Kubernetes

Part 2

Kubernetes

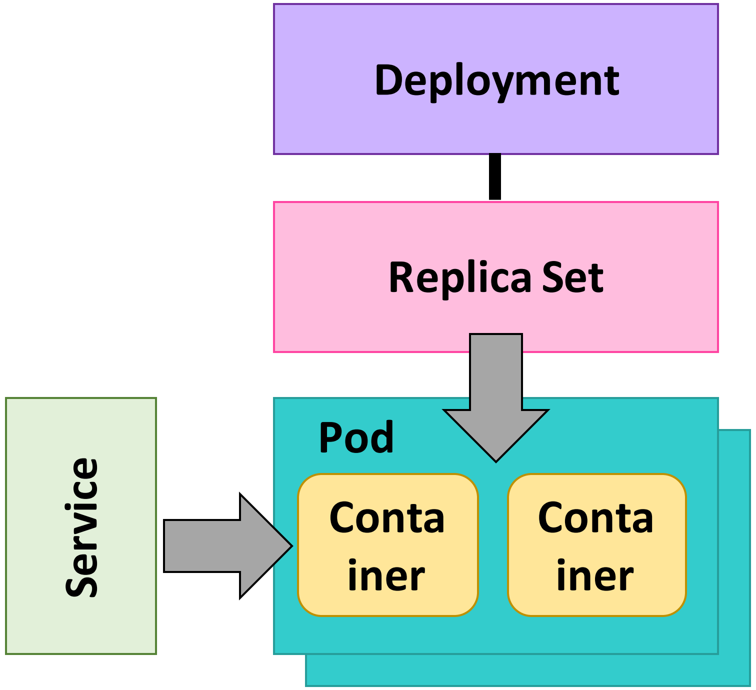


Image idea from https://www.slideshare.net/imesh/deep-dive-into-kubernetes-part-1

Volumes

- Volumes are storage that are shared by containers in a Pod
  - Allocated by the Pod, usually a shared directory in the Pod
  - Not visible outside of Pod
- Tied to the lifecycle of a Pod viz. its removed when the Pod is delete
  - Unlike Docker volumes where they are durable
- Different types of volumes
  - Eg. hostPath, NFS, iSCSI, fibre channel, empty directory, etc.
- **`hostPath`** and **`emptyDir`** type is good for sharing data between containers in a Pod
  - Eg. The example of file puller and web server

Defining a Volume

Same syntax for creating for Pod templates in deployment objects

```
kind: Pod
...
spec:
  volumes:
    - name: data-volume
      hostPath:
        path: /data
        type: DirectoryOrCreate
  containers:
    - name: webserver-pod
      volumeMounts:
        - name: data-volume
          mountPath: /var/www/html
          ...
```

Specify the volumes in this Pod

Volume’s name

Directory in the node’s filesystem

Type of path

Configuration details of this volume

Specify where to mount the volume(s)

Mount details

Using ConfigMaps

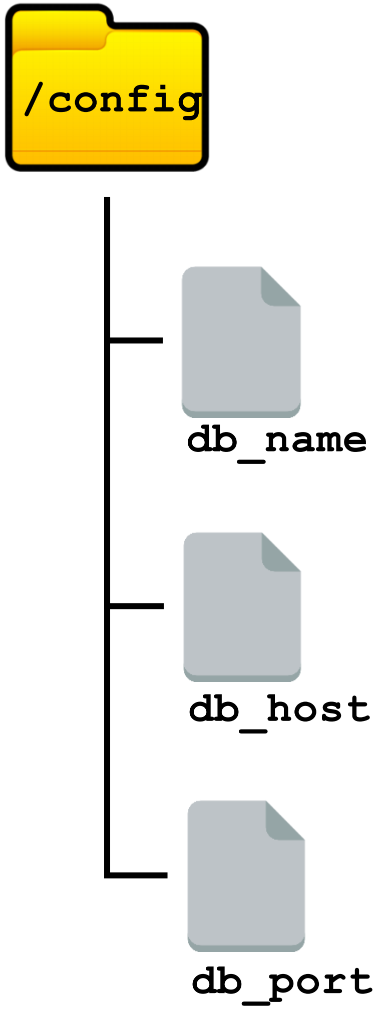
Injecting as environment variables

```
containers:
  ...
  env:
    - name: DB_NAME
      valueFrom:
         configMapKeyRef:
           name: myapp-config
           key: db_name
    - name: DB_HOST:
      valueFrom:
         configMapKeyRef:
           name: myapp-config
           key: db_host
```

Mounting as a volume

```
volumes:
  - name: config-volume
    configMap:
       name: myapp-config

containers:
  ...
  volumeMounts:
    - name: config-config
    - mountPath: /config
```



Kubernetes

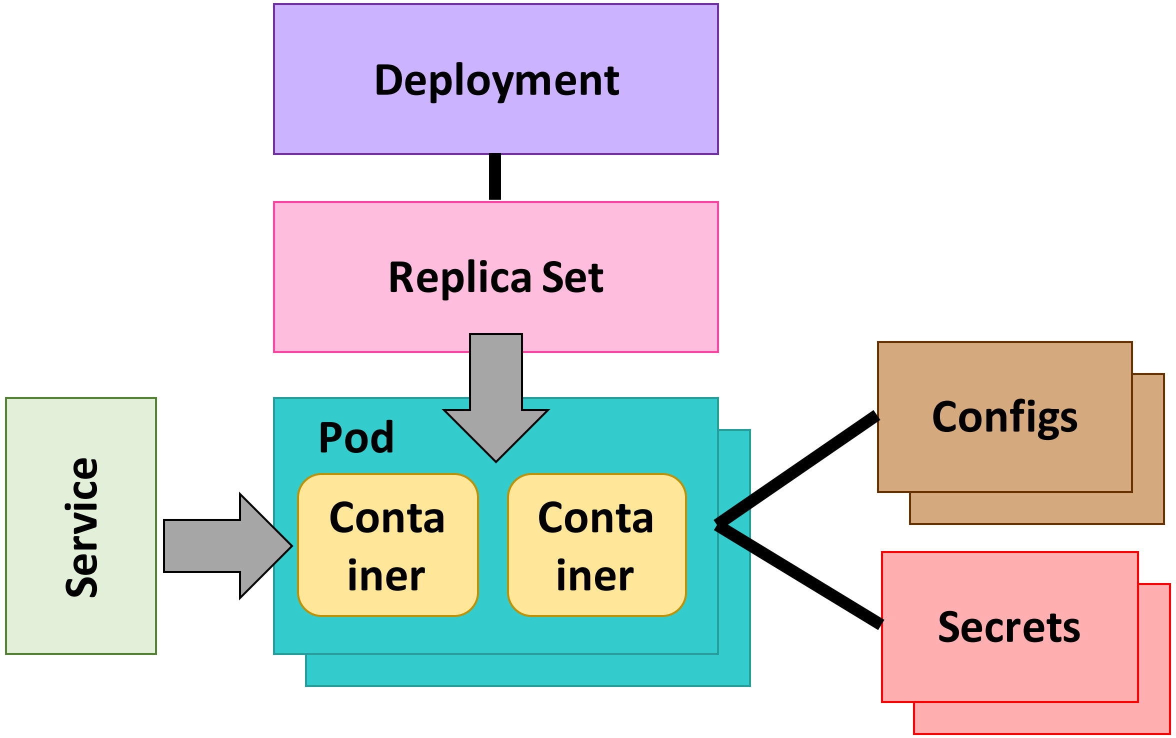


Image idea from https://www.slideshare.net/imesh/deep-dive-into-kubernetes-part-1

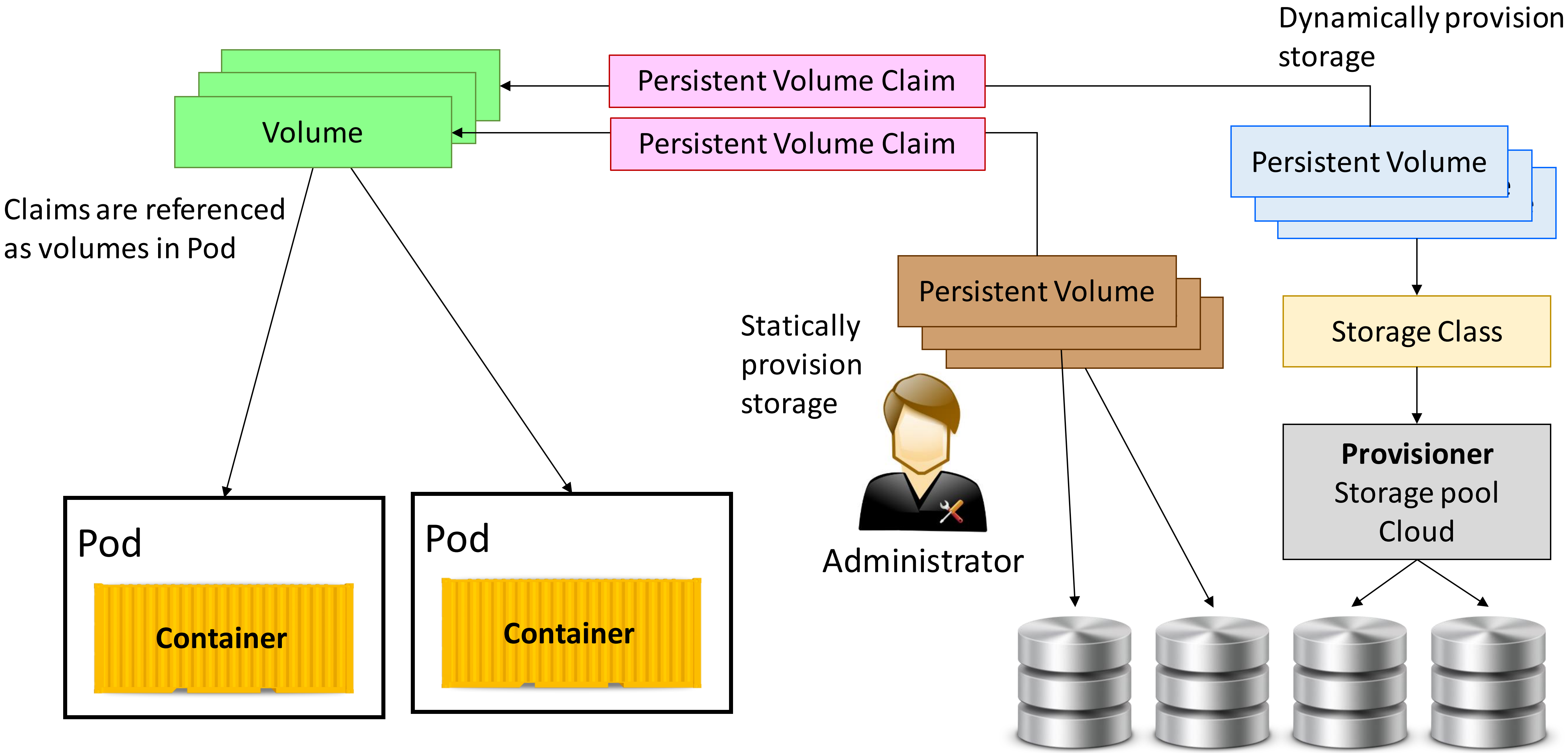
Persistent Storage

- Kubernetes can dynamically provision storage
  - Eg. User ask for 50GB volume to caching images
- Kubernetes allows storage to be either statically or dynamically provisioned
  - Static provision - an administrator will need to first provision the storage manually
  - Dynamic provision - the user describes the type of storage that is required; Kubernetes will attempt to provision based on the user's requirements
- Once a persistent storage has been allocated and claimed/reserved, a Pod can mount the volume like any regular volume
- Persistent volumes lifecycle are not tied to the Pod's lifecycle
  - Unlike volumes, persistent volumes will not be deleted when a Pod is deleted
  - This behaviour can be configured

Key Concepts

- Storage class - a type of storage
  - Who the provisioner, storage specific details, retention policy, etc.
- Persistent volume - the actual storage
  - A piece of storage provisioned by an administrator or thru storage class
  - Supports may different storage type
    - AWS EBS, Azure File Service, Cinder, fibre channel, GCP Disk, NFS, etc.
  - Different type of access mode - exclusive or shared
- Persistent Volume claim - when a persistent volume has been allocated for use, the volume is staid to be claimed

Persistent Volume



Static vs Dynamic

Static

- Administrator has to manually allocate storage and map it to a persistent volume
- Users can then claim this volume

Dynamic

- When Kubernetes tries to resolve a claim and the persistent volume is unavailable
- It looks for a storage class that best matches the request storage
- Dynamically creates the persistent volume using the provisioner

Defining a Persistent Volume Claim

```
apiVersion: v1
kind: PersistentVolumeClaim

meta-data:
  name: myapp-pvc
  annotations:
    volume.beta.kubernetes.io/storage-provisioner: “provisioner”

spec:
  accessModes:
    - ReadWriteOnly
  resources:
    requests:
      storage: 5Gi
  storageClassName: standard
```

Specify the provisioner that will provisions the storage class

`kubectl get storageclass`
for a list of provisioners

Mounting a Persistent Volume

```
apiVersion: v1
kind: Pod
meta-data:
  name: myapp

spec:
  volumes:
    - name: data-volume
      persistentVolumeClaim:
        claimName: myapp-pvc
  containers:
    - name: myapp
      volumeMounts:
        - mountPath: /app/public
          name: myapp-pvc
      ...
```

Specify the claim name

Kubernetes

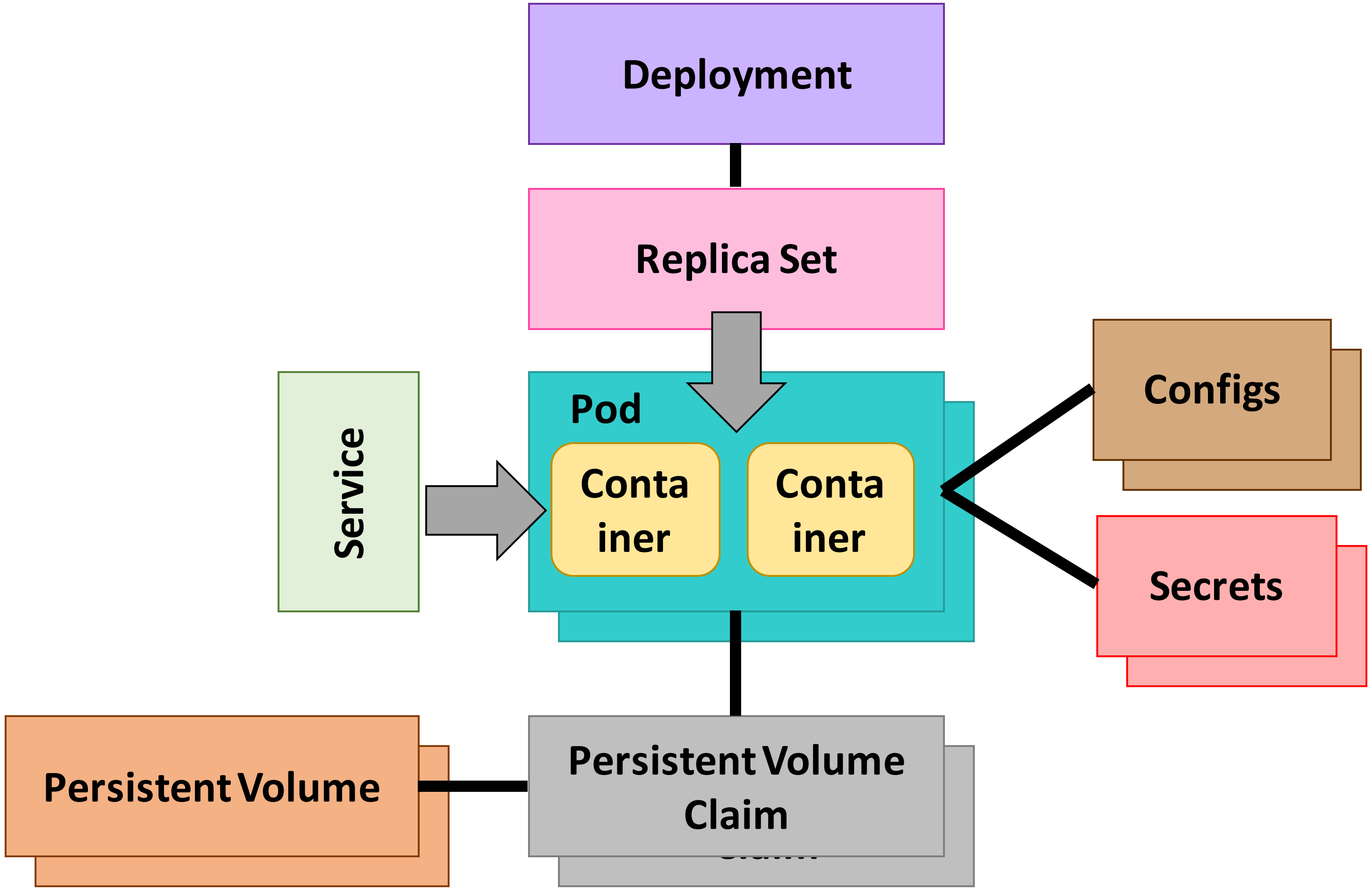


Image idea from https://www.slideshare.net/imesh/deep-dive-into-kubernetes-part-1

Persistence Volume Management

- Display persistence volume detail
  - Persistence volume - `kubectl get pv`
  - Persistence volume claim - `kubectl get pvc`
  - Storage classes - `kubectl get sc`
- Delete persistence volume

```
kubectl delete pvc <name>

kubectl delete pv <name>
```

Load Balancer and Ingress

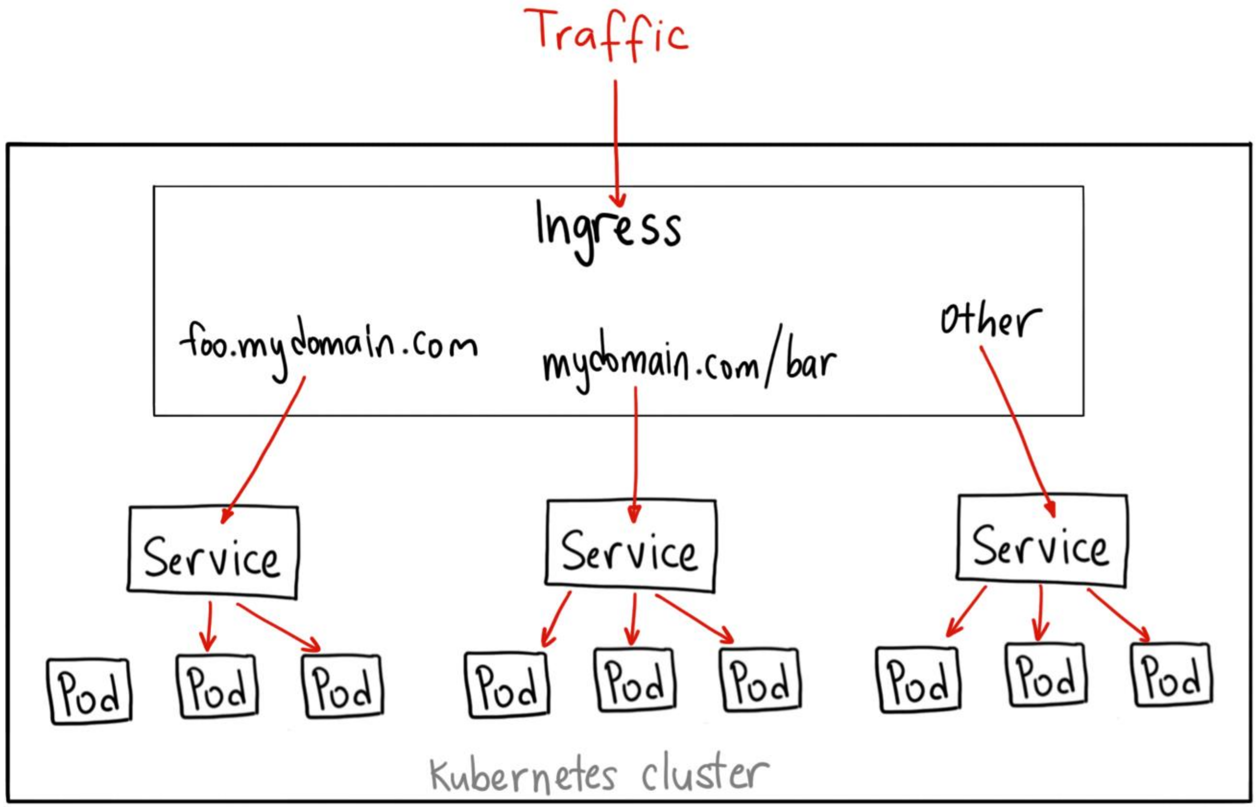
- By default services are allocated a cluster IP
  - Only accessible within the cluster
- Load balancer exposes the service to the public
  - Accessible from outside of the cluster
  - Load balancer will redirect the request to pods based on its routing policy
  - Another way to allow external access is via node port
- Load balancer are resources that are provisioned from the underlying cloud platform
  - May have more features that you require
  - Also cost more

Load Balancer and Ingress

- An ingress is a load balancer but its is provisioned as a service (with pods) inside a Kubernetes cluster
  - It’s a resource running in Kubernetes
- Typically less feature and potentially cheaper
  - However you will need to manage it
- NGINX Ingress controller is a popular ingress controller
  - Deploys NGINX as Ingress
  - https://github.com/kubernetes/ingress-nginx

Ingress

- Application layer (L7) router that sits in front of multiple services
- Define a set of routing rules on how services are access externally
  - Eg. 2 services, one for search one for checkout. Might map to `/search` and `/checkout`
- Rules are applied to ingress controllers which performs the actual routing
  - Controllers might be a cloud provider's load balancer or Nginx reverse-proxy



Defining an Ingress

Change/rewrite a matched resource to its root e.g /hello to /

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: myapp
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: “/”
    nginx.ingress.kubernetes.io/ssl-redirect: “false”
spec:
  backend:
    serviceName: landing
    servicePort: 8080
  rules:
    - http:
          paths:
            - path: /hello
              backend:
                   serviceName: myapp
                   servicePort: 8080
```

Used to configure NGINX ingress controller

Default backend if no rule matches

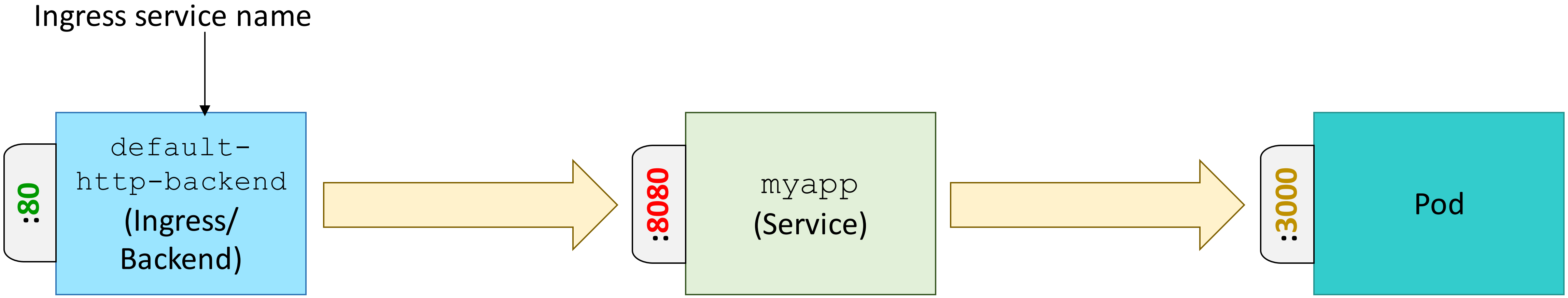
One or more of these rules to specify which services to handle what resource

Ingress Ports

```
kind: Ingress
spec:
  rules:
    http:
      paths:
        - backend:
            serviceName: mysvc
            servicePort: 8088
```

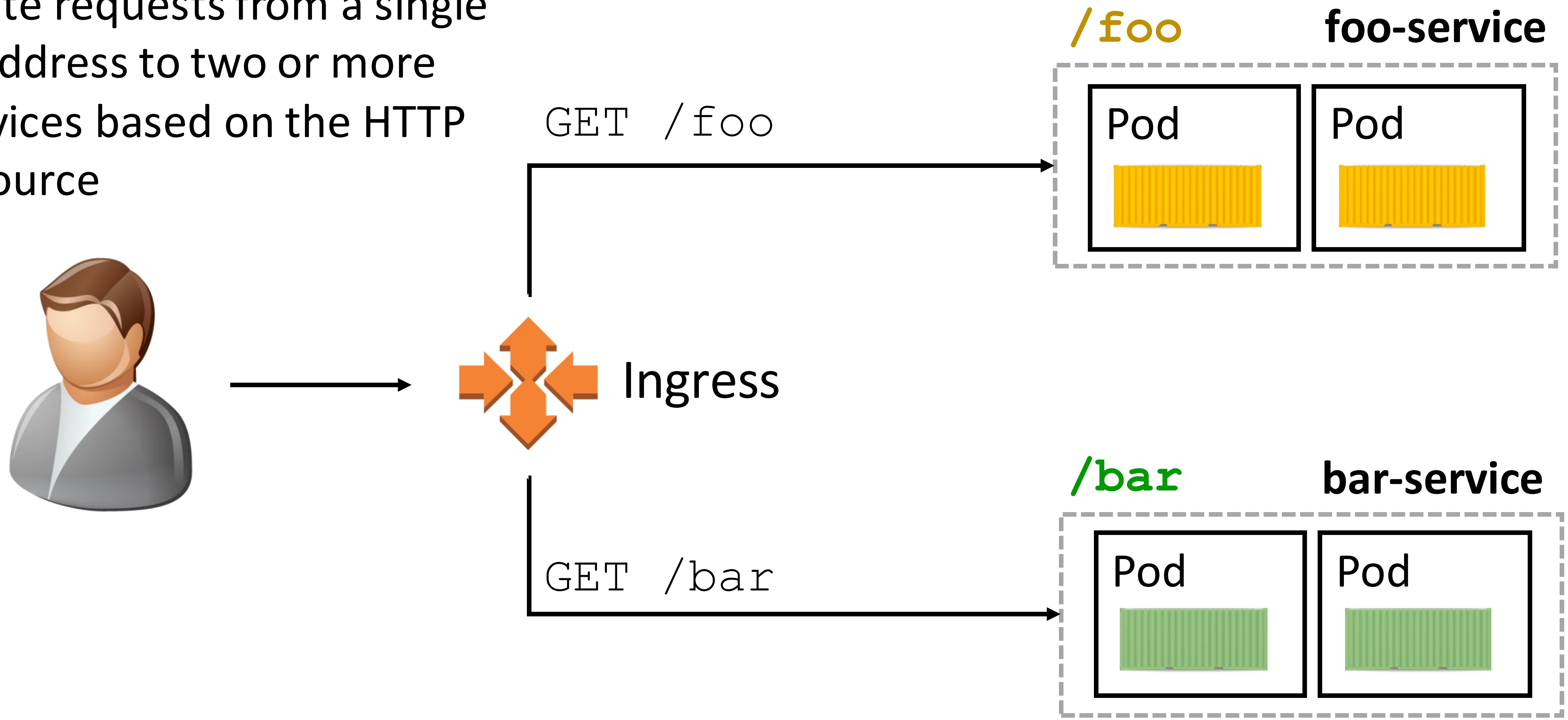
```
kind: Service
spec:
  ports:
    - port: 8080
      targetPort: 3000
```

```
kind: Deployment
spec:
  containers:
    ports:
      - containerPort: 3000
```



Ingress - Fan Out

Route requests from a single IP address to two or more services based on the HTTP resource



Ingress Fan Out Example

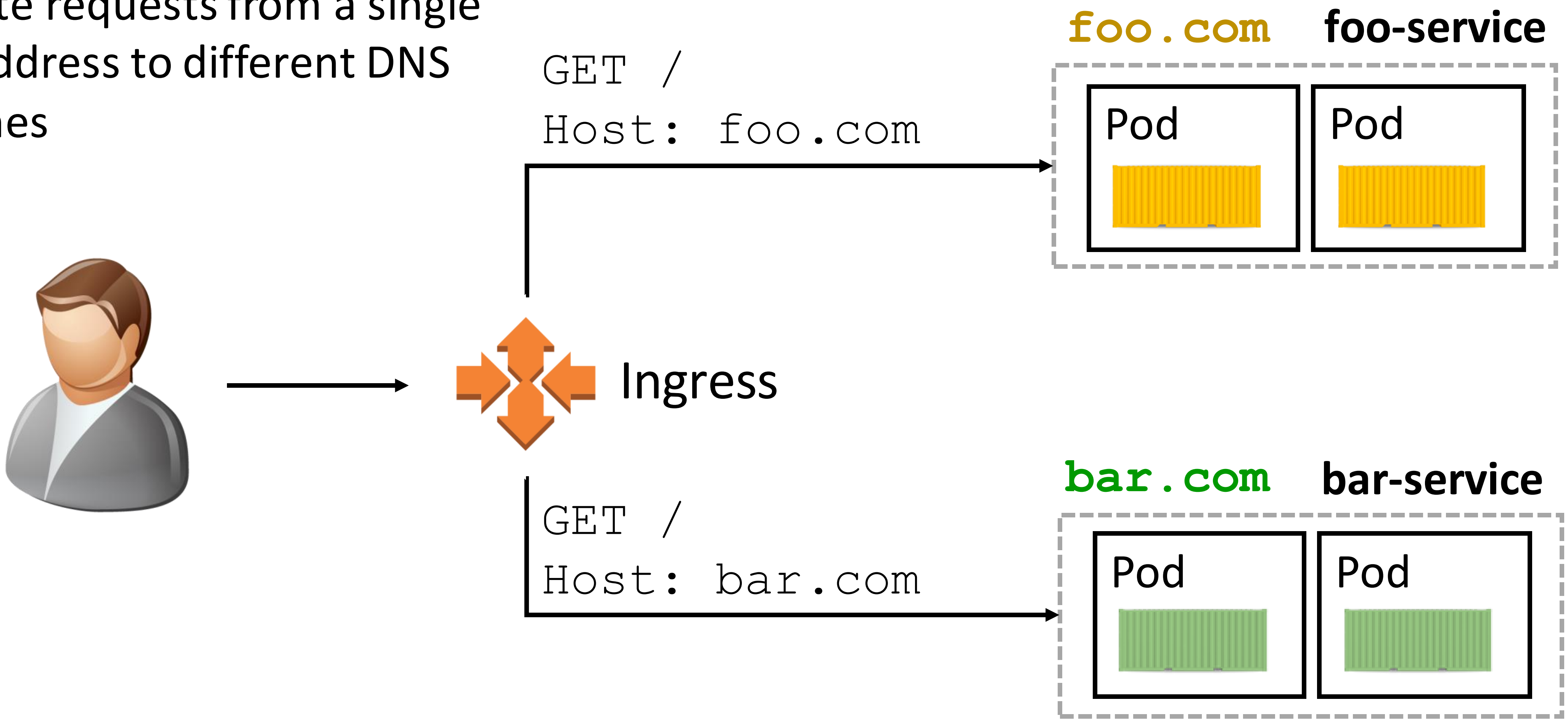
```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
     nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
       paths:
       - path: /foo
         backend:
            serviceName: foo-service
            servicePort: 8000
       - path: /bar
         backend:
            serviceName: bar-service
            servicePort: 8001
```

Request is routed to these 2 services depending on the URI

Default service not shown

Ingress - Virtual Host

Route requests from a single IP address to different DNS names



Ingress Virtual Host Example

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: foo.com
    http:
      paths:
      - backend:
          serviceName: foo-service
          servicePort: 80
    - host: bar.com
      http:
        paths:
        - backend:
            serviceName: bar-service
            servicePort: 80
```

foo.com host

bar.com host

Request is routed to these 2 services depending on the `Host` attribute. Since the request is to host without specifying a port, the service must expose port 80

Kubernetes

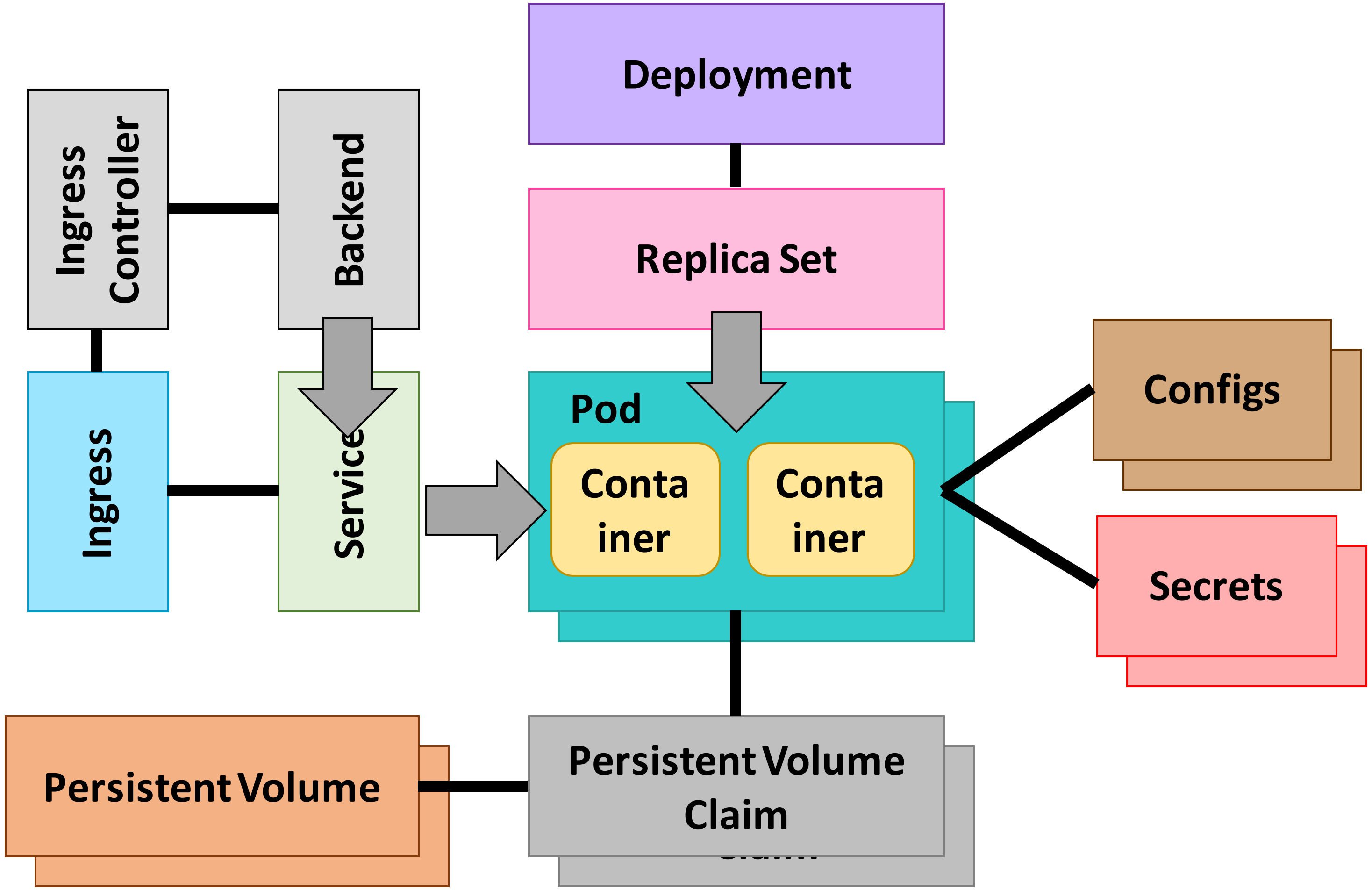
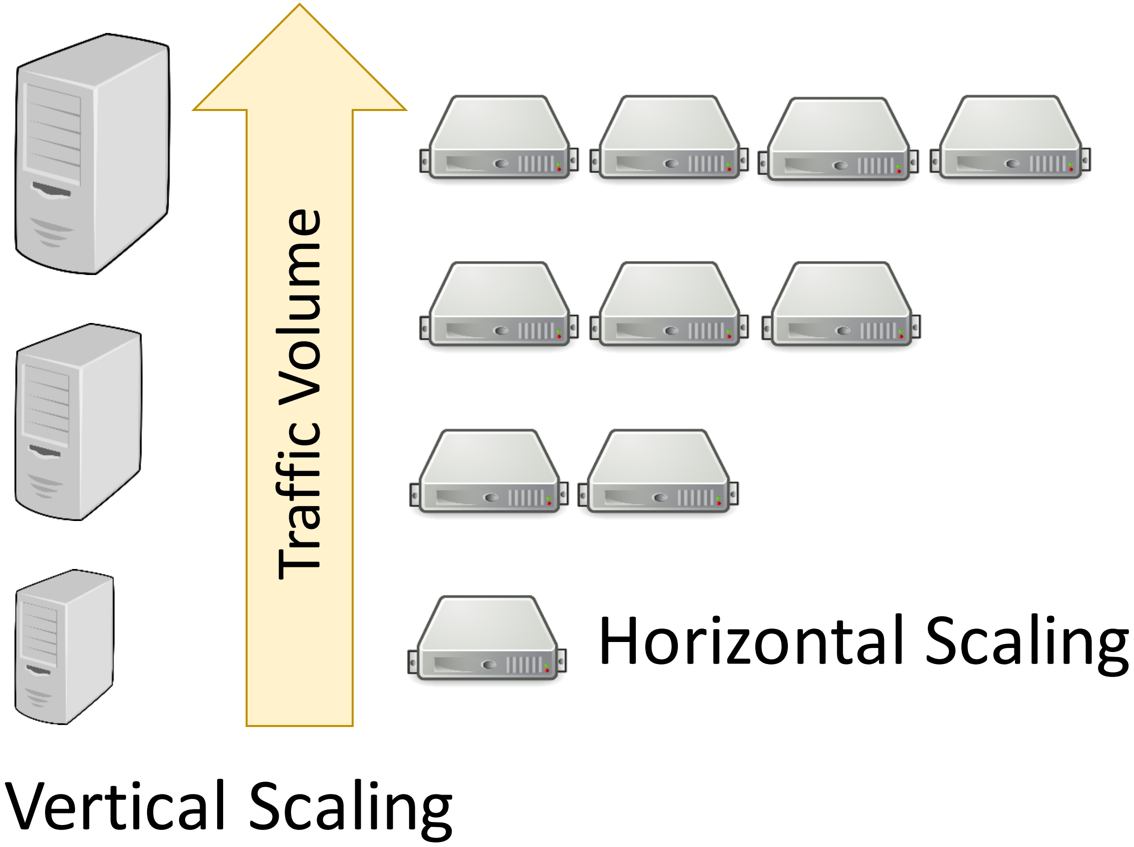


Image idea from https://www.slideshare.net/imesh/deep-dive-into-kubernetes-part-1

Scaling



- Scaling is the capability of the system to handle more workload by provisioning more resources
- Two types of scaling
  - Horizontal scaling - scales by provision more Pods
    - Applications must be stateless allowing the ingress controller to route the request to any Pod
  - Vertical scaling - scaling by giving the application more resources
    - Application must be able to utilize the extra resources eg. more vCPUs or memory

Why Scale?

- Efficient use of resources
  - Ensure that the actual usage is on parity with the current usage
- Dynamically respond to workload fluctuation
  - Elasticity - providing an agreed on SLA
- Cost optimization
  - Pay only what you use

Horizontal Manual Scaling

- Types of scaling
  - Manual
  - Automatic - Horizontal Pod Autoscaler
- Use `kubectl` to scale up or down

```
kubectl scale --replicas <number> deployment <deployment>
```

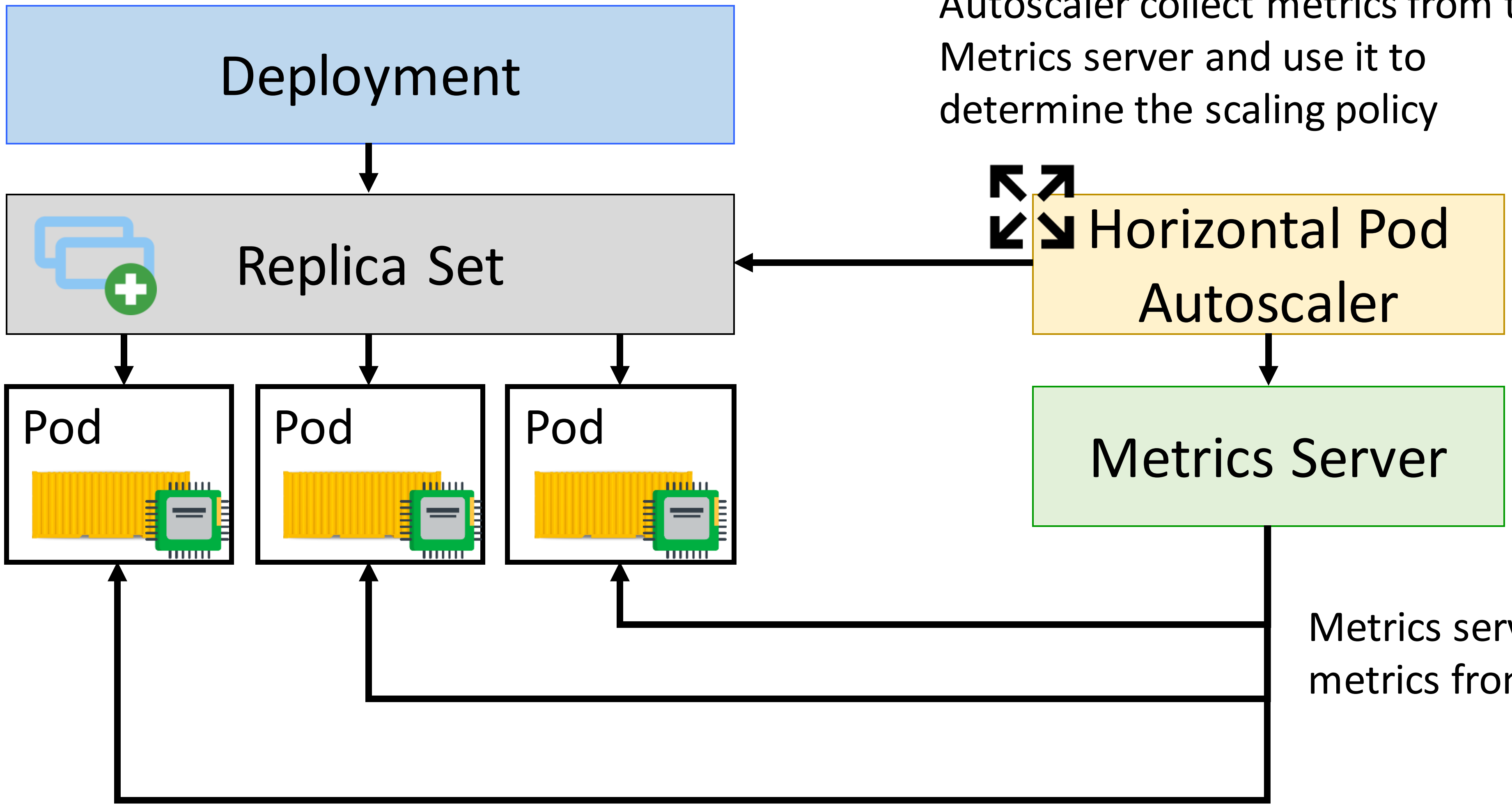
Horizontal Pod Autoscaler

- HPA scales a deployment based on one or more metrics
  - Eg. trigger scaling when CPU utilization breaches 80%
  - Metrics to scale the Pods can be
    - Build in metrics , custom metrics, external metrics
- HPA runs a control loop - runs every 30 seconds (default)
  - Queries metrics server
  - Match that against the specified threshold
  - Updates the number of replicas in a deployment if required to meet the load
  - Deployment would then perform the scaling (in or out)
- Reduces cluster size if utilization is low for a period of time
  - Scaling in

Setting Pod Request

- Horizontal scaler scales a Pod by determining if the Pod has breached a certain threshold
  - For memory and CPU
- Set the request for CPU and memory
  - Specify the minimum amount of compute resources required
- Resource type
  - CPU - measured in CPU units eg 100m is 100 millicores
    - 1 CPU in Kubernetes == 1 vCPU, Core, vCore, Hyperthread
  - Memory - 16M

Horizontal Pod Autoscaler



Autoscaler collect metrics from the Metrics server and use it to determine the scaling policy

Metrics server collects metrics from the Pods

Requesting Resources

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp
spec:
  containers:
    - name: myapp
      image: myapp:sha256:...
      resources:
        requests:
          cpu: 100m
          memory: 16M
        limits:
          memory: 32M
      ...
```

HPA only looks at the CPU

Request the minimum amount of compute resources

Describe the maximum amount of compute resources required

Defining a Horizontal Pod Autoscaler

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoScaler
metadata:
  name: myapp
spec:
  minReplicas: 3
  maxReplicas: 8
  targetCPUUtilizationPercentage: 80
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: myapp
```

Minimum and maximum number of replicas. Since the HPA is managing the replica set, this setting takes precedence over the deployment setting

Percentage of the CPU utilization over all the Pods

The deployment that this HPA is targeting

Kubernetes

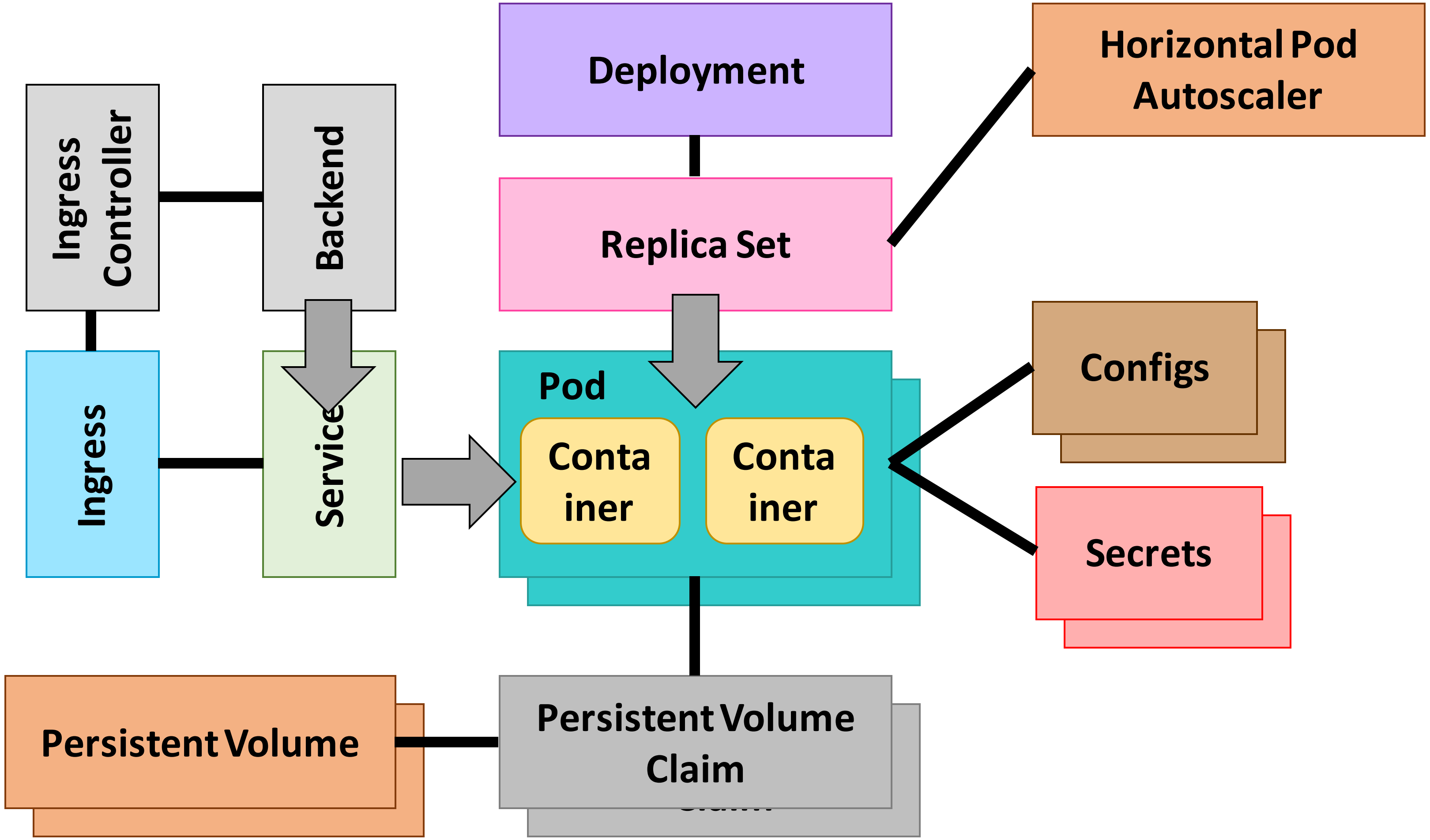


Image idea from https://www.slideshare.net/imesh/deep-dive-into-kubernetes-part-1

Appendix

Managing Context

- For grouping access parameters under a common name
  - Like a profile
  - Set the namespace, do not need the `−n` option
- Create a context

```
kubectl config set-context <context_name> --namespace=<name> \
  --cluster=<cluster_name> --user=<user_name>
```

- View current contexts

```
kubectl config view
```

- Use a context

```
kubectl config use-context <context_name>
```