Kubernetes 101

Instructer: Damrongwit N.
Email: jack@racksync.com

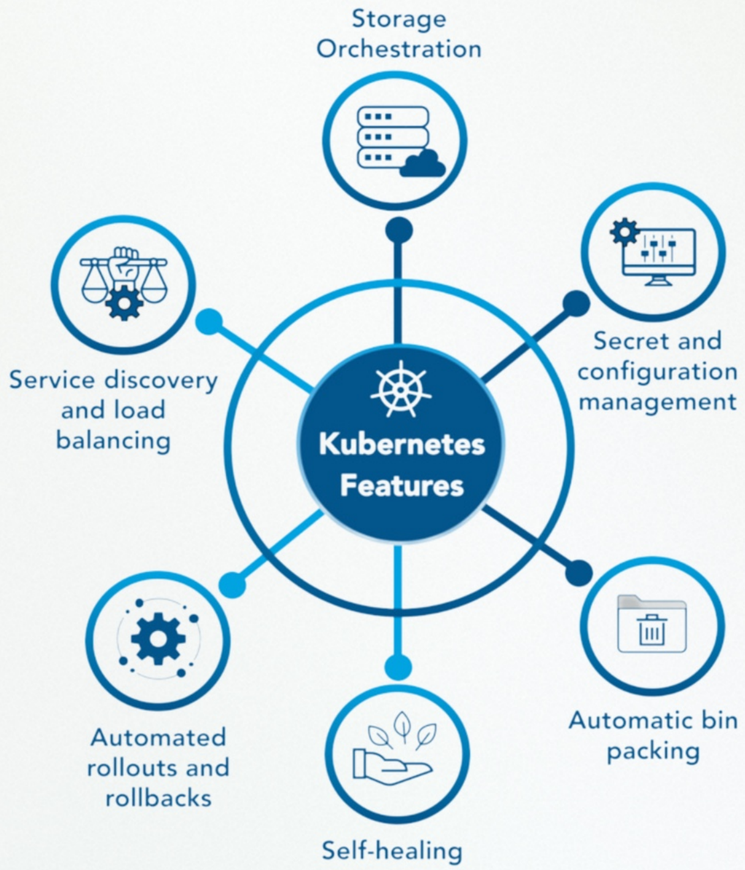
Intro - What is Kubernetes?

Kubernetes or K8s was a project spun out of Google as a open source next-gen container scheduler designed with the lessons learned from developing and managing Borg and Omega.
Kubernetes was designed from the ground-up as a loosely coupled collection of components centered around deploying, maintaining, and scaling applications.

Intro - What Does Kubernetes do?

Kubernetes is the linux kernel of distributed systems.
It abstracts away the underlying hardware of the nodes and provides a uniform interface for applications to be both deployed and consume the shared pool of resources.

K8s Main Features

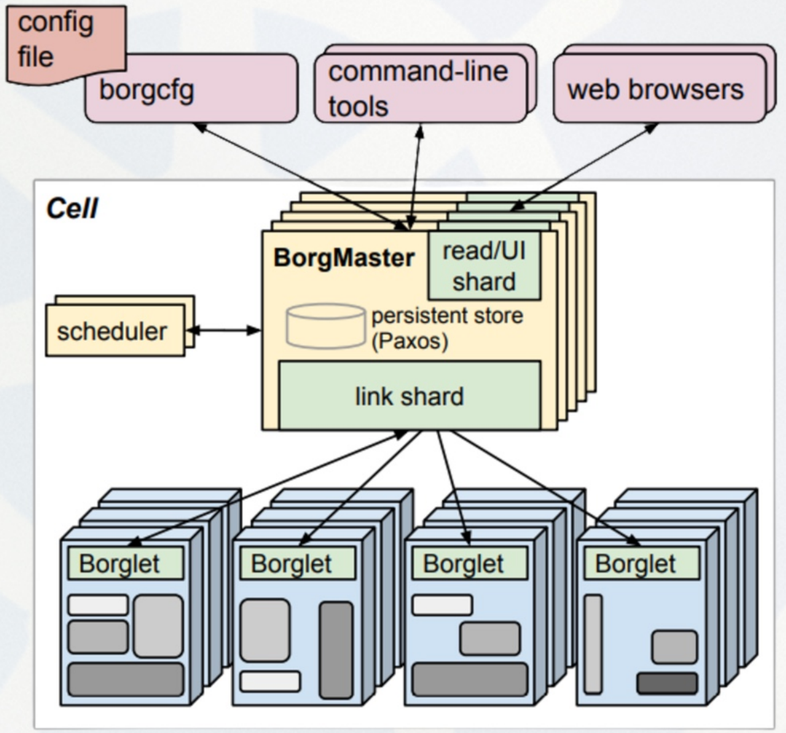


- *Service discovery and load balancing*
- *Storage orchestration*
- *Secret and configuration management*
- *Automatic bin packing*
- *Self-healing*
- *Automated rollouts and rollbacks*

History of K8s (Kubernetes)

Kubernetes (K8s), inspired by ***Google's*** internal ***Borg*** project, launched in 2014 as an open-source container orchestration platform. In 2015, Google donated it to ***CNCF***, accelerating its adoption and community-driven growth. Kubernetes quickly became the global standard for managing containers, praised for scalability, declarative configurations, self-healing capabilities, and extensive ecosystem support, transforming cloud-native applications worldwide.

https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes

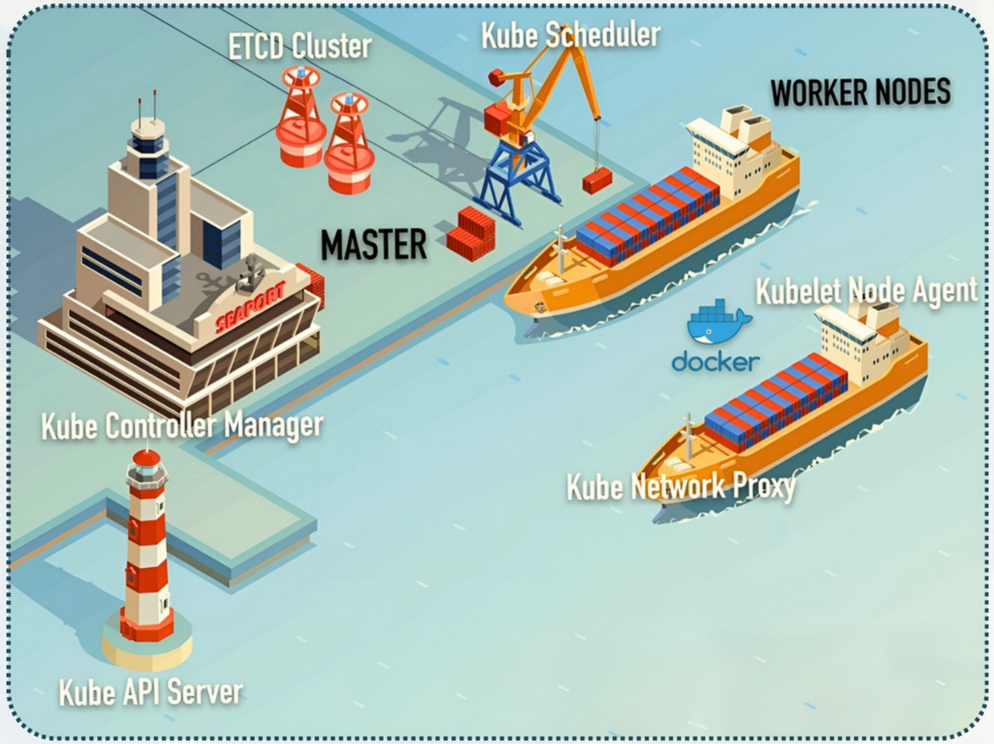


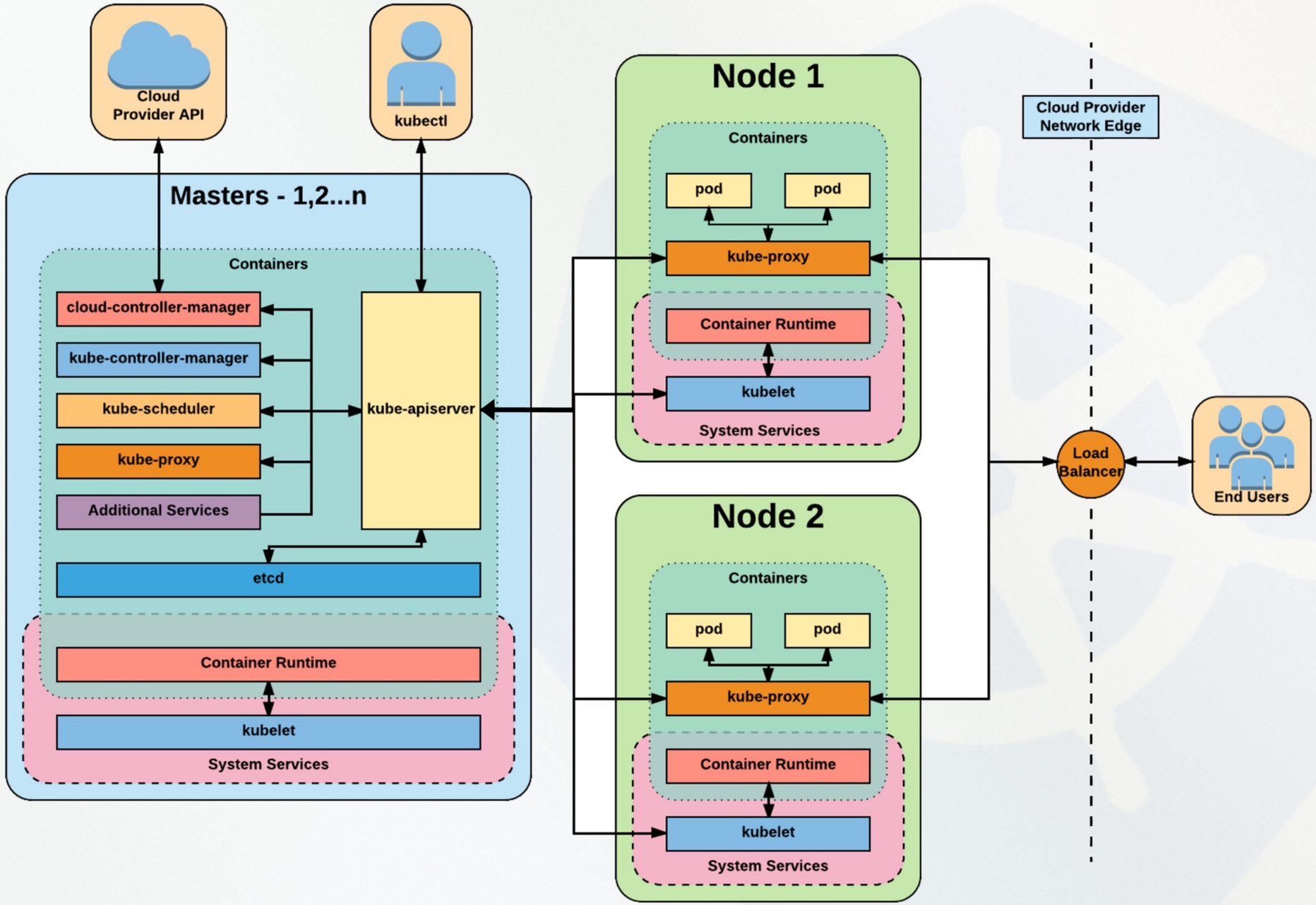
Kubernetes Architecture

Architecture Overview

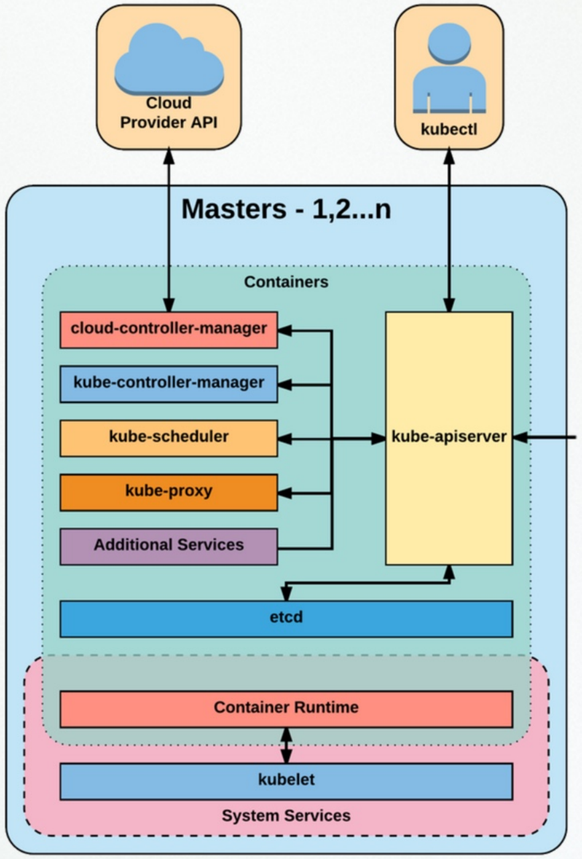
Masters - Acts as the primary **control plane** for Kubernetes. Masters are responsible at a minimum for running the API Server, scheduler, and cluster controller. They commonly also manage storing cluster state, cloud-provider specific components and other cluster essential services.

Nodes - Are the **workers** of a Kubernetes cluster. They run a minimal agent that manages the node itself, and are tasked with executing workloads as designated by the master.




Cloud
Provider API
kubectl
Masters - 1,2...n
Containers
cloud-controller-manager
kube-controller-manager
kube-scheduler
kube-proxy
Additional Services
kube-apiserver
etcd
Container Runtime
kubelet
System Services
Node 1
Containers
pod
pod
kube-proxy
Container Runtime
kubelet
System Services
Node 2
Containers
pod
pod
kube-proxy
Container Runtime
kubelet
System Services
Cloud Provider
Network Edge
Load
Balancer
End Users

Master Components



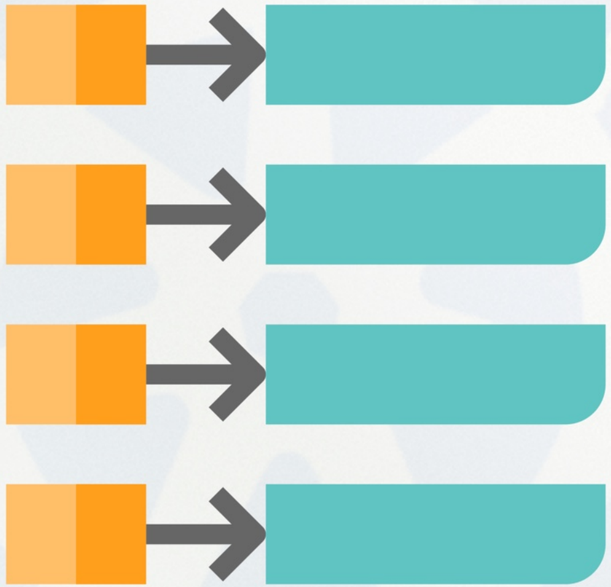
- Kube-apiserver
- Etcd
- Kube-controller-manager
- Cloud-controller-manager
- Kube-scheduler

kube-apiserver

The apiserver provides a forward facing REST interface into the kubernetes control plane and datastore. All clients, including nodes, users and other applications interact with kubernetes strictly through the API Server.
It is the true core of Kubernetes acting as the gatekeeper to the cluster by handling authentication and authorization, request validation, mutation, and admission control in addition to being the front-end to the backing datastore.

etcd

Etcd acts as the cluster datastore; providing a strong, consistent and highly available key-value store used for persisting cluster state.



kube-controller-manager

The controller-manager is the primary daemon that manages all core component control loops. It monitors the cluster state via the apiserver and steers the cluster towards the desired state.

Core controllers list:

https://github.com/kubernetes/kubernetes/blob/master/cmd/kube-controller-manager/app/controllermanager.go#L332

cloud-controller-manager

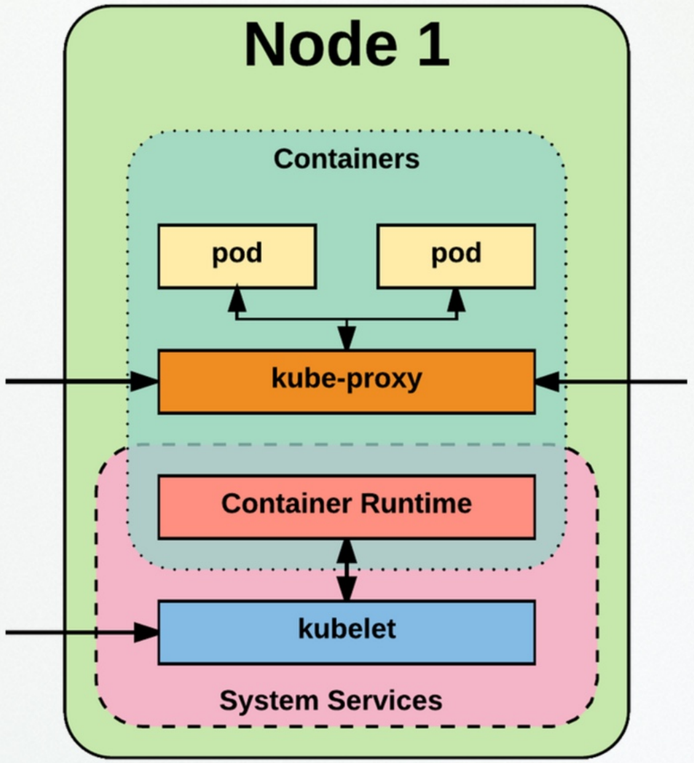
The cloud-controller-manager is a daemon that provides cloud-provider specific knowledge and integration capability into the core control loop of Kubernetes. The controllers include Node, Route, Service, and add an additional controller to handle PersistentVolumeLabels .

Node Components

kube-scheduler

Kube-scheduler is a verbose policy-rich engine that evaluates workload requirements and attempts to place it on a matching resource. These requirements can include such things as general hardware reqs, affinity, anti-affinity, and other custom resource requirements.

Node Components



- Kubelet
- Kube-proxy
- Container runtime engine

kubelet

Acts as the node agent responsible for managing pod lifecycle on its host. Kubelet understands YAML container manifests that it can read from several sources:

- File path
- HTTP Endpoint
- Etcd watch acting on any changes
- HTTP Server mode accepting container manifests over a simple API.

kube-proxy

Manages the network rules on each node and performs connection forwarding or load balancing for Kubernetes cluster services.

Available Proxy Modes:

- **Userspace**
- **iptables**
- ipvs (alpha in 1.8)

Container Runtime

With respect to Kubernetes, A container runtime is a CRI (Container Runtime Interface) compatible application that executes and manages containers.

- Containerd (docker)
- Cri-o
- Rkt
- Kata (formerly clear and hyper)
- Virtlet (VM CRI compatible runtime)

Additional Services

Kube-dns - Provides cluster wide DNS Services. Services are resolvable to <service>.<namespace>.svc.cluster.local.

Heapster - Metrics Collector for kubernetes cluster, used by some resources such as the Horizontal Pod Autoscaler. (required for kubedashboard metrics)

Kube-dashboard - A general purpose web based UI for kubernetes.

Networking

Networking - Fundamental Rules

1. All Pods can communicate with all other Pods without NAT
2. All nodes can communicate with all Pods (and vice-versa) without NAT.
3. The IP that a Pod sees itself as is the same IP that others see it as.

Networking - Fundamentals Applied

Containers in a pod exist within the same network namespace and share an IP; allowing for intrapod communication over localhost.

Pods are given a cluster unique IP for the duration of its lifecycle, but the pods themselves are fundamentally ephemeral.

Services are given a persistent cluster unique IP that spans the Pods lifecycle.

External Connectivity is generally handed by an integrated cloud provider or other external entity (load balancer)

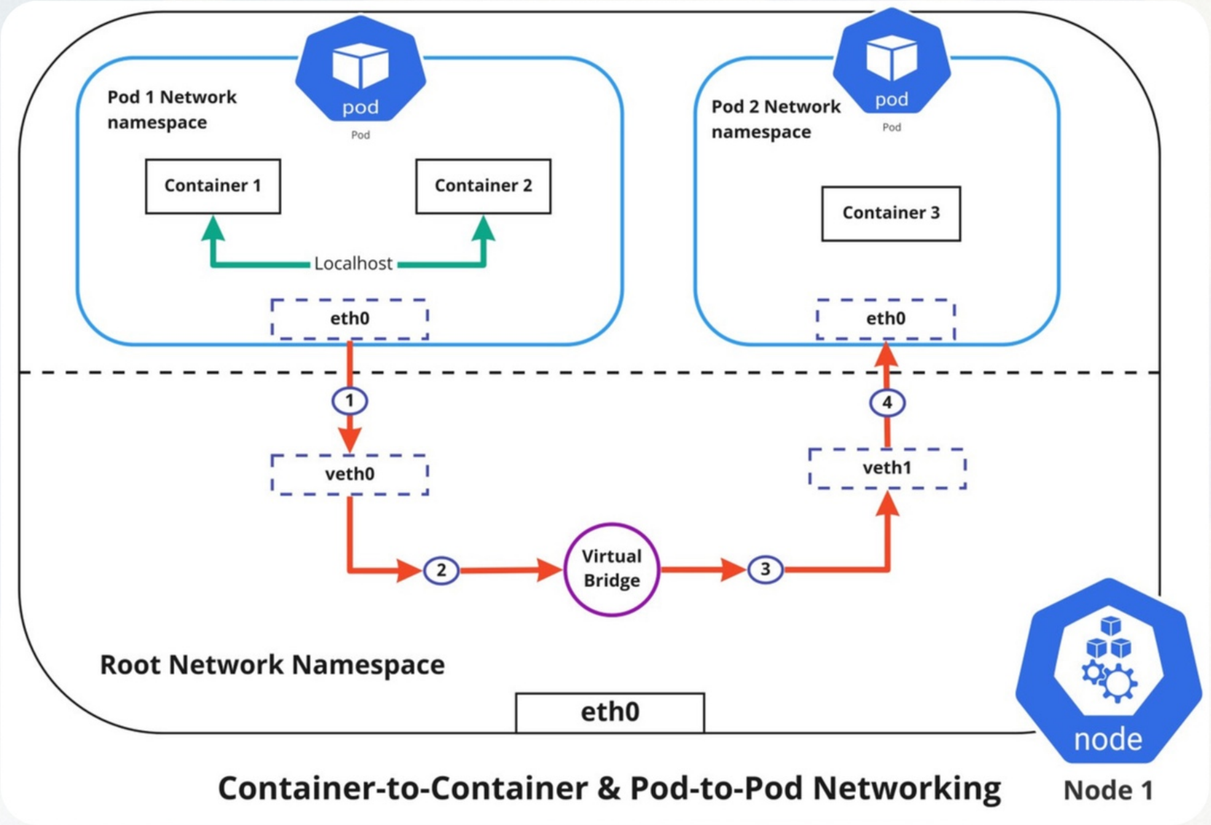
Networking - CNI

Networking within Kubernetes is plumbed via the Container Network Interface (CNI), an interface between a container runtime and a network implementation plugin.
Compatible CNI Network Plugins:

- Calico
- Cillium
- Contiv
- Contrail
- Flannel
- GCE
- kube-router
- Multus
- OpenVSwitch
- OVN
- Romana
- Weave

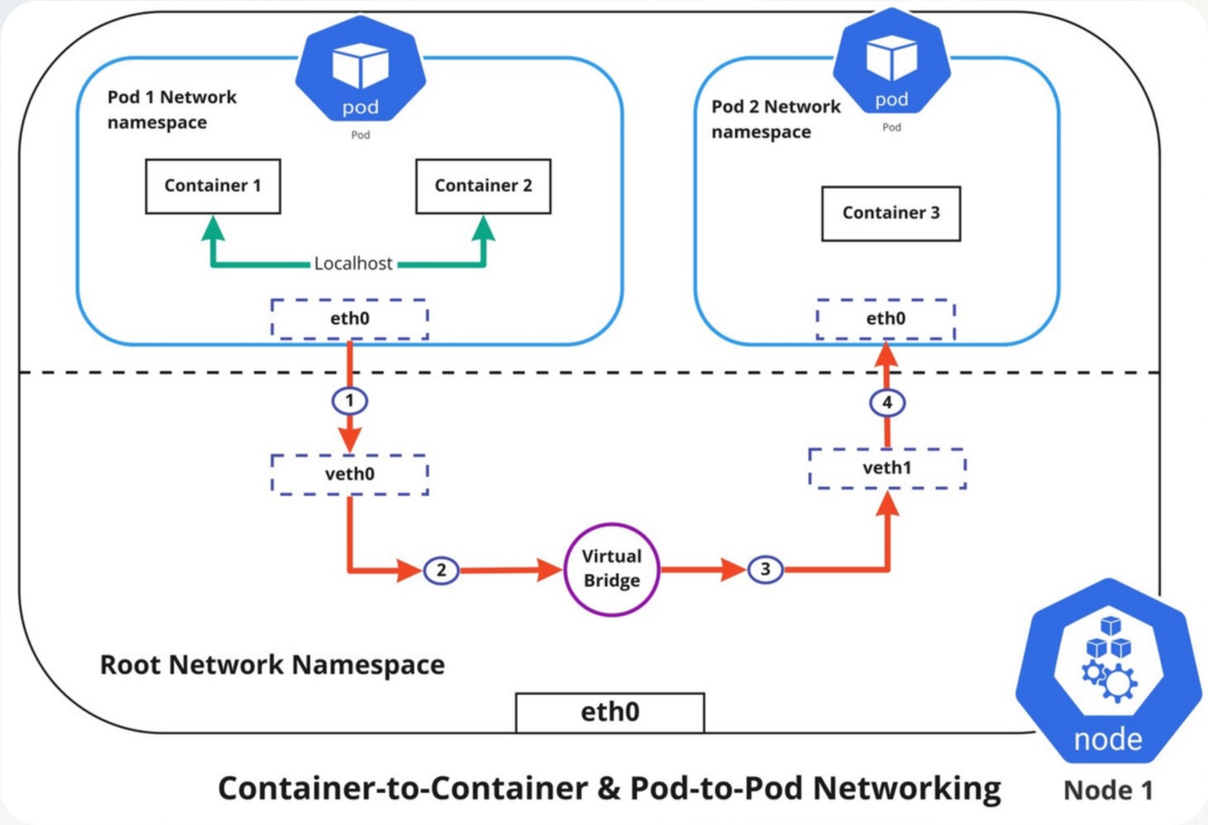
Container-to-Container Networking

Container-to-Container networking happens through the Pod network namespace. Network namespaces allow us to have separate network interfaces and routing tables that are isolated from the rest of the system and can operate independently. Every Pod will have its own network namespace and containers inside that Pod share the same IP address and ports. All communication between these containers would happen via the localhost as they are all part of the same namespace. (Represented by the green line in the diagram)



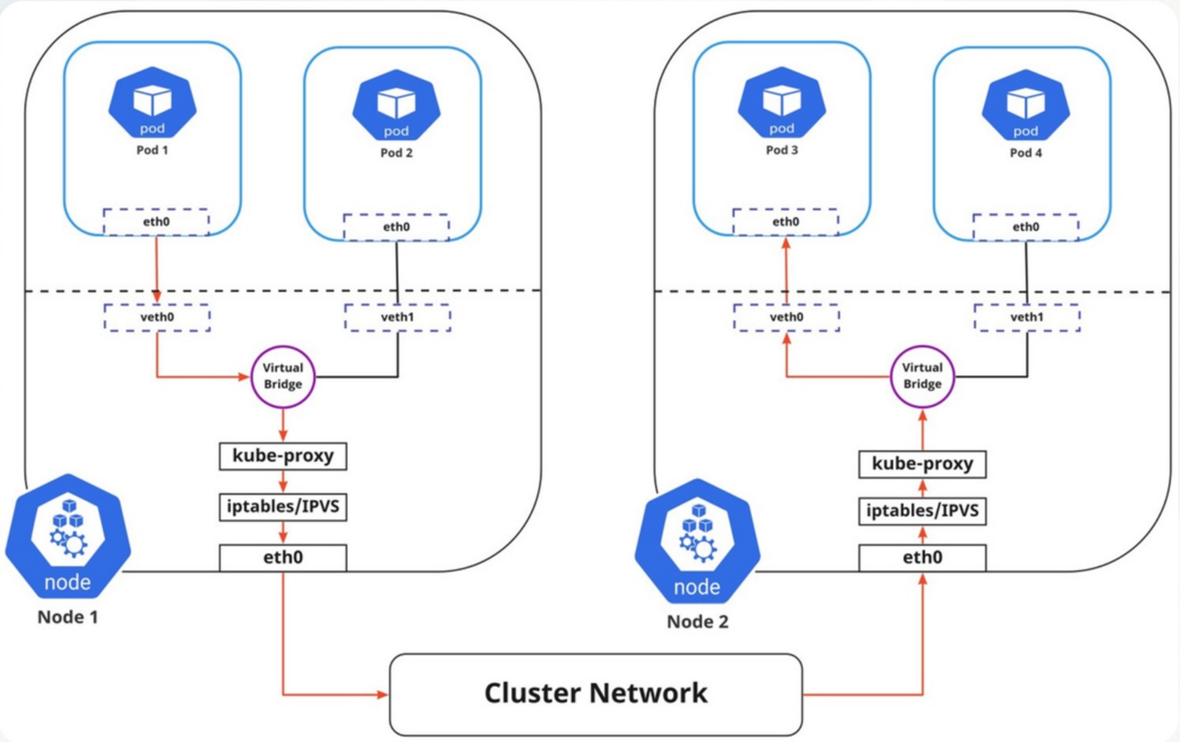
Pod-to-Pod Networking

In Kubernetes, every node has a designated CIDR range of IPs for Pods. This would ensure that every Pod gets a unique IP address that can be seen by other Pods in the cluster and also ensures that when a new Pod is created, the IP address never overlaps. Unlike Container-to-Container networking, Pod-to-Pod communication happens using real IPs, whether the Pod is deployed on the same node or a different node in the cluster.



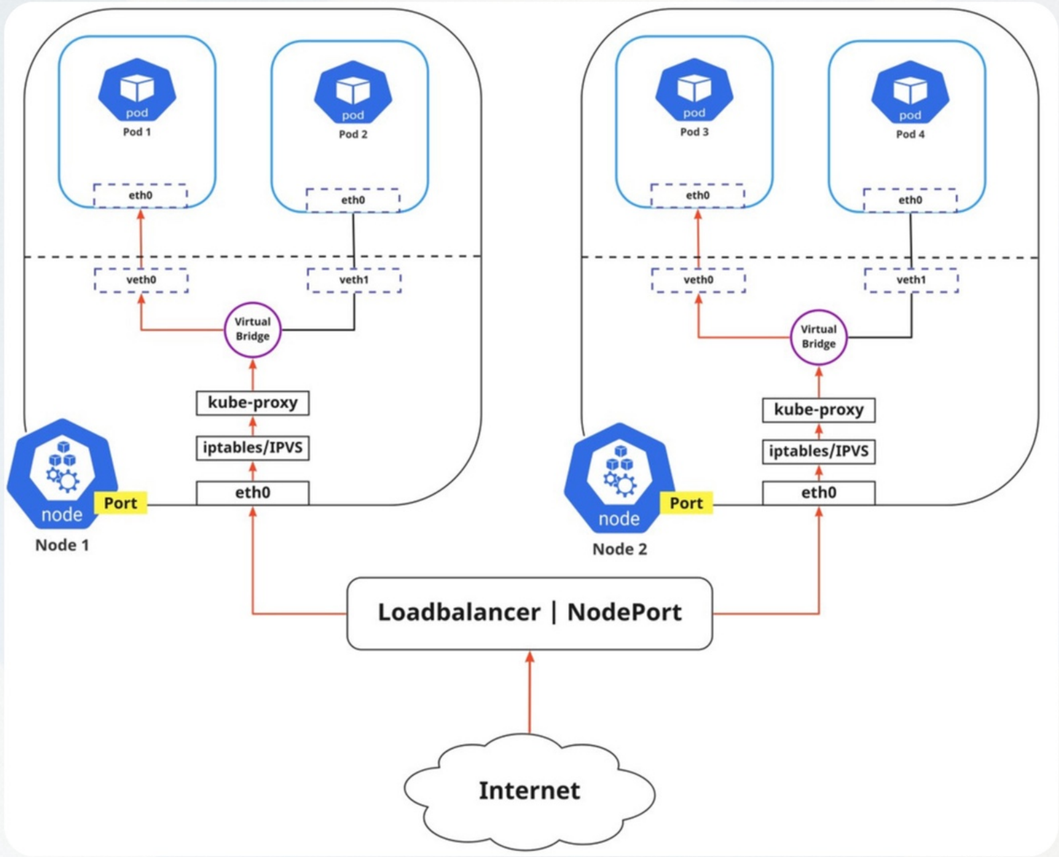
Pod-to-Service Networking

In Kubernetes, every node has a designated CIDR range of IPs for Pods. This would ensure that every Pod gets a unique IP address that can be seen by other Pods in the cluster and also ensures that when a new Pod is created, the IP address never overlaps. Unlike Container-to-Container networking, Pod-to-Pod communication happens using real IPs, whether the Pod is deployed on the same node or a different node in the cluster.



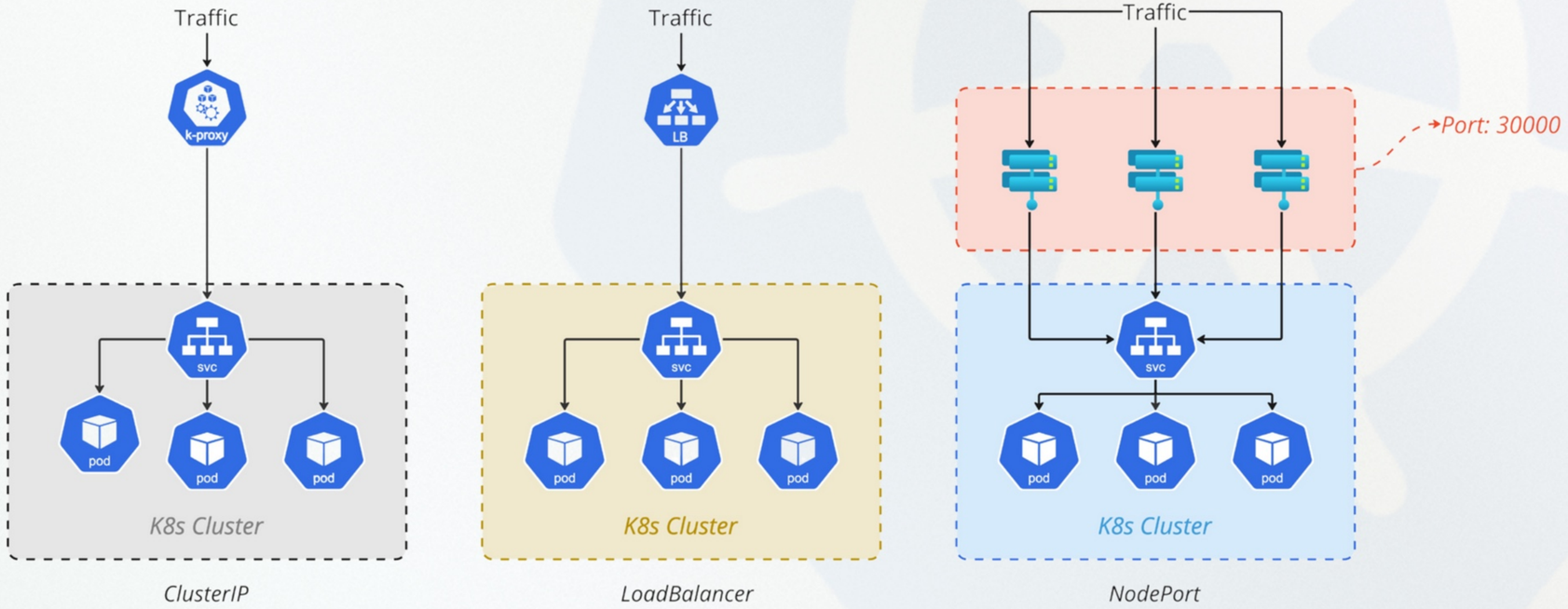
Internet-to-Service Networking

In Kubernetes, every node has a designated CIDR range of IPs for Pods. This would ensure that every Pod gets a unique IP address that can be seen by other Pods in the cluster and also ensures that when a new Pod is created, the IP address never overlaps. Unlike Container-to-Container networking, Pod-to-Pod communication happens using real IPs, whether the Pod is deployed on the same node or a different node in the cluster.



ServiceTypes for Publishing Services

Kubernetes Services provide us with a way of accessing a group of Pods which may usually be defined by using a label selector. This could be applications trying to access other applications within the cluster or it could also allow us to expose an application running in the cluster to the external world. Kubernetes ServiceTypes allow you to specify what kind of Service you want.



Kubernetes Concepts

Kubernetes Concepts - Core

Cluster - A collection of hosts that aggregate their available resources including cpu, ram, disk, and their devices into a usable pool.

Master - The master(s) represent a collection of components that make up the control plane of Kubernetes. These components are responsible for all cluster decisions including both scheduling and responding to cluster events.

Node - A single host, physical or virtual capable of running pods. A node is managed by the master(s), and at a minimum runs both kubelet and kube-proxy to be considered part of the cluster.

Namespace - A logical cluster or environment. Primary method of dividing a cluster or scoping access.

Kubernetes Concepts - Core (continue)

Label - Key-value pairs that are used to identify, describe and group together related sets of objects. Labels have a strict syntax and available character set. *

Annotation - Key-value pairs that contain non-identifying information or metadata. Annotations do not have the the syntax limitations as labels and can contain structured or unstructured data.

Selector - Selectors use labels to filter or select objects. Both equality-based (=, ==, !=) or simple key-value matching selectors are supported.

Labels, and Annotations, and Selectors

In this both the Deployment and Pod are labeled with app:nginx, tier: frontend

The replicaset definition targets them with simple selectors matching the key-value pairs of both app:nginx and tier: frontend

```yaml
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: nginx
  annotations:
    description: "nginx frontend"
  labels:
    app: nginx
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
      tier: frontend
  template:
    metadata:
      labels:
        app: nginx
        tier: frontend
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

Set-based selectors

Valid Operators:

- In
- NotIn
- Exists
- DoesNotExist

Supported Objects with set-based selectors:

- Job
- Deployment
- ReplicaSet
- DaemonSet
- PersistentVolumeClaims

```yaml
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: nginx
  annotations:
    description: "nginx frontend"
  labels:
    app: nginx
    tier: frontend
spec:
  replicas: 3
  selector:
    matchExpressions:
      - {key: app, operator: In, values: [nginx]}
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: nginx
        tier: frontend
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

Concepts - Workloads

Pod - A pod is the smallest unit of work or management resource within Kubernetes. It is comprised of one or more containers that share their storage, network, and context (namespace, cgroups etc).

ReplicaSet - Next Generation ReplicationController. Supports set-based selectors.

Deployment - A declarative method of managing stateless Pods and ReplicaSets. Provides rollback functionality in addition to more granular update control mechanisms.

Deployment

Contains configuration of how updates or ‘deployments’ should be managed in addition to the pod template used to generate the ReplicaSet.

```yaml
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: nginx
  annotations:
    description: "nginx frontend"
  labels:
    app: nginx
    tier: frontend
spec:
  replicas: 3
  mindReadySeconds: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 5
      maxUnavailable: 2
  selector:
    matchLabels:
      app: nginx
      tier: frontend
  template:
    metadata:
      labels:
        app: nginx
        tier: frontend
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

ReplicaSet

Generated ReplicaSet from Deployment spec.

```yaml
apiVersion: apps/v1beta2
kind: ReplicaSet
metadata:
  name: nginx
  annotations:
    description: "nginx frontend"
  labels:
    app: nginx
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
      tier: frontend
  template:
    metadata:
      labels:
        app: nginx
        tier: frontend
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

Concepts - Workloads (continue)

StatefulSet - A controller tailored to managing Pods that must persist or maintain state. Pod identity including hostname, network, and storage will be persisted.

DaemonSet - Ensures that all nodes matching certain criteria will run an instance of a supplied Pod. Ideal for cluster wide services such as log forwarding, or health monitoring.

StatefulSet

- Attaches to ‘headeless service’ (not shown) nginx.
- Pods given unique ordinal names using the pattern <statefulset name>-<ordinal index>.
- Creates independent persistent volumes based on the ‘volumeClaimTemplates’.

```yaml
apiVersion: apps/v1beta2
kind: StatefulSet
metadata:
  name: nginx
spec:
  serviceName: "nginx"
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

DaemonSet

- Bypasses default scheduler
- Schedules a single instance on every host while adhering to tolerances and taints.

```yaml
apiVersion: apps/v1beta2
kind: DaemonSet
metadata:
  name: nginx
  namespace: kube-system
  labels:
    app: nginx
spec:
  selector:
    matchLabels:
      name: nginx
  template:
    metadata:
      labels:
        name: nginx
    spec:
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
          name: web
```

Concepts - Workloads (continue)

Job - The job controller ensures one or more pods are executed and successfully terminates. It will do this until it satisfies the completion and/or parallelism condition.

CronJob - An extension of the Job Controller, it provides a method of executing jobs on a cron-like schedule.

Jobs

- Number of pod executions can be controlled via spec.completions
- Jobs can be parallelized using spec.parallelism
- Jobs and Pods are NOT automatically cleaned up after a job has completed.

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: hello
spec:
  completions: 10
  parallelism: 2
  template:
    metadata:
      name: hello
    spec:
      containers:
      - name: hello
        image: alpine:latest
        command: ["echo",  "hello there!"]
      restartPolicy: Never
  backoffLimit: 4
```

CronJob

Adds cron schedule to job template

```yaml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "30 8 * * *"
  jobTemplate:
    spec:
      template:
        metadata:
          name: hello
        spec:
          containers:
          - name: hello
            image: alpine:latest
            command: ["echo",  "hello there!"]
          restartPolicy: OnFailure
```

Concepts - Network

Service - Services provide a method of exposing and consuming L4 Pod network accessible resources. They use label selectors to map groups of pods and ports to a cluster-unique virtual IP.

Ingress - An ingress controller is the primary method of exposing a cluster service (usually http) to the outside world. These are load balancers or routers that usually offer SSL termination, name-based virtual hosting etc.

Service

- Acts as the unified method of accessing replicated pods.
- Four major Service Types:
  - **CluterIP** - Exposes service on a strictly cluster-internal IP (default)
  - **NodePort** - Service is exposed on each node's IP on a statically defined port.
  - **LoadBalancer** - Works in combination with a cloud provider to expose a service outside the cluster on a static external IP.
  - **ExternalName** - used to references endpoints OUTSIDE the cluster by providing a static internally referenced DNS name.

```yaml
kind: Service
apiVersion: v1
metadata:
  name: nginx
spec:
  type: ClusterIP
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

Ingress Controller

- Deployed as a pod to one or more hosts
- Ingress controllers are an external controller with multiple options.
  - Nginx
  - HAproxy
  - Contour
  - Traefik
- Specific features and controller specific configuration is passed through annotations.

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: "nginx"
  name: nginx-ingress
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /nginx
        backend:
          service: nginx
          servicePort: 80
```

Concepts - Storage

Volume - Storage that is tied to the Pod Lifecycle, consumable by one or more containers within the pod.

PersistentVolume - A PersistentVolume (PV) represents a storage resource. PVs are commonly linked to a backing storage resource, NFS, GCEPersistentDisk, RBD etc. and are provisioned ahead of time. Their lifecycle is handled independently from a pod.

PersistentVolumeClaim - A PersistentVolumeClaim (PVC) is a request for storage that satisfies a set of requirements instead of mapping to a storage resource directly. Commonly used with dynamically provisioned storage.
StorageClass - Storage classes are an abstraction on top of an external storage resource. These will include a provisioner, provisioner configuration parameters as well as a PV reclaimPolicy.

Volumes

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:latest
    name: nginx
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: www
  volumes:
  - name: www
    emptyDir: {}
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:latest
    name: nginx
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: www
  volumes:
  - name: www
    awsElasticBlockStore:
      volumeID: <volume-id>
      fsType: ext4
```

Persistent Volumes

```yaml
apiVersion: v1
  kind: PersistentVolume
  metadata:
    name: pv-nfs
  spec:
    capacity:
      storage: 500Gi
    volumeMode: Filesystem
    accessModes:
      - ReadWriteMany
    persistentVolumeReclaimPolicy: Recycle
    storageClassName: slow
    mountOptions:
      - hard
      - nfsvers=4.1
    nfs:
      path: /data
      server: 10.255.100.10
```

- **PVs are a cluster-wide resource**
- **Not directly consumable by a Pod**
- **PV Parameters:**
  - Capacity
  - accessModes
    - ReadOnlyMany (ROX)
    - ReadWriteOnce (RWO)
    - ReadWriteMany (RWX)
  - persistentVolumeReclaimPolicy
    - Retain
    - Recycle
    - Delete
  - StorageClass

Persistent Volumes Claims

- PVCs are scoped to namespaces
- Supports accessModes like PVs
- Uses resource request model similar to Pods
- Claims will consume storage from matching PVs or StorageClasses based on storageClass and selectors.

```yaml
apiVersion: v1
  kind: PersistentVolumeClaim
  metadata:
    name: my-nfs-pvc
  spec:
    accessModes:
    - ReadWriteMany
    resources:
      requests:
        storage: 50Gi
    storageClass: slow
```

Storage Classes

- Uses an external system defined by the provisioner to dynamically consume and allocate storage.
- Storage Class Fields
  - Provisioner
  - Parameters
  - reclaimPolicy

```yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fast
provisioner: kubernetes.io/rbd
reclaimPolicy: Delete
parameters:
  monitors: 10.16.153.105:6789
  adminId: kube
  adminSecretName: ceph-secret
  adminSecretNamespace: kube-system
  pool: kube
  userId: kube
  userSecretName: ceph-secret-user
  fsType: ext4
  imageFormat: "2"
  imageFeatures: "layering"
```

Concepts - Configuration

ConfigMap - Externalized data stored within kubernetes that can be referenced as a commandline argument, environment variable, or injected as a file into a volume mount. Ideal for separating containerized application from configuration.

Secret - Functionally identical to ConfigMaps, but stored encoded as base64, and encrypted at rest (if configured).

ConfigMaps and Secrets

- **Can be used in Pod Config:**
  - Injected as a file
  - Passed as an environment variable
  - **Used as a container command (requires passing as env var)**

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:latest
    name: nginx
    volumeMounts:
      - name: myConfigMap
        path: /etc/config
  volumes:
  volumes:
    - name: myConfigMap
      configMap:
        name: my-cm
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx:latest
    name: nginx
    env:
      - name: USERNAME
        valueFrom:
          secretKeyRef:
            name: my-secret
            key: username
```

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-cm
data:
  name: mydata.txt
  contents: |
    you can store
    multiline content
    and configfiles
```

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
data:
  username: aGVycGRlcnA=
  password: aW1tYWNvbXB1dGVy
```

Concepts - Auth and Identity (RBAC)

[Cluster]Role - Roles contain rules that act as a set of permissions that apply verbs like “get”, “list”, “watch” etc over resources that are scoped to apiGroups. Roles are scoped to namespaces, and ClusterRoles are applied cluster-wide.

[Cluster]RoleBinding - Grant the permissions as defined in a [Cluster]Role to one or more “subjects” which can be a user, group, or service account.

ServiceAccount - ServiceAccounts provide a consumable identity for pods or external services that interact with the cluster directly and are scoped to namespaces.

[Cluster]Role

- Permissions translate to url path. With "" defaulting to core group.
- Resources act as items the role should be granted access to.
- Verbs are the actions the role can perform on the referenced resources.

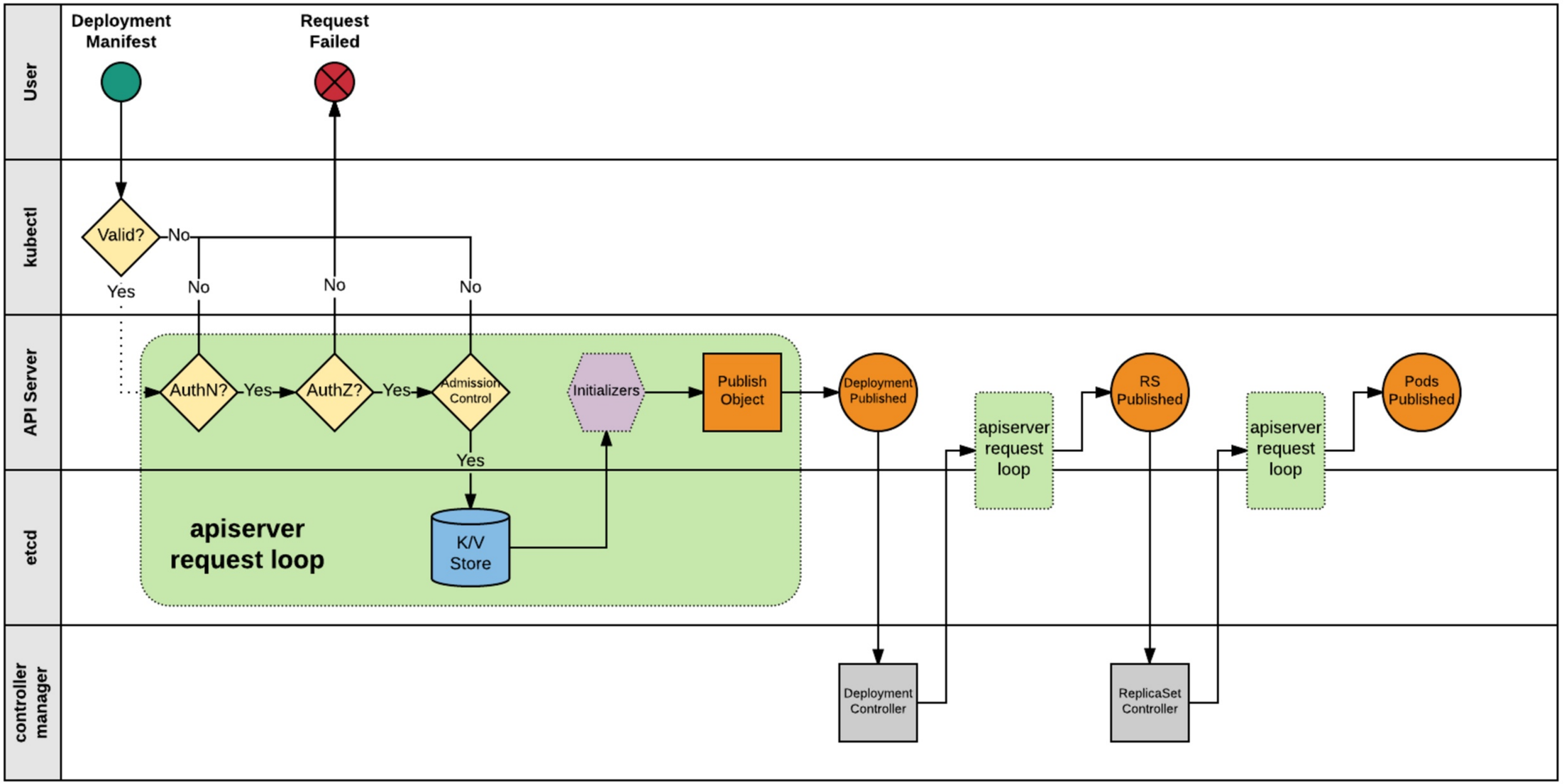
```yaml
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: monitor-things
rules:
- apiGroups: [""]
  Resources: ["services", "endpoints", "pods"]
  verbs: ["get", "list", "watch"]
```

[Cluster]RoleBinding

- **Can reference multiple subjects**
- **Subjects can be of kind:**
  - User
  - Group
  - ServiceAccount
- **roleRef targets a single role only.**

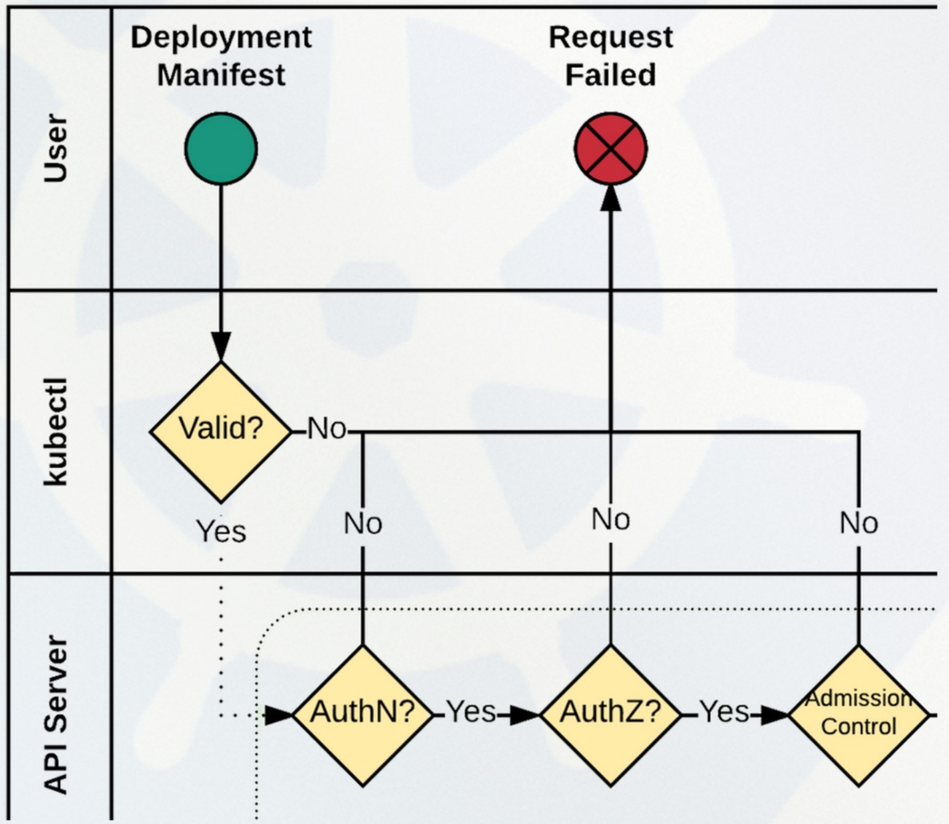
```yaml
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: monitor-things
subjects:
- kind: User
  name: bob
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: monitor-things
  apiGroup: rbac.authorization.k8s.io
```

Deployment


User
kubectl
API Server
etcd
controller manager
Deployment Manifest
Request Failed
Valid?
No
Yes
No
No
No
AuthN?
Yes
AuthZ?
Yes
Admission Control
Yes
Initializers
Publish Object
Deployment Published
apiserver request loop
RS Published
apiserver request loop
Pods Published
apiserver request loop
K/V Store
Deployment Controller
ReplicaSet Controller

Kubectl

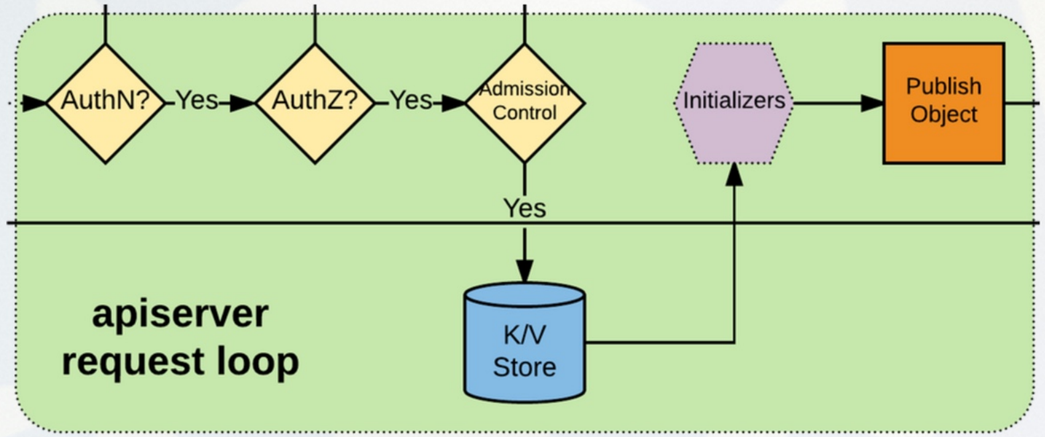
1) Kubectl performs client side validation on manifest (linting).
2) Manifest is prepared and serialized creating a JSON payload.



API Server Request Loop

3) Kubectl authenticates to apiserver via x509, jwt, http auth proxy, other plugins, or http-basic auth.

4) Authorization iterates over available AuthZ sources: Node, ABAC, RBAC, or webhook.



5) AdmissionControl checks resource quotas, other security related checks etc.

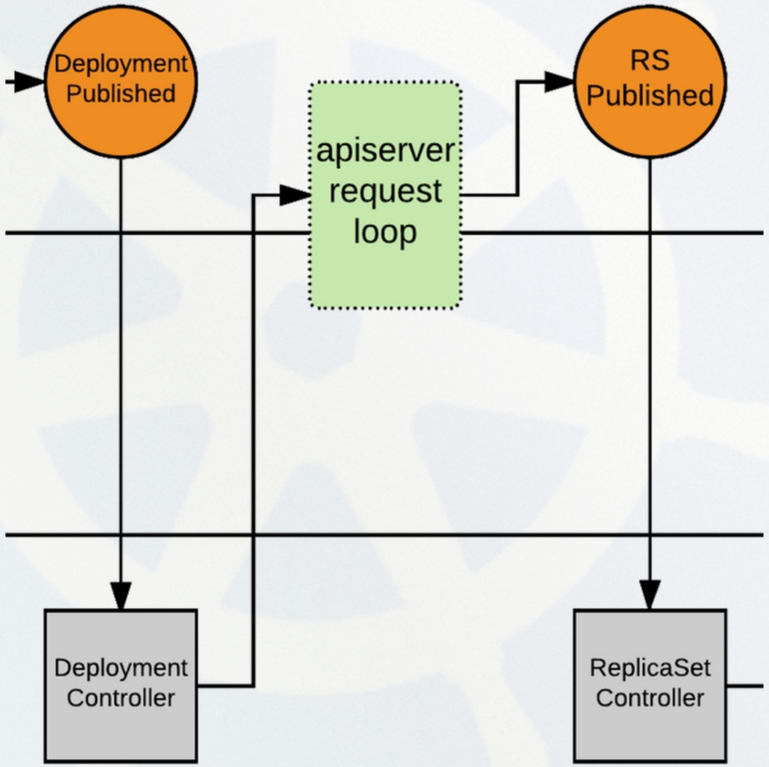
6) Request is stored in etcd.

7) Initializers are given opportunity to mutate request before the object is published.

8) Request is published on apiserver.

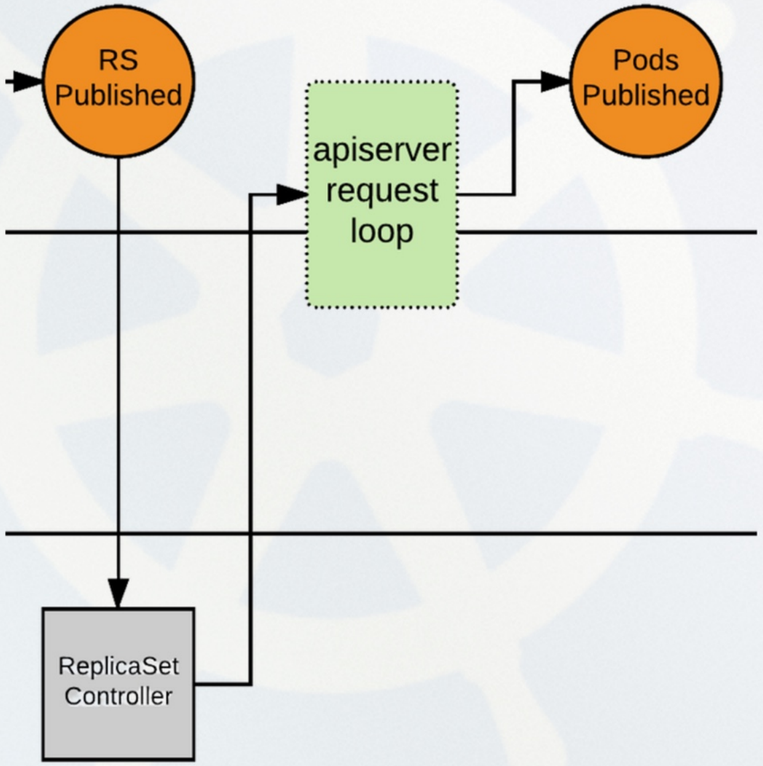
Deployment Controller

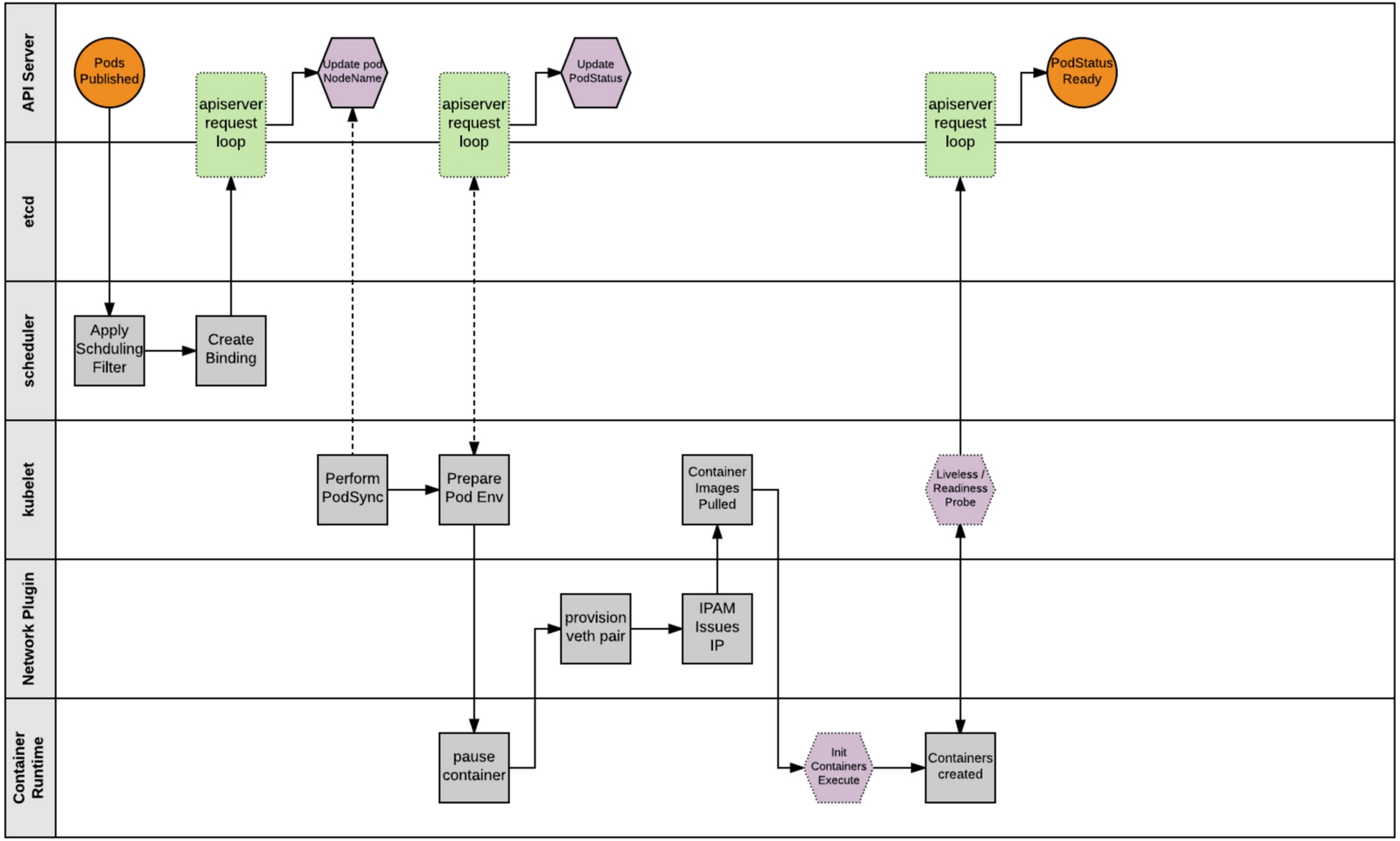
9) Deployment Controller is notified of the new Deployment via callback.
10) Deployment Controller evaluates cluster state and reconciles the desired vs current state and forms a request for the new ReplicaSet.
11) apiserver request loop evaluates Deployment Controller request.
12) ReplicaSet is published.



ReplicaSet Controller

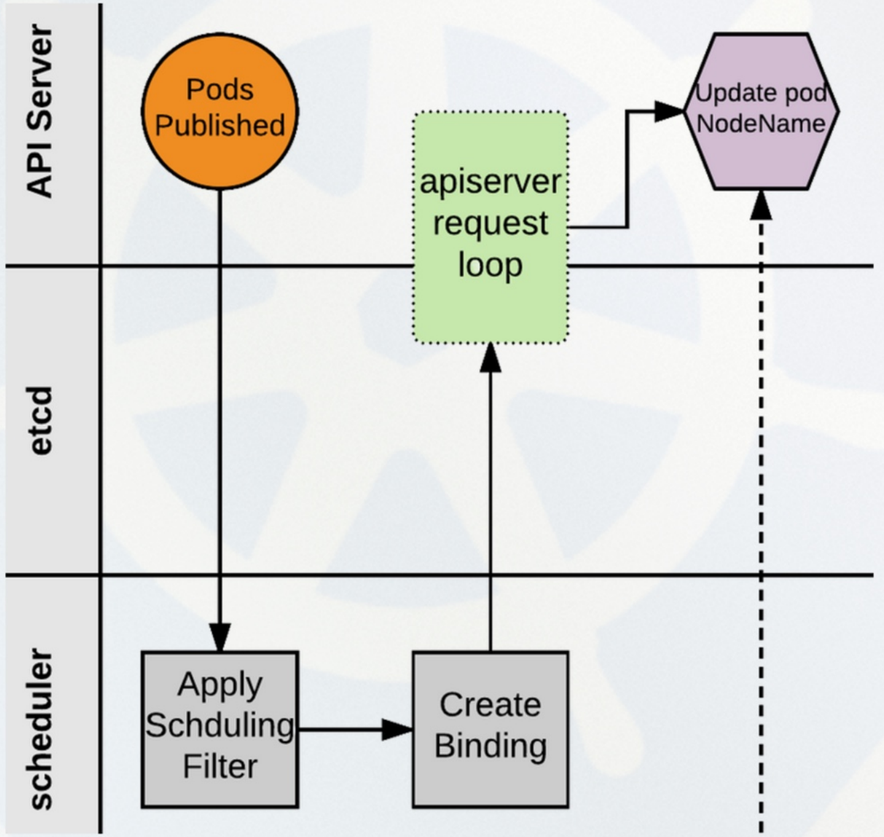
13) ReplicaSet Controller is notified of the new ReplicaSet via callback.
14) ReplicaSet Controller evaluates cluster state and reconciles the desired vs current state and forms a request for the desired amount of pods.
15) apiserver request loop evaluates ReplicaSet Controller request.
16) Pods published, and enter ‘Pending’ phase.




API Server
etcd
scheduler
kubelet
Network Plugin
Container Runtime
Pods Published
apiserver request loop
Update pod NodeName
apiserver request loop
Update PodStatus
apiserver request loop
PodStatus Ready
Apply Schduling Filter
Create Binding
Perform PodSync
Prepare Pod Env
Container Images Pulled
Liveless / Readiness Probe
provision veth pair
IPAM Issues IP
pause container
Init Containers Execute
Containers created

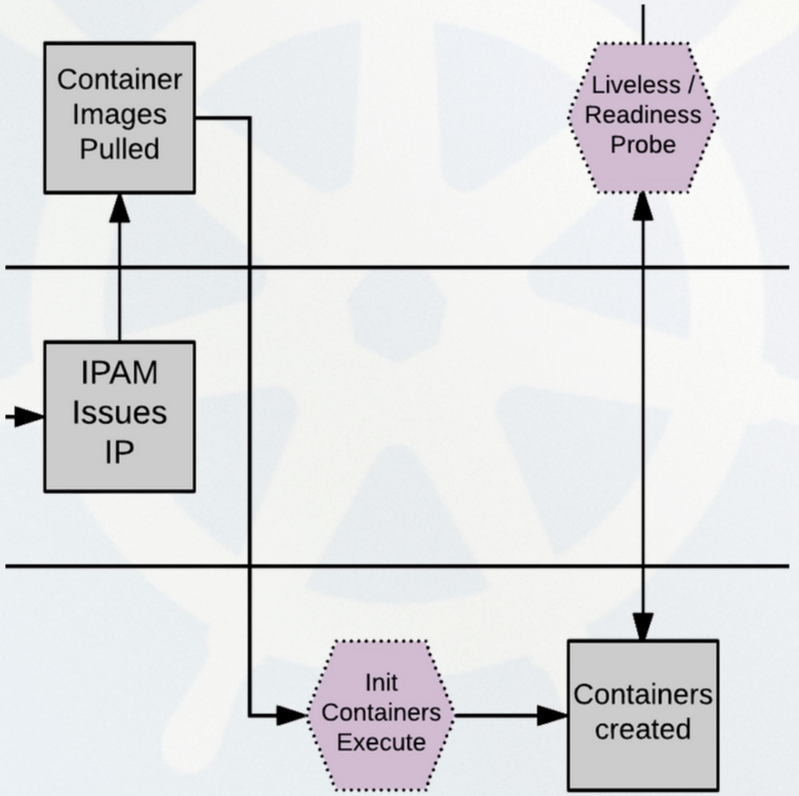
Scheduler

17) Scheduler monitors published pods with no ‘NodeName’ assigned.
18) Applies scheduling rules and filters to find a suitable node to host the Pod.
19) Scheduler creates a binding of Pod to Node and POSTs to apiserver.
20) apiserver request loop evaluates POST request.
21) Pod status is updated with node binding and sets status to ‘*PodScheduled*’.



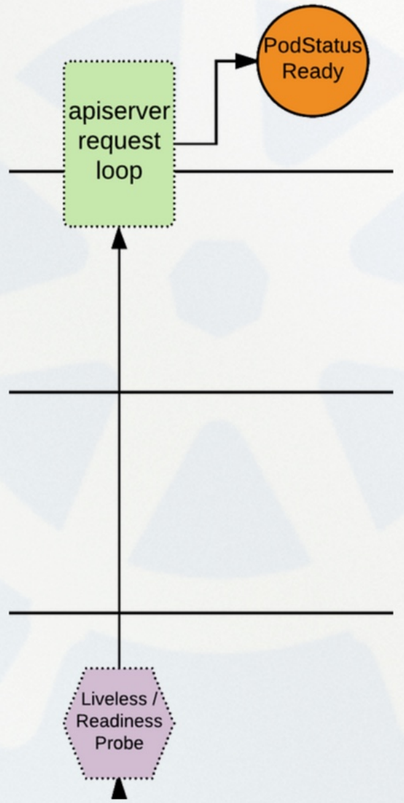
Kubelet - Create Containers

24) Kubelet pulls the container Images.
25) Kubelet first creates and starts any init containers.
26) Once the optional init containers complete, the primary pod containers are started.



The Pods is Deployed!

27) If there are any liveless/readiness probes, these are executed before the PodStatus is updated.
28) If all complete successfully, PodStatus is set to ready and the container has started successfully.



Deploy first Pod

Create Namespace

Create Namespace : *basic-demo*

```
$ kubectl create namespace basic-demo
```

or Create Namespace with YAML as *namespace.yaml*

```
apiVersion: v1
kind: Namespace
metadata:
  name: basic-demo
```

Create Namespace (continue)

Apply *namespace.yaml*

```
$ kubectl apply -f namespace.yaml
```

Check namespace information

```
$ kubectl get namespaces
```

Create Pod

Create pod with *nginx-pod.yaml*

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  namespace: basic-demo
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
```

Apply YAML file

Apply *nginx-pod.yaml*

```
$ kubectl apply -f nginx-pod.yaml
```

Check pod status

```
$ kubectl get pods -n basic-demo
```

Pod Description

Check more pod information

```
$ kubectl describe pod nginx-pod -n basic-demo
```

Create NGINX directly connecting to Pod with port-forward

```
$ kubectl port-forward pod/nginx-pod 8080:80 -n basic-demo
```

Request or open your browser to *http://localhost:8080*
you will see the NGINX welcome page.

Create Service

Create pod with *nginx-service.yaml*

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
  namespace: basic-demo
spec:
  selector:
    app: nginx
  ports:
  - port: 80
    targetPort: 80
  type: ClusterIP
```

Add Pod Label to Match Service

Edit pod *labels* with *nginx-pod.yaml*

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  namespace: basic-demo
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
```

Apply Service & Pod Together

Update pod labels with apply command

```
$ kubectl apply -f nginx-pod.yaml
```

Apply new service

```
$ kubectl apply -f nginx-service.yaml
```

Test Service

Check service

```
$ kubectl get services -n basic-demo
```

Run port-forward service to specific namespace basic-demo

```
$ kubectl port-forward service/nginx-service 8080:80 -n basic-demo
```

Request or open your browser to *http://localhost:8080*
you will see the NGINX welcome page.

Install NGINX Ingress Controller

Check if Ingress Controller is installed:

```
$ kubectl get pods -n ingress-nginx
```

Install with single command line

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.8.2/deploy/static/provider/cloud/deploy.yaml
```

Ref: https://docs.nginx.com/nginx-ingress-controller

Check NGINX Ingress Controller

Wait for NGINX Ingress

```
$ kubectl wait --namespace ingress-nginx \
  --for=condition=ready pod \
  --selector=app.kubernetes.io/component=controller \
  --timeout=120s
```

Check if Ingress Controller is installed:

```
$ kubectl get pods -n ingress-nginx
```

You should now see a Pod with a name starting with *ingress-nginx-controller* and a status of Running.

Create Ingress YAML

Create *nginx-ingress.yaml*

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx-ingress
  namespace: basic-demo
spec:
  rules:
  - host: nginx.k8s.local
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: nginx-service
            port:
              number: 80
```

Test Ingress

Apply *nginx-ingress.yaml*

```
$ kubectl apply -f nginx-ingress.yaml
```

Check ingress

```
$ kubectl get ingress -n basic-demo
```

Access via Hostname

Note: If you want to test on localhost, you may need to edit the hosts file:

For Linux/macOS: /etc/hosts
For Windows: C:\Windows\System32\drivers\etc\hosts

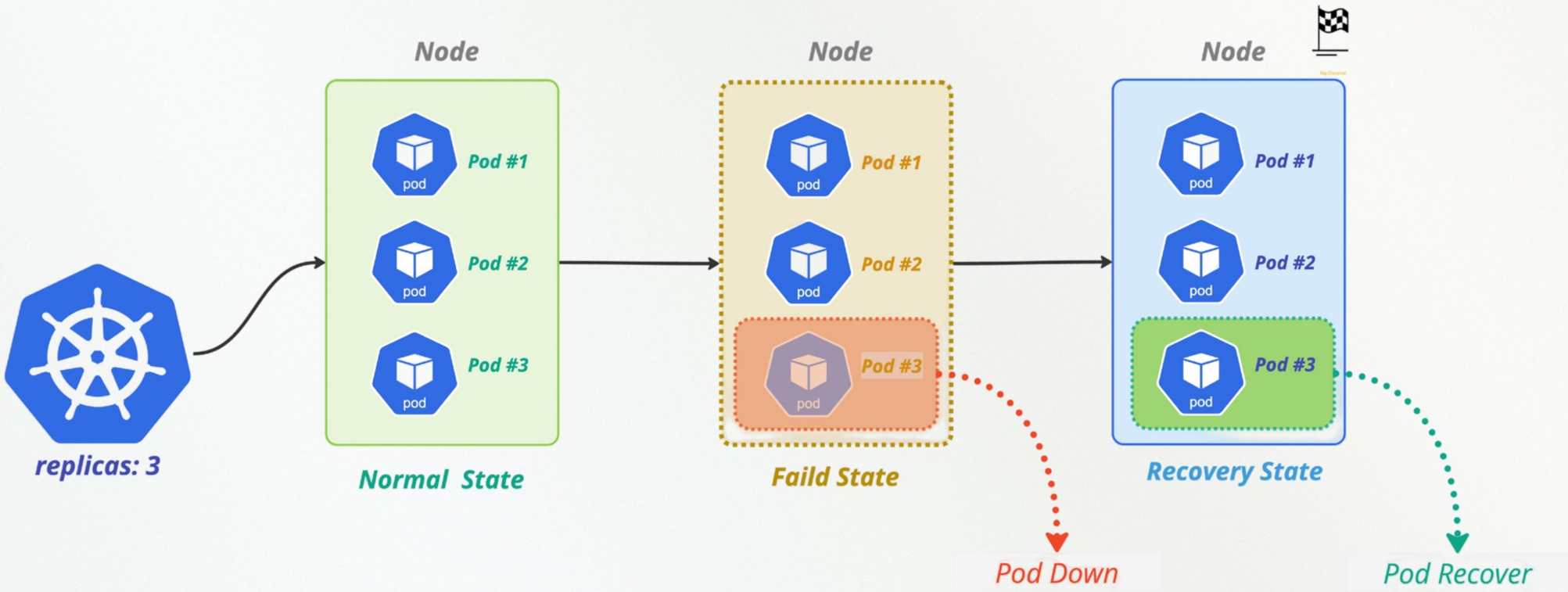
add this line to bottom of file.

```
127.0.0.1    nginx.k8s.local
```

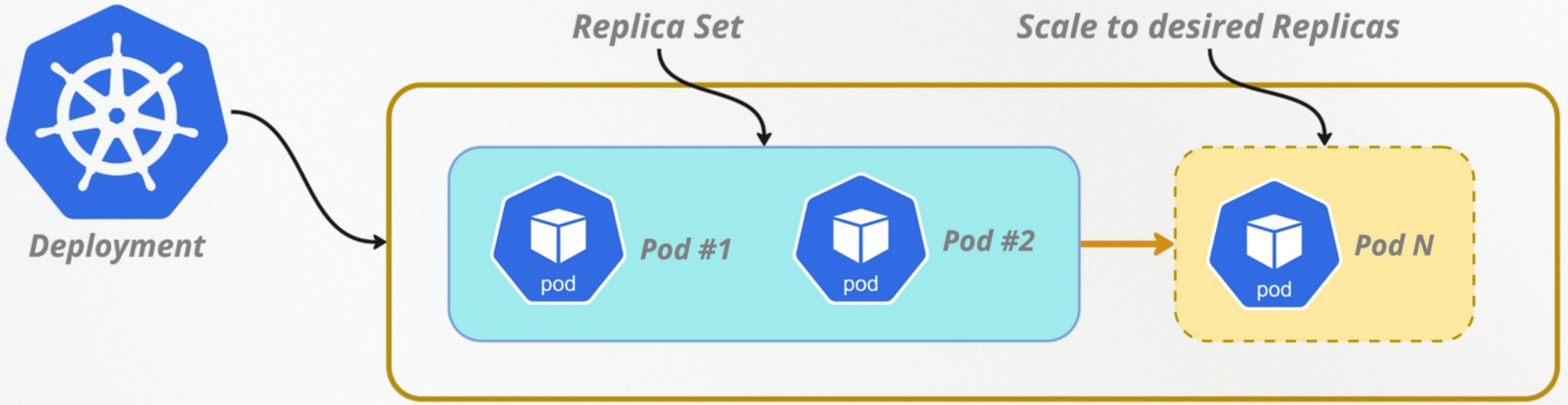
Request or open your browser to *http://nginx.k8s.local*
you will see the NGINX welcome page.

Basic Topology

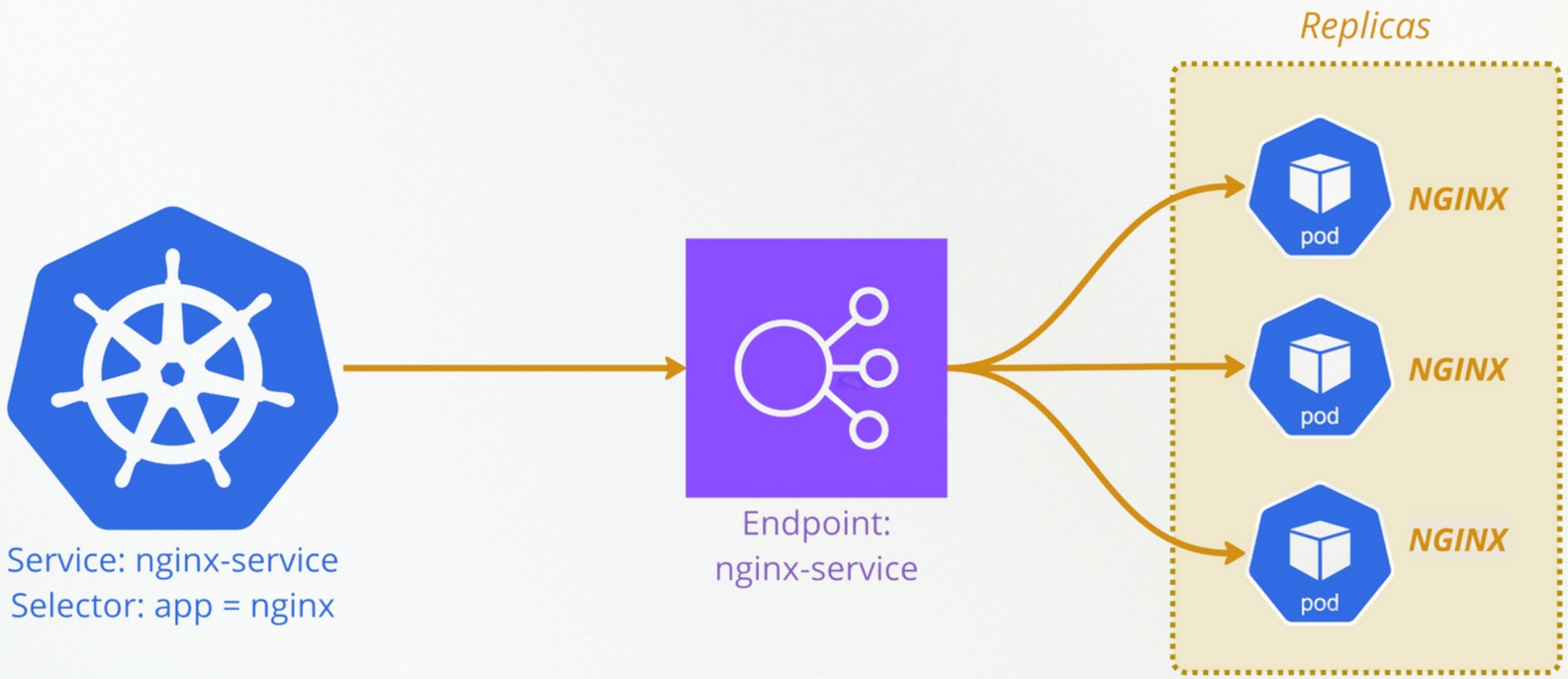
Pod Recover


Node
Node
Node
Pod #1
Pod #2
Pod #3
pod
replicas: 3
Normal State
Faild State
Recovery State
Pod Down
Pod Recover

HPA


Replica Set
Scale to desired Replicas
Deployment
pod
Pod #1
pod
Pod #2
pod
Pod N

Load Balancer



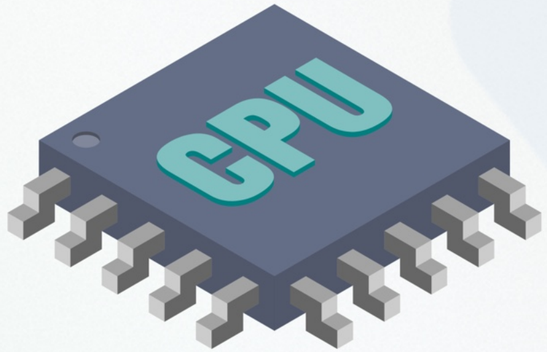
Best Practices

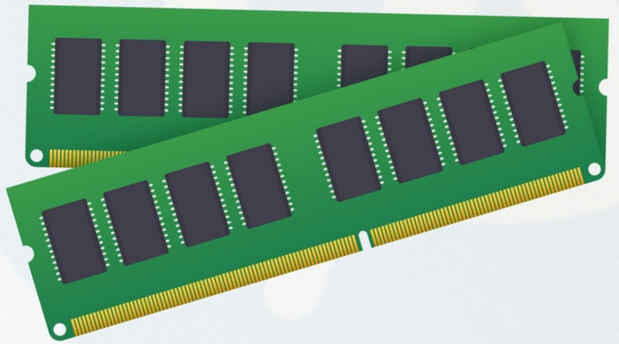
Use Namespaces for Isolation

Namespaces separate applications, prevent resource conflicts, and simplify access control for better multi-tenant management.

Limit Resource Usage per Pod

Defining clear resource boundaries ensures fair usage, avoids overconsumption, and helps maintain cluster stability.

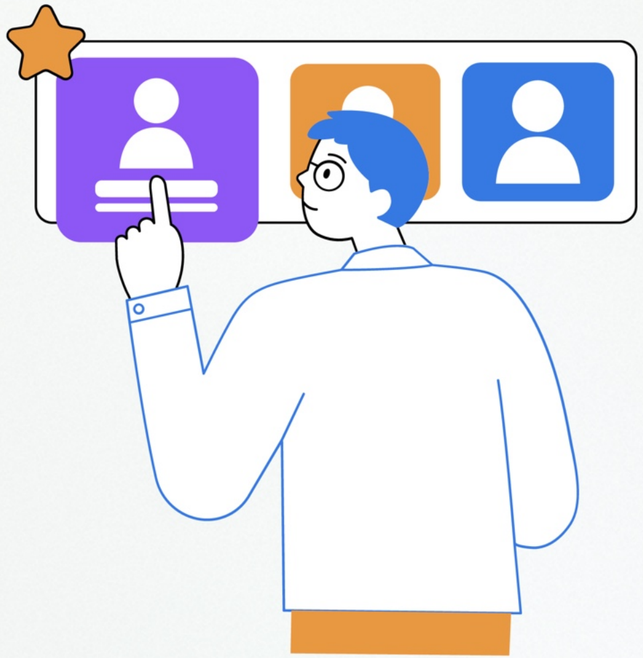




Use Readiness & Liveness Probes

Probes detect unhealthy containers for quick restarts and prevent routing traffic to pods that are not ready, improving app reliability.

Enable RBAC for Security



Role-Based Access Control safeguards cluster resources by limiting what users or services can do, reducing security risks.

Use Network Policies

Network policies define allowed traffic paths, isolating pods and thwarting unauthorized access within the cluster.

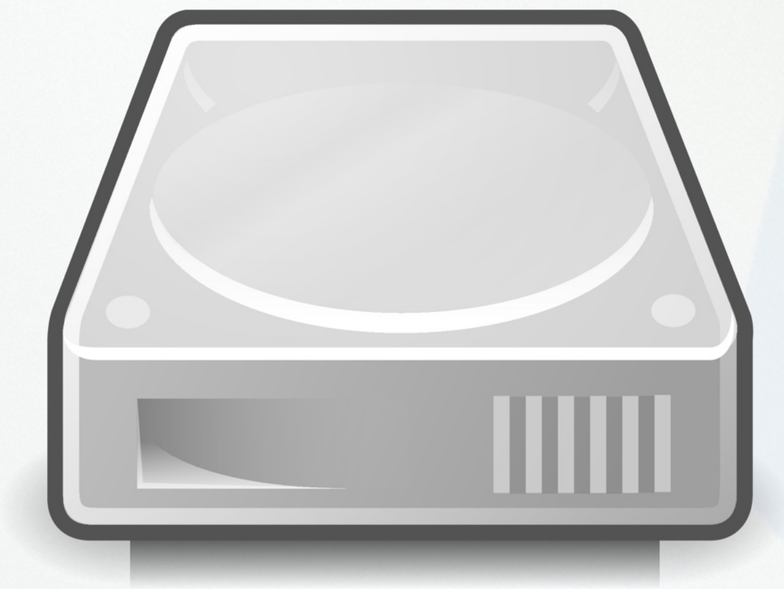
Use Horizontal Pod Autoscaler

HPA automatically adds or removes pods to match workload demands, ensuring efficient resource use and better performance.

Store Configurations in ConfigMaps & Secrets

Externalizing configuration simplifies updates, secures credentials, sensitive data and fosters better environment consistency.

Use Persistent Volumes for Storage



Persistent volumes ensure data durability, preventing loss and simplifying storage management for stateful applications.

Monitor with Prometheus & Grafana

Effective monitoring and visualization help detect bottlenecks, troubleshoot problems faster, and maintain optimal system health.

Regularly Update Kubernetes & Dependencies

Frequent upgrades protect against known vulnerabilities, ensure compatibility with new features, and improve cluster stability.

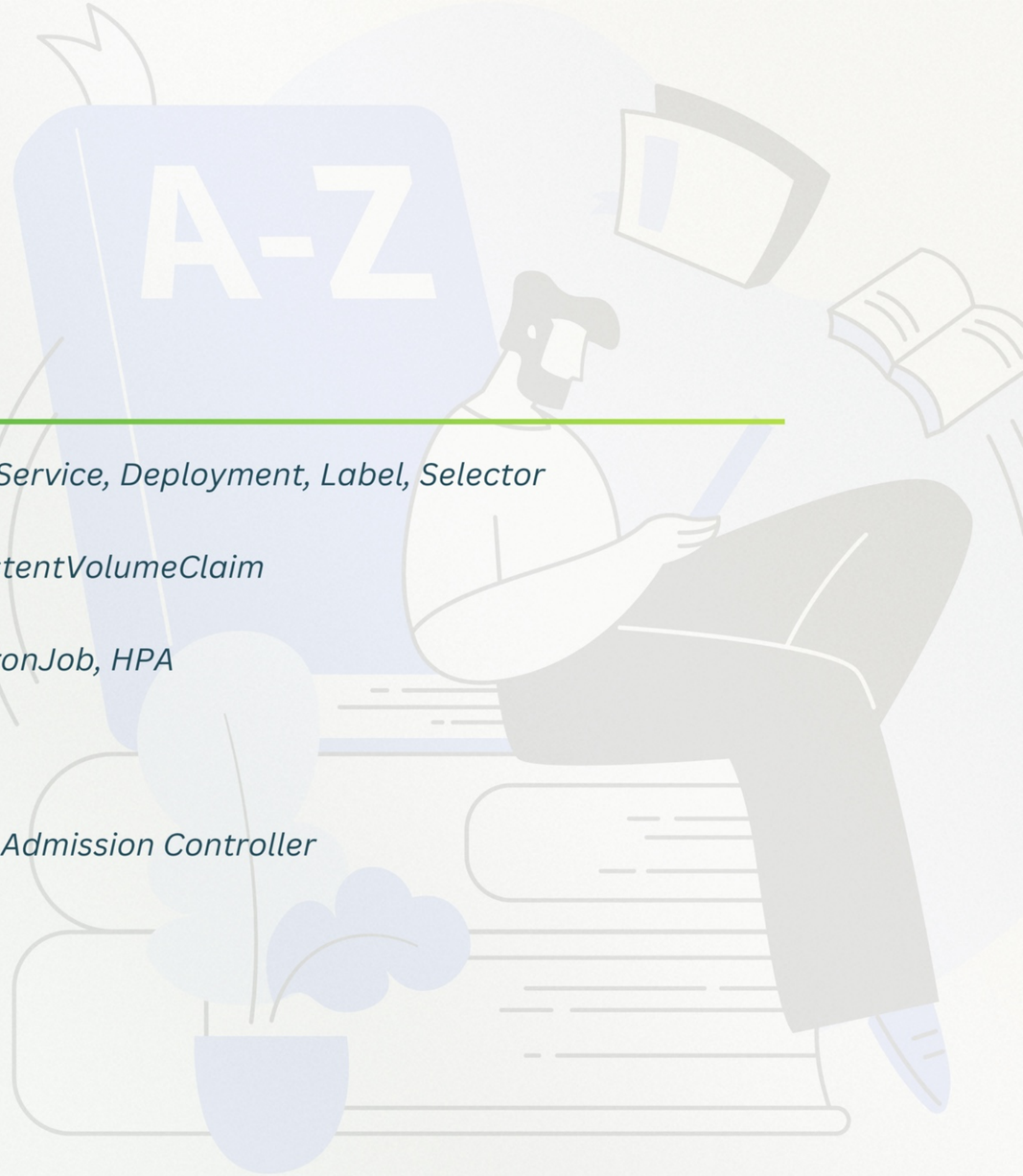
Adopt Declarative Config & GitOps

Keep cluster configs in Git for a single source of truth and simpler change tracking, then let GitOps tools (e.g., ArgoCD, Flux) auto-sync clusters to maintain consistency.

Glossary

K8s Glossary

Category	Terms
Fundamental	*Cluster, Node, Pod, Container, Namespace, Service, Deployment, Label, Selector*
Core Object	*ConfigMap, Secret, PersistentVolume, PersistentVolumeClaim*
Workload	*ReplicaSet, StatefulSet, DaemonSet, Job, CronJob, HPA*
Networking	*Ingress, Network Policy*
Security	*RBAC, Pod Security Policy, Service Account, Admission Controller*
Storage	*StorageClass*
Tool	*Kubectl, Helm, Kustomize*
User Type	*Cluster Operator, Application Developer*
Extension	*CustomResourceDefinition, Operator*



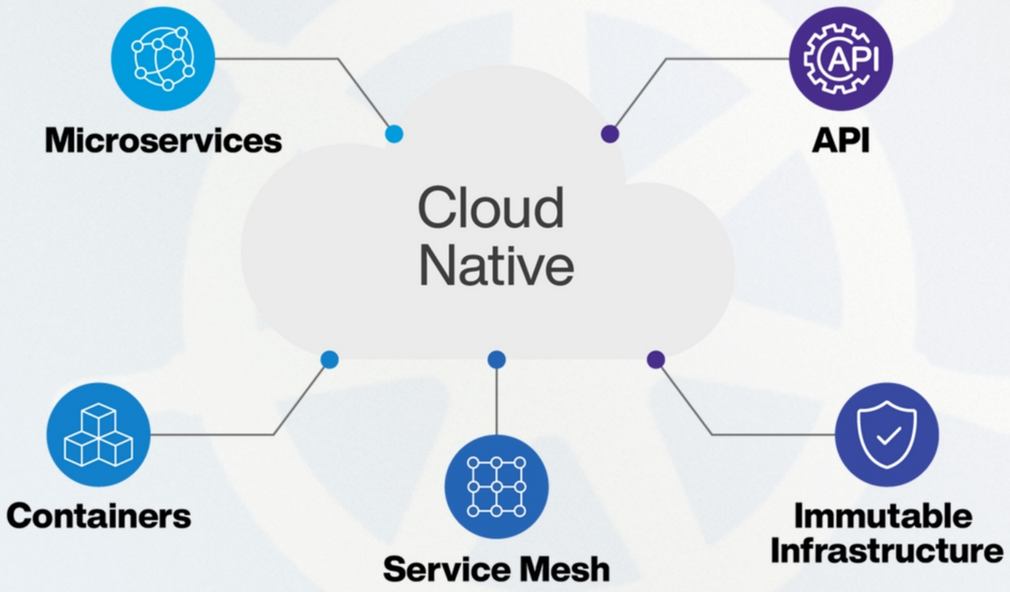
Cloud Native
CNCF

Cloud Native Computing Foundation (CNCF)

The CNCF, home to Kubernetes, drives cloud-native innovation by fostering open-source tools for scalable, resilient systems. It promotes a collaborative ecosystem for containerized, microservices-based apps in dynamic cloud environments.

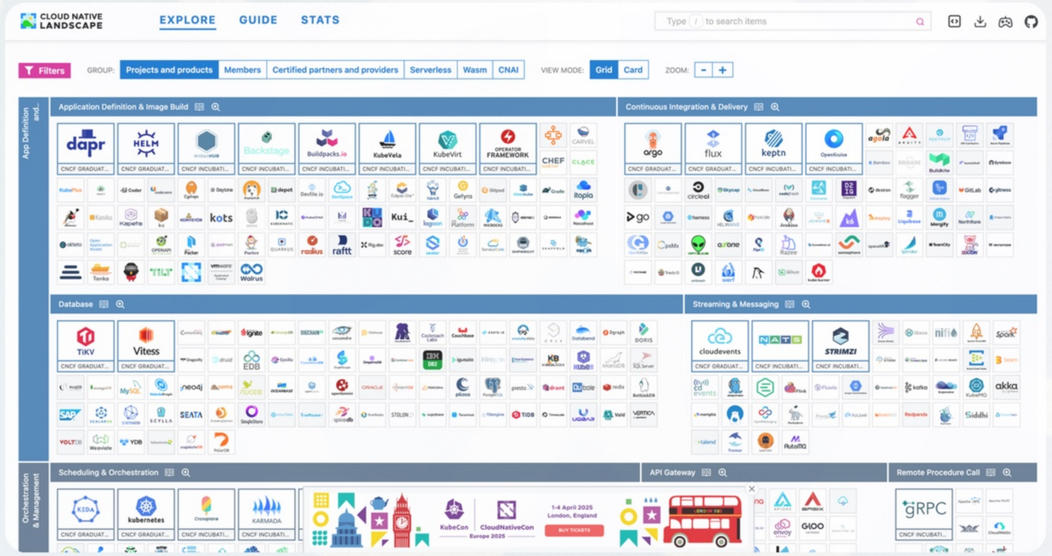
K8s in Cloud Native Core Concept

Kubernetes embodies cloud-native principles, enabling scalable, resilient, and portable applications. It leverages microservices, containers, and automation to optimize resource use and adaptability in dynamic cloud environments.



Exploring into Cloud Native at CNCF

Cloud Native leverages Kubernetes, Prometheus, and Istio for orchestration, monitoring, and service mesh. It embraces microservices, CI/CD, and observability, ensuring scalable, resilient apps with tools like Helm and OpenTelemetry in dynamic cloud ecosystems.



Cloud Native Landscape & DevOps Roadmap

- **CNCF Cloud Native Landscape** - https://landscape.cncf.io
- **Official CNCF** - https://www.cncf.io
- **CNCF GitHub** - https://github.com/cncf
- **CNCF Trail Map** - https://github.com/cncf/trailmap
- **CNCF Project** - https://www.cncf.io/projects
- **CNCF YouTube Channel** - https://www.youtube.com/@cncf)
- **CNCF Blog** - https://www.cncf.io/blog
- **CNCF Slack** - https://slack.cncf.io
- **CNCF Events** - https://www.cncf.io/events
- **CNCF Community** - https://community.cncf.io
- **CNCF Training** - https://www.cncf.io/certification/training
- **DevOps Roadmap** - https://roadmap.sh/devops
- **Kubernetes Official Documentation** - https://kubernetes.io/docs
- **Docker Official Documentation** - https://docs.docker.com
- **HashiCorp Learn** - https://learn.hashicorp.com
- **Microsoft Learn** - https://docs.microsoft.com/en-us/learn/paths/implement-devops-practices
- **Google Cloud** - https://cloud.google.com/solutions/devops
- **AWS Training** - https://aws.amazon.com/training/learn-about/devops
- **Kubernetes Documentation** - https://kubernetes.io/docs/home

DevOps Most Famous YouTube Channel

- **ByteByteGo -** https://www.youtube.com/@ByteByteGo
- **TechWorldwithNana -** https://www.youtube.com/@TechWorldwithNana
- **Simplilearn -** https://www.youtube.com/@SimplilearnOfficial
- **AbhishekVeeramalla -** https://www.youtube.com/@AbhishekVeeramalla
- **tutoriaLinux -** https://www.youtube.com/@tutoriaLinux
- **KodeKloud -** https://www.youtube.com/@KodeKloud
- **AzureDevOps -** https://www.youtube.com/@AzureDevOps
- **DevOpsJourne -** https://www.youtube.com/@DevOpsJourne
- **thelinuxfoundation -** https://www.youtube.com/user/thelinuxfoundation

Going Further

- https://docs.nginx.com/nginx-ingress-controller
- https://github.com/LondheShubham153/kubestarter
- https://github.com/kubernetes/kubectl
- https://github.com/hashicorp/vault
- https://github.com/wmariuss/awesome-devops
- https://github.com/milanm/DevOps-Roadmap
- https://github.com/kamranahmedse/developer-roadmap
- https://github.com/bregman-arie/devops-exercises

Workshop Materials

https://github.com/racksync/devops-workshop

DevOps Workshop

Welcome to the DevOps Workshop repository. This repository contains tutorials and guides on various DevOps tools and practices.

📚 Table of Contents

1. Debian Linux Tutorial - Introduction to Linux commands, file system, package management, and more
2. Docker Tutorial - Learn Docker, containers, images, and Docker Compose
3. Git Tutorial - Version control, branches, commits, and collaboration
4. CI/CD with GitHub Actions - Continuous Integration/Continuous Deployment with GitHub Actions
5. CI/CD with GitLab - Implementing CI/CD pipelines with GitLab
6. CI/CD with BitBucket - Setting up CI/CD workflows in BitBucket
7. CI/CD Testing & Linting - Advanced testing and linting in CI/CD pipelines
8. Container Orchestration - Introduction to Kubernetes and container orchestration
9. Project Monitoring - Tools and techniques for monitoring applications
10. Load Balancing & CDN - Implementing load balancing and content delivery networks
11. Deploy on Platforms - Deploying applications to various cloud platforms
12. Load Testing - Performance testing and load testing techniques
13. Cloud Native Computing Foundation - Introduction to CNCF and cloud native technologies

RACKSYNC CO., LTD.

Address : 135/5 Moo.1, Khuntalae Sub-District, Muang District, Surat Thani, 84100

info@racksync.com

https://racksync.com

FB : https://www.fb.com/racksync

Tel : 08 5880 8885