You may continue to install Kubernetes on the vm you already installed docker. If you are installing this on a different machine make sure docker is already installed.

Part 1

Installing kubeadm, kubelet, and kubectl:

1. Update the apt package index:

sudo apt-get update

2. Install packages needed to use the Kubernetes apt repository:

sudo apt-get install -y apt-transport-https ca-certificates curl vim git

3. Download the public signing key for the Kubernetes package repositories:

curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.28/deb/Release.key | sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg

4. Add the Kubernetes apt repository:

echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.28/deb/ /' | sudo tee /etc/apt/sources.list.d/kubernetes.list

Update the apt package index again:

sudo apt-get update

5. Install kubelet, kubeadm, and kubectl:

sudo apt-get install -y kubelet kubeadm kubectl

6. Pin installed versions of kubelet, kubeadm, and kubectl to prevent them from being accidentally updated:

sudo apt-mark hold kubelet kubeadm kubectl

7. Check installed versions:

kubectl version --client

kubeadm version

Disable Swap Space

8. Disable all swaps from /proc/swaps.

```
sudo swapoff -a

sudo sed -i.bak -r 's/(.+ swap .+)/#\1/' /etc/fstab
```

9. Check if swap has been disabled by running the free command.

```
free -h
```

Install Container runtime

10. Configure persistent loading of modules

```
sudo tee /etc/modules-load.d/k8s.conf <<EOF
overlay
br_netfilter
EOF
```

11. Load at runtime

```
sudo modprobe overlay

sudo modprobe br_netfilter
```

12. Ensure sysctl params are set

```
sudo tee /etc/sysctl.d/kubernetes.conf<<EOF

net.bridge.bridge-nf-call-ip6tables = 1

net.bridge.bridge-nf-call-iptables = 1

net.ipv4.ip_forward = 1

EOF
```

13. Reload configs

```
sudo sysctl --system
```

14. Install required packages

```
sudo apt install -y containerd.io
```

15. Configure containerd and start service

sudo mkdir -p /etc/containerd

sudo containerd config default | sudo tee /etc/containerd/config.toml

16. Configuring a cgroup driver

Both the container runtime and the kubelet have a property called "cgroup driver", which is essential for the management of cgroups on Linux machines.

sudo sed -i 's/SystemdCgroup \= false/SystemdCgroup \= true/g' /etc/containerd/config.toml

17. Restart containerd

sudo systemctl restart containerd

sudo systemctl enable containerd

systemctl status containerd

Initialize control plane

18. Make sure that the br_netfilter module is loaded:

lsmod | grep br_netfilter

Output should similar to:

```
br_netfilter        22256  0
bridge             151336  1 br_netfilter
```

19. Enable kubelet service.

sudo systemctl enable kubelet

20. Pull container images (it will take some time):

sudo kubeadm config images pull --cri-socket /run/containerd/containerd.sock

21. Bootstrap the endpoint. Here we use 10.244.0.0/16 as the pod network:

sudo kubeadm init --pod-network-cidr=10.244.0.0/16 --cri-socket /run/containerd/containerd.sock

You will see *Your Kubernetes control-plane has initialized successfully!*

22. To start the cluster, you need to run the following as a regular user (For this scenario we will only use a single host):

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

23. Check cluster info:

```
kubectl cluster-info
```

24. Install a simple network plugin.

```
wget https://raw.githubusercontent.com/flannel-io/flannel/master/Documentation/kube-flannel.yml
kubectl apply -f kube-flannel.yml
```

25. Check the plugin is working

```
kubectl get pods -n kube-flannel
```

26. Confirm master node is ready: (If you see the status as Notready, give it a around 10mins)

```
kubectl get nodes -o wide
```

27. On a master/ control node to query the nodes you can use:

```
kubectl get nodes
```

Scheduling Pods on Kubernetes Master Node

28. By default, Kubernetes Cluster will not schedule pods on the master/control-plane node for security reasons. It is recommended you keep it this way, but for test environments you need to schedule Pods on control-plane node to maximize resource usage.

```
kubectl taint nodes --all  node-role.kubernetes.io/control-plane-
```

Part 2

Create a file simple-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

To create the Pod shown above, run the following command:

```
kubectl apply -f simple-pod.yaml
```

Pods are generally not created directly and are created using workload resources. See Working with Pods

Links to an external site. for more information on how Pods are used with workload resources

Part 3

Deploying a Simple Web Application on Kubernetes

1. Create a Deployment Manifest:

A Deployment ensures that a specified number of pod replicas are running at any given time. Let's create a simple Deployment for a web application using the nginx image.

Save the following YAML to a file named webapp-deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp-deployment
  labels:
    app: webapp
spec:
  replicas: 2
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

2. Create a Service Manifest:

A Service is an abstraction that defines a logical set of Pods and enables external traffic exposure, load balancing, and service discovery. For our web application, we'll use a NodePort service.

Save the following YAML to a file named webapp-service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: webapp-service
```

```
spec:
  selector:
    app: webapp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 30080
  type: NodePort
```

3. Deploy the Application:

Apply the Deployment and Service manifests:

```
kubectl apply -f webapp-deployment.yaml
kubectl apply -f webapp-service.yaml
```

4. Verify the Deployment:

Check the status of the Deployment and Service:

```
kubectl get deployments
kubectl get services
```

You should see your webapp-deployment with 2 replicas. Give it a time to take both replicas online.

5. Access the Web Application:

Since we used a NodePort service, the web application should be accessible on node's IP at port 30080.
If you're unsure of your node IPs, you can get them with:

```
kubectl get nodes -o wide
```

Then, in a web browser (though ssh tunnel if you are in UiS cloud) or using a tool like curl, access the web application:

```
curl http://<NODE_IP>:30080
```

You should see the default nginx welcome page, indicating that your web application is running.

Part 4

Deploying WordPress and MySQL on Kubernetes

Installing dependancies:

Download rancher.io/local-path storage class:

```
kubectl apply -f https://raw.githubusercontent.com/rancher/local-path-provisioner/master/deploy/local-path-storage.yaml
```

Check with kubectl get storageclass

Make this storage class (local-path) the default:

```
kubectl patch storageclass local-path -p '{"metadata": {"annotations":{"storageclass.kubernetes.io/is-default-class":"true"}}}'
```

1. Create a PersistentVolumeClaim for MySQL:

MySQL needs persistent storage to store its data. Save the following YAML to a file named mysql-pvc.yaml:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Apply the PVC:

```
kubectl apply -f mysql-pvc.yaml
```

2. Deploy MySQL:
Save the following YAML to a file named mysql-deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
spec:
  replicas: 1
  selector:
```

```
  matchLabels:
    app: mysql
 template:
  metadata:
    labels:
      app: mysql
  spec:
    containers:
    - name: mysql
      image: mysql:5.7
      env:
      - name: MYSQL_ROOT_PASSWORD
        value: "password"
      - name: MYSQL_DATABASE
        value: "wordpress"
      ports:
      - containerPort: 3306
      volumeMounts:
      - name: mysql-persistent-storage
        mountPath: /var/lib/mysql
    volumes:
    - name: mysql-persistent-storage
      persistentVolumeClaim:
        claimName: mysql-pvc
```

Apply the Deployment:

```
kubectl apply -f mysql-deployment.yaml
```

3. Create a Service for MySQL:

This will allow WordPress to communicate with MySQL. Save the following YAML to a file named mysql-service.yaml:

```
apiVersion: v1
kind: Service
metadata:
 name: mysql
spec:
 selector:
   app: mysql
 ports:
   - protocol: TCP
     port: 3306
     targetPort: 3306
```

Apply the Service:

```
kubectl apply -f mysql-service.yaml
```

4. Deploy WordPress:
Save the following YAML to a file named wordpress-deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wordpress
spec:
  replicas: 1
  selector:
    matchLabels:
      app: wordpress
  template:
    metadata:
      labels:
        app: wordpress
    spec:
      containers:
      - name: wordpress
        image: wordpress:latest
        env:
        - name: WORDPRESS_DB_HOST
          value: mysql
        - name: WORDPRESS_DB_USER
          value: "root"
        - name: WORDPRESS_DB_PASSWORD
          value: "password"
        ports:
        - containerPort: 80
```

Apply the Deployment:

```
kubectl apply -f wordpress-deployment.yaml
```

5. Create a Service for WordPress:

This will expose WordPress to external traffic. Save the following YAML to a file named wordpress-service.yaml:

```
apiVersion: v1
kind: Service
metadata:
  name: wordpress
spec:
  selector:
    app: wordpress
  ports:
    - protocol: TCP
      port: 80
```

```
    targetPort: 80
  type: NodePort
```

Apply the Service:

```
kubectl apply -f wordpress-service.yaml
```

6. Access WordPress:

Since we used a NodePort service, WordPress should be accessible on node's IP at a dynamically allocated port above 30000.
To find the NodePort assigned to WordPress:

```
kubectl get svc wordpress
```

Then, in a web browser with the ssh tunnel, access WordPress:

```
http://< INTERNAL-IP>:<NODE_PORT>
```

Part 5

Convert your Docker deployment into a Kubernetes deployment, you may compose your own service, deployment manifests as needed. Use the docker images you used previously when creating the pods/deployments.

Additional ref: https://kubebyexample.com/