Kubernetes Learning [Day 01]

Courtesy : Bibin Wilson (https://www.linkedin.com/in/bibinwilson/) devopscube.com

Prerequisites to learn Kubernetes 🚀

From a DevOps engineer standpoint, I believe one should know the following before getting started with Kubernetes.

➡ Learn Container concepts & Gets Hands-on with Docker

➡ Understand what is a Distributed system. CAP theorem is good to have knowledge.

➡ Understand Authentication & Authorization: A fundamental concept in IT. However, engineers starting their IT career tend to get confused. So please get a good understanding by learning from analogies.

➡ Understand what is a Key-Value Store: It is a type of NoSQL Database. Understand just enough basics and their use cases.

➡ Learn the basics of REST API: Kubernetes is an API-driven system. So you need to have an understanding of RESTFUL APIs. Also, try to understand gRPC API. It's good to have knowledge. I would suggest creating a simple flask API and learning it practically.

➡ Learn YAML: YAML stands for YAML Ain't Markup Language. It is a data serialization language that can be used for data storage and configuration files. It's very easy to learn and from a Kubernetes standpoint, we will use it for configuration files. So understanding YAML syntax is very important.

➡ Understand Service Discovery

➡ Learn Networking Basics
- L4 & L7 Layers (OSI Layers)
- SSL/TLS
- Network Proxy Basics
- DNS concepts
- IPTables
- Software Defined Networking (Good to have knowledge)
- Network Interfaces
- CIDR
- Firewall basic concepts(inbound/outbound)

I have created a repo for the kubernetes learning roadmap with reference links to learn all the concepts listed above.

Roadmap Repo: https://lnkd.in/gAvXQYJy

Kubernetes Learning [Day 02]

Let's learn about Kubernetes Cluster Components 🚀

It is essential to understand that Kubernetes is a distributed system.

Meaning, it has multiple components spread across different servers over a network. These servers could be Virtual machines or bare metal servers.

As a whole, we call it a Kubernetes cluster.

A Kubernetes cluster consists of a control plane and worker nodes.

☑ The control plane is responsible for maintaining the desired state of the cluster. It is also responsible for node/pod lifecycle management and exposing cluster API. It has the following components.

1. kube-apiserver
2. etcd
3. kube-scheduler
4. kube-controller-manager
5. cloud-controller-manager

☑ The Worker nodes are responsible for running containerized applications. The worker Node has the following components.

1. kubelet
2. kube-proxy
3. Container runtime (CRI-O, Docker Engine, Containerd, etc)

☑ Additionally there are add-on components that you can add to the cluster to extend its functionality and make the cluster fully functional for application deployments.

Following are some of the common Kubernetes add-ons

1. Web UI
2. Core DNS
3. Metrics Server
4. CNI Plugins (Container Network Interface)

In this Kubernetes distributed scenario, the connection happening through the network to connect with different Kubernetes components should be secure and all the components should be able to authenticate each other.

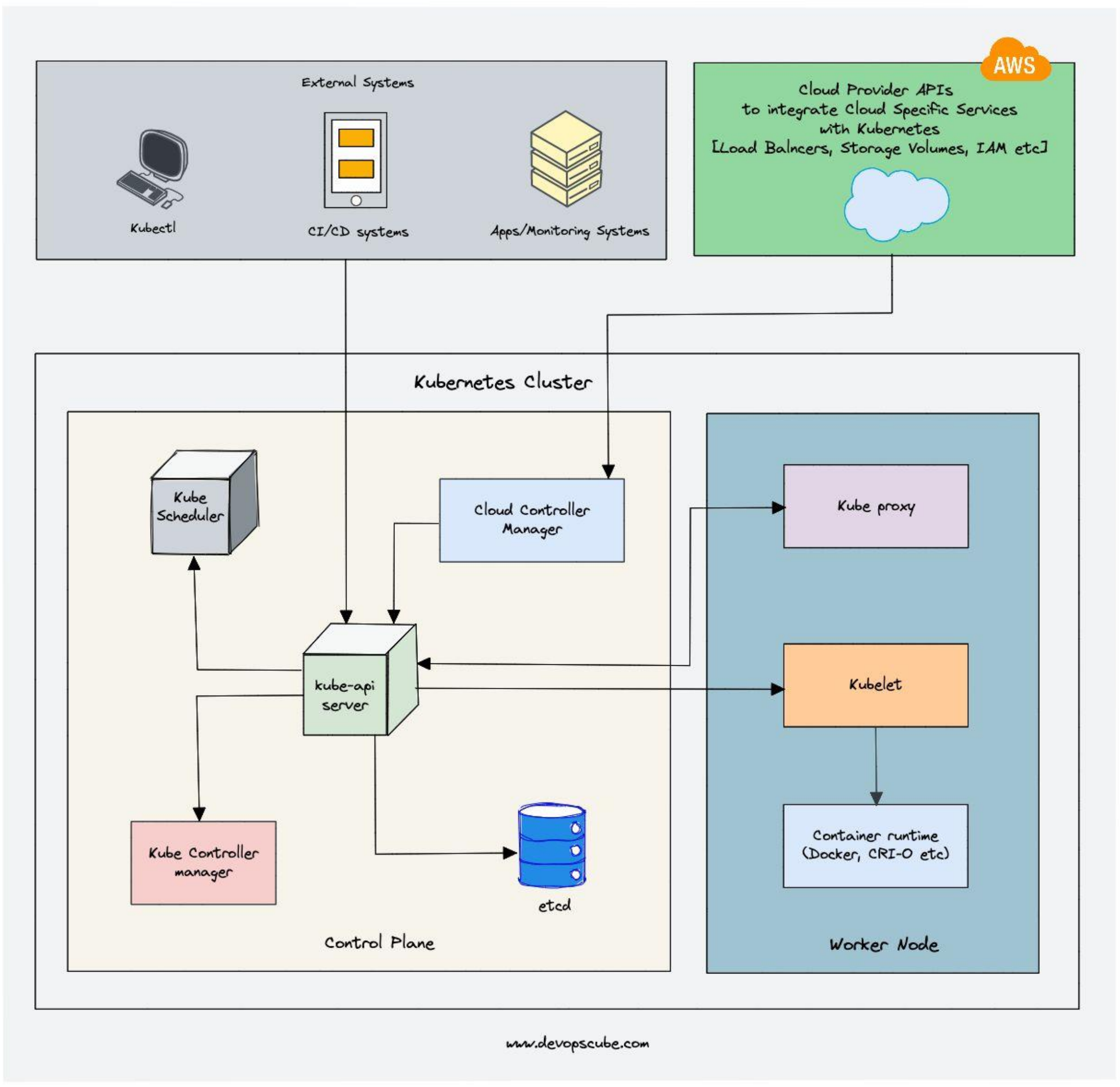
Kubernetes uses PKI certificates for authentication over TLS.

In the upcoming posts, I will cover each component and its significance in detail.

Tomorrow we will learn about kube-apiserver in detail.

Note: Open the image in a new tab for good resolution.

Check out the comments for links to my Kubernetes learning resources.



Prerequisites To learn Kubernetes: https://www.linkedin.com/posts/bibinwilson_kubernetes-learningeveryday-devops-activity-6998531491534163969-DHZv?utm_source=share&utm_medium=member_desktop

Kubernetes Learning Roadmap: https://devopscube.com/learn-kubernetes-complete-roadmap/

Kubernetes Learning Github Repo: https://github.com/techiescamp/kubernetes-learning-path

Learning Kubernetes [Day 03]

Let's learn about **kube-apiserver** 🚀

The kube-api server is the central hub of the Kubernetes cluster that exposes the Kubernetes API.

It is the main entry point to the cluster.

End users, and other cluster components, talk to the cluster via the API server.

Also, Monitoring systems and third-party services may talk to API servers to interact with the cluster.

All these systems communicate with the API server through HTTP REST APIs.

However, the internal cluster components like the scheduler, controller, etc talk to the API server using gRPC.

Kubernetes api-server is responsible for the following

☑ **API management**: Exposes the cluster API endpoint and handles all API requests.

☑ **Authentication**: Using client certificates, bearer tokens, and HTTP Basic Authentication

☑ **Authorization**: ABAC and RBAC evaluation

☑ **Validation**: Processing API requests and validating data for the API objects like pods, services, etc. (Validation and Mutation Admission controllers)

☑ It is the only component that communicates with etcd.

☑ API-server coordinates all the container orchestration processes between the control plane and worker node components.

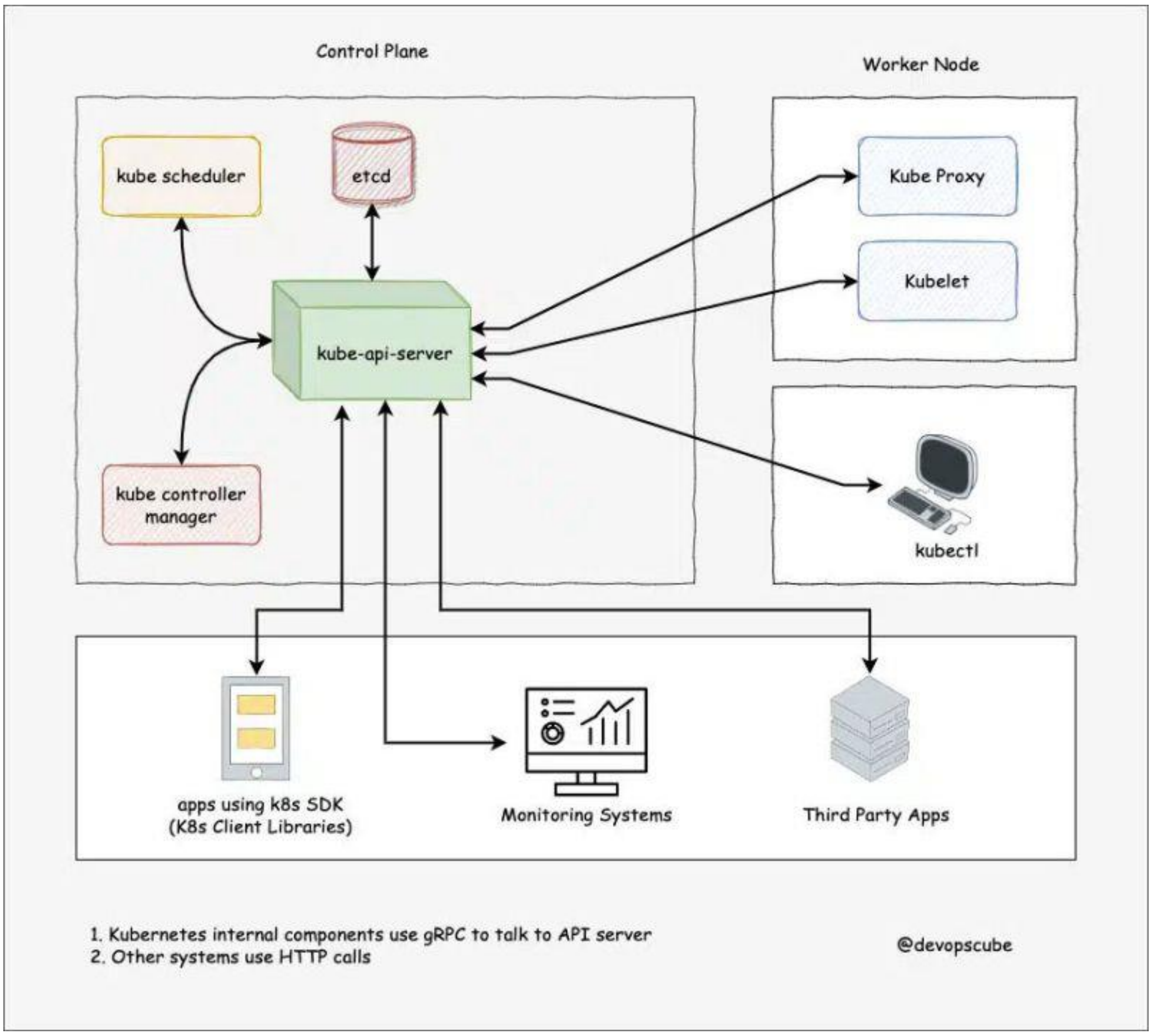
The communication between the API server and other components in the cluster happens over TLS to prevent unauthorized access to the cluster.

From a security standpoint, it is important to reduce the cluster attack surface by securing the API server.

⚠ The Shadowserver Foundation has conducted an experiment that discovered **380000 publicly accessible Kubernetes API servers**.

Note: Admission controller is an important concept you should know. I will cover it in a future post with examples.

Tomorrow we will learn about the etcd component in detail.



Learning Kubernetes [Day 04]

Let's learn about **etcd** and how Kubernetes uses it. 🚀

Kubernetes is a distributed system and it needs an efficient distributed database like etcd.

etcd acts as both a backend service discovery and a database.

You can call it the brain of the Kubernetes cluster.

etcd is an open-source strongly consistent, distributed key-value store. So what does it mean?

☑ **Strongly consistent**: If an update is made to a node, strong consistency will ensure it gets updated to all the other nodes in the cluster immediately. Also if you look at the CAP theorem, achieving 100% availability with strong consistency & Partition Tolerance is impossible.

☑ **Distributed**: etcd is designed to run on multiple nodes as a cluster without sacrificing consistency.

☑ **Key Value Store**: A nonrelational database that stores data as keys and values. It also exposes a key-value API. The datastore is built on top of BboltDB which is a fork of BoltDB.

etcd uses **raft consensus algorithm** for strong consistency and availability. It works in a leader-member fashion for high availability and to withstand node failures.

You should have a minimum of 3 etcd nodes to withstand 1 failure. To withstand 2 node failures, you should have 5 nodes, and so on.

So how etcd works with Kubernetes?

To put it simply, when you use kubectl to get Kubernetes object details, you are getting it from etcd.

Also, when you deploy an object like a pod, an entry gets created in etcd.

In a nutshell, here is what you need to know about etcd.

☑ etcd stores all configurations, states, and metadata of Kubernetes objects (pods, secrets, daemonsets, deployments, configmaps, statefulsets, etc).

☑ Kubernetes uses the etcd's watch functionality to track the change in the state of an object (Watch() API)

☑ etcd exposes key-value API using gRPC. Also, the gRPC gateway which is a RESTful proxy that translates all the HTTP API calls into gRPC messages. It makes it an ideal database for Kubernetes.

☑️ etcd stores all objects under the /registry directory key in key-value format. For example, information on a pod named Nginx in the default namespace can be found under /registry/pods/default/nginx

Also, etcd it is the only **Statefulset** component in the control plane.

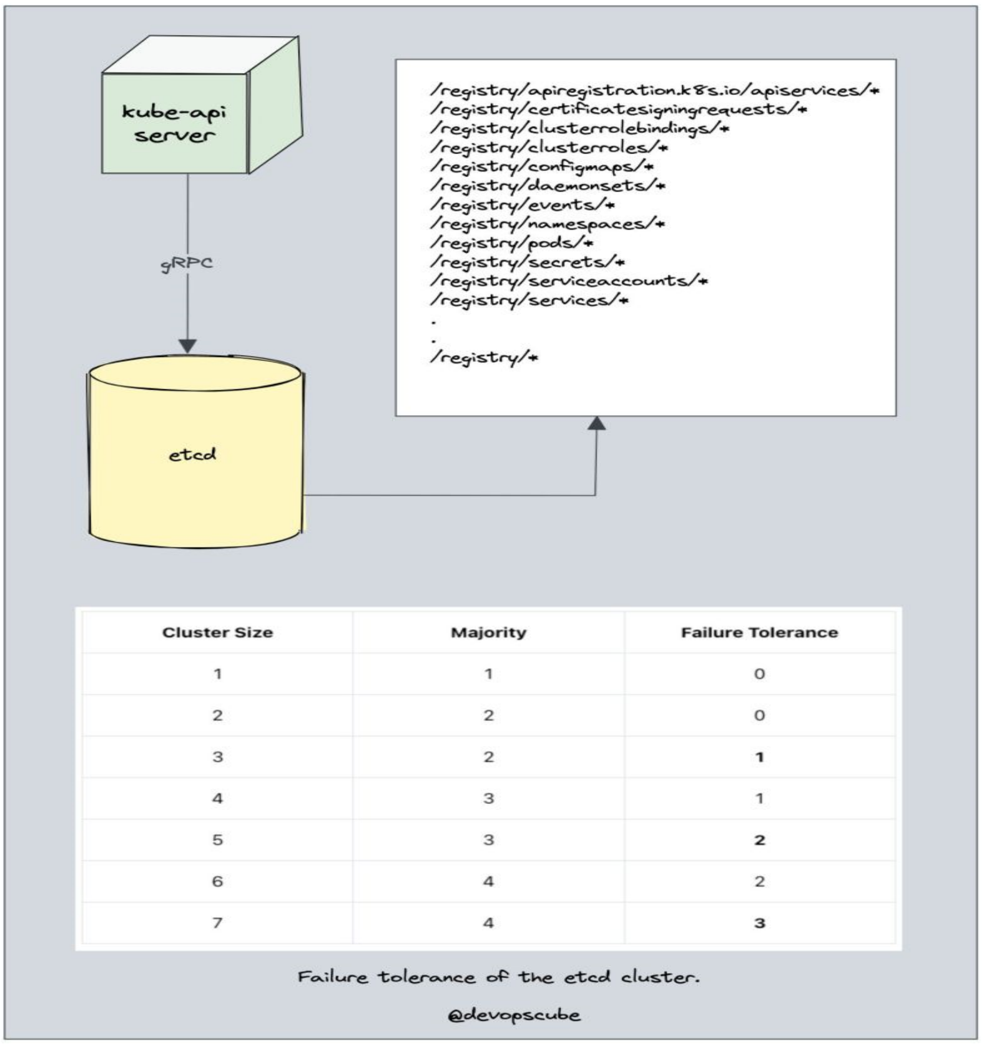
When it comes to etcd HA architecture, there are two modes.

➡️ **Stacked etcd**: etcd deployed along with control plane nodes

➡️ **External etcd cluster**: Dedicated etcd cluster.

Tomorrow we will learn about Kube scheduler component in detail.

My previous Kubernetes posts https://lnkd.in/gyvQrrAs



```
/registry/apiregistration.k8s.io/apiservices/*
/registry/certificatesigningrequests/*
/registry/clusterrolebindings/*
/registry/clusterroles/*
/registry/configmaps/*
/registry/daemonsets/*
/registry/events/*
/registry/namespaces/*
/registry/pods/*
/registry/secrets/*
/registry/serviceaccounts/*
/registry/services/*
.
.
/registry/*
```

Cluster Size	Majority	Failure Tolerance
1	1	0
2	2	0
3	2	**1**
4	3	1
5	3	**2**
6	4	2
7	4	**3**

Failure tolerance of the etcd cluster.

@devopscube

Learning Kubernetes [Day 05]

Let's learn about the **Kube Scheduler** component 🚀

The kube-scheduler is responsible for scheduling pods on worker nodes.

When you deploy a pod, you specify the pod requirements such as CPU, memory, Volume mounts, etc.

The scheduler’s primary task is to identify the create request and choose the best node for a pod that satisfies the requirements.

So how does the scheduler select the node out of all worker nodes?

Here is how 👇

➡ To choose the best node, the Kube-scheduler uses **filtering and scoring** operations.

➡ In **filtering**, the scheduler finds the best-suited nodes where the pod can be scheduled. For example, if there are five worker nodes with resource availability to run the pod, it selects all five nodes.

➡ If there are no nodes, then the pod is unschedulable and moved to the scheduling queue. If It is a large cluster, let’s say 100 worker nodes, and the scheduler doesn’t iterate over all the nodes.

➡ There is a scheduler configuration parameter called **percentageOfNodesToScore**. The default value is typically **50**%. So it tries to iterate over 50% of nodes in a round-robin fashion. If the worker nodes are spread across multiple zones, then the scheduler iterates over nodes in different zones. For very large clusters the default **percentageOfNodesToScore** is 5%.

➡ In the **scoring phase**, the scheduler ranks the nodes by assigning a score to the filtered worker nodes. The scheduler makes the scoring by calling multiple scheduling plugins.

➡ Finally, the worker node with the highest rank will be selected for scheduling the pod. If all the nodes have the same rank, a node will be selected at random.

➡ Once the node is selected, the scheduler creates a binding event in the API server. Meaning an event to bind a pod and node.

Here is what you should know about a scheduler.

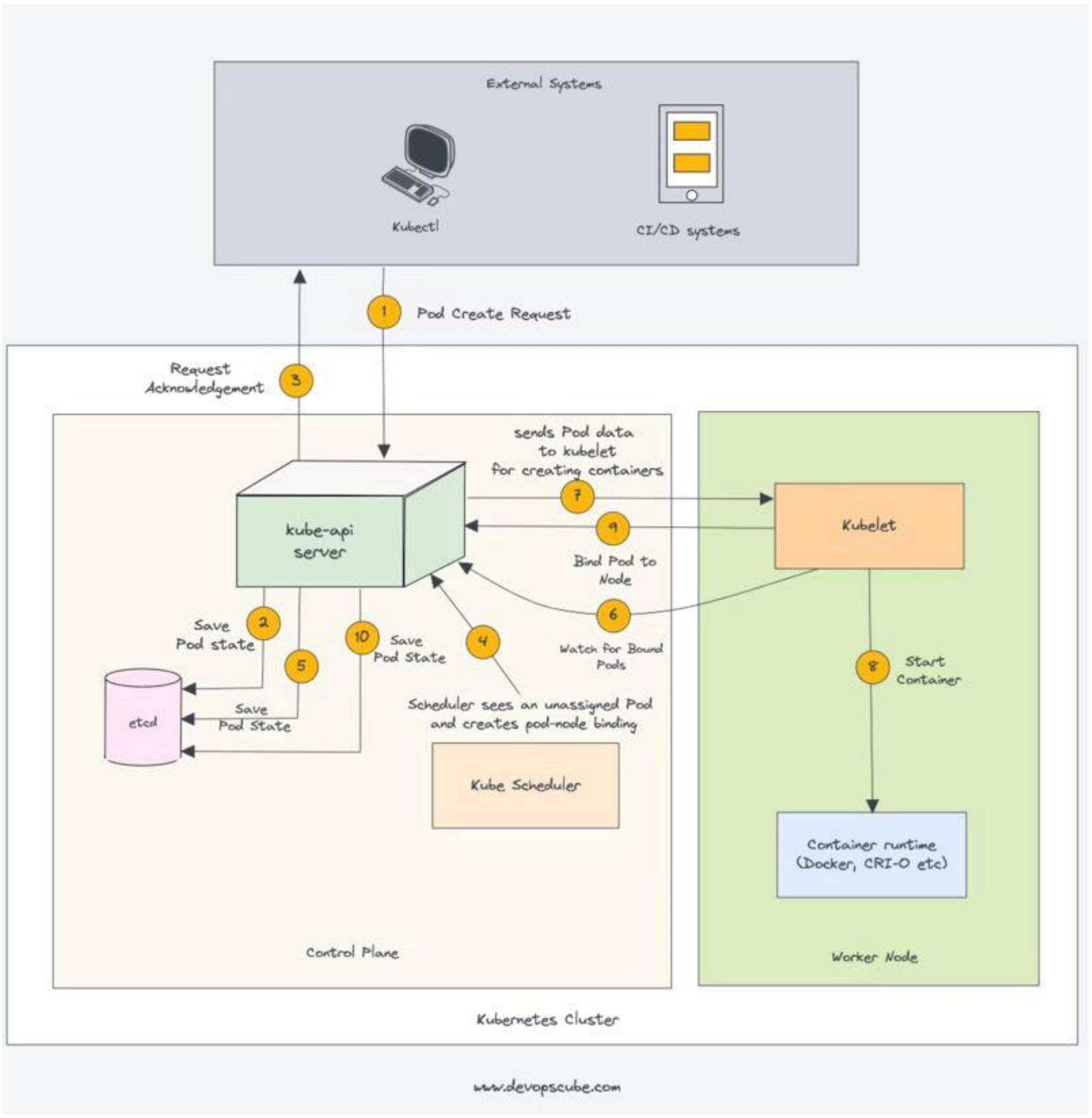
☑ It is a controller that listens to pod creation events in the API server.

☑ The scheduler has two phases. **Scheduling cycle** and the **Binding cycle**. Together it is called the scheduling context. The scheduling cycle selects a worker node and the binding cycle applies that change to the cluster.

☑ The scheduler always places the high-priority pods ahead of the low-priority pods for scheduling. Also, in some cases, after the pod started running in the selected node, the pod might get evicted or moved to other nodes.

☑ You can create custom schedulers and run multiple schedulers in a cluster along with the native scheduler. When you deploy a pod you can specify the custom scheduler in the pod manifest. So the scheduling decisions will be taken based on the custom scheduler logic.

Tomorrow we will learn about **Kube Controller Manager** component in detail.



Learning Kubernetes [Day 06]

Let's learn about **Kube Controller Manager** 🚀

So what is a controller?

Controllers are programs that run infinite control loops.

Meaning it runs continuously and watches the actual and desired state of objects via the API server.

Controllers use the Kubernetes API to watch for changes in the state of the cluster and take appropriate action when a change is detected.

If there is a difference in the actual and desired state, it ensures that the kubernetes resource/object is in the desired state.

📌 Let's look at an example.

If you want to create a deployment, you specify the desired state in the manifest YAML file (declarative approach).

For example, 2 replicas, one volume mount, configmap, etc. The in-built deployment controller ensures that the deployment is in the desired state all the time.

If a user updates the deployment with 5 replicas, the deployment controller recognizes it and ensures the desired state is 5 replicas.

Kube Controller Manager is a component that manages all the Kubernetes controllers. Kubernetes resources/objects like pods, namespaces, jobs, replicaset are managed by respective controllers.

Following are some examples of core built-in Kubernetes controllers.

➡️ Deployment controller

➡️ Replicaset controller

➡️ DaemonSet controller

➡️ Job Controller

➡️ Node controller

Here is what you should know about the Kube controller manager.

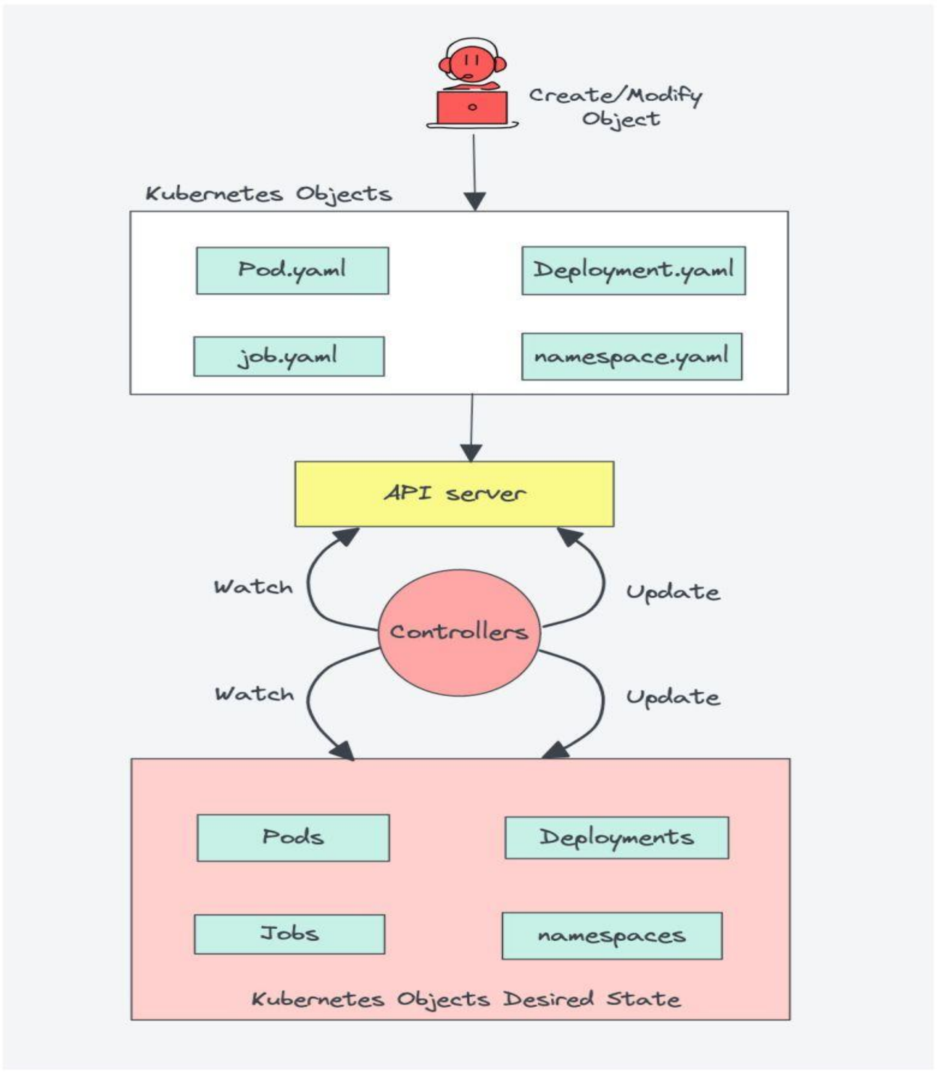
☑ It manages all the controllers and the controllers try to keep the cluster in the desired state.

☑ You can extend kubernetes with custom controllers associated with a custom resource definition.

☑ Controller uses the Kubernetes API to get the current state of the cluster and to update the state of the cluster by creating, updating, or deleting resources.

☑ If there are multiple instances of Kube controller manager (Control Plane HA), it runs in a leader election mode. One instance is elected as a leader for actively making changes to the cluster at a time. This prevents conflicts and ensures that the cluster remains in a consistent state.

In the next post, we will learn about the **Cloud Controller Manager** component in detail.



earning Kubernetes [Day 07]

Let's learn about **Cloud Controller Manager** 🚀

When Kubernetes is deployed in cloud environments (AWS, GCP, Azure, etc),

The cloud controller manager acts as a bridge between Cloud Platform APIs and the Kubernetes cluster.

This way the core Kubernetes core components can work independently and allows the cloud providers to integrate with Kubernetes using plugins. (For example, an interface between Kubernetes cluster and AWS cloud API)

Cloud controller integration allows Kubernetes cluster to manage cloud resources like servers, Load Balancers (for services), and Storage Volumes (for persistent volumes).

Cloud Controller Manager contains a set of **cloud platform-specific controllers** that ensure the desired state of cloud-specific components (nodes, Loadbalancers, storage, etc).

Controllers are programs that run infinite control loops. Meaning it runs continuously and watches the actual and desired state of objects.

There are three main controllers that are part of the cloud controller manager.

➡️ **Node controller**: This controller updates node-related information by talking to the cloud provider API. For example, node labeling & annotation, getting hostname, CPU & memory availability, nodes health, etc.

➡️ **Route controller**: It is responsible for configuring networking routes on a cloud platform. So that pods in different nodes can talk to each other.

➡️ **Service controller**: It takes care of deploying Load balancers for Kubernetes services, assigning IP addresses, etc.

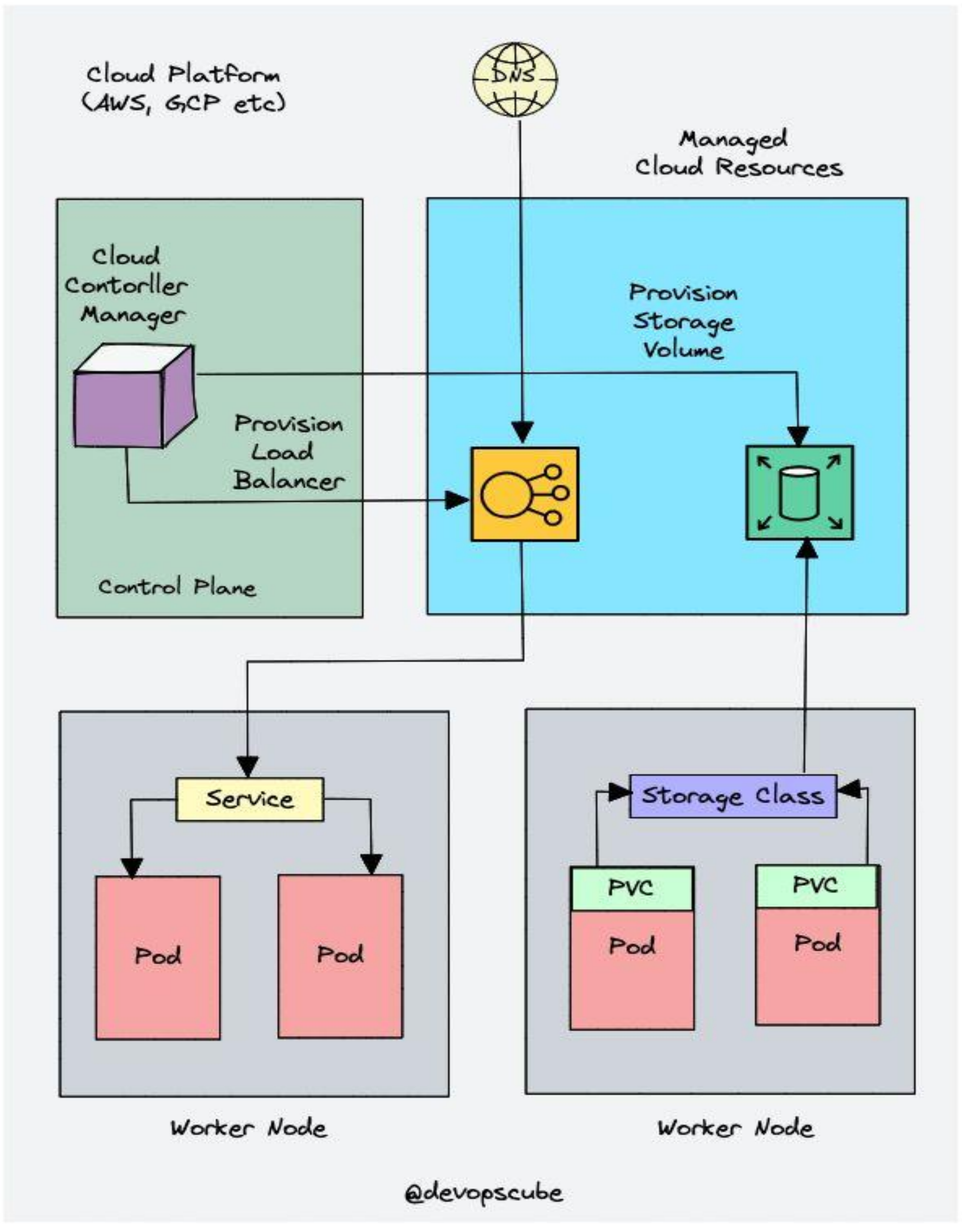
Following are some of the classic examples of cloud controller manager integrations.

☑️ Deploying Kubernetes Service of type Load balancer. Here Kubernetes provisions a cloud-specific Loadbalancer and integrates with Kubernetes Service.

☑️ Provisioning storage volumes (PV) for pods backed by cloud storage solutions.

Overall Cloud Controller Manager manages the lifecycle of cloud-specific resources used by Kubernetes.

Tomorrow we will learn about Kubelet component in detail.


Cloud Platform (AWS, GCP etc)
DNS
Managed Cloud Resources
Cloud Contorller Manager
Provision Storage Volume
Provision Load Balancer
Control Plane
Service
Storage Class
PVC
PVC
Pod
Pod
Pod
Pod
Worker Node
Worker Node
@devopscube

Learning Kubernetes [Day 08]

Let's learn the important concepts of **Kubelet** 🚀

Kubelet is an agent component that runs on every node in the cluster.

It does not run as a container instead runs as a daemon, managed by systemd.

It is responsible for registering worker nodes with the API server and working with the podSpec (Pod specification – YAML or JSON) it receives from the API server.

podSpec defines the containers that should run inside the pod, their resources (e.g. CPU and memory limits), and other settings such as environment variables, volumes, and labels.

It then brings the podSpec to the desired state by creating containers interacting with container runtime.

To put it simply, kubelet is responsible for the following.

☑ Creating, modifying, and deleting containers for the pod.

☑ Runs the startup, liveness, and readiness probes

☑ Mounting volumes

☑ Collecting and reporting Node and pod status via calls to the API server.

Kubelet is also a controller where it watches for pod changes and utilizes the node’s container runtime to pull images, run containers, etc.

Other than PodSepcs from the API server, Kubelet can accept podSpec from a file, HTTP endpoint, and HTTP server.

A good example of “podSpec from a file” is **Kubernetes static pods**.

Static pods are pods controlled by Kubelet on its nodes, not the API servers.

This means you can create pods by providing a pod YAML location to the Kubelet component. However, static pods created by Kubelet are not managed by the API server.

Here is a real-world example use case of the static pod.

While bootstrapping the control plane, kubelet starts the api-server, scheduler, and controller manager as static pods from podSpecs located at **/etc/kubernetes/manifests**.

Following are some of the key things you should know about Kubelet.

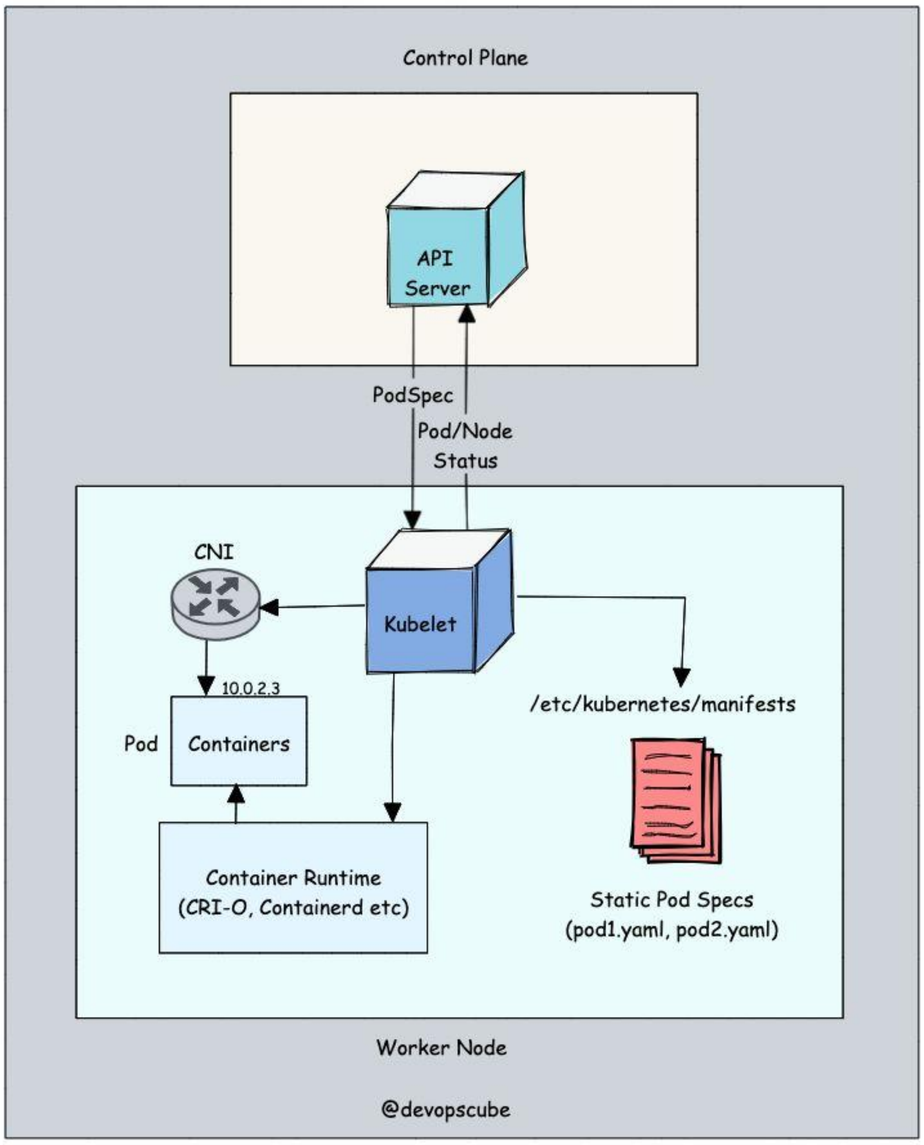
➡ Kubelet uses the CRI (container runtime interface) gRPC interface to talk to the container runtime for managing the lifecycle of the workloads.

➡ It also exposes an HTTP endpoint to stream logs and provides exec sessions for clients.

➡ Uses the CSI (container storage interface) gRPC to configure block volumes.

➡ It uses the CNI plugin configured in the cluster to allocate the pod IP address and set up any necessary network routes and firewall rules for the pod.

Tomorrow we will learn about the Kube Proxy component in detail.


Control Plane
API Server
PodSpec
Pod/Node Status
CNI
Kubelet
10.0.2.3
Pod
Containers
/etc/kubernetes/manifests
Container Runtime (CRI-O, Containerd etc)
Static Pod Specs (pod1.yaml, pod2.yaml)
Worker Node
@devopscube

Learning Kubernetes [Day 09]

Let's learn about **Kube Proxy** Component 🚀

To understand kube proxy, you need to have a basic knowledge of Kubernetes Service & endpoint objects.

☑️ Service in Kubernetes is a way to expose a set of pods internally or to external traffic. When you create the service object, it gets a **virtual IP** assigned to it, called a **ClusterIP**. It is only accessible within the Kubernetes cluster.

☑️ The Endpoint object contains all the IP addresses and ports of pods grouped under a Service object.

☑️ The endpoints controller is responsible for maintaining the list of pod IP addresses (endpoints). The service controller is responsible for configuring endpoints to a service.

☑️ You cannot ping the clusterIP because it is only used for service discovery, unlike pod IPs which are pingable.

Now, let's look into Kube Proxy.

Kube-proxy is a daemon that runs on every node as a daemonset.

It is a proxy component that implements the Kubernetes Services concept for pods (single DNS for a set of pods with load balancing).

When you expose pods using a Service (ClusterIP), Kube-proxy creates network rules to send traffic to the backend pods (endpoints) grouped under the Service object.

Meaning, all the load balancing, and service discovery are handled by the Kube proxy.

So, **how does Kube-proxy work**?

Kube proxy talks to the API server to get the details about the Service (ClusterIP) and respective pod IPs & ports (endpoints). It also monitors for changes in service and endpoints.

Kube-proxy then uses any one of the following modes to create/update rules for routing traffic to pods behind a Service:

➡️ **IPTables**: It is the default mode. In IPTables mode, the traffic is handled by IPtable rules. In this mode, kube-proxy chooses the backend pod randomly for load balancing. Once the connection is established, the requests go to the same pod until the connection is terminated.

➡️ **IPVS**: For clusters with services exceeding 1000, IPVS offers performance improvement. It supports the following load-balancing algorithms for the backend:

- rr: round-robin : It is the default mode.
- lc: least connection (smallest number of open connections)
- dh: destination hashing
- sh: source hashing
- sed: shortest expected delay
- nq: never queue

➡ **Userspace**: (legacy & not recommended)

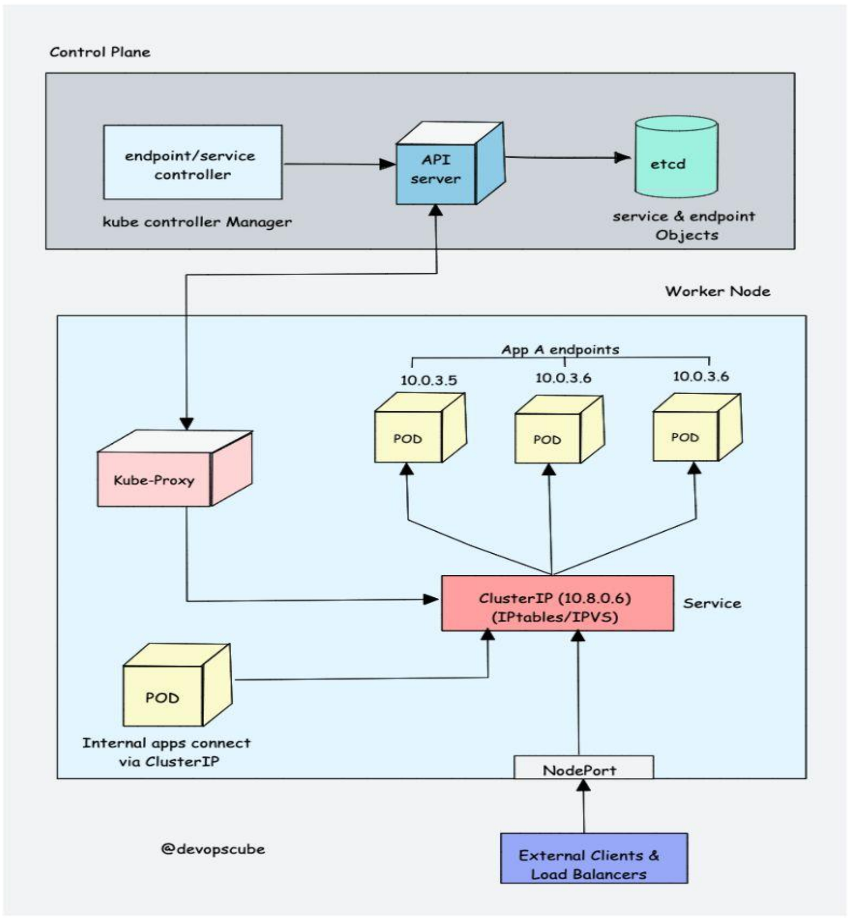
➡ **Kernelspace**: This mode is only for Windows systems.

If you would like to understand the performance difference between kube-proxy IPtables and IPVS mode, check out the link in the comment.

Also, you can run a Kubernetes cluster without kube-proxy by replacing it with Cilium.

Note: There are service types that do not have a clusterIP. We will look at service objects and networking in detail in future posts.

Tomorrow we will learn about **Container Runtime** in detail.


Control Plane
endpoint/service controller
kube controller Manager
API server
etcd
service & endpoint Objects
Worker Node
App A endpoints
10.0.3.5
10.0.3.6
10.0.3.6
POD
POD
POD
Kube-Proxy
ClusterIP (10.8.0.6) (IPtables/IPVS)
Service
POD
Internal apps connect via ClusterIP
NodePort
@devopscube
External Clients & Load Balancers

Learning Kubernetes [Day 10]

Let's learn about **Container Runtime** 🚀

A container runtime is a software component that runs on all the nodes in a Kubernetes cluster.

It is responsible for pulling images from container registries, running containers, allocating and isolating resources for containers, and managing the entire lifecycle of a container on a host.

To understand this better, let's take a look at two key concepts:

☑️ **Container Runtime Interface** (**CRI**): It is a set of APIs that allows Kubernetes to interact with different container runtimes. It allows different container runtimes to be used interchangeably with Kubernetes. The CRI defines the API for creating, starting, stopping, and deleting containers, as well as for managing images and container networks.

☑️ **Open Container Initiative** (**OCI**): It is a set of standards for container formats and runtimes

Kubernetes supports multiple container runtimes (such as CRI-O, Docker Engine, and containerd) that are compliant with the CRI.

This means that all these container runtimes implement the CRI interface and expose gRPC CRI APIs (runtime and image service endpoints).

So how does Kubernetes make use of the container runtime?

As we learned in the Kubelet section, the kubelet agent is responsible for interacting with the container runtime using CRI APIs to manage the lifecycle of a container. It also gets all the container information from the container runtime and provides it to the control plane.

Let’s take an example of CRI-O container runtime interface.

Here is a high-level overview of CRI-O it works with Kubernetes.

➡️ When there is a new request for a pod from the API server, the kubelet talks to CRI-O daemon to launch the required containers via Kubernetes Container Runtime Interface.

➡️ CRI-O then checks and pulls the required container image from the configured container registry using containers/image library.

➡️ It then generates OCI runtime specification (JSON) for a container.

➡️ CRI-O then launches an OCI-compatible runtime (runc) to start the container process as per the runtime specification.

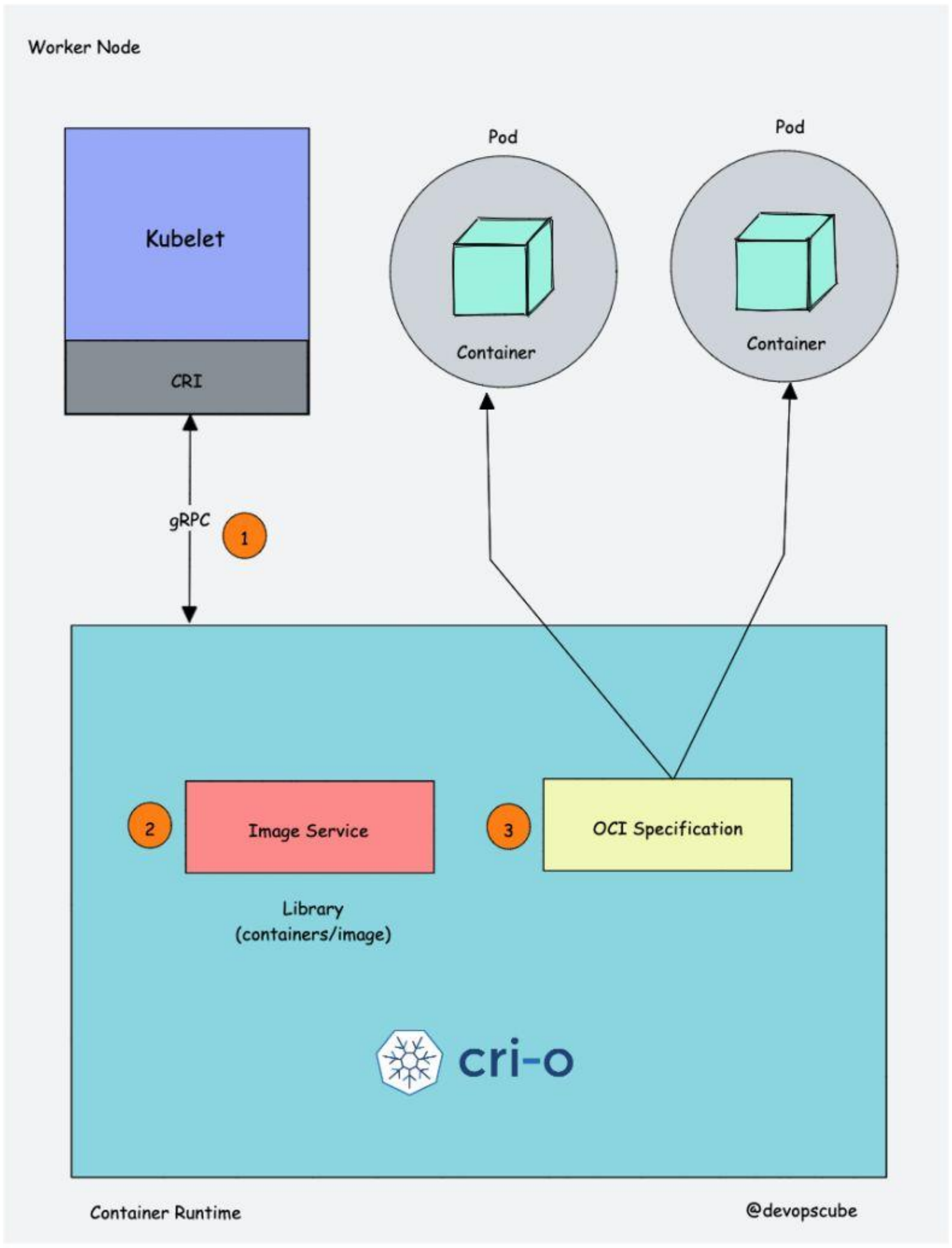
Just a quick note on some popular container runtimes:

👉 **containerd**: - It is a CNCF project, originally created by Docker.

👉 **CRI-O**: It is a CNCF Incubating project created by Redhat

👉 **runc**: It is a low-level container runtime that was originally developed by Docker.

Tomorrow we will learn about the Addon component **CNI plugins** in detail.



Learning Kubernetes [Day 11]

Let's learn about **CNI plugins** and why we need them.

First, you need to understand the **Container Networking Interface** (**CNI**).

It is a plugin-based architecture with vendor-neutral specifications and libraries for creating network interfaces for containers.

It is not specific to Kubernetes. With CNI, container networking is standardized across container orchestration tools such as Kubernetes, Mesos, CloudFoundry, Podman, and Docker.

When it comes to container networking, companies might have different requirements such as network isolation, security, and encryption.

As container technology advanced, many network providers created CNI-based solutions for containers with a wide range of networking capabilities.

These solutions can be referred to as CNI plugins. This allows users to choose a networking solution that best fits their needs from different providers.

How does the CNI plugin work with Kubernetes?

➡ The kube-controller-manager is responsible for assigning a pod CIDR to each node. Each pod gets a unique IP address from the pod CIDR.

➡ The kubelet interacts with the container runtime to launch the scheduled pod.

➡ The CRI (Container Runtime Interface) plugin, which is part of the container runtime, interacts with the CNI plugin to configure the pod network.

➡The CNI plugin then selects an available IP address from the configured pod CIDR and assigns it to the pod.

➡ Once the container is started, the pod will be able to connect to pods spread across the same or different nodes using an overlay network.

The following are high-level functionalities provided by CNI plugins:

☑ Pod networking

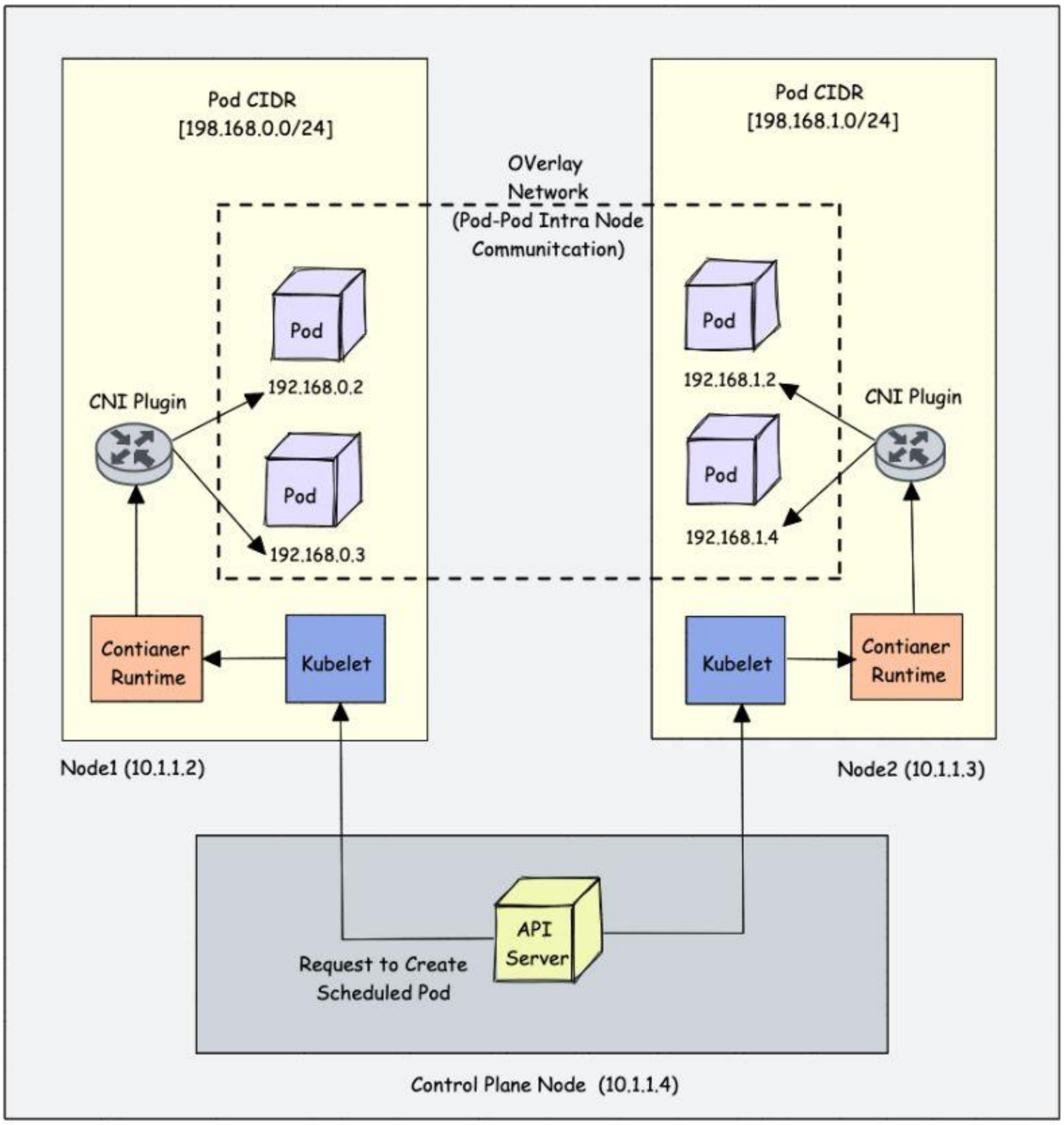
☑ Pod network security and isolation using Network Policies to control the traffic flow between pods and between namespaces.

Some popular CNI plugins include:

☞ Calico
☞ Flannel
☞ Weave Net
☞ Cilium (Uses eBPF)
☞ Amazon VPC CNI (For AWS VPC)
☞ Azure CNI (For Azure Virtual network)

Kubernetes networking is a big topic and it differs based on underlying network infrastructure and requirements.

Cloud platforms offer cloud-specific networking functionalities and options to use other Network Provider CNI plugins. We will look more into networking in hands-on tutorials.



Kubernetes [Day 12]

Let's learn about **Kubernetes Cluster High Availability** 🚀

Kubernetes is a distributed system and it is subject to multiple faults.

By implementing HA in Kubernetes, the risk of downtime is reduced, applications and services that run on the cluster remain available and accessible to users and the system can quickly recover from failures without human intervention.

At a high level, this can be achieved by deploying multiple replicas of control plane components with a network topology spanning across multiple availability zones or regions.

Running a single-node control plane could lead to a single point of failure of all the control plane components.

To have a highly available Kubernetes control plane, you should have a minimum of three quorum control plane nodes with control plane components replicated across all three nodes.

It's very important for you to understand the nature of each control plane component when deployed as multiple copies across nodes.

➡️ **etcd**: When it comes to etcd HA architecture, there are two modes.

1. Stacked etcd:- etcd deployed along with control plane nodes
2. External etcd cluster:- Etcd cluster running externally on dedicated nodes.

To have fault tolerance you should have a minimum of three node etcd cluster.

➡️ **API Server**: The API server is a stateless application that primarily interacts with the etcd cluster to store and retrieve data. This means that multiple instances of the API server can be run across different control plane nodes.

To ensure that the cluster API is always available, a Load Balancer should be placed in front of the API server replicas. This Load Balancer endpoint is used by worker nodes, end-users, and external systems to interact with the cluster.

➡️ **Kube Scheduler**, **Controller Manager and Cloud Controller Manager**:

All these components run in a leader-follower fashion.

So when you run multiple replicas of these components one instance will be elected as a leader and others will be marked as followers.

This ensures that there is always a single instance of making decisions and avoiding conflicts and inconsistencies.

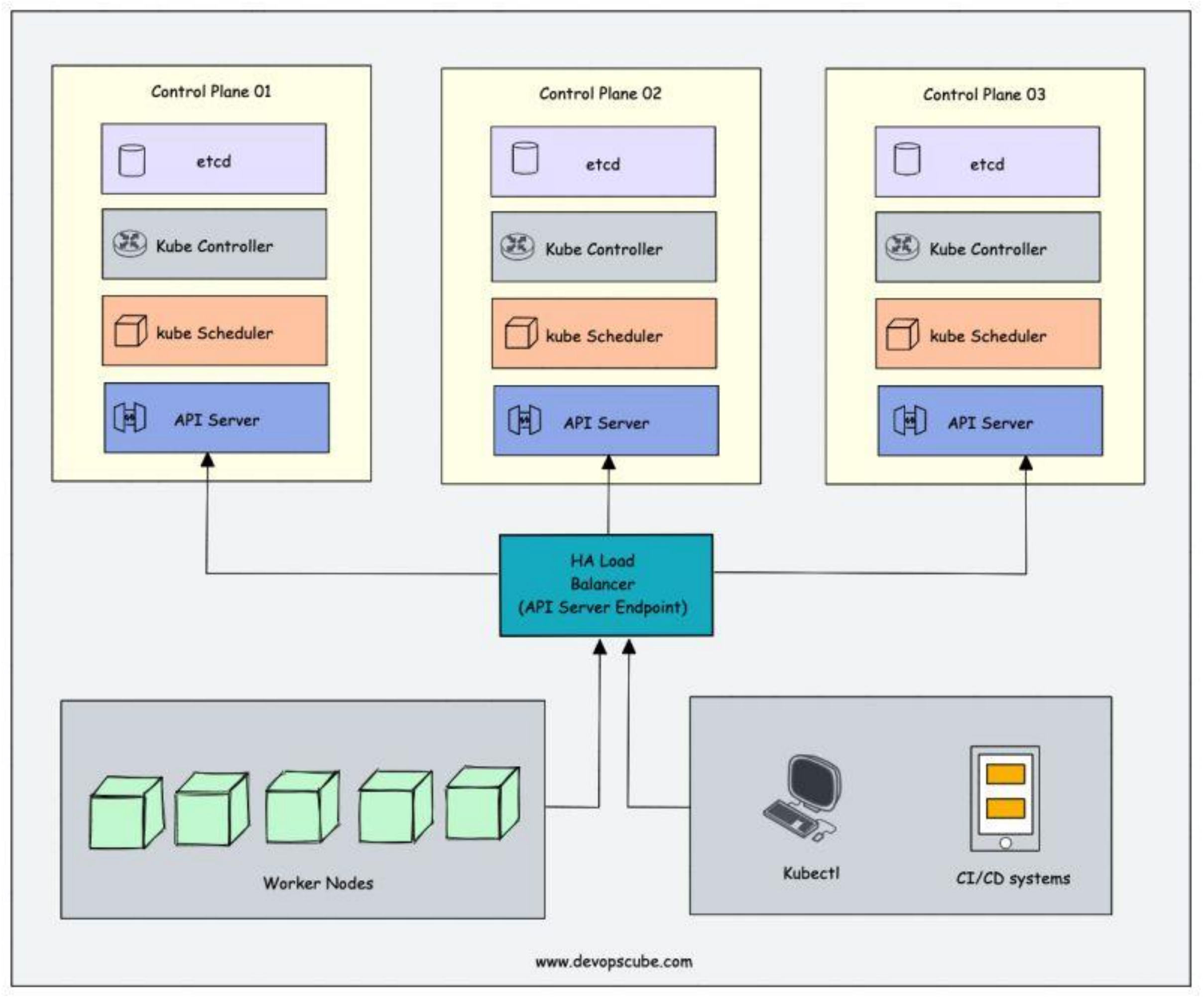
In the event, that a leader fails, a follower will be elected a leader and takes over all the operations.

What Happens During Control Plane Failure?

Control Plane Failure leads to a loss of communication between the control plane components, such as the API server, and the worker nodes.

As a result, new deployments, scaling, and updates to existing resources may not be possible until the control plane is restored. However, the applications that are already running on worker nodes should continue to function, as they are not directly affected by control plane failures.

Note: I have covered only high-level concepts due to word limitation.



Kubernetes Learning [Day 13]

Let's learn about **Kubernetes Network Design** 🚀

Creating a Kubernetes cluster in an open cloud network is pretty easy.

However, it is not easy when it comes to cluster implementation in corporate networks.

While creating a Kubernetes cluster, you need the following set of IP address ranges.

➡️ **Node CIDR**: IP range for Cluster Nodes
➡️ **Pod CIDR**: IP range for Pods
➡️ **Service CIDR** (Specific cases like GKE)

Each node would have /24 pod network (255 IPs) assigned with recommended pod limit of 110

When it comes to managed Kubernetes services like GKE, EKS, etc ..the control plane will be part of a different network managed by the cloud provider.

When deploying Kubernetes in a hybrid corporate network, try to avoid using the corporate network range for pods. It could eat up a lot of IP addresses.

Most cloud providers offer Secondary IP ranges in VPC that can be used for POD CIDR ranges. These ranges are normally different from the corporate network range.

For example, if the corporate network range is **10.x.x.x**, secondary ranges could be **172.x.x**, **198.x.x or 100.64.xx** ranges.

The secondary ranges are routable within the VPC network or peered networks.

However, if there is hybrid connectivity to on-prem data centers, in order for the pod to make requests to on-prem services, the second range should be part of corporate network routes.

But if you don't want to expose your pod IPs, you might need to use something like an IP masquerading agent in your cluster so that the outgoing traffic will always have the Node IP as the source identity. I have tried this in GKE.

Kubernetes supports **IPv4/IPv6 dual-stack**. Most managed Kubernetes services like GKE, EKS, etc support IPv6 networking.

The most important thing is, the network range for Kubernetes should not be overlapping with the corporate network.

☑️ Always discuss with the network team and carve out an IP range from the available IP address space.

❌ Don't allocate big ranges just because it is available. Calculate total node/pod/service requirements and choose a range based on that.

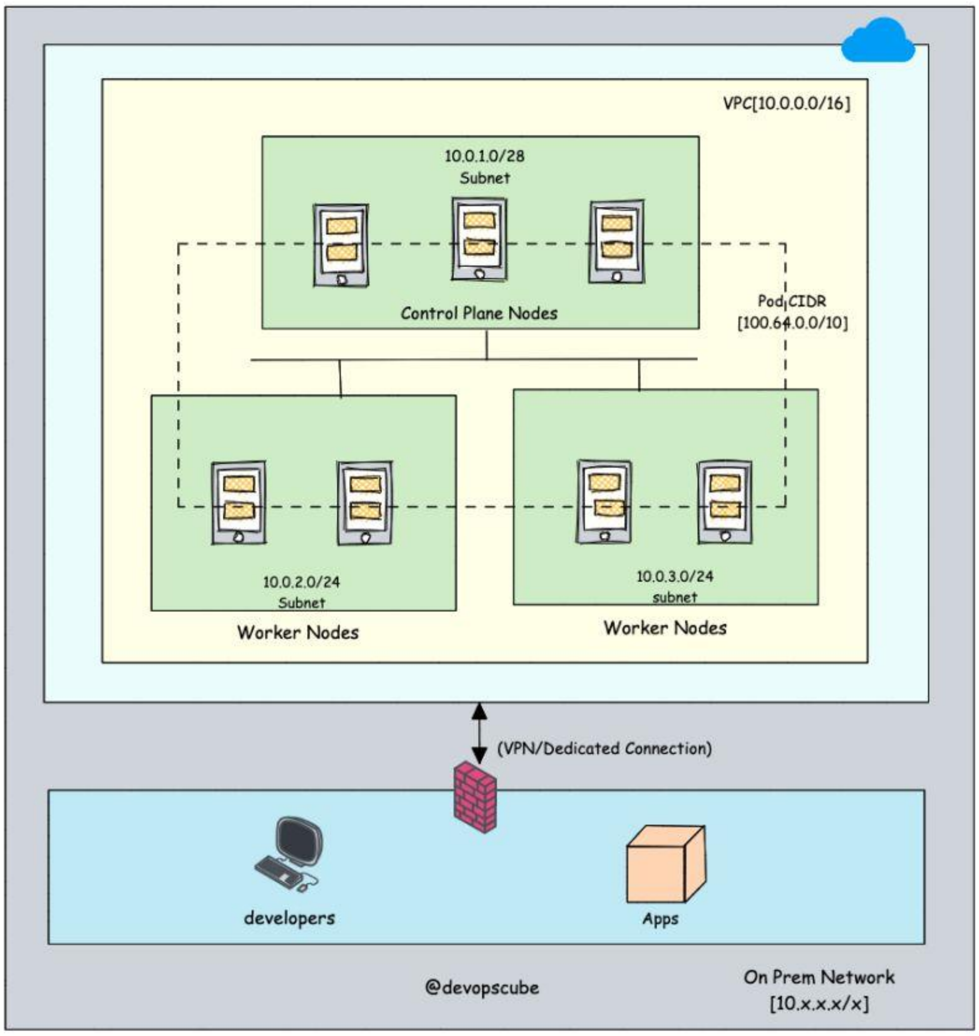
◐ Also, if you plan to host PCI/PII-compliant apps, you need to discuss with both network/security teams about the compliance requirements.

Networking is a complex topic and learnings come after project implementation.

☑ So be a good devops engineer and collaborate with the right teams to accommodate future networking requirements and avoid potential issues.

Please feel free to add your experiences and suggestions.

Tomorrow we will look at using free cloud credits for launching clusters.



Kubernetes Learning [Day 14]

What is the best way to learn about Kubernetes Cluster configurations?

By **setting up the cluster the hard way** on a cloud platform.

This way, you will learn most of the concepts we discussed in a hands-on manner.

Meaning, you need to do the following:

- ☑ Create a VPC network for the cluster
- ☑ Configure firewall rules and network routes
- ☑ Provision servers for control plane and worker nodes
- ☑ Provision CA certificates to generate SSL for cluster components
- ☑ Install each and every Kubernetes component and set up authentication
- ☑ Set up Pod Network routes
- ☑ Test the cluster by deploying a test application.

There is a popular repo named "**Kubernetes the Hard Way**" which contains step-by-step instructions to perform all of the above on Google cloud with $300 free credits.

Spend some time and finish the lab, trying to understand everything with further research.

Github Repo: https://lnkd.in/g3y_6waD

If you face any problems during the cluster setup, you are lucky. The best learning comes when fixing systems.