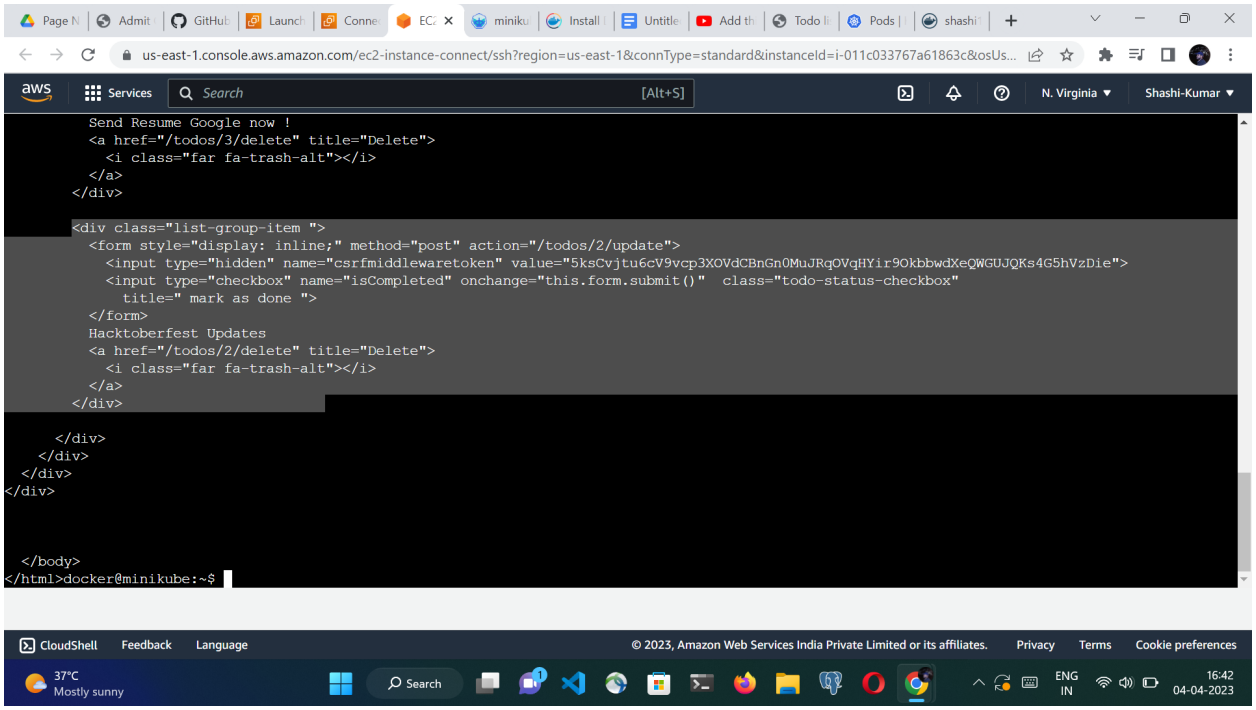
If getting server not connecting error-Delete your minikube and start: minikube delete && minikube start

kubectl get pods
kubectl get pods -o wide

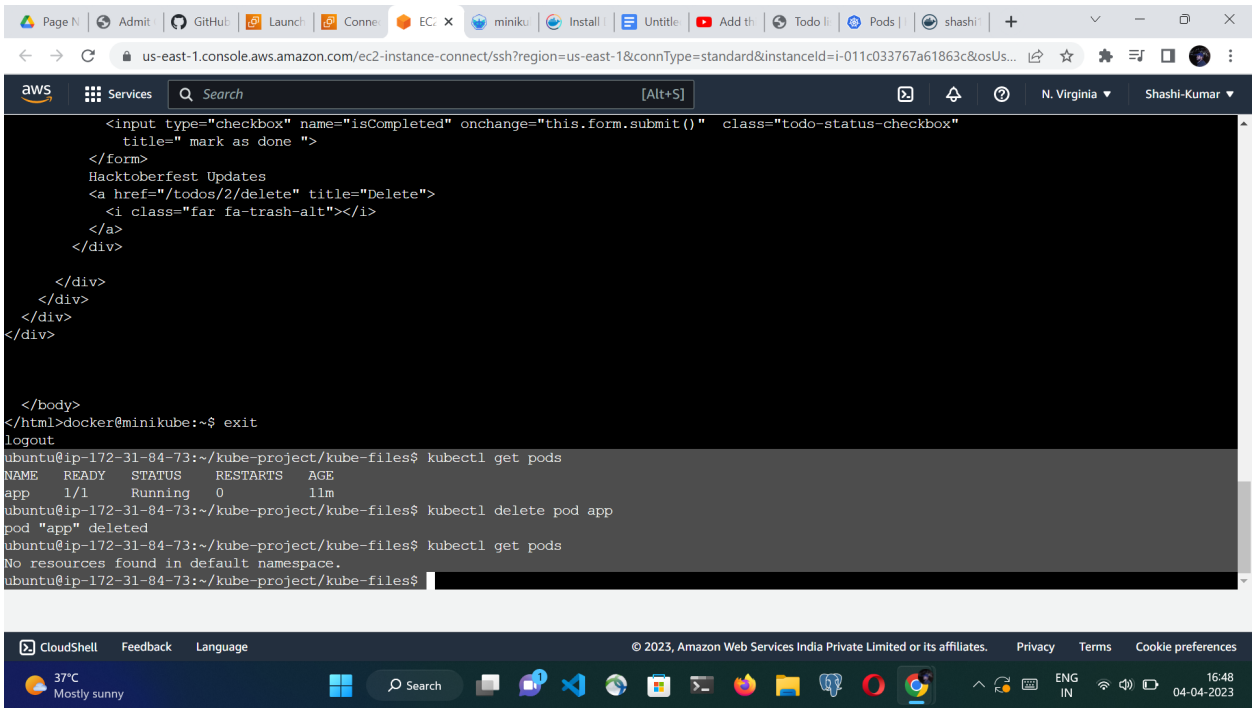


4. Now check if our app is running or not internally because as of now we have not exposed it to the outer world.

   curl -L http://podip:8000

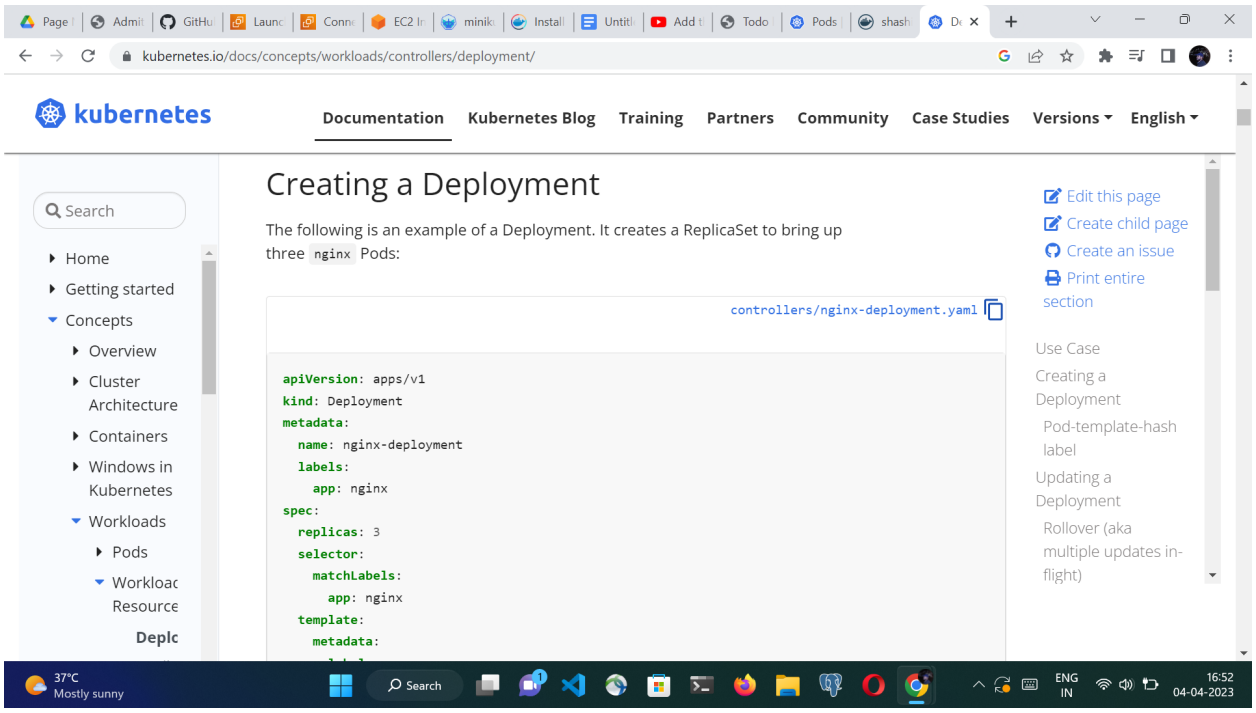


5. **But again this pod can be deleted or can be crashed.**
   For example: lets delete our pod
   - kubectl delete pod app-name



6. **So still we are suffering from this production ready deployment like auto healing and auto scaling.**
   But this is problem of pods only.

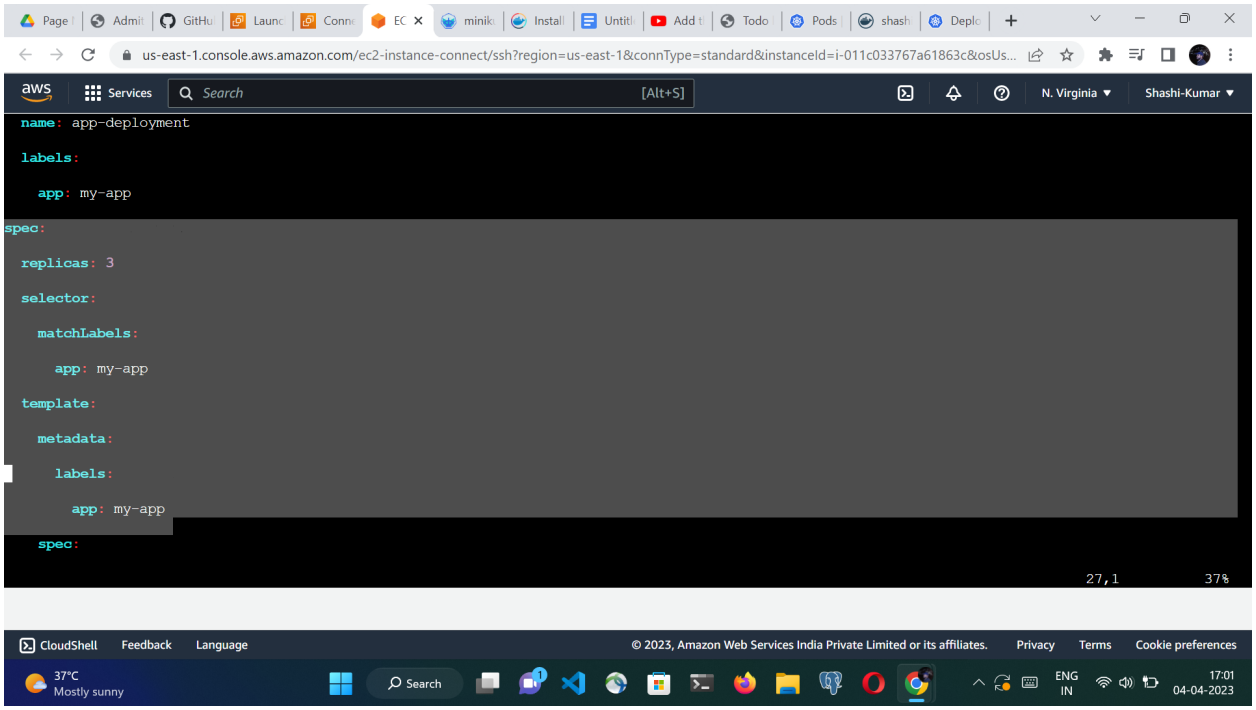
   In reality in production there is no deployment of pods.
   Actually there is a proper kubernetes deployment.



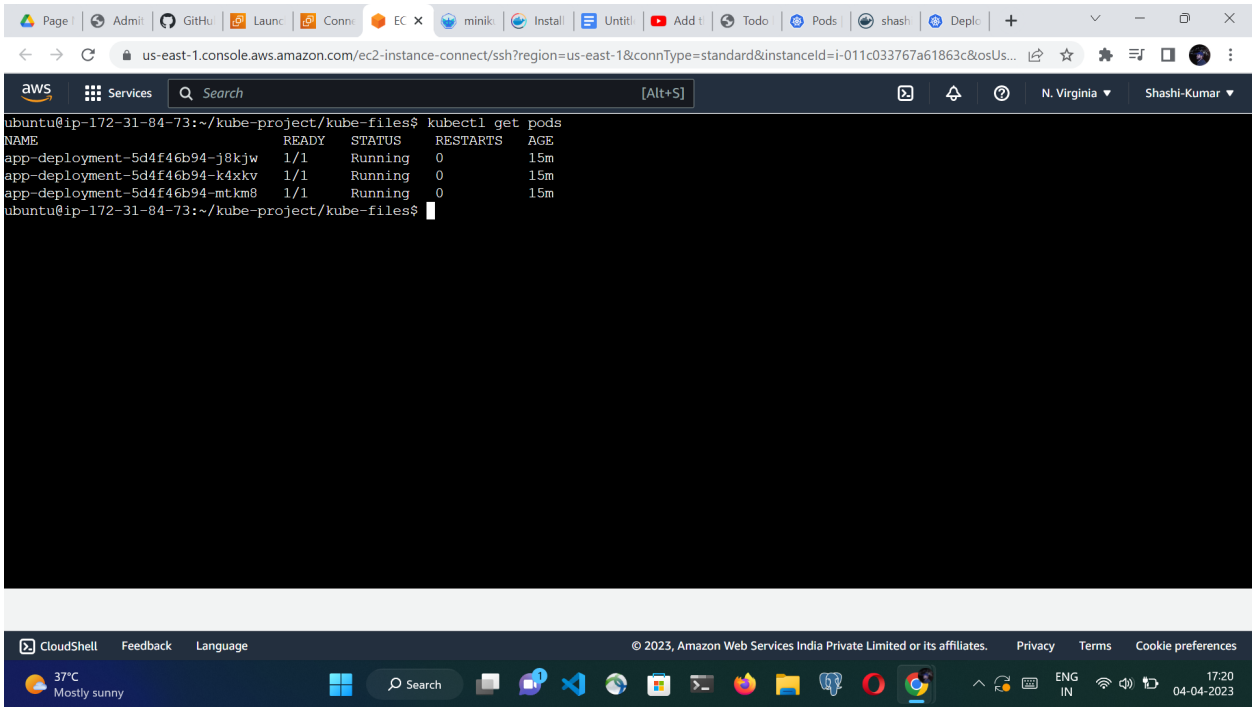
7. Create ‘deployment.yaml’ file
   For label and selector have the same name, because using this information controller checks the status of our deployment and accordingly takes actions.

   - vim deployment.yaml

   Edit the details like name and lebel-selector with same name, port -8000 and give the image path from dockerhub repo.



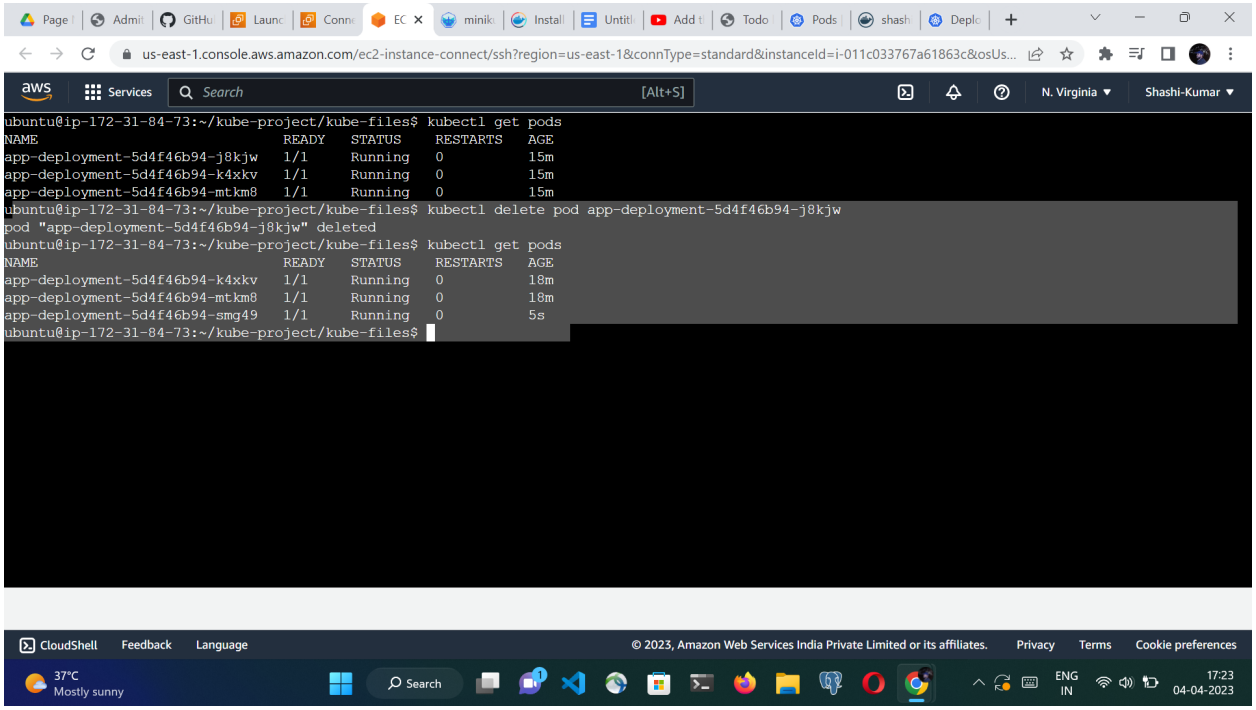
- kubectl get pods



3 pods are running and Now if you delete anyone, controller automatically creates a new container just in seconds.

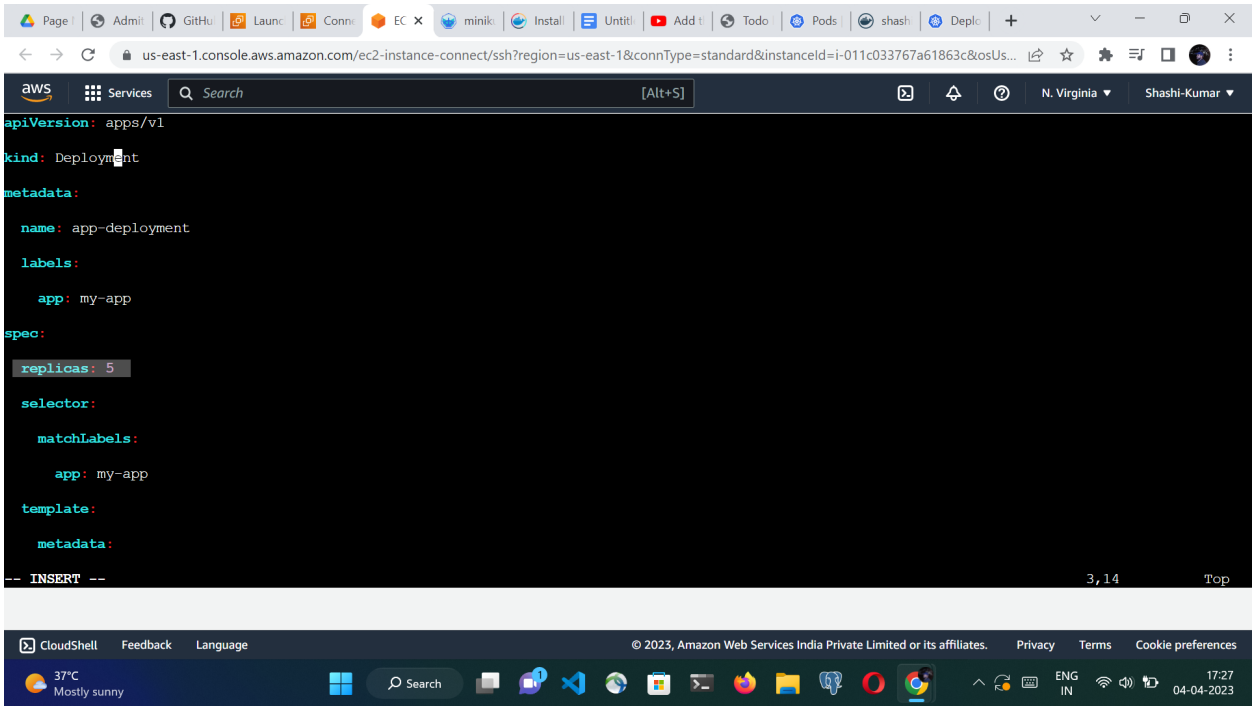
Try this by deleting:

- kubectl delete pod pod-name



Important - As you can see after deleting still there are 3 pods running because kubernetes now managing the **Auto-Healing**.

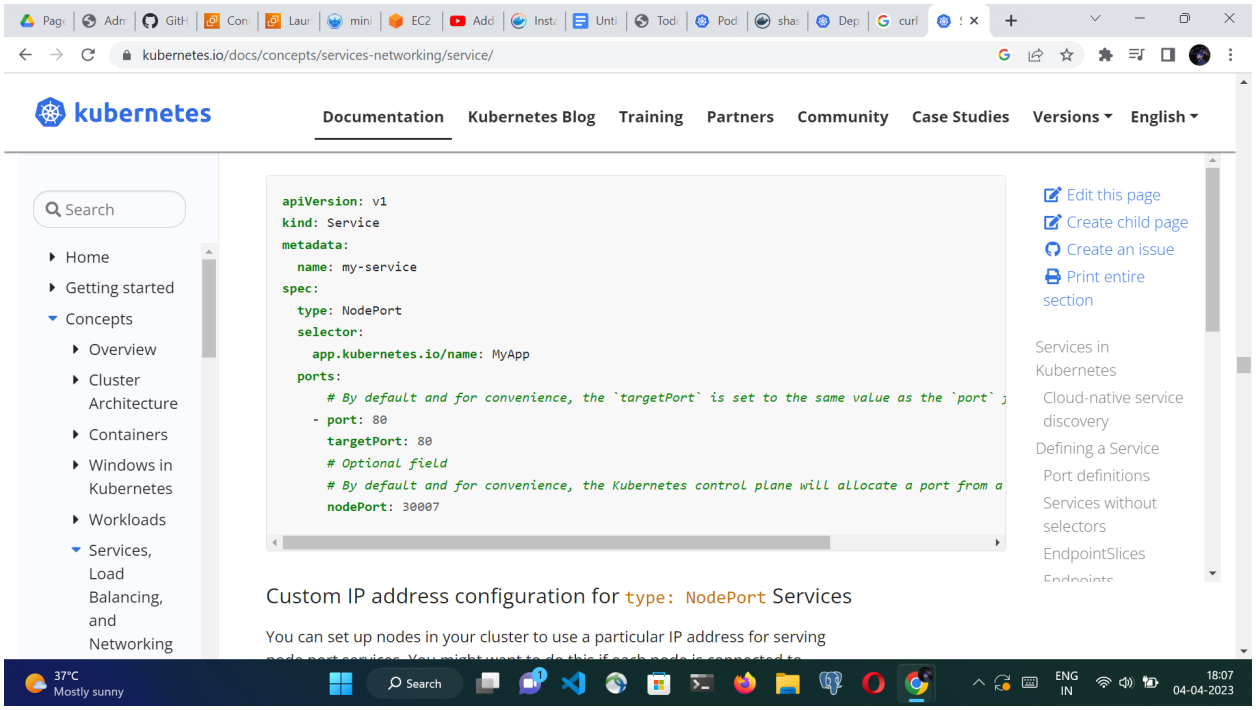
For **auto-scaling** you can chage replicas uder “deployment.yaml” to any number.



“Learn How to Manage Network And Services”

Step-8. Now curl and see if our app is running;
curl -L http://pod-ip:8000
It will not work because app is running inside cluster.

Now we will create service node port file for binding ports.



Why Service.yaml -

All the containers runs inside the cluster. So they are not exposed for outside world. There can be n number of containers so to manage all of them together we will have service.yaml. In which we creates a binding between inside(8000) to outside(80) port.

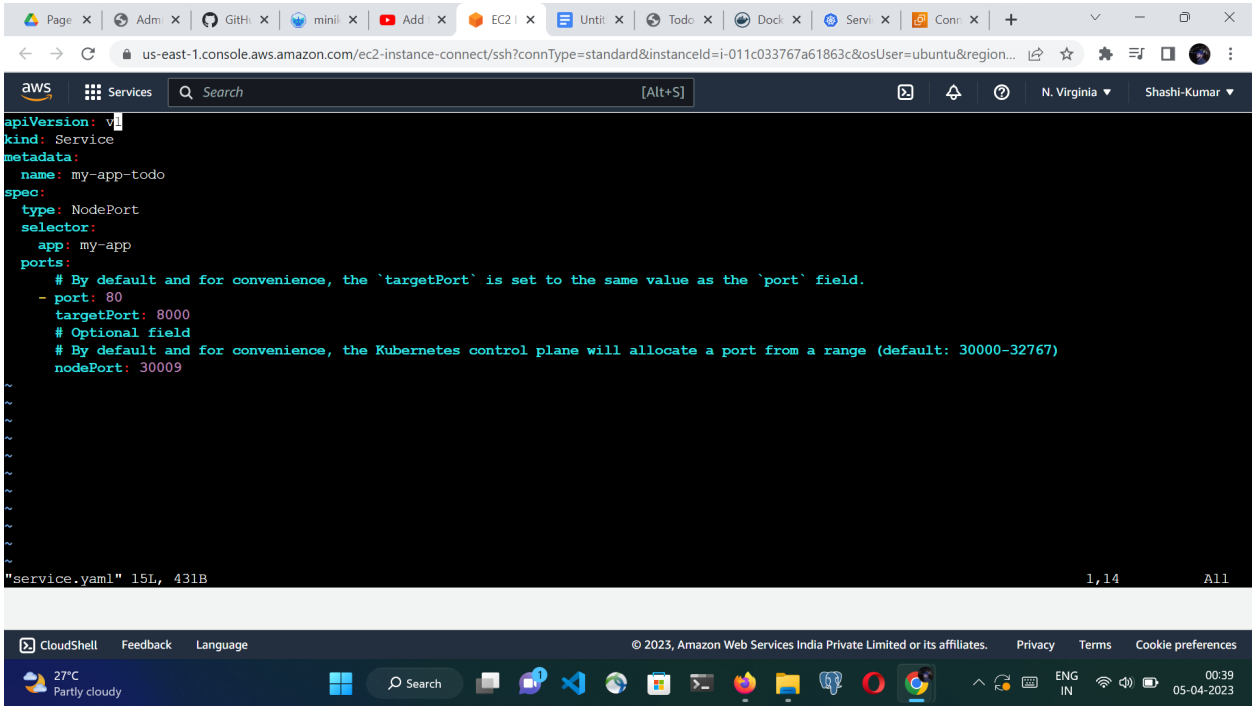
Change target port as 8000.

- vim service.yaml
- kubectl apply -f service.yaml
- kubectl get svc

Have service.yaml like this only

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-todo
spec:
  type: NodePort
  selector:
    app: my-app
  ports:
     # By default and for convenience, the `targetPort` is set to the same value as the `port` field.
    - port: 80
```

targetPort: 8000
Optional field
By default and for convenience, the Kubernetes control plane will allocate a port from a range (default: 30000-32767)
nodePort: 30009

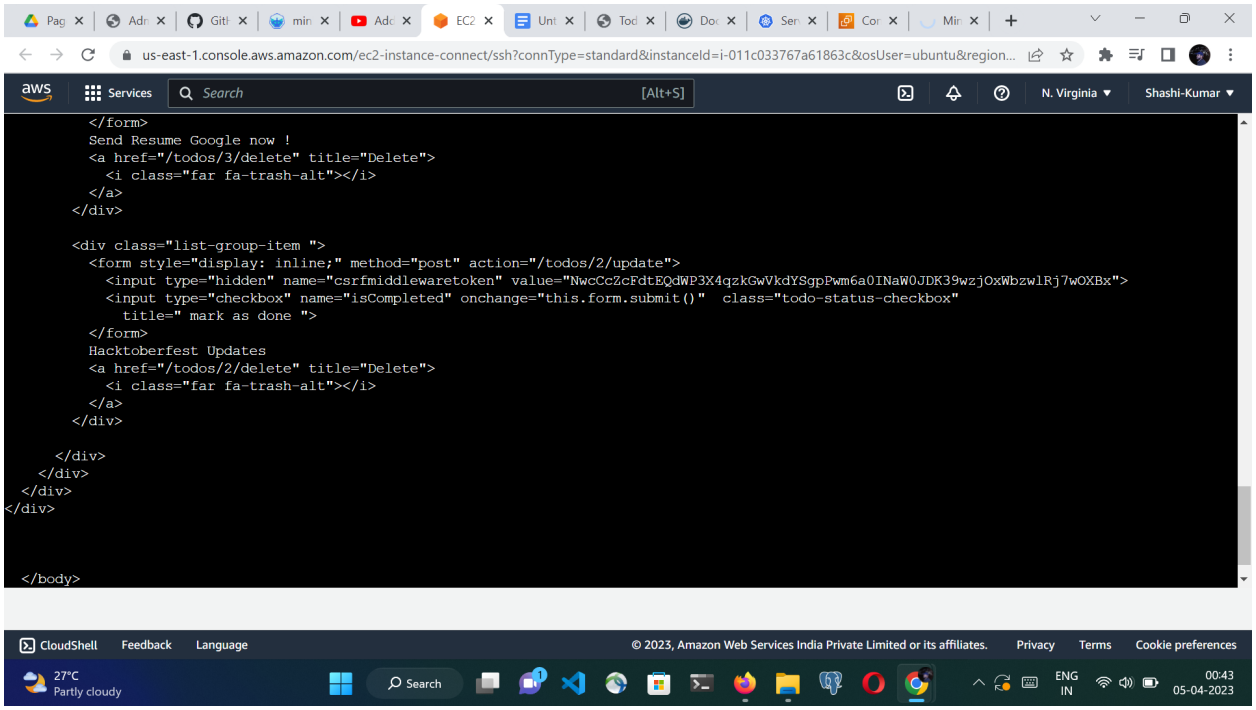


Now curl the url:
minikube service my-app-todo --url

curl -L url-from above command.

E.g curl -L http://192.168.49.2:30009

See ip app is runnning;

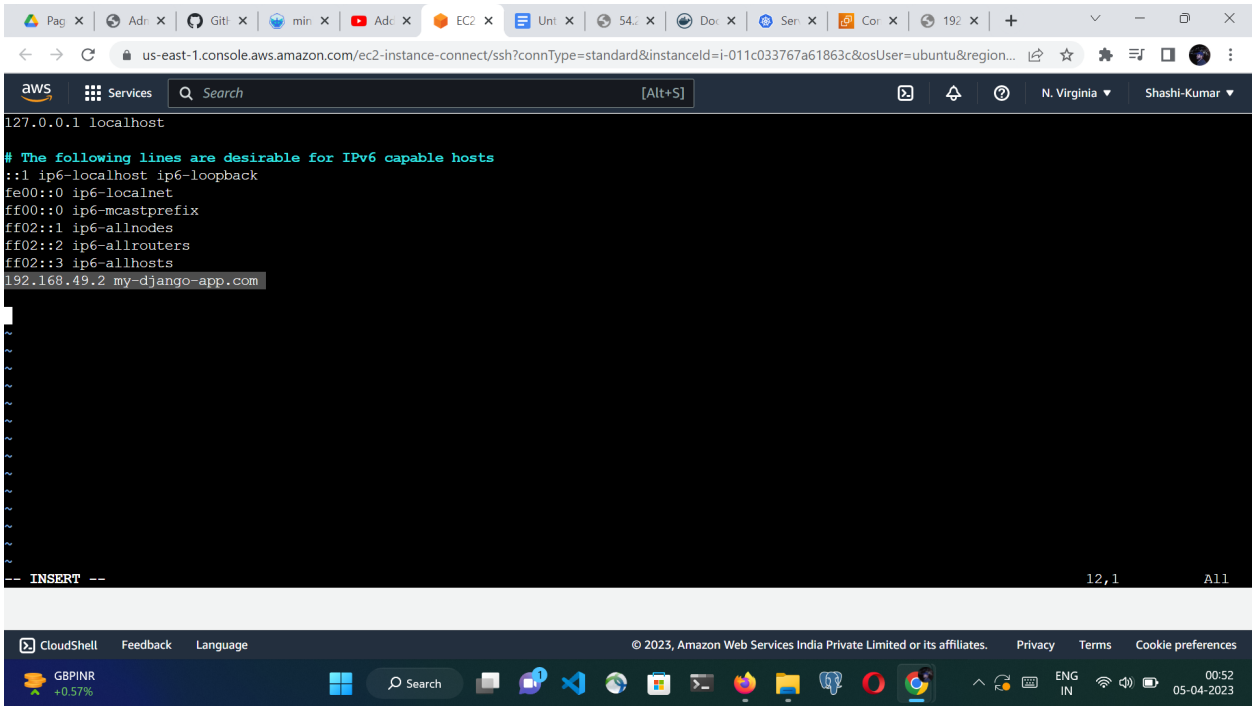


“Learn How to do Host Ip allocation through proxy”

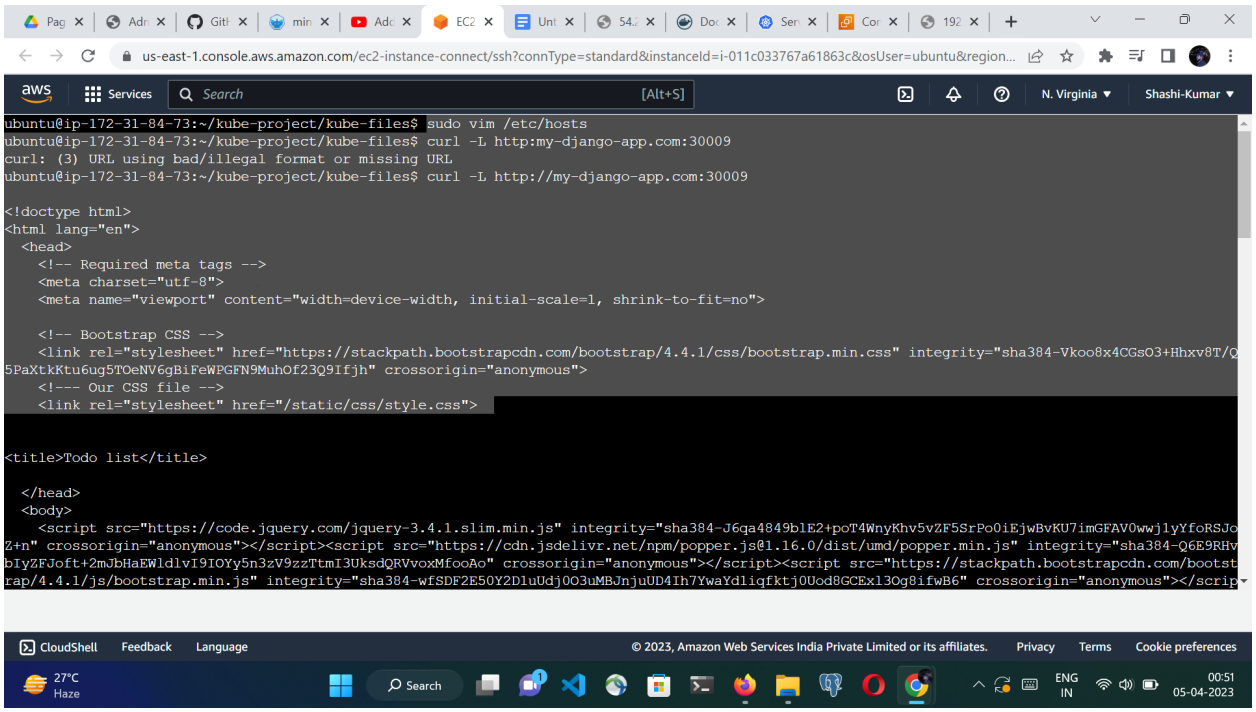
- sudo vim /etc/hosts

Add your host ip with any name of your choice.

E.g. “192.168.19.5 my-app.com”



- curl -L http://my-django-app.com:30009



Finally all done.

SUMMARY

- **“Learn How to Manage Docker Containers.”**
- **“Learn How to Manage Deployment, Replication, Auto-Healing and Auto-Scaling.”**
- **“Learn How to Manage Network And Services”**
- **“Learn How to do Host Ip allocation through proxy”**

Thank You