Caltech | Center for Technology & Management Education

Post Graduate Program in DevOps

simplilearn

Continuous Orchestration Using Kubernetes

Learning Objectives

By the end of this lesson, you will be able to:

- Describe container orchestration
- Explain the Kubernetes components and its features
- Demonstrate the creation of a Kubernetes cluster
- Deploy an application to the Kubernetes cluster

Container Orchestration

Container Orchestration

Container orchestration is the automation and management of lifecycle of containers and services.

Purpose

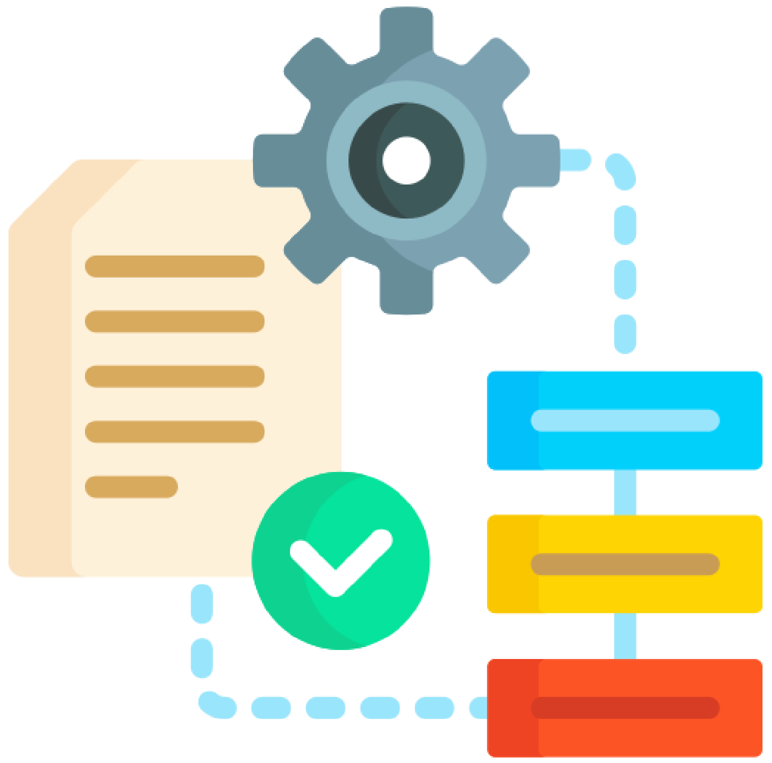
- A container orchestrator automatically deploys and manages containerized apps.
- It responds dynamically to changes in the environment to increase or decrease the deployed instances of the managed app.
- It ensures all deployed container instances get updated if a new version of a service is released.

Why Do We Need Container Orchestration?

Container orchestration is essential to automate and manage tasks such as:

- Provisioning and deployment
- Configuration and scheduling
- Resource allocation
- Container availability
- Scaling or removing containers
- Load balancing and traffic routing
- Monitoring container health
- Secure container interactions

How Does Container Orchestration Work?



- Container orchestration tools use configuration files (YAML or JSON) that specify where to find the container images, how to establish a network, and where to store the logs.
- While deploying a new container, the orchestration tool automatically schedules the deployment to a cluster and finds the right host, taking into account any defined requirements or restrictions.
- It then manages the container’s lifecycle based on the specifications in the config files.
- These tools can be used in any environment that runs containers.

Container Orchestration Tools

Docker Swarm

kubernetes

HashiCorp
Nomad

OPENSHIFT

Apache
MESOS™

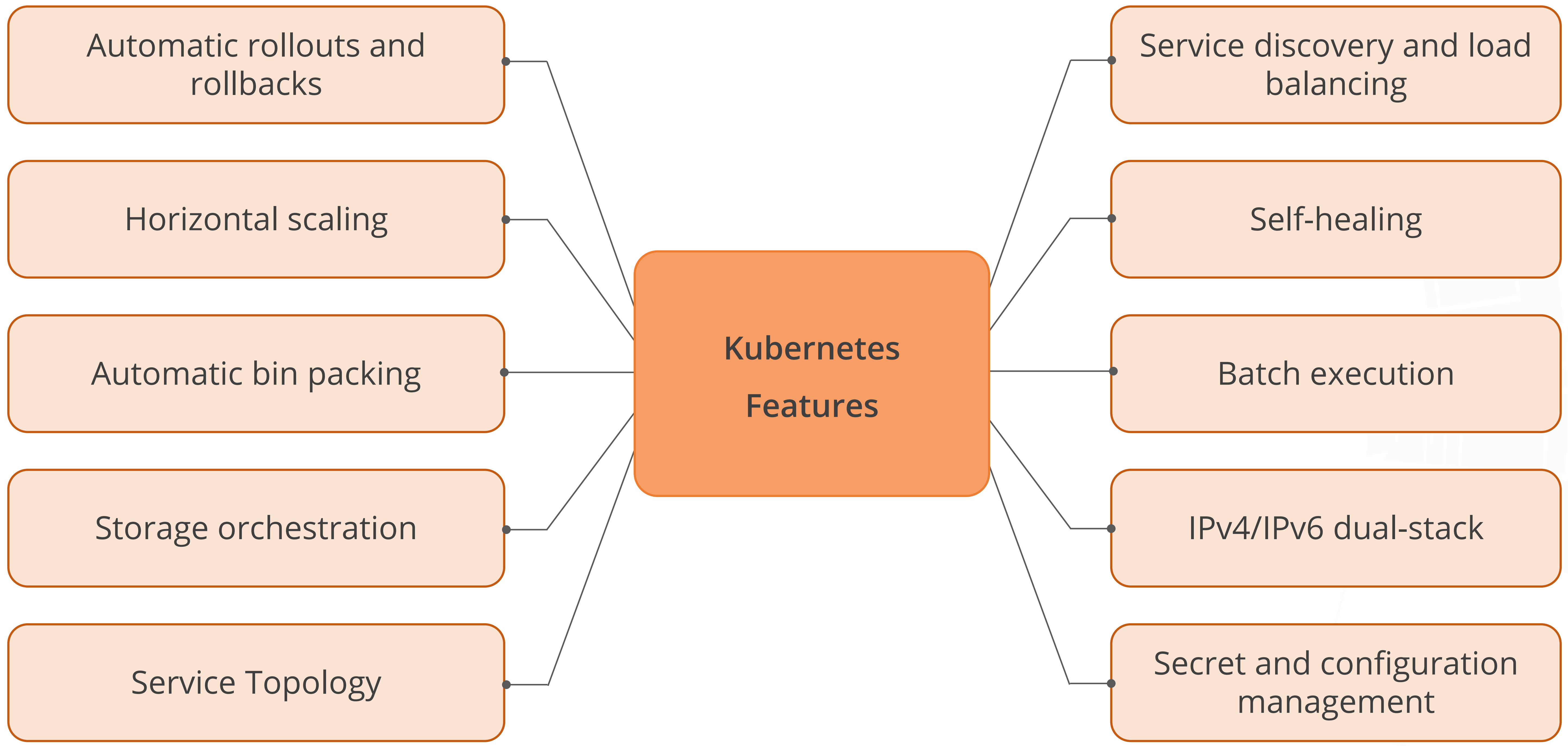
Introduction to Kubernetes

Introduction to Kubernetes

- Open-source orchestration tool used to manage containerized applications
- Originally developed by Google and acquired by Cloud Native Computing Foundation
- Automates deployment and scales containers across multiple nodes in Kubernetes clusters
- Designed in GO language
- Acts as a cloud service in major cloud providers such as EKS in AWS and Kubernetes Engine in GCP

kubernetes

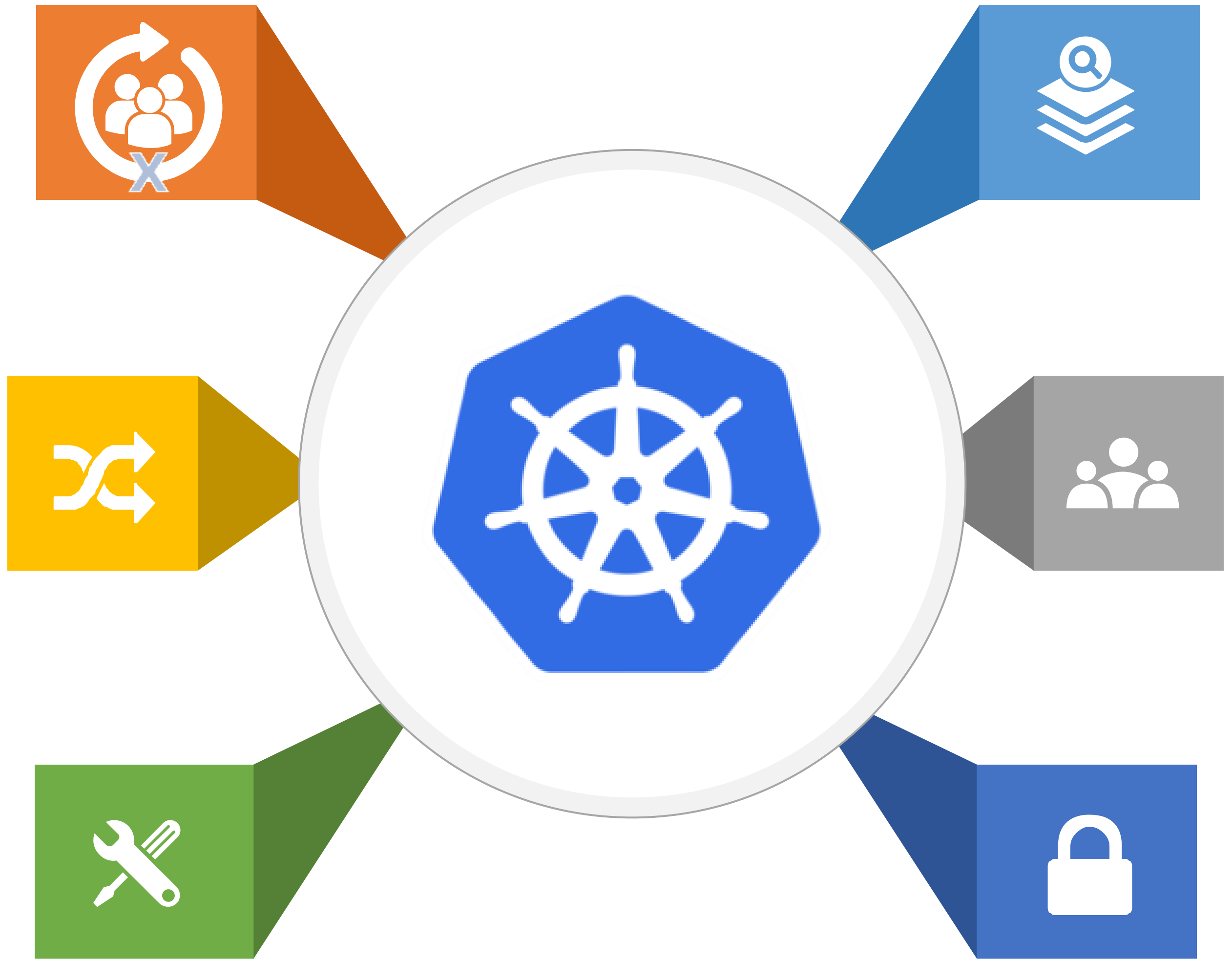
Features of Kubernetes



Docker Swarm Vs. Kubernetes

	Docker Swarm	Kubernetes
Installation and cluster setup	Easy and fast to install and configure	Takes some work to get up and running
Autoscaling	Cannot do autoscaling	Can do autoscaling
GUI	There is no GUI	GUI is the Kubernetes dashboard
Scalability	Highly scalable, scales five times faster than Kubernetes	Highly scalable, but scaling and deployment are slow
Load balancing	Automatic load balancing of traffic	Manual intervention needed for load balancing
Container setup	Limited to Docker API capabilities	Can overcome Docker API constraints
Logging and monitoring	Need third party tools for logging and monitoring	Built-in tools for logging and monitoring

Benefits of Kubernetes



Open source and modular

Multi-cloud capability

Easy service organization with pods

Portability and flexibility

Huge community support

Increased developer productivity

Case Study: Spotify

Spotify is an audio-streaming platform launched in 2008 and has grown to over 200 million monthly active users across the world.

With a goal to enable an immersive listening experience for all of its consumers, Spotify became an early adopter of microservices and Docker.

Case Study: Spotify

Challenge

Spotify had containerized microservices running across its fleet of virtual machines with a homegrown container orchestration system called Helios. By late 2017, it was clear that having a small team working on the features was just not as efficient as adopting something that was supported by a much bigger community.

Case Study: Spotify

Solution

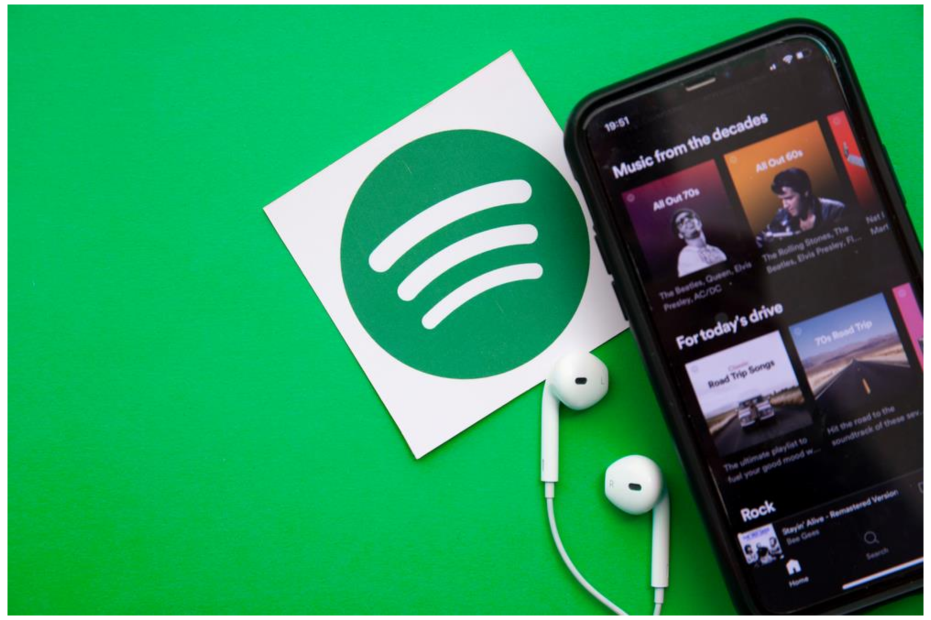
Adopting Kubernetes was the solution!

It was more feature-rich than Helios. The company could benefit from added velocity and reduced cost and also align with the rest of the industry on best practices and tools.

The migration would be smooth and in parallel with Helios running, as Kubernetes fits very nicely as a complement and would be a great replacement to Helios.

Case Study: Spotify

The impacts of adopting Kubernetes by Spotify are given below:

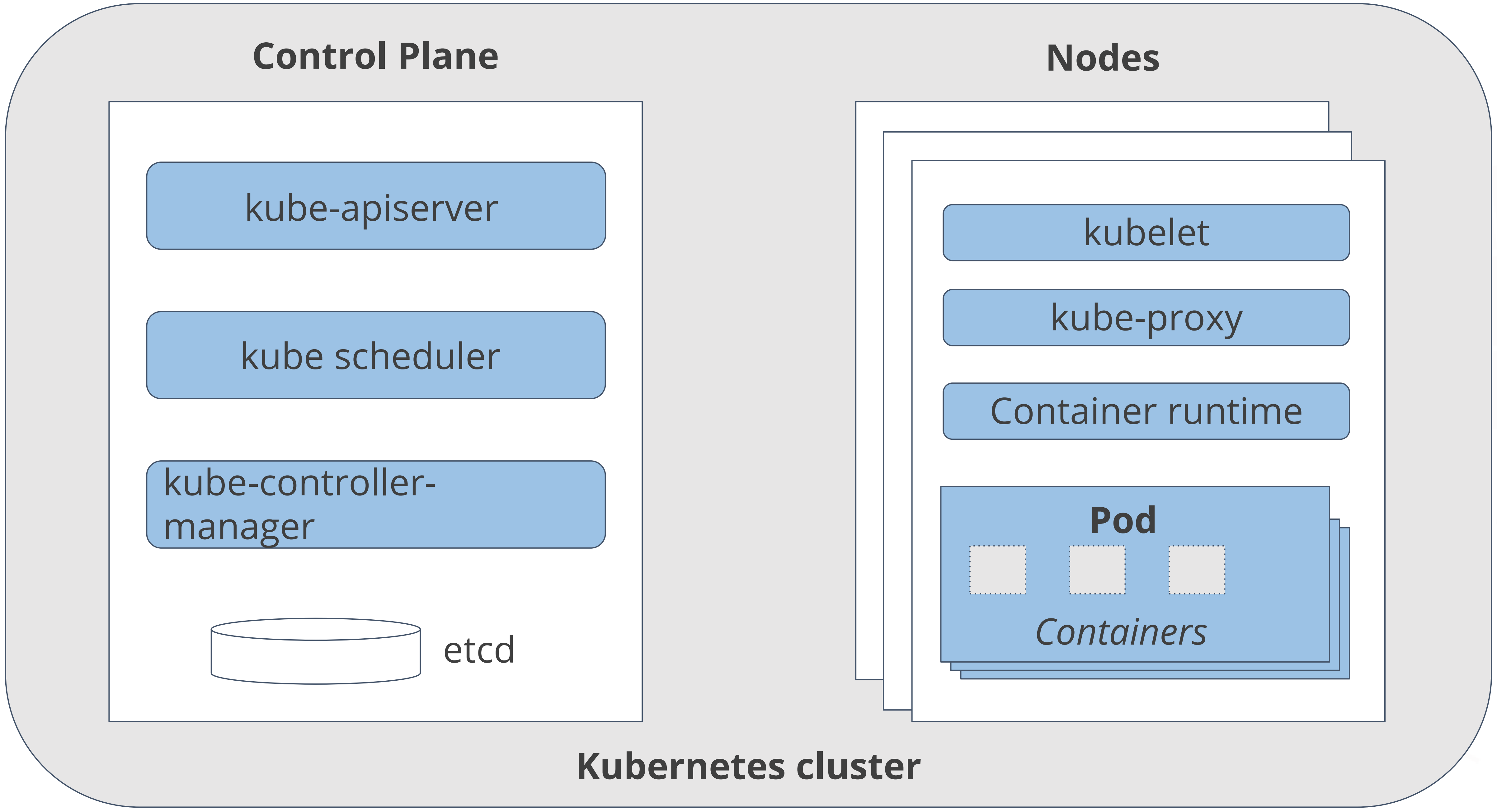


- Less need to concentrate on manual capacity provisioning and more time to focus on delivering features for Spotify.
- The largest service currently running on Kubernetes takes around 10 million requests each second and benefits significantly from autoscaling.
- Teams that previously had to wait for an hour to create a new service and get an operational host to run it in production, can do it in the order of seconds and minutes with Kubernetes.
- With it's bin-packing and multi-tenancy capacities, CPU usage has enhanced on average two- to threefold.

Kubernetes Architecture

Kubernetes Components

A working Kubernetes deployment is called a cluster. It has the following components:



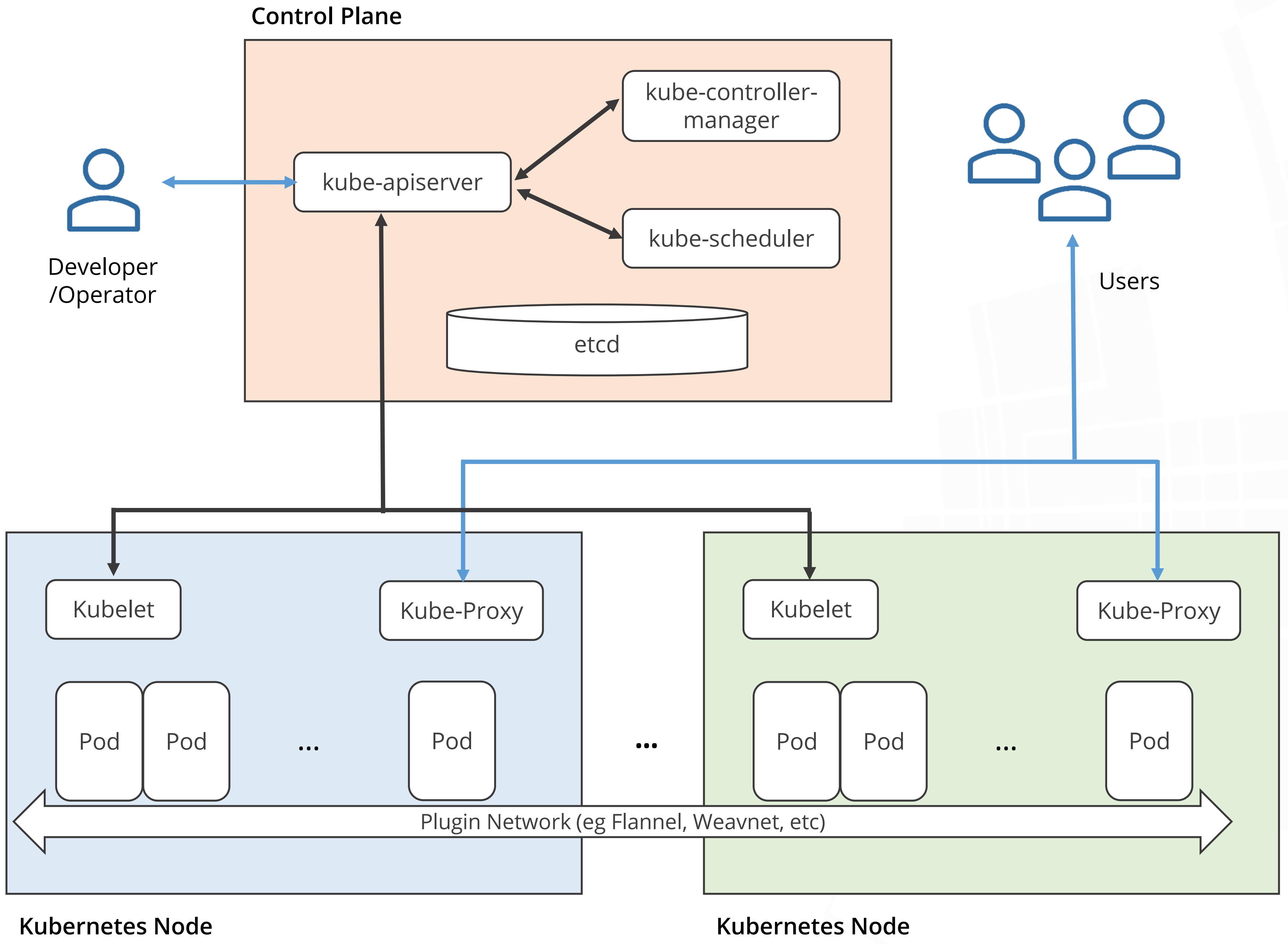
Kubernetes Components

- **Nodes**: Sets of worker machines on the Kubernetes cluster that run containerized applications
- **Pods**: Components of the application workload that run on the worker nodes
- **Control Plane**: Manages the worker nodes and the pods in the cluster

In production environments, the control plane runs across multiple computers and a cluster runs multiple nodes, providing fault tolerance and high availability.

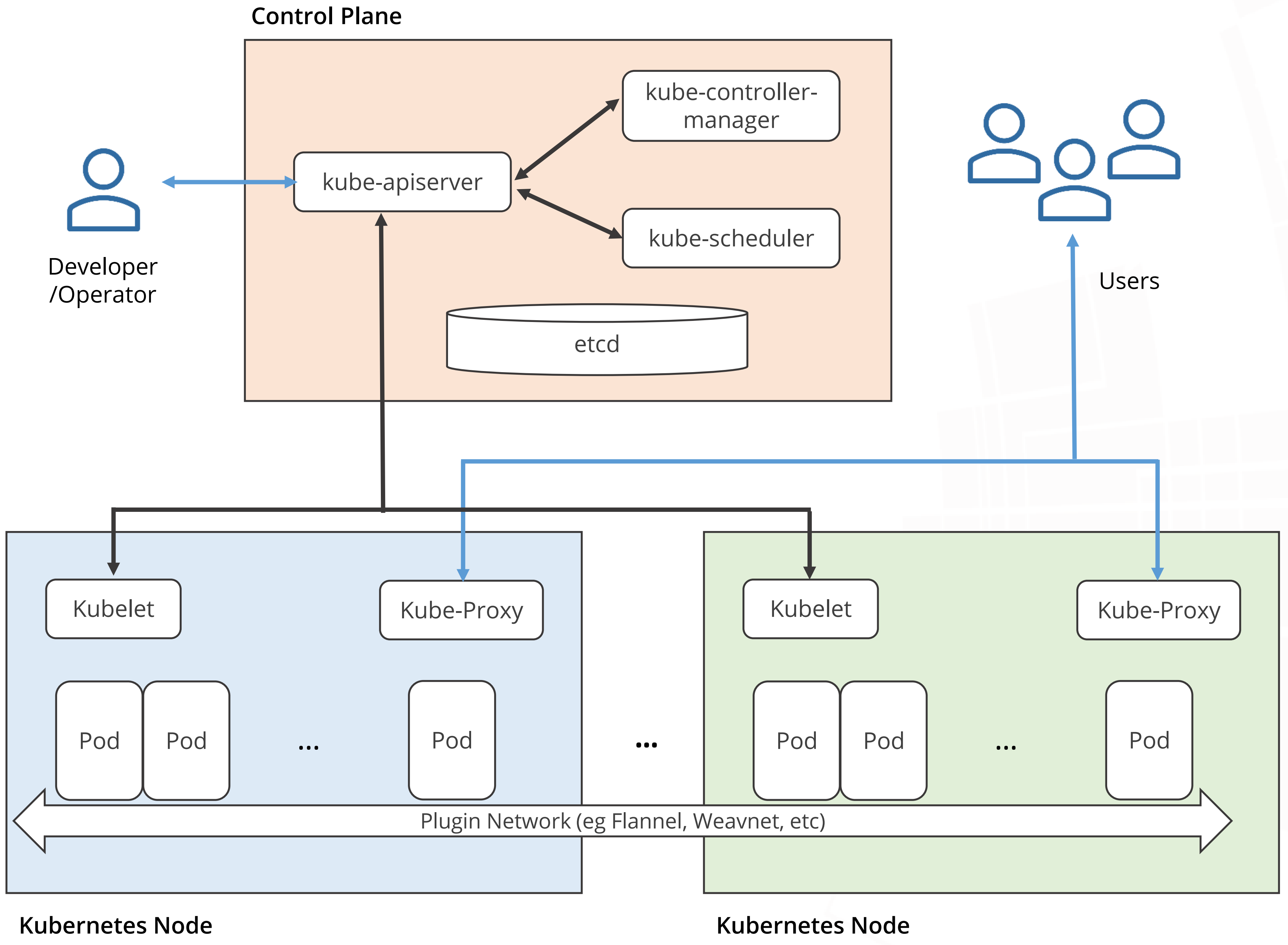
Kubernetes Architecture

- The Kubernetes cluster contains at least one control plane and one or more worker nodes.
- Control plane contains the following components:
    - etcd
    - kube-apiserver
    - kube-scheduler
    - kube-controller-manager

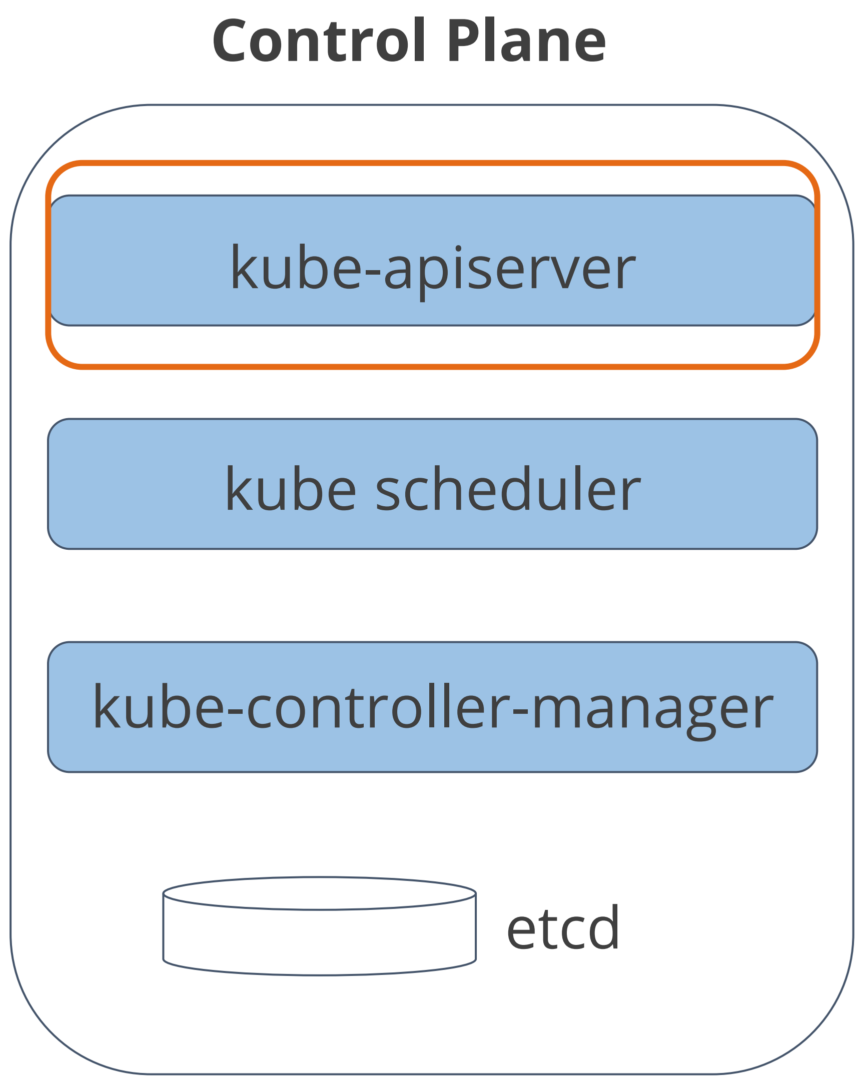


Kubernetes Architecture

- Kubernetes node contains the following architecture components:
  - kubelet
  - kube-proxy
  - Pod
  - Container runtime



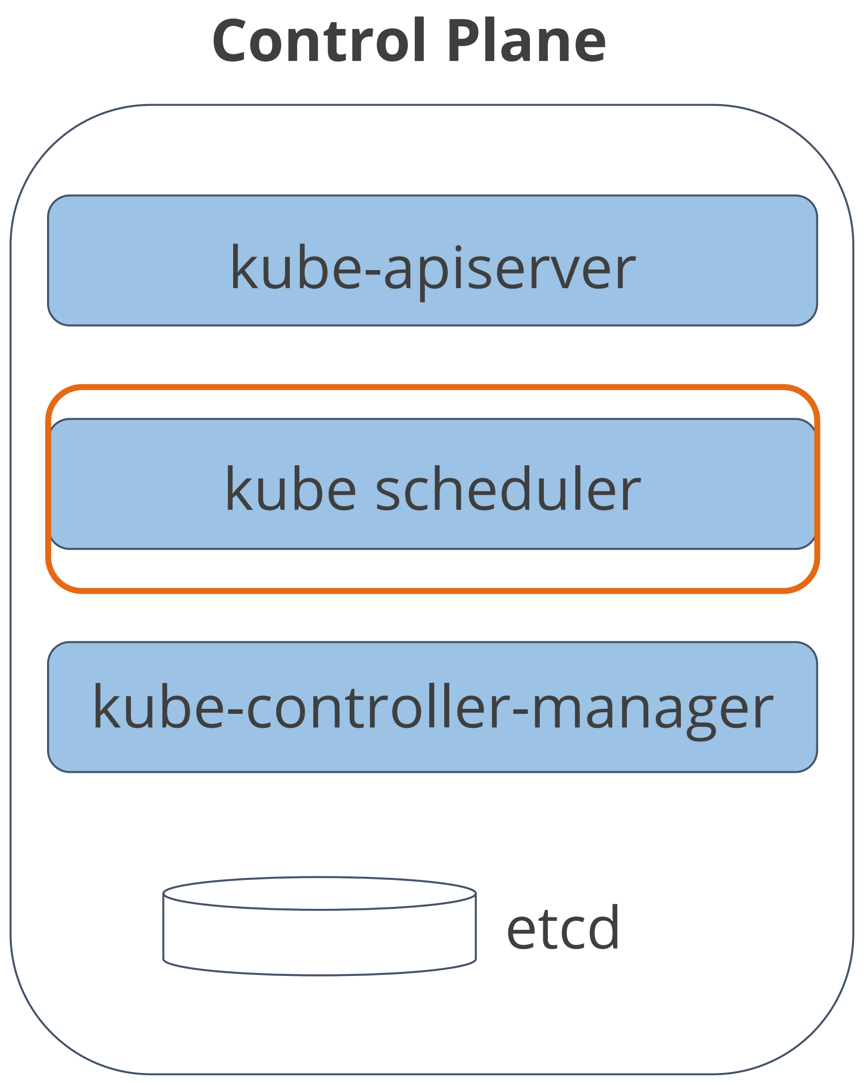
Kubernetes Control Plane Components



kube-apiserver

- kube-apiserver supports Kubernetes API and processes all the requests from various components.
- It handles the REST requests and JSON requests and updates the state of each object in etcd.

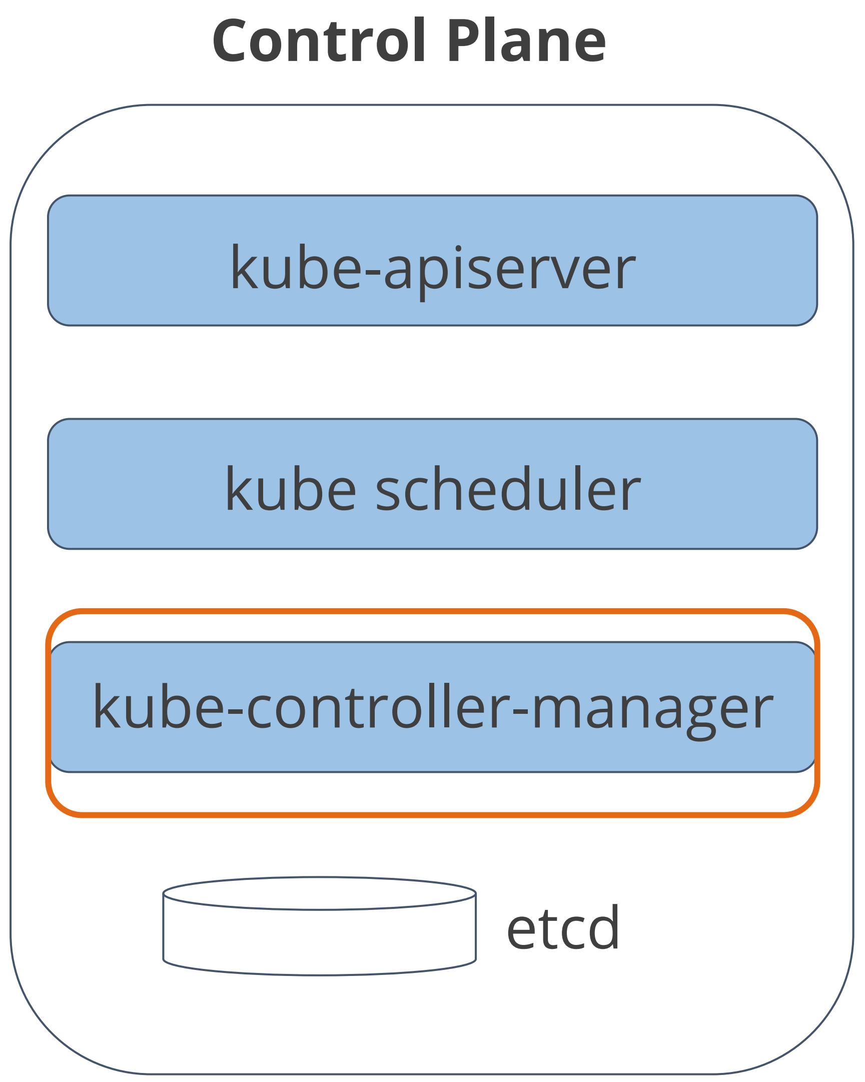
Kubernetes Control Plane Components



kube-scheduler

- kube-scheduler is the component of Kubernetes responsible for managing workloads in a cluster.
- It identifies the unutilized node and the process to schedule pods on unutilized nodes based on the requirements.
- It helps to manage all Kubernetes resources effectively.

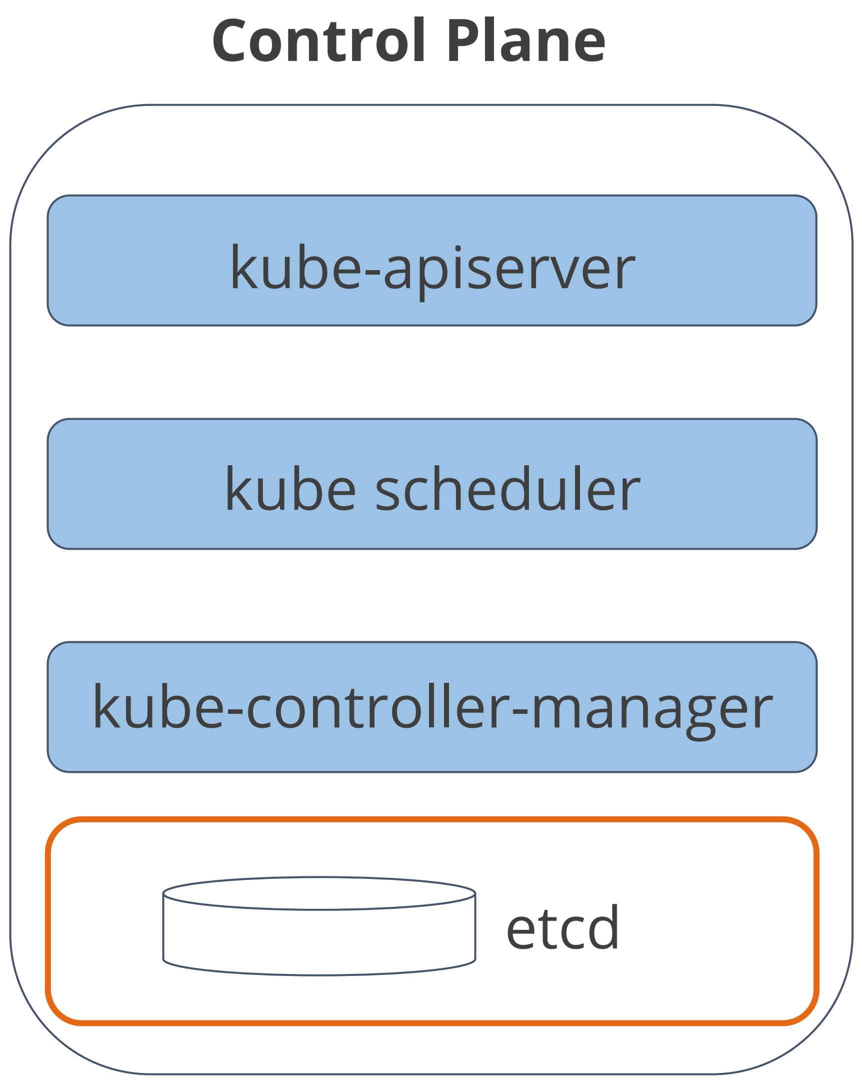
Kubernetes Control Plane Components



kube-controller-manager

- kube-controller-manager manages all controllers in Kubernetes such as DaemonSet and ReplicationController.
- It interacts with the API server to create, edit, and delete any resources being managed.

Kubernetes Control Plane Components



etcd

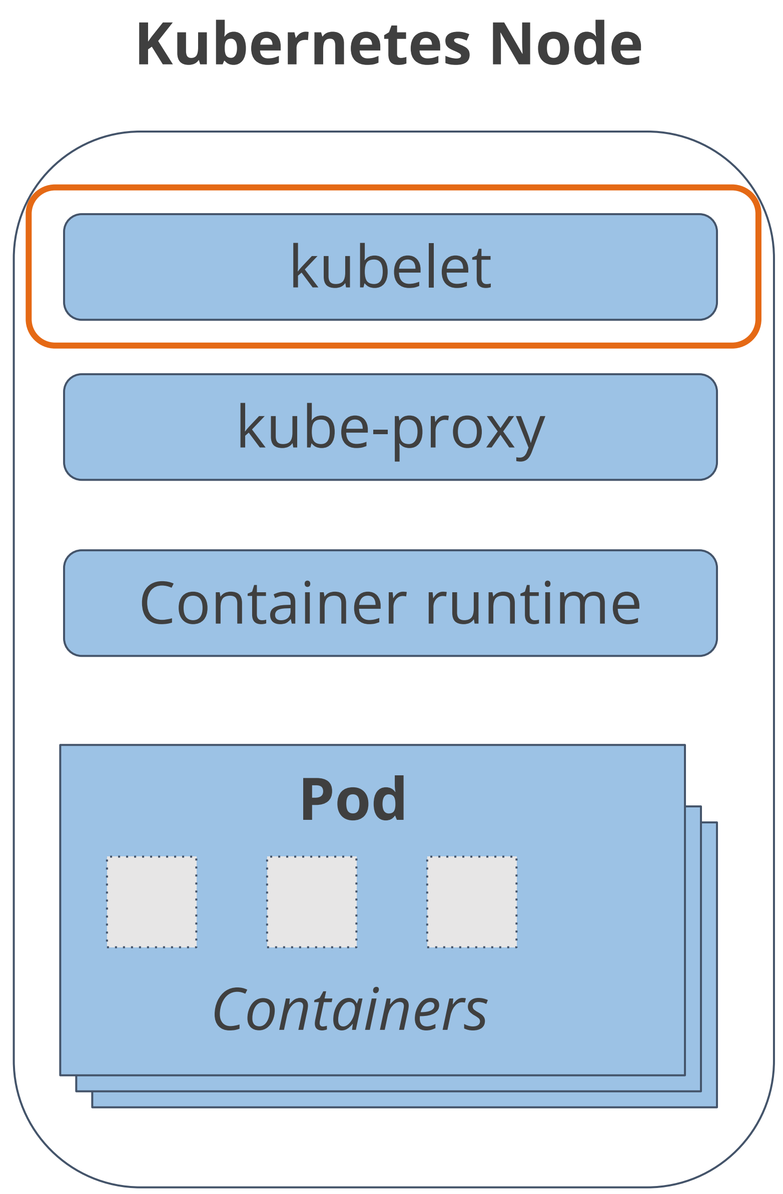
- etcd is a persistent, lightweight, and key-value data store.
- It stores the complete configuration data of a Kubernetes cluster.
- You can check the state of a cluster with the available data anytime.
- This data store can be shared with other components.
- It provides a data layer in Kubernetes clusters.

Kubernetes Control Plane Components

cloud-controller-manager

- It occurs as a part of the control plane components only if you run kubernetes with a specific cloud provider.
- It incorporates cloud-specific control logic and helps to link your kubernetes cluster with the cloud provider’s API.
- It only runs the controllers that are specific to your cloud provider.

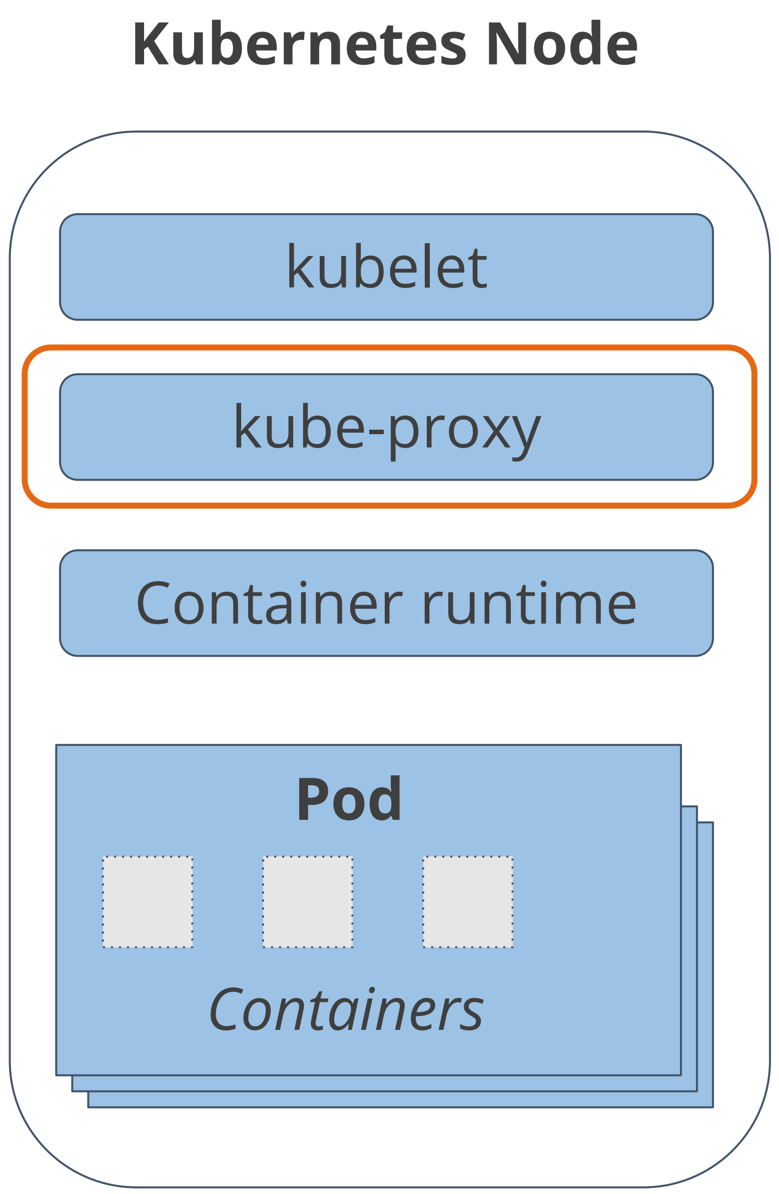
Kubernetes Node Components



kubelet

- Kubelet is responsible for the working of each node and ensuring the container's health.
- It monitors how the pods start, stop, and are maintained.
- It does not manage containers that are not created by kubernetes.

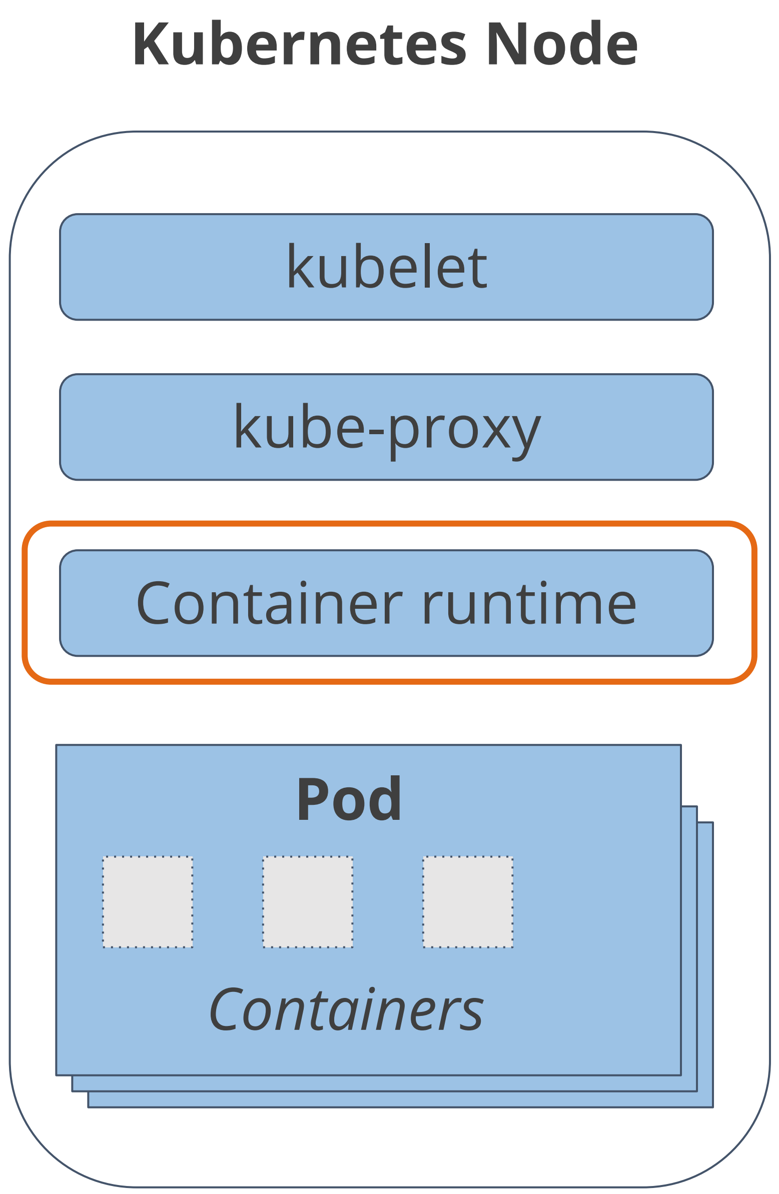
Kubernetes Node Components



kube-proxy

- kube-proxy implements network proxy and acts as a load balancer in Kubernetes cluster.
- It helps to redirect traffic to a specific container in a pod based on the incoming port and IP details.

Kubernetes Node Components



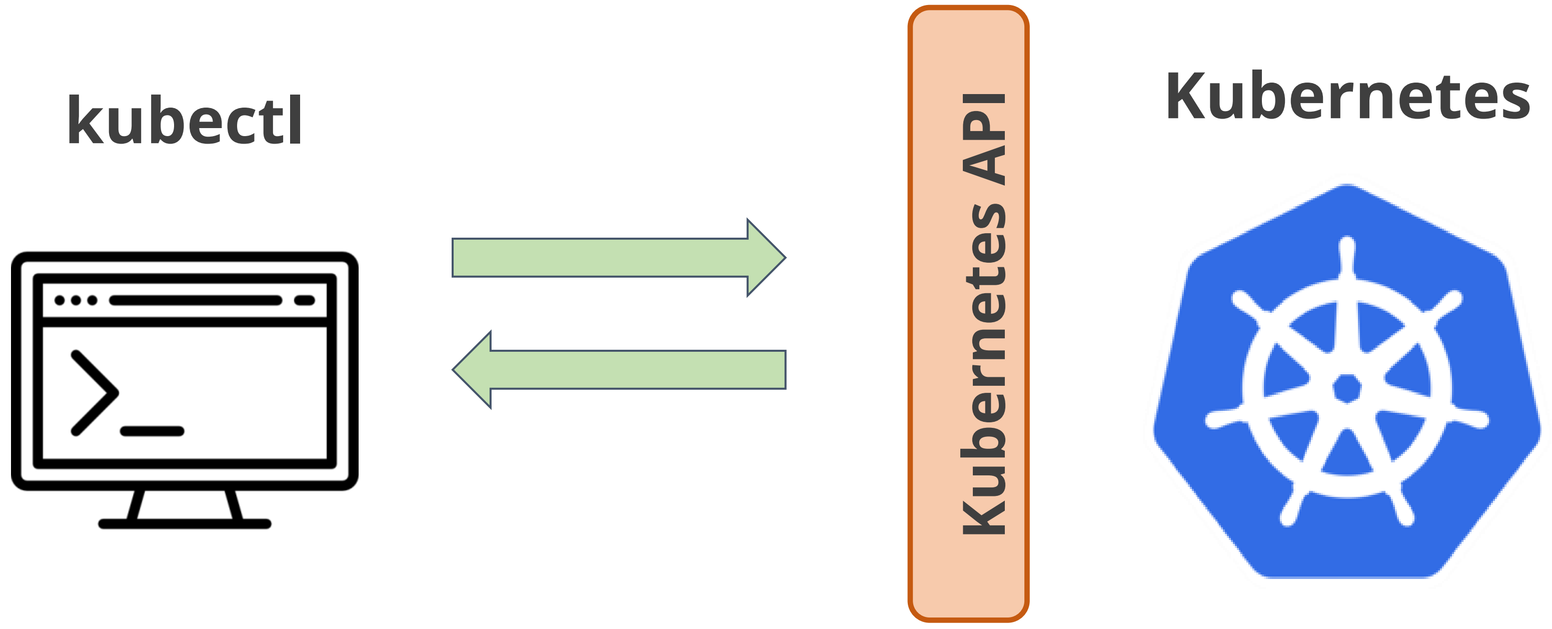
Container runtime

- The container runtime is the underlying software that runs containers on a Kubernetes cluster.
- It is responsible for fetching, starting, and stopping container images.
- Kubernetes supports several container runtimes, including but not limited to Docker, rkt, CRI-O, and any implementation of the Kubernetes CRI (Container Runtime Interface).

Interacting with a Kubernetes Cluster

Overview of kubectl

Kubernetes provides a command-line tool called *kubectl* to manage your cluster.



You use kubectl to send commands to the cluster's control plane, or fetch information about all Kubernetes objects via the API server.

Overview of kubectl

kubeconfig file

- kubectl uses a configuration file to find the information it needs to choose a cluster and communicate with the API server of a cluster.
- The config files store information about clusters, users, namespaces, and authentication mechanisms.
- You can configure kubectl to connect to multiple clusters by providing the correct context as part of the command-line syntax.

Overview of kubectl

Use the following syntax to run *kubectl* commands from your terminal window:

```
kubectl [command] [TYPE] [NAME] [flags]
```

Where the arguments are as follows:

- **command**: Refers to the operation you want to perform on one or more resources, for example *create*, *get*, *describe*, *delete*
- **TYPE**: Refers to the resource type, for example *pod*.
- **NAME**: Refers to the name of the resource
- **flags:** Refers to optional flags

Assisted Practice

Kubernetes Installation and Cluster Setup

Duration: 25 Min.

Problem Statement:

You are given a project to install Kubernetes and set up a Kubernetes cluster.

Assisted Practice: Guidelines

Steps to install and setup Kubernetes on Linux:

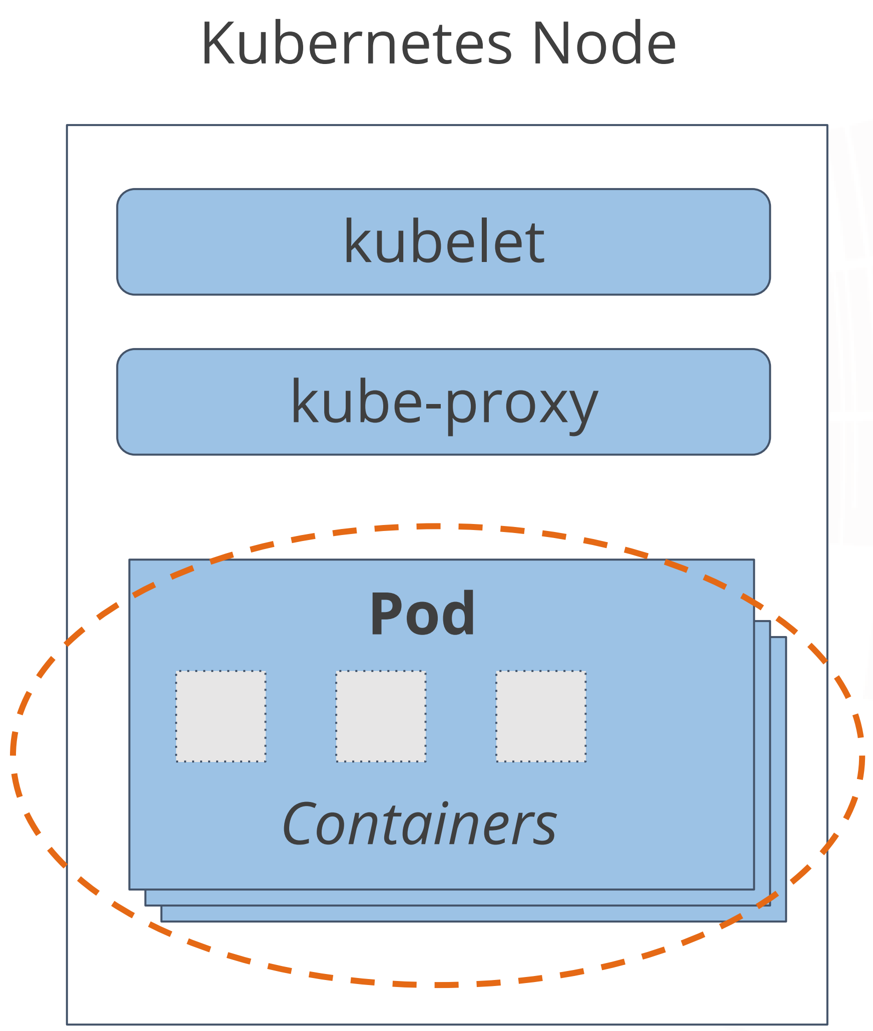
1. Install Kubernetes.
2. Set up a Kubernetes cluster using the kubeadm command.

Kubernetes Basics

Pods

Pod is a collection of one or more containers with shared storage and network resources and a specification to run its containers. It is the smallest unit of a Kubernetes application.

- Each pod comprises one or more containers that can be initialized on any host.
- Each pod is assigned a unique IP using which we can redirect traffic from outside to the pod.
- Pods are managed using the kubelet command line in a Kubernetes cluster.



Pods

- Containers in a pod can consist of multiple applications.
- Pod templates are used to define how pods will be created and deployed.
- Pods share physical resources from host machines in forms of CPU, RAM, and storage.

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: hello
spec:
  template:
    # This is the pod template
    spec:
      containers:
      - name: hello
        image: busybox
        command: ['sh', '-c', 'echo
"Hello, Kubernetes!" && sleep 3600']
      restartPolicy: OnFailure
    # The pod template ends here
```

Pod Template

Pods

Pods in a Kubernetes cluster are used in two main ways.

Single Container Pods

- Common use case
- Act as a wrapper around a single container

Multi Container Pods

- Advanced use case
- Can encapsulate an application composed of multiple co-located, tightly coupled containers

Assisted Practice

Pod Creation in Kubernetes

Duration: 20 Min.

Problem Statement:

You are given a project to create a Kubernetes pod using a yaml file.

Assisted Practice: Guidelines

Steps to create a Kubernetes cluster on Linux:

1. Create multi-container pods.
2. Create a single container pod.

Labels and Selectors

Labels

- Kubernetes attaches key-value pairs called labels for various objects such as services, pods, and nodes.
- They are intended for easy identification by users and do not have any semantic implication on the core system.
- They can be used to organize and to select subsets of objects.

Labels

```
"metadata": {
  "labels": {
    "key1" : "value1",
    "key2" : "value2"
  }
}
```

- Labels can be attached to objects at creation time and can be added or modified at any time.
- Each object can have a set of key/value labels defined.
- Each Key must be unique for a given object.

Labels

Some examples of labels are as shown below:

"release" : "stable"

"environment" : "dev"

"tier" : "frontend"

"track" : "daily"

"partition" : "customerA"

Labels and Selectors

Selectors

- Labels do not provide uniqueness, many objects can have a similar label.
- In Kubernetes, the label selector is the core grouping primitive.
- They are used by the users to select a group of objects.
- The Kubernetes API currently supports two kinds of selectors: equality based and set based.

Selectors

Equality-based selectors allow filtering by using label keys and values.

```
environment = production
tier != frontend
```

Example: Equality-based selector

- Matching objects must satisfy all the specified label constraints.
- Three types of operators are permitted namely **=, ==, !=**.

Selectors

Set-based selectors allow filtering keys according to a set of values.

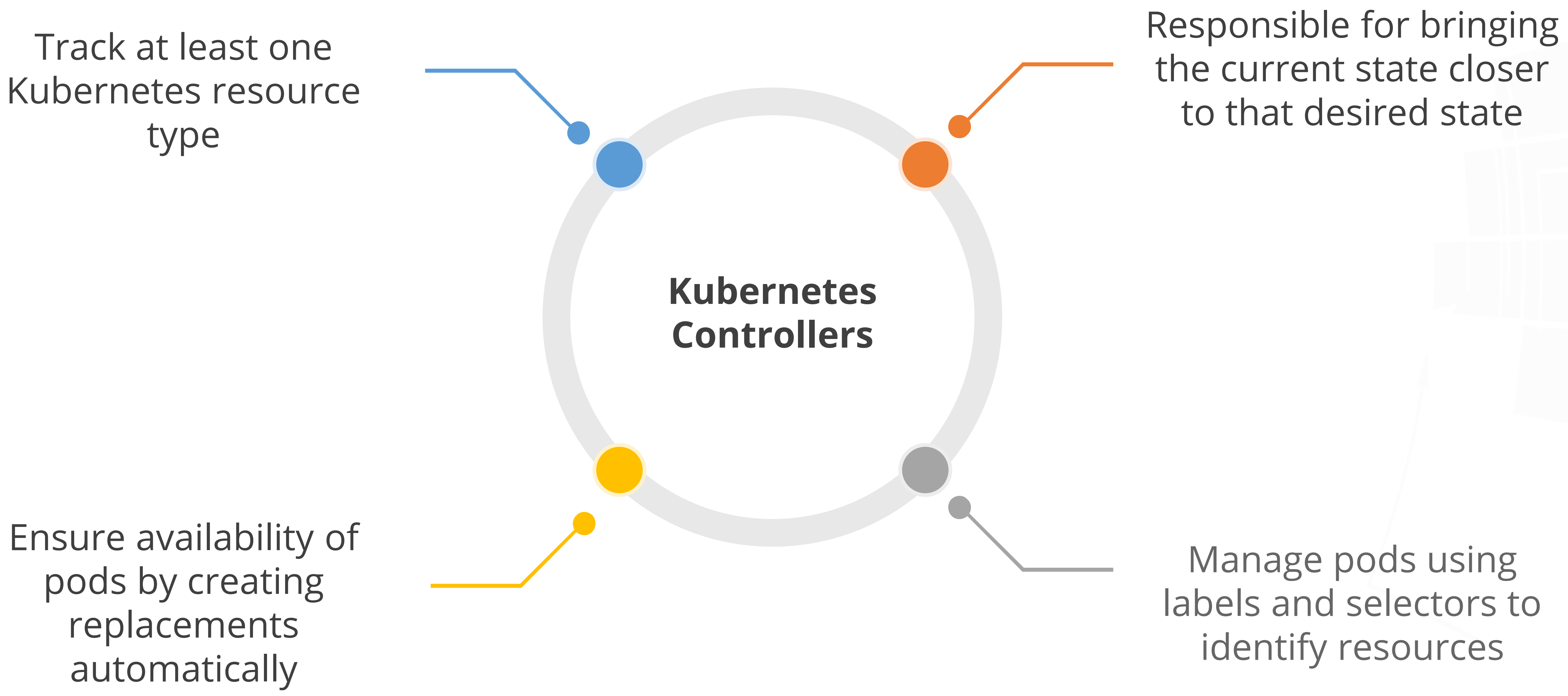
```
environment in (production, qa)
tier notin (frontend, backend)
partition
!partition
```

Example: Set-based selector

- Three types of operators are permitted namely **in**, **notin**, and **exists**.
- Set-based requirements can be mixed with equality-based requirements.

Controllers

Controllers are control loops that monitor the state of your Kubernetes cluster, and make or request changes wherever needed.


Kubernetes Controllers
Track at least one Kubernetes resource type
Responsible for bringing the current state closer to that desired state
Ensure availability of pods by creating replacements automatically
Manage pods using labels and selectors to identify resources

Controllers

The controller can directly carry out the action on its own or can control via the API server in order to bring the desired state.

Direct control

- Controllers interact with the external state, find the desired state from the API server, then communicate directly with an external system to bring the current state closer in line.
- Example: Job controller.

Control via API server

- Inbuilt controllers manage the current state by interacting with the cluster API server.
- Example: Autoscaler controller.

Controllers

Types

- ReplicationController replicates and scales pods across Kubernetes clusters.
- Controllers take care of availability of pods, and if it fails, a replacement pod gets created automatically.
- DaemonSet controller ensures only one pod runs on each node.
- Job controller manages all the batch jobs of pods which are executed in a Kubernetes cluster.

Controllers

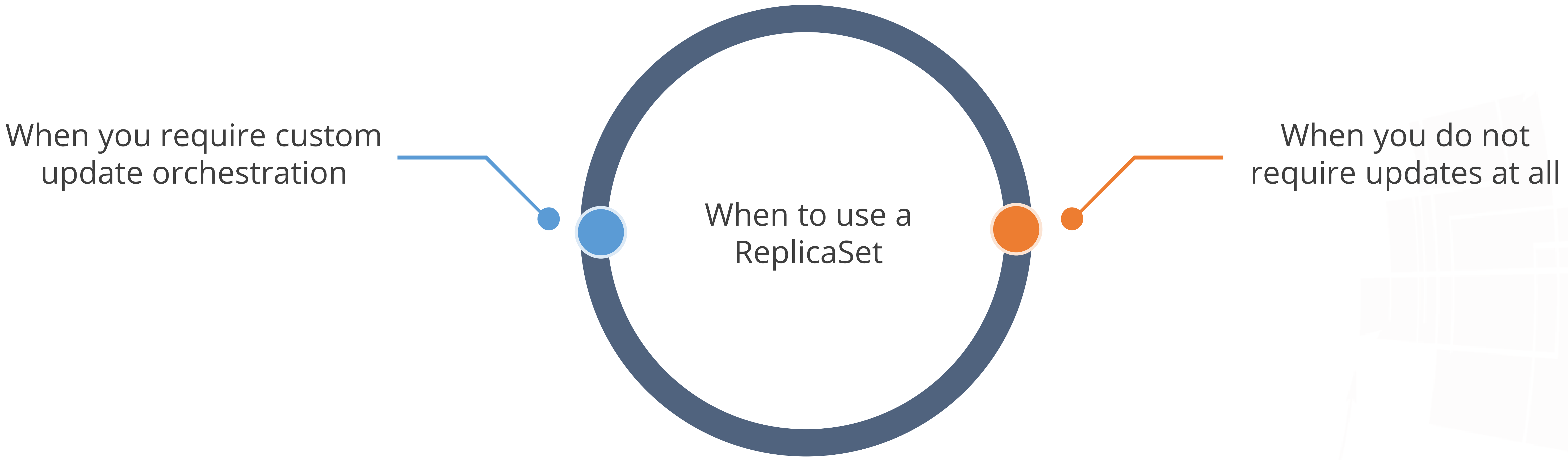
- Kubernetes has a set of built-in controllers which provide important core behaviors.
- They run inside the kube-controller-manager.
- Some examples of such controllers are deployment controller and job controller.
- There are a few controllers that run outside the control plane.
- You can also write your own controllers to extend the functionalities to suit your needs.

ReplicaSet

A ReplicaSet is used to ensure that a set of replica pods is running at any given time. It is commonly used to guarantee the availability of a specified number of identical pods.

- The ReplicaSet uses the selector to identify the pods running and based on the result, it creates or deletes the pods.
- It is similar to a ReplicationController, the main difference being, ReplicaSet uses set-based selectors, unlike the replication controller that uses the equality-based selectors.
- It acquires the pod if the pod does not have an OwnerReference and matches the selector of ReplicaSet.

ReplicaSet



When to use a ReplicaSet

When you require custom update orchestration

When you do not require updates at all

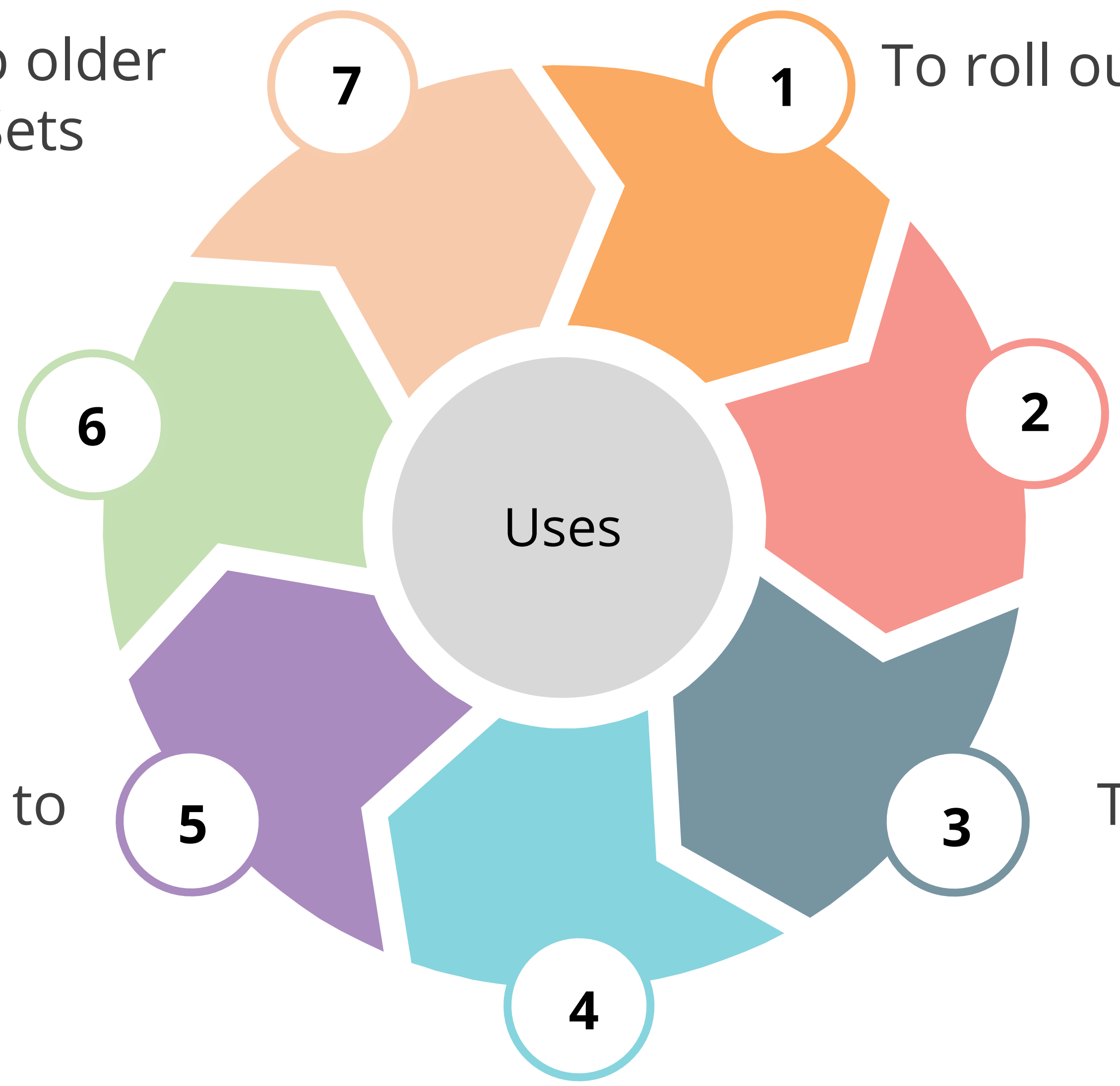
ReplicaSet

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # modify replicas according to your case
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v3
```

Example of a ReplicaSet

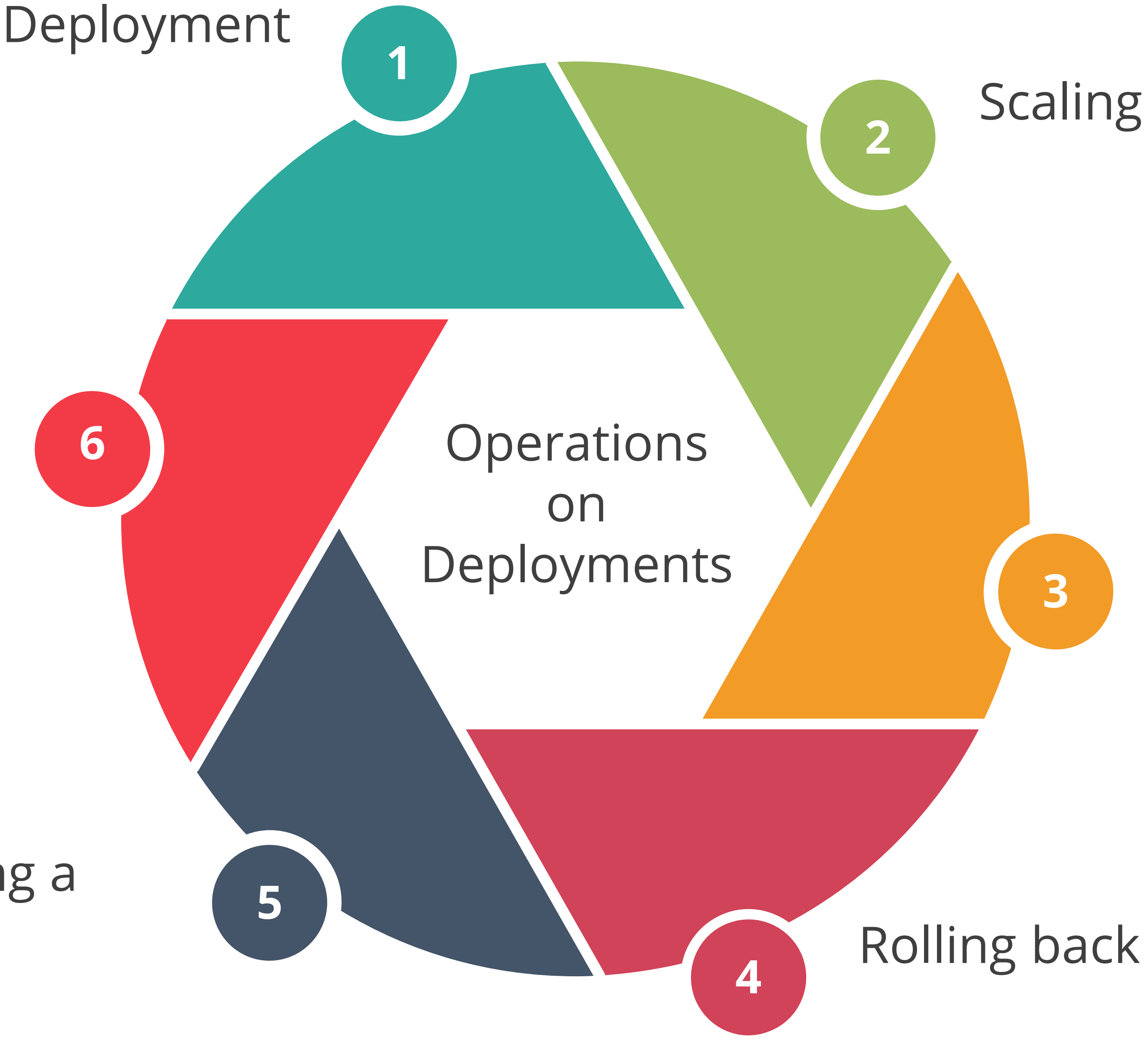
Deployments

A Deployment is used to provide updates for pods and ReplicaSets. It is a controller that changes the actual state to the desired state as specified.



1. To roll out a ReplicaSet
2. To declare the new state of pods
3. To rollback to an earlier Deployment version
4. To facilitate more load
5. To apply multiple fixes to pods
6. To indicate that a rollout is stuck
7. To clean up older ReplicaSets

Deployments



Creating a Deployment

Scaling a Deployment

Updating a Deployment

Rolling back a Deployment

Pausing and resuming a Deployment

Cleaning up a Deployment policy

Services

Service is an abstraction which defines a logical set of pods and a policy which can be used to access them.

- The set of pods targeted by a service is commonly determined by a selector.
- Kubernetes allocates a unique port and DNS to each service. The port and DNS details are changed only if the service object is recreated.
- There can be multiple replicated pods in a service.
- In case of multiple pods, an in-built load balancer is used to share the load between pods running on different nodes.

Services

Services are defined in YAML, similar to all other Kubernetes objects.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: TestApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9377
```

Example: Defining a Service

- Kubernetes assigns a service an IP address on creation, just like a node or pod.
- The example specification creates a new service object named **my-service**, which targets TCP port 9377 on any Pod with the app=TestApp label.
- The controller for the service selector continuously scans for pods that match its selector, then POSTs any updates to an endpoint object also named **my-service**.

Services

When you define a service without a pod selector, the corresponding endpoints object is not created automatically.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9377
```

Service without a Selector

```
apiVersion: v1
kind: Endpoints
metadata:
  name: my-service
subsets:
  - addresses:
      - ip: 192.0.2.42
    ports:
      - port: 9377
```

Endpoints object

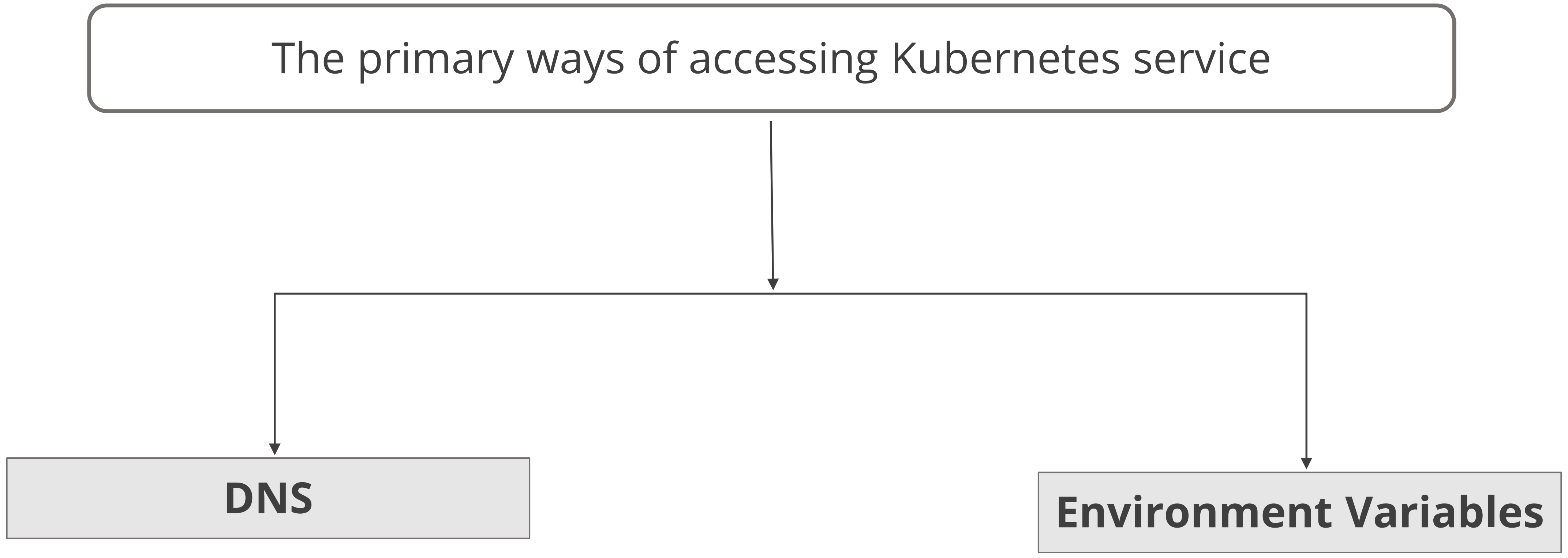
You can manually map the service to the network address and port where it's running, by adding an endpoints object manually.

How Do Kubernetes Services Work?

A Kubernetes service enables communication between nodes, pods, and users of your app, both internal and external, to the cluster.

- Services point to pods via labels.
- They are not node specific and can point to a pod irrespective of where the pod runs in the cluster at any given point in time.
- By exposing a service IP address along with the DNS service name, the application can be accessed by either method as long as the service exists.

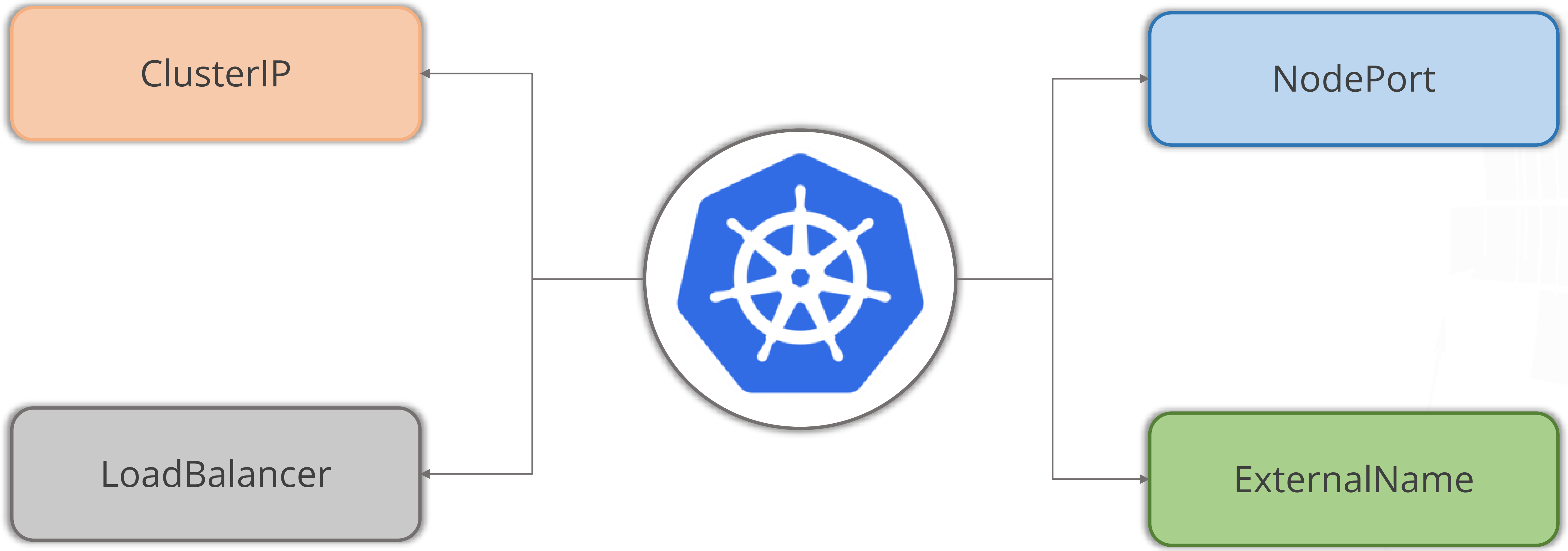
Accessing a Kubernetes Service



- The DNS server monitors the Kubernetes API for new services and creates a set of DNS records for each.
- The kubelet adds a set of environment variables for each active service for every node a pod is running on.

Types of Kubernetes Services

Kubernetes provides four types of services in order to expose a service onto an external IP address that's outside the cluster.



Types of Kubernetes Services

- **ClusterIP**: Exposes the service within the Kubernetes cluster, default ServiceType
- **NodePort**: Exposes the service via a static port on each node's IP. The NodePort service routes to a ClusterIP service that is automatically created. To access the service from outside the cluster use *NodeIP:nodePort*.
- **LoadBalancer**: Exposes the service externally via a cloud provider's load balancer. The external load balancer routes to NodePort and ClusterIP Services that are automatically created.
- **ExternalName**: Maps a service to a predefined externalName field by returning a CNAME record with its value.
- Ingress can also be used to expose the service although it's not a service type.

Kubernetes Networking

Kubernetes networking allows Kubernetes components to communicate with each other and with other applications.

It is primarily concerned with:

- Containers within a pod using networking to communicate via loopback
- Communication between different pods provided by cluster networking
- The service resource that enables you to expose an application running in pods to make it reachable from outside your cluster
- Services used to publish services only for consumption inside your cluster

Kubernetes Storage

Kubernetes storage architecture relies on volumes as a central abstraction.

- Kubernetes uses the same storage volume concept that you find when using Docker.
- The Kubernetes volume's lifetime is an explicit lifetime that matches the pod's lifetime. This means a volume outlives the containers that run in the pod. However, if the pod is removed, so is the volume.
- Volumes may be persistent or non-persistent, and Kubernetes allows containers to request storage resources dynamically, using a mechanism called volume claims.

Kubernetes Configuration

Kubernetes has two types of objects that are used to inject configuration data into a container when it starts up: Secrets and ConfigMaps.

Secrets

Used to store and manage sensitive information, such as passwords, OAuth tokens, and ssh keys

ConfigMaps

Used to store non-confidential configuration data in key-value pairs

Key Takeaways

- Container orchestration is the automation and management of lifecycle of containers and services.
- Kubernetes is an open-source orchestration tool used to manage containerized applications.
- A working Kubernetes deployment is called a cluster and it contains at least one control plane and one or more worker nodes.
- kubectl is a command-line tool provided by Kubernetes to manage your cluster.
- Kubernetes automates deployment and scales containers across multiple nodes in Kubernetes clusters.

Lesson-End Project

Deploy an App to the Kubernetes Cluster

Project Agenda: To deploy a Node.js application to the Kubernetes cluster.

Description: You have created an application and want to containerize it. For efficient load balancing and auto-scaling of the containers depending on the requirement, you decide to deploy your app on a Kubernetes cluster.

Perform the following:

- Create a Node.js application
- Create a Docker image for the application
- Create a Kubernetes deployment using a Yaml file
- Verify the deployment of the app on the Kubernetes cluster

Thank You