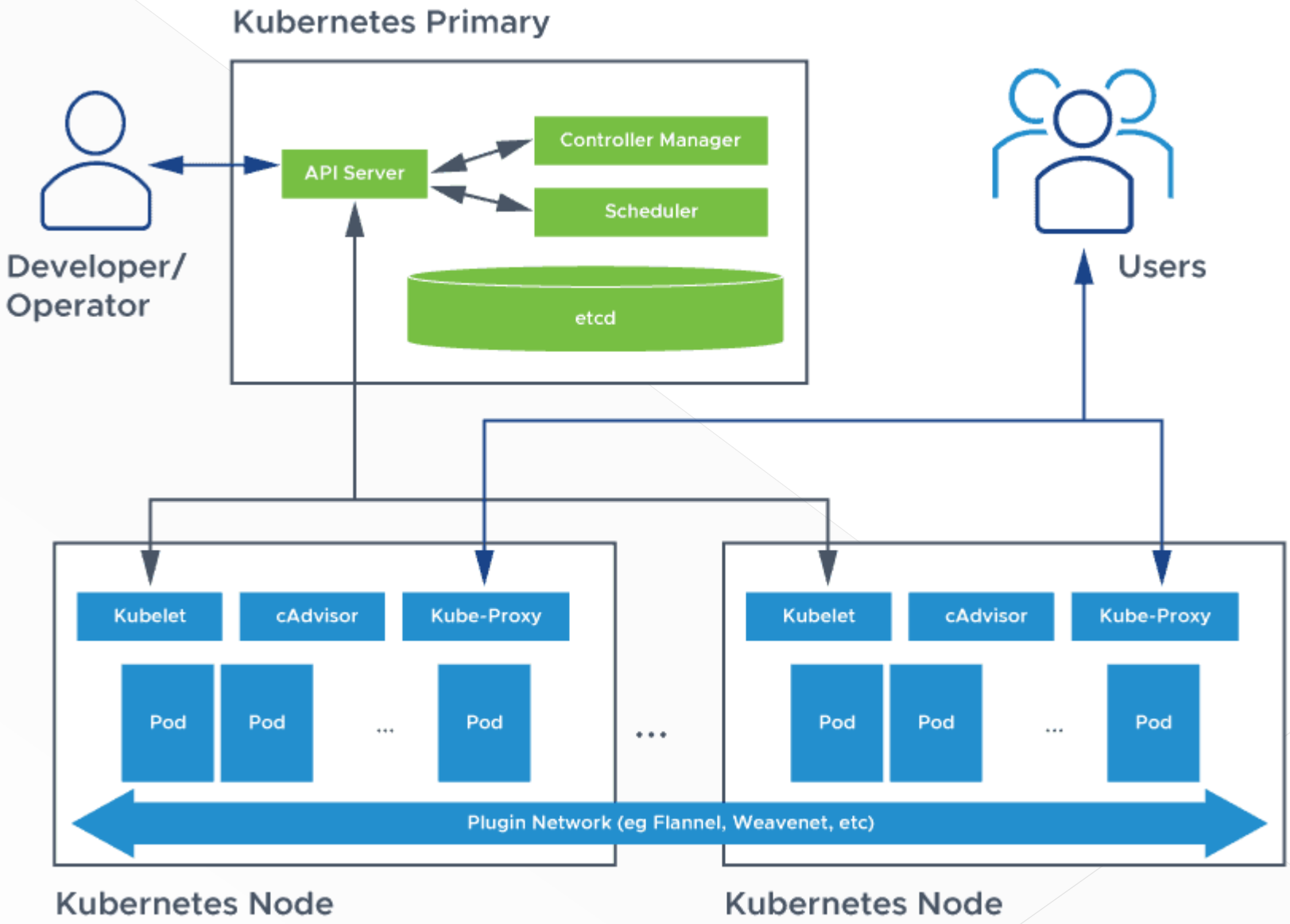
Introduction to Orchestration

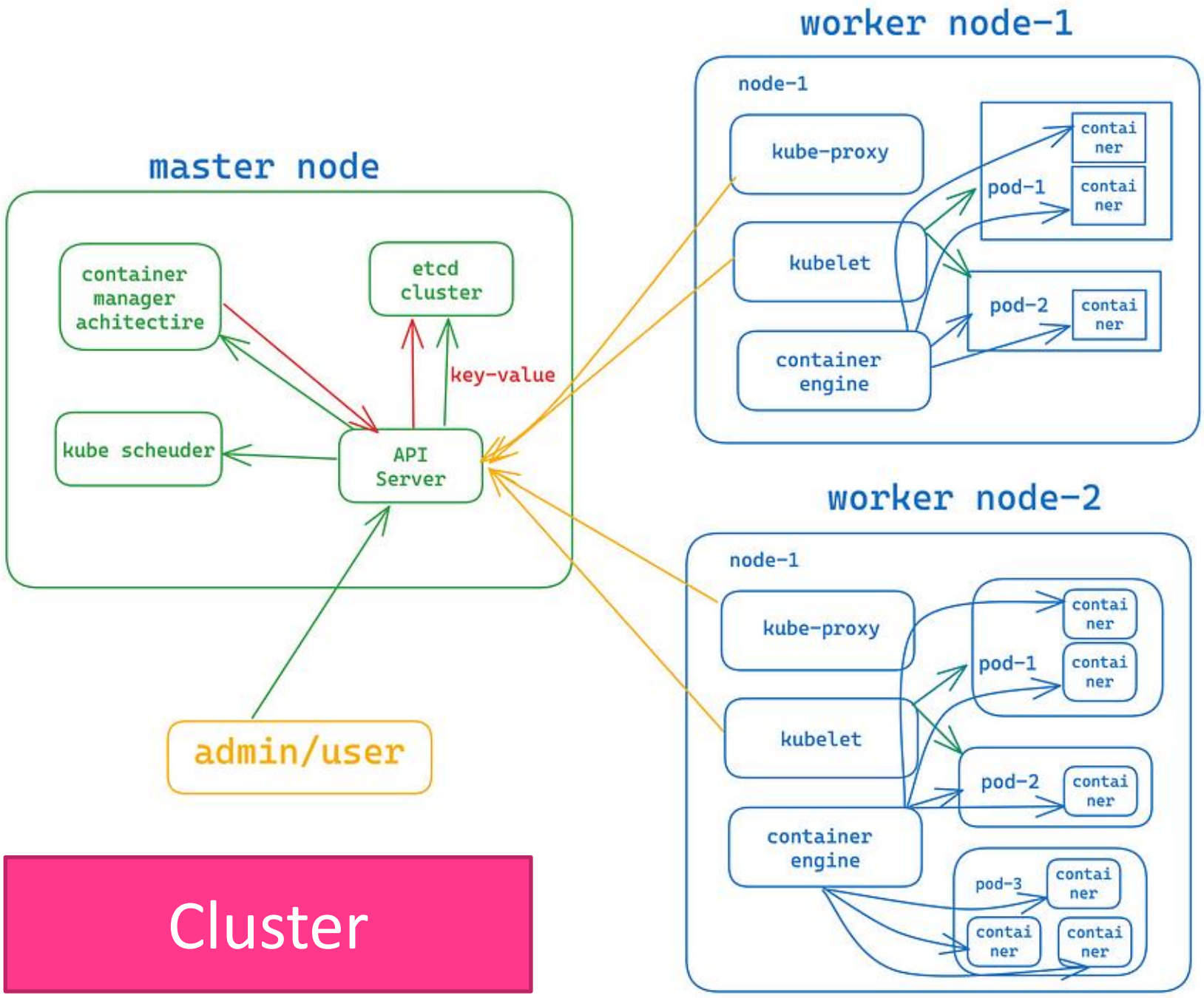
Kubernetes

Kubernetes

- Kubernetes, an open-source container orchestration platform, is designed to automate the deployment, scaling, and operation of application containers.

Kubernetes Architecture




worker node-1
node-1
kube-proxy
kubelet
container engine
pod-1
pod-2
contai ner
contai ner
contai ner
contai ner
master node
container manager achitectire
etcd cluster
key-value
kube scheuder
API Server
admin/user
worker node-2
node-1
kube-proxy
kubelet
container engine
pod-1
pod-2
pod-3
contai ner
contai ner
contai ner
contai ner
contai ner
contai ner
Cluster

Key Components

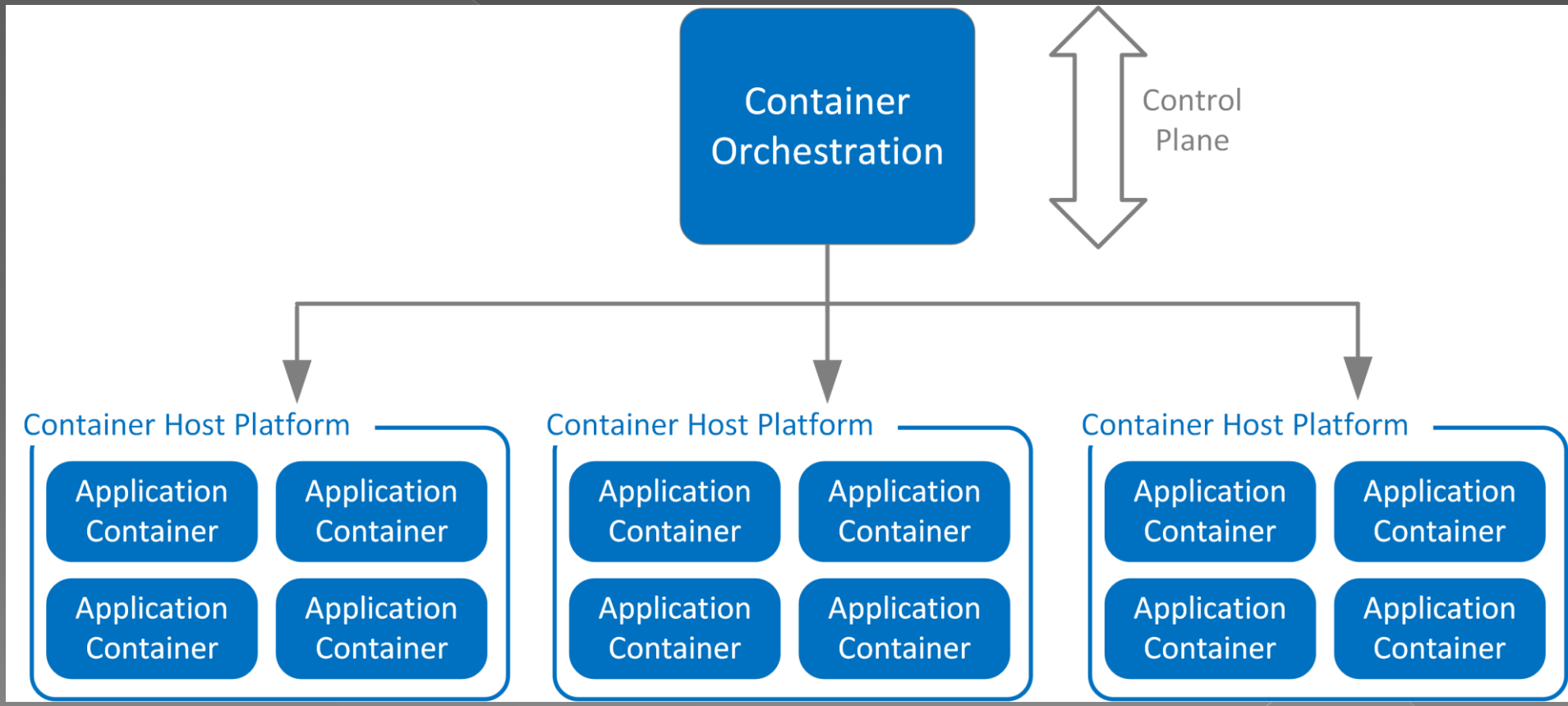
Master Node: Controls the cluster, responsible for the management of the Kubernetes cluster.

- **API Server**: The front-end for the Kubernetes control plane. It exposes the Kubernetes API, acting as the main management point for the cluster.
- **Scheduler**: Assigns workloads to specific nodes based on resource availability and policies.
- **Controller Manager**: Runs controller processes to regulate the state of the cluster, ensuring that the desired state matches the current state.
- **etcd**: A consistent and highly-available key-value store used as Kubernetes' backing store for all cluster data.

Worker Nodes: Run containerized applications. Each node has the necessary services to run pods and is managed by the master node.

- **Kubelet**: An agent that ensures containers are running in a pod. It communicates with the master node and manages the containers' lifecycle.
- **Kube-proxy**: Maintains network rules on nodes, enabling communication to and from pods within the cluster.
- **Container Runtime**: manages and runs components required to run containers
- Pod: Represents a single instance of an application

Container Orchestration



Kubernetes API

- The Kubernetes API is a powerful interface that allows users to interact with and manage their Kubernetes clusters.
- It provides a RESTful interface for querying and manipulating the state of various Kubernetes objects, such as pods, services, deployments, and more.

Key Features of Kubernetes API

- **Resource-Oriented Design**:
  - The Kubernetes API is structured around resources, such as Pods, Services, Nodes, ConfigMaps, and more.
  - Each resource has a corresponding URL endpoint, allowing for operations like create, read, update, and delete (CRUD).
- **RESTful Endpoints**:
  - The API uses standard HTTP verbs: GET (retrieve), POST (create), PUT (update), PATCH (partially update), DELETE (remove).
  - URLs are structured in a hierarchical format, e.g., /api/v1/namespaces/{namespace}/pods.

Key Features of Kubernetes API

- **API Versions**:
  - The Kubernetes API is versioned to maintain compatibility. Common versions include v1, v1beta1, and v1alpha1.
  - Different API groups (e.g., core, apps, batch) may have different versions.

Common API Resources

- **Pods**:
  - Smallest and simplest Kubernetes object, representing a single instance of a running process.
- **Services**:
  - Abstract way to expose an application running on a set of Pods as a network service.
- **Deployments**:
  - Provide declarative updates to applications, managing ReplicaSets to ensure the desired number of Pods are running.

Common API Resources

- **ConfigMaps**:
  - Store configuration data in key-value pairs.
- **Secrets**:
  - Store sensitive data, such as passwords, OAuth tokens, and SSH keys.
- **Nodes**:
  - Represent a worker node in the cluster.

Interacting with the API

- **kubectl**:
  - The primary command-line tool for interacting with the Kubernetes API.
  - Commands like kubectl get, kubectl create, kubectl delete, etc., interact with the API to manage resources.

Interacting with the API

- **Client Libraries**:
  - Kubernetes provides client libraries for different programming languages (e.g., Go, Python, Java, JavaScript).
  - These libraries wrap the RESTful API calls, making it easier to programmatically interact with Kubernetes.
- **Direct HTTP Calls**:
  - Users can make direct HTTP requests to the API server endpoints.
  - Authentication is required, typically using tokens, certificates, or other supported methods.

Creating and managing Cluster

- Open portal.azure.com
- Search for kubernates service
- Create one kubernates cluster in US WEST 2 zone
- Give the clustername like cluster1
- Once the deployment is ready you can click on connect and connect either using powershell or bash

Check basic commands

- On the master node, enter the following command to review cluster information:
  - kubectl cluster-info
- On the master node, enter the following command to view complete cluster information
  - kubectl cluster-info dump
- Use the following command to create a namespace:
  - kubectl create namespace firstnamespace

- Confirm the creation of the new namespace with the following command:
  - kubectl get namespaces
- To view the cluster configuration, use the command below:
  - kubectl config view
- Run the following command to view the current cluster:
  - kubectl config current-context
- To identify the API server, execute and copy the 127.0.0.1:8080 port as shown below:
  - kubectl proxy --port=8080

Activity: Managing working of Nodes

- List all the running nodes in a cluster using the following command:
  - kubectl get nodes
- Verify the status of the worker node you wish to inspect by running the following command:
  - kubectl describe node worker-node-1.example.com
- Use the following command to delete a worker node:
  - kubectl delete node worker-node-1.example.com
  - Again check the running nodes

Create Deployment in Cluster

- kubectl create deployment nginx --image=nginx
- kubectl expose deployment nginx --port=80 --type=LoadBalancer
- kubectl get service nginx

```
PS /home/sfj> kubectl get nodes
NAME                                STATUS   ROLES   AGE   VERSION
aks-agentpool-62348252-vmss000000   Ready    agent   11m   v1.28.9
aks-agentpool-62348252-vmss000001   Ready    agent   11m   v1.28.9
PS /home/sfj> kubectl create deployment nginx --image=nginx
deployment.apps/nginx created
PS /home/sfj> kubectl expose deployment nginx --port=80 --type=LoadBalancer
service/nginx exposed
PS /home/sfj> kubectl get service nginx
NAME    TYPE           CLUSTER-IP    EXTERNAL-IP     PORT(S)        AGE
nginx   LoadBalancer   10.0.188.58   172.179.147.5   80:31882/TCP   15s
PS /home/sfj>
```

Configuring Pods in Cluster

- Create YAML file: vi pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: nginx:latest
    ports:
    - containerPort: 80
```

- Apply the updated Pod Definition YAML file:
  - kubectl apply -f my-pod.yaml
- Verify the Pod is running:
  - kubectl get pods

Kubernates Service

- To access the Nginx container running in your AKS cluster, you can expose it using a Kubernetes service.
- Create a file named my-service.yaml

```
PS /home/sfj> cat my-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: LoadBalancer
  selector:
    app: my-nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

Modify the pod to match service

- Update my-pod.yaml to include a label that matches the selector in the service.

```
PS /home/sfj> cat my-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
  labels:
        app: my-nginx
spec:
  containers:
  - name: my-container
    image: nginx:latest
    ports:
    - containerPort: 80
```

- Apply the updated Pod Definition YAML file
  - kubectl apply -f my-pod.yaml
- Deploy the service to expose the pod:
  - kubectl apply -f my-service.yaml
- Check service status:
  - kubectl get services
- Copy external Ip address and check in browser for running service

```
PS /home/sfj> kubectl get services
NAME          TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)          AGE
kubernetes    ClusterIP      10.0.0.1       <none>           443/TCP          38h
my-service    LoadBalancer   10.0.49.202    172.179.99.34    80:31060/TCP     35s
nginx         LoadBalancer   10.0.188.58    172.179.147.5    80:31882/TCP     38h
```

Running Containers in Pods

- A Pod is the smallest and simplest Kubernetes object.
- It represents a single instance of a running process in your cluster.
- Pods are designed to host one or more containers that share the same network namespace, storage, and specifications.
- You run your application containers inside Pods.
- Each Pod has its own IP address, and containers within a Pod can communicate with each other using localhost.
- Pods can be created, destroyed, and recreated by Kubernetes, especially if part of a ReplicaSet, Deployment, or other higher-level abstraction.

Running Services in Pods

- A Service in Kubernetes is an abstraction that defines a logical set of Pods and a policy by which to access them.
- Services provide stable network endpoints to Pods, allowing you to expose your applications internally or externally.
- Services are used to expose Pods to other Pods within the cluster, or to external users.
- They can load balance traffic across multiple Pods.
- Kubernetes supports several types of Services, including
  - ClusterIP
  - NodePort
  - LoadBalancer

Practical Use Case

Pods:

- You deploy an application by creating Pods.
- For example, deploying an Nginx server involves creating a Pod with an Nginx container.

Services:

- You expose the Nginx application to the internet or within the cluster by creating a Service that targets the Nginx Pod.

Let’s Summarize

- Pods run the actual application containers and manage their lifecycle
- while Services provide stable network endpoints to access those Pods, handle load balancing, and expose the application to other components or external clients.

Deployment

- A Deployment in Kubernetes is a higher-level abstraction that manages the lifecycle of Pods and ReplicaSets.

Deployment Features

- **Declarative Updates:** You declare the desired state of your application, and the Deployment controller makes the necessary changes to achieve that state.
- **Scaling:** Deployments allow you to scale the number of replicas of your application up or down.
- **Rolling Updates:** You can update your application with zero downtime by rolling out updates incrementally.
- **Rollbacks:** If an update causes issues, you can roll back to a previous stable state.
- **Self-Healing:** If a Pod fails, the Deployment controller replaces it automatically.

Creating and Configuring a Deployment

```
PS /home/sfj> cat deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.17.4
        ports:
        - containerPort: 80
```

- Create a Deployment YAML File:
- Nano deployment.yaml

- Apply the Deployment:
  - kubectl apply -f deployment.yaml
- Verify the Deployment:
  - kubectl get deployments
  - kubectl get pods
- Deployment entire Description
  - kubectl describe deployment nginx-deployment
- Scaling the Deployment:
  - kubectl scale deployment/nginx-deployment --replicas=5
  - You can see the 5 replica running in deployment

Update Deployment

```
PS /home/sfj> cat deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.18.0 #Updated Image
        ports:
        - containerPort: 80
```

- changing the container image
- modify the YAML file: nano deployment.yaml

- Apply the updated YAML file:
  - kubectl apply -f deployment.yaml
- Rolling Back a Deployment:
  - kubectl rollout undo deployment/nginx-deployment
- Monitoring the deployment
  - kubectl rollout status deployment/nginx-deployment
- Check the deployment description which will show the version rollbacked
  - kubectl describe deployment nginx-deployment

Create Deployment direct with image

- Create an Nginx deployment
- kubectl create deployment nginx --image=nginx
- Expose the deployment as a service
  - kubectl expose deployment nginx --port=80 --type=LoadBalancer
- Get the service details to find the ExternalIP
  - kubectl get services
  - Copy paste the IP in browser and it will work

Activity: Create yaml file using command

- Create Yaml file
  - kubectl create deployment myhttpd --image=docker.io/httpd --dry-run=client -o yaml > myapp1.yaml
- Edit it you want update
  - Nano myapp1.yaml
- Create Deployment
  - kubectl apply -f myapp1.yaml
- Check the Deployments
  - Kubectl get deployments
  - Kubectl get pods

- Expose the Service
  - kubectl expose deployment myhttpd --port=8080
- Check service:
  - kubectl get svc
- Describe the service
  - kubectl describe svc myhttpd

Activity: Create Deployment in namespace

- Create Namespace
  - kubectl create namespace mynamespace
- Verify namespace
  - kubectl get namespace
- Create deployment in namespace
  - kubectl create deployment myapp1 --image=docker.io/httpd -n mynamespace
- Get deployment of namespace
  - kubectl get deployment -n mynamespace

Activity: Create Deployment in namespace

- Scale and verify the deployment by using the following commands:
  - kubectl scale --replicas=3 deployment myapp1 -n mynamespace
  - kubectl get deployment -n mynamespace
- Retrieve the endpoints by using the following commands:
  - kubectl get endpoints
  - kubectl describe endpoints
- Delete the deployment
  - kubectl delete deployment myapp1

Activity: Create Deployment in namespace

- Delete the Service:
  - kubectl delete svc myapp1
- Verify the deleted service
  - kubectl get svc
- Delete namespace
  - kubectl delete namespace mynamespace
- Verify events
  - kubectl get events
- Verify the nodes state
  - kubectl get nodes
- Describe the node configuration
  - kubectl describe node worker-node-1.example.com

Create Node

- Create node config file
  - vi nodereg.json

```
{
 "kind": "Node",
 "apiVersion": "v1",
 "metadata": {
 "name": "worker-node-1.example.com",
 "labels": {
 "name": "firstnode"
 }
 }
}
```

Create Node

- Apply file for node creation:
  - kubectl create -f ./nodereg.json
- Check created node
  - kubectl get nodes
- To view node conditions, status
  - kubectl describe node worker-node-1.example.com

Taints

- In Kubernetes, a "taint" is a property you can add to a node that prevents it from accepting any pods that do not tolerate the taint.
- Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes.

Three Components of taints

- **Key**: Identifies the taint.
- **Value**: Provides additional context or specifics for the taint.
- **Effect**: Specifies the action to take when a pod without the corresponding toleration is scheduled on the node. The possible effects are:
  - NoSchedule: Pods that do not tolerate this taint are not scheduled on the node.
  - PreferNoSchedule: Kubernetes will try to avoid scheduling pods that do not tolerate the taint on the node, but it is not a hard requirement.
  - NoExecute: Pods that do not tolerate this taint will be evicted from the node if they are already running there.

Taints

- Taints and tolerations work together to ensure that pods are not scheduled onto inappropriate nodes. One or more taints are applied to a node; this marks that the node should not accept any pods that do not tolerate the taints
- kubectl taint nodes node1 key1=value1:NoSchedule (Add taint)
- kubectl taint nodes node1 key1=value1:NoSchedule- (Remove Taint)
- Get taints on Node:
- kubectl get nodes -o json | jq '.items[] | {name: .metadata.name, taints: .spec.taints}'

Understanding key value

- When you create a taint, you can choose any key and value that make sense for your use case. The key typically represents the attribute you want to mark, and the value can provide additional context.
- kubectl taint nodes <node-name> gpu=true:NoSchedule
  - **key**: gpu
  - **value**: true
  - **effect**: NoSchedule

Activity: Launch a pod an establish service to pod

```
PS /home/sfj> vi mydeployment.yaml
PS /home/sfj> cat mydeployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-deployment
spec:
 selector:
 matchLabels:
 app: httpd
 replicas: 2
 template:
 metadata:
 labels:
 app: httpd
 spec:
 containers:
 - name: httpd
 image: httpd:latest
 ports:
 - containerPort: 80
```

- kubectl apply -f mydeployment.yaml
- kubectl get deploymen
- kubectl get pods

```
PS /home/sfj> vi myservice.yaml
PS /home/sfj> cat myservice.yaml
apiVersion: v1
kind: Service
metadata:
 name: myservice
spec:
 selector:
 app: httpd
 ports:
 - protocol: TCP
 port: 8080
 targetPort: 80
```

- kubectl apply -f myservice.yaml
- kubectl describe svc myservice
- kubectl get pods -l app=httpd
- kubectl get endpoints myservice

Backing Up and Restoring Etcd Cluster Data

- Install the etcd-client using the command below:
  sudo apt install etcd-client
- List all the pods in the kube-system namespace
  kubectl get pods -n kube-system
- Describe the etcd pod in the kube-system namespace and note the IP
  - kubectl describe pods -n kube-system
- After noting the client URL, set it as an environment variable and confirm its value
  - export advertise_url= <type client url>
  - echo $advertise_url

- Back up the etcd data
  - sudo ETCDCTL_API=3 etcdctl \
- Check the presence of the newly created etcd_backup.db file:
  - ls
- Run the following command to verify the etcd backup:
  - sudo ETCDCTL_API=3 etcdctl \

Restore Data

- Restore the etcd cluster data
  - sudo ETCDCTL_API=3 etcdctl \
- Set the proper ownership for the new data directory
  - stat -c %U:%G /var/lib/etcd
  - sudo chown -R root:root /var/lib/etcd
- Confirm the state of the cluster
  - sudo ETCDCTL_API=3 etcdctl endpoint health \

Configuring DNS for Kubernetes Services and Pods

- To identify the core DNS deployment
  - kubectl get deploy coredns -n kube-system
- To identify the coredns pods using selector:
  - kubectl get pods -l k8s-app=kube-dns -n kube-system
- To identify the coredns service
  - kubectl get svc kube-dns -n kube-system
- get endpoints:
  - kubectl get endpoints kube-dns -n kube-system

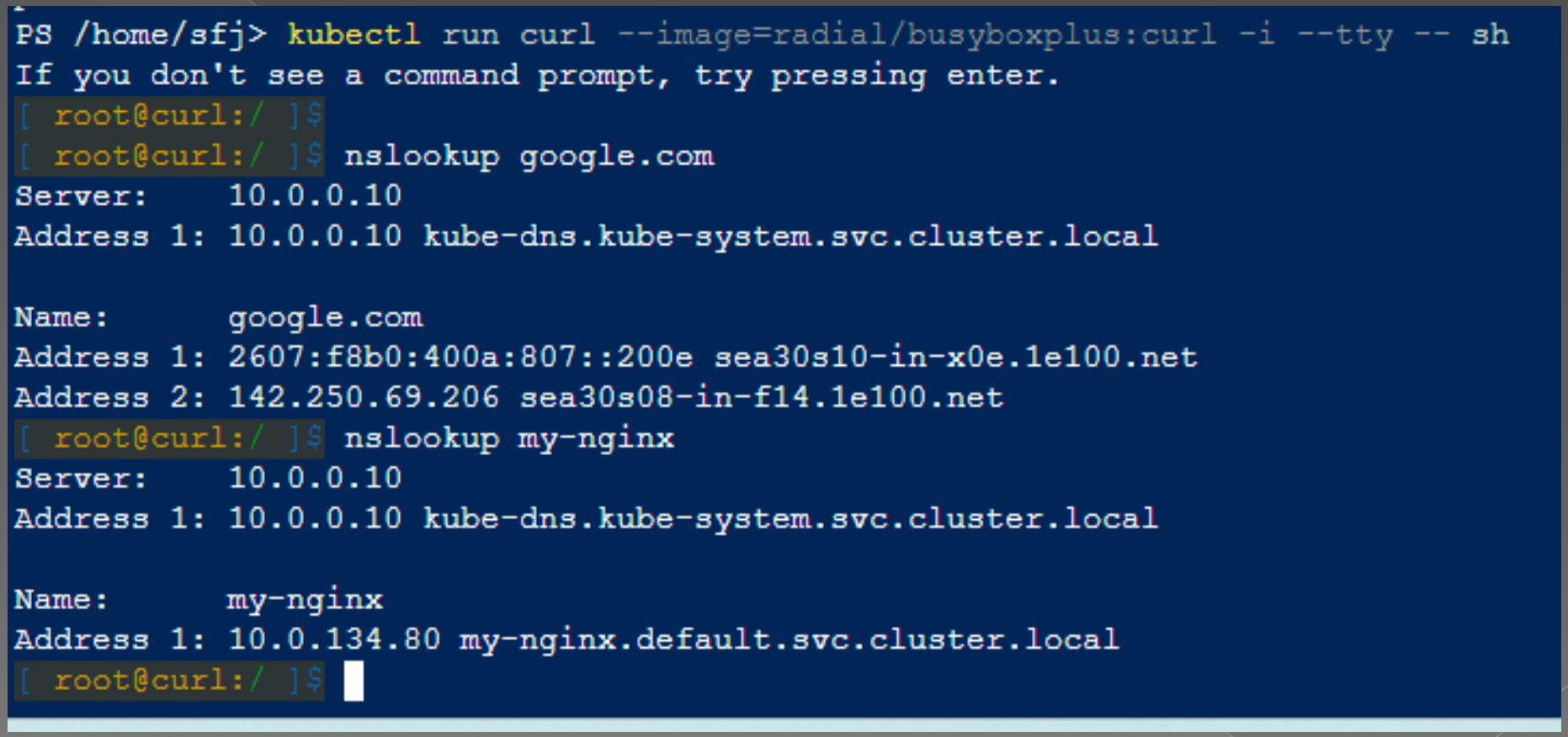
- describe the endpoints:
- kubectl describe endpoints kube-dns -n kube-system
- Execute DNS query: vi nginx.yaml

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

- kubectl apply -f nginx.yaml
- kubectl get deploy my-nginx
- kubectl get pods -l run=my-nginx
- Create Service: vi my-nginx-service.yaml
- kubectl apply -f my-nginx-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 80
    nodePort: 30000
  selector:
    run: my-nginx
```

- kubectl get svc my-nginx
- kubectl get ep my-nginx
- Create Curl pod to perform DNS query
- kubectl run curl --image=radial/busyboxplus:curl -i --tty -- sh
- nslookup google.com
- nslookup my-nginx
- nslookup my-nginx.default.svc.cluster.local (local cluster lookup)
- curl my-nginx (you can see the entire code)
- exit


PS /home/sfj> kubectl run curl --image=radial/busyboxplus:curl -i --tty -- sh
If you don't see a command prompt, try pressing enter.
[root@curl:/]$
[root@curl:/]$ nslookup google.com
Server: 10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local
Name: google.com
Address 1: 2607:f8b0:400a:807::200e sea30s10-in-x0e.1e100.net
Address 2: 142.250.69.206 sea30s08-in-f14.1e100.net
[root@curl:/]$ nslookup my-nginx
Server: 10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local
Name: my-nginx
Address 1: 10.0.134.80 my-nginx.default.svc.cluster.local
[root@curl:/]$

Configure DNS (vi dnspolicy.yaml)

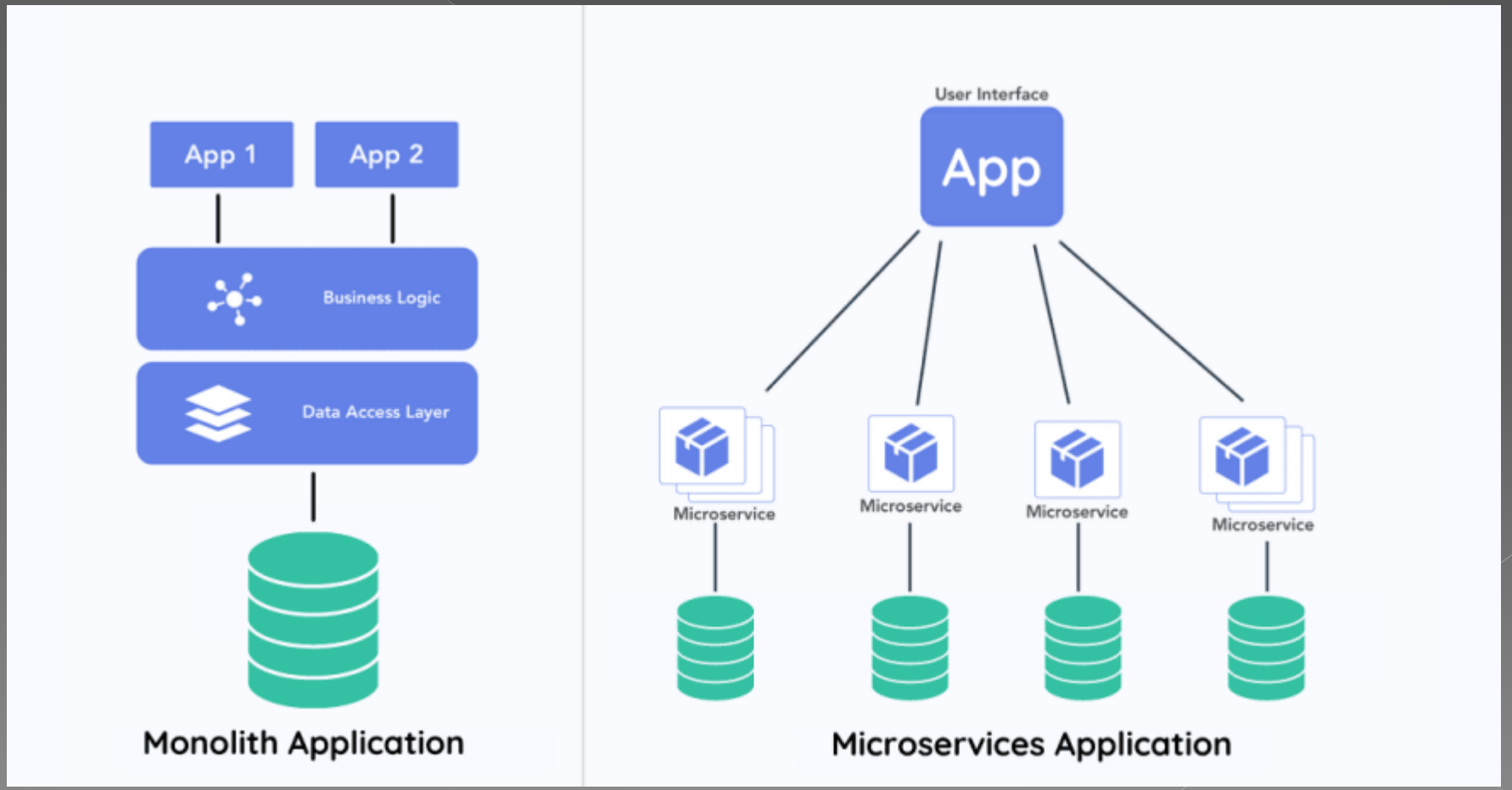
```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
  - name: busybox
    image: busybox:1.28
    command:
    - sleep
    - "3600"
    imagePullPolicy: IfNotPresent
  restartPolicy: Always
  hostNetwork: true
  dnsPolicy: ClusterFirstWithHostNet
```

- kubectl apply -f dnspolicy.yaml
- kubectl get pods
- kubectl describe pod busybox
- Create a configuration YAML file:
- vi dnsconfig.yaml
- kubectl apply -f dnsconfig.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: dnscustomconfig
  namespace: default
spec:
  containers:
  - name: test
    image: nginx
  dnsPolicy: None
  dnsConfig:
    nameservers:
    - 1.2.3.4
    searches:
    - ns1.svc.cluster-domain.example
    - my.dns.search.suffix
    options:
    - name: ndots
      value: "2"
    - name: edns0
```

- Set up the IPv6 for the DNS connectivity:
- kubectl exec -it dnscustomconfig -- cat /etc/resolv.conf

Monolith Vs Microservice



Microservices