Applied Virtual Networks

COMP 4912

Instructor: **Dawood Sajjadi**

PhD, SMIEEE, CISSP

ssajjaditorshizi@bcit.ca

Winter-Spring 2025

Week #4

BCIT™

YAML Intro

```
# What does YAML mean?
YAML:
    - Y: YAML
    - A: Ain't
    - M: Markup
    - L: Language
```

RECAP

YAML is a human-readable data-serialization format.
Commonly used for configuration files in Cloud Applications.

YAML is designed to be friendly to people working with data and achieves "unique cleanness" by minimizing the use of structural characters, allowing the data to appear in a **natural and meaningful way**.

YAML achieves easy inspection of data's structures by using indentation-based scoping (similar to Python).

XML	JSON	YAML

<Servers>
<Server>
<name>Server1</name>
<owner>John</owner>
<created>123456</created>
<status>active</status>
</Server>
</Servers>
	{
	Servers: [
	{
	name: Server1,
	owner: John,
	created: 123456,
	status: active
	}
	]
	}
		Servers:
		- name: Server1
		owner: John
		created: 123456
		status: active

BCIT™

Docker Compose and YAML

RECAP

Setup a web server using Nginx and a static HTML website served from a local directory.
This example demonstrates key Docker Compose concepts like defining services, volumes, and ports.

docker-compose.yml

```yaml
version: '3.9'

services:
  web:
    image: nginx:latest
    ports:
      - "8080:80"  # Map port 8080 on the host to port 80 in the container
    volumes:
      - ./html:/usr/share/nginx/html:ro   # Mount the local html/ directory to Nginx's web root
```

Index.html

```html
<!DOCTYPE html>
<html lang="en">
<body>
    <h1>Welcome to My Simple Website!</h1>
    <p>Served using Docker Compose with Nginx.</p>
</body>
</html>
```

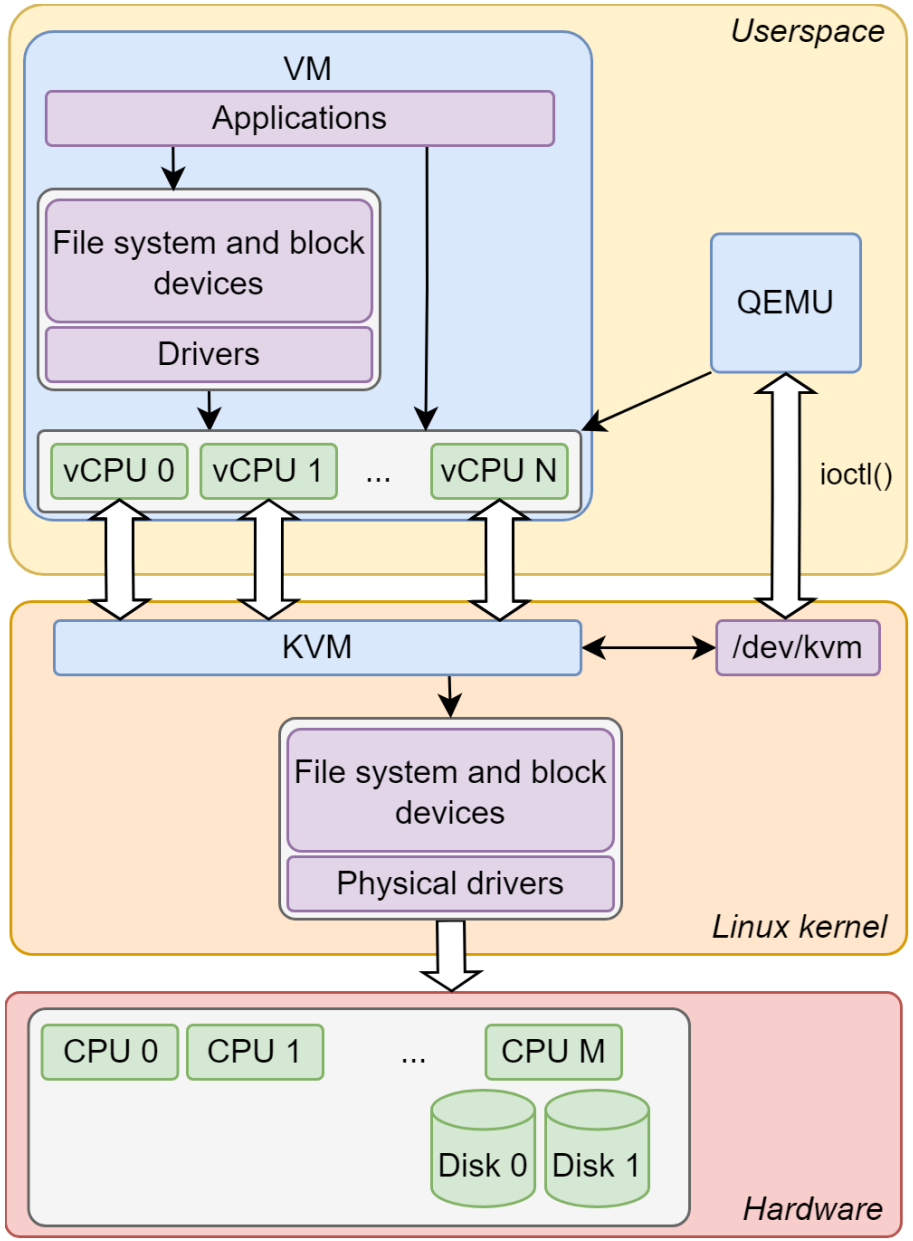
$ docker-compose up -d --build → Access your website via http://localhost:8000

BCIT™

KVM (Kernel-based Virtual Machine)

RECAP

- ✓ **KVM** is the lowest component in the stack and as a Linux kernel module, interacts directly with the hardware.
- ✓ It makes use of hardware-assisted virtualization supported by the CPU, in order to allow **Hardware-based Full Virtualization**.
- ✓ **QEMU** sits above **KVM** and complements its functionality.
- ✓ **QEMU** manages the processes of the guest systems and emulates devices being passed to them.
- ✓ **Libvirt** is the library that is used in order to control **QEMU** and manage the Virtual Machines (VMs).



BCIT™

Virsh (Virtual Shell)

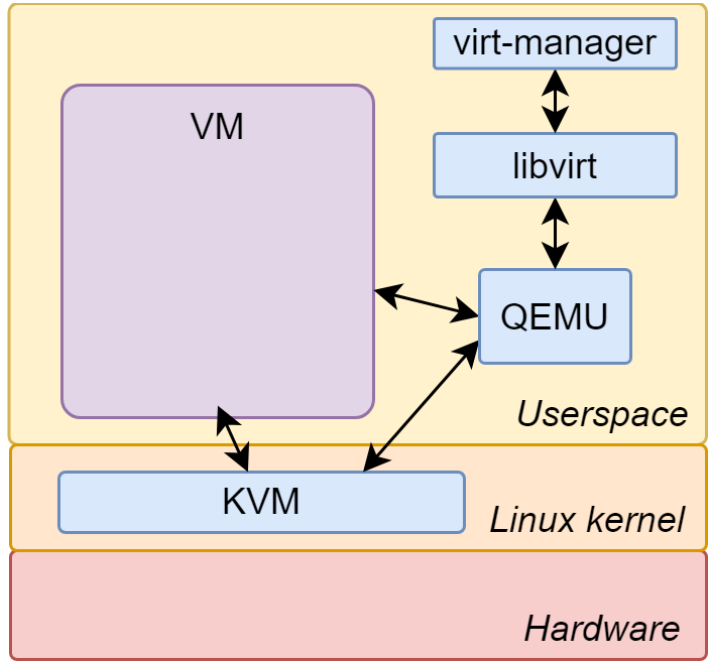
RECAP

✓ **Virsh** is a command-line interface for managing VMs that are handled by the **libvirt** library. It acts as a management layer on top of **KVM**, **QEMU**, or other hypervisors.

✓ **Virsh** communicates with the **libvirt** daemon (**libvirtd**), which interfaces with virtualization back-ends like KVM/QEMU, or LXC. It provides a higher-level, user-friendly way to manage VMs.

KEY FEATURES

- Manage VMs: start, stop, list, pause, destroy.
- Define virtual machines via XML configuration files.
- Connect to remote hypervisors.
- Works with multiple hypervisors (KVM/QEMU, VMware ESXi, etc.).
- Provides more granular control of VMs and their resources.



BCIT™

LXC & LXD (LinuX Containers & LinuX Daemon)

RECAP

LXC: Lightweight Containers

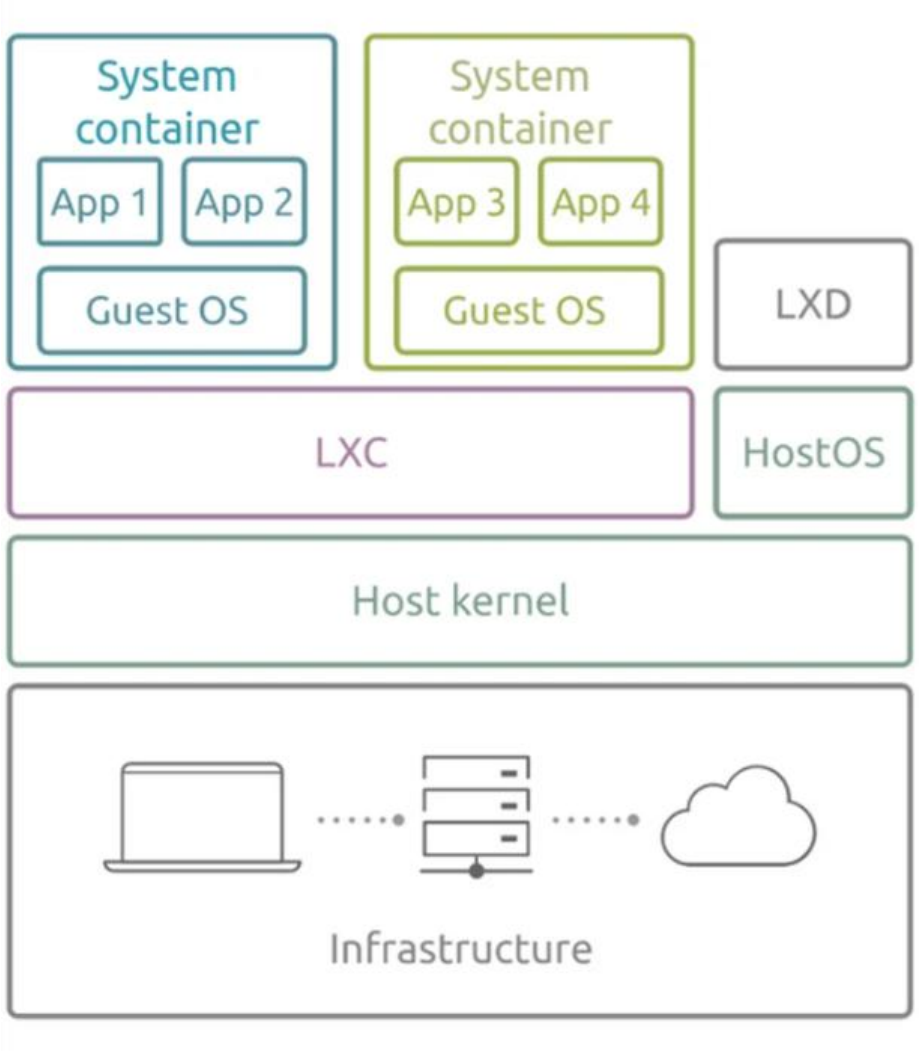
enables you to run applications or full Linux distributions in isolated environments without the overhead of full virtualization.

- OS-level virtualization.
- Containers share the host kernel but run in isolated user spaces.
- Fast startup times and low resource overhead.
- Ideal for testing or running micro-services.

LXD: A System Container Manager

LXD is a container manager built on top of LXC, designed to provide a simple and easy-to-use interface for managing system containers.

- Provides higher-level management features like live migration/snapshots/cloning.
- Can manage both **containers** and **virtual machines** (using QEMU/KVM).
- Supports **multi-architecture** virtual machines (e.g., x86, ARM).
- REST API available for automation and cloud management.



BCIT™

https://documentation.ubuntu.com/lxd/en/latest/explanation/lxd_lxc/

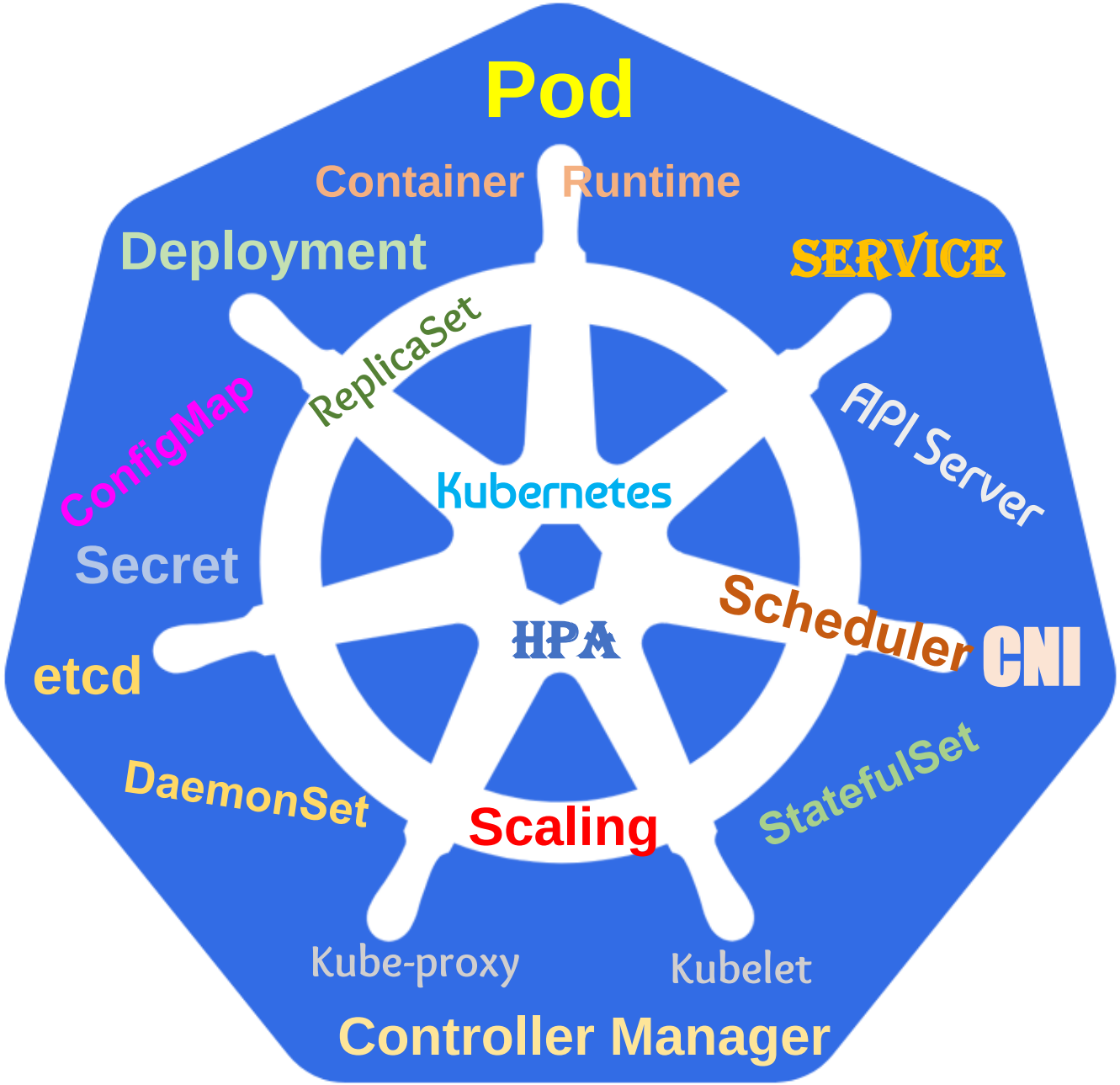
https://canonical.com/lxd

Learning Outcomes of Week #4

1. When/Why we need Container Orchestration?
2. Explain Kubernetes Architecture and its components
3. Describe POD and Core concepts of Kubernetes for service provisioning
4. Explain basic Networking and Storage management in Kubernetes
5. How scaling mechanism works in Kubernetes to handle varying workloads?

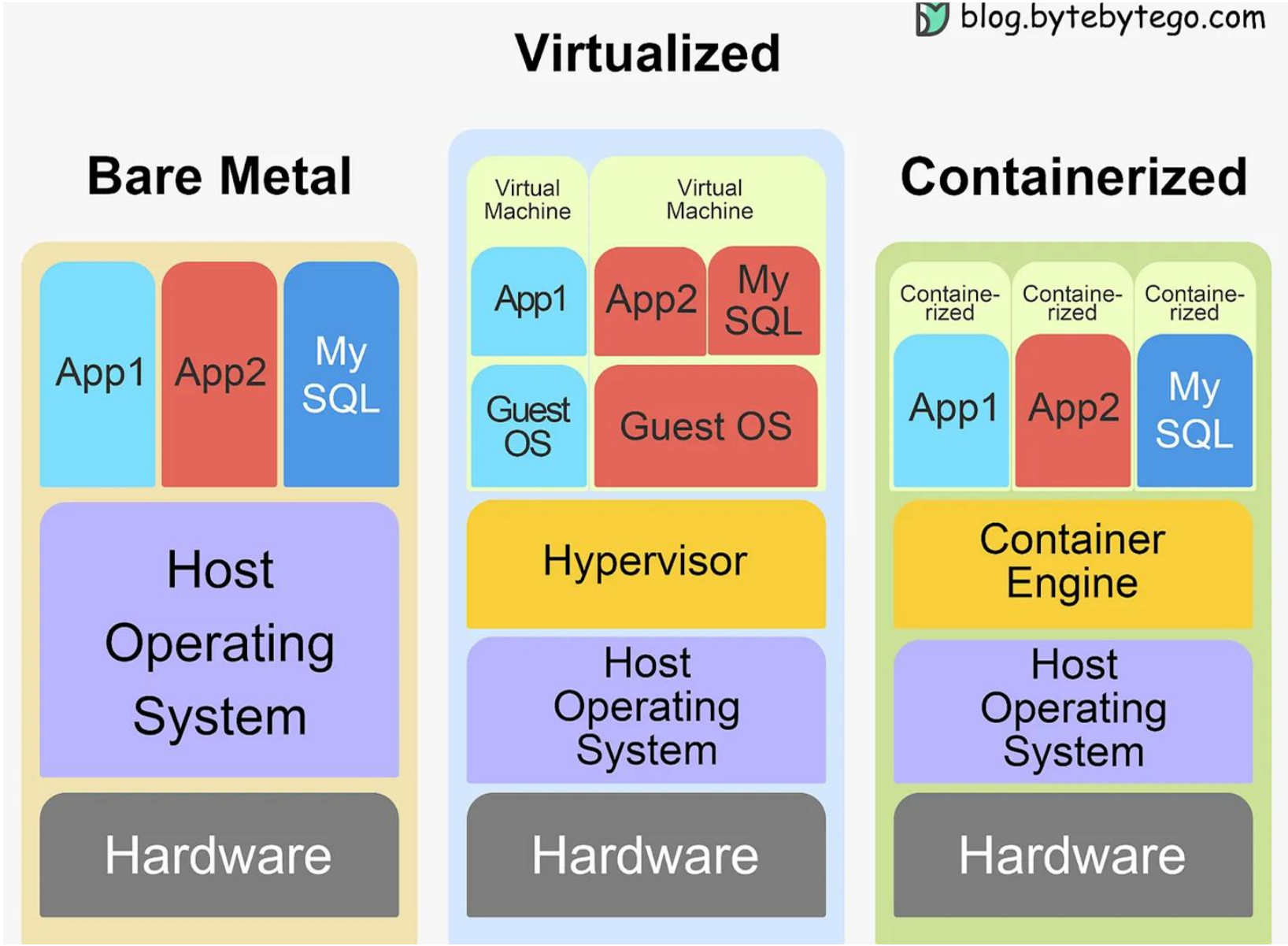
BCIT™

Learning Outcomes of Week #4



BCIT™

Why Need Kubernetes?

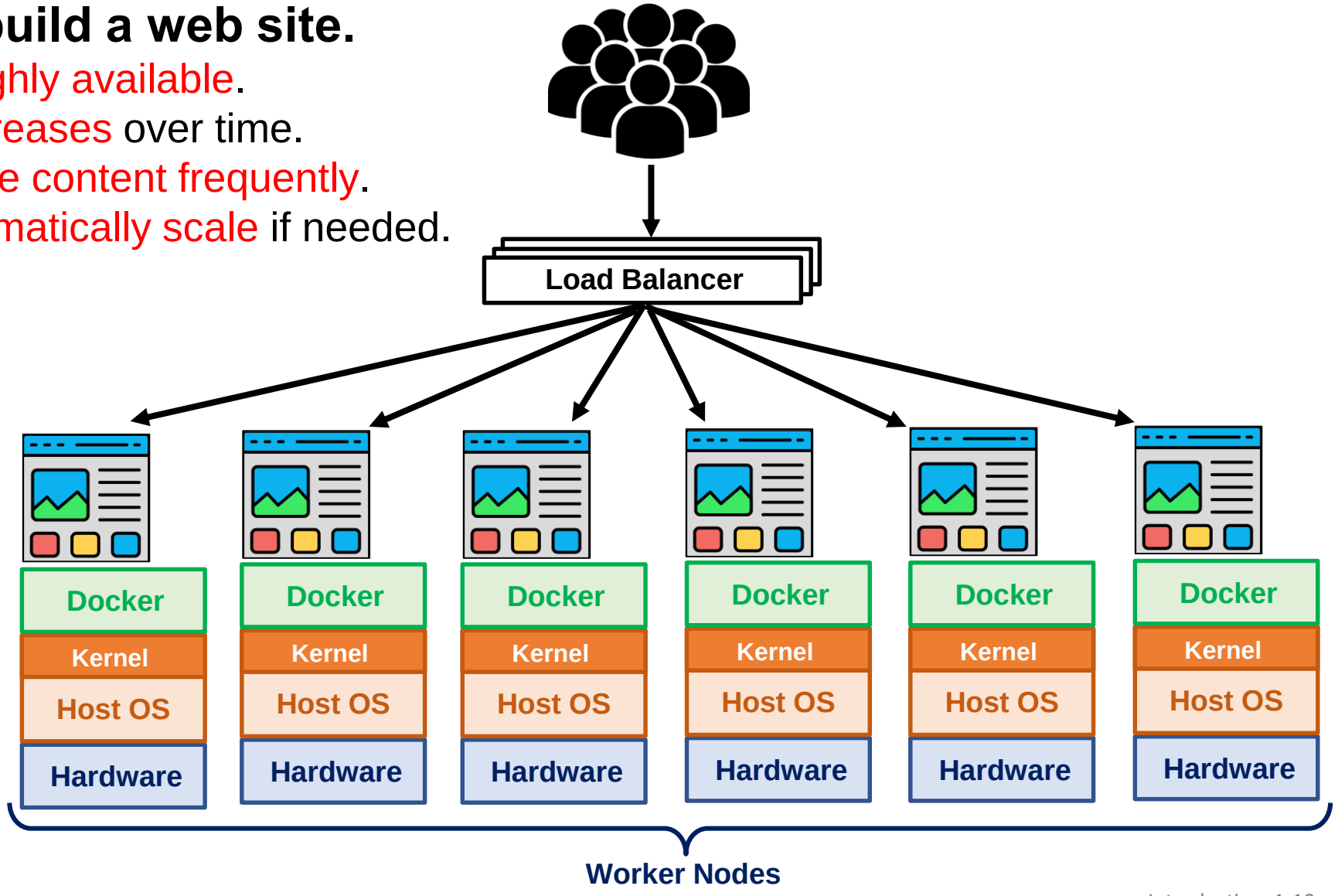


BCIT™

Why Need Kubernetes?

Let's say you want to build a web site.

- The web site should be highly available.
- The number of visitors increases over time.
- You need to update the site content frequently.
- The web site need to automatically scale if needed.



BCIT™

Kubernetes (K8s) is an open-source container orchestration engine that can manage containerized applications.

maintained by the Cloud Native Computing Foundation (**CNCF**).

✓ **Origins at Google (2003–2014):**
- Inspired by Google's internal cluster management systems, Borg & Omega.
- Provided a foundation for container orchestration best practices.

✓ **Open-Sourcing Kubernetes (2014):**
- Released by Google under the Cloud Native Computing Foundation (CNCF).
- Key event that drove community adoption and innovation.

✓ **Growth & Community (2015–2019):**
- Rapid adoption by major cloud providers (AWS, Azure, GCP).
- Expanding ecosystem of tools, integrations, and managed K8s services.

✓ **Mainstream Adoption (2019–Present):**
- De facto standard for container orchestration.
- Evolving to support advanced use cases (e.g., edge computing, AI/ML workloads).

containerd

BCIT™

Why Need Kubernetes?

Applications holy grail:

- ✓ Developers should not care about Servers, Networking, and Storage
- ✓ One-click provisioning (Automate Deployment)
- ✓ High-Availability
- ✓ Load-Balancing
- ✓ Auto-Scaling

- Apps are easier to deploy but harder to maintain and scale over time.
- Apps can be stateful or stateless with different computing, storage, and networking requirements.
- Apps are often managed by DevOps teams that have high-level API requirements rather than low-level VM or physical machine requirements.

BCIT™

Why Need Kubernetes?

5 Key Reasons

1. **Simplified App Deployment**: all nodes are treated as a single platform, developers don't need to know anything about the servers.
2. **Simplified App Development**: developers and leverage the features mentioned above to create fault-tolerant apps.
3. **Better Hardware Utilization**: hardware resources utilization efficiency
4. **Self-Healing via Health Monitoring**: Kubernetes monitors both the App and nodes it runs on and automatically reschedules them to other nodes.
5. **Auto-Scaling**: based on monitoring, Kubernetes can automatically scale Apps in reaction to load spikes.

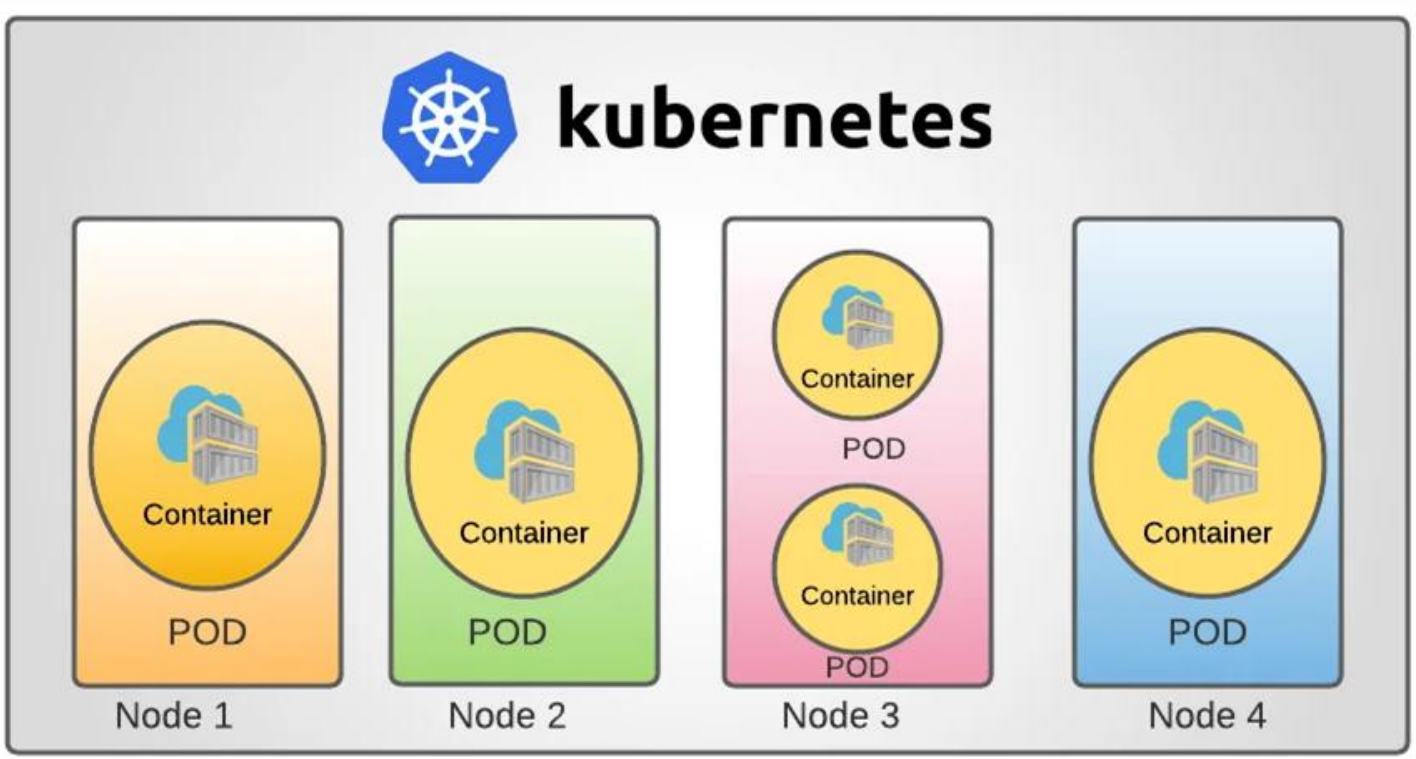
BCIT™

Containers and PODs (Kubernetes Architecture)

The Containers are encapsulated into a Kubernetes object known as PODs.

A POD is the smallest and simplest deployable unit in Kubernetes.

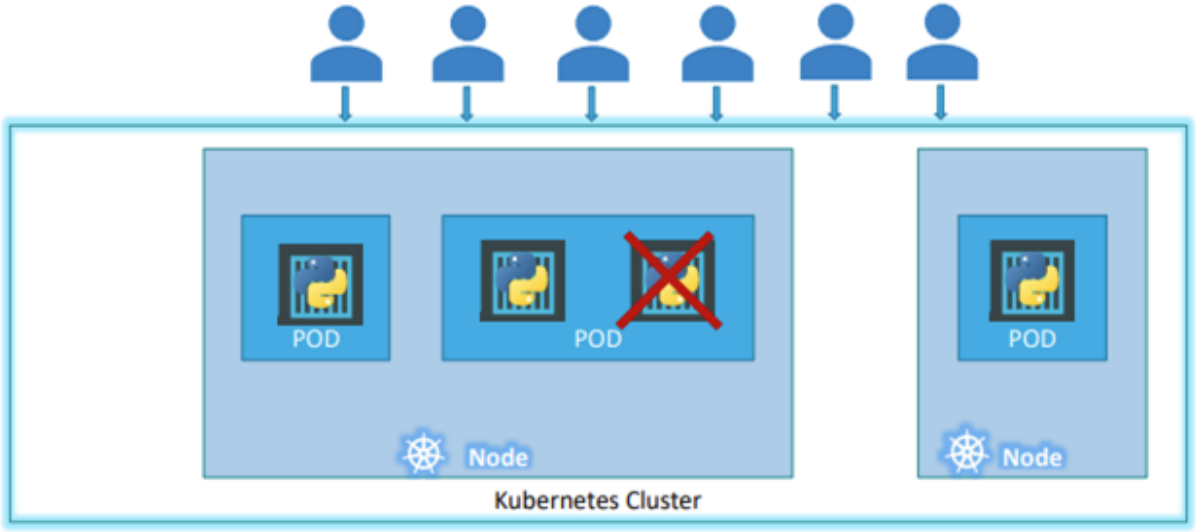
A POD is supposed to be a single instance of an Application.

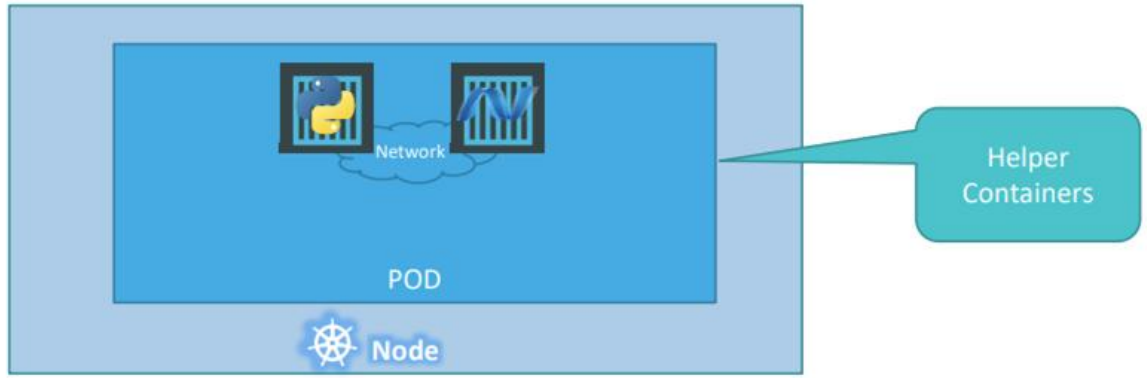


Pods
have a
One-to-One
relationship with
Containers
running your Apps.

BCIT™

Containers and PODs (Kubernetes Architecture)





Multi-Container PODs

Pod is the smallest unit of **computation** & **replication** in K8s. If your application needs to scale up, you simply increase number of pods.

Therefore, if you have multiple container running in the same pod and you want just to scale one of them, the other containers will be scaled too which will lead to waste of resources, so have to keep the POD as small as possible (only one process and its **helpers**)

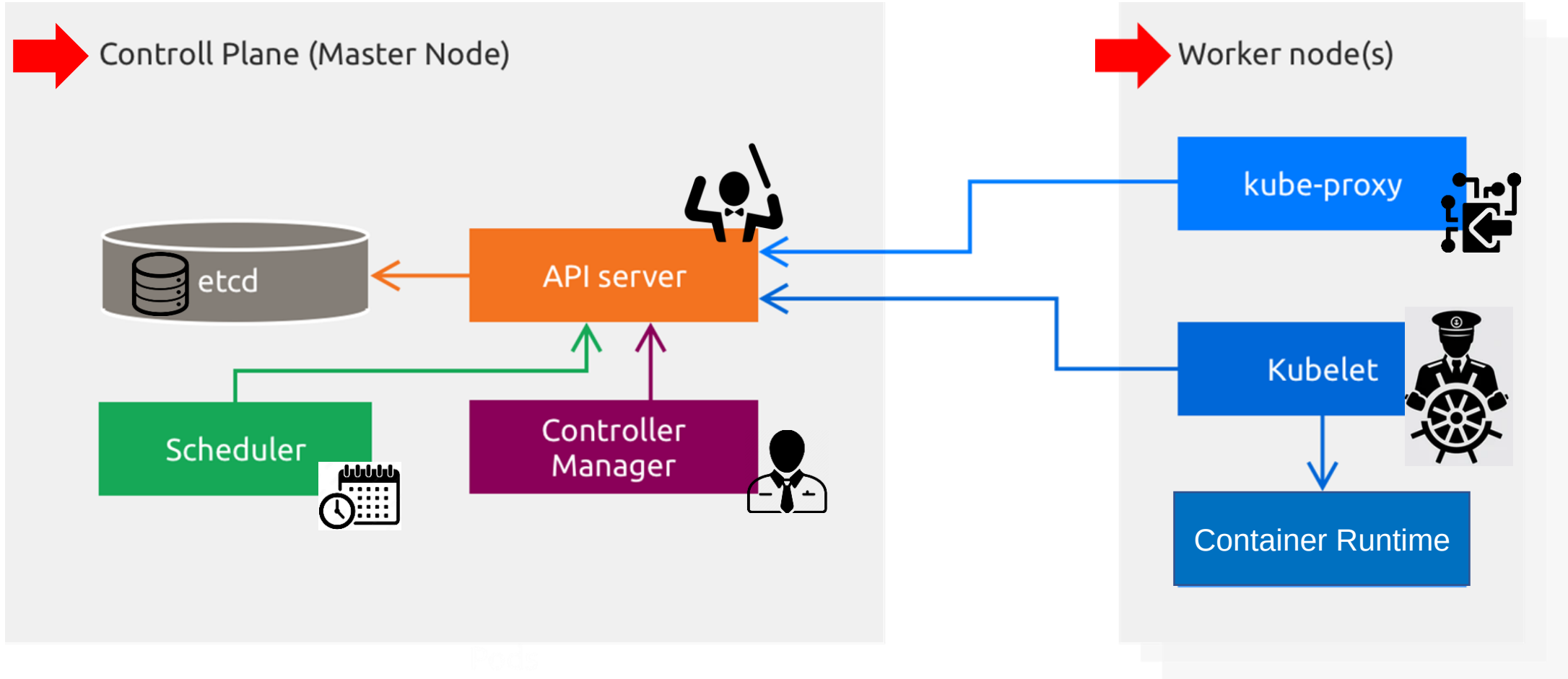
Pods are **ephemeral** in nature, when it is deleted it will lose all the state and associated data if it is stored locally. To keep the data, we need to create Persistent Volume Claim.

NEVER TRY TO SCALE YOUR APP BY ADDING MORE CONTAINERS INTO AN EXISTING POD.

BCIT

https://kodekloud.com/wp-content/uploads/2020/11/Kubernetes-for-Beginners.pdf

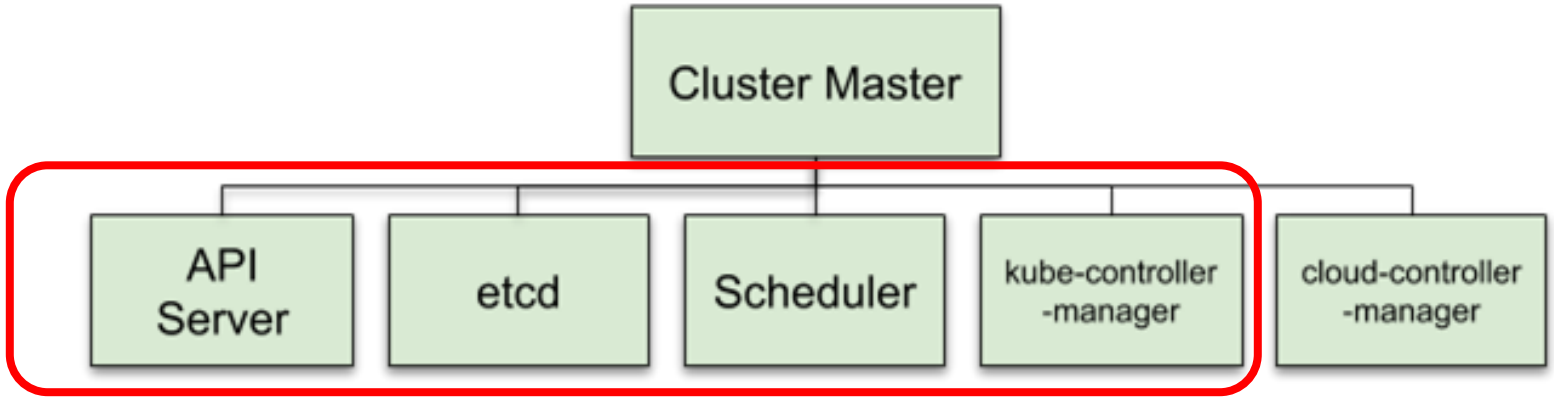
Kubernetes Architecture



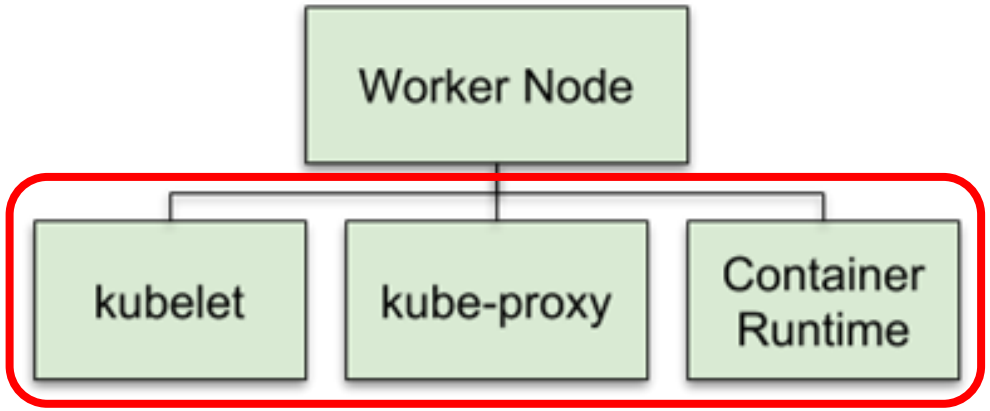
BCIT™

Kubernetes Architecture

Master Node Sub-processes

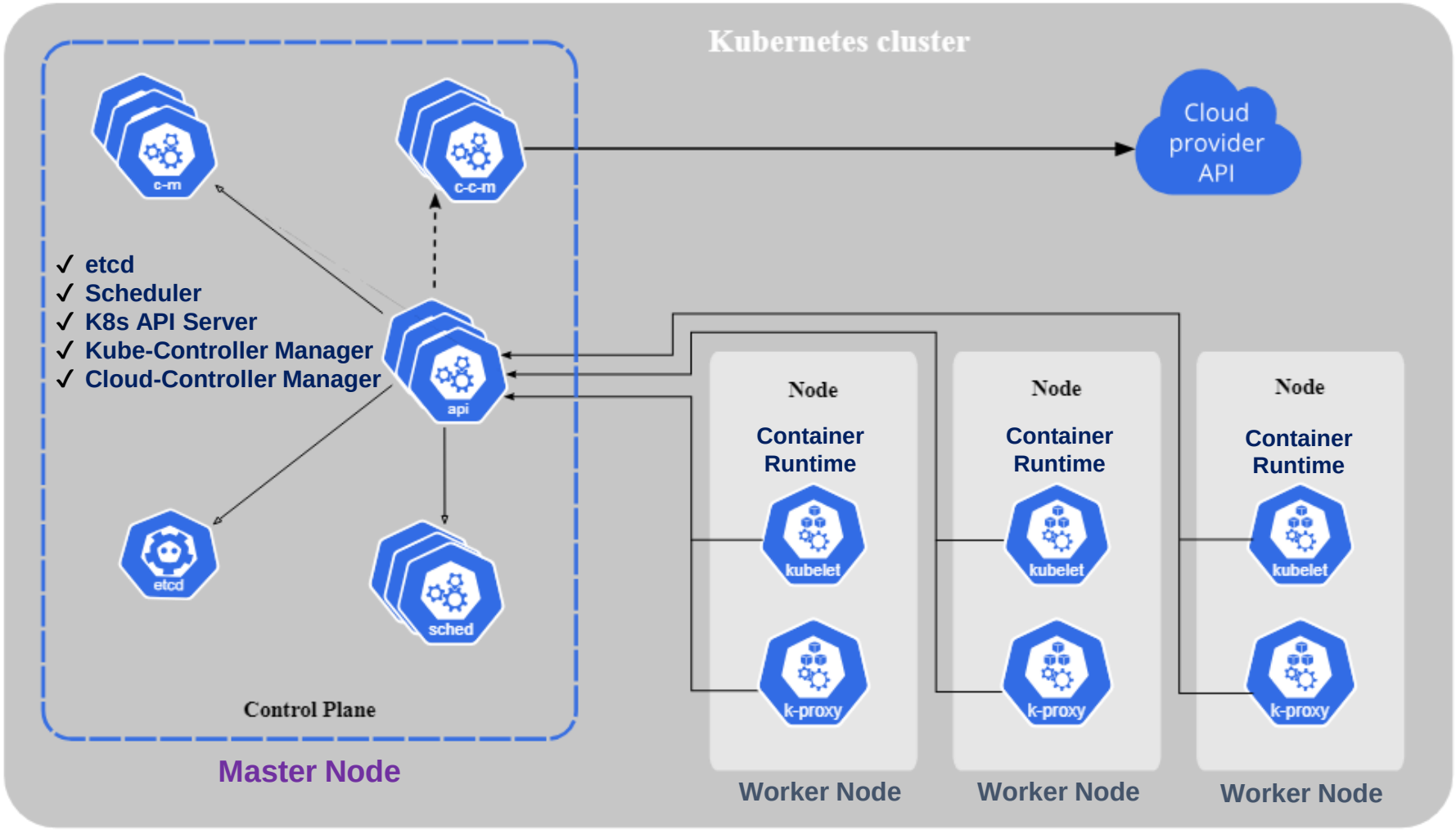


Worker Node Sub-processes



BCIT™

Kubernetes Architecture



API server

Cloud controller manager *(optional)*

Controller manager

etcd (persistence store)

kubelet

kube-proxy

Scheduler

Control plane

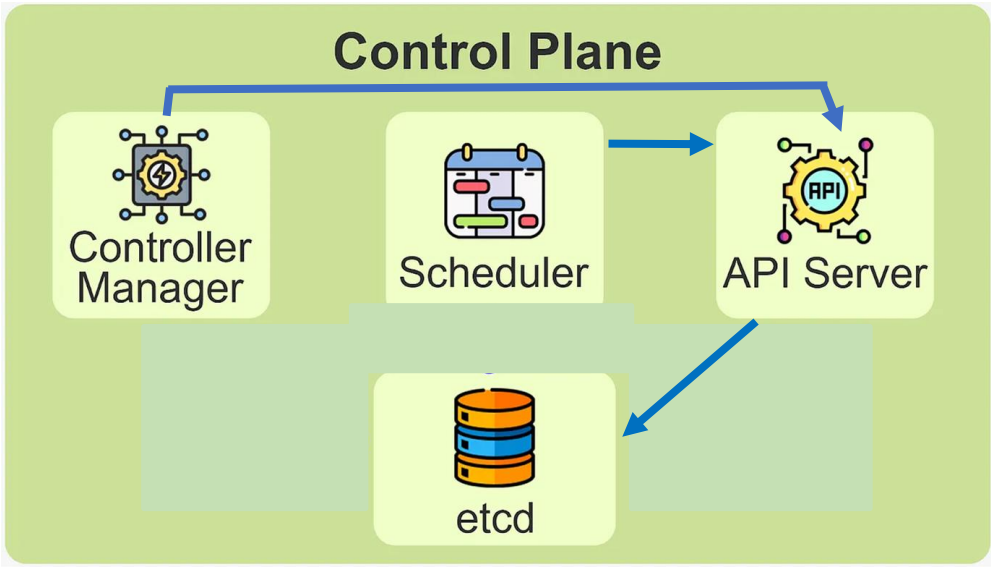
Node

BCIT™

Kubernetes Architecture ➡ etcd

Imagine **etcd** as a **Notebook** where K8s writes down everything it needs to remember about the cluster.

- Distributed key-value store
- Cluster configuration and state is stored in etcd
- Because it's distributed, more than one etcd can run
- Distributed consensus based on a quorum model
- Only API server talks directly to etcd, benefits of this:
  - request validation
  - Optimistic concurrency control
  - Keep consistency of the cluster state

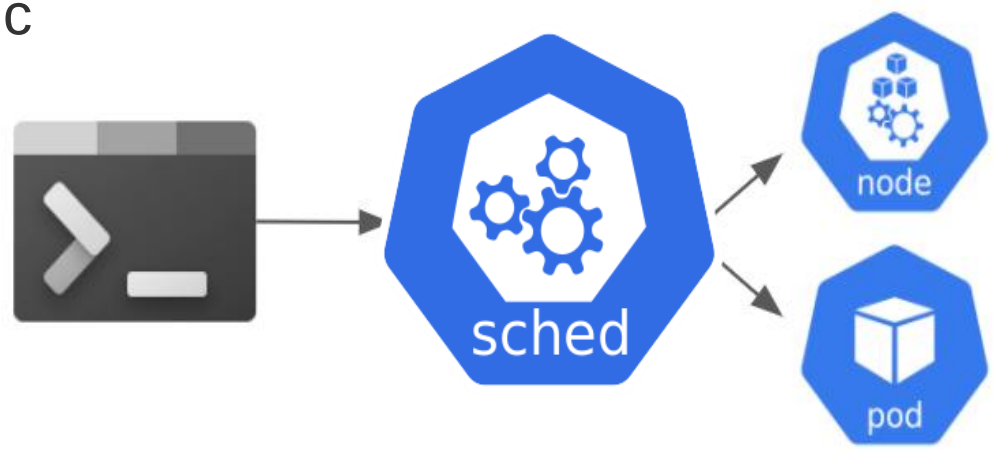


BCIT™

Kubernetes Architecture ➡ Scheduler

Think of the Scheduler as the **seating manager** in a restaurant. If we deploy 5 pods, the Scheduler finds nodes with enough CPU and memory to run them.

- Makes pod scheduling decisions based on:
  - resource requirements
  - hardware, software and policy constraints
  - affinity or anti-affinity specifications
  - data locality
- Multiple Schedulers can run, for each pod a specific Scheduler can be chosen
- Monitors system state for newly created pods that do not have nodes assigned



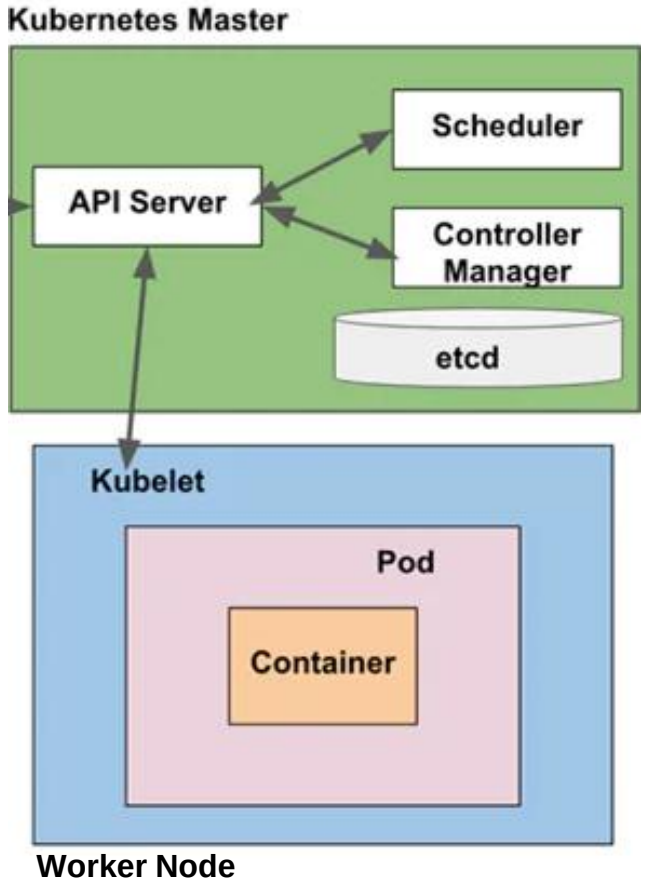
BCIT™

Kubernetes Architecture ➡ Kubelet: Worker Nodes

The Kubelet is the main worker on each node. If the API Server tells the Kubelet to run 2 containers for an application, the Kubelet makes sure those containers are created and kept alive.

Kubelet is a node daemon that is present on every node and manages:

- communication with an API server
- container runtime
- networking (via CNI)
- storage volumes
- image lifecycle
- metric and event collection, logging
- management of special devices (GPUs)
- resource constraint and QoS enforcement

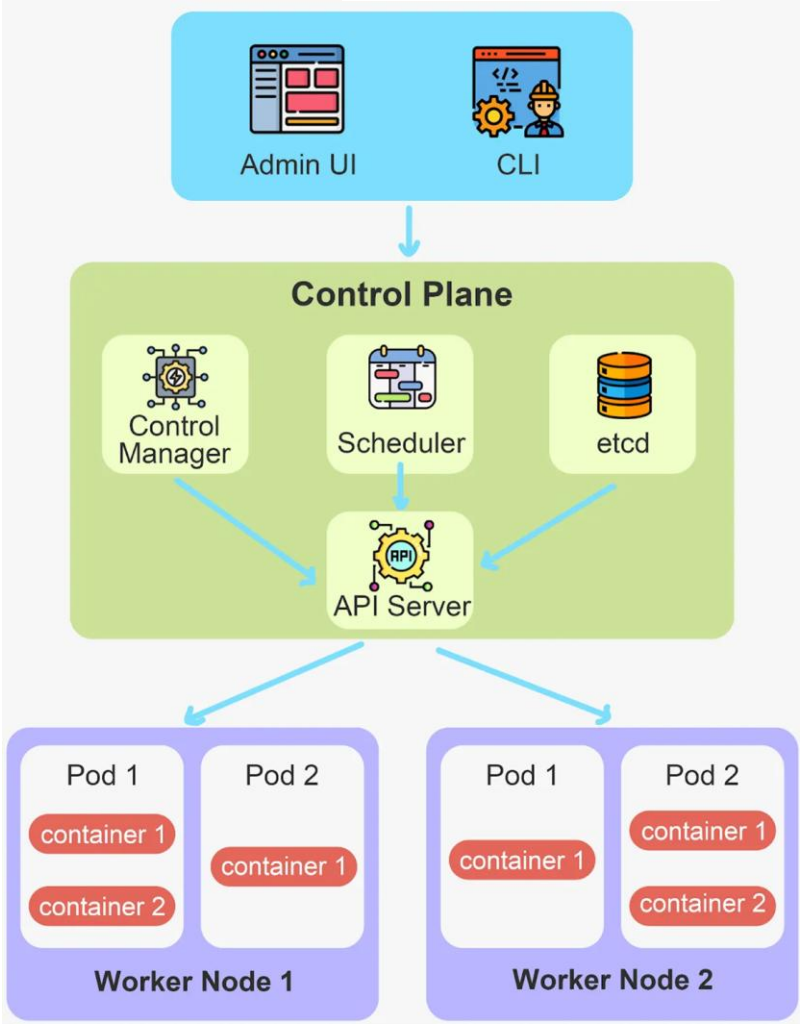


BCIT™

Kubernetes Architecture ➡ API Server

API Server is a **Traffic Controller** that accepts requests (instructions) & directs them to the right K8s component.

- Stores object data in a backend database (**etcd**).
- Multiple API servers can handle requests at the same time (provided that load-balancing or load-sharing is done via an external mechanism).
- Uses API Access Extensions in order to authenticate and authorize (via **RBAC**) REST API requests/users.
- Dynamic Admission Control is also supported.



When we use a command like **kubectl** apply to deploy an application, the **API Server** processes the request and ensures the cluster takes action.

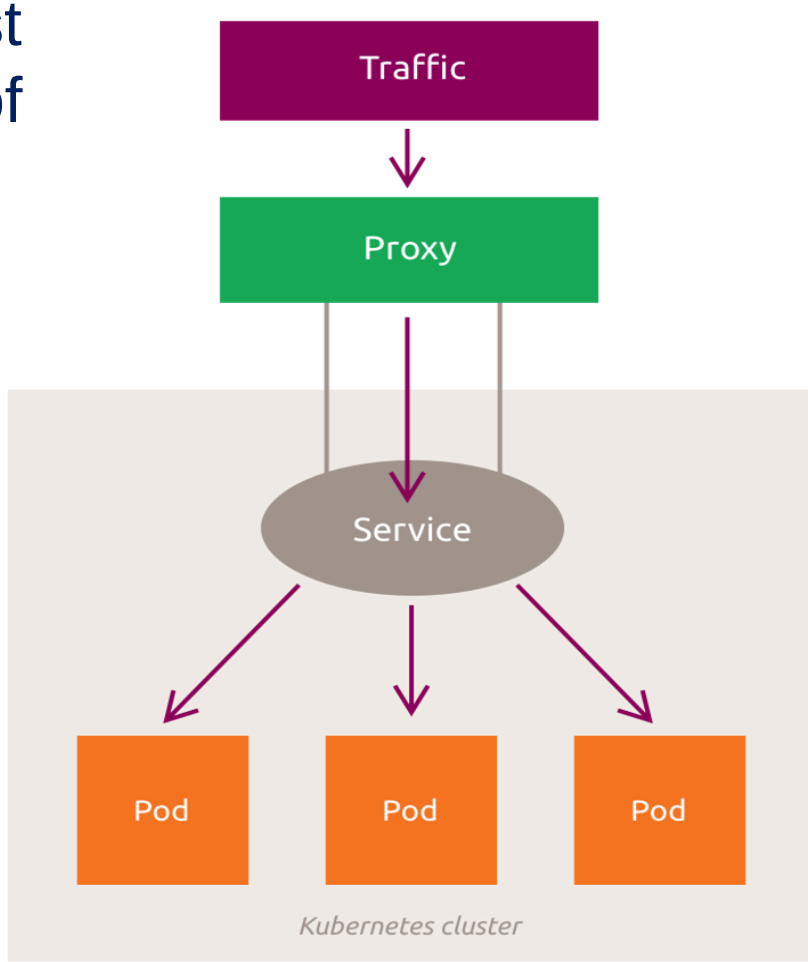
BCIT™

Kubernetes Architecture ➡ Kube-proxy: Worker Nodes

Kube-proxy handles networking on each node. It ensures that services can communicate with pods and traffic is properly routed across the cluster. If an external request for a web app is received, Kube-proxy routes it to one of the pods running that particular application.

Kube-proxy runs on each node and provides:

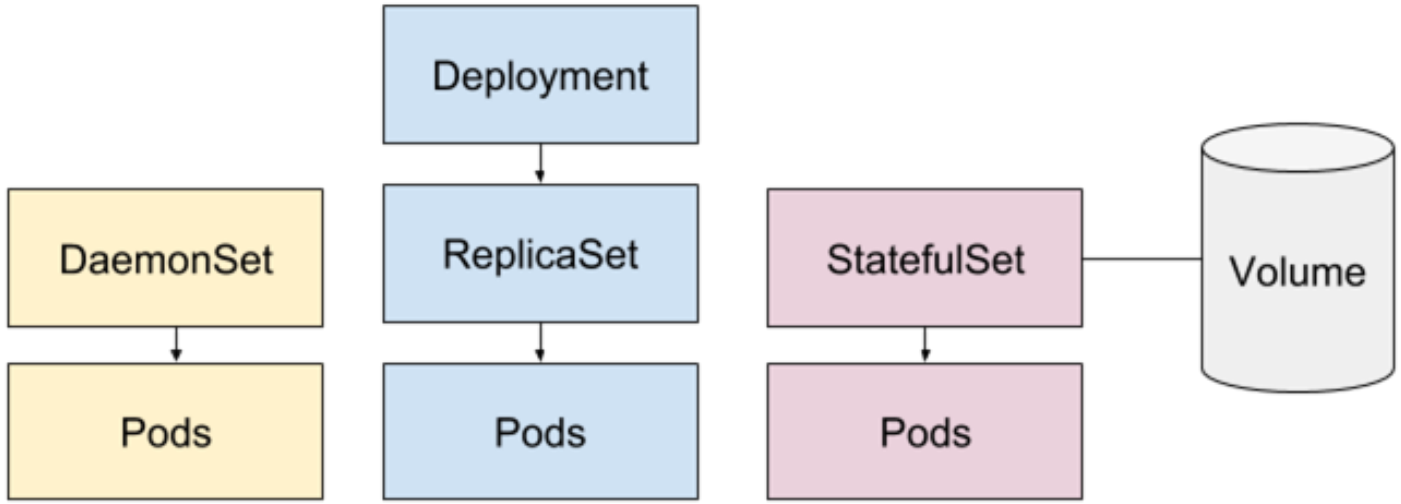
- load-balancing for cluster IPs (Virtual IPs)
- ensures clients can connect to the defined services through the API



BCIT™

Kubernetes Architecture ➡ Controller Manager

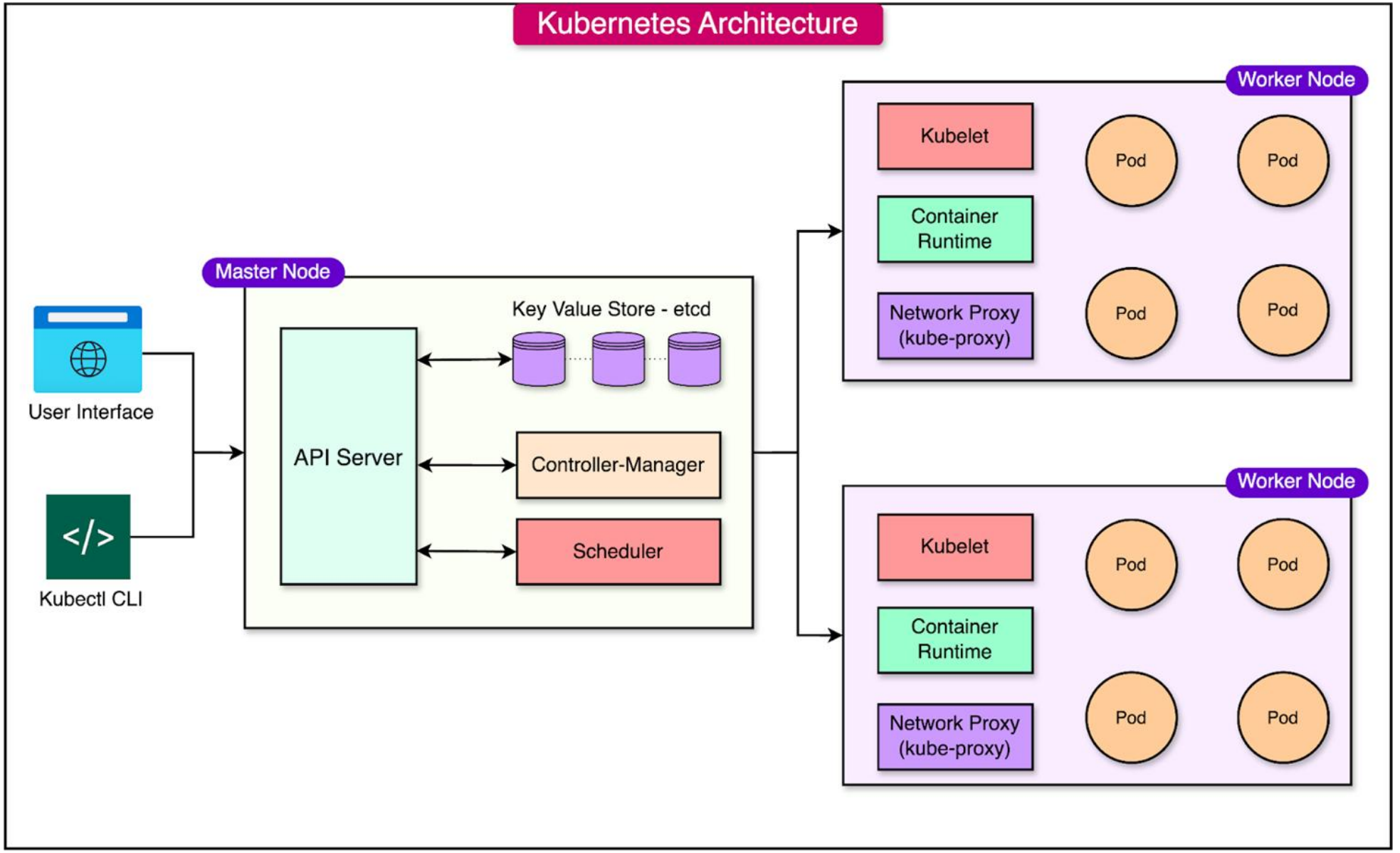
- Runs different controllers:
  ReplicaSet, DaemonSet, StatefulSet, Namespace, Persistent Volume.
- Manages worker threads and participates in leader elections with other controller managers in an HA scenario.
- If a node crashes and some pods go offline, the Controller Manager identifies the issue and spins up new pods on healthy nodes to match the desired state.



BCIT™

The Controller Manager ensures that the cluster's desired state matches the actual state.

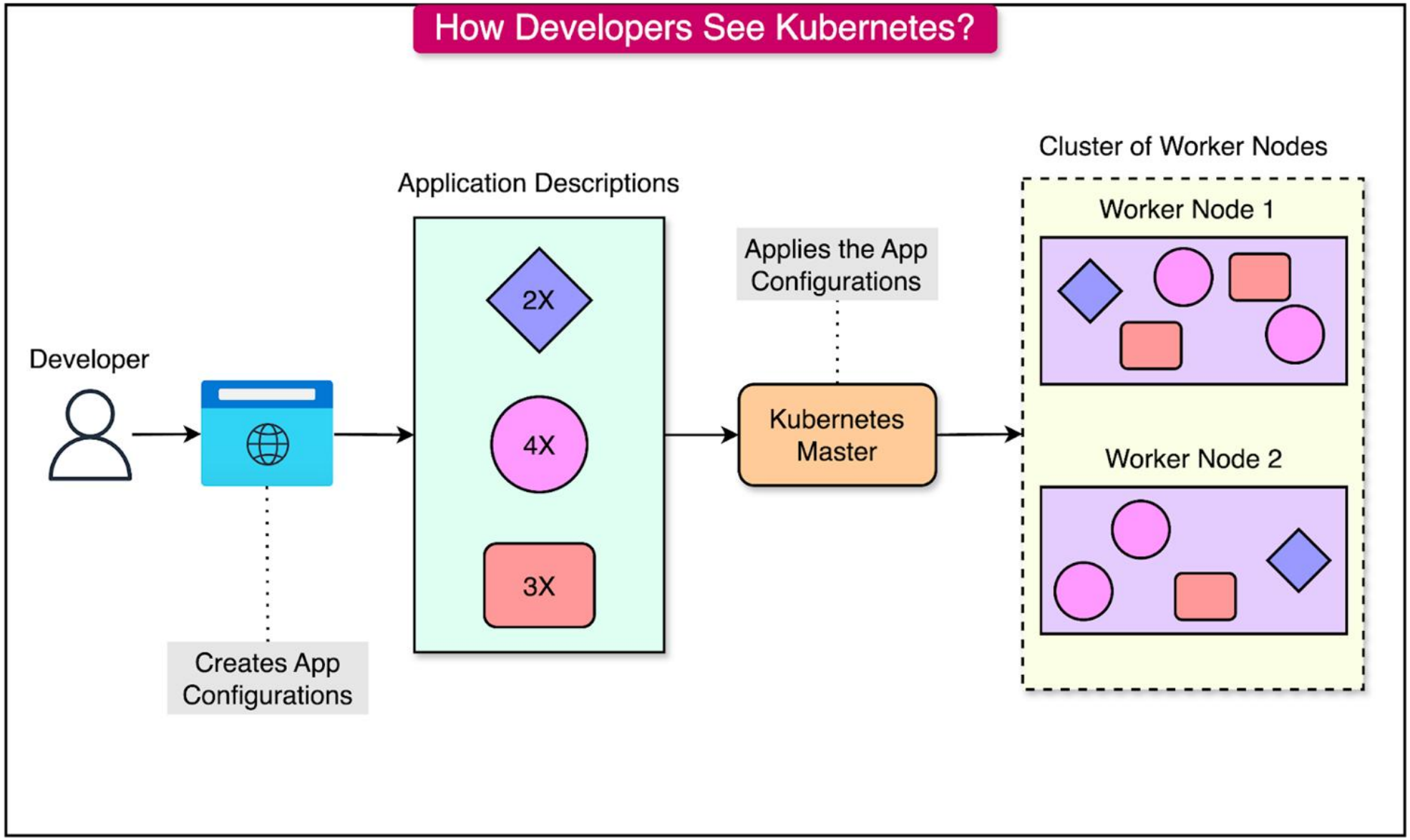
Kubernetes Architecture



https://blog.bytebytego.com/p/kubernetes-made-easy-a-beginners

BCIT™

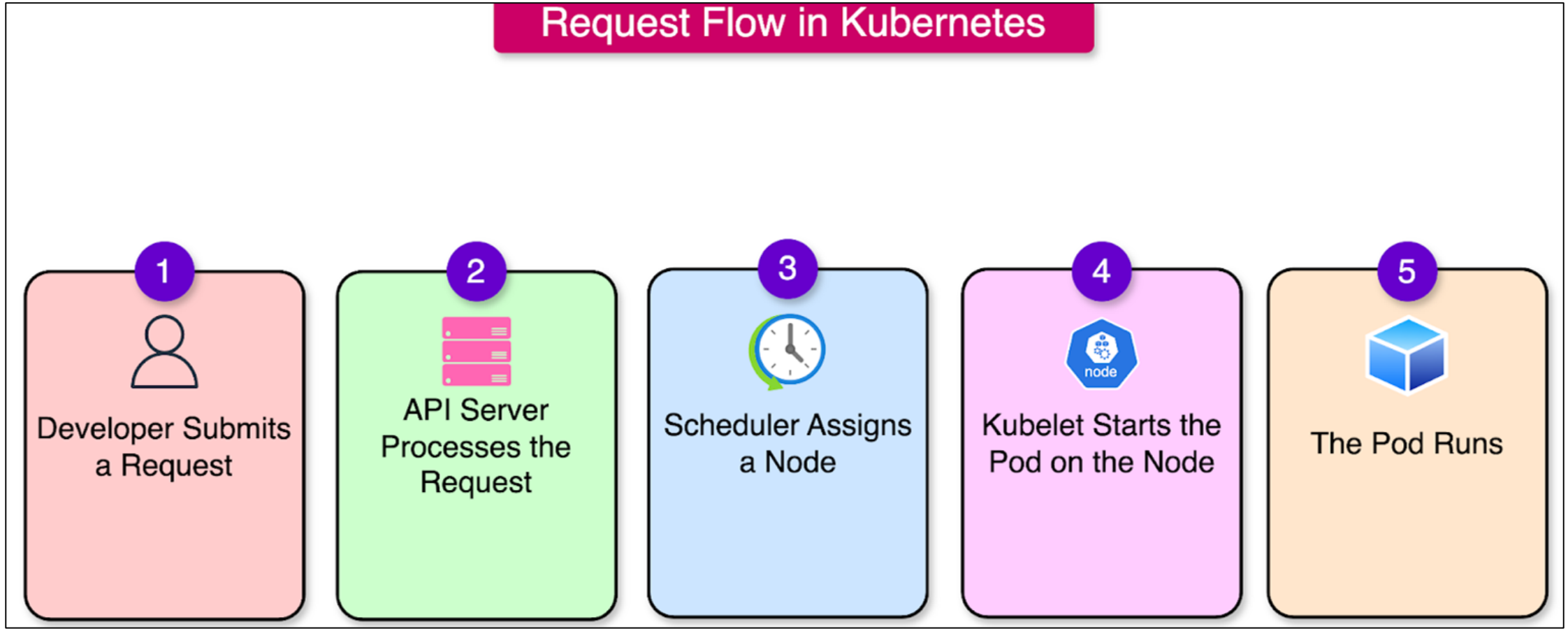
Kubernetes Architecture



https://blog.bytebytego.com/p/kubernetes-made-easy-a-beginners

BCIT™

Kubernetes Architecture

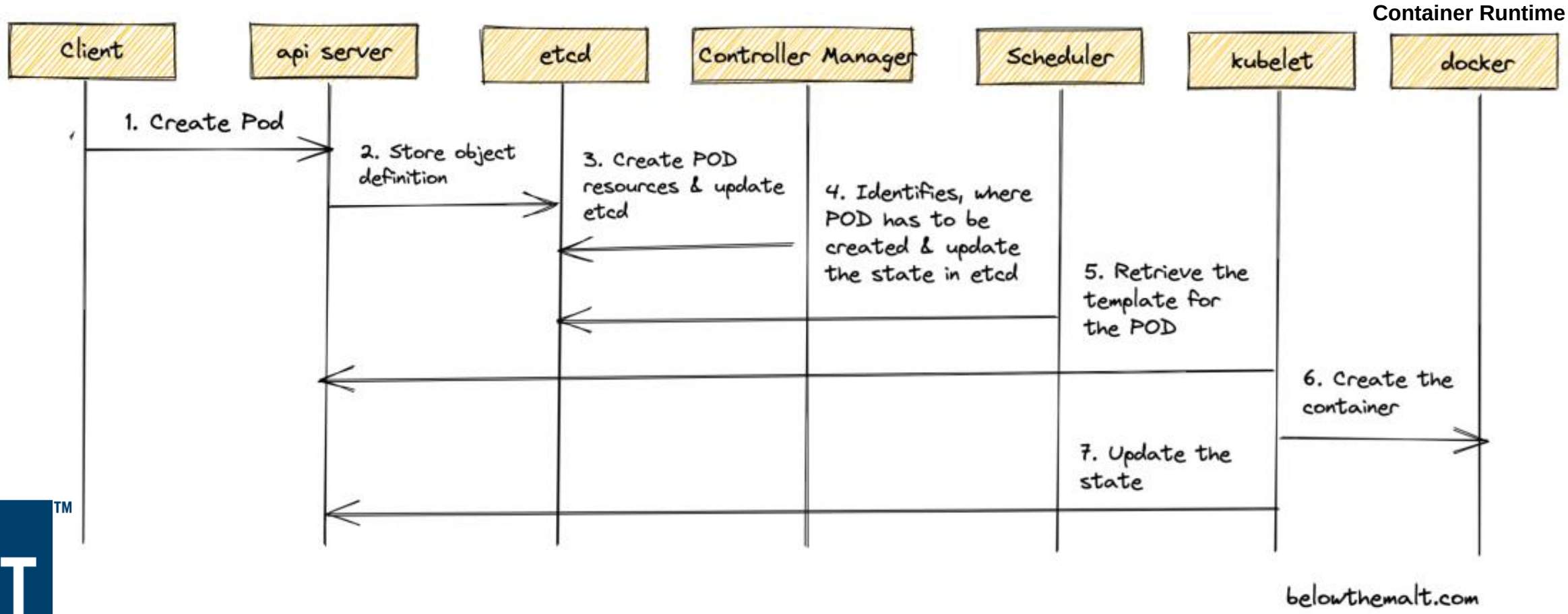


BCIT™

https://blog.bytebytego.com/p/kubernetes-made-easy-a-beginners

Kubernetes Architecture

POD Creation Process



BCIT™

Kubernetes in 100 seconds!

https://www.youtube.com/watch?v=PziYflu8cB8

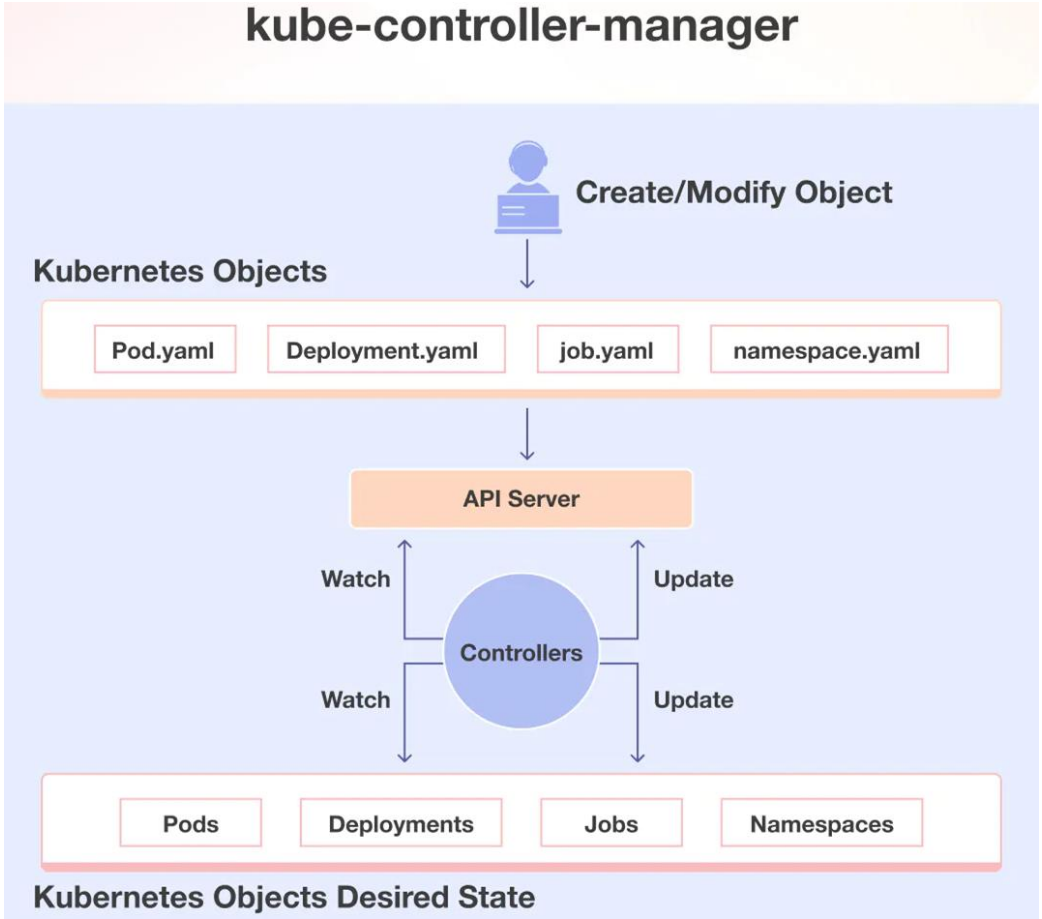
BCIT™

Kubernetes Data Model

Kubernetes data model is governed by versioned API object specifications.

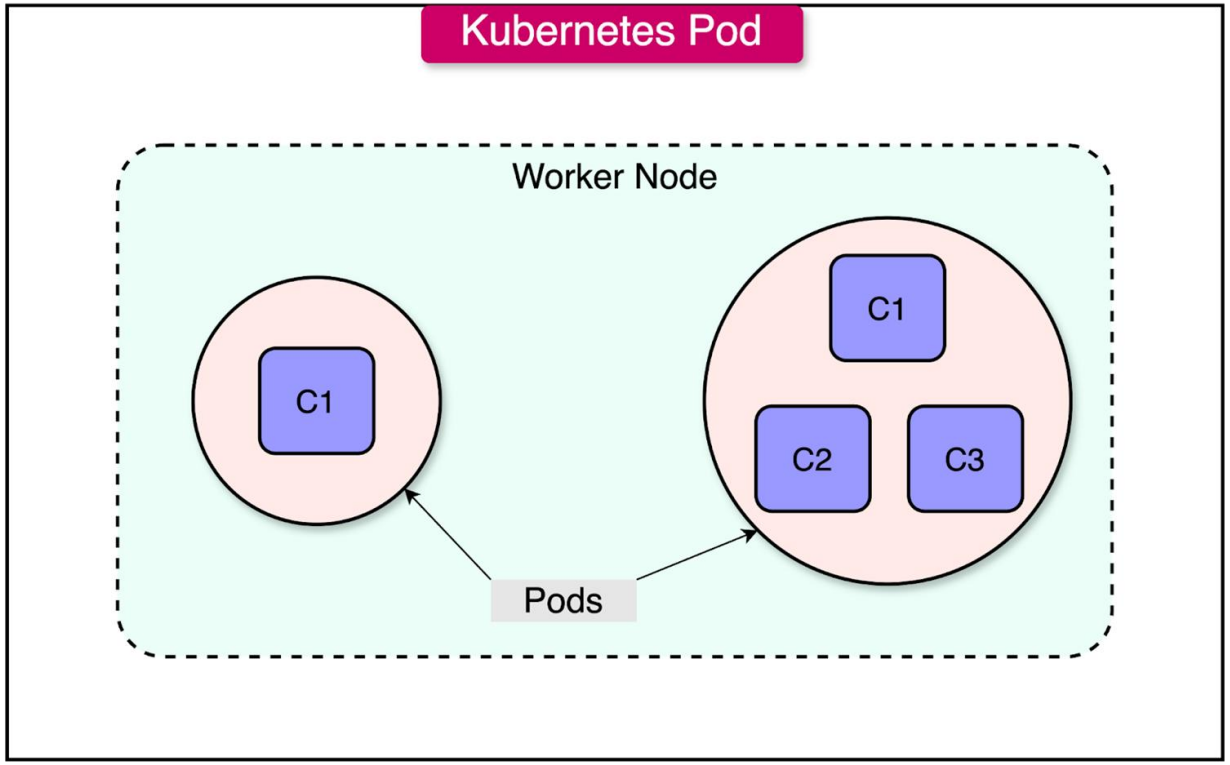
The Kubernetes Data Model organizes & manages resources in a cluster, ensuring the system's desired state is met. It's based on **declarative configuration**, where users define the desired state, and Kubernetes works to reconcile the actual state. We use **YAML** to manage resources in Kubernetes.

- **Resource Objects**: Represents Kubernetes entities like Pods, Services, Deployments, Namespace.
- **Desired State**: The configuration users define (e.g., number of replicas, container images).
- **Controllers**: Components that monitor the actual state and ensure it matches the desired state.



BCIT™

Kubernetes Data Model: Pod



1. A Pod can host a single container, which is the most common use case. Alternatively, a Pod can run multiple containers that are coupled in some way. For example, a primary app container and a **sidecar (helper) container** for logging.

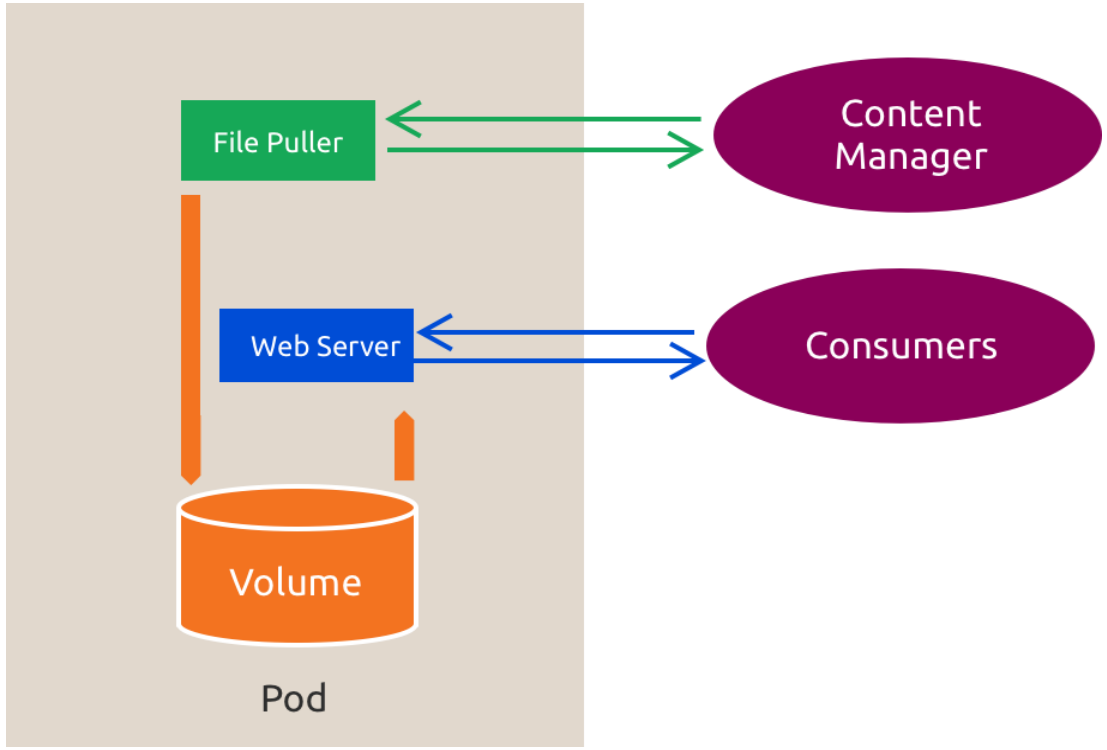
1. The containers in a Pod share the same IP address and port space. They can also share volumes for persistent storage.

1. Pods are designed to be short-lived (ephemeral). If a Pod fails, Kubernetes will restart it (or replace it) to match the desired state.

BCIT™

https://blog.bytebytego.com/p/kubernetes-made-easy-a-beginners

Kubernetes Data Model: Pod creation



nginx-pod.yaml

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

$ kubectl apply -f nginx-pod.yaml

BCIT™

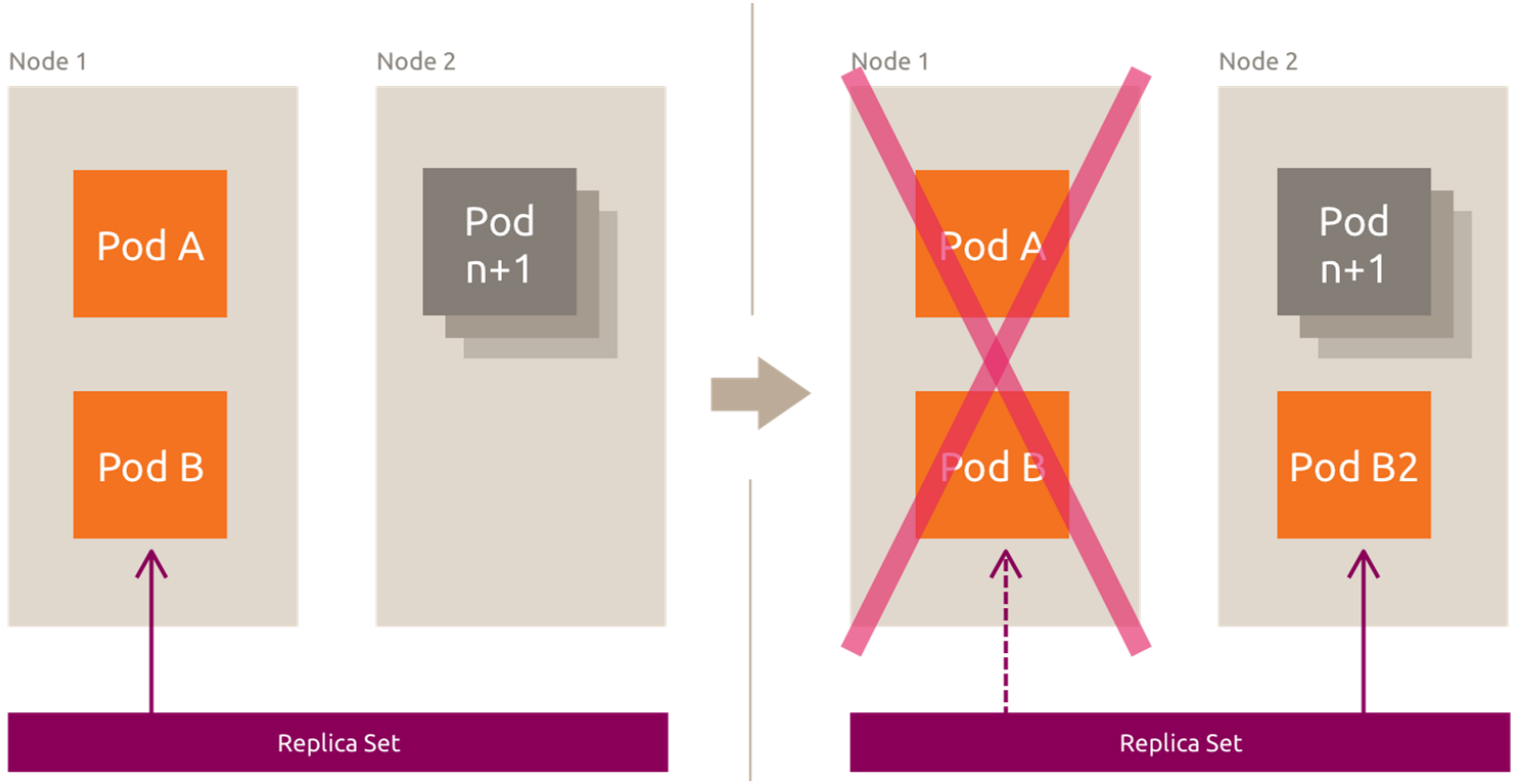
Kubernetes Data Model: ReplicaSet

ReplicaSet: A Self-healing group of similar pods

- Pods are ephemeral copies of each other.
- Make sure desired number of pods running.
- Automatically replaces any pod that failed or deleted.

guarantees
High Availability and **Fault Tolerance**
for the application.

ReplicaSet are typically used alongside
Deployments
to manage pod scaling and rolling updates.



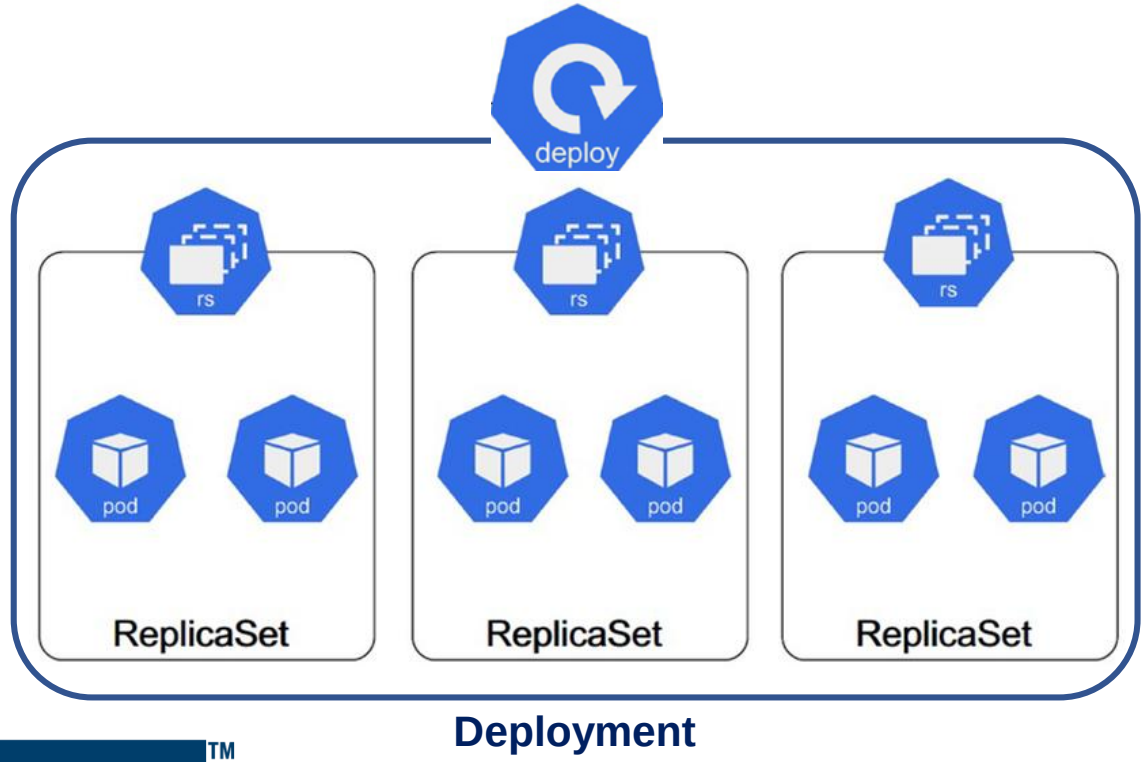
BCIT™

Pod A is not part of the ReplicaSet, but **Pod B** is and as soon as Node 1 is down, **Pod B** will be rescheduled to another node.

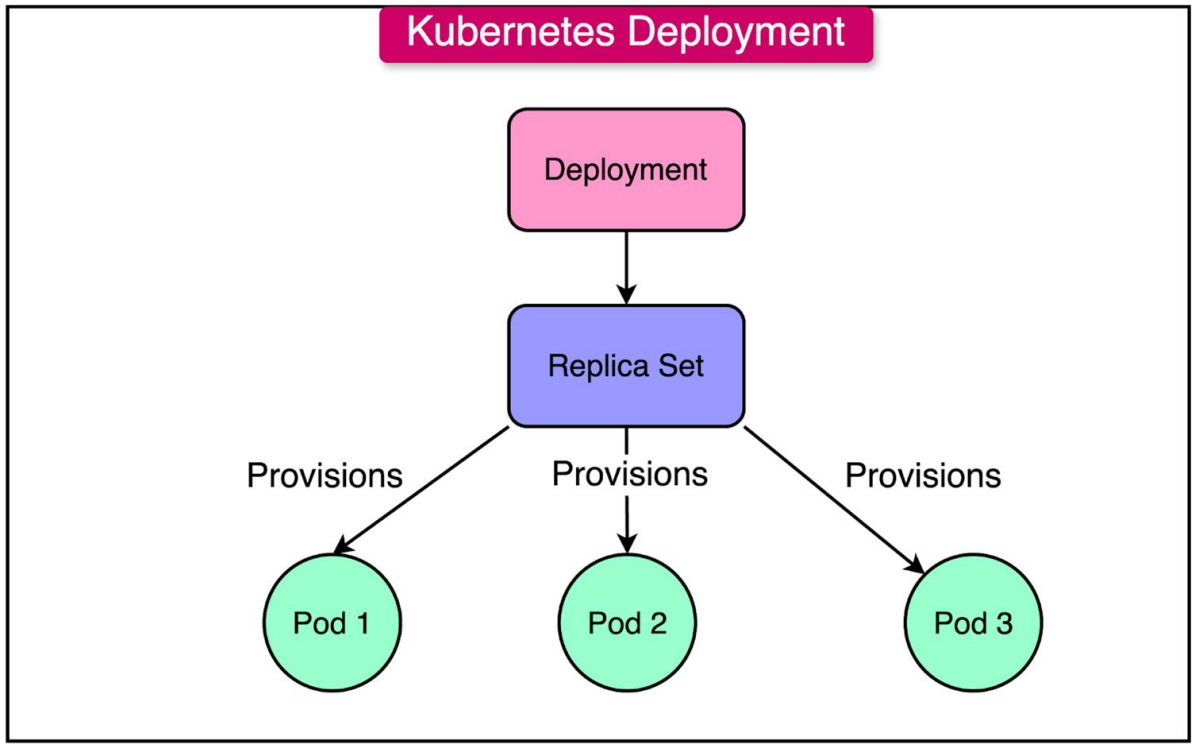
Kubernetes Data Model: Deployment

ReplicaSet ensures that a specified number of pod replicas are running at any given time.

Deployment is a higher-level abstraction that manages Pods and ensures they run reliably.



Deployment



- **Deployment** creates a **ReplicaSet** in the background
- **ReplicaSet** creates **Pods** in the background.

BCIT™

Kubernetes Data Model: Deployment

Deployment ensures a container for pods deployed, managed, and removed together.

- **Deployment** creates a **ReplicaSet** in the background
- **ReplicaSet** creates **Pods** in the background.

Key Advantages of using **Deployment**

1. **Declarative Updates**
2. **Rolling Updates**
3. **Automatic Rollback**
4. **Scaling**
5. **Self-healing**
6. **Version Control**
7. **Ease of Management**

BCIT™

To scale the Deployment from **3 to 5 replicas**, we can run:

$ kubectl scale deployment **nginx-deployment** --replicas=5

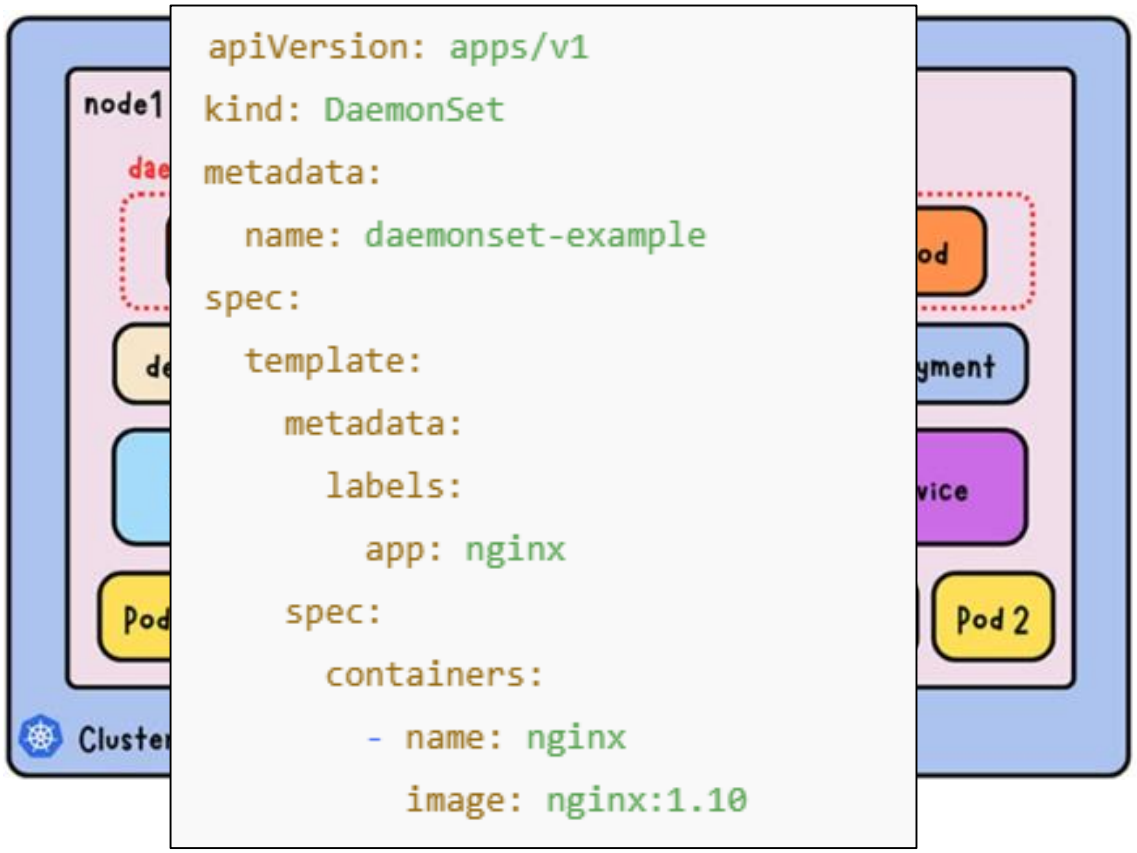
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx-container
          image: nginx:latest
          ports:
            - containerPort: 80
```

$ kubectl apply -f deployment.yaml

Kubernetes Data Model: DaemonSet

A **DaemonSet** ensures that all (or some) Nodes run a copy of a Pod. New nodes added to the cluster get pods added to them automatically. As nodes are removed from the cluster, those Pods are automatically deleted. Why we need it?

- Running a cluster storage daemon, such as ceph, on each node.
- Running a logs collection daemon on every node, such as logstash.
- Running a node monitoring daemon on every node, such as Zabbix.
- Running ingress controller worker pods (a load balancer to handle L7 traffic)



BCIT™

Kubernetes from different vendors

Minikube: A tool that enables running Kubernetes locally.

- ➢ Lightweight Kubernetes implementation for development and testing.
- ➢ Supports multiple drivers (Docker, Hyper-V, etc.).

- ✓ Easy to install and use.
- ✓ Runs Kubernetes in a single-node cluster.
- ✓ Suitable for local development and testing.
- ✓ Compatible with most Kubernetes tools and workflows.

https://microk8s.io/

MicroK8s

Amazon EKS

Azure Kubernetes Service (AKS)

https://kubernetes.io/releases/download/

Vanilla K8s

https://minikube.sigs.k8s.io/docs/

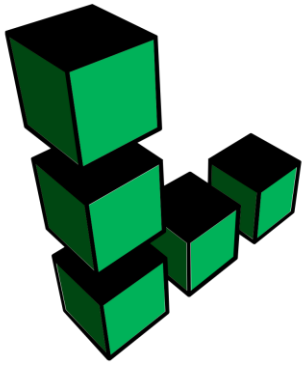
MiniKube

ORACLE

Google Kubernetes Engine

BCIT™

Building your K8s Cluster in Linode



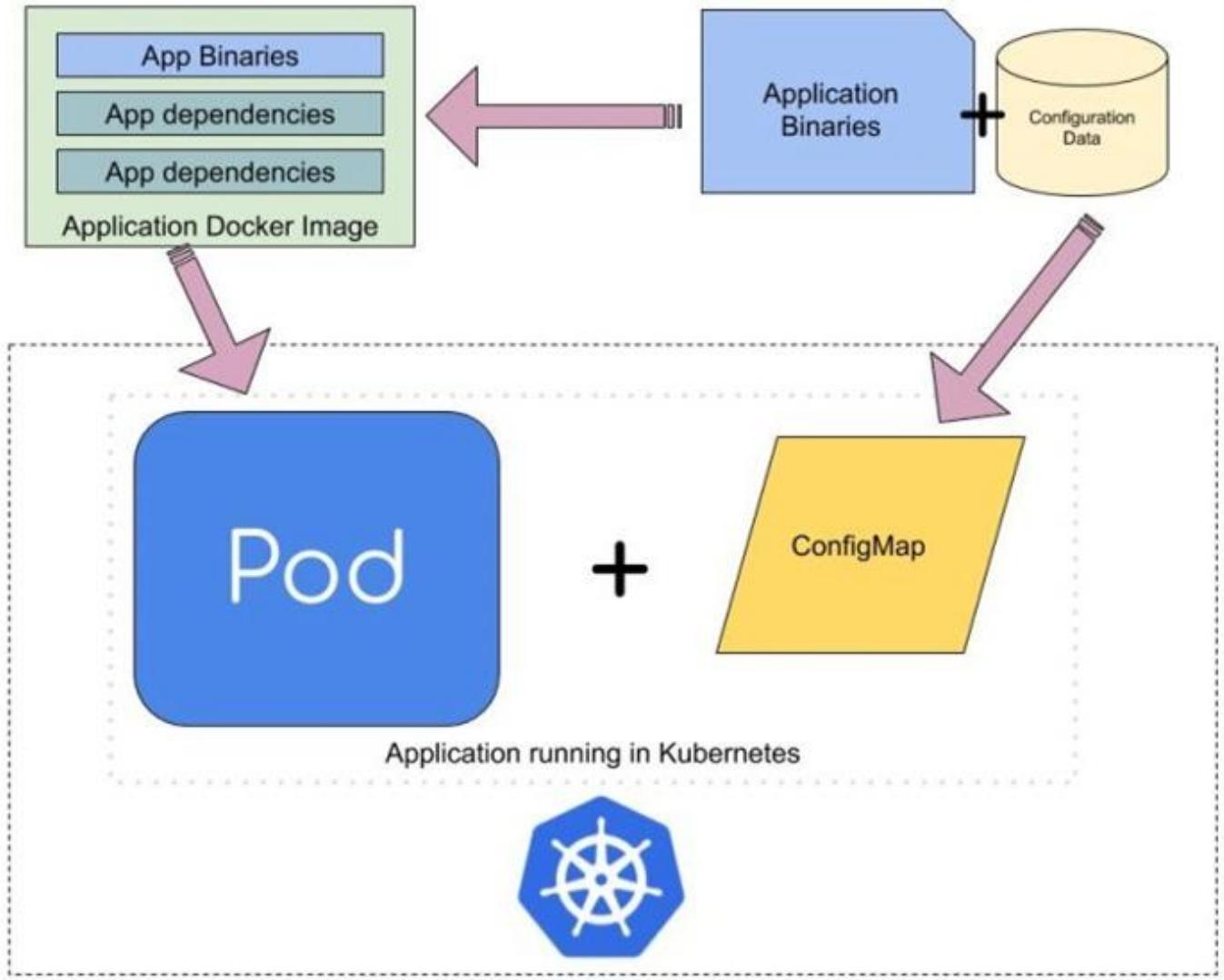
linode

BCIT™

Kubernetes Data Model: ConfigMaps

The flow highlights the separation of **application code** from **configuration** data, making the system more flexible and easier to manage in a K8s environment.

The application uses **ConfigMap** to access configuration data separately from the application binaries. This allows configuration updates without modifying the Docker image or redeploying the app.



BCIT™

Kubernetes Data Model: ConfigMaps

Decouple Configuration from Code:

ConfigMaps allow you to separate configuration from application code. This makes your applications more flexible and portable because you can update configuration values without changing the application code.

Environment-specific Settings:

ConfigMaps can store environment-specific configurations, making it easier to deploy the same application in multiple environments (e.g., development, staging, production) with different settings.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
data:
  nginx.conf: |
    server {
      listen 80;
      server_name localhost;
      location / {
        root /usr/share/nginx/html;
        index index.html;
      }
    }
```

ConfigMap is a K8s object that allows you to store non-confidential data in key-value pairs. In this example, a **ConfigMap** named nginx-config stores the contents of an nginx.conf configuration file.

BCIT™

Kubernetes Data Model: Secrets

A **Secret** is similar to a ConfigMap but is designed to store sensitive (confidential) data such as passwords, tokens, SSH keys, and API credentials. The data in Secrets is base64-encoded to add a basic level of obfuscation.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_NAME: "MyK8sApp"
  APP_ENV: "production"
```

Create a ConfigMap

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
type: Opaque
data:
  DB_PASSWORD: bXlzdXBlcnNlY3JldA==    # Base64-encoded value
```

Create a Secret

BCIT™

Kubernetes Data Model: ConfigMaps/Secrets example

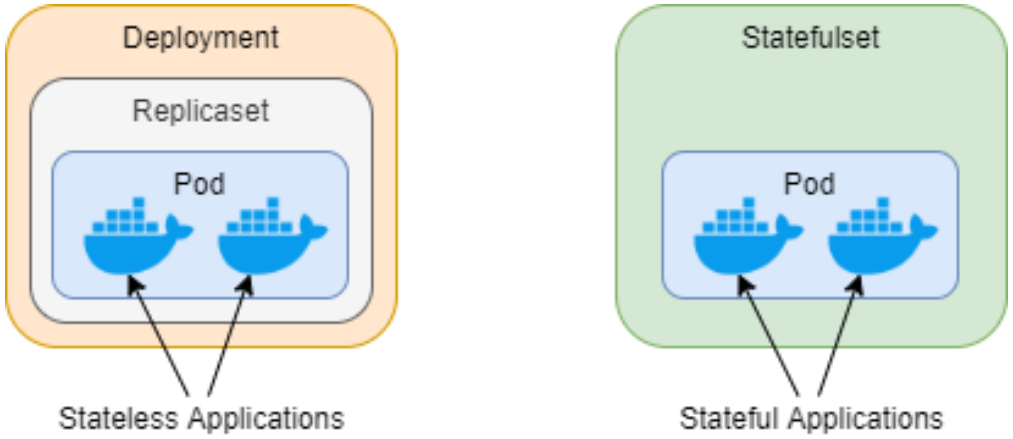
This YAML defines a Pod named app-pod running a single container named app-container based on the **nginx:latest** image. The container is configured to use environment variables sourced dynamically from a ConfigMap and a Secret. The APP_NAME and APP_ENV environment variables retrieve their values from the app-config ConfigMap, while the DB_PASSWORD environment variable retrieves its value from the app-secret Secret. This allows sensitive data like passwords and configuration values to be managed externally, enhancing security and flexibility.

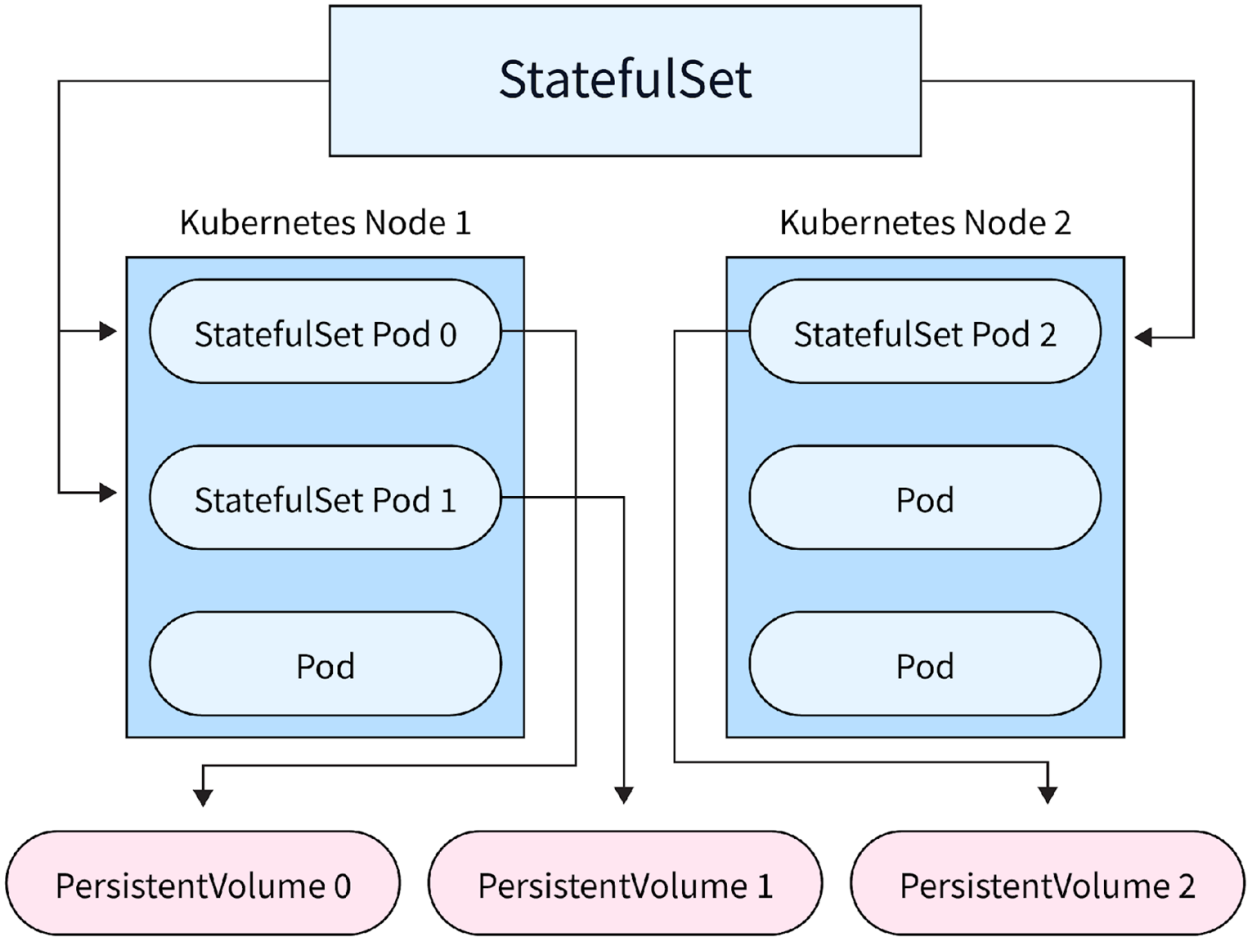
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: app-pod
spec:
  containers:
    - name: app-container
      image: nginx:latest
      env:
        - name: APP_NAME
          valueFrom:
            configMapKeyRef:
              name: app-config       # Name of the ConfigMap
              key: APP_NAME          # Key in the ConfigMap
        - name: APP_ENV
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: APP_ENV
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: app-secret       # Name of the Secret
              key: DB_PASSWORD       # Key in the Secret
```

By referencing to **ConfigMaps** and **Secrets**, the configuration decouples the application settings and credentials from the application code, enabling easy updates and **secure management**.

BCIT™

Kubernetes Data Model: StatefulSets





Deployment: Best suited for stateless applications where individual pods do not need to maintain any state between restarts (e.g., Web servers, DNS servers, Micro-services).

StatefulSet: Ideal for **Stateful Applications** such as databases (e.g., MySQL, PostgreSQL), where each instance (Pod) needs to have a stable identity and persistent data storage across restarts.

BCIT™

Pods are **ephemeral** in nature, when it is deleted it will lose all the state and associated data if it is stored locally. To keep the data, we need to create Persistent Volume Claim.

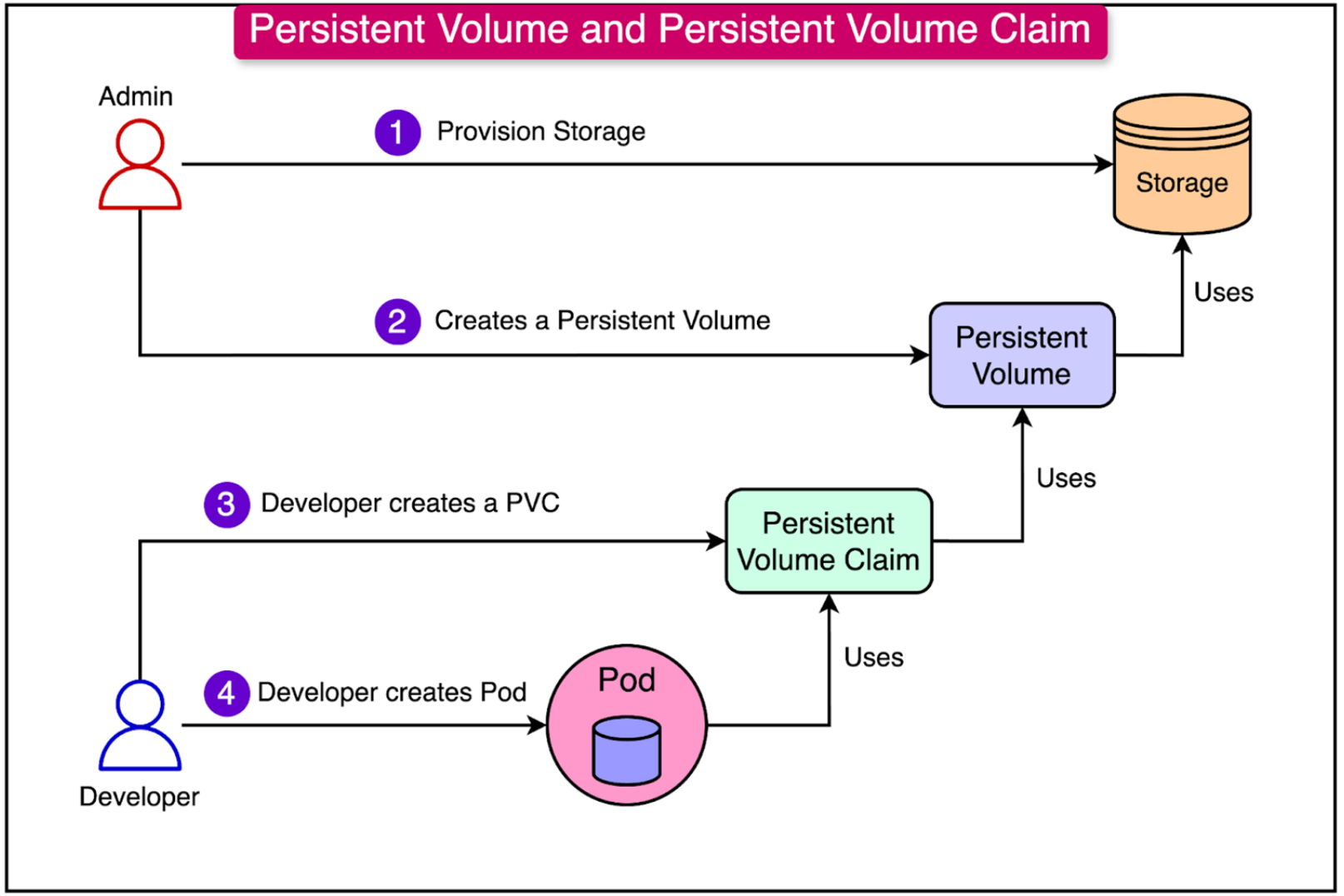
Kubernetes Data Model: StatefulSets

YAML file to define a **PV**

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

YAML file to define a **PVC**

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

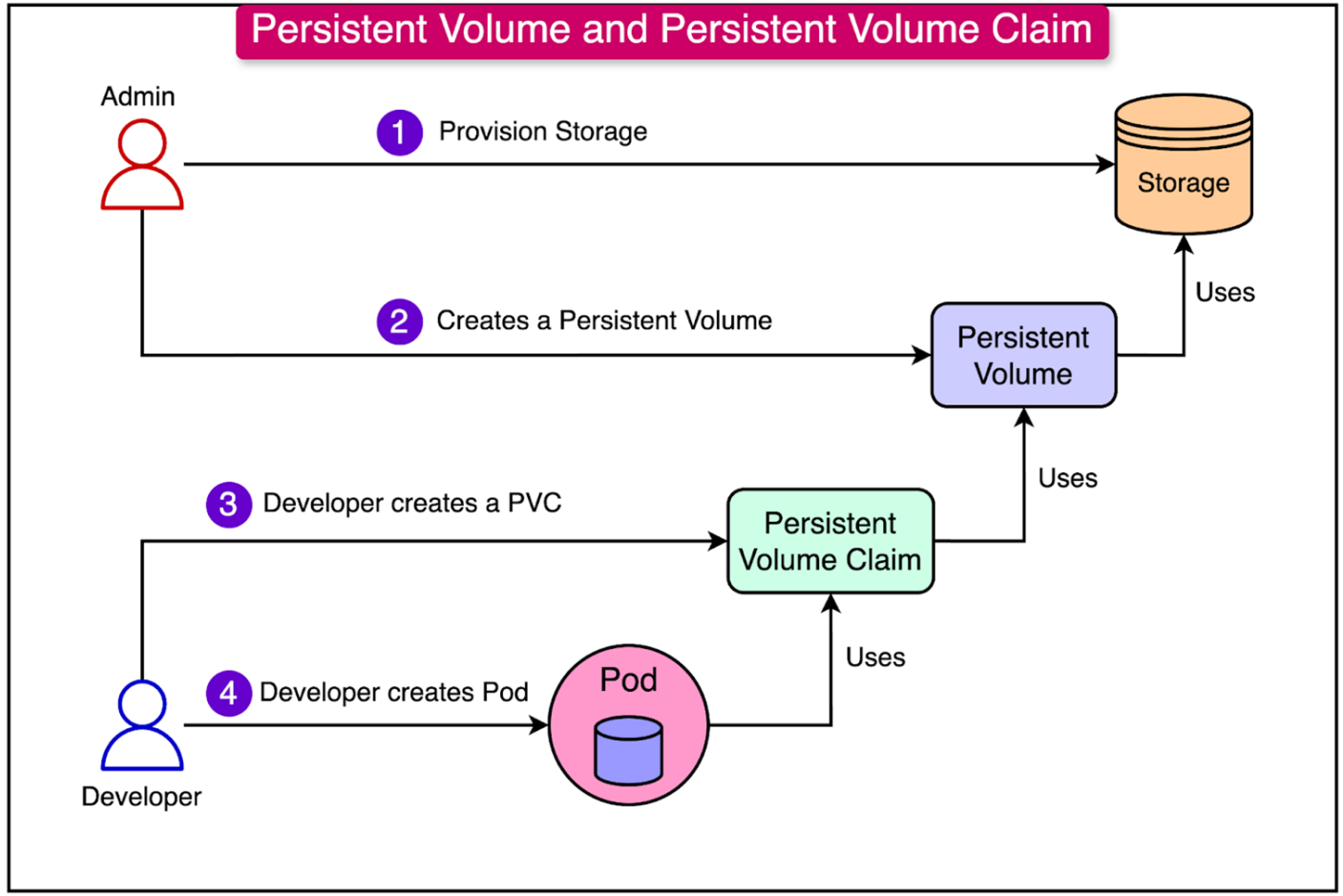


BCIT™

https://blog.bytebytego.com/p/kubernetes-made-easy-a-beginners

Kubernetes Data Model: StatefulSets

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mysql-pod
  labels:
    app: mysql
spec:
  containers:
    - name: mysql-container
      image: mysql:5.7
      ports:
        - containerPort: 3306
      env:
        - name: MYSQL_ROOT_PASSWORD
          value: "mypassword"
      volumeMounts:
        - name: mysql-storage
          mountPath: /var/lib/mysql
  volumes:
    - name: mysql-storage
      persistentVolumeClaim:
        claimName: mysql-pvc
```

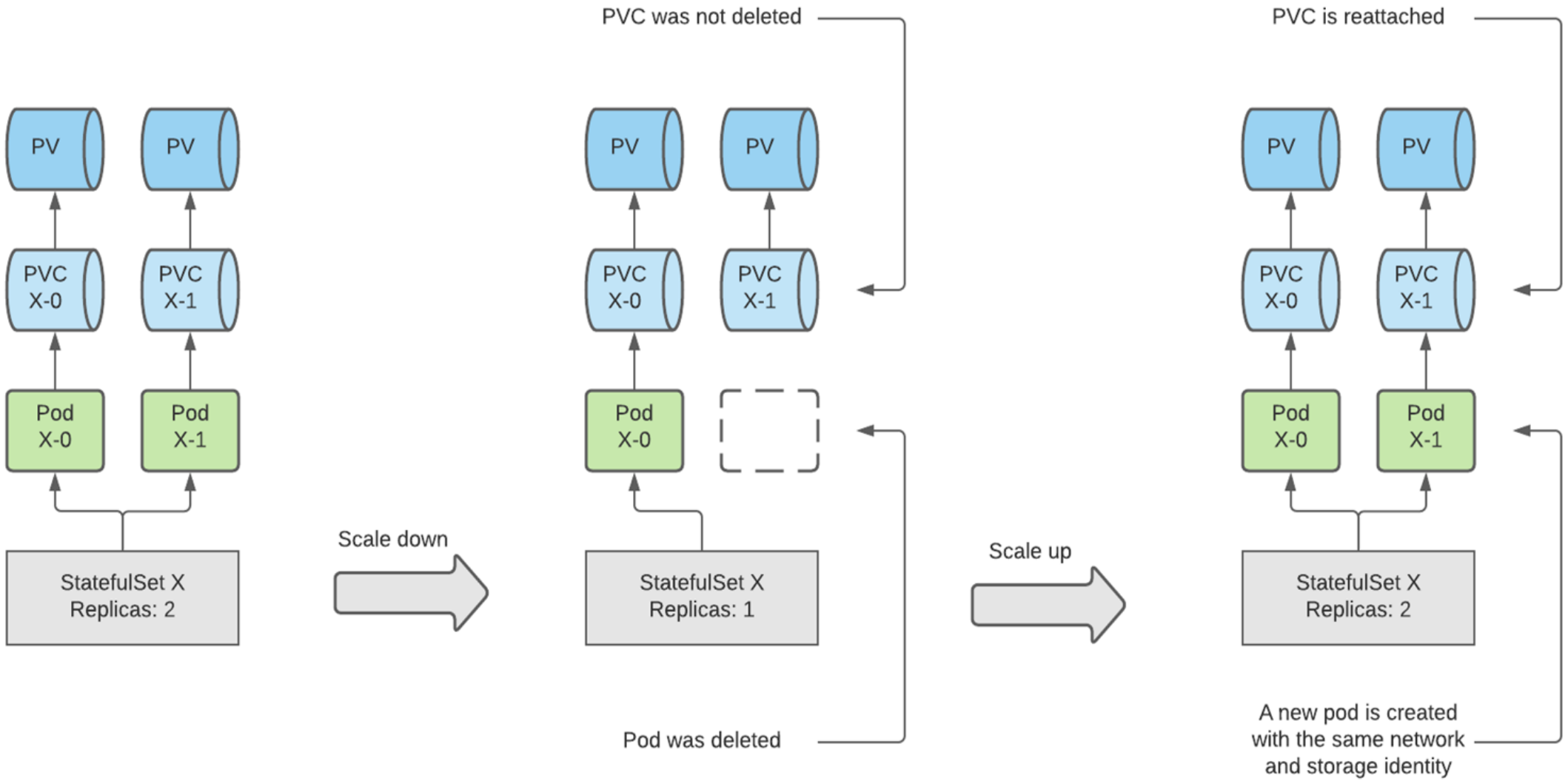


This configuration mounts the **PV** at **/var/lib/mysql**, which is the default directory for **MySQL** data. It uses **mysql-pvc** to attach the storage to the pod.

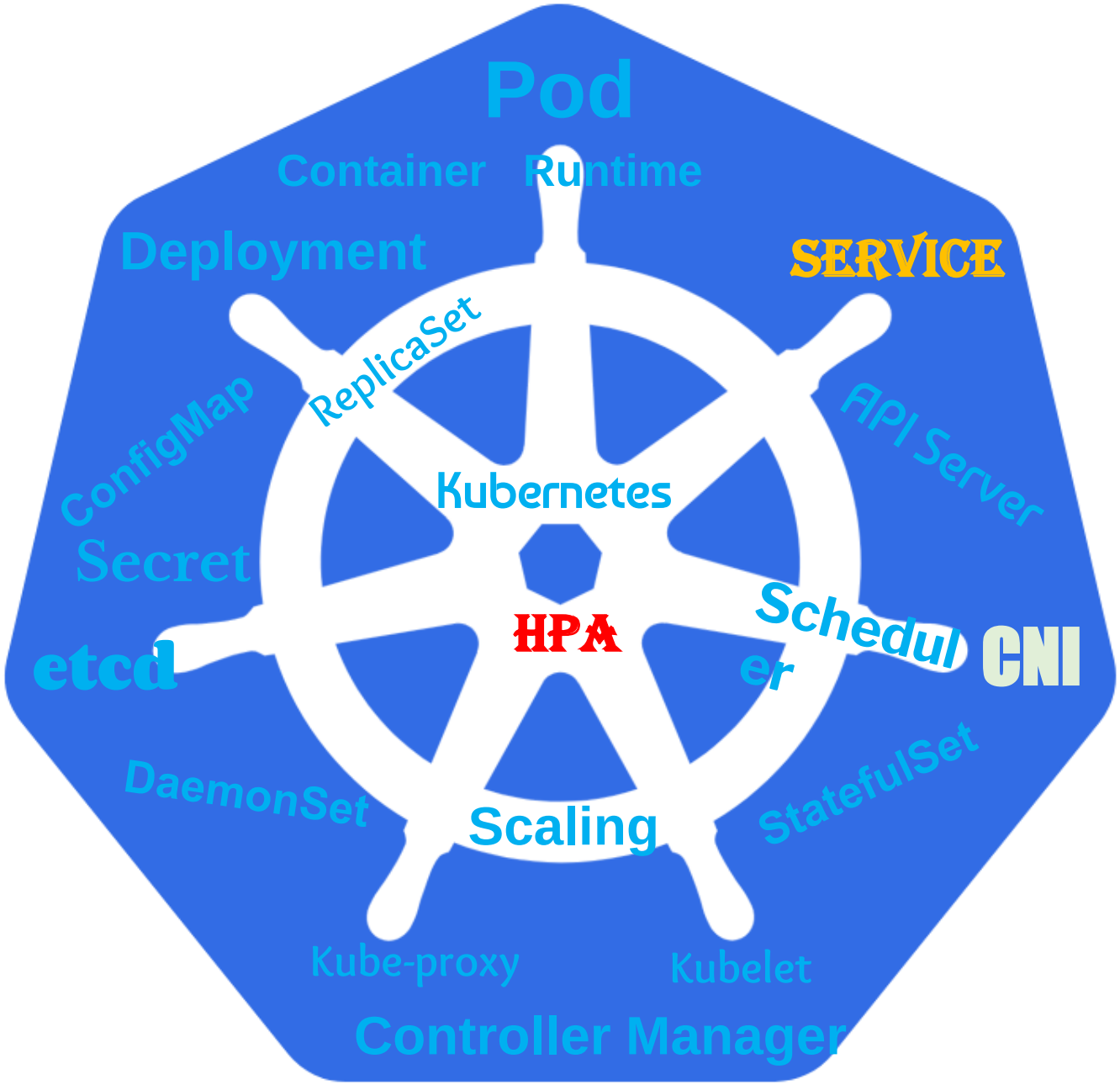
https://blog.bytebytego.com/p/kubernetes-made-easy-a-beginners

BCIT™

Kubernetes Data Model: StatefulSets



BCIT™



Networking

CNI

Service

HPA

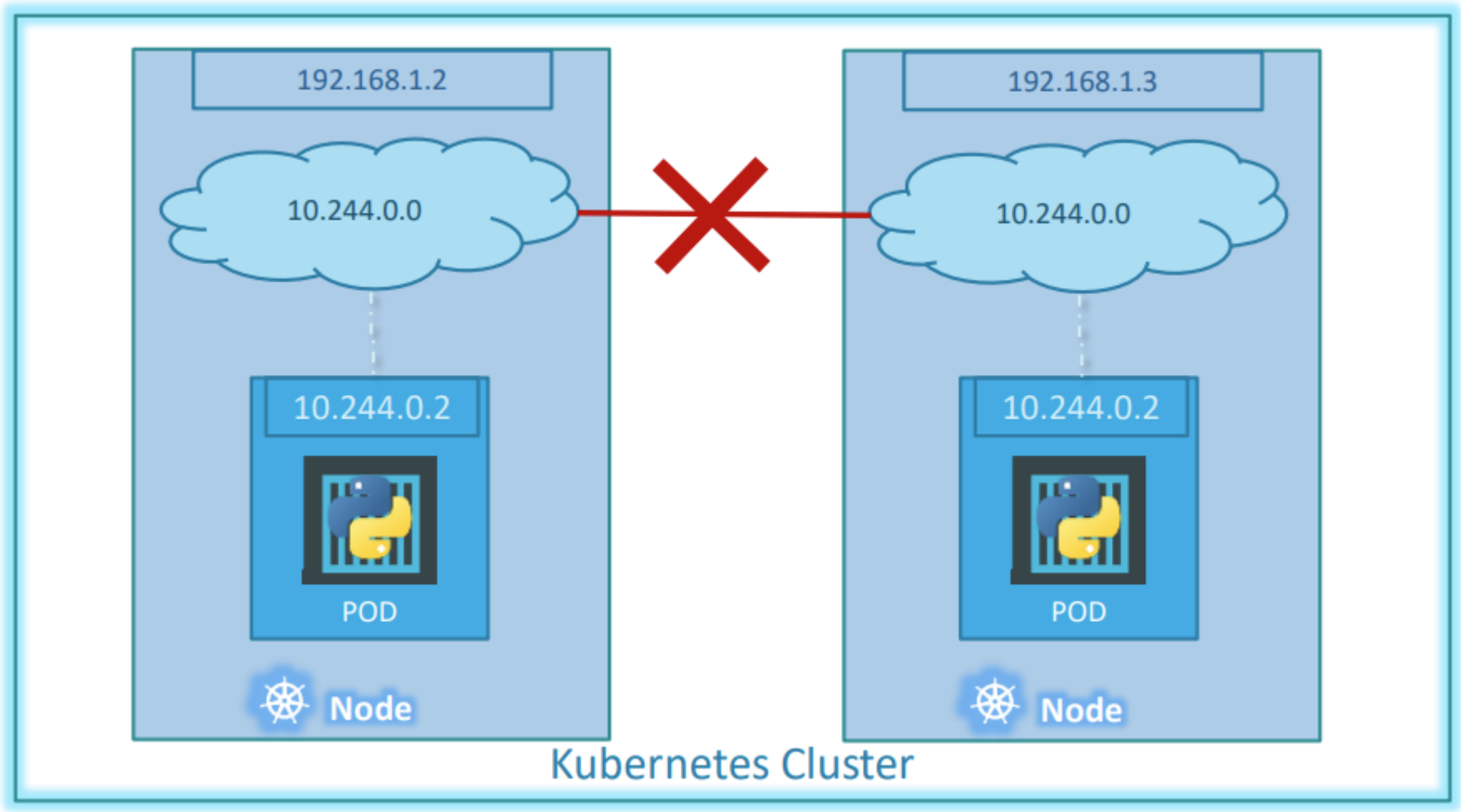
Scaling

BCIT™

K8s Cluster Networking

If Kubernetes Cluster Networking is configured properly, then

1. All containers/PODs can communicate to one another without NAT.
2. All nodes can communicate with all containers and vice-versa without NAT.



BCIT™

K8s Cluster Networking: CNI

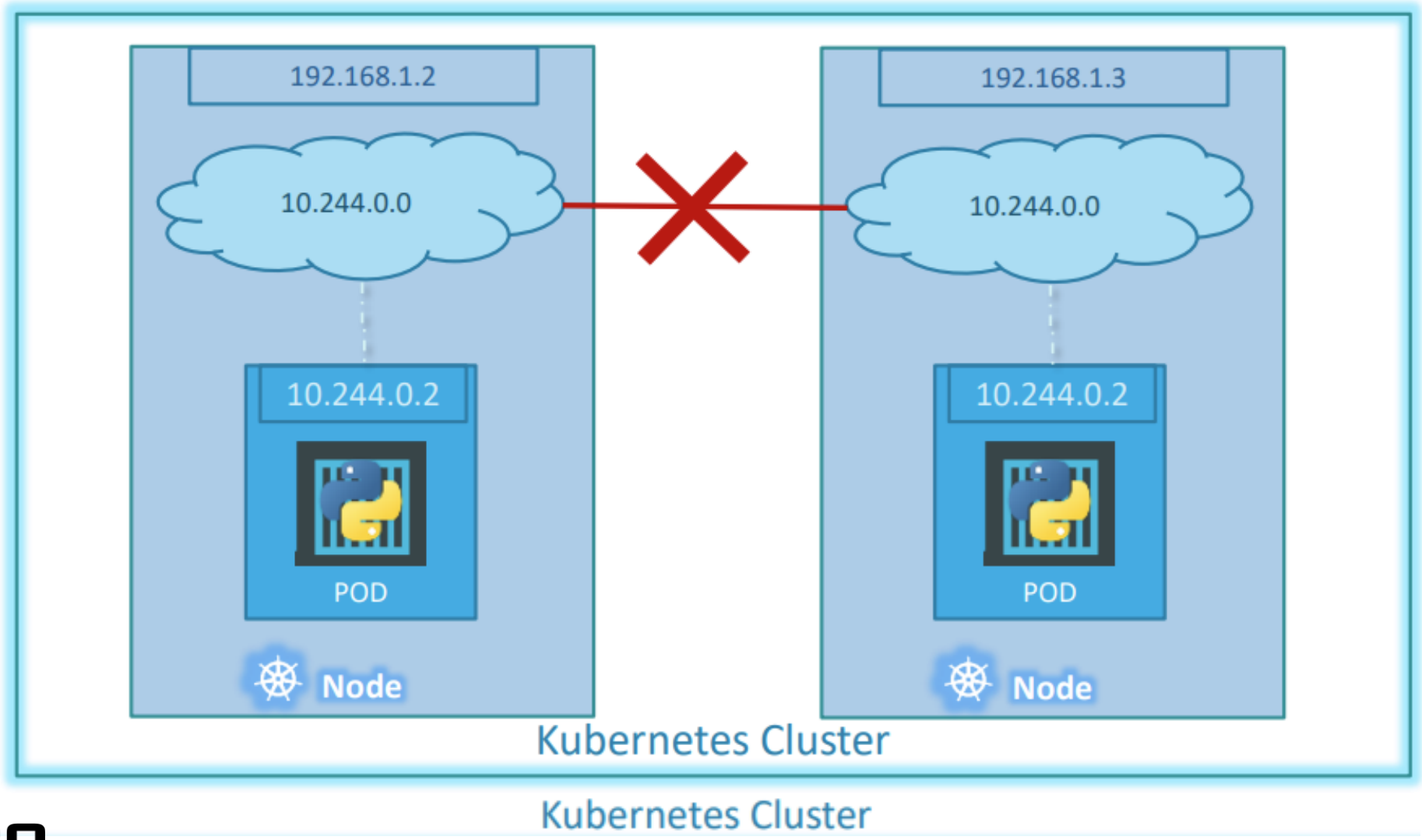
Container Network Interface (CNI) a standard network implementation interface that allows Kubelet to call different Network Plugins to implement network configuration methods.

CNI is not a specific networking solution, it is a framework that enables the use of various networking plugins. Indeed, it defines how to connect containers to networks and manage their IP addresses.

Networking Plugins such as Calico/Cilium/flannel use CNI to provide advanced features like networking, network policies & security for K8s clusters.

BCIT™

K8s Cluster Networking: CNI



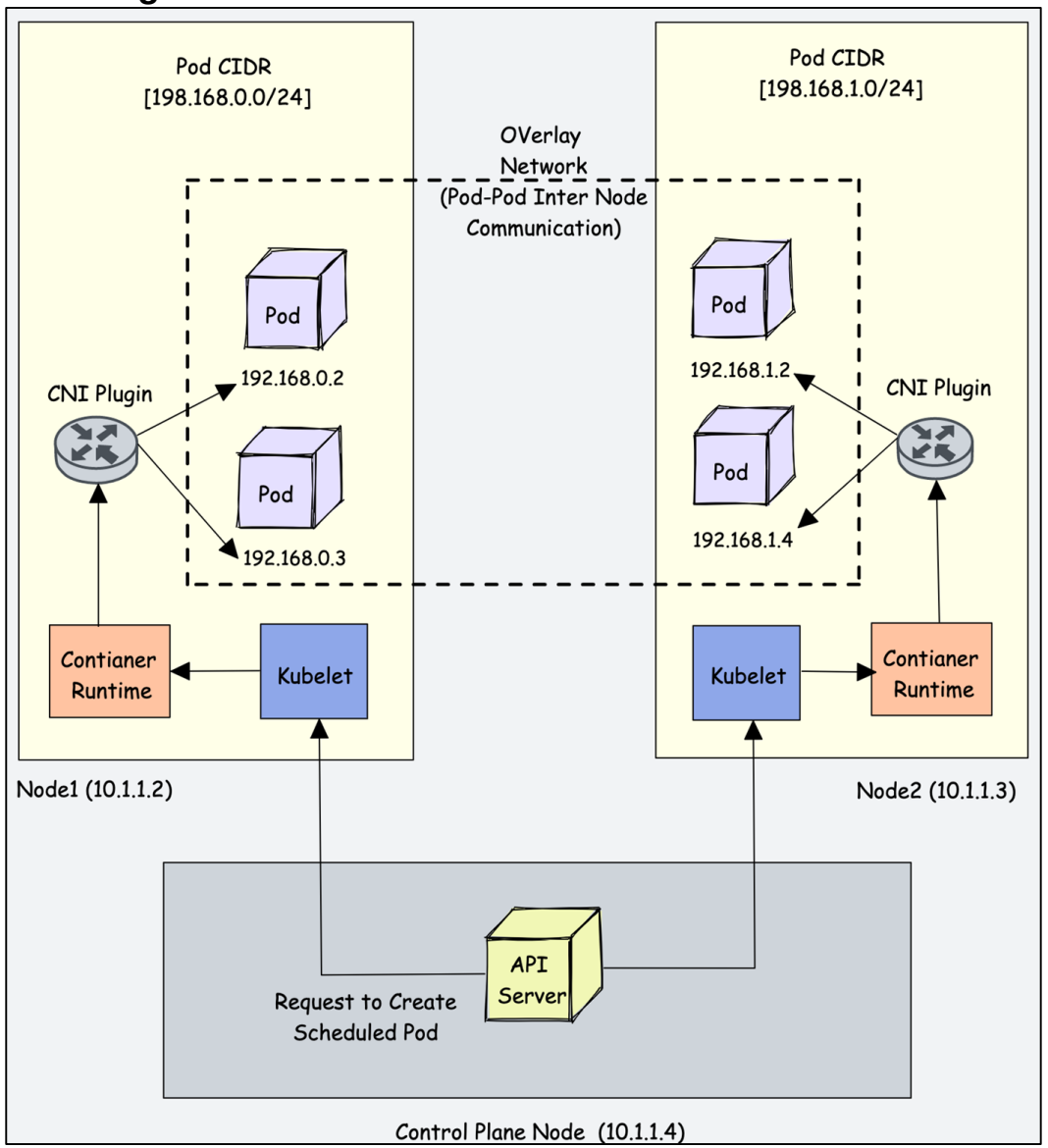
Kubernetes Cluster

BCIT™

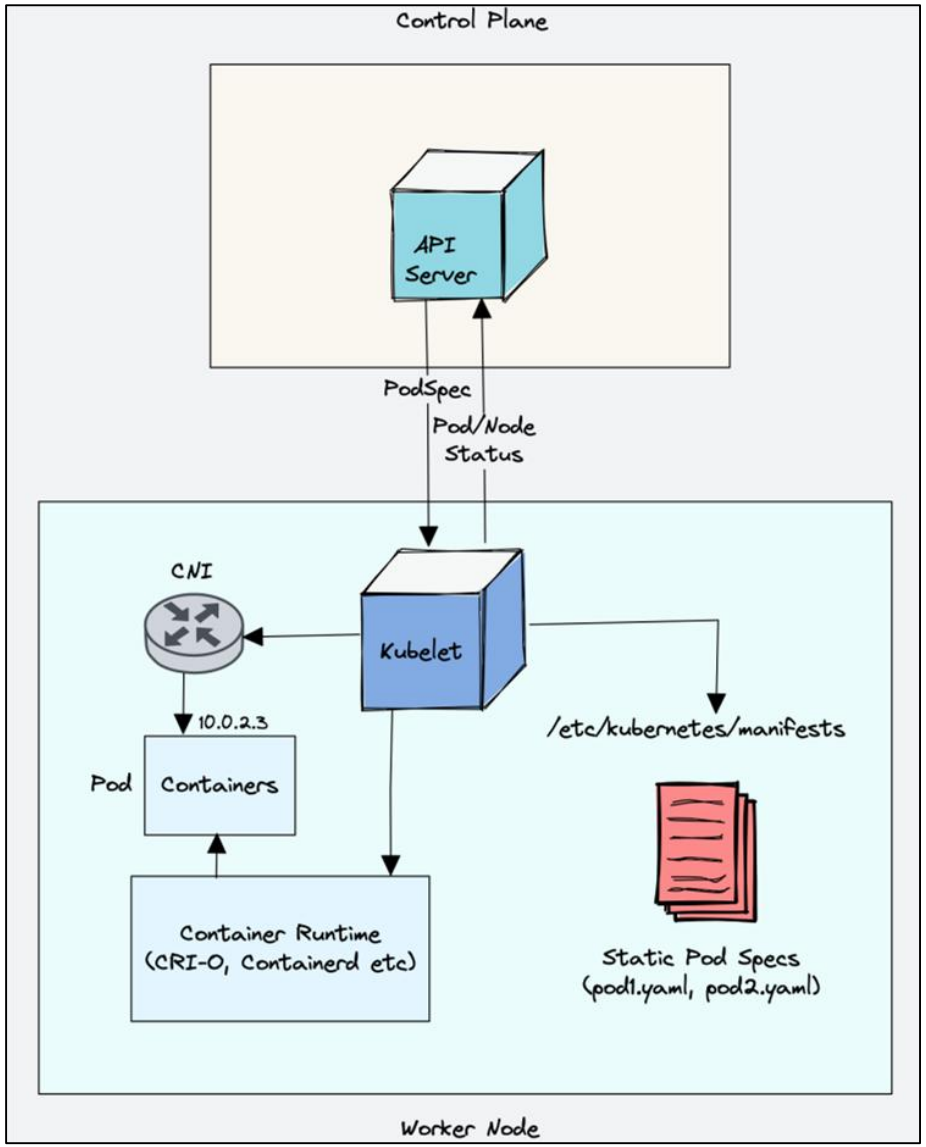
PROJECT CALICO

K8s Cluster Networking: CNI

CNI Plugin Architecture



Kubelet Architecture



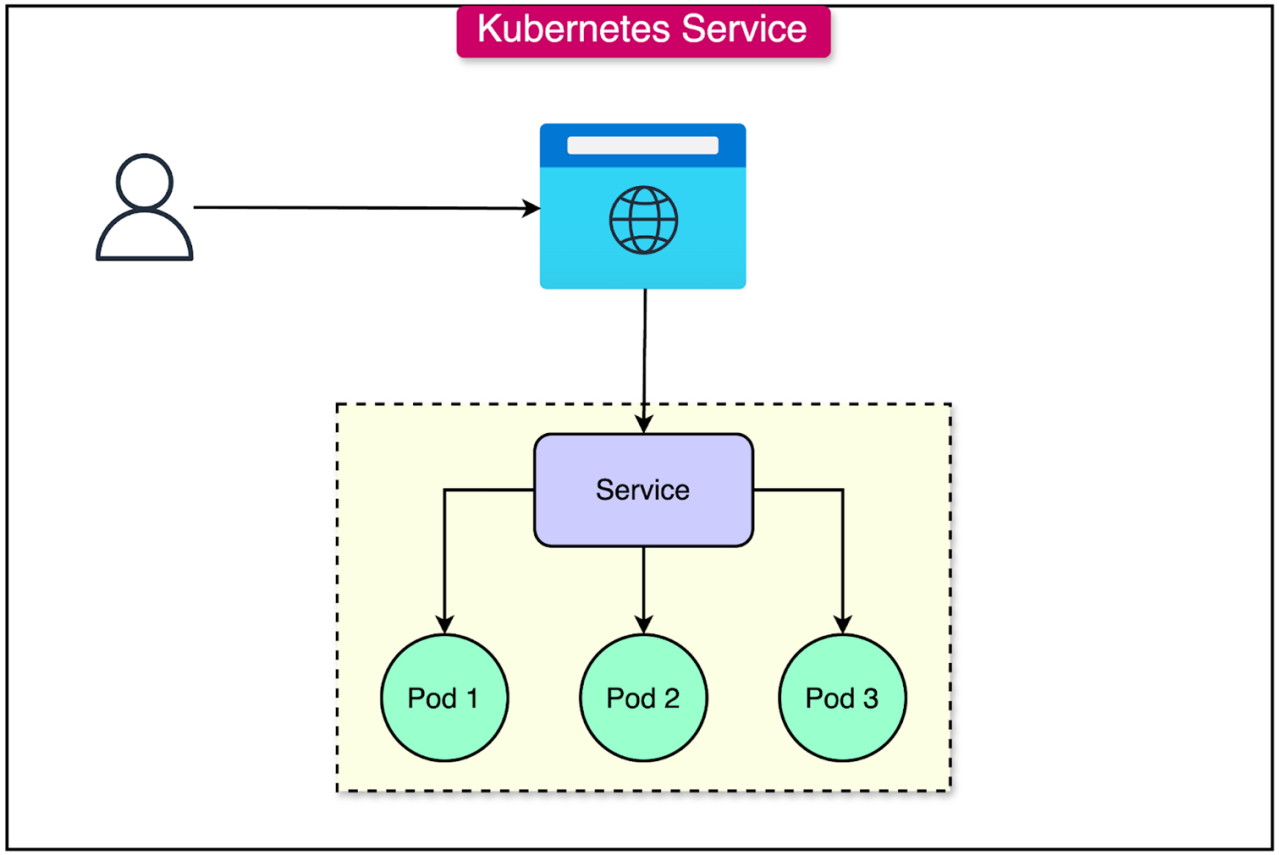
https://devopscube.com/kubernetes-architecture-explained/

Kubernetes Data Model: Service

Service is an abstraction that provides a stable way to expose & access a set of Pods.

Since Pods are ephemeral (they can be terminated or restarted), their IP addresses keep changing (not static).

Services solve this problem by providing a Static Endpoint (IP/DNS name) to connect to a group of Pods.



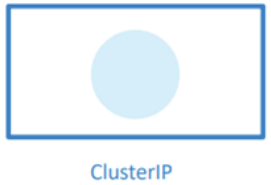
BCIT™

Kubernetes Data Model: Service

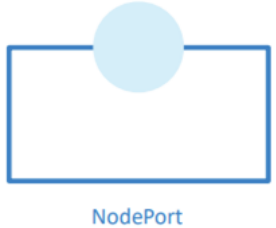
Service uses labels & selectors to determine which Pods they should route traffic to.

Most commonly used types of Services → **ClusterIP, NodePort, LoadBalancer**

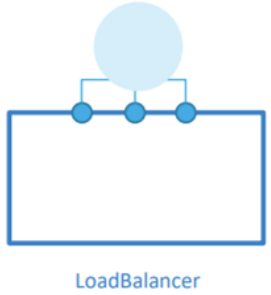
Exposes the service internally within the cluster and creates a virtual IP address only accessible from within the cluster. This is suitable for internal communication between Pods.



It is useful for testing or exposing applications externally without a cloud load balancer. Indeed it exposes the service on a static port (range: 30000–32767) on every node in the cluster. This allows external traffic to access the service using the NodeIP and NodePort.

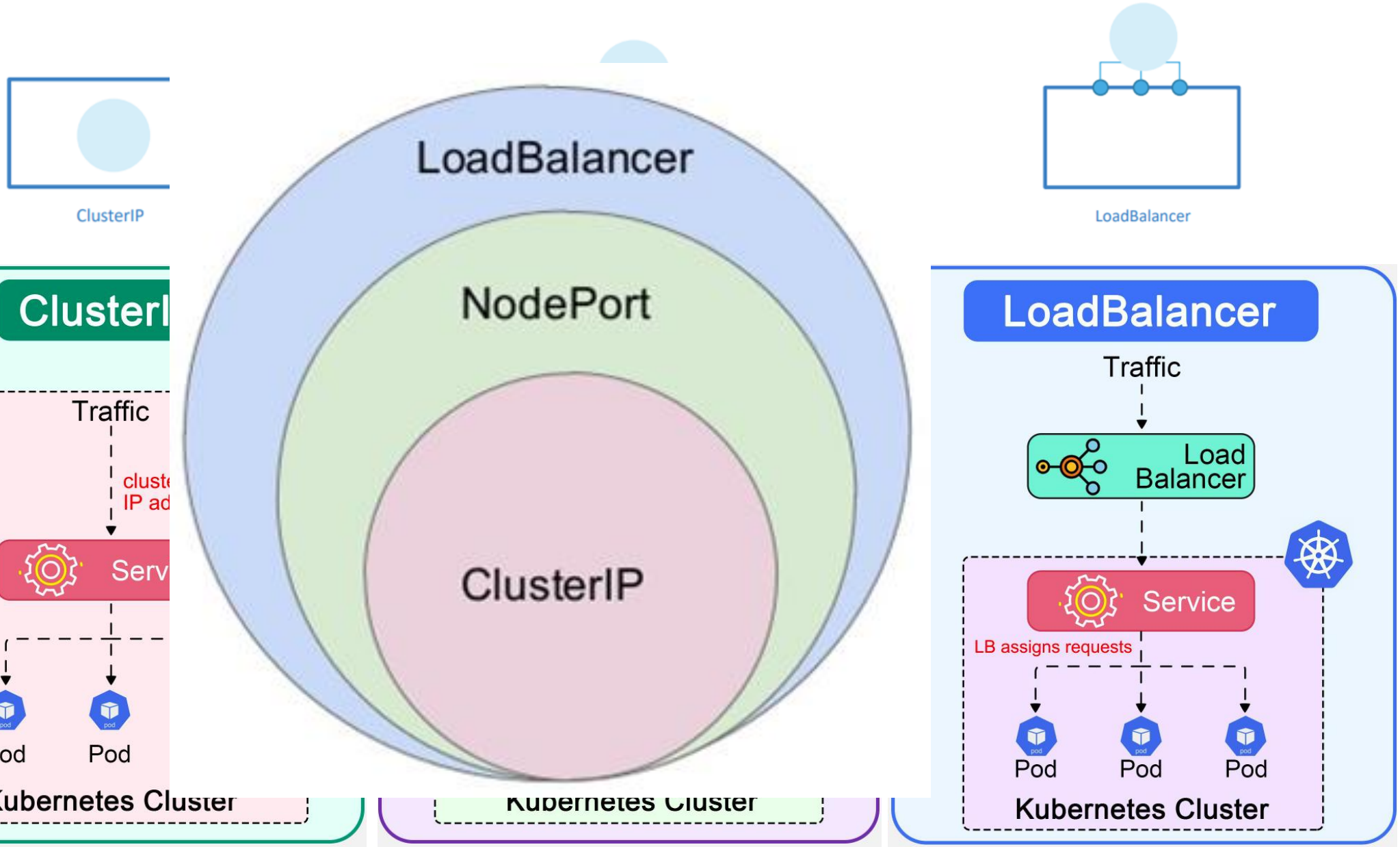


Exposes the service externally using a cloud provider's load balancer (for example, AWS ELB, Azure Load Balancer, or GCP Load Balancer). The service automatically provisions an external IP address and is ideal for production workloads that require external access to the application.



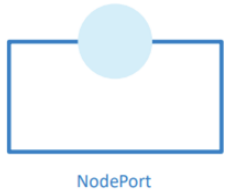
BCIT™

Kubernetes Data Model: Service



BCIT™

Kubernetes Data Model: Service – NodePort example



Service uses labels & selectors to determine which Pods they should route traffic to.

type: NodePort:
Specifies that this service exposes the app on a static port on each node in the K8s cluster.

selector: app: my-app:
Matches the pods labeled with app: my-app, ensuring the service routes traffic to the correct pods (from your deployment).

ports:
port: 80: The port on which the service is accessible.
targetPort: 8080: The container port where traffic will be directed (from nginx container definition in Deployment).
nodePort: 30007: The port exposed on each node's IP. External clients can access app using <NodeIP>:30007.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
  labels:
    app: my-app
spec:
  type: NodePort
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
      nodePort: 30007
```

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

Access **http://<NodeIP>:30007** via browser.

This configuration ensures that your nginx deployment is accessible outside the cluster through the NodePort.

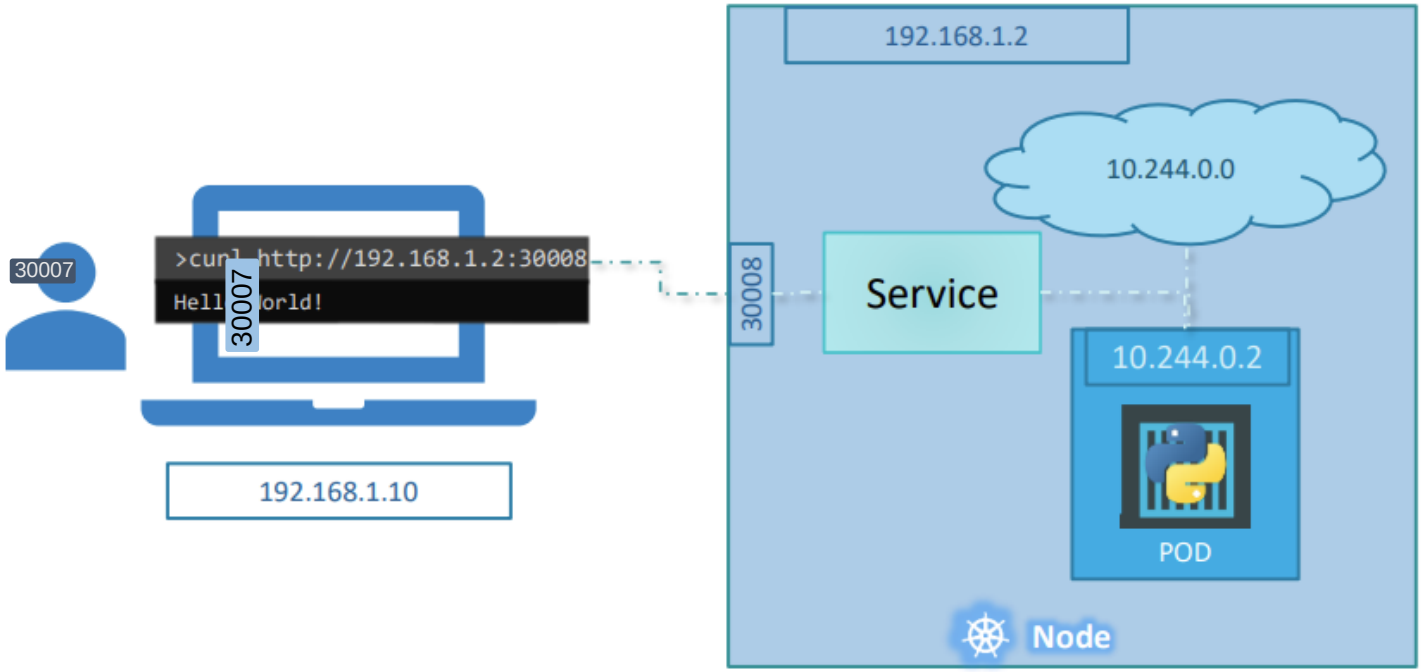
BCIT™

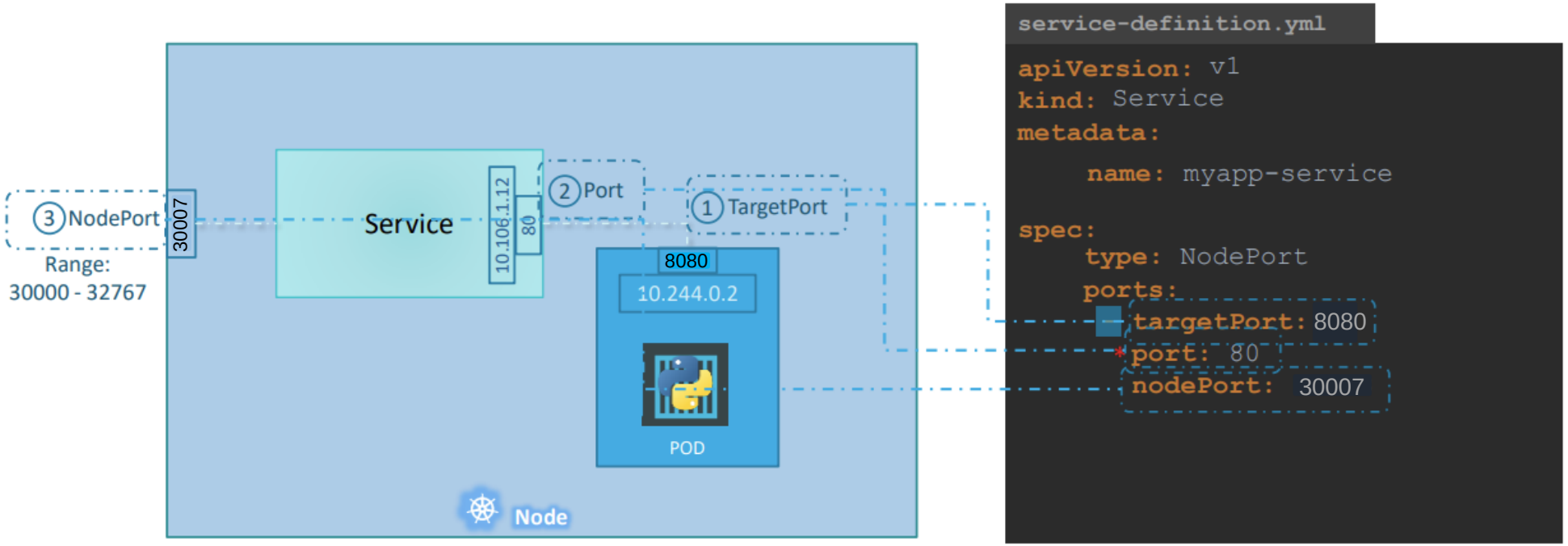
K8s Service (NodePort)

targetPort specifies the port on the pod where the application is running.

port is the internal cluster port that other services/pods use to access this service.

nodePort is the port on each node's IP through which external traffic can reach the service (only applies to NodePort or LoadBalancer types).





BCIT™

K8s Service (NodePort)

service-definition.yml

```yaml
apiVersion: v1
kind: Service
metadata:
    name: myapp-service

spec:
    type: NodePort
    ports:
     - targetPort: 8080
       port: 80
       nodePort: 30007
    selector:
```

pod-definition.yml

```
> kubectl create -f service-definition.yml
service "myapp-service" created
```

```
> kubectl get services
NAME            TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
kubernetes      ClusterIP   10.96.0.1        <none>        443/TCP          16d
myapp-service   NodePort    10.106.127.123   <none>        80:30007 /TCP    5m
```

```
app: myapp
```

```
> curl http://192.168.1.2: 30007
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
```

BCIT™

Linode Kubernetes

What you will learn from this hands-on:

- ✓ **K8s architecture**
- ✓ **Build a FREE K8s Lab**
- ✓ **Install Kubectl**
- ✓ **K8s Pod**
- ✓ **K8s Deployment**
- ✓ **K8s Service (Load Balancer)**

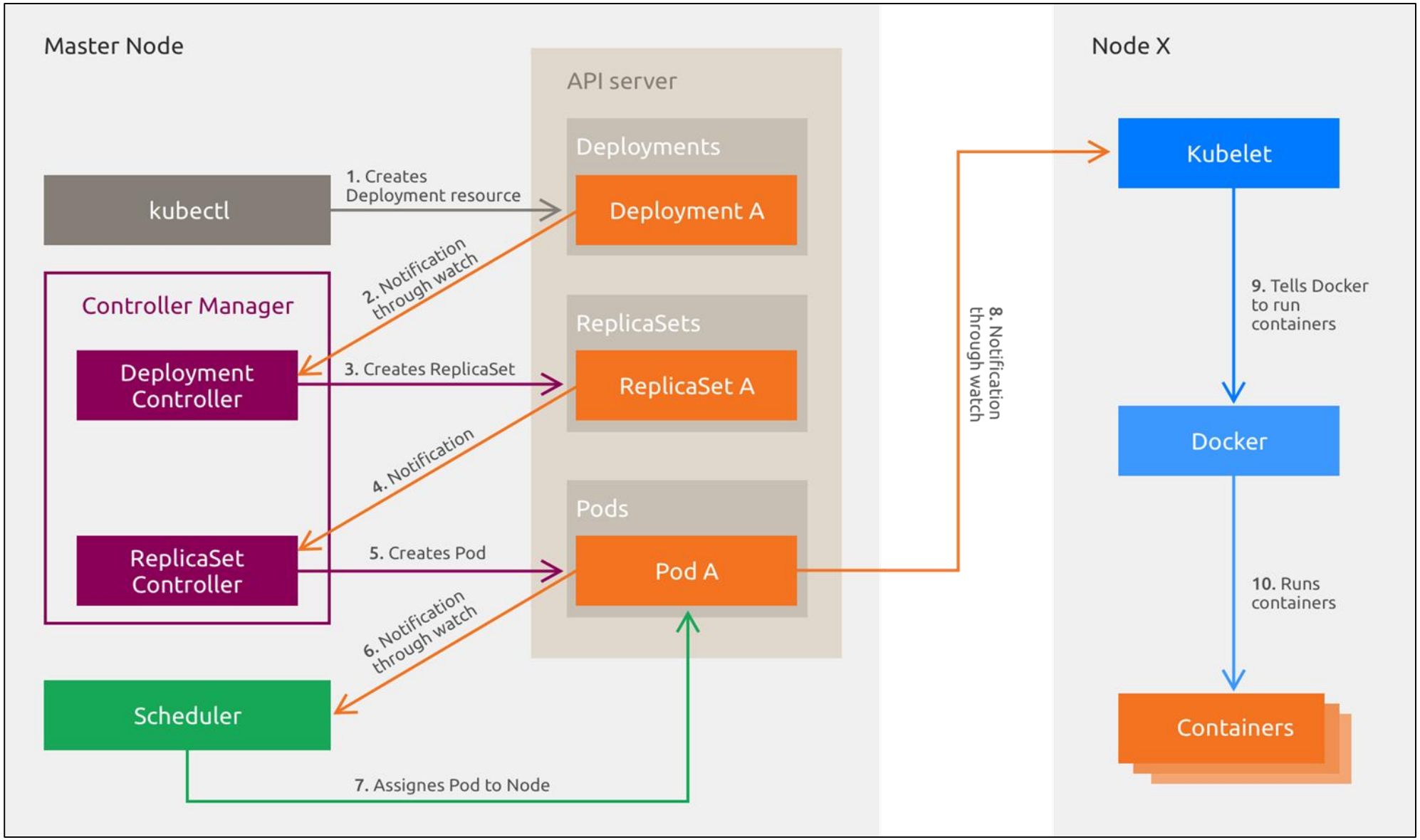
BCIT™

NetworkChuck YouTube Channel

You need to learn Kubernetes Right Now!!

https://www.youtube.com/watch?v=7bA0gTroJjw

Kubernetes Chain of Events



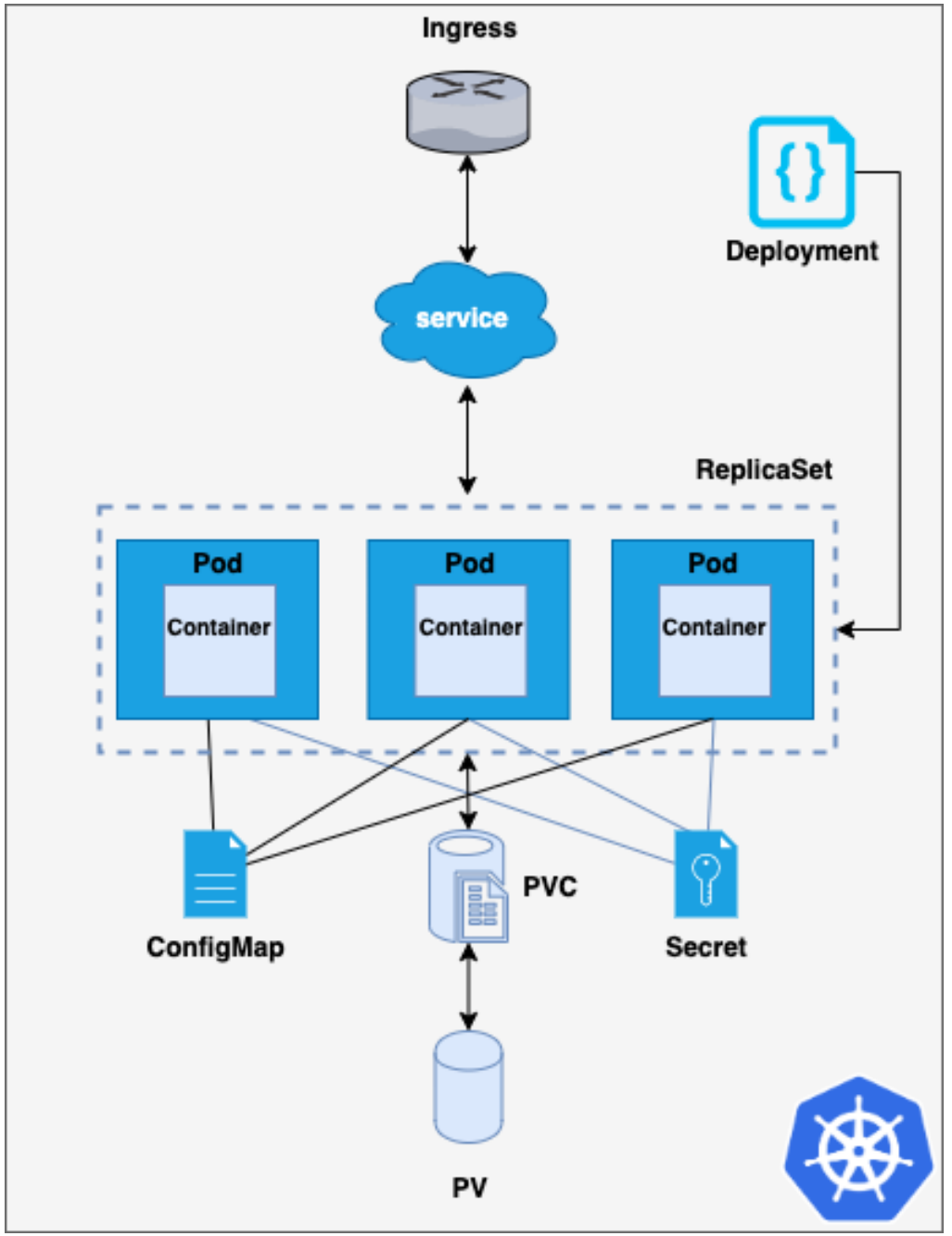
BCIT™

K8s Deployment

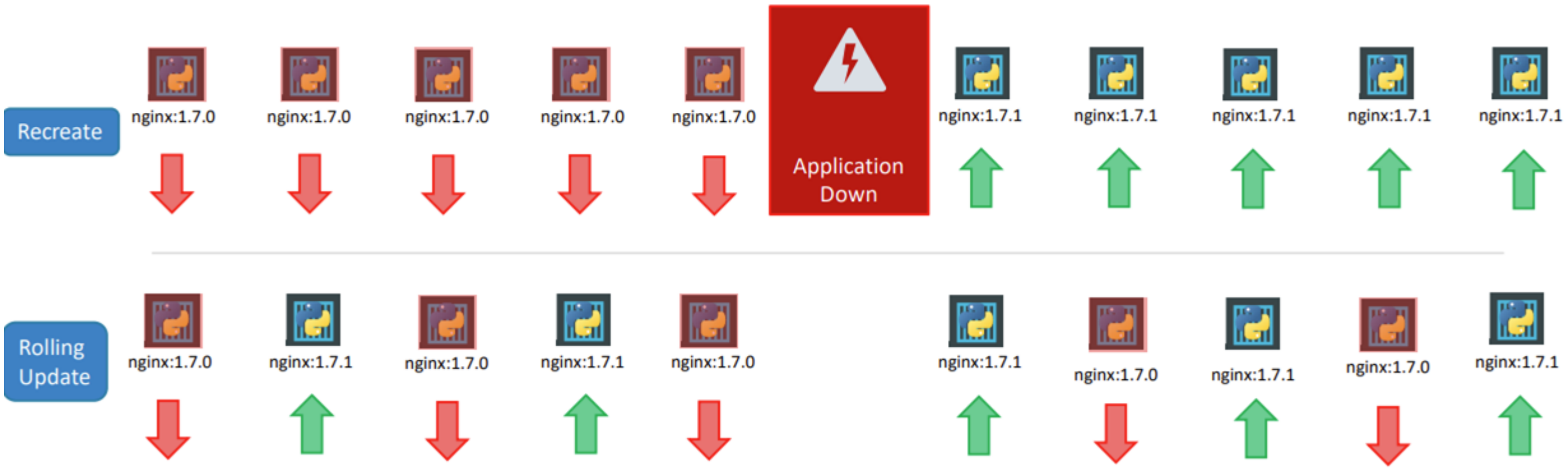
Let's forget about PODs and other K8s concepts & talk about how to deploy an application in a production environment.

Say for example you have a web server that needs to be deployed in a production environment. You need many instances of the web server running. Also, when newer versions of application builds become available on the **Docker Registry**, you would like to **UPGRADE** your Docker instances seamlessly. However, when you upgrade your instances, you do not want to upgrade all of them at once as we just did. This may impact users accessing our applications, so you may want to upgrade them one after the other.

How this can be done seamlessly?



K8s Deployment – Recreate vs. Rolling Update

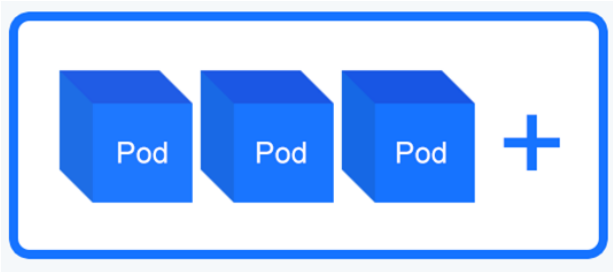


BCIT™

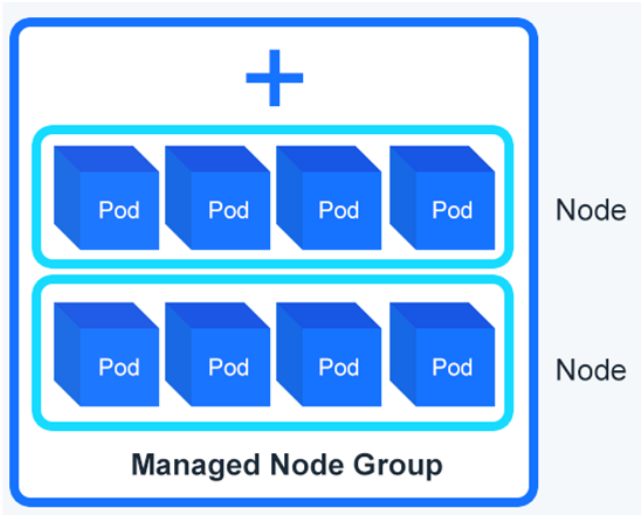
Scaling Applications with Kubernetes

K8s provides powerful scaling mechanisms to ensure applications can handle varying workloads efficiently.

Pod-Level Scaling:
Increasing or decreasing the number of Pods running in a Deployment.
It is achieved using the Horizontal Pod Auto-scaler (HPA).



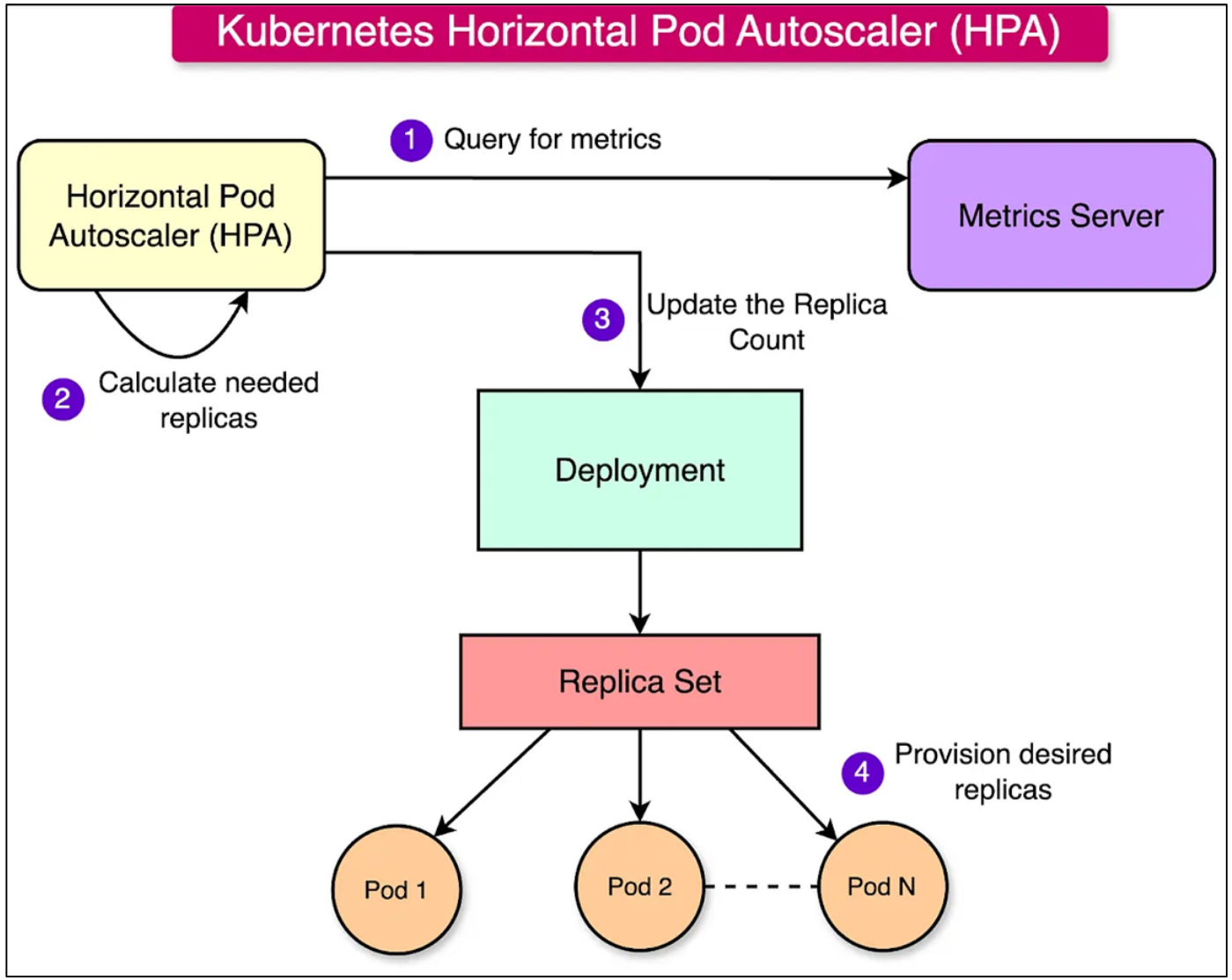
Cluster-Level Scaling:
Dynamically adding or removing nodes in a Kubernetes cluster to accommodate changing resource demands. It is managed by the **Cluster Auto-scaler.**



BCIT™

Scaling Applications with Kubernetes (HPA)

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nginx-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx-deployment
  minReplicas: 2
  maxReplicas: 5
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

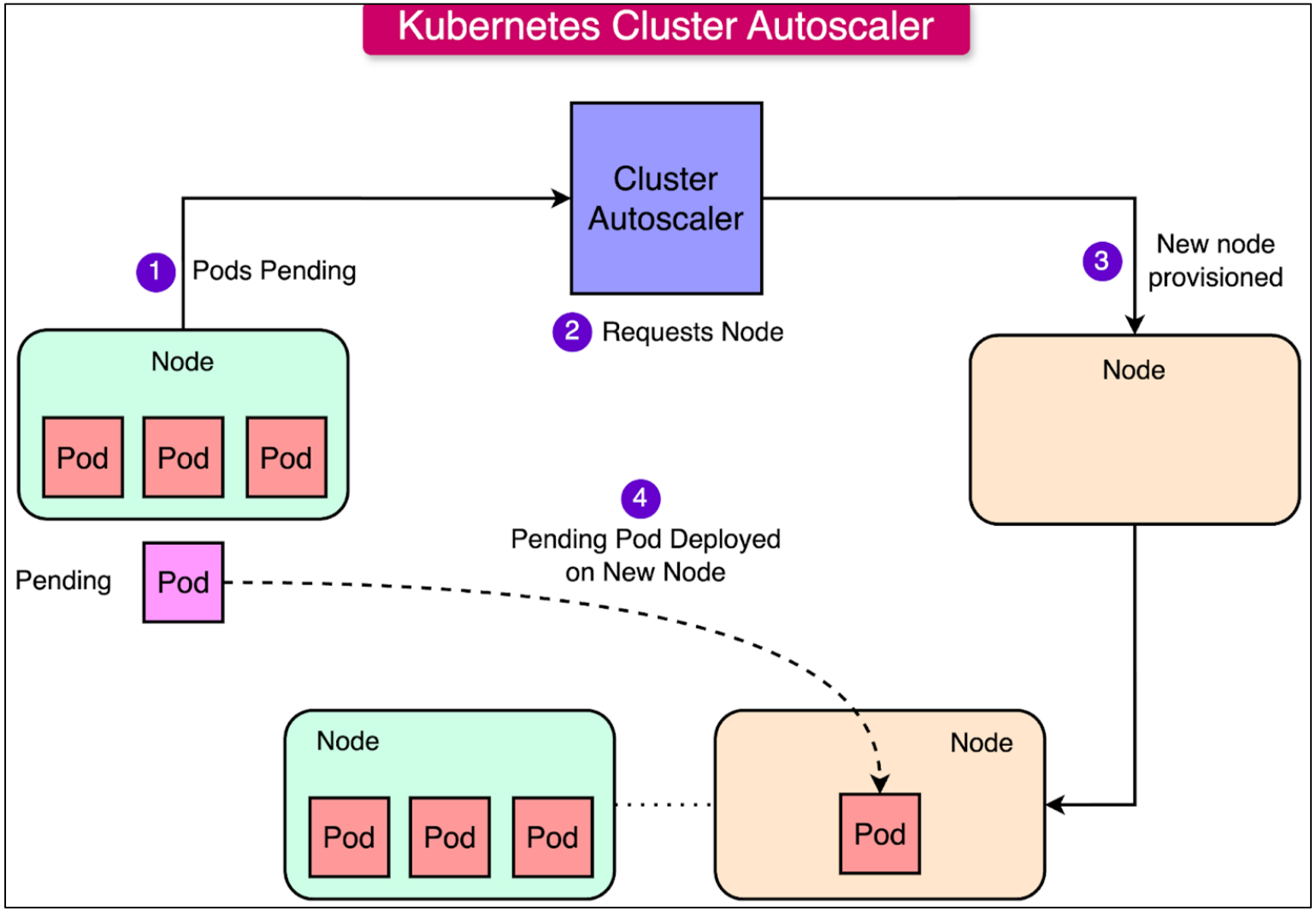


BCIT™

Scaling Applications with Kubernetes (Cluster Auto-scaler)

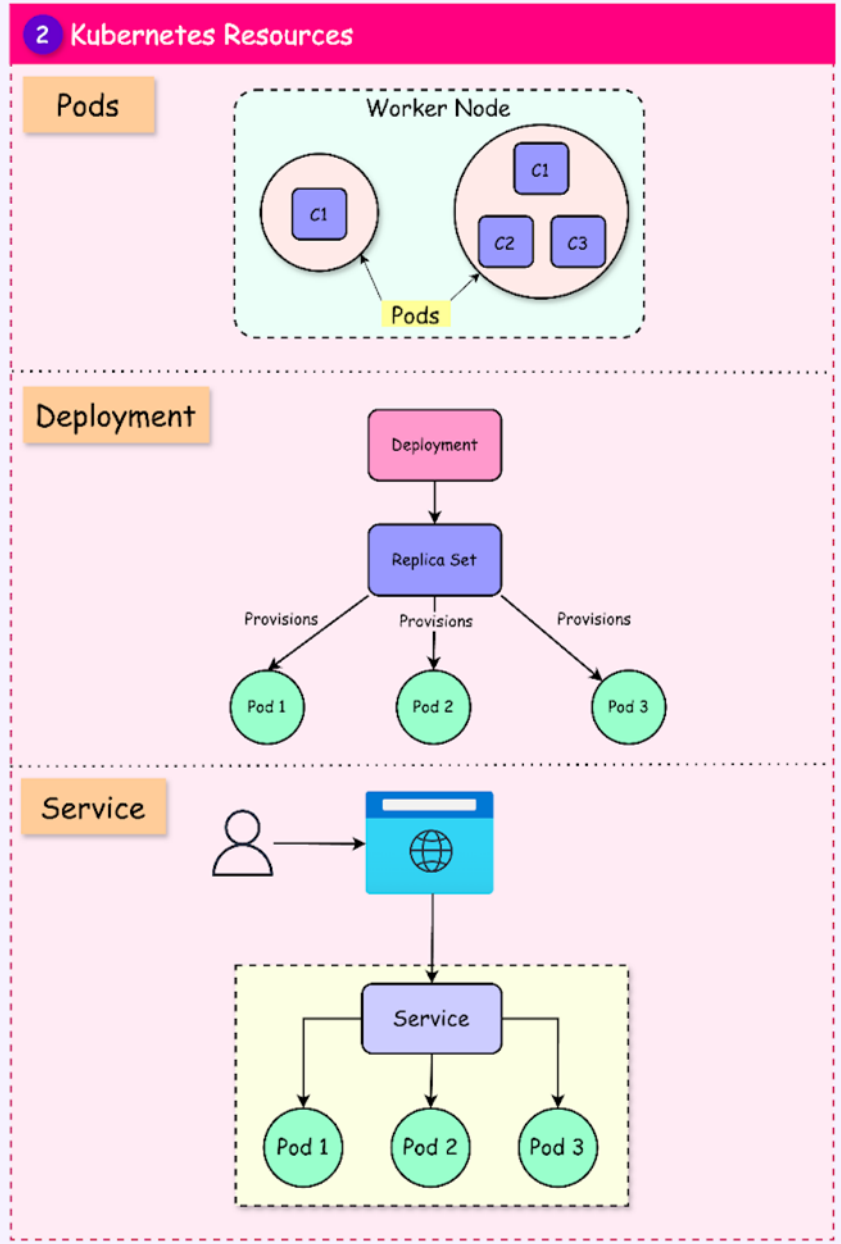
The Cluster Auto-scaler automatically adjusts the number of nodes in a Kubernetes cluster.

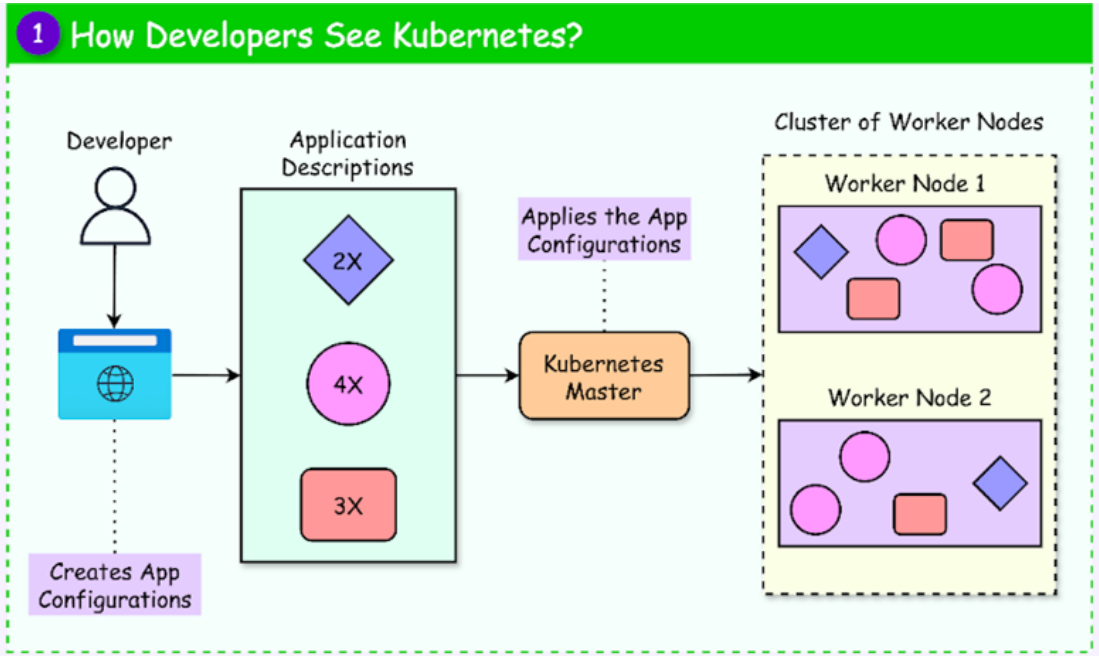
- If the Pods cannot be scheduled due to insufficient resources, the **Cluster Auto-scaler** adds **Nodes**.
- When nodes are underutilized (for example, no Pods running), it removes nodes to optimize resource costs.
- Nodes can be VM or Bare Metal Servers.

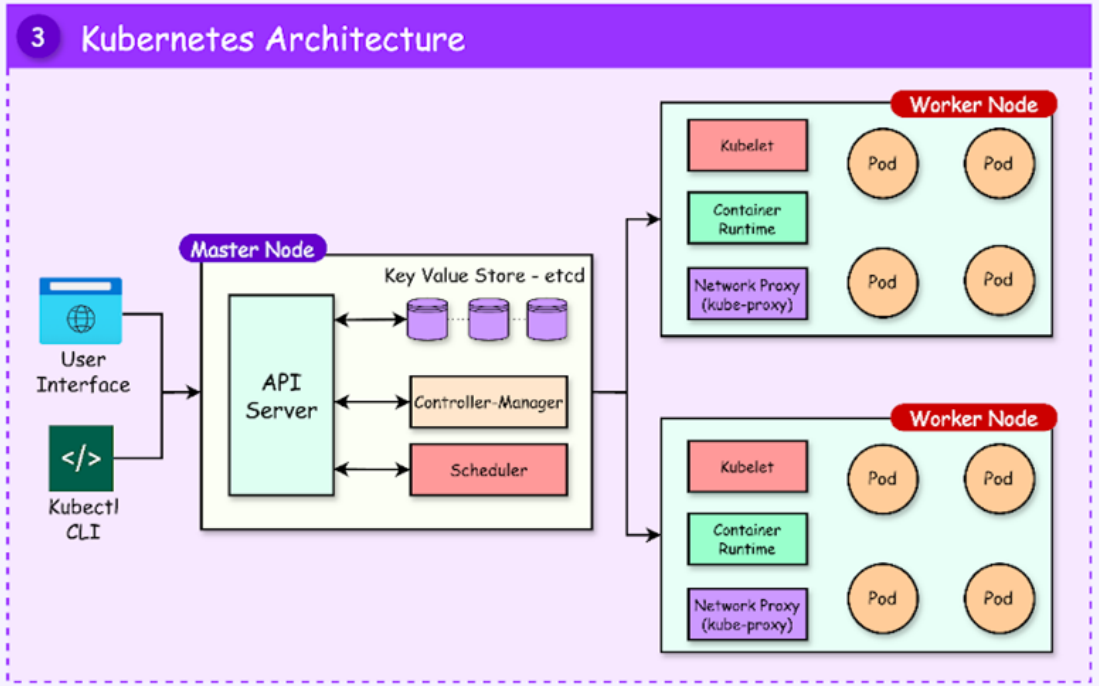


BCIT™

Lecture #4 Summary







BCIT™

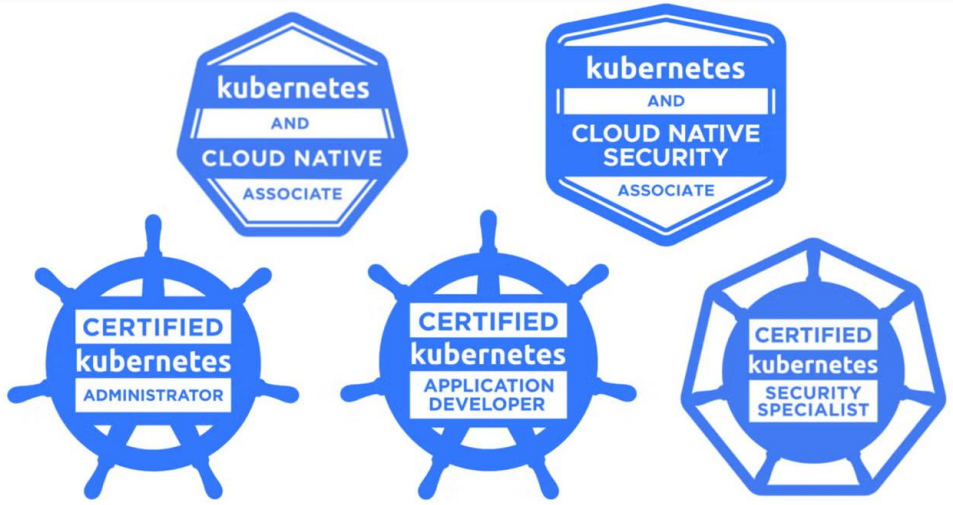
Lecture #4 Summary

- K8s solves challenges like container sprawl, scaling, and failure recovery through automated orchestration.
- K8s operates using a Control Plane and a group of Nodes.
- The **Control Plane** manages the cluster state and includes components such as:
  API Server, ETCD, Controller Manager, and Scheduler.
- The **Nodes** contain the **Kubelet** and **Kube-proxy.**
- Some core K8s concepts are Pods, Deployments, Services, ConfigMaps, Secrets, and Persistent Volumes.
- A **Pod** is the smallest and simplest unit in K8s that represents a single instance of a running process.
- A **Deployment** in K8s is a higher-level abstraction that manages Pods & ensures they run consistently/reliably.
- In K8s, a **Service** is an abstraction that provides a stable way to expose and access a set of Pods.
- K8s provides **ConfigMaps** and **Secrets** to manage application configurations and sensitive data.
- To manage stateful applications in K8s, we need Persistent Volumes (**PVs**) and Persistent Volume Claims (**PVCs**)
- K8s provides **powerful scaling mechanisms** to ensure applications can handle varying workloads efficiently.

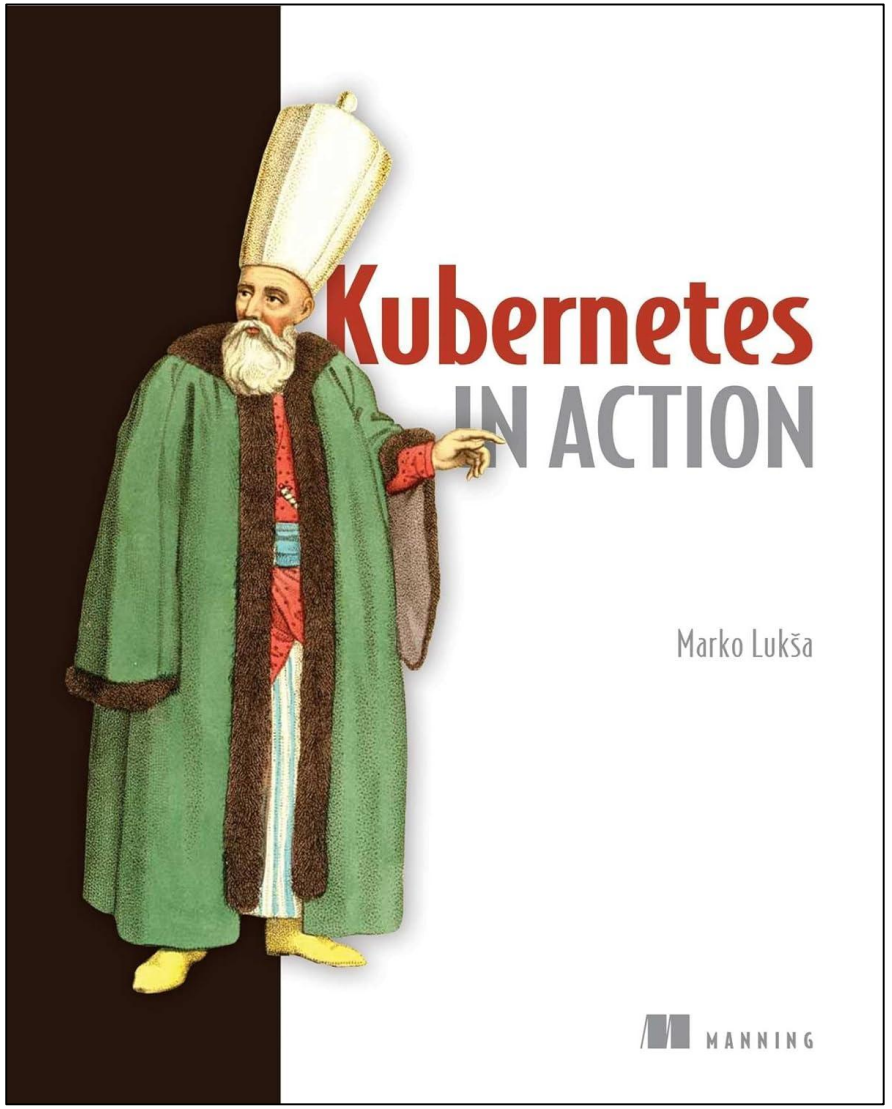
BCIT™

Some Good References

- ✓ **Kubernetes Official Documents**
  https://kubernetes.io/docs/tutorials/kubernetes-basics/
- ✓ **DevOpsCube Blog (nice K8s images)**
  https://devopscube.com/kubernetes-architecture-explained/
- ➢ **NetworkChuck (YouTube Channel)**
- ➢ **TechWorld with Nana (YouTube Channel)**
- ○ **Kubernetes in Action,** Book (Buy from Amazon)



https://training.linuxfoundation.org/certification/kubestronaut-bundle/
https://www.cncf.io/training/kubestronaut/



By Marko Luksa, 2018

End of Lecture #4

THANK

Y☺U

- Dawood Sajjadi

BCIT™