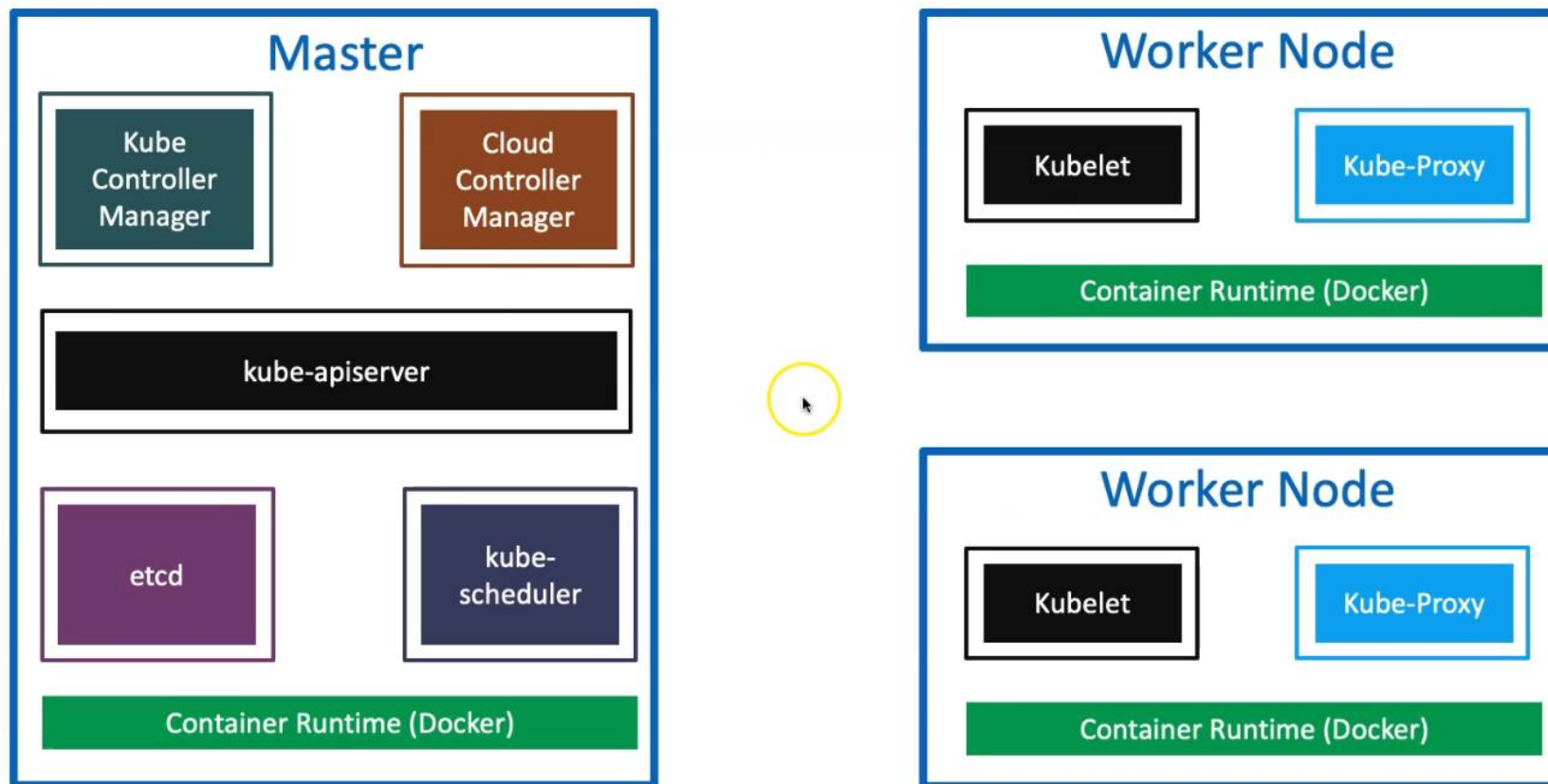


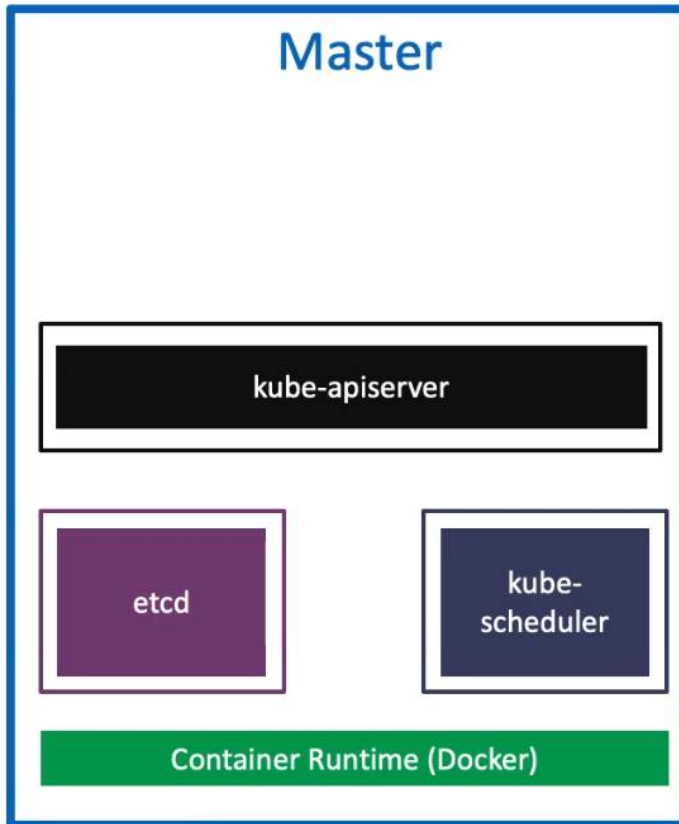
Kubernetes Architecture Explained



Kubernetes - Architecture

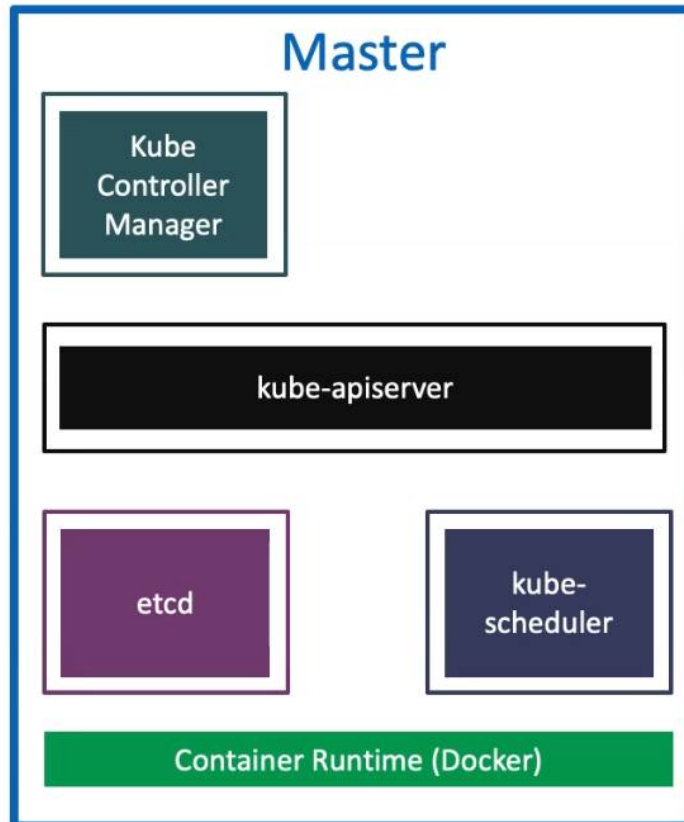


Kubernetes Architecture - Master



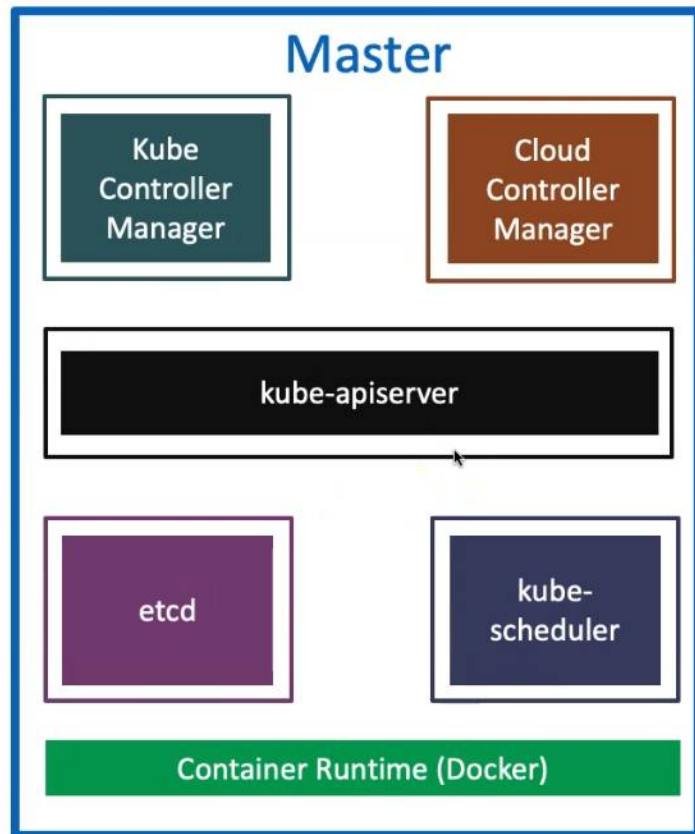
- **kube-apiserver**
 - It acts as **front end** for the Kubernetes control plane. It **exposes** the Kubernetes API
 - CLI tools (like **kubectl**), Users and even Master components (**scheduler, controller manager, etcd**) and Worker node components like (**Kubelet**) **everything talk** with API Server.
- **etcd**
 - Consistent and highly-available **key value store** used as Kubernetes' **backing store** for all cluster data.
 - It **stores** all the masters and worker node information.
- **kube-scheduler**
 - Scheduler is responsible for **distributing containers** across multiple nodes.
 - It **watches** for newly created Pods with no assigned node, and selects a node for them to run on.

Kubernetes Architecture - Master



- **kube-controller-manager**
 - Controllers are **responsible for noticing** and responding when nodes, containers or endpoints go **down**. They make decisions to **bring up new containers** in such cases.
 - **Node Controller**: Responsible for noticing and responding when **nodes go down**.
 - **Replication Controller**: Responsible for maintaining the **correct number of pods** for every replication controller object in the system.
 - **Endpoints Controller**: **Populates** the Endpoints object (that is, joins Services & Pods)
 - **Service Account & Token Controller**: Creates default accounts and API Access for **new namespaces**.

Kubernetes Architecture - Master



- cloud-controller-manager

- A Kubernetes control plane component that embeds **cloud-specific control logic**.
- It only runs controllers that are **specific** to your cloud provider.
- **On-Premise** Kubernetes clusters will not have this component.
- **Node controller**: For **checking** the cloud provider to determine if a node has been deleted in the cloud after it stops responding
- **Route controller**: For setting up **routes** in the underlying cloud infrastructure
- **Service controller**: For creating, updating and deleting cloud provider **load balancer**

Kindly explain the above images in details



The provided images illustrate the architecture of Kubernetes and the role of its various components. Here's a detailed explanation of each:

Kubernetes Architecture Overview (First Image)

1. Master Node:

- Acts as the control plane and manages the cluster.
- Components include:
 - **Kube Controller Manager:** Ensures the cluster's desired state by managing nodes, replication controllers, and endpoints.
 - **Cloud Controller Manager:** Manages cloud-specific integrations such as routes and load balancers.
 - **Kube API Server:** The front-end for the Kubernetes cluster, responsible for exposing the Kubernetes API to interact with the cluster.
 - **etcd:** A distributed key-value store for storing configuration and cluster state.
 - **Kube Scheduler:** Allocates resources by assigning newly created Pods to nodes based on resource availability and constraints.
 - **Container Runtime (Docker):** Executes containerized applications.

2. Worker Nodes:

- Responsible for running the application workloads.
- Components include:
 - **Kubelet:** Ensures containers are running as expected by communicating with the API server.
 - **Kube Proxy:** Manages network rules for routing traffic to the appropriate containers.
 - **Container Runtime (Docker):** Runs the application containers.

Kubernetes Master Node Components (Second to Fourth Images)

1. Kube API Server (Second Image)

- **Purpose:**
 - Acts as the cluster's communication hub.
 - Exposes the Kubernetes API, enabling interaction with the cluster via CLI tools like `kubectl`.
- **Key Roles:**
 - Facilitates communication between Master components (scheduler, controller manager, etc.) and Worker Node components (Kubelet).
 - Handles REST API requests.

2. etcd (Second Image)

- **Purpose:**
 - A consistent, highly-available key-value store that acts as Kubernetes' database.
- **Key Roles:**
 - Stores all cluster data, including node information, configurations, and current cluster states.

3. Kube Scheduler (Second Image)

- **Purpose:**
 - Assigns Pods to Worker Nodes.
- **Key Roles:**
 - Monitors unassigned Pods and selects suitable nodes for execution based on resources and constraints.

4. Kube Controller Manager (Third Image)

- **Purpose:**
 - Manages various controllers that monitor the cluster's state and take corrective action.
- **Key Controllers:**
 - **Node Controller:** Monitors node health.
 - **Replication Controller:** Ensures the correct number of Pods are running.
 - **Endpoints Controller:** Manages connections between services and Pods.
 - **Service Account Controller:** Manages API access tokens.

5. Cloud Controller Manager (Fourth Image)

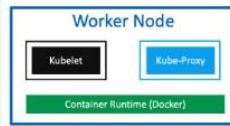
- **Purpose:**
 - Manages cloud-specific integrations in the cluster.
- **Key Features:**
 - **Node Controller:** Checks if nodes are removed from the cloud provider.
 - **Route Controller:** Sets up network routes.
 - **Service Controller:** Handles cloud-based load balancers.

Summary

- **Master Node** handles the cluster's control plane operations.
- **Worker Nodes** execute application workloads.
- Kubernetes ensures scalability, high availability, and automation of deployments through its components. Each component has a specialized role for efficient cluster management.



Kubernetes Architecture – Worker Nodes



- **Container Runtime**
 - Container Runtime is the underlying software where we run all these Kubernetes components.
 - We are using Docker, but we have other runtime options like rkt, container-d etc.

- **Kubelet**
 - Kubelet is the agent that runs on every node in the cluster
 - This agent is responsible for making sure that containers are running in a Pod on a node.
- **Kube-Proxy**
 - It is a network proxy that runs on each node in your cluster.
 - It maintains network rules on nodes
 - In short, these network rules allow network communication to your Pods from network sessions inside or outside of your cluster.

Kindly explain the above images in details



The image explains the architecture of Kubernetes **Worker Nodes**, which are responsible for running application workloads. Here's a detailed explanation of the components:

Worker Node Components

1. Kubelet:

- **Definition:** It is the agent that runs on every node in the cluster.
- **Purpose:**
 - Ensures that containers defined in Pod specifications are running and healthy.
 - Continuously communicates with the Kubernetes API server to get work instructions and report node and pod status.
- **Key Responsibilities:**
 - Monitors the containers running on the node.
 - Ensures that the state of the containers matches the desired state defined in Kubernetes.
 - Handles Pod lifecycle events and updates.

2. Kube-Proxy:

- **Definition:** A network proxy and load balancer that runs on every node in the cluster.
- **Purpose:**
 - Manages network rules and ensures seamless communication between Pods, as well as between external users and services inside the cluster.

- Handles service discovery and routing traffic to the appropriate Pod based on the network rules.
- **Key Responsibilities:**
 - Maintains network rules for Pod-to-Pod communication.
 - Enables access to services from both within and outside the cluster.

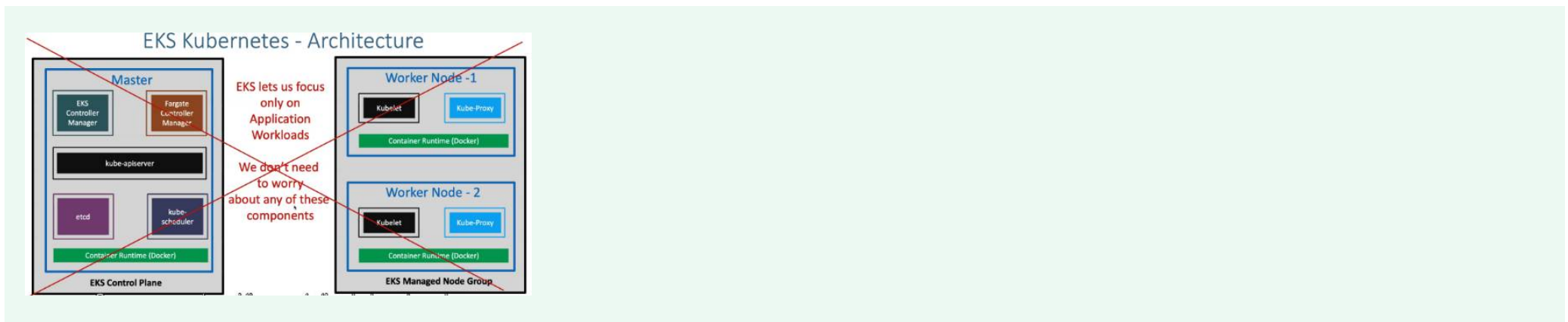
3. Container Runtime:

- **Definition:** The software responsible for running containers on a node.
- **Purpose:**
 - Provides the environment required for containers to run.
 - Supports container execution, pulling container images, and resource isolation.
- **Examples:**
 - Docker (used in the image), containerd, CRI-O, etc.
- **Key Role:**
 - Acts as the foundation on which containers are executed and managed by the Kubelet.

Summary of Worker Node Workflow:

1. The **Kubelet** ensures that the desired state of the Pods (as defined by the Kubernetes API server) is maintained on the Worker Node.
2. The **Kube-Proxy** handles the network traffic and routes it to the correct Pod or service.
3. The **Container Runtime** executes and isolates containerized workloads.

This setup ensures efficient management of application workloads and seamless communication between different parts of the Kubernetes cluster.



Kindly explain the above images in details



The image depicts the architecture of **EKS (Amazon Elastic Kubernetes Service)**, which is a managed Kubernetes service provided by AWS. Here's a detailed explanation of the architecture and its significance:

Overview of EKS Architecture

1. EKS Control Plane (Managed by AWS):

- AWS manages the Kubernetes **Master Node** components for you, including:
 - **EKS Controller Manager:** Handles Kubernetes controllers for managing workloads and cluster state.
 - **Fargate Controller Manager:** Manages tasks when using AWS Fargate for serverless Kubernetes.
 - **Kube-API Server:** Acts as the interface for Kubernetes, exposing APIs for users and other components.
 - **etcd:** A distributed key-value store that stores cluster configuration, state, and metadata.
 - **Kube Scheduler:** Assigns Pods to nodes based on resource availability and constraints.
- **Key Point:** Since AWS manages the control plane, users do not need to worry about maintaining or scaling these components.

2. EKS Managed Node Groups (Worker Nodes):

- These are the nodes (EC2 instances or Fargate tasks) where your workloads run.
- Components of a Worker Node:
 - **Kubelet:**
 - Ensures that the desired state of containers is maintained on the node.
 - Communicates with the Kubernetes API server.
 - **Kube-Proxy:**
 - Manages networking and routing traffic to Pods.
 - Enables seamless service discovery and communication within the cluster.
 - **Container Runtime (Docker):**
 - Runs containers as defined in the Pod specifications.
- **Key Point:** While you manage Worker Nodes in EKS, AWS provides tools to simplify this process, like auto-scaling and managed node groups.

Key Advantages of EKS:

1. Focus on Application Workloads:

- Developers and administrators can focus on deploying and managing applications rather than worrying about the operational overhead of managing Kubernetes control plane components.

2. Reduced Complexity:

- AWS handles upgrades, backups, and scaling of the control plane, ensuring high availability and resilience.

3. Flexibility with Node Groups:

- You can use EC2 instances for more control or AWS Fargate for a serverless approach to run workloads.

4. Seamless Integration with AWS Services:

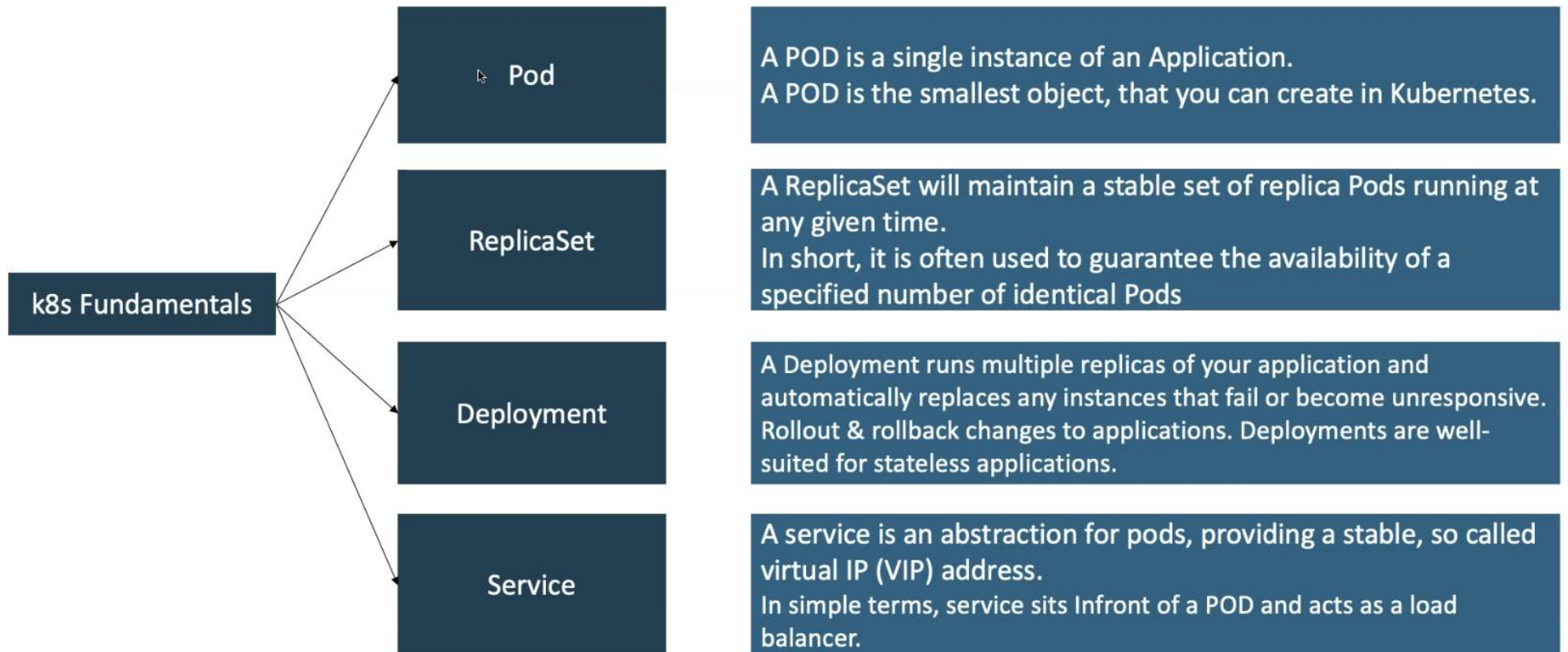
- EKS integrates well with other AWS services such as IAM, CloudWatch, and ELB for managing access control, monitoring, and load balancing.
-

Summary

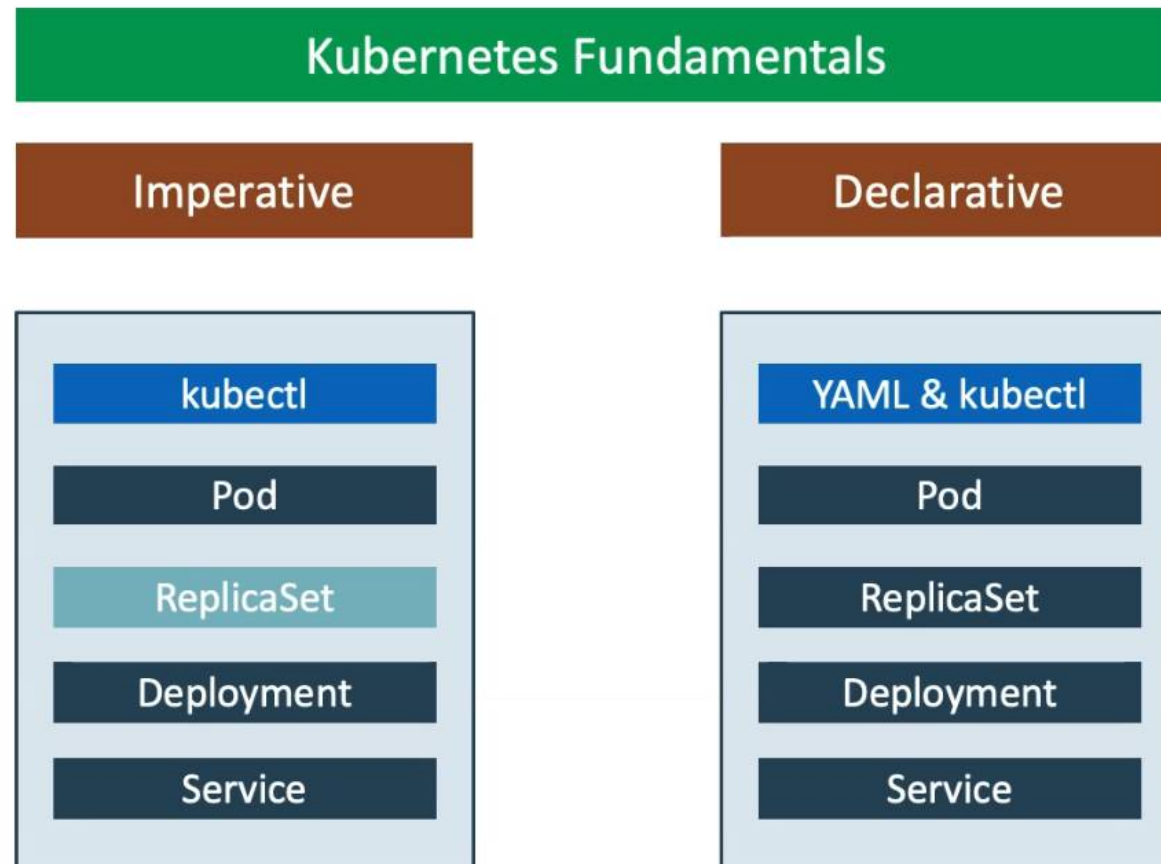
- In EKS, the Kubernetes control plane (Master Node components) is fully managed by AWS.
- Users only need to manage the Worker Nodes where application workloads run, but AWS simplifies this with tools like Managed Node Groups and Fargate.
- This architecture allows users to focus on deploying and scaling their applications while benefiting from AWS's infrastructure reliability and scalability.



Kubernetes - Fundamentals



Kubernetes - Imperative & Declarative



Kindly explain the above images in details



The provided images illustrate two key aspects of Kubernetes: its **fundamentals** and the difference between **imperative** and **declarative** approaches. Here's a detailed explanation:

Kubernetes Fundamentals (First Image)

1. Pod

- **Definition:** The smallest deployable unit in Kubernetes, representing a single instance of an application.
- **Details:**
 - A Pod encapsulates one or more containers that share the same storage, network, and namespace.
 - Used as the basic building block for running applications in Kubernetes.

2. ReplicaSet

- **Definition:** Ensures a specified number of identical Pods are running at any given time.
- **Details:**
 - Monitors Pods and replaces them if they fail or are deleted.
 - Guarantees high availability by maintaining the desired state of Pods.

3. Deployment

- **Definition:** Manages ReplicaSets and Pods, providing declarative updates to applications.
- **Details:**
 - Supports rollouts and rollbacks for application changes.
 - Useful for deploying stateless applications with multiple replicas.

4. Service

- **Definition:** An abstraction that exposes a set of Pods as a network service.
- **Details:**
 - Provides a stable IP and DNS name for Pods, even as Pods are created and destroyed.
 - Acts as a load balancer to distribute traffic to Pods.

Imperative vs Declarative Approaches (Second Image)

Imperative Approach

- **Definition:** Directly executes commands to manage Kubernetes resources.
- **Details:**
 - Uses CLI tools like `kubectl` for managing resources.
 - Examples:
 - `kubectl run` to create Pods.
 - `kubectl scale` to adjust replicas.
- **Advantages:**
 - Quick and straightforward for small-scale tasks.
- **Disadvantages:**
 - Less maintainable for complex or large-scale deployments.

Declarative Approach

- **Definition:** Uses configuration files (e.g., YAML) to define the desired state of Kubernetes resources.
- **Details:**
 - Resources are applied using `kubectl apply`.
 - Kubernetes ensures the actual state matches the desired state defined in the YAML files.
- **Advantages:**
 - Easier to manage and version control.
 - Better suited for automation and complex environments.
- **Disadvantages:**
 - Requires more initial effort to define configurations.

Summary

- **Kubernetes Fundamentals:**
 - Pods, ReplicaSets, Deployments, and Services form the foundation of Kubernetes for running and managing applications.
- **Imperative vs Declarative:**
 - The imperative approach is manual and suitable for quick tasks.
 - The declarative approach is configuration-driven, making it ideal for scalable and maintainable deployments.

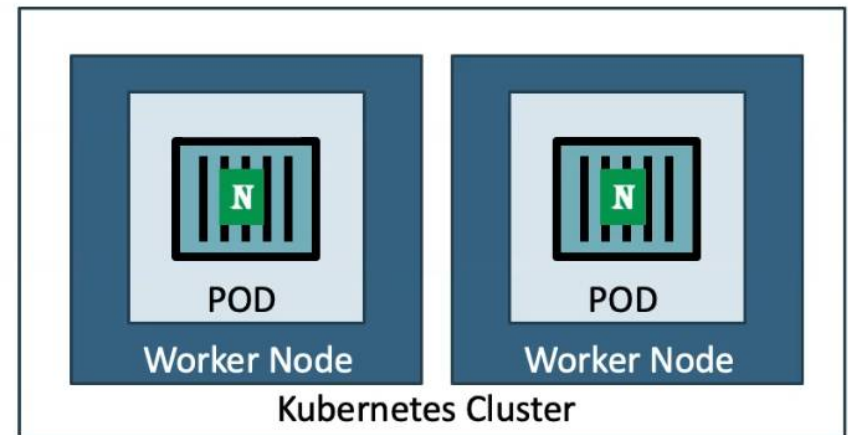


Kubernetes - POD

- With Kubernetes our core goal will be to **deploy our applications** in the form of **containers** on **worker nodes** in a k8s cluster.
- Kubernetes **does not** deploy containers directly on the worker nodes.
- Container is **encapsulated** in to a Kubernetes Object named **POD**.
- A POD is a **single instance** of an application.
- A POD is the **smallest object** that we can create in Kubernetes.

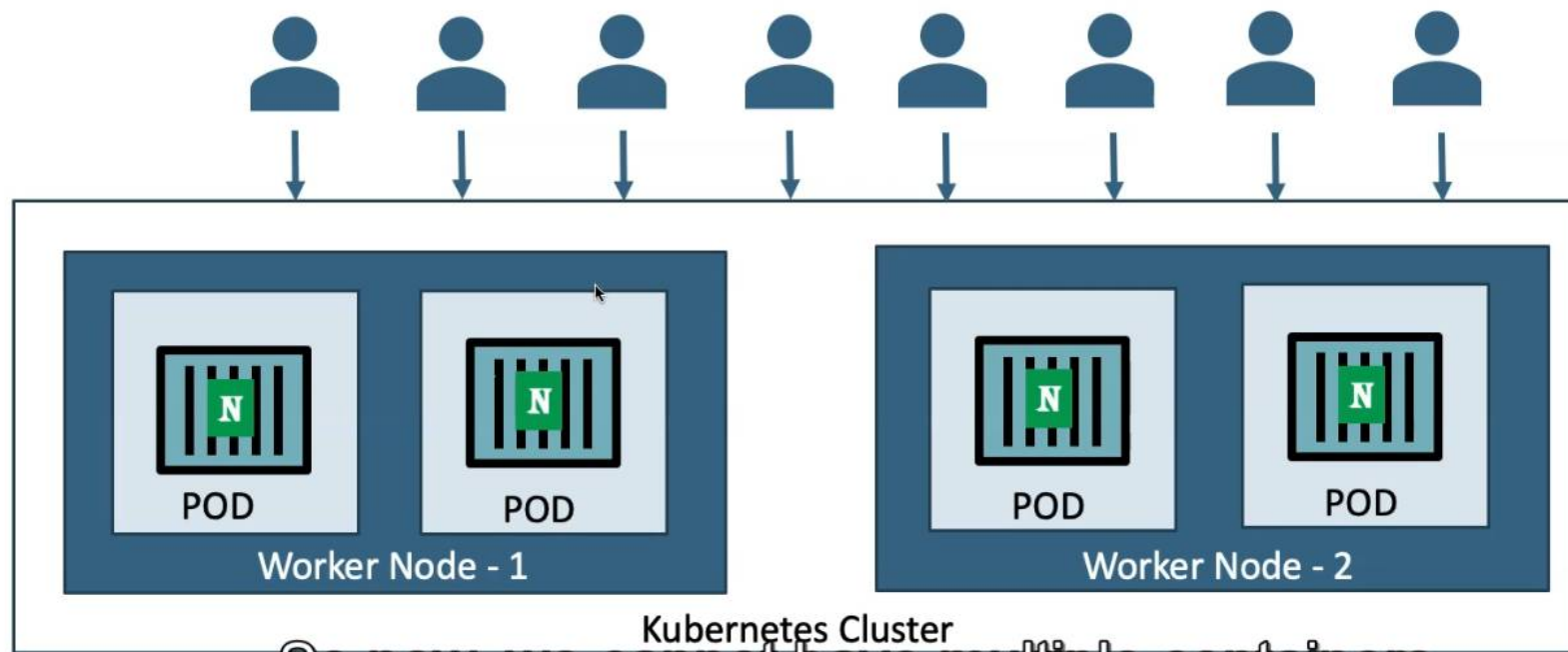


Nginx Container
Image



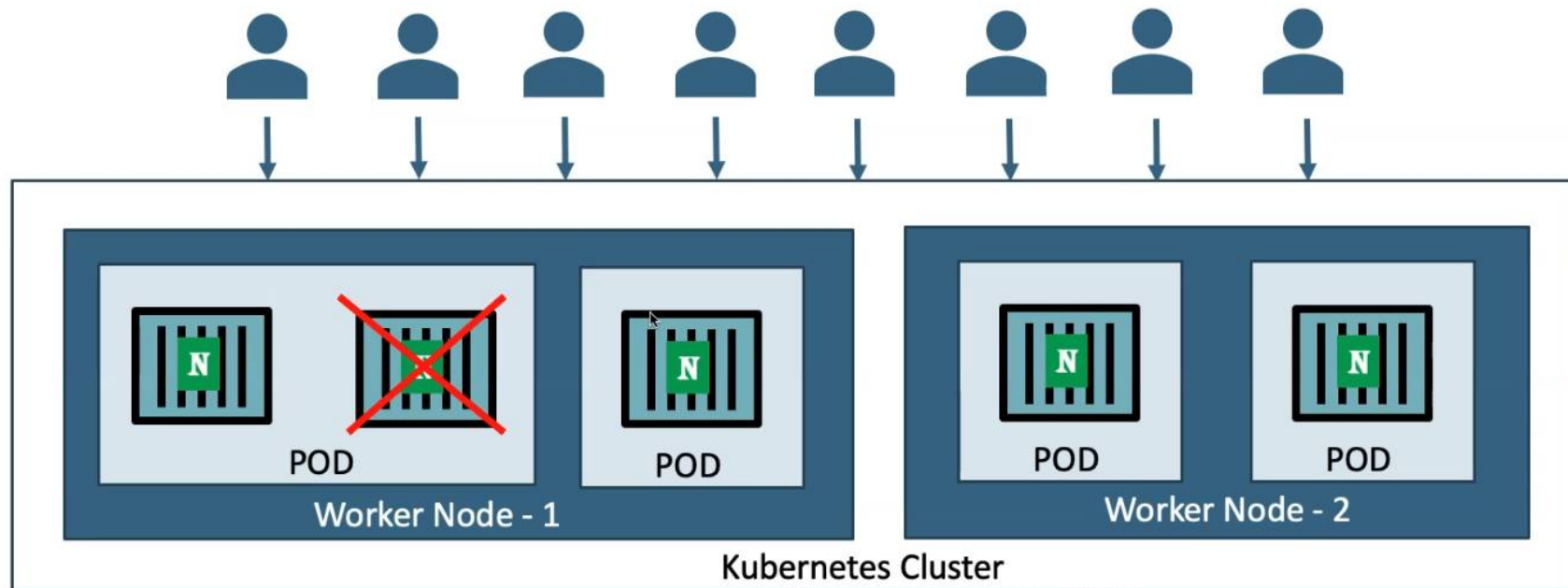
Kubernetes - POD

- PODs generally have **one to one** relationship with containers.
- To scale up we **create** new POD and to scale down we **delete** the POD.



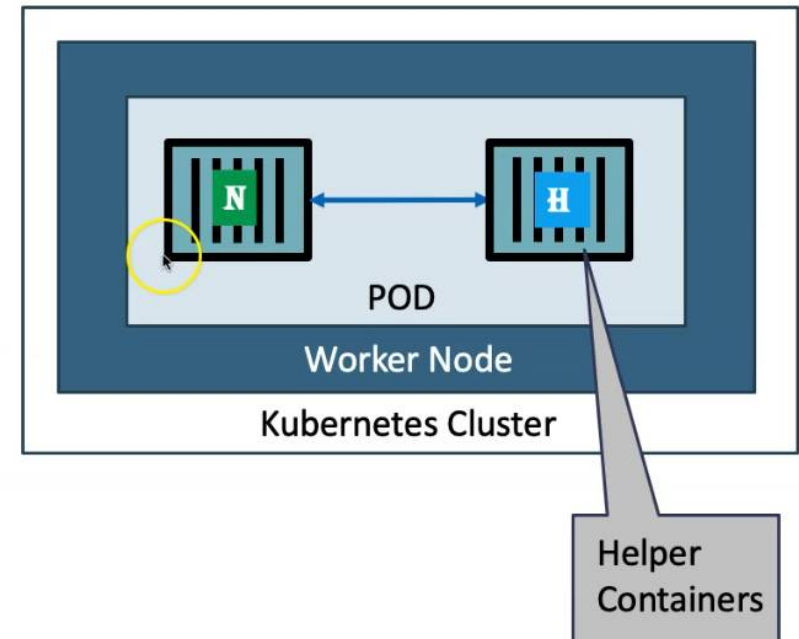
Kubernetes – PODs

- We **cannot have** multiple containers of **same kind** in a single POD.
- **Example:** Two NGINX containers in single POD serving same purpose is **not recommended**.



Kubernetes – Multi-Container Pods

- We can have multiple containers in a single POD, provided **they are not of same kind**.
- **Helper Containers (Side-car)**
 - **Data Pullers:** Pull data required by Main Container
 - **Data pushers:** Push data by collecting from main container (logs)
 - **Proxies:** Writes static data to html files using Helper container and Reads using Main Container.
- **Communication**
 - The two containers can easily communicate with each other easily as they share same **network space**.
 - They can also easily share **same storage space**.
- Multi-Container Pods is a **rare use-case** and we will try to focus on core fundamentals.



Kindly explain the above images in details



The images focus on the concept of Kubernetes **Pods** and provide details about their characteristics, scaling, usage of multiple containers, and scenarios when multiple containers are used. Here's a detailed explanation:

Kubernetes - Pod (First Image)

1. Pod Basics:

- A **Pod** is the smallest deployable unit in Kubernetes and represents a single instance of an application.
- Kubernetes **does not deploy containers directly** on worker nodes; instead, containers are encapsulated in Pods.

2. Encapsulation:

- A Pod can encapsulate one or more containers (usually one for most use cases).
- The example shows a Pod running an NGINX container.

3. Deployment:

- Pods are scheduled on **Worker Nodes** in the Kubernetes cluster.
 - The goal of Kubernetes is to manage and scale these Pods to meet application demands.
-

Kubernetes - Pod Characteristics (Second Image)

1. One-to-One Relationship:

- Typically, a Pod has a **1:1 relationship** with a container, meaning one Pod runs one container.

2. Scaling:

- To **scale up**, new Pods are created.
- To **scale down**, existing Pods are deleted.

3. Worker Node Distribution:

- Pods are distributed across multiple Worker Nodes for high availability and efficient resource utilization.
-

Kubernetes - Pod Restrictions (Third Image)

1. Same Kind Containers:

- Kubernetes **does not recommend** having multiple containers of the same kind within a single Pod.
- Example: Running two NGINX containers in a single Pod is not recommended.

2. Reasoning:

- Each container within a Pod shares the same network namespace and storage, which might lead to resource conflicts or redundancy if identical containers are used.

Kubernetes - Multi-Container Pods (Fourth Image)

1. Use Case for Multi-Container Pods:

- A Pod can have multiple containers as long as they are **not of the same kind**.
- Commonly used when a helper or sidecar container is needed to support the main container.

2. Examples of Helper Containers:

- **Data Pullers:** Containers that fetch data required by the main container.
- **Data Pushers:** Containers that handle logs or push data from the main container.
- **Proxies:** Containers that process static data or provide utility services to the main container.

3. Communication:

- Containers within a Pod share the same **network space** and **storage volume**, enabling efficient communication and data sharing.

4. Rare Use Case:

- Multi-container Pods are used sparingly and typically for specialized tasks (e.g., logging, caching).

Summary

- **Pods** are the smallest unit in Kubernetes, encapsulating one or more containers.
- For scaling, Kubernetes creates or deletes Pods dynamically.
- It is recommended to have a **single container per Pod**, except in specific scenarios where helper containers (sidecars) are needed.
- Multi-container Pods are a rare use case and are generally used to achieve specialized functionality like logging or caching alongside the main application.



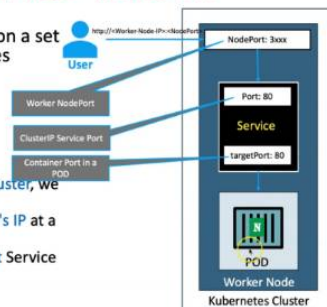
Kubernetes – Service - NodePort

- We can expose an application running on a set of **PODs** using different types of Services available in k8s.

- ClusterIP
- NodePort
- LoadBalancer

- **NodePort Service**

- To access our application **outside of k8s cluster**, we can use NodePort service.
- Exposes the Service on each **Worker Node's IP** at a static port (nothing but NodePort).
- A **ClusterIP** Service, to which the **NodePort** Service routes, is **automatically** created.
- Port Range **30000-32767**



Kindly explain the above images in details



The image explains the **NodePort Service** in Kubernetes, a type of service used to expose applications running in a cluster to the outside world. Here's a detailed explanation:

Kubernetes - Services Overview

1. Purpose of Services:

- Services in Kubernetes allow communication with Pods, which are ephemeral in nature.
- They provide stable networking and abstraction for Pods.

2. Types of Services:

- **ClusterIP**: Default service type, exposes the service within the cluster.
- **NodePort**: Exposes the service on a static port on each Worker Node's IP, allowing external access.
- **LoadBalancer**: Integrates with cloud provider load balancers to expose services externally.

NodePort Service

1. Definition:

- A **NodePort Service** allows you to expose your application to external traffic outside the Kubernetes cluster.
- It maps a **Node's IP** and a specific **port** to the service running inside the cluster.

2. How It Works:

- **Target Port:** The container port inside the Pod where the application runs (e.g., `80`).
- **Port:** The port where the service listens within the cluster (e.g., `80`).
- **NodePort:** A static port in the range `30000-32767` that is exposed on the Worker Node's IP to route traffic to the service.

3. Example Workflow:

- A user sends a request to `**Worker Node IP**:**NodePort**` (e.g., `192.168.1.100:30001`).
- The request is routed to the service listening on its defined **Port**.
- The service forwards the request to the container's **Target Port** in the Pod running the application.

4. Automatically Creates a ClusterIP:

- When you create a NodePort Service, a **ClusterIP Service** is automatically created to handle internal communication within the cluster.

Benefits of NodePort Service

1. External Access:

- Allows access to applications from outside the cluster without additional tools like ingress or load balancers.

2. Direct Connection:

- You can access the application using any node's IP in the cluster and the assigned NodePort.

Limitations

1. Port Range:

- NodePorts are limited to the range `30000-32767`, which might conflict with other services.

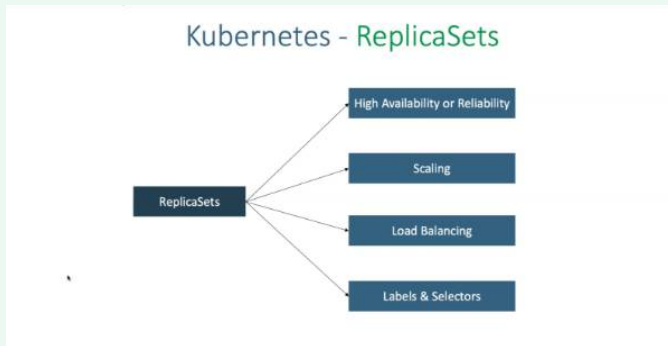
2. Scalability:

- Not ideal for production-grade traffic distribution since there is no load balancing across multiple nodes.

Summary

- A **NodePort Service** is used to expose applications outside the Kubernetes cluster.
- Traffic flows from **NodePort** → **Service Port** → **Target Port** (inside the Pod).

- It is simple to use for external access but has limitations in terms of scalability and load balancing. For production use, **LoadBalancer** or ingress controllers are preferred.



Kindly explain the above images in details



The image explains **ReplicaSets** in Kubernetes, which are essential for managing the availability and scalability of Pods. Below is a detailed explanation of the key aspects highlighted in the image:

What is a ReplicaSet?

- A **ReplicaSet** is a Kubernetes resource used to ensure a specified number of **replicas (instances)** of a Pod are running at all times.
- It monitors the state of Pods and makes adjustments (e.g., creating or deleting Pods) to maintain the desired state.

Key Features of ReplicaSets

1. High Availability or Reliability:

- Ensures that the application remains highly available by maintaining the desired number of Pod replicas.
- If a Pod fails or is deleted, the ReplicaSet automatically replaces it to meet the defined replica count.
- Helps in avoiding downtime for applications.

2. Scaling:

- ReplicaSets allow horizontal scaling of applications.
- You can increase or decrease the number of Pod replicas dynamically by changing the `replicas` field in the ReplicaSet specification.`
- Scaling can be manual or automated (using Horizontal Pod Autoscalers).

3. Load Balancing:

- When combined with Kubernetes **Services**, ReplicaSets help distribute traffic across multiple Pods.
- Load balancing ensures that incoming requests are evenly distributed among Pods, improving performance and reliability.

4. Labels and Selectors:

- ReplicaSets use **labels** to identify and manage the Pods they control.
- **Selectors** define the criteria for matching Pods to the ReplicaSet.
- This mechanism ensures that the ReplicaSet only manages the intended Pods, even if other Pods exist in the cluster.

How it Works

- When a ReplicaSet is created, it looks for existing Pods that match its label selector.
- If the number of matching Pods is less than the desired replicas, it creates new Pods.
- If there are more Pods than required, it terminates the excess Pods.
- This constant monitoring and adjustment maintain the desired state.

Use Cases

1. Stateless Applications:

- ReplicaSets are commonly used for stateless applications like web servers, where each Pod serves similar functionality.

2. Application Recovery:

- Ensures that failed Pods are replaced automatically to keep the application running.

3. Scaling for Demand:

- Adjusting replicas to handle increased or decreased traffic during peak or off-peak hours.

Summary

- A **ReplicaSet** is a fundamental Kubernetes object that provides high availability, reliability, and scalability for Pods.
- By leveraging labels and selectors, ReplicaSets ensure precise control over Pod management.
- It plays a crucial role in maintaining consistent application performance and reliability in dynamic environments.