

Docker Containers and Docker

Containers and Docker

Section overview

1

Why use containers?

1. Compare how deployments happen with and without containers
2. Understand the main benefits of using containers

2

Containers and Virtual Machines

1. Compare and highlight the differences between using containers and VMs for running workloads

3

Docker components

1. Discuss the different components from a Docker system
2. Understand how these components interact when running common operations in Docker

Docker

Docker

Why Containers?

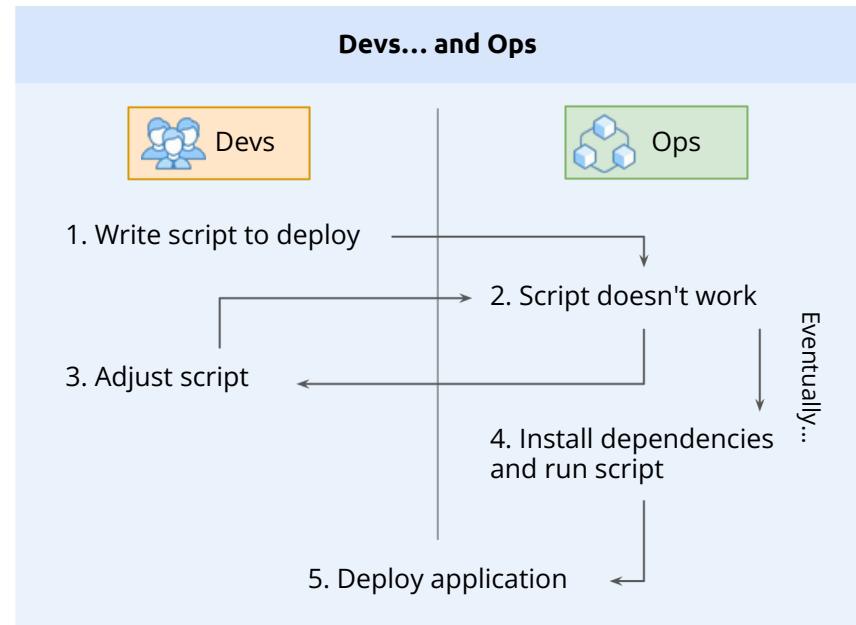
Why Containers?

How did we build and deploy applications, really?

- What do we need to deploy a NodeJS app?
 - Install NodeJS dependencies
 - Install NodeJS itself
 - Install the app's dependencies
 - Run the application
- How about with other programming languages?



- How about with other NodeJS versions?
- Not to mention managing multiple applications running side by side...



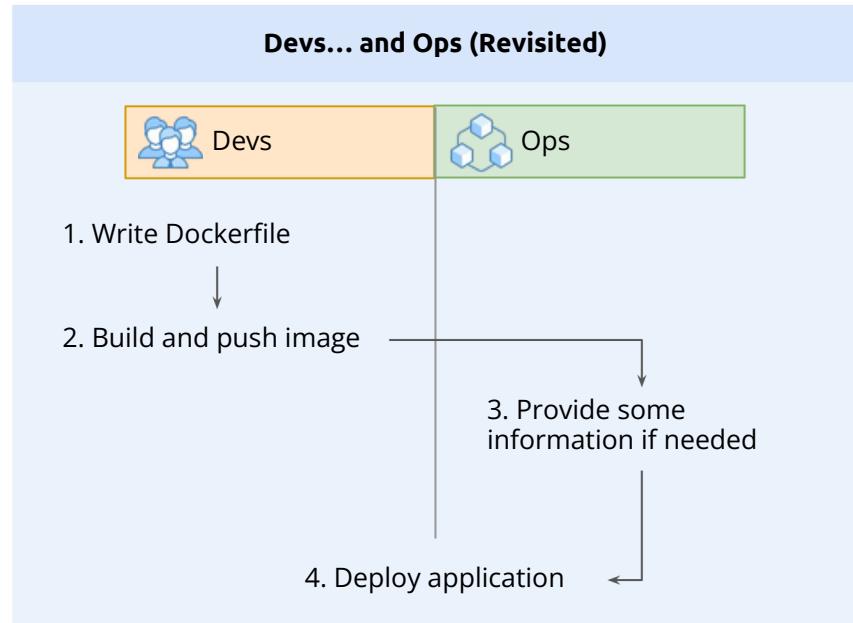
Docker

Why Containers?

How do containers come to the rescue?

- Containers encapsulate all the dependencies and configuration necessary to run whatever application. From the outside, they all look the same and are all run (almost) the same way. This leads to, among others:

- Simplified setup
- Portability
- Consistent environments
- Isolation
- Efficiency
 - Better resource control
 - Easily scalable applications

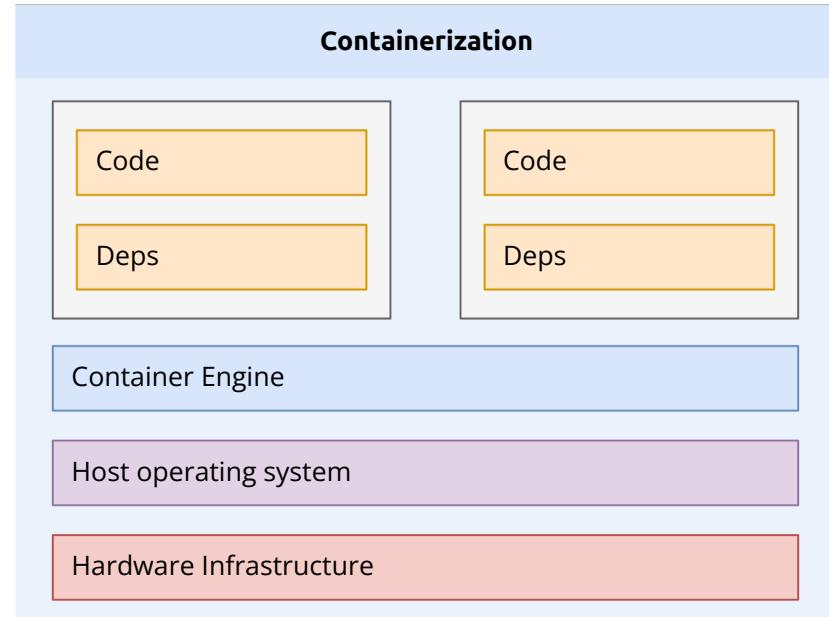
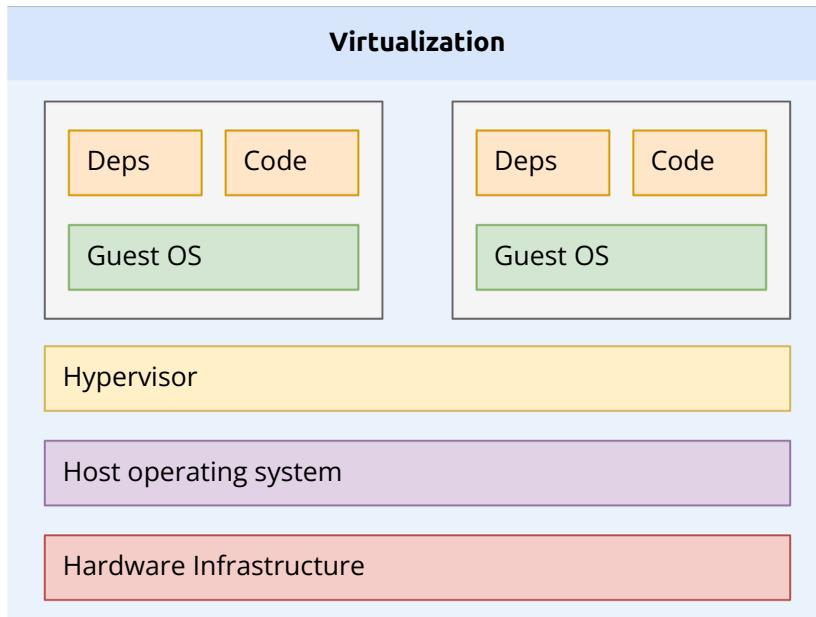


Docker

Containers and Virtual Machines

Containers and Virtual Machines

Containers are not the only solution



Docker

Containers and Virtual Machines

Containers are not the only solution

Virtualization



Containerization



Docker

Containers and Virtual Machines

Containers are not the only solution

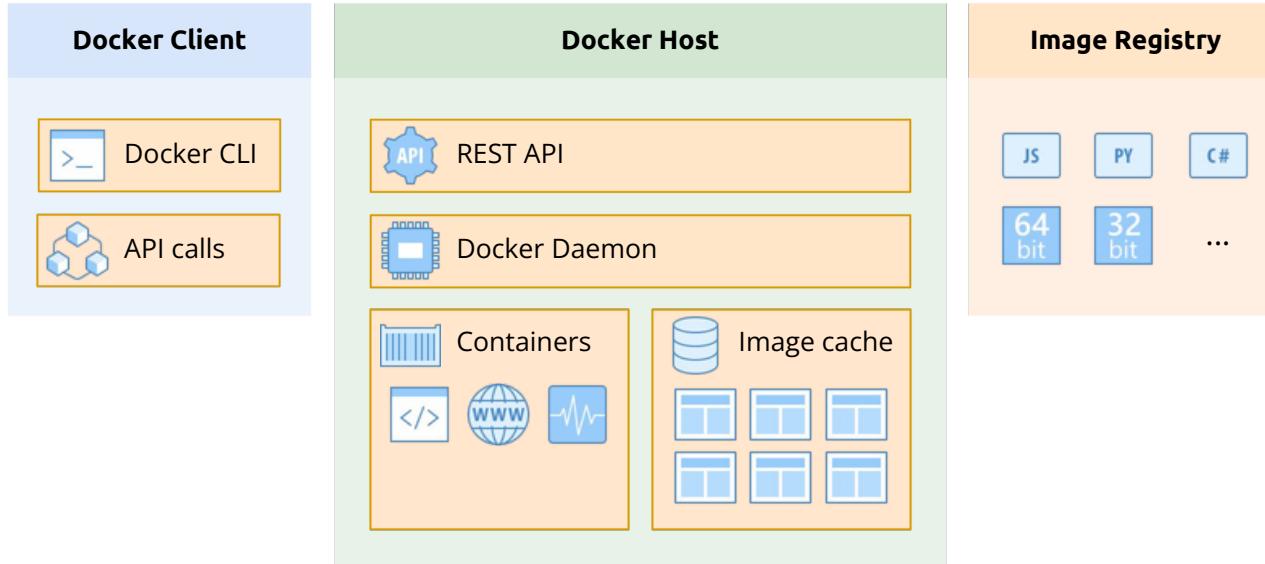
Feature	Virtual Machines (VMs)	Docker Containers
Isolation	Strong isolation: Each VM has its own OS, providing complete isolation.	Process-level isolation: Containers share the host OS kernel.
Size/Overhead	Larger: VMs have a larger footprint due to the guest OS and virtual hardware.	Lightweight: Containers have minimal overhead, as they share the kernel.
Portability	Less portable: VMs can be tied to specific hypervisors and guest OS configurations.	Highly portable: Containers are platform-agnostic and run consistently.
When to use	<ul style="list-style-type: none">▪ You need strong isolation between different environments.▪ You're dealing with legacy applications that might not be easily containerized.▪ You want to replicate a complete system environment for testing or development.	
Docker	<ul style="list-style-type: none">▪ You're building modern, cloud-native applications using microservices architecture.▪ You need to scale your applications quickly and efficiently.▪ Portability across different environments is a top priority.	

Docker

Docker Components

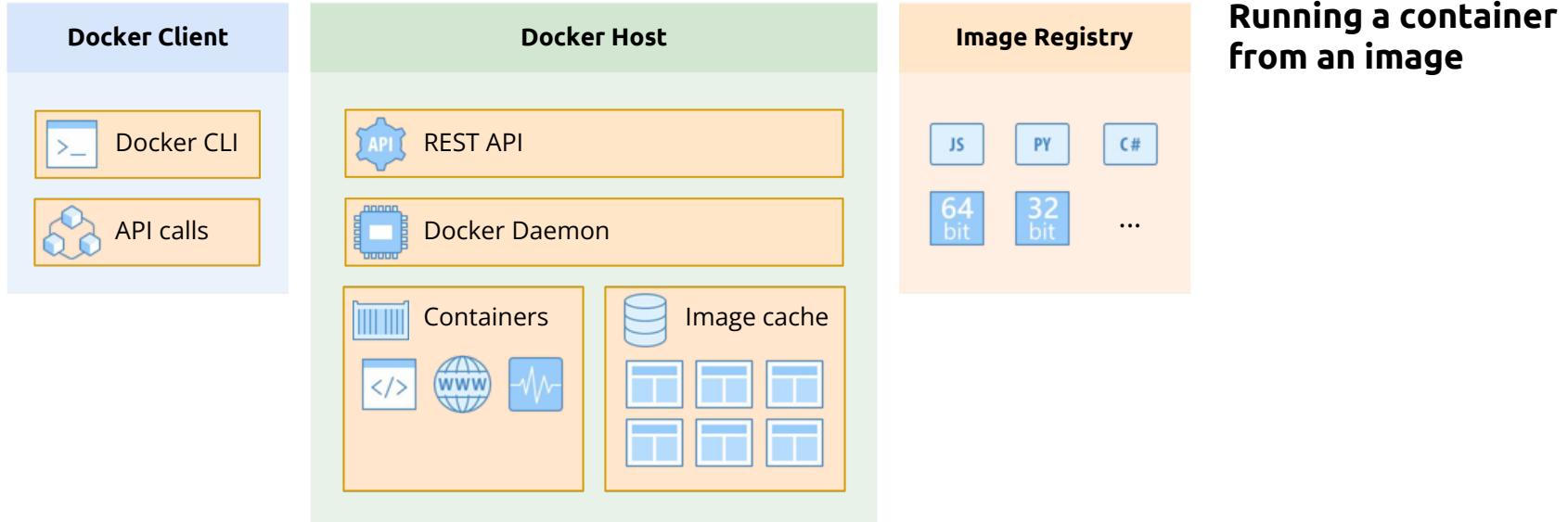
Docker Components

What are the different parts in a Docker-based system?



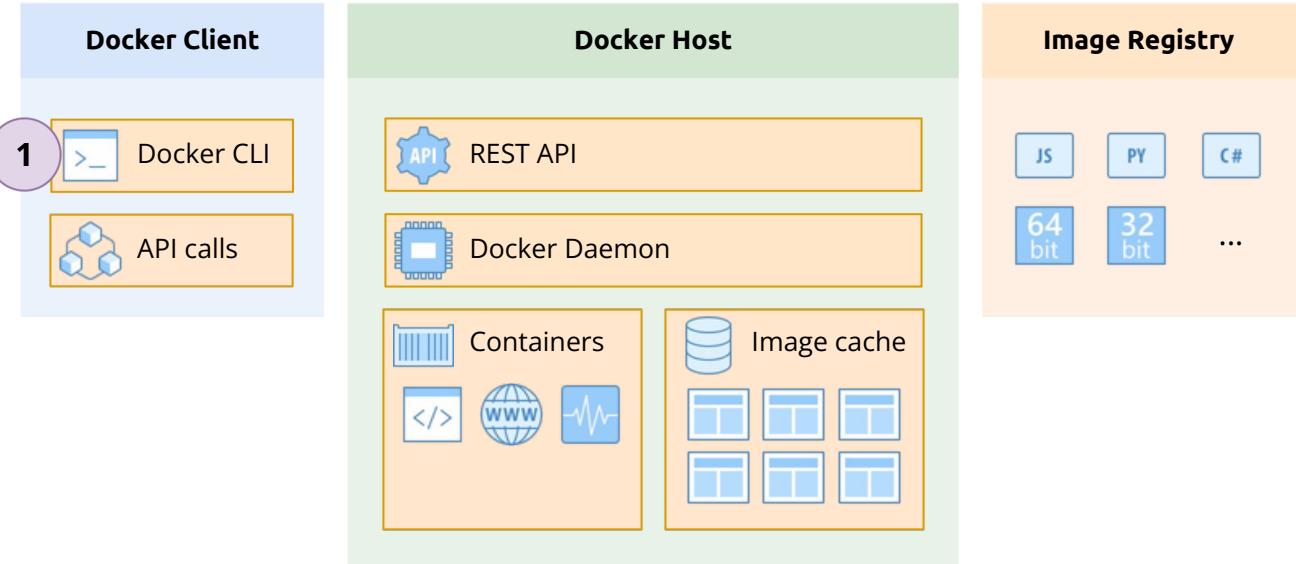
Docker Components

What are the different parts in a Docker-based system?



Docker Components

What are the different parts in a Docker-based system?

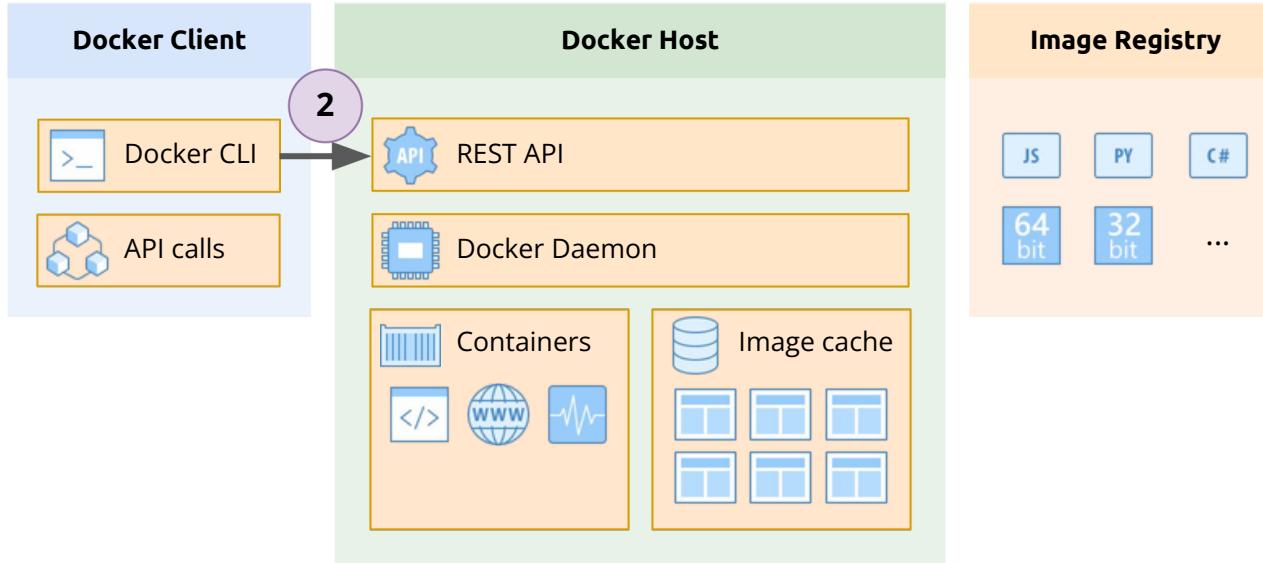


Running a container from an image

1. Issue `docker run` in the CLI.

Docker Components

What are the different parts in a Docker-based system?

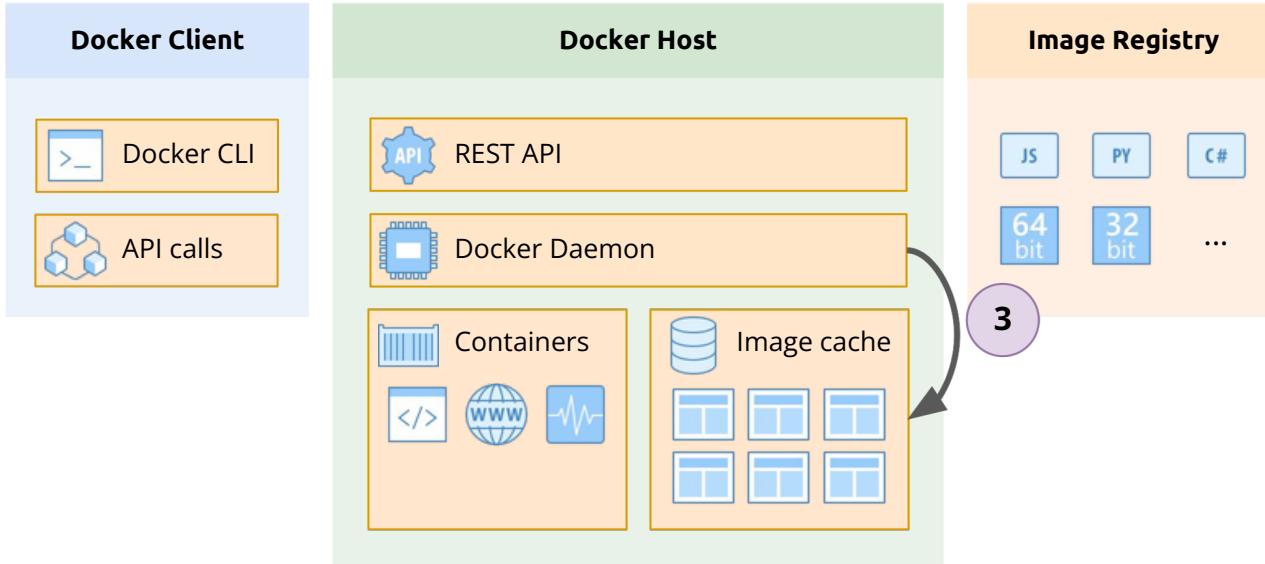


Running a container from an image

- 1.Issue `docker run` in the CLI.
- 2.CLI sends a request to the host's REST API.

Docker Components

What are the different parts in a Docker-based system?

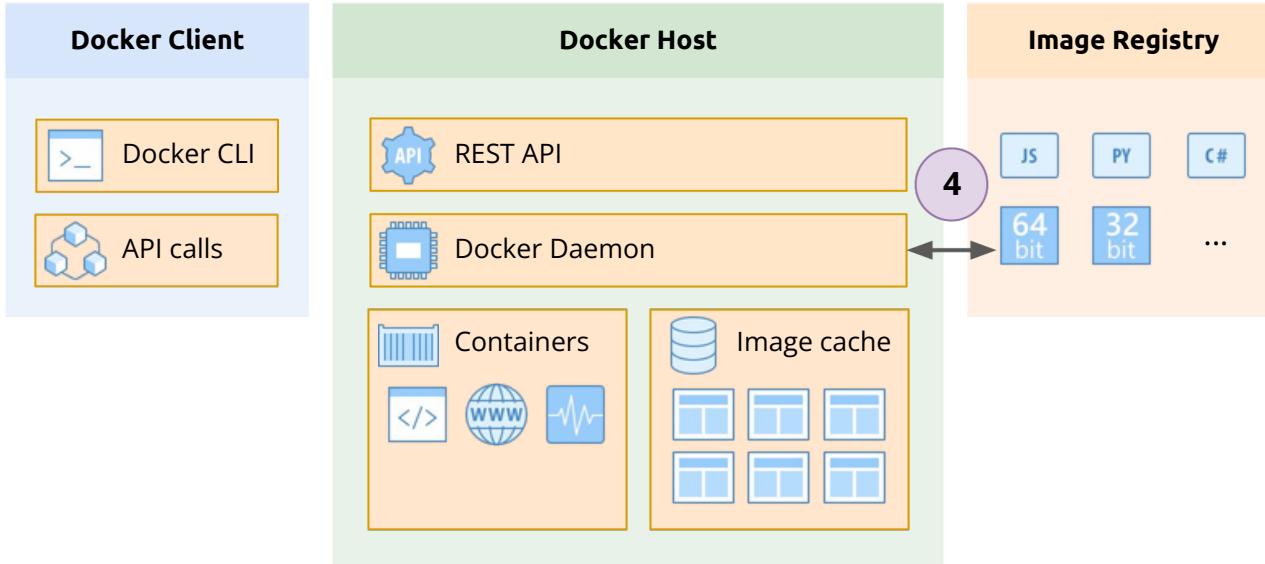


Running a container from an image

- 1.Issue `docker run` in the CLI.
- 2.CLI sends a request to the host's REST API.
- 3.Docker Host checks if the image is present in the local cache.

Docker Components

What are the different parts in a Docker-based system?

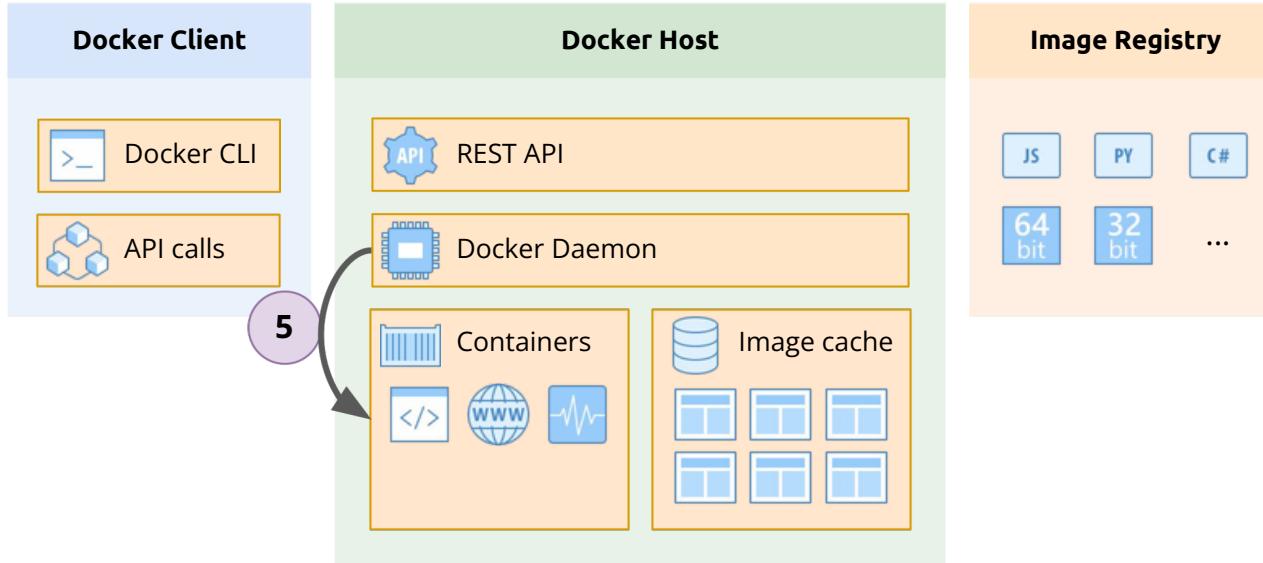


Running a container from an image

- 1.Issue `docker run` in the CLI.
- 2.CLI sends a request to the host's REST API.
- 3.Docker Host checks if the image is present in the local cache.
- 4.If not, it downloads it from the image registry.

Docker Components

What are the different parts in a Docker-based system?

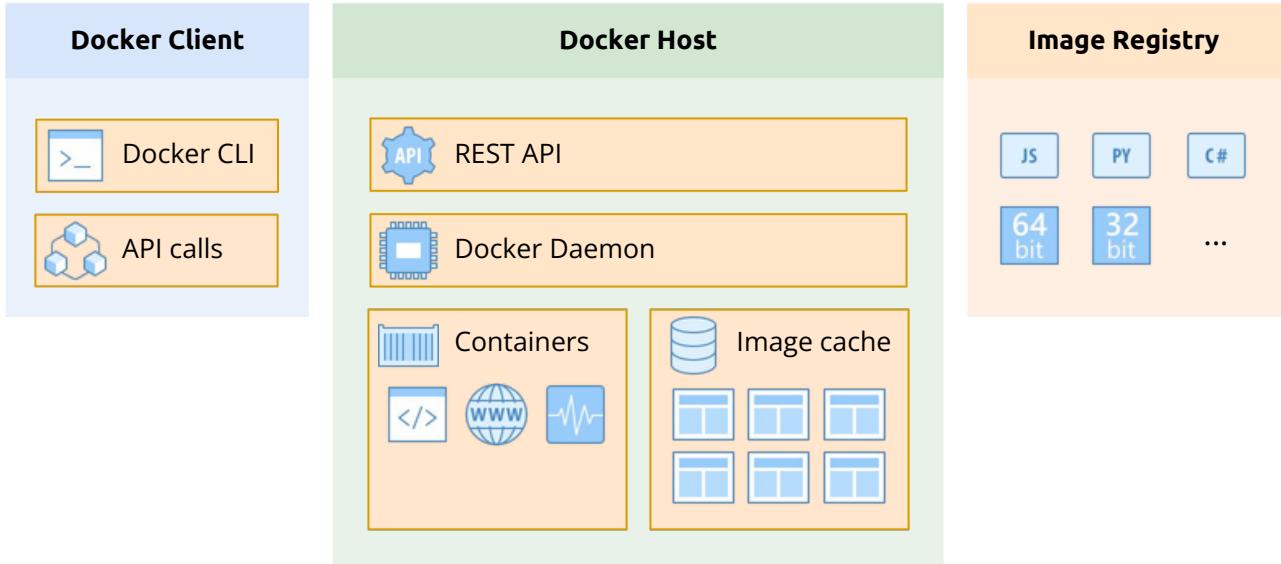


Running a container from an image

- 1.Issue `docker run` in the CLI.
- 2.CLI sends a request to the host's REST API.
- 3.Docker Host checks if the image is present in the local cache.
- 4.If not, it downloads it from the image registry.
- 5.Docker Host instantiates a new container based on the image.

Docker Components

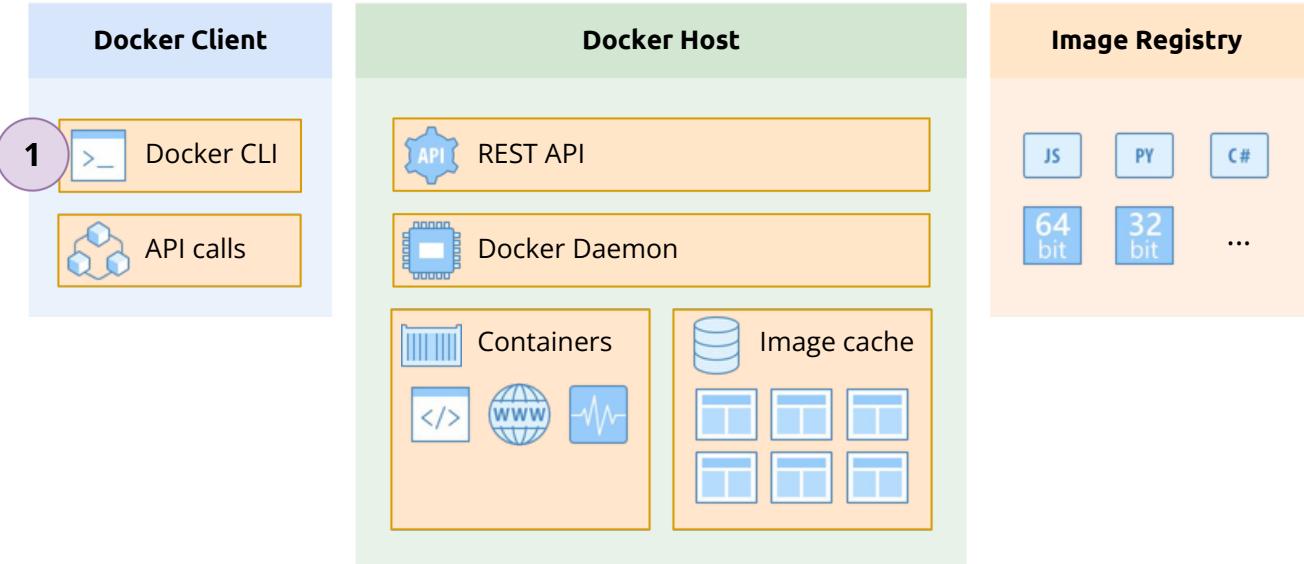
What are the different parts in a Docker-based system?



Building and pushing an Image

Docker Components

What are the different parts in a Docker-based system?

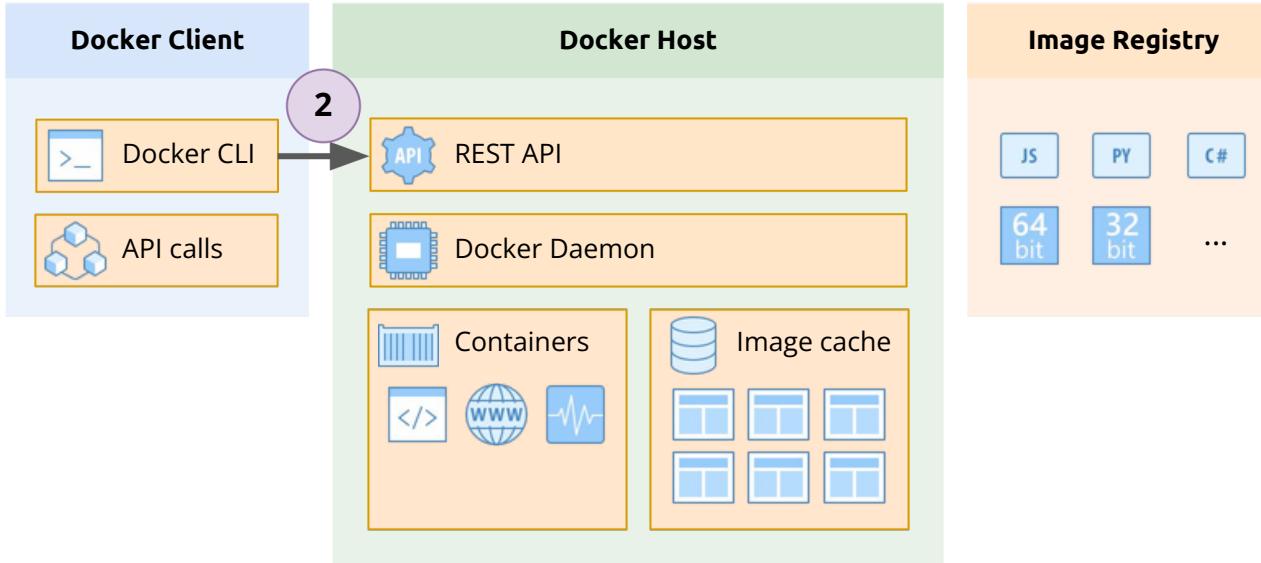


Building and pushing an Image

1. Issue `docker build` in the CLI.

Docker Components

What are the different parts in a Docker-based system?

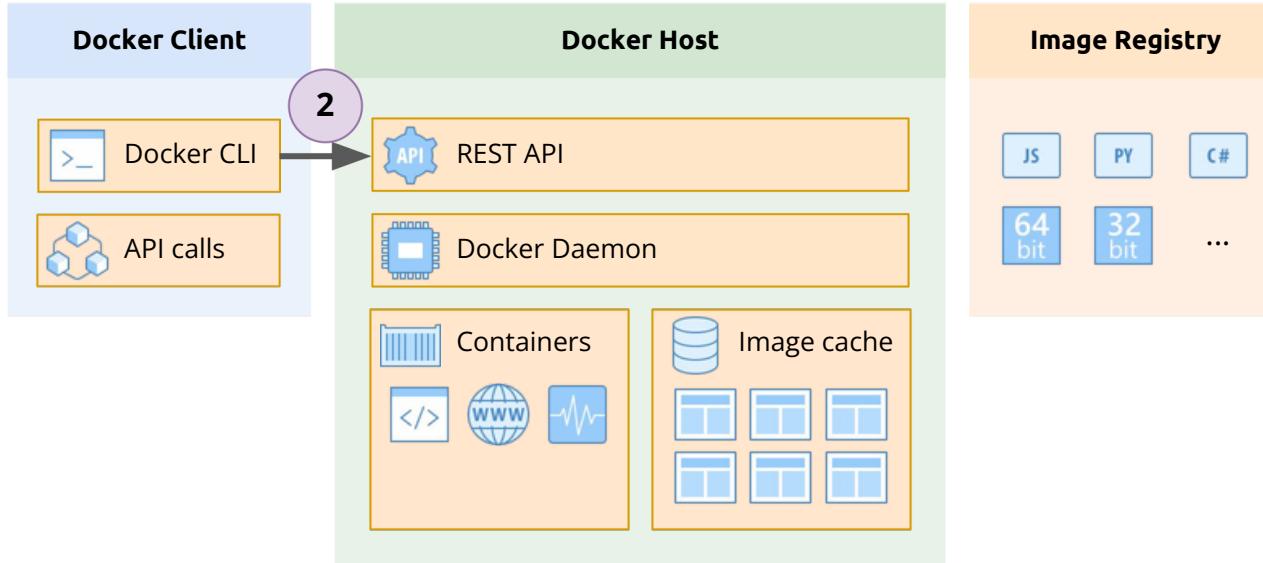


Building and pushing an Image

- 1.Issue `docker build` in the CLI.
- 2.CLI sends a request to the host's REST API.

Docker Components

What are the different parts in a Docker-based system?

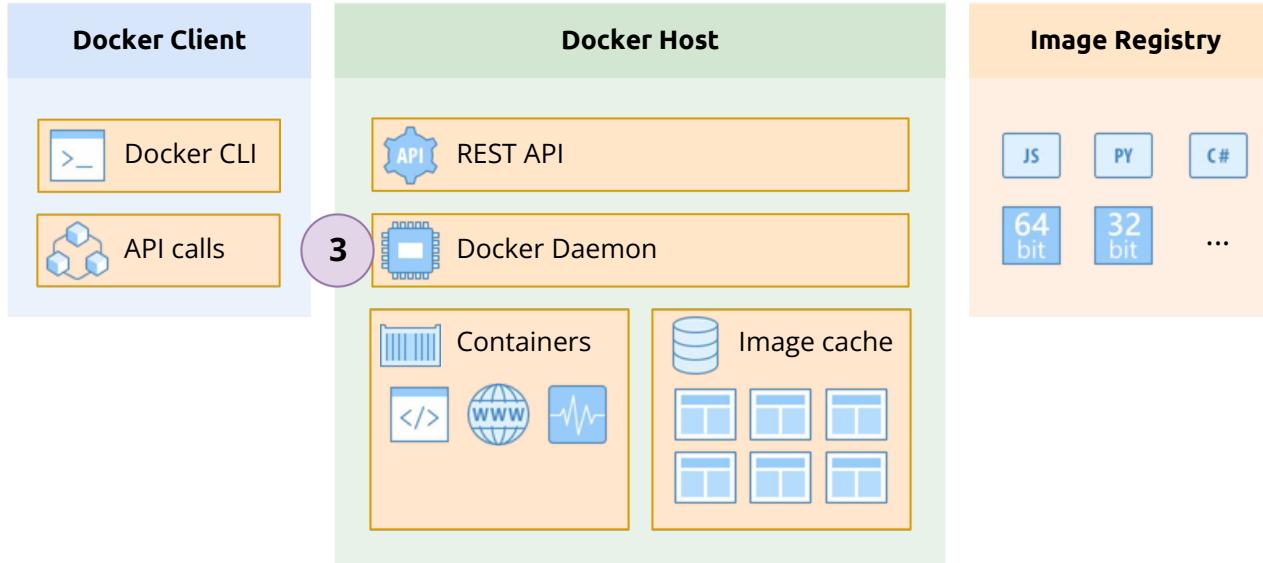


Building and pushing an Image

- 1.Issue `docker build` in the CLI.
- 2.CLI sends a request to the host's REST API.
 - a. This also includes the respective Dockerfile and context.

Docker Components

What are the different parts in a Docker-based system?

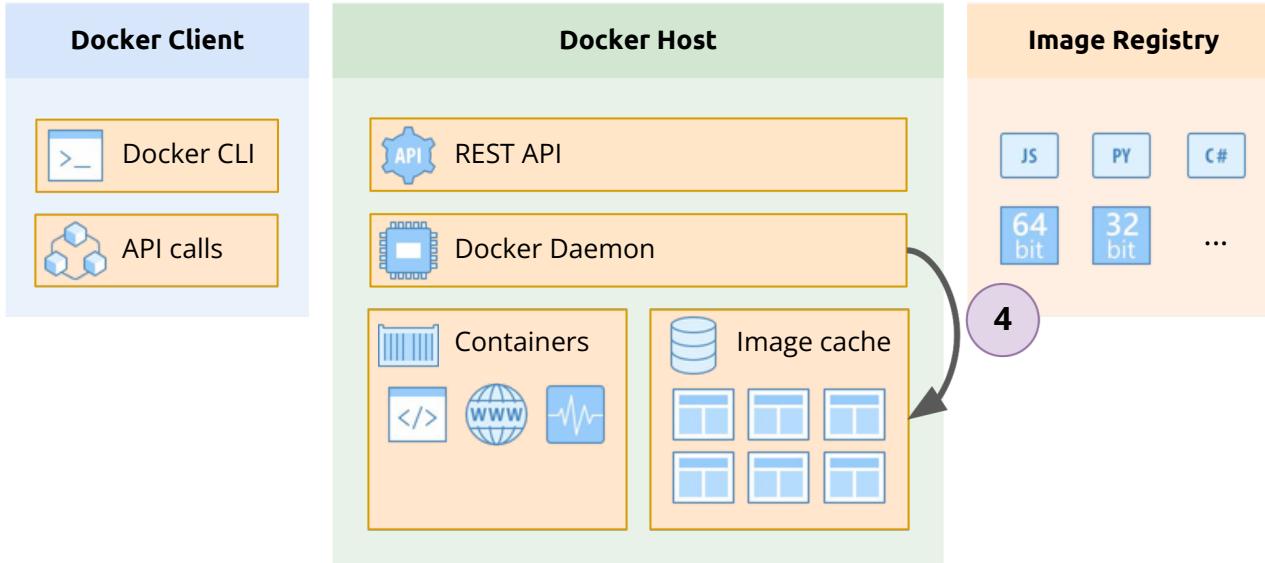


Building and pushing an Image

- 1.Issue `docker build` in the CLI.
- 2.CLI sends a request to the host's REST API.
 - a. This also includes the respective Dockerfile and context.
- 3.Docker Host builds the image according to the Dockerfile.

Docker Components

What are the different parts in a Docker-based system?

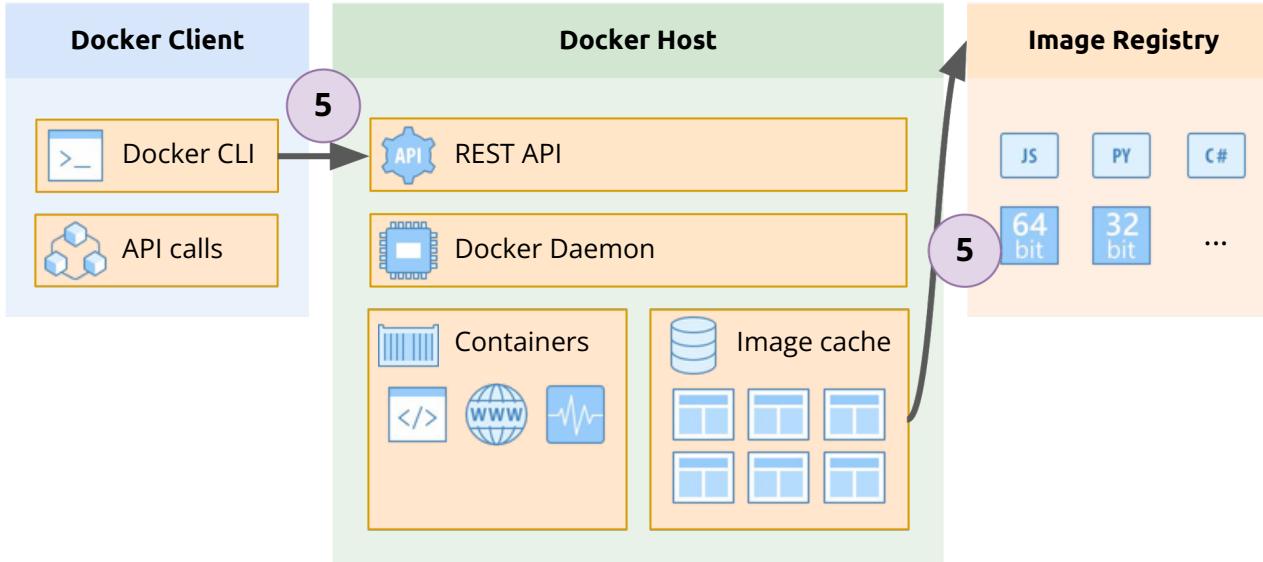


Building and pushing an Image

- 1.Issue `docker build` in the CLI.
- 2.CLI sends a request to the host's REST API.
 - a. This also includes the respective Dockerfile and context.
- 3.Docker Host builds the image according to the Dockerfile.
- 4.Docker Host tags the image and stores it locally.

Docker Components

What are the different parts in a Docker-based system?



Building and pushing an Image

- 1.Issue `docker build` in the CLI.
- 2.CLI sends a request to the host's REST API.
 - a. This also includes the respective Dockerfile and context.
- 3.Docker Host builds the image according to the Dockerfile.
- 4.Docker Host tags the image and stores it locally.
- 5.Issue `docker push` command on the CLI.

Docker Installing Tools

Docker Running Containers

Running Containers

Section overview

1

Run your first container

2

Understand the container lifecycle

- 1.Discuss the different states a container can be in
- 2.Understand which CLI commands result in which states

3

Explore the Docker CLI

- 1.Demonstrate essential Docker CLI commands for managing containers and images.
- 2.Explain container behavior and Docker's management of short-lived vs. persistent containers.
- 3.Demonstrate commands for logging, running commands within containers, and executing shells.
- 4.Introduce the Docker build process by creating a simple Dockerfile and building a custom Docker image.

4

Get help with the CLI

Pull an image: docker pull nginx

Run a container: docker run nginx

Run in detached mode: docker run -d nginx

List running containers: docker ps

Stop a container: docker stop <container_name>

Kill a container forcefully: docker kill <container_name>

Run with port mapping: docker run -d -p 8080:80 --name webserver nginx

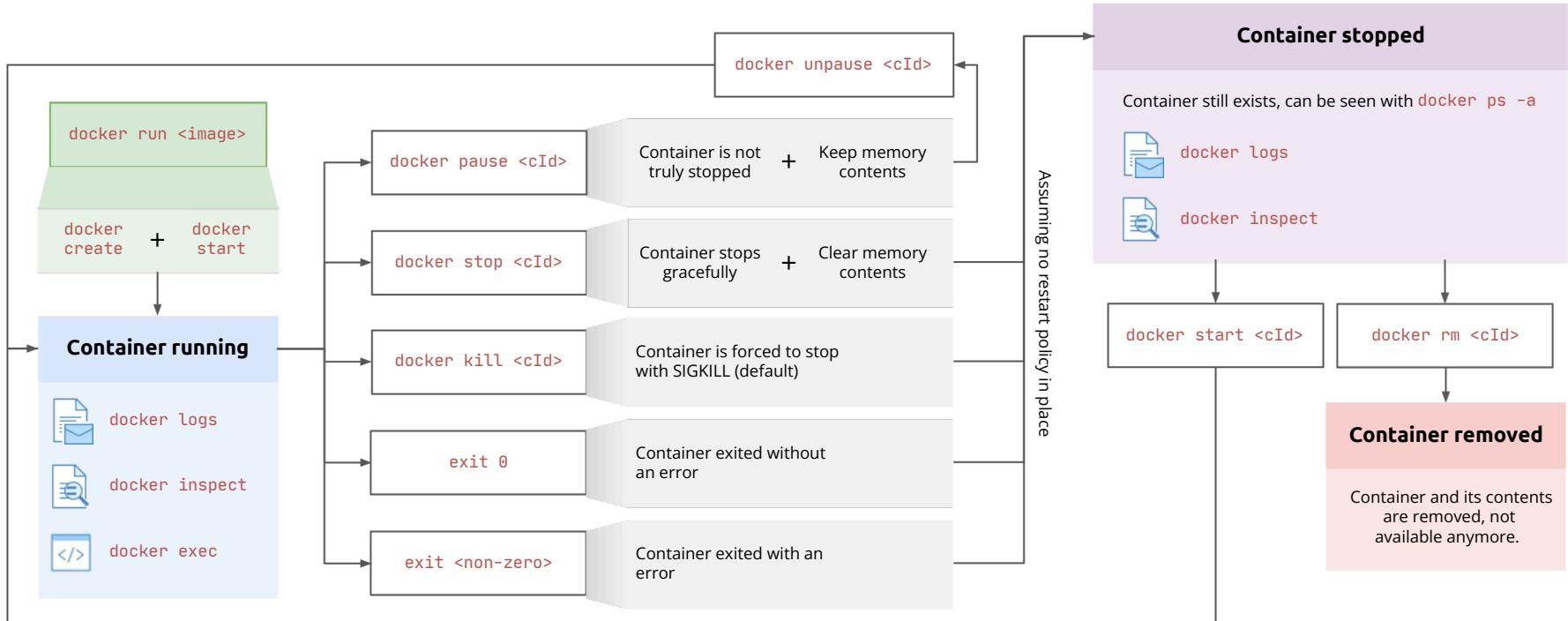
Check response via curl: curl http://localhost:8080

Access web page in a browser: http://localhost:8080

Docker Container Lifecycle

Container Lifecycle

Understand what happens with containers throughout their lifetime



State	Description	Command(s)
Created	A new container is created but not started yet.	<code>docker create <image></code>
Running	The container is actively running.	<code>docker run <image></code> or <code>docker start <container_id></code>
Paused	The container is frozen but not stopped.	<code>docker pause <container_id></code>
Unpaused	Resumes a paused container.	<code>docker unpause <container_id></code>
Stopped (Graceful Exit)	The container stops naturally or via a stop signal.	<code>docker stop <container_id></code>
Killed (Force Exit)	The container is abruptly stopped (risk of data loss).	<code>docker kill <container_id></code>
Exited (Successful)	The container has completed execution without issues.	<code>docker ps -a</code> (to check)
Exited (Error)	The container stopped due to an error (non-zero exit code).	<code>docker ps -a</code> , check logs with <code>docker logs <container_id></code>
Restarted	A stopped container is started again.	<code>docker start <container_id></code>
Removed	The container is deleted from the system.	<code>docker rm <container_id></code>

Key Takeaways

`docker run` is a shortcut for creating and starting a container.

Containers can be paused and unpaused without stopping them.

Containers exit normally (exit code 0) or due to an error (non-zero exit code).

Stopped containers still exist and can be restarted (`docker start`).

Removing a container (`docker rm`) permanently deletes it.

Restart policies can automatically restart failed containers (covered later in the course).

Docker CLI

Command	Description	Example Usage
<code>docker images</code>	List all locally stored images	<code>docker images</code>
<code>docker ps -a</code>	Show all containers (running and stopped)	<code>docker ps -a</code>
<code>docker pull <image></code>	Download an image from Docker Hub	<code>docker pull ubuntu</code>
<code>docker run <image></code>	Run a container (creates & starts it)	<code>docker run nginx</code>
<code>docker start <container_id></code>	Restart an existing container	<code>docker start abc123</code>
<code>docker stop <container_id></code>	Gracefully stop a container	<code>docker stop abc123</code>
<code>docker rm <container_id></code>	Remove a stopped container	<code>docker rm abc123</code>
<code>docker ps --filter "name=<name>"</code>	Filter containers by name	<code>docker ps --filter "name=web_server"</code>
<code>docker ps -q</code>	Get only container IDs (quiet mode)	<code>docker ps -q</code>
<code>docker stop \$(docker ps -q)</code>	Stop all running containers	<code>docker stop \$(docker ps -q)</code>
<code>docker rm \$(docker ps -aq)</code>	Remove all stopped containers	<code>docker rm \$(docker ps -aq)</code>

Docker CLI

Checking Images & Containers

Command	Description	Options / Explanation
docker images	Lists all locally stored images	Shows only downloaded images, not images available on Docker Hub
docker image rm <image_id>	Removes an image from local storage	If an image is removed and later needed, Docker re-downloads it
docker ps	Shows only running containers	Does not display stopped containers
docker ps -a	Shows all containers (running + stopped)	Useful for inspecting exited containers

Running & Managing Containers

Command	Description	Options / Explanation
docker run -d -p 80:80 --name web_server nginx	Runs a new Nginx container in detached mode with port mapping	-d (detached mode), -p 80:80 (map host port 80 to container port 80), --name web_server (custom container name)
docker stop <container_name>	Gracefully stops a running container	Allows a proper shutdown, preventing data corruption
docker rm <container_name>	Removes a stopped container	Frees system resources

Viewing Container Logs

Command	Description	Options / Explanation
<code>docker logs <container_name></code>	Viewing Container Logs	Useful for debugging without real-time monitoring
<code>docker logs -f <container_name></code>	Follows container logs live	-f (follow) keeps streaming logs in real time
<code>curl http://localhost</code>	Sends a request to the running Nginx container	Triggers a log entry for GET requests

Executing Commands Inside a Running Container

Command	Description	Options / Explanation
<code>docker exec -it <container_name> sh</code>	Opens an interactive shell inside the container	-it (interactive mode), sh (default shell for Alpine-based images)
<code>docker exec -it <container_name> /bin/bash</code>	Opens an interactive Bash shell	Use Bash if available (e.g., Ubuntu-based containers)
<code>ls</code>	Lists files in the current directory	Standard Linux command
<code>echo "Hello World"</code>	Prints "Hello World" to the terminal	Useful for testing shell access
<code>ls /usr/share/nginx/html</code>	Shows the directory where Nginx serves files	Location of the default index.html
<code>cat /usr/share/nginx/html/index.html</code>	Displays the content of the index.html file	Confirms what is served on the Nginx web server

Creating Custom Docker Images with a Dockerfile

Command	Description	Options / Explanation
vim Dockerfile	Creates a new Dockerfile	You can use any text editor (VS Code, Nano, Vim)
Dockerfile Contents:	Defines the base image and command to execute	FROM ubuntu:latest (Base OS), RUN echo "Hello from my first Docker image" (Prints message)
docker build -t my_first_image .	Builds a custom Docker image	-t (tag name for the image), . (build context = current directory)
docker images	Lists all images, including newly built ones	Confirms if the new image was successfully created
docker run my_first_image	Runs a container from the new custom image	Executes the echo command inside the container
Expected Output:	Hello from my first Docker image	This confirms the custom image was built correctly

Docker

Working with Images

Working with Images

Section overview

1 Understand Docker images and their purpose

2 Explore container registries

- 1.Understand the purpose of container registries, and the different options available
- 2.Learn how to browse images in Docker Hub

3 Manage images with the CLI

4 Create Dockerfiles and use them to build images

5 Clarify the difference between images and containers

Docker

Docker Images

Docker Images

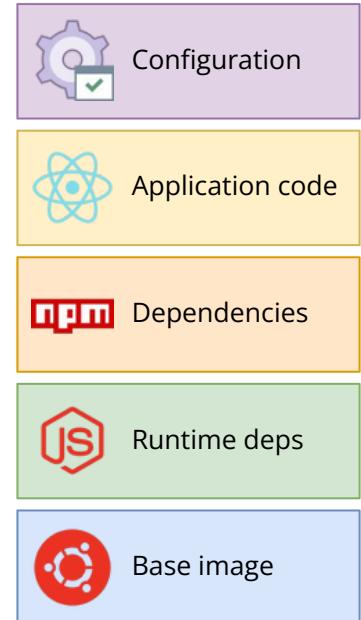
Understanding the DNA of our containers

Images are a self-contained, read-only template that encapsulates everything needed to run your application:

- **The base layer:** often a minimal Linux distribution like Alpine, or a more full-fledged one like Ubuntu.
- **Runtime Environment:** Specific software (e.g., Python, Node.js) required by your application.
- **Libraries & Dependencies:** All the external code your application relies on.
- **Application Code:** Your own source code or compiled binaries.
- **Configuration:** Settings for your application and its environment.

A Docker image is like a snapshot of your application and its complete runtime environment, frozen in time and ready to be brought to life as a container. Images can be sourced from multiple locations:

- **Docker Hub:** This is the official Docker image repository, offering a vast collection of images for various purposes.
- **Private Registries:** For organizations or individuals with proprietary software, private registries provide secure storage for custom images.
- **Building Your Own Images:** This is where the real power of Docker shines. By writing Dockerfiles, you can create images that perfectly match your application's requirements.



Docker Container Registries

Container Registries

Storing and managing Docker images

- Container registries offer a multitude of benefits:
 - **Collaboration:** Share your images with teammates, clients, or the wider community.
 - **Versioning:** Track different versions of your images for easy rollback and updates.
 - **Security:** Private registries provide a secure environment for storing sensitive images.
 - **Automation:** Automate image building and deployment as part of your CI/CD pipeline.
- Types of Container Registries:
 - **Public Registries:** Open to everyone and host a vast collection of images from various sources. Docker Hub is the most prominent example.
 - **Private Registries:** Used for storing proprietary or sensitive images and offer granular access control.



Docker

Dockerfile

What is a Dockerfile?

A Dockerfile is a script that defines step-by-step instructions to build a Docker image. These instructions are executed sequentially from top to bottom and are responsible for setting up the dependencies, environment, and configurations required for a containerized application.

The first instruction must always be FROM, which specifies the base image to use.

Each subsequent instruction adds configurations, dependencies, or file modifications.

The order of instructions is important, as each step creates a new image layer.

Structure of a Dockerfile

Instruction	Description	Options / Explanation
FROM <image>	Specifies the base image to use	The first line of every Dockerfile. Example: FROM ubuntu:latest
RUN <command>	Executes a command during the image build process	Example: RUN apt-get update && apt-get install -y curl
COPY <src> <dest>	Copies files from the host machine to the container	Example: COPY app /usr/src/app
ADD <src> <dest>	Similar to COPY but also extracts compressed files	Example: ADD archive.tar.gz /data/
WORKDIR <dir>	Sets the working directory inside the container	Example: WORKDIR /usr/src/app
CMD ["command", "arg1"]	Defines the default command that runs when the container starts	Example: CMD ["node", "app.js"]
ENTRYPOINT ["command", "arg1"]	Sets a command that always executes when the container starts	Works similarly to CMD but is less flexible
EXPOSE <port>	Documents which port the container listens on	Example: EXPOSE 8080
ENV <key>=<value>	Defines environment variables inside the container	Example: ENV NODE_ENV=production

Kubernetes

10000-Foot Overview

10000-Foot Overview

Section overview

1 What is Kubernetes? And why use it?

2 Discuss Kubernetes' architecture

1. Control plane vs. data plane

2. Nodes and Kubernetes objects

3 Introduce the Kubernetes CLI: `kubectl`

1. Differentiate `kubectl` and the Kubernetes cluster

2. Understand how `kubectl` communicates with the cluster

3. Discuss the structure of `kubectl` commands

Kubernetes

Managing containers at scale

Managing containers at scale

Can't we get by with just Docker?

- Docker and Docker Compose alone are enough for running individual containers or creating development environments, but they do not meet the requirements of production workloads.

Challenge	Docker	Kubernetes
Container Scheduling	 Manual assignment of containers to servers  Time consuming and error-prone	 Declarative definition of criteria for node (server) assignment  Handled automatically by Kubernetes based on resource definitions
Load Balancing	 No built-in load balancing	 Services provide built-in load balancing across Pods
Scaling Applications	 Lacks automation for horizontal scaling  Requires low-level adjustments for directing load to newly created containers	 Horizontal Pod Autoscaler can scale pods horizontally and automatically through real-time metrics
Self-Healing	 Lacks automation for detecting and restarting unhealthy workloads	 Health checks and continuous monitoring allow for considerably more robust self-healing mechanisms

Kubernetes

Managing containers at scale

Can't we get by with just Docker?

- Docker and Docker Compose alone are enough for running individual containers or creating development environments, but they do not meet the requirements of production workloads.

Challenge	Docker	Kubernetes
Service Discovery	✖ No built-in solution for service discovery at scale	✓ Internal DNS allows for robust service discovery mechanisms through Services
Configuration Management	✖ Requires low-level configuration and adjustments, and can become cumbersome for larger applications and multiple environments	✓ Configuration maps and secrets allow decoupling and scaling of application configuration

Kubernetes

Kubernetes

What is Kubernetes?

What is Kubernetes?

Understanding the goal, features, and architecture of Kubernetes

- **Open-source** tool, which has become the **de facto standard for container orchestration**.
 - Kubernetes is designed to be easily extensible, and as a result the K8s ecosystem has grown to thousands of different tools.
- **Goal:** address the complexities of **running containers in production**. As such, the tool is designed around:

Automation	Declarative	Scalability	Resilience
Reduces the need for manual intervention in deploying and managing applications.	Declare the final, desired state and let Kubernetes figure out how to get there.	Effortlessly scales applications horizontally to meet demand.	Ensures that applications are always available, even in the face of failures.

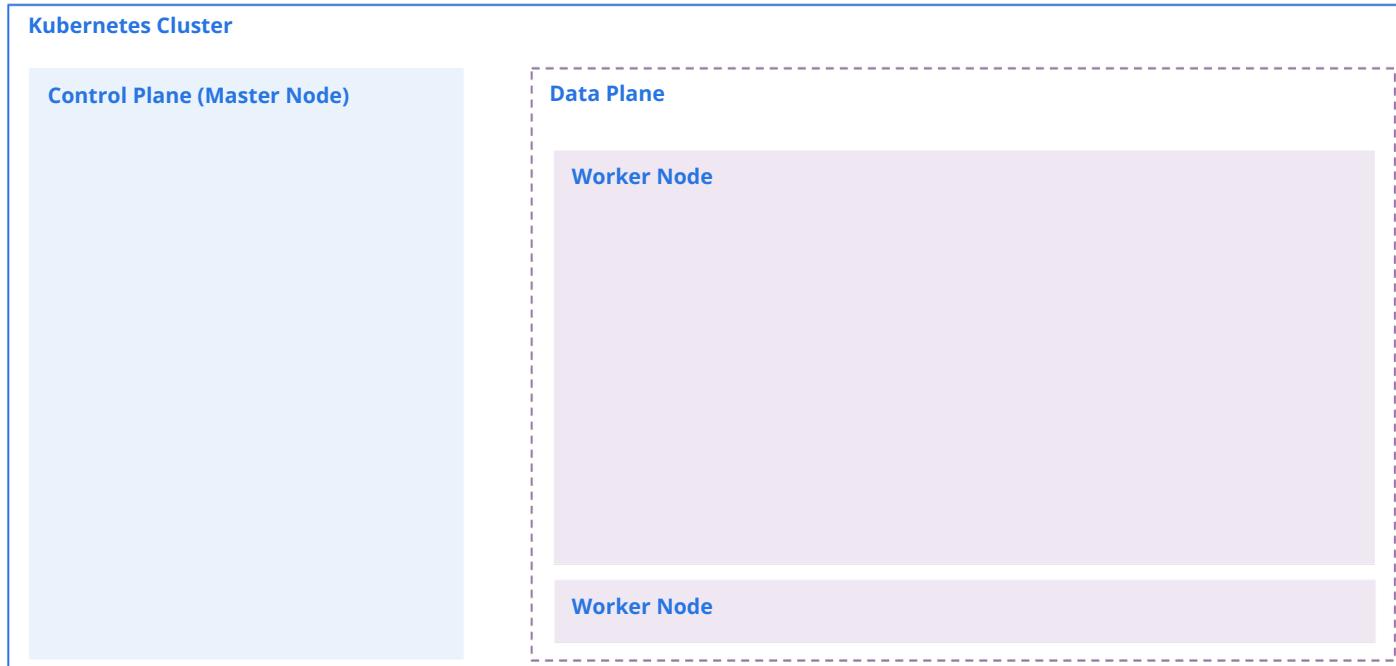
- A Kubernetes cluster has many components, and they are either placed in the control plane or in worker nodes:
 - **Control plane:** it's the brain of the cluster, containing the components responsible for managing the entire system.
 - **Worker nodes:** they are the machines where your applications actually run.

Kubernetes

Kubernetes Architecture

Kubernetes Architecture

Understanding the main components of a Kubernetes Cluster



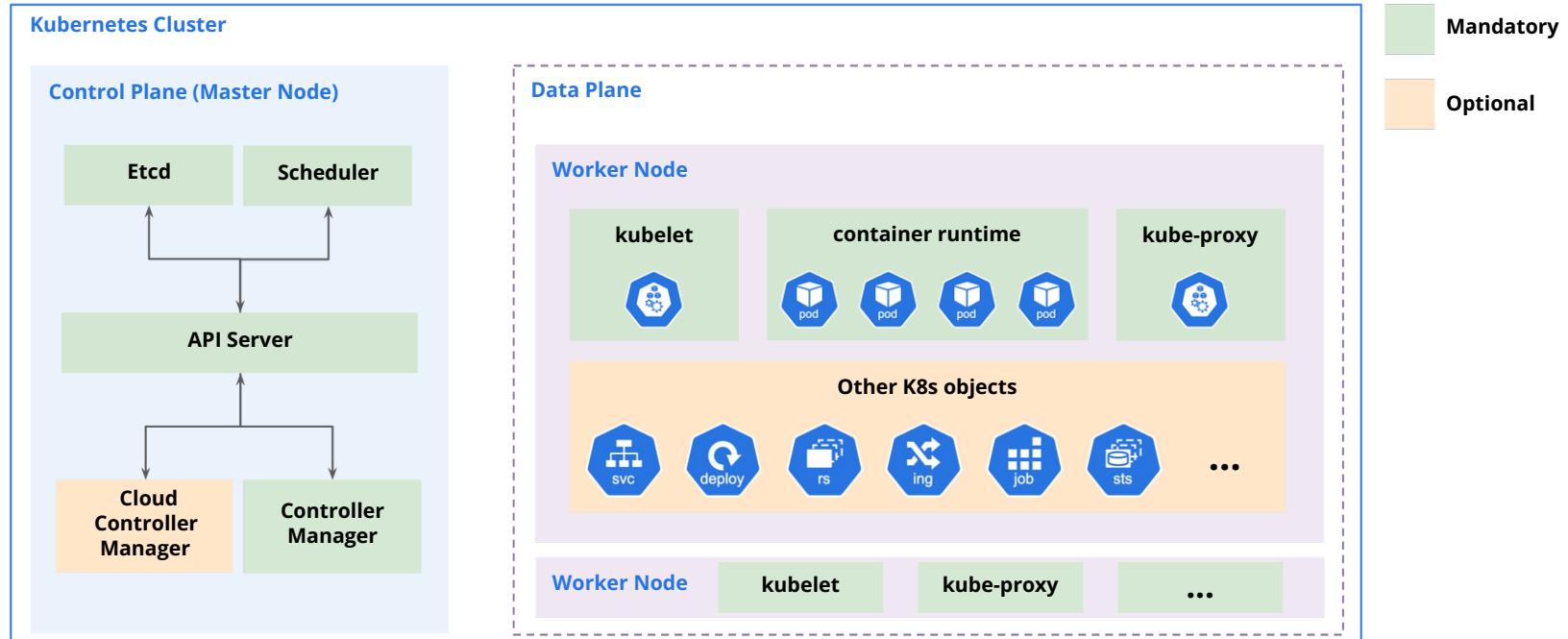
Kubernetes Architecture

Understanding the main components of a Kubernetes Cluster



Kubernetes Architecture

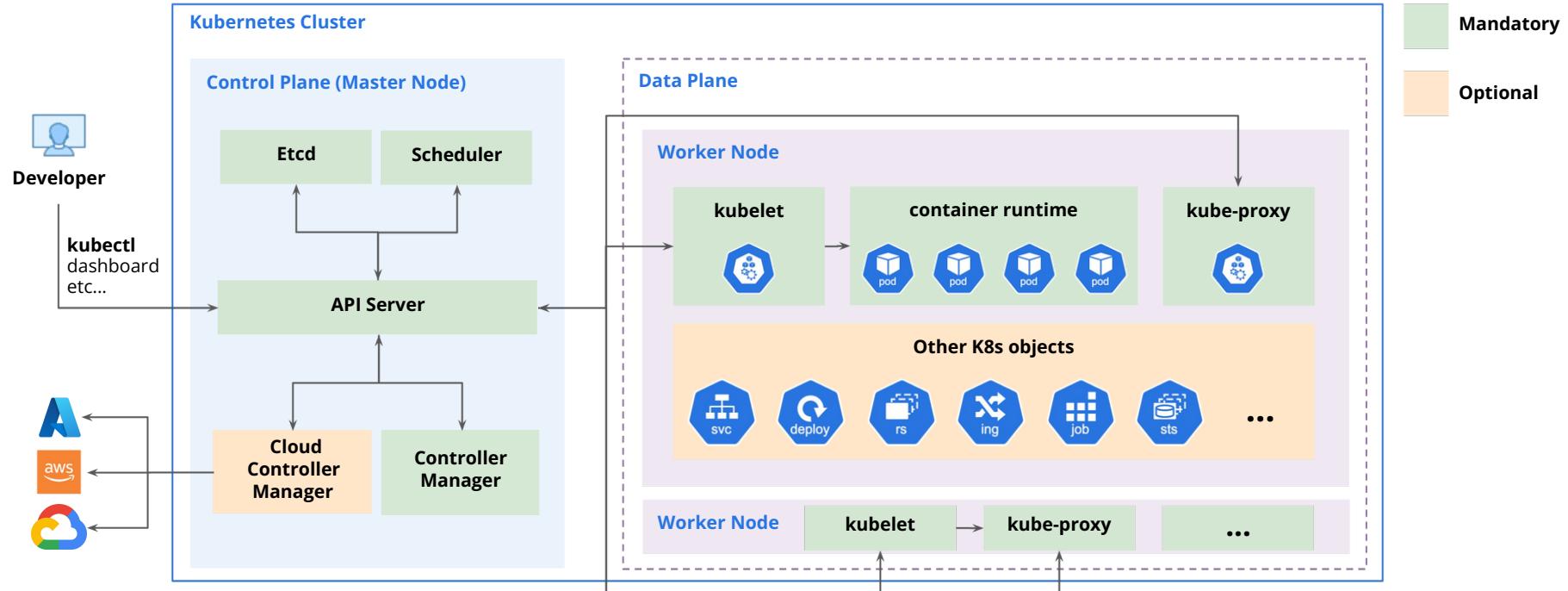
Understanding the main components of a Kubernetes Cluster



Kubernetes

Kubernetes Architecture

Understanding the main components of a Kubernetes Cluster



Kubernetes

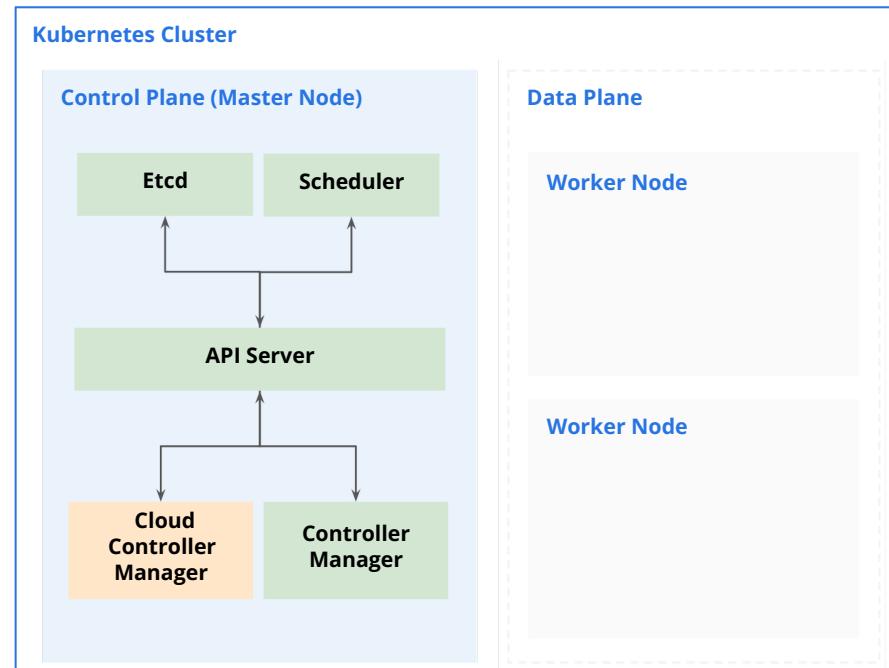
Kubernetes

The Control Plane

The Control Plane

Taking a closer look at the brains of Kubernetes

- **API Server:** exposes the Kubernetes API, which is the main entry point for all administrative tasks (kubectl, dashboard, etc.).
- **Scheduler:** responsible for placing pods on the most suitable nodes. It selects nodes based on resource availability and other constraints.
- **Controller Manager:** responsible for running controller processes that handle routine tasks (e.g. ReplicaSet controller, node controller, job controller, etc.).
- **Etcd:** distributed key-value store that stores all cluster data, including the configuration and state of the cluster. Acts as the single source of truth for the cluster's state.
- **Cloud Controller Manager:** enables Kubernetes to interact with the underlying cloud infrastructure. It handles tasks such as managing cloud-based load balancers, persistent storage, and node management.



Kubernetes

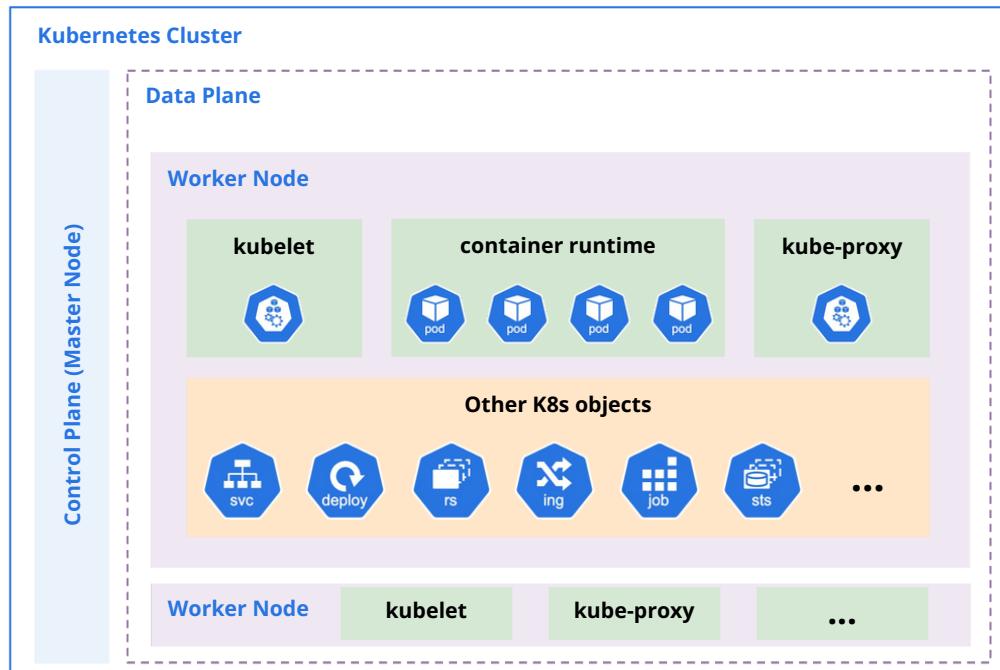
The Worker Nodes

The Worker Nodes

Taking a closer look at where containers are actually run

- **kubelet:** agent that runs on each worker node and communicates with the control plane. It ensures that containers are running in the pods as defined by the pod specifications.

- **Container Runtime:** responsible for running the containers on each worker node. Kubernetes supports several runtimes, such as Docker, containerd, or CRI-O.
- **kube-proxy:** manages network rules on each worker node. It maintains network connectivity and load balancing for services, ensuring that requests are routed to the appropriate pods.
- **Other K8s Objects:** higher-level Kubernetes objects such as ReplicaSets, Deployments, Jobs, Services, StatefulSets, among others.



Kubernetes

The kubectl CLI

The kubectl CLI

Interacting with Kubernetes clusters via the CLI

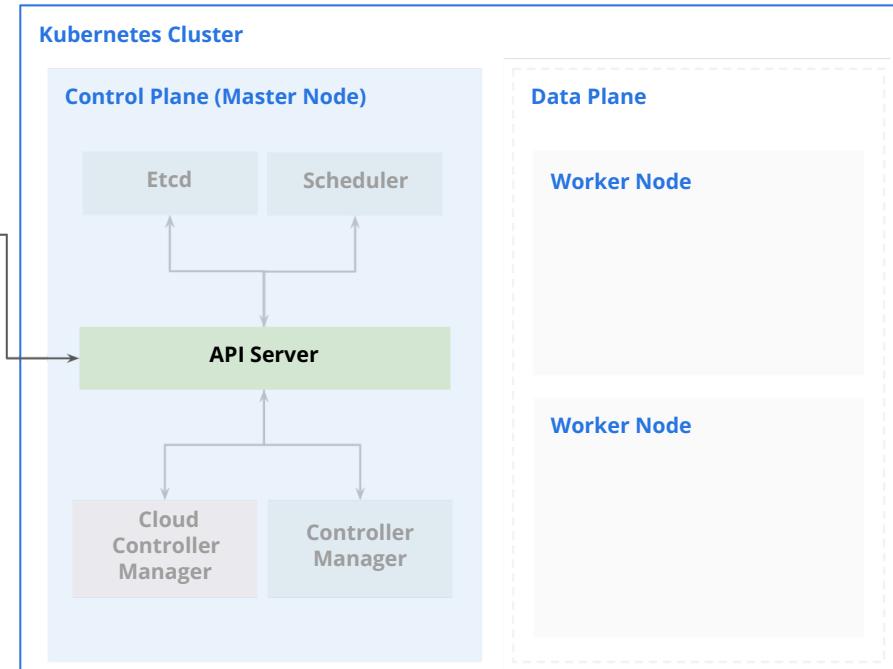
- `kubectl` is the primary command-line utility for interacting with a Kubernetes cluster. It allows you to manage and control Kubernetes resources, such as pods, services, deployments, and more, by sending API requests to the Kubernetes control plane.

```
kubectl [command] [res. type] [name] [flags]
```

Main types of commands / use-cases

- **Inspect cluster resources:** Commands such as `get`, `describe`, and `logs` can be used to describe the state of individual resources or groups of resources.
- **Imperative resource management:** Commands such as `run`, `scale`, `create`, and `delete` can be used to imperatively create, update, and delete resources.
- **Declarative resource management:** Commands such as `apply` and `diff` can be used to declaratively manage resources.

Kubernetes



Using the Alias for kubectl

```
$ alias k=kubectl
```

```
$ k version
```

```
...
```

Create an alias for the kubectl command line tool in your local environment



Use the shortcut to refer to kubectl



Using Auto-Completion for kubectl

```
$ kubectl cre<tab>
```



```
$ kubectl create
```

```
source <(kubectl completion bash) # set up autocomplete in bash into the current shell, bash-completion package should be installed first.  
echo "source <(kubectl completion bash)" >> ~/.bashrc # add autocomplete permanently to your bash shell.
```

Kubernetes

Installing K8s Tooling

Kubernetes

Running Containers in K8s

Running Containers in K8s

Section overview

1 Discuss Pods and their role in Kubernetes

2 Manage Pods in Kubernetes

- 1.Create Pods to run containers
- 2.Retrieve Pod information and delete Pods

3 Expose Pods via Services

- 1.Create services to facilitate communication with Pods
- 2.Understand why services are helpful

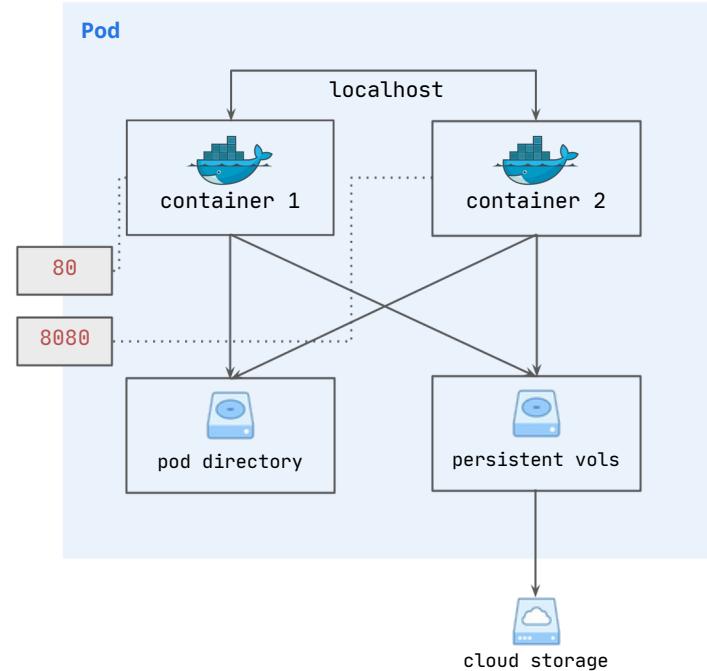
4 Understand how to go from Dockerfile to Pods

Kubernetes Pods

Pods

Meet the smallest and simplest unit that you create to run containers.

- Pods represent a single instance of a running process in your cluster. They:
 - Encapsulate one or more containers.
 - Allow containers to share storage and network resources.
 - Provides multiple options to configure how containers are run (ports, environment variables, volumes, security configuration, among others).
- Pods provide a higher level of abstraction than working directly with containers, allowing multiple containers to work together if necessary.
 - Containers running in the same Pod can communicate with each other via **localhost**.
 - They can also read/write to the same volumes.
- Pods can all communicate with each other by default in Kubernetes.
- We can set health probes in each container so that they are restarted or stop receiving traffic if considered unhealthy.

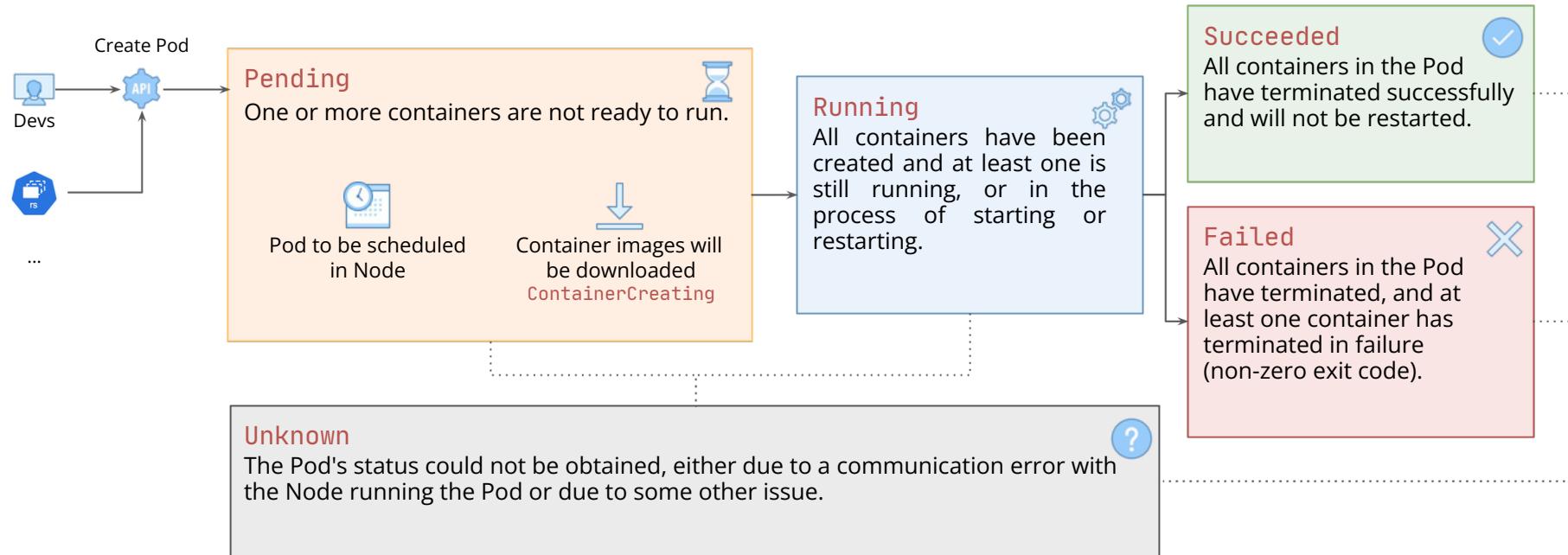


Kubernetes

Kubernetes Pod lifecycle

Pod lifecycle

Understand the different phases of a Pod's lifecycle



Kubernetes Pod Lifecycle

A pod in Kubernetes moves through multiple phases from creation to termination. These states determine whether a pod is scheduled, running, or stopped.

Phase	Description	Additional Information
Pending	The pod is created but not yet running.	It is waiting for a node assignment or image download.
ContainerCreating	The container images are being pulled and started.	This is a sub-phase of Pending.
Running	The pod is active, and at least one container is running.	If multiple containers exist, one must be running for the pod to be considered "Running".
Succeeded	All containers have exited successfully.	This is common for batch jobs that complete their tasks.
Failed	At least one container has terminated with an error.	Indicates a failure in execution.
Unknown	Kubernetes cannot determine the pod state.	Often due to communication issues with the node.

How Kubernetes Handles Container Failures

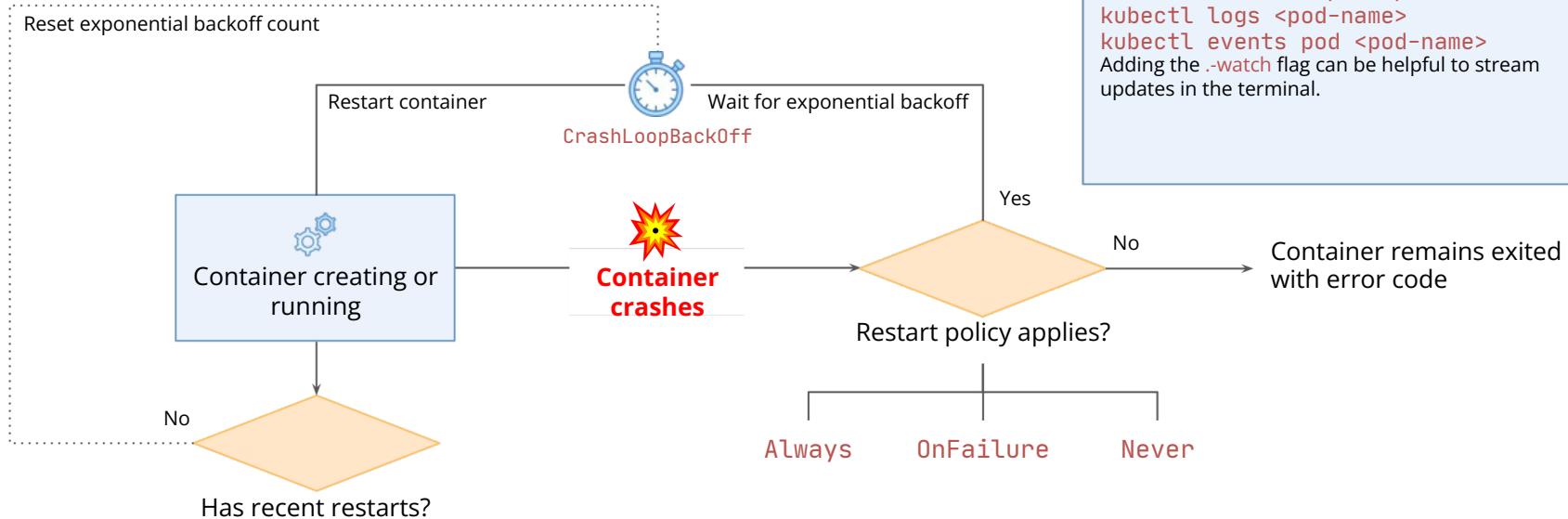
If a container crashes or exits unexpectedly, Kubernetes evaluates its restart policy.

Restart Policy	Behavior
Always	The container is restarted regardless of the exit code.
OnFailure	The container is restarted only if it fails (non-zero exit code).
Never	The container is not restarted, even if it fails.

If no restart policy is set, Kubernetes defaults to Always.
The OnFailure policy applies only to batch jobs or one-time scripts.

Pod lifecycle

Understand how Pods handle container errors



Exponential Backoff & CrashLoopBackOff

When a container repeatedly crashes, Kubernetes increases the delay before restarting it, preventing endless restart loops.

This mechanism is called Exponential Backoff and follows this pattern:

First restart: Instant

Next restarts: Delays increase (few seconds → 1 min → 2 mins → max 5 mins)

If repeated failures continue, the pod enters CrashLoopBackOff status.

The CrashLoopBackOff message indicates:

Kubernetes is delaying the restart due to multiple failures.

The application might be misconfigured or crashing immediately after startup.

You need to debug using logs or pod descriptions.

Resetting Exponential Backoff Count

If a container stops crashing and remains stable, Kubernetes:

Monitors it for a few minutes.

Resets the restart count to zero.

If the container crashes again later, Kubernetes restarts it immediately, starting the cycle fresh.

Debugging Pod Failures & Restart Issues

To troubleshoot pod crashes, Kubernetes provides several useful commands:

Command	Description
kubectl describe pod <pod_name>	Shows detailed information, including event history & errors.
kubectl logs <pod_name>	Retrieves logs from the container.
kubectl logs -f <pod_name>	Follows logs in real-time (similar to tail -f).
kubectl get events	Displays cluster-wide events, useful for identifying scheduling issues.
kubectl get pods --watch	Continuously watches the pod status in real-time.

Creating and Managing Pods with kubectl

This lecture introduces how to create, manage, and inspect Kubernetes pods using the kubectl CLI tool. It covers basic pod creation, setting the Kubernetes context, and verifying pod status.

Command	Description	Options / Explanation
kubectl version	Checks if Kubernetes is running and retrieves client & server versions.	Confirms cluster accessibility.
kubectl config current-context	Displays the current active Kubernetes context.	Ensures you are interacting with the correct cluster.
kubectl config set-context minikube	Sets the context to use Minikube as the Kubernetes cluster.	Use if the context is incorrectly set.

Verifying Kubernetes Cluster & Context

Before creating pods, ensure Kubernetes is running and correctly configured.

The context determines which cluster kubectl interacts with.

If working with multiple clusters, always verify the correct context is selected to avoid accidental modifications.

Creating a Pod using kubectl run

The `kubectl run` command is the primary way to create pods.

Command	Description	Options / Explanation
<code>kubectl run nginx-pod --image=nginx:1.27.0</code>	Creates a pod with a single Nginx container (version 1.27.0).	--image=nginx:1.27.0 specifies the container image.
<code>kubectl get pods</code>	Lists all running pods and their statuses.	Shows columns: NAME, READY, STATUS, RESTARTS, AGE.

READY: 1/1 means one container is running and ready.

STATUS: Running confirms the pod is active.

RESTARTS: 0 indicates no failures.

Understanding the kubectl run Command

`kubectl run <pod-name> --image=<container-image>:<tag>`

`<pod-name>` → Specifies the pod's name.

`--image=<container-image>:<tag>` → Defines the container image.

Ex: `kubectl run nginx-pod --image=nginx:1.27.0`

`kubectl run --help` This provides usage examples and all available options.

Exploring Pod Inspection, Communication, and Management with kubectl

Command	Description	Options / Explanation
kubectl get pods	Lists all running pods and their basic status.	Shows NAME, READY, STATUS, RESTARTS, AGE. The kubectl describe output includes: Namespace where the pod is created. Node where it is running. Start time of the pod. Pod IP address (internal to the cluster). Container image details for debugging. Conditions & Events showing pod scheduling and health status.
kubectl describe pod <pod_name>	Provides detailed information about a specific pod.	Includes Node assignment, IP address, Status, Conditions, and Events.

Communicating with a Pod Internally

Kubernetes assigns an internal IP to each pod, but this IP is not accessible externally. It can only be used within the cluster.

Command	Description	Options / Explanation
<code>kubectl get pod <pod_name> -o wide</code>	Shows additional details, including the pod's internal IP address.	Useful for internal communication.
<code>curl <pod_ip></code> (from another pod)	Tests direct communication with the pod's IP address.	Works only inside the cluster.

Running `curl <pod_ip>` from outside the cluster will fail because the pod IP is private.
To test communication, we must create another pod inside the cluster.

Creating an Interactive Shell Inside a Pod

To interact with a pod's container, we can start an interactive shell session.

Command	Description	Options / Explanation
kubectl run alpine -- image=alpine:3.20 -it -- sh	Creates an interactive pod with Alpine Linux.	-it allows interaction, sh opens a shell.
apk add curl	Installs curl inside the Alpine container.	Alpine uses apk instead of apt/yum.

Using Pod Names for Communication:

- Unlike Docker, Kubernetes does not allow pod-to-pod communication using names by default.
- Instead, we must use internal IP addresses or configure Kubernetes Services (covered later).

After entering the container shell, we can test communication using:

```
curl <nginx_pod_ip>
```

If successful, we receive an HTML response from Nginx.

Retrieving Logs:

Use `kubectl logs <pod_name>` to check container output and errors.
Follow logs live with: `kubectl logs -f <pod_name>`.

Deleting Pods:

`kubectl delete pod <pod_name>` removes a pod normally.
If needed, force deletion using `--force --grace-period=0`.

Exposing Kubernetes Pods Using Services (`kubectl expose`)

This lecture covers how to expose Kubernetes pods using a Service, enabling stable network communication between containers. It also explains how to delete services and clean up resources.

1. Why Use a Kubernetes Service?

By default, pods in Kubernetes only have dynamic IPs that change upon restarts. This makes it difficult to establish persistent communication between different pods.

A Kubernetes Service provides:

- A stable IP address for reaching a pod.
- A DNS name to access the pod instead of using IPs.
- Load balancing for multiple pods.

Listing Existing Pods

Before creating a Service, ensure your pods are running.

Command	Description
<code>kubectl get pods</code>	Lists all running pods and their status.

If an Nginx pod is running, it can be exposed through a Service.

Creating a Service Using kubectl expose

The `kubectl expose` command creates a Service for an existing pod.

Command	Description	Options / Explanation
<code>kubectl expose pod nginx --type=NodePort -port=80</code>	Creates a Service named nginx that exposes the nginx pod.	--type=NodePort makes the Service accessible within the cluster.
<code>kubectl get services</code>	Lists all services and their Cluster IPs.	Displays columns: NAME, TYPE, CLUSTER-IP, PORT(S), AGE.

Understanding Service Types

Service Type	Behavior
ClusterIP (default)	Exposes a pod internally (accessible only inside the cluster).
NodePort	Exposes a pod on a high-range port (30000-32767) on all cluster nodes.
LoadBalancer	Provides an external IP (used in cloud environments).

- A ClusterIP Service assigns a stable IP that does not change even when the pod restarts.
- A NodePort Service allows external access via <node_ip>:<node_port>.

Accessing the Pod via the Service

After creating a Service, use its Cluster IP for communication between pods.

Command	Description
kubectl describe service nginx	Displays the Service details, including its Cluster IP.
kubectl run alpine --image=alpine:3.20 -it - - sh	Starts an interactive shell in a new pod.
apk add curl	Installs curl in Alpine (for making HTTP requests).
curl <cluster_ip>	Sends a request to the nginx Service.

- ✓ The response confirms that the Nginx service is accessible.

Using DNS Names Instead of IP Addresses

Instead of using the Cluster IP, we can communicate using the Service name.

Command	Description
curl nginx	Accesses the pod using the Service name.

- The Service automatically resolves pod IP changes, making it a more reliable method for inter-pod communication.

Deleting a Service & Cleaning Up Pods

Command	Description
kubectl delete service nginx	Deletes the Service, removing its Cluster IP.

Command	Description
kubectl delete pod alpine	Deletes the Alpine pod.
kubectl delete pod nginx	Deletes the Nginx pod.
kubectl get pods	Verifies that no pods are running.

Kubernetes Objects

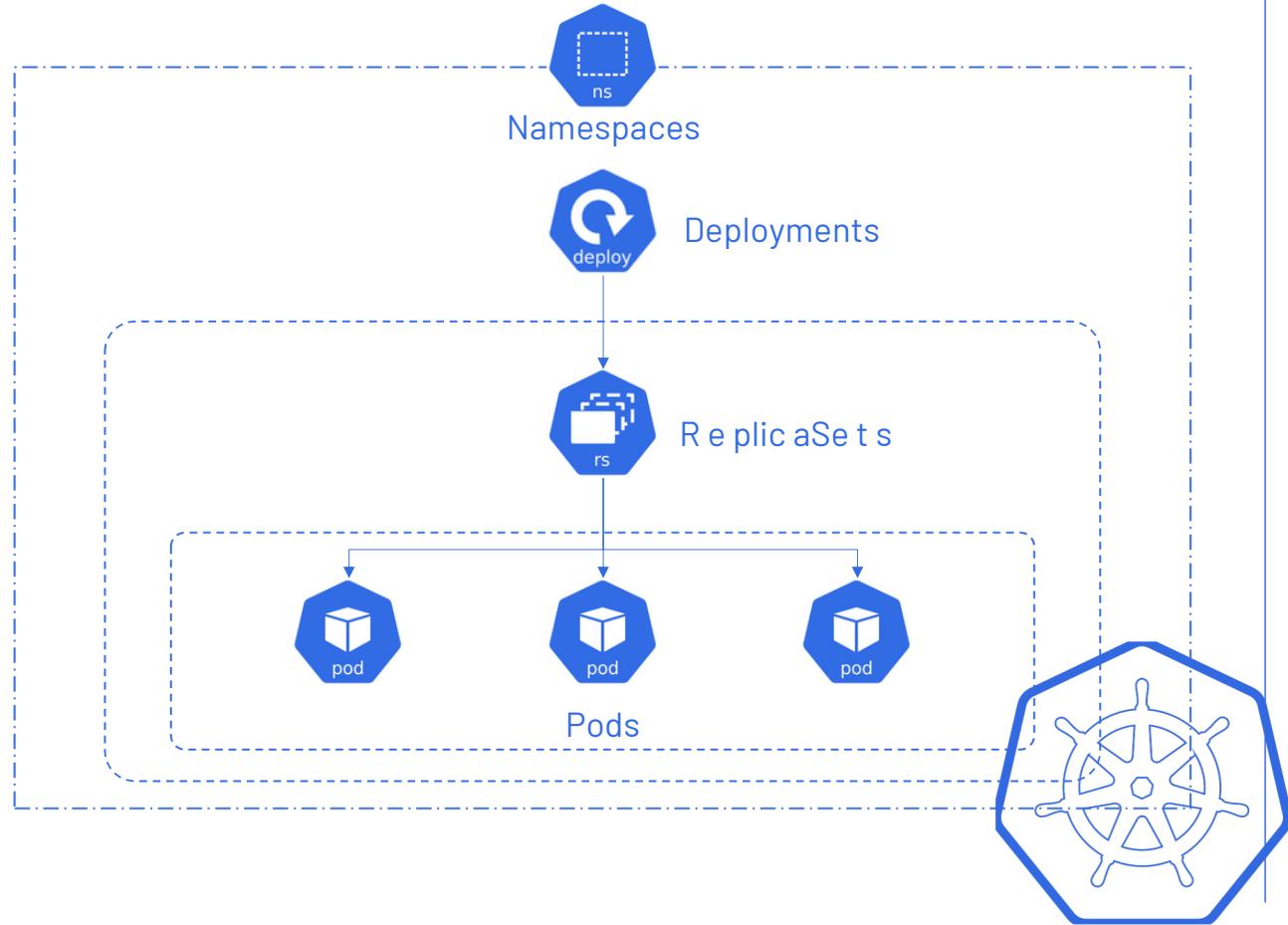
Kubernetes Objects

Provides a way to create logical partitions within a cluster

Manage Pod's lifecycle & provide advanced capabilities for application management

Ensure continuous pod availability by automatically replacing failed pods with new ones

Smallest installable units of computing that can be created and managed in K8s



Namespaces

Namespaces

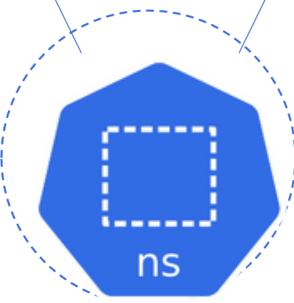
Default Namespaces

Creates four default namespaces
▪ default
▪ kube-node-lease
▪ kube-public
▪ kube-system
to manage cluster resources

to manage cluster resources

Resource Isolation

Isolate resources in a cluster, preventing interference between users or teams

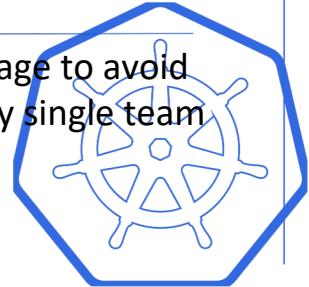


Resource Organization

Helps in organizing and managing resources specific to application or environment

Resource Quotas

Enforce limits on resource usage to avoid excessive consumption by any single team



```
apiVersion: v1
kind: Namespace
metadata:
  name: jaitif
---
apiVersion: v1
kind: Namespace
metadata:
  name: testns
---
apiVersion: v1
kind: ResourceQuota
metadata:
  name: testnsquota
  namespace: testns
spec:
  hard:
    pods: "1"
    requests.memory: "1Gi"
    requests.cpu: "1"
# If you are specifying requests and limits on quotas you have to define them in the POD sepc also
---
apiVersion: v1
kind: Pod
metadata:
  name: testnspod
  namespace: testns
spec:
  containers:
    - name: testnscon
      image: httpd:latest
  resources:
    requests:
      memory: "100Mi"
      cpu: ".2"
  apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: quotatest
    namespace: jaitif
  spec:
    hard:
      pods: "1"
```

```
# List namespaces  
kubectl get namespaces
```

```
# Verify ResourceQuota  
kubectl get resourcequota -n testns
```

```
# Describe ResourceQuota to check limits  
kubectl describe resourcequota testnsquota -n testns
```

```
# List pods in the namespace  
kubectl get pods -n testns
```

```
# Describe the pod to check resource requests  
kubectl describe pod testnspod -n testns
```

Command to Apply a Pod Requesting More Memory (Should Fail)

Command to Apply an Extra Pod (Should Fail)

```
cat <<EOF | kubectl apply -f -  
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: extratestpod  
  namespace: testns  
spec:  
  containers:  
    - name: extratestcon  
      image: httpd:latest  
EOF
```

```
cat <<EOF | kubectl apply -f -
```

```
---  
apiVersion: v1  
kind: Pod  
metadata:  
  name: highmem-pod  
  namespace: testns  
spec:  
  containers:  
    - name: highmem-container  
      image: httpd:latest  
      resources:  
        requests:  
          memory: "2Gi" # Exceeding the quota  
EOF
```

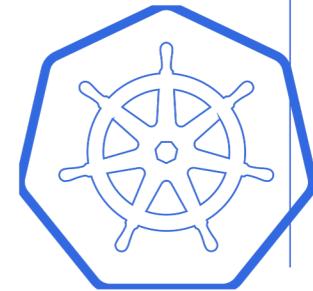
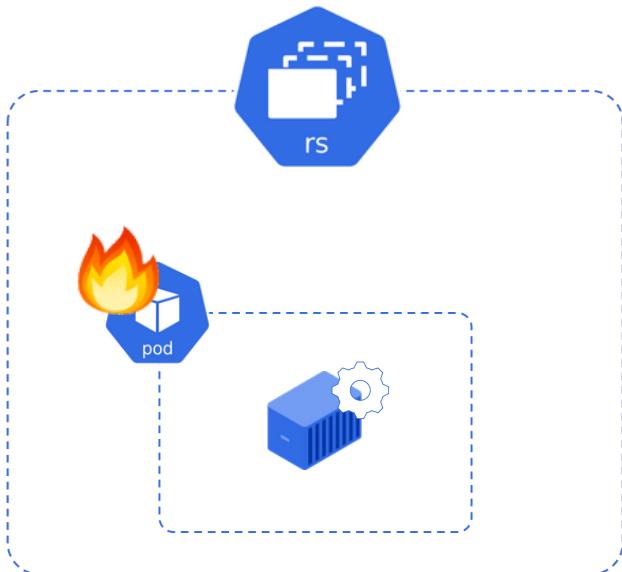
Command to Apply a Pod Requesting More CPU (Should Fail)

```
cat <<EOF | kubectl apply -f -
---
apiVersion: v1
kind: Pod
metadata:
  name: highcpu-pod
  namespace: testns
spec:
  containers:
  - name: highcpu-container
    image: httpd:latest
    resources:
      requests:
        cpu: "2" # Exceeding the quota
EOF
```

kubectl describe resourcequota testnsquota -n testns

ReplicaSets

ReplicaSets

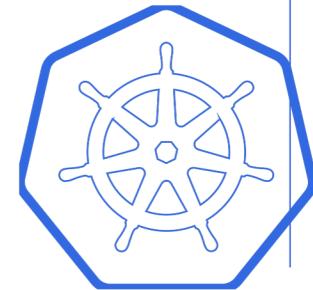


How ReplicaSets Works?

Define
ReplicaSet
Object

Monitor
running Pods

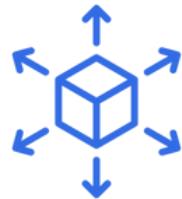
Effortless
Scaling



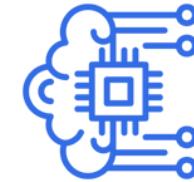
Benefits of ReplicaSets



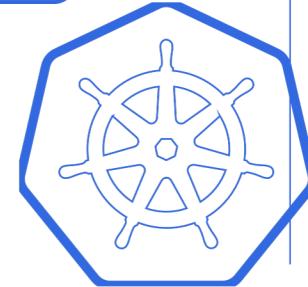
High Availability



Scalability



Self-healing



```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
    app: myapp
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
    spec:
      containers:
        - name: httpd-container
          image: httpd

  replicas: 1
  selector:
    matchLabels:
      app: myapp
```

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
    app: myapp
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
    spec:
      containers:
        - name: httpd-container
          image: httpd

  replicas: 3
  selector:
    matchLabels:
      app: myapp
```

ReplicaSets

Ensure that a certain number of pods is running at any given time

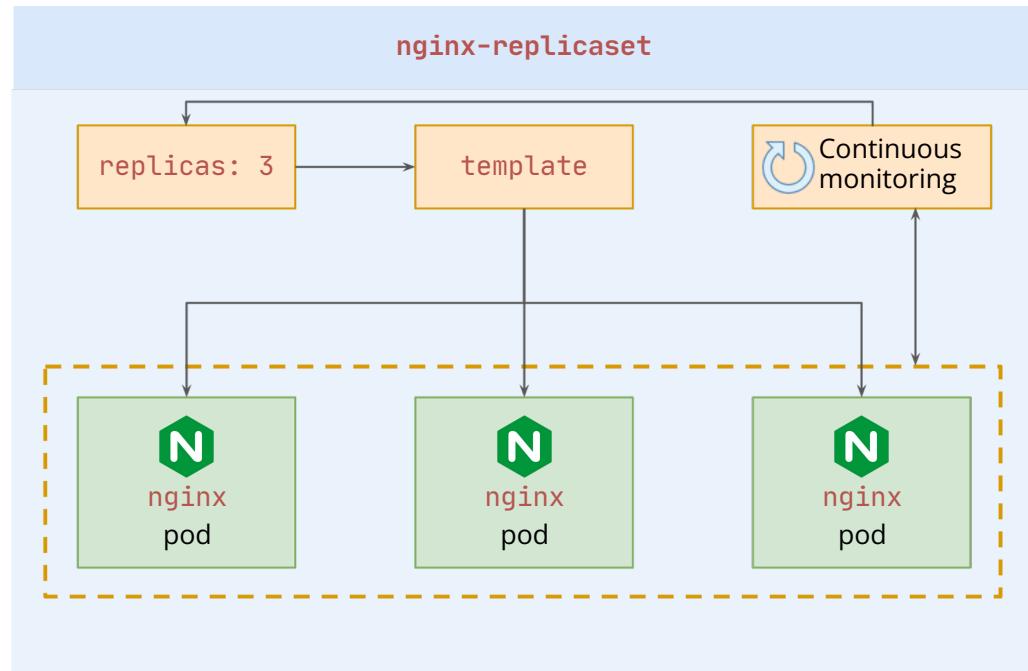
- So far, we ran pods manually and individually. If a pod breaks or enters into an unhealthy state, we need to manually intervene.
- ReplicaSets address the issue of replacing exited or unhealthy pods and making sure that a stable number of identical pods are running at any point in time.
 - Ensures high availability and fault tolerance by automatically replacing failed or terminated pods.
- ReplicaSets work by identifying pods via **selectors**, and continuously checking whether the number of running pods matches the desired number of pods specified in the configuration.
- ReplicaSets will recreate pods automatically based on the **template** section in their **spec**.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replicaset
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.27.0
```

ReplicaSets

Ensure that a certain number of pods is running at any given time

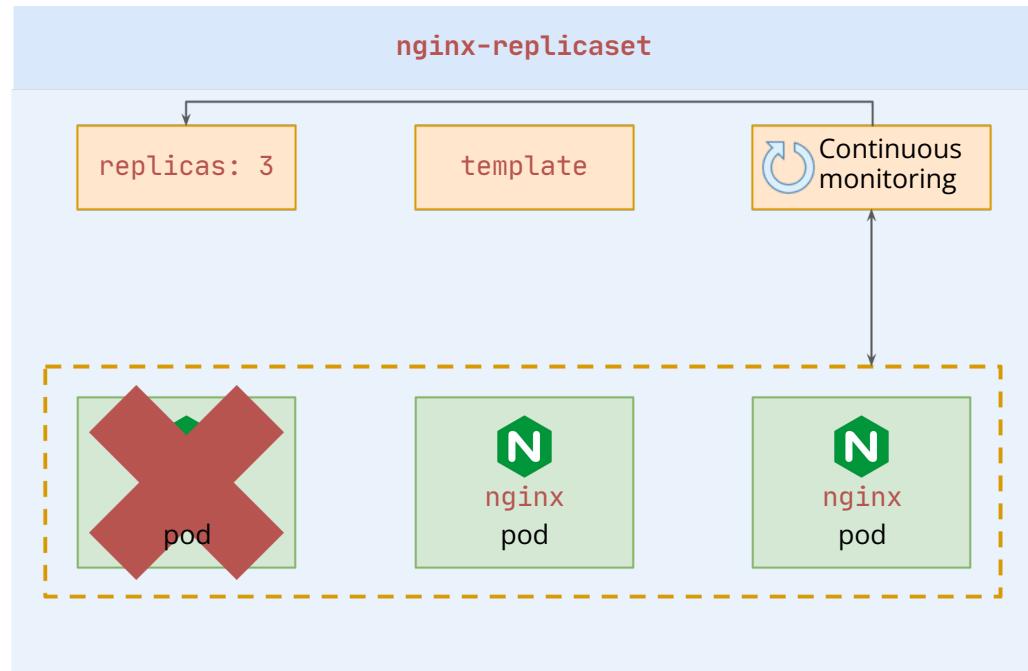
```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replicaset
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.27.0
```



ReplicaSets

Ensure that a certain number of pods is running at any given time

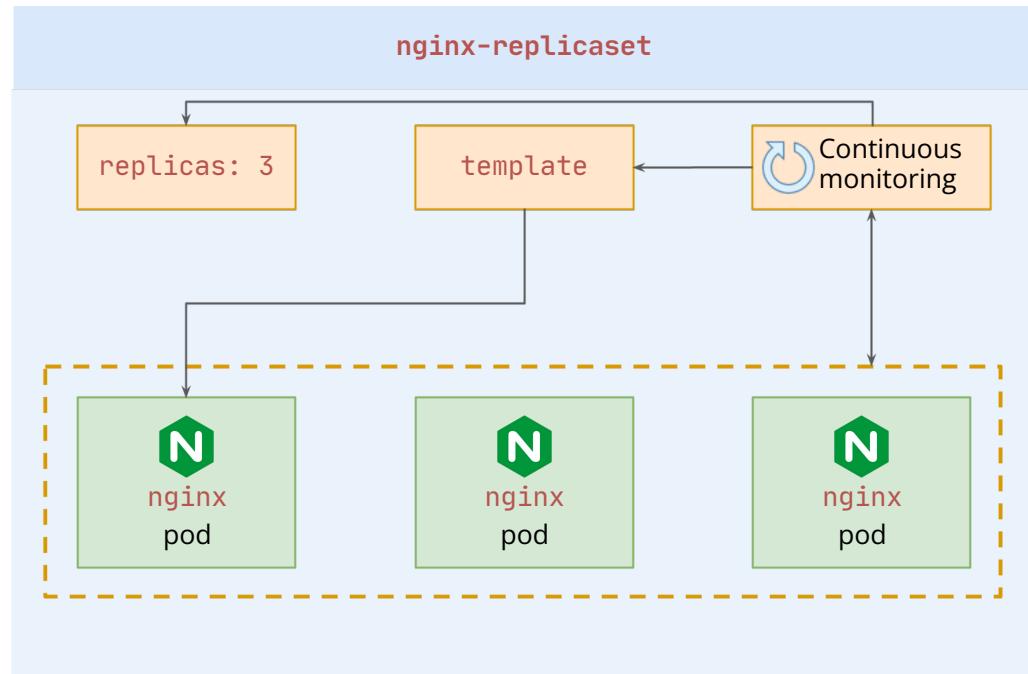
```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replicaset
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.27.0
```



ReplicaSets

Ensure that a certain number of pods is running at any given time

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replicaset
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.27.0
```



Creating a Replica Set in Kubernetes

```
# What We Are Doing:  
# 1. Clean up the Kubernetes cluster to remove any existing resources.  
# 2. Create a new YAML file to define a ReplicaSet.  
# 3. Apply the configuration using `kubectl apply`.  
# 4. Verify the ReplicaSet and its managed pods.  
# 5. Test Kubernetes' self-healing mechanism by deleting a pod.
```

```
# Step 1: Clean Up Existing Resources (if any)
kubectl delete all --all

# Step 2: Create a YAML File for the Replica Set
cat <<EOF > nginx-rs-replicaset.yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replica-set
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.27.0
          ports:
            - containerPort: 80
EOF
```

```
# Step 3: Apply the Replica Set Configuration  
kubectl apply -f nginx-rs-replicaset.yaml
```

```
# Step 4: Verify that the Replica Set and Pods are Running  
kubectl get rs  
kubectl get pods
```

```
# Step 5: Test Auto-Recovery by Deleting a Pod  
POD_NAME=$(kubectl get pods -l app=nginx -o jsonpath='{.items[0].metadata.name}')  
kubectl delete pod $POD_NAME
```

```
# Step 6: Recheck Pods to Confirm Auto-Recovery  
kubectl get pods
```

```
# Step 7: Clean Up (Optional)  
kubectl delete -f nginx-rs-replicaset.yaml
```

Creating a Kubernetes Deployment

What We Are Doing:

- # 1. Create a deployment YAML file.
- # 2. Define API version, metadata, labels, and spec.
- # 3. Specify replicas and the deployment strategy.
- # 4. Apply the deployment using kubectl.
- # 5. Verify deployment and pod status.

Kubernetes Deployments

Deployments

Manage and update applications at scale

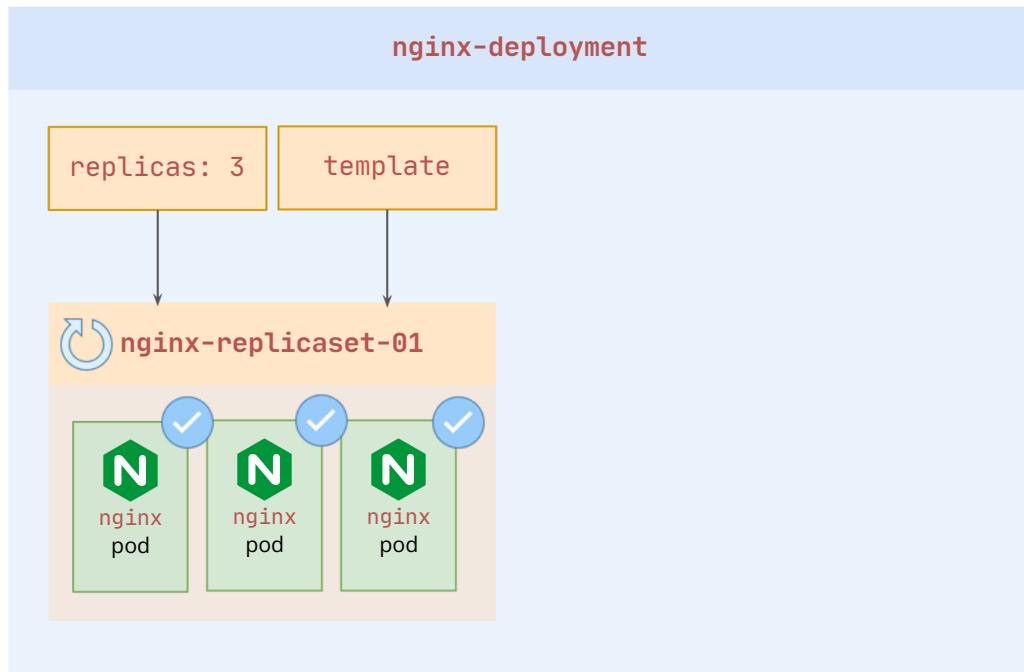
- ReplicaSets offer a great way to ensure a pre-defined number of pods are running at any given time. However, they do not provide all the needed functionality for managing applications at scale.
- Deployments offer a higher level of abstraction on top of ReplicaSets, and add the following features:
 - Rolling updates:** Gradually replace old pods with new ones without downtime, ensuring the application remains available.
 - Rollbacks:** roll back to a previous version in case something goes wrong during an update. This feature is crucial for maintaining application stability and quickly recovering from errors.
 - Declarative updates** of replicas over time.
 - History and revision control:** keep track of the history of all changes made, allowing you to view and revert to previous versions.
 - Advanced rollouts:** limit risk by leveraging controlled rollouts of new versions, ensuring that updates are applied gradually and safely.

```
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: ...
strategy:
  type: RollingUpdate
  rollingUpdate: ...
```

Deployments

Manage and update applications at scale

```
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:
        - ...
strategy:
  type: RollingUpdate
```

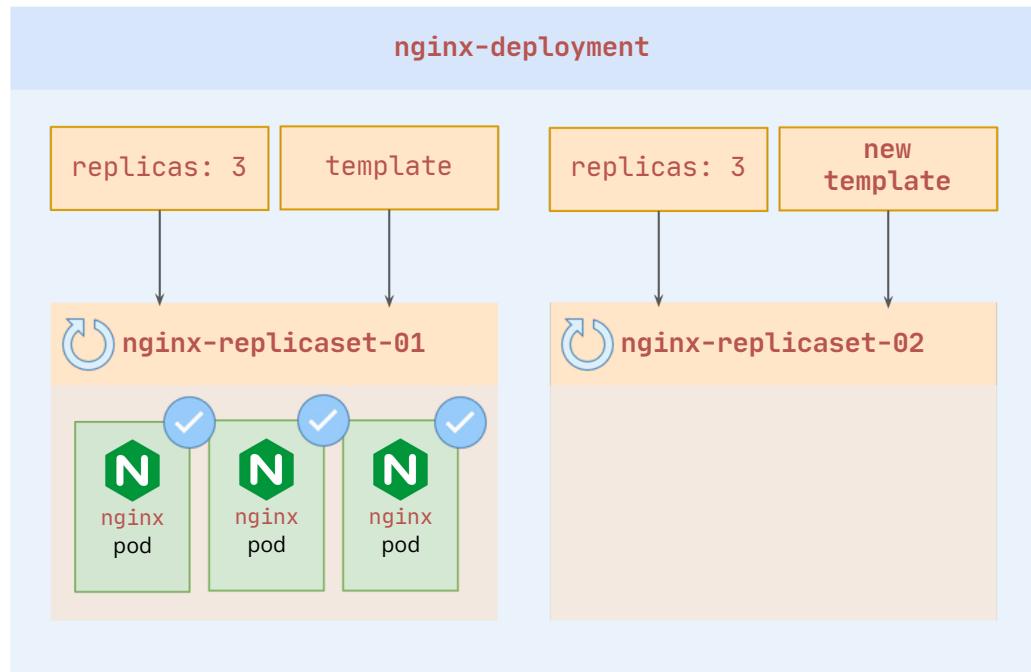


Kubernetes

Deployments

Manage and update applications at scale

```
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:
        - ...
strategy:
  type: RollingUpdate
```

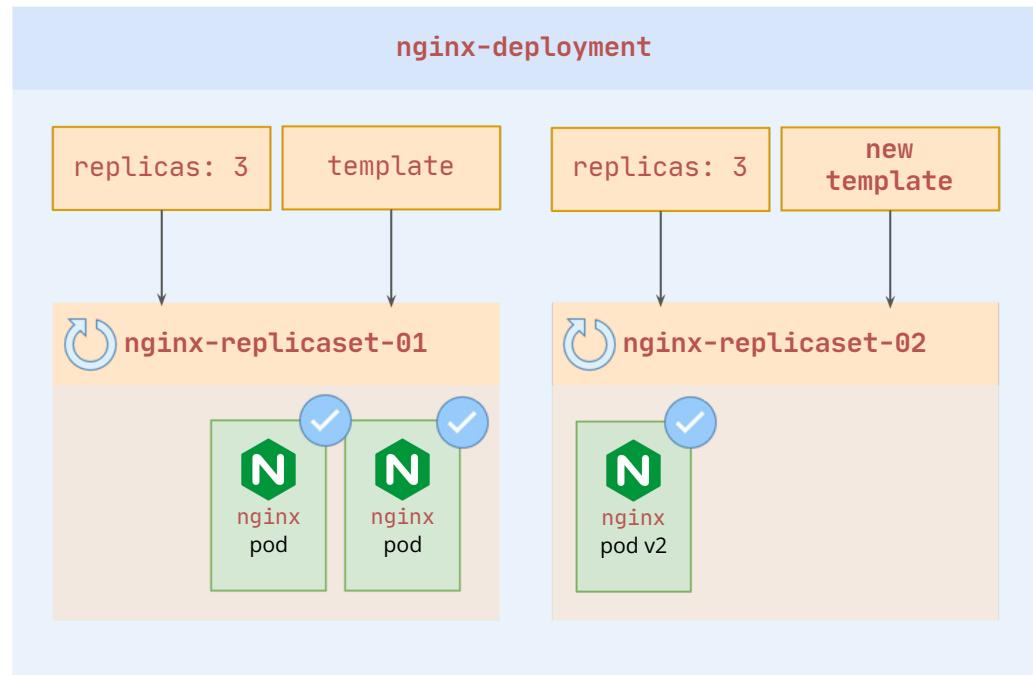


Kubernetes

Deployments

Manage and update applications at scale

```
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:
        - ...
strategy:
  type: RollingUpdate
```

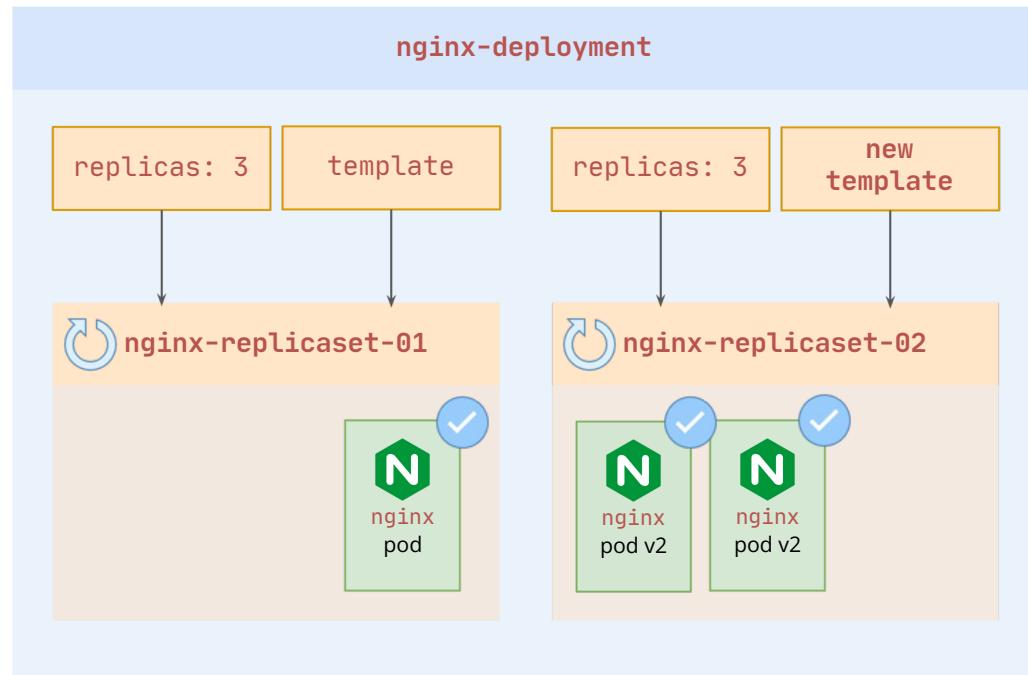


Kubernetes

Deployments

Manage and update applications at scale

```
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:
        - ...
strategy:
  type: RollingUpdate
```

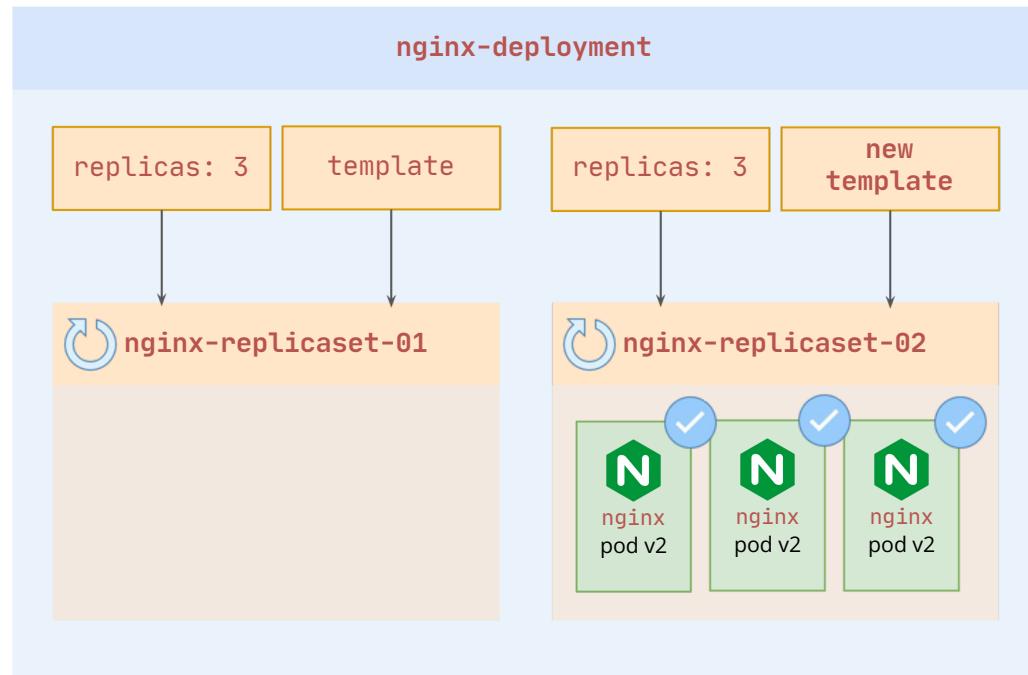


Kubernetes

Deployments

Manage and update applications at scale

```
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:
        - ...
strategy:
  type: RollingUpdate
```

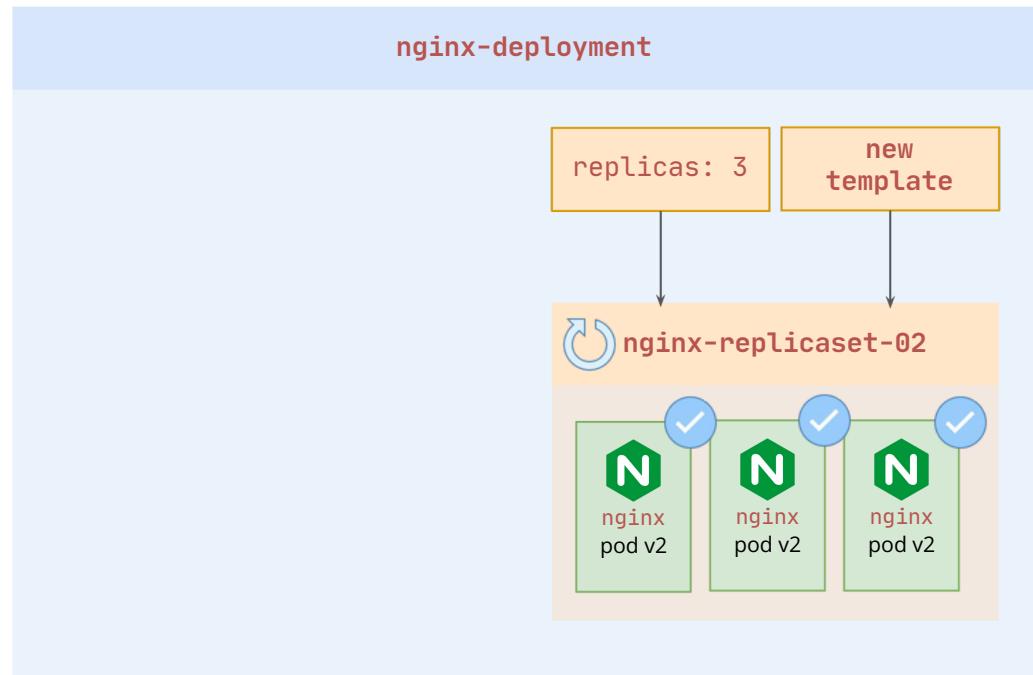


Kubernetes

Deployments

Manage and update applications at scale

```
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:
        - ...
strategy:
  type: RollingUpdate
```

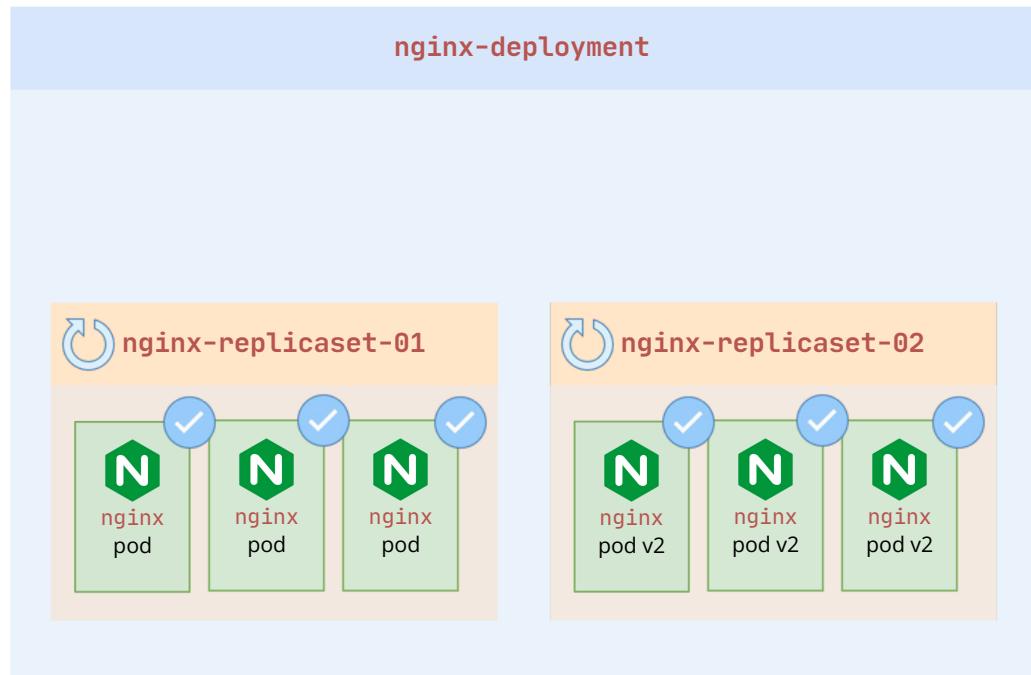


Kubernetes

Deployments

Manage and update applications at scale

```
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: ...
strategy:
  type: RollingUpdate
  rollingUpdate: ...
```



Kubernetes

```
# Step 1: Create the Deployment YAML File
cat <<EOF > nginx-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 5
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.27.0
          ports:
            - containerPort: 80
strategy:
  type: RollingUpdate
EOF

# Step 2: Apply the Deployment
kubectl apply -f nginx-deployment.yaml

# Step 3: Check Deployment Status
kubectl get deploy

# Step 4: Describe Deployment
kubectl describe deployment nginx-deployment

# Step 5: List the Pods Created by the Deployment
kubectl get pods

# Step 6: View Configuration of a Specific Pod
kubectl get pod $(kubectl get pods -l app=nginx -o jsonpath='{.items[0].metadata.name}') -o yaml

# Step 7: Cleanup (Optional)
kubectl delete -f nginx-deployment.yaml
```

Updating Pod Templates in Kubernetes

```
# What We Are Doing:  
# 1. Open and modify the deployment YAML file.  
# 2. Change the image from `nginx:1.27.0` to `nginx:1.27.0-alpine`.  
# 3. Dry-run to check changes before applying.  
# 4. Apply the update using kubectl.  
# 5. Monitor pods and replica sets for rolling update progress.  
# 6. Verify the updated image version.
```

```
# Step 1: Modify the Deployment YAML File
cat <<EOF > nginx-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 5
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.27.0-alpine
          ports:
            - containerPort: 80
strategy:
  type: RollingUpdate
EOF

# Step 2: Check What Will Change Before Applying
kubectl apply --dry-run=client -f nginx-deployment.yaml

# Step 3: Monitor Current Pods and Replica Sets
kubectl get pods --watch &
kubectl get rs --watch &

# Step 4: Apply the Update
kubectl apply -f nginx-deployment.yaml

# Step 5: Monitor the Rolling Update Progress
kubectl rollout status deployment nginx-deployment

# Step 6: Describe the Deployment to Verify Changes
kubectl describe deploy nginx-deployment

# Step 7: Confirm the Image Update in the Running Pods
kubectl get pods
kubectl describe pod $(kubectl get pods -l app=nginx -o jsonpath='{.items[0].metadata.name}') | grep Image

# Step 8: Cleanup (Optional)
kubectl delete -f nginx-deployment.yaml
```

Exploring Rollouts in Kubernetes

What We Are Doing:

- # 1. Verify deployment status.
- # 2. Check rollout history.
- # 3. Undo rollouts if necessary.
- # 4. View revision details.
- # 5. Add annotations for tracking changes.
- # 6. Apply and verify changes.

```
# Step 1: Verify Deployment Status  
kubectl get deploy
```

```
# Step 2: Check Rollout History  
kubectl rollout history deployment/nginx-deployment
```

```
# Step 3: Undo a Rollout (Rollback to Previous Version)  
kubectl rollout undo deployment/nginx-deployment
```

```
# Step 4: View Details of a Specific Revision  
kubectl rollout history deployment/nginx-deployment --revision=2
```



```
# Step 6: Apply the Deployment with the Change Cause  
kubectl apply -f nginx-deployment.yaml
```

```
# Step 7: Annotate the Deployment Post-Rollout (Alternative CI/CD Approach)  
kubectl annotate deployment/nginx-deployment  
kubernetes.io/change-cause="Update nginx to tag 1.27.1-alpine"
```

```
# Step 8: Verify Annotations  
kubectl describe deployment/nginx-deployment
```

```
# Optional Cleanup  
kubectl delete -f nginx-deployment.yaml
```

```
# Step 5: Modify Deployment YAML to Add an Annotation  
cat <<EOF > nginx-deployment.yaml  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: nginx-deployment  
  labels:  
    app: nginx  
  annotations:  
    kubernetes.io/change-cause: "Update nginx to tag 1.27.0-alpine"  
spec:  
  replicas: 5  
  selector:  
    matchLabels:  
      app: nginx  
  template:  
    metadata:  
      labels:  
        app: nginx  
    spec:  
      containers:  
        - name: nginx  
          image: nginx:1.27.0-alpine  
          ports:  
            - containerPort: 80  
      strategy:  
        type: RollingUpdate  
EOF
```

Scaling Deployments in Kubernetes

What We Are Doing:

- # 1. Check current deployment status.
- # 2. Scale deployment replicas up or down.
- # 3. Verify the changes.
- # 4. Understand that `kubectl scale` is temporary.
- # 5. (Optional) Reset replicas for troubleshooting.

```
# Step 1: Get Current Deployments
```

```
kubectl get deploy
```

```
# Step 2: Scale Deployment to 3 Replicas
```

```
kubectl scale deployment nginx --replicas=3
```

```
# Step 3: Verify the Scaling
```

```
kubectl get deploy
```

```
# Step 4: Apply the Original Configuration (Optional)
```

```
kubectl apply -f nginx-deployment.yaml
```

```
# Step 5: Scale Down to Zero (For Recovery Purposes)
```

```
kubectl scale deployment nginx --replicas=0
```

```
# Step 6: Scale Back to Desired Number
```

```
kubectl scale deployment nginx --replicas=5
```

```
# Optional Cleanup
```

```
kubectl delete deployment nginx
```

Understanding Failed Rollouts in Kubernetes

What We Are Doing:

- # 1. Deploy an invalid configuration (incorrect image tag).
- # 2. Investigate the deployment and pod errors.
- # 3. Roll back to the previous stable version if needed.
- # 4. Fix the invalid configuration and reapply.
- # 5. Verify the deployment.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:invalid-tag # Intentionally incorrect tag
          ports:
            - containerPort: 80
```

```
# Step 1: Create an Invalid Deployment (Incorrect Image Tag)
```

```
kubectl apply -f nginx-invalid-deployment.yaml
```

```
# Step 2: Check Deployment Status
```

```
kubectl describe deployment nginx
```

```
# Step 3: Investigate Pod Errors
```

```
kubectl get pods # Look for ImagePullBackOff status
```

```
kubectl describe pod <pod-name>
```

```
# Step 4: Identify the Issue (Check Image Pull Failures)
```

```
kubectl logs <pod-name>
```

```
# Step 5: Roll Back Deployment to Last Working Version  
(if needed)
```

```
kubectl rollout undo deployment/nginx
```

```
# Step 6: Fix the Invalid Configuration in YAML and Reapply
```

```
kubectl apply -f nginx-deployment.yaml
```

```
# Step 7: Verify the Deployment and Ensure Pods Are Running
```

```
kubectl get pods
```

```
kubectl rollout status deployment/nginx
```

```
# Optional Cleanup
```

```
kubectl delete deployment nginx
```

Labels & Selectors

Labels & Selectors

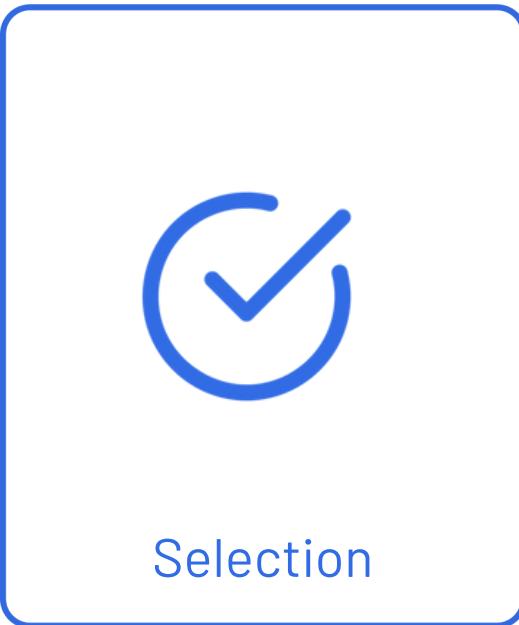
Provides a way to categorize & target resources within a cluster

Labels categorize resources in Kubernetes with identifying tags
Selectors filter resources based on their assigned labels

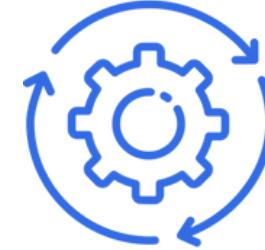
Benefits of Labels & Selectors



Categorization
/Organization



Selection



Automation

Labels are key-value pairs attached to Kubernetes objects (e.g., Pods, Services, Deployments). They help organize, group, and select resources.

Selectors are filters used to query and select Kubernetes resources based on their labels.

📌 Why Use Labels & Selectors? ✓ Organize resources logically.

✓ Select groups of Pods for a Service, Deployment, or NetworkPolicy.

✓ Enable dynamic management of workloads.

Feature	Explanation
Labels	Metadata (key-value pairs) assigned to objects.
Selectors	Used to query and filter objects based on labels.
Immutable?	Labels can be added/modified but not removed once set.
Use Cases	- Grouping resources - Attaching Pods to Services - Identifying environments (prod, dev, test)

Labels and Selectors

Identify and group resources meaningfully

- Labels are key-value pairs attached to Kubernetes objects (e.g., pods, nodes, services). They provide metadata that helps identify and organize these objects.
Labels allow Kubernetes users to categorize and organize resources, enabling
 - sophisticated grouping and selection mechanisms. **Labels are not unique**, and multiple objects can have the same label.
- Selectors** are expressions used to filter Kubernetes objects based on their labels. Selectors allow users and Kubernetes components to target specific objects that match certain criteria.

 **labels:**
:
app: color-api
environment: dev
release: frontend

 **labels:**
:
app: color-api
environment: dev
release: backend

 **labels:**
:
app: color-api
environment: dev
release: backend

```
apiVersion: v1
kind: Pod
metadata: name: api-server
labels:
  app: color-api
  tier: backend
spec: ...
```

```
apiVersion: v1
kind: Service
metadata: ...
spec:
  selector:
    app: color-api
    tier: backend
```

```
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: label-demo
  labels:
    app: web
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
        tier: frontend
    spec:
      containers:
        - name: nginx
          image: nginx:latest
EOF
```

1 Show all pods with labels
kubectl get pods --show-labels
2 Filter pods based on label "tier=frontend"
kubectl get pods -l tier=frontend
3 Add a new label "env=production" to the first pod
POD_NAME=\$(kubectl get pods -l app=web -o jsonpath='{.items[0].metadata.name}')
kubectl label pod \$POD_NAME env=production
4 Verify label was added
kubectl get pods --show-labels
5 Remove the label "env" from the pod
kubectl label pod \$POD_NAME env-
6 Show updated labels
kubectl get pods --show-labels
7 Delete all pods with label "tier=frontend"
kubectl delete pods -l tier=frontend

Inbuilt (System) Labels in Kubernetes

In Kubernetes, inbuilt labels are automatically assigned by the system to resources (like pods, nodes, etc.). These labels provide metadata that helps Kubernetes manage and schedule workloads effectively.

```
kubectl get nodes --show-labels
```

Kubernetes Annotations

Annotations

Add useful information and configuration to resources

- Annotations are key-value pairs attached to Kubernetes objects, just like labels. Unlike labels, annotations are not supposed to store identifying metadata, and they are often used by tools or the Kubernetes system itself.
- Common use-cases:
 - Tool-specific metadata and configuration:** external tools (monitoring systems, logging agents) leverage annotations to attach custom data to resources (configurations, metrics collection endpoints, etc.)
 - Configuration for ingress controllers:** used to configure ingress controllers (traffic routing, SSL termination, security settings). For example, you can define path rewrites or custom load balancing rules via annotations.
 - Storing build and version information:** annotations can store metadata such as build timestamps, version numbers, or Git commit hashes.
 - Runtime configuration for operators:** Kubernetes operators or controllers can use annotations to customize runtime behavior for specific resources.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
annotations:
  nginx.ingress.kubernetes.io/rewrite-target: /
  nginx.ingress.kubernetes.io/ssl-redirect: "true"
  nginx.ingress.kubernetes.io/force-ssl-redirect: "true"
  nginx.ingress.kubernetes.io/proxy-body-size: 10m
spec: ...
```

Prefix

Name

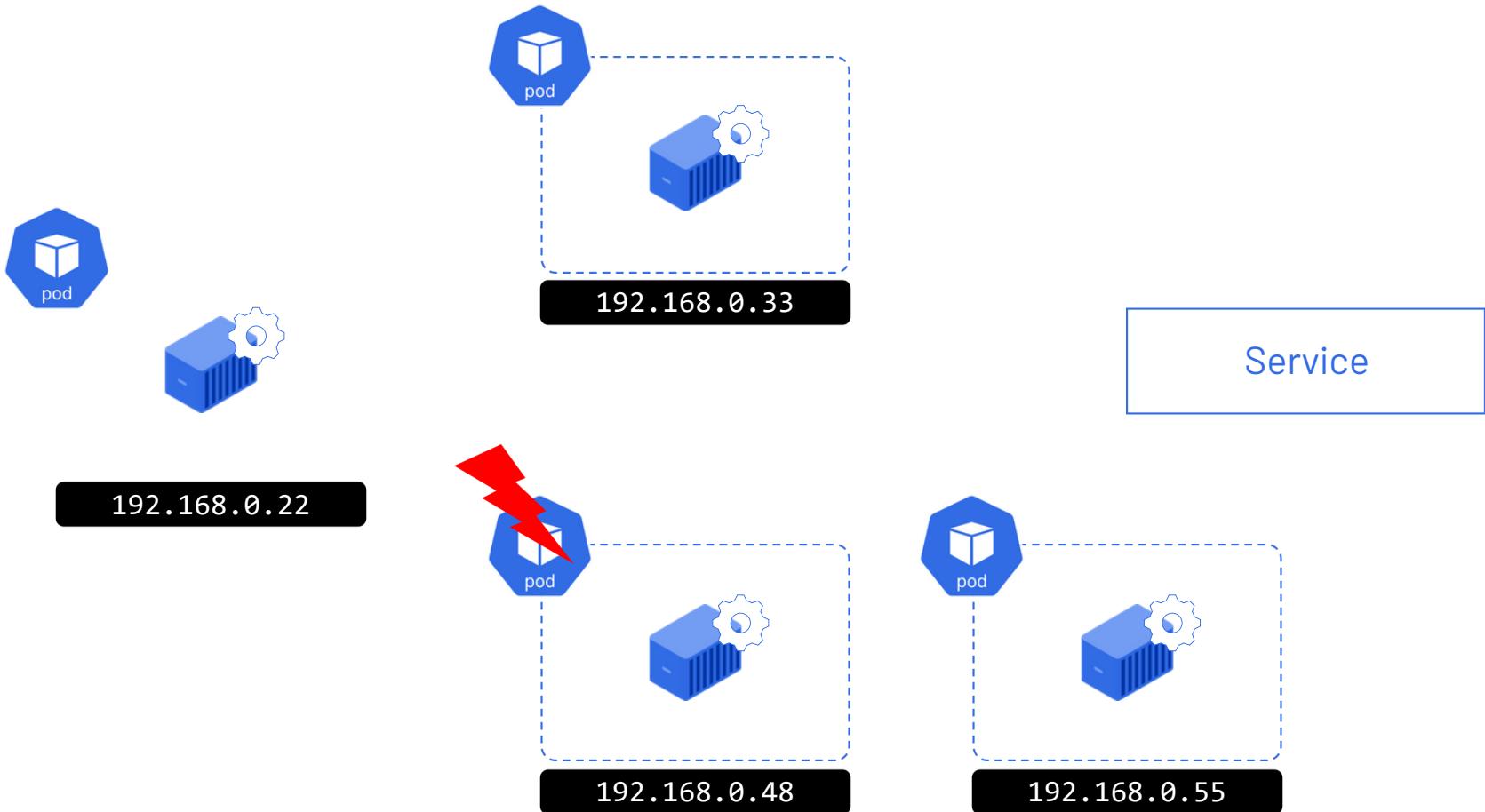
- Must be a DNS subdomain
- Less than 63 characters
- Alphanumeric, dots, underscores and dashes

```
# 1 Create a Pod with Annotations
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: annotation-demo
  annotations:
    build-version: "1.2.3"
    owner: "dev-team"
    description: "This pod is used for annotation demo"
spec:
  containers:
  - name: nginx
    image: nginx:latest
EOF
# 2 Retrieve Annotations for the Pod
kubectl get pod annotation-demo -o jsonpath='{.metadata.annotations}'
# 3 Add a New Annotation
kubectl annotate pod annotation-demo env=production
# 4 Verify the New Annotation
kubectl get pod annotation-demo -o jsonpath='{.metadata.annotations}'
# 5 Remove an Annotation (Example: Remove 'description' annotation)
kubectl annotate pod annotation-demo description-
# 6 Show Final Annotations After Removal
kubectl get pod annotation-demo -o jsonpath='{.metadata.annotations}'
# 7 Clean Up (Optional)
kubectl delete pod annotation-demo
```

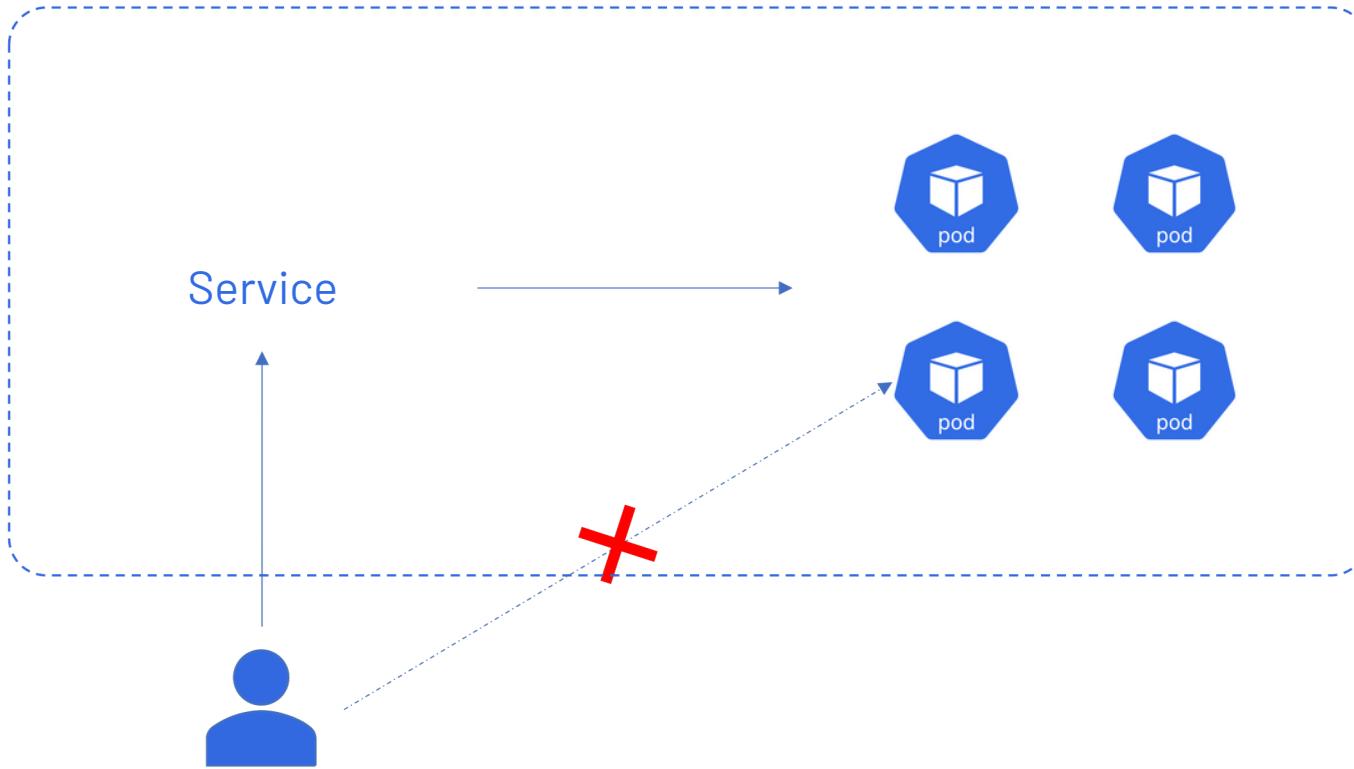
Kubernetes Networking

Services

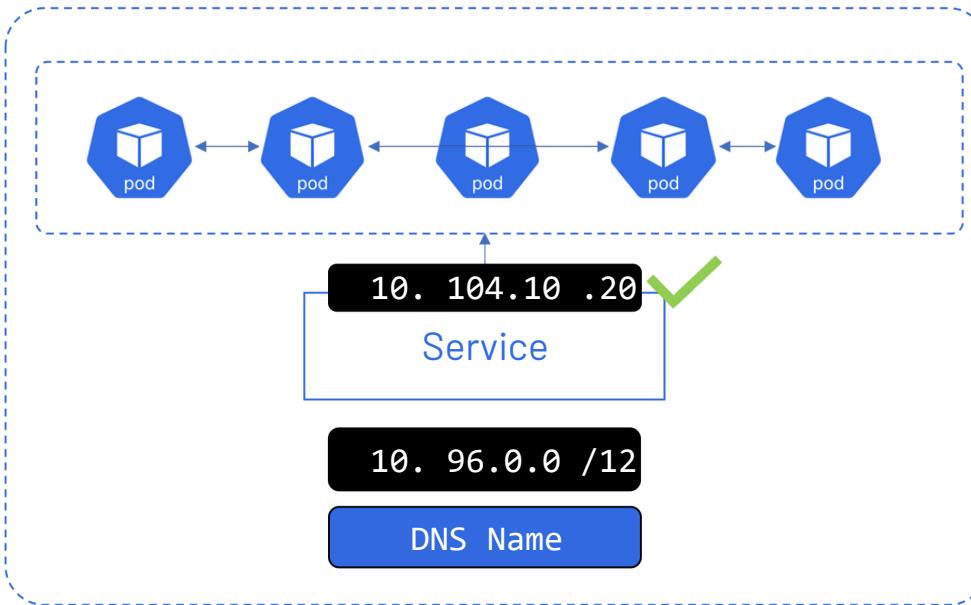
Services



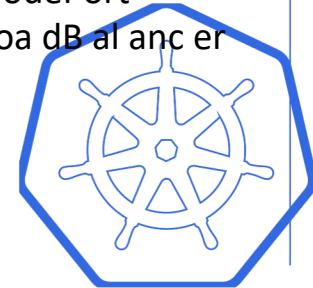
Services



Services



- ✓ ClusterIP
- ✓ NodePort
- ✓ LoadBalancer



Multiple Pods :

The blue hexagons represent Pods, which could be replicas of your application. These Pods each have their own IP address, but those IPs are short-lived (ephemeral).

Service Object :

The Service is assigned a single, stable virtual IP address (e.g., 10.104.10.20) within the cluster's IP range (like 10.96.0.0/12).

Internally, Kubernetes routes traffic from this Service IP to the appropriate Pods (using label selectors or other mechanisms).

This IP address does not change, even if the underlying Pods are added, removed, or replaced.

DNS Name:

Services are also given a DNS name within the cluster (e.g., my-service.my-namespace.svc.cluster.local), allowing applications to refer to the Service by name rather than by IP address.

Service Types:

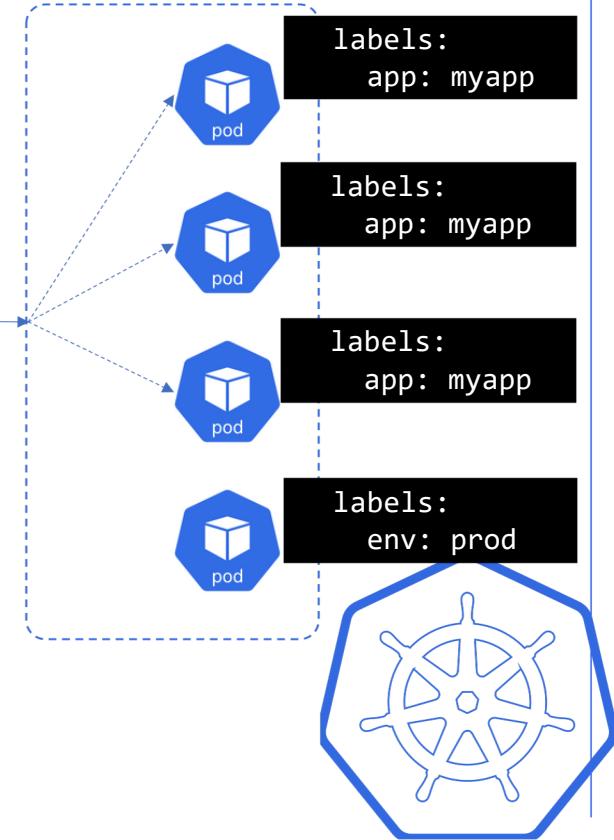
ClusterIP: The default type. Exposes the Service on an internal IP that is only reachable within the cluster.

NodePort: Exposes the Service on a static port on each Node's IP address, making it accessible externally (e.g., NodeIP:NodePort).

LoadBalancer: Integrates with a cloud provider to provision an external load balancer. Assigns a publicly accessible IP (or hostname) to route traffic into the cluster.

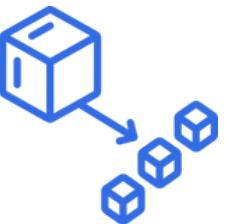
Services

Services can
perform Internal
load balancing

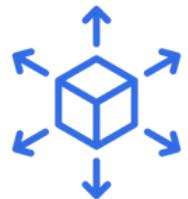


Benefits of Service Objects

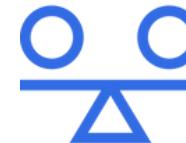
Benefits of Service Objects



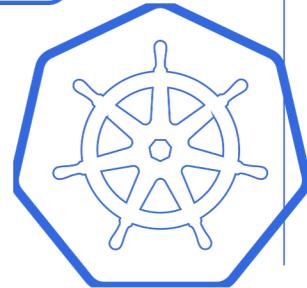
Decoupling



Scalability



Stability



ClusterIP

ClusterIP

Expose applications exclusively to other pods within the same cluster for internal communication



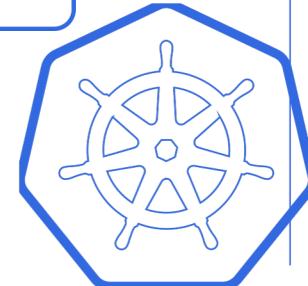
Communication



Security



Limitations



ClusterIP is the default Service type that exposes your application on an internal IP only accessible within the cluster.

Imperative Approach

Create a Deployment:

```
kubectl create deployment nginx-deployment --image=nginx:latest
```

```
kubectl expose deployment nginx-deployment --port=80 --target-port=80 --name=nginx-clusterip
```

```
kubectl get svc nginx-clusterip
```

Test the Service (from within the cluster):

```
kubectl run curlpod --rm -it --image=curlimages/curl -- /bin/sh  
# Inside the container, run:  
curl http://nginx-clusterip
```

Declarative Approach

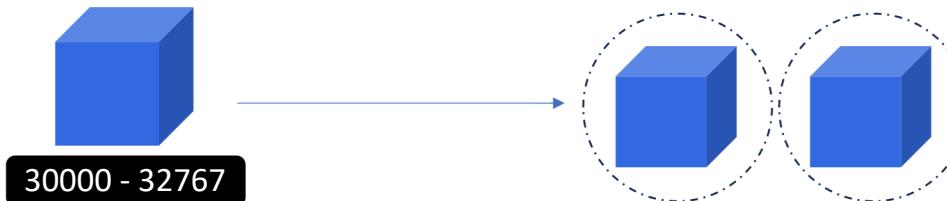
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
template:
  metadata:
    labels:
      app: nginx
spec:
  containers:
    - name: nginx
      image: nginx:latest
      ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: nginx-clusterip
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
```

kubectl apply -f clusterip.yaml
kubectl get svc nginx-clusterip

NodePort Service

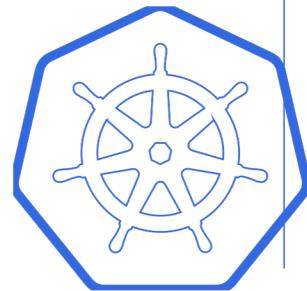
NodePort

- Expose application on a static port across all nodes in the cluster
- Default port range is 30000 – 32767



NodePort Benefits

- ✓ Simple External Access
- ✓ No Load Balancer Required



NodePort exposes your Service on each Node's IP at a static port (within the range 30000–32767), which can be accessed externally.

Imperative Approach

Expose the Deployment as a NodePort Service:

```
kubectl expose deployment nginx-deployment --port=80 --target-port=80 --type=NodePort --name=nginx-nodeport
```

```
kubectl get svc nginx-nodeport
```

```
curl http://<NODE-IP>:31234
```

Declarative Approach

Create a file named nodeport.yaml with the following content:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-nodeport
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      # Optionally, specify a fixed nodePort; if omitted, Kubernetes assigns one automatically.
      nodePort: 30080
      type: NodePort
```

```
kubectl apply -f nodeport.yaml
```

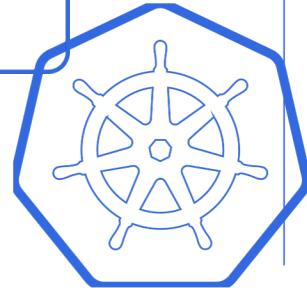
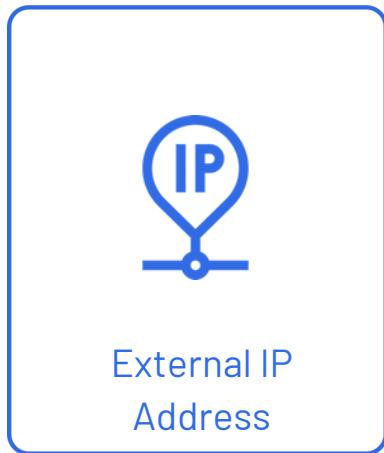
```
kubectl get svc nginx-nodeport
```

```
curl http://<NODE-IP>:30080
```

LoadBalancer Service

Load Balancer

- Leverages external or cloud provider-specific load balancers to distribute incoming traffic across pods
- Highly available & scalable solution for exposing application to the public internet



LoadBalancer provisions an external load balancer (supported by your cloud provider) and assigns an external IP to your Service.

Imperative Approach

Expose the Deployment as a LoadBalancer Service

```
kubectl expose deployment nginx-deployment --port=80 --target-port=80 --type=LoadBalancer --name=nginx-loadbalancer
```

```
kubectl get svc nginx-loadbalancer
```

Test the Service:

Once the external IP is ready, test with:

```
curl http://a1b2c3d4e5f6.us-west-2.elb.amazonaws.com
```

If you're using a local environment like Minikube, you can simulate a LoadBalancer using:

```
minikube tunnel
```

Declarative Approach

Create a file named loadbalancer.yaml with the following content:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-loadbalancer
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      type: LoadBalancer
      curl http://<EXTERNAL-IP>
      kubectl apply -f loadbalancer.yaml
      kubectl get svc nginx-loadbalancer
```

Kubernetes

Object Management & YAML Files

Kubernetes Manifest Files

K8s Manifest Files

Understanding the structure of K8s YAML manifests

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

Kubernetes

K8s Manifest Files

Understanding the structure of K8s YAML manifests

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

apiVersion

The field defines which API group and the respective version of the API is being used to create the object.

kind

The **kind** field defines which exact object or resource is being managed by the configuration file. It must be supported by the specified **apiVersion**.

metadata

The **metadata** field defines data that is used to uniquely identify the object, such as name, namespace, labels, and annotations. K8s may also add information to the **metadata** section.

spec

The **spec** field defines the actual configuration for the objects being managed by the configuration file. The exact shape of the configuration varies according to both the **apiVersion** and the **kind** fields.

Declarative Object Management in Kubernetes Using

Approach	Description	YAML Usage
Imperative	Uses kubectl commands to create and manage objects manually.	<code>kubectl run, kubectl expose, kubectl delete pod</code>
Declarative	Uses YAML configuration files to define and manage objects.	<code>kubectl apply -f <file.yaml</code>

✓ Why Use Declarative Management?

- Ensures consistency across environments.
- Facilitates version control using Git.
- Allows automated deployments with tools like GitOps.

Imperative Object Management

```
$ kubectl create namespace ckad  
namespace/ckad created
```

```
$ kubectl run nginx --image=nginx -n ckad  
pod/nginx created
```

```
$ kubectl edit pod nginx -n ckad  
pod/nginx edited
```

Creates an object
e.g. a namespace

Creates a Pod object
in a namespace

Modifies the live
object e.g. a Pod

Declarative Object Management

```
$ kubectl create -f nginx-deployment.yaml  
deployment.apps/nginx-deployment created
```

```
$ kubectl apply -f nginx-deployment.yaml  
deployment.apps/nginx-deployment configured
```

```
$ kubectl delete -f nginx-deployment.yaml  
deployment.apps "nginx-deployment" deleted
```

Creates object if it
doesn't exist yet

Creates object or
update if it already
exists

Deletes object

Hybrid Approach

```
$ kubectl run nginx --image=nginx  
--dry-run=client -o yaml > nginx-pod.yaml  
  
$ vim nginx-pod.yaml  
  
$ kubectl create -f nginx-pod.yaml  
pod/nginx created
```

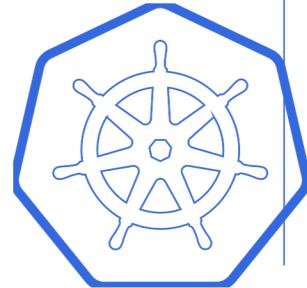
Capture YAML for
Pod object in file

Create the object
from YAML file

Understanding Kubernetes API Ecosystem

The Kubernetes API Ecosystem: A Structured Playground for Container Orchestration

- Serves as central nervous system for any internal & external communication within or outside a cluster
- Offers a programmatic interface for controlling & managing every aspect of a cluster



At the heart of Kubernetes lies its API (Application Programming Interface). Every action—whether creating a Pod, scaling a Deployment, or configuring a Service—ultimately goes through the Kubernetes API server. This API-centric design makes Kubernetes highly extensible and flexible, enabling a wide ecosystem of tools, controllers, and custom resources.

Kubernetes Object Model

Kubernetes organizes functionality around objects, each of which represents a specific cluster resource. Common examples include:

- Pod: The smallest deployable unit in Kubernetes, encapsulating one or more containers.
- Deployment: A higher-level concept that manages replica sets of Pods, providing rolling updates and rollbacks.
- Service: Defines a stable networking endpoint for a set of Pods.
- ConfigMap / Secret: Externalize configuration data for applications.
- PersistentVolume / PersistentVolumeClaim: Abstract storage resources.

Each resource in Kubernetes has a kind, apiVersion, metadata, spec, and status:

- kind: Indicates the type of resource (e.g., Deployment, Service).
- apiVersion: Specifies the API group and version (e.g., apps/v1).
- metadata: Contains identifying data like name, namespace, and labels.
- spec: The desired state of the resource.
- status: The observed state of the resource (populated by the system).

API Groups and Versions

Kubernetes organizes resources into API groups, each with one or more versions. Examples include:

- Core Group (no prefix, often apiVersion: v1): Contains core resources like Pods, Services, ConfigMaps.
- apps/v1: Contains workloads such as Deployments, StatefulSets, and DaemonSets.
- batch/v1: Contains Jobs and CronJobs.
- networking.k8s.io/v1: Contains Ingress, NetworkPolicy, etc.

These groups and versions allow Kubernetes to evolve while maintaining backward compatibility.

Controllers and Reconciliation Loops

Kubernetes is built around the controller pattern. A controller continuously:

- Watches the API server for changes in objects (e.g., new Deployment).
- Observes the current state of the cluster.
- Reconciles the difference between the desired state (in the object's spec) and the actual state (in the cluster).

For example, a Deployment controller ensures that the number of running Pods matches the replicas specified in the Deployment's spec. If Pods crash or are removed, the controller creates new ones to maintain the desired count.

Custom Resources and Operators

One of the most powerful features of the Kubernetes API ecosystem is extensibility via:

- CRDs (Custom Resource Definitions): Allow users to define their own resource types (e.g., MyDatabase, MyCache).
- Operators: Controllers that implement operational logic for these custom resources. For instance, a “database operator” might automatically handle backups, failover, and scaling for a custom Database resource.

By defining CRDs and Operators, organizations can encapsulate operational best practices directly into the Kubernetes API, turning it into a robust “platform for platforms.”

Interacting with the API

You can interact with the Kubernetes API in multiple ways:

- kubectl (CLI): The primary command-line tool for sending requests to the API server.
- YAML/JSON Manifests: Declarative files describing the desired state of resources.
- Client Libraries: Official libraries in Go, Python, Java, JavaScript, etc., that let you write custom applications and controllers interacting with the Kubernetes API.
- API Endpoints: Directly make REST calls to the Kubernetes API server, typically secured by certificates or tokens.

Ecosystem and Tools

Because of its open and well-defined API, Kubernetes has a rich ecosystem of tools:

- Helm (package manager) uses charts to manage collections of YAML manifests.
- Kustomize (built into kubectl) customizes YAML without forking.
- Prometheus, Grafana, Jaeger (observability and monitoring) integrate with the Kubernetes API to watch cluster metrics and events.
- Service Meshes (e.g., Istio, Linkerd) hook into Kubernetes resources to provide traffic routing, security, and telemetry.

All these tools leverage the same Kubernetes API for consistent, centralized management.

Why “Structured Playground”?

- Structured: The API defines clear resource schemas and versioning. Controllers enforce the desired state, ensuring consistency and predictability.
- Playground: The flexibility of CRDs, Operators, and a robust plugin ecosystem means users can experiment with new ideas, build advanced workloads, and integrate a wide range of technologies, all within the Kubernetes framework.

Update Strategies & Rollback

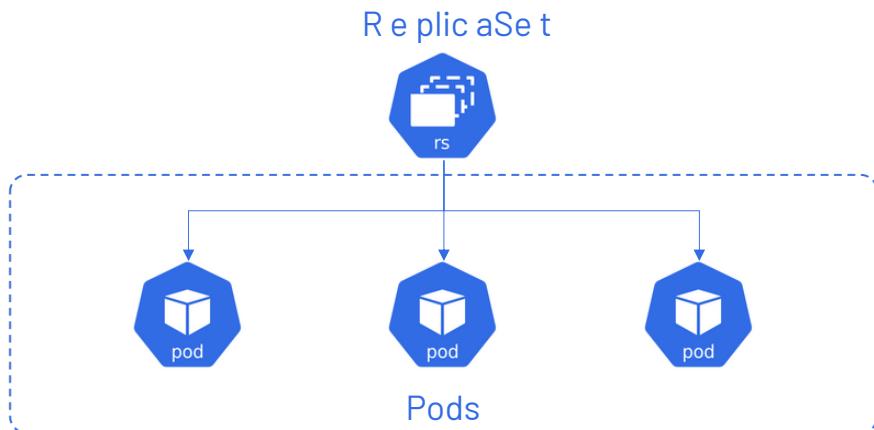
Recreate Strategy

Deployment Configuration

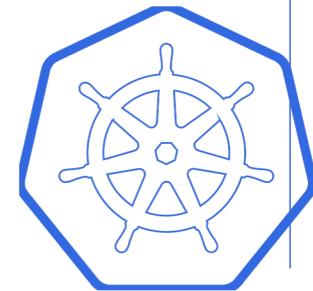
Image version

Number of replicas

Other parameters



✓ Imperative
✓ Declarative

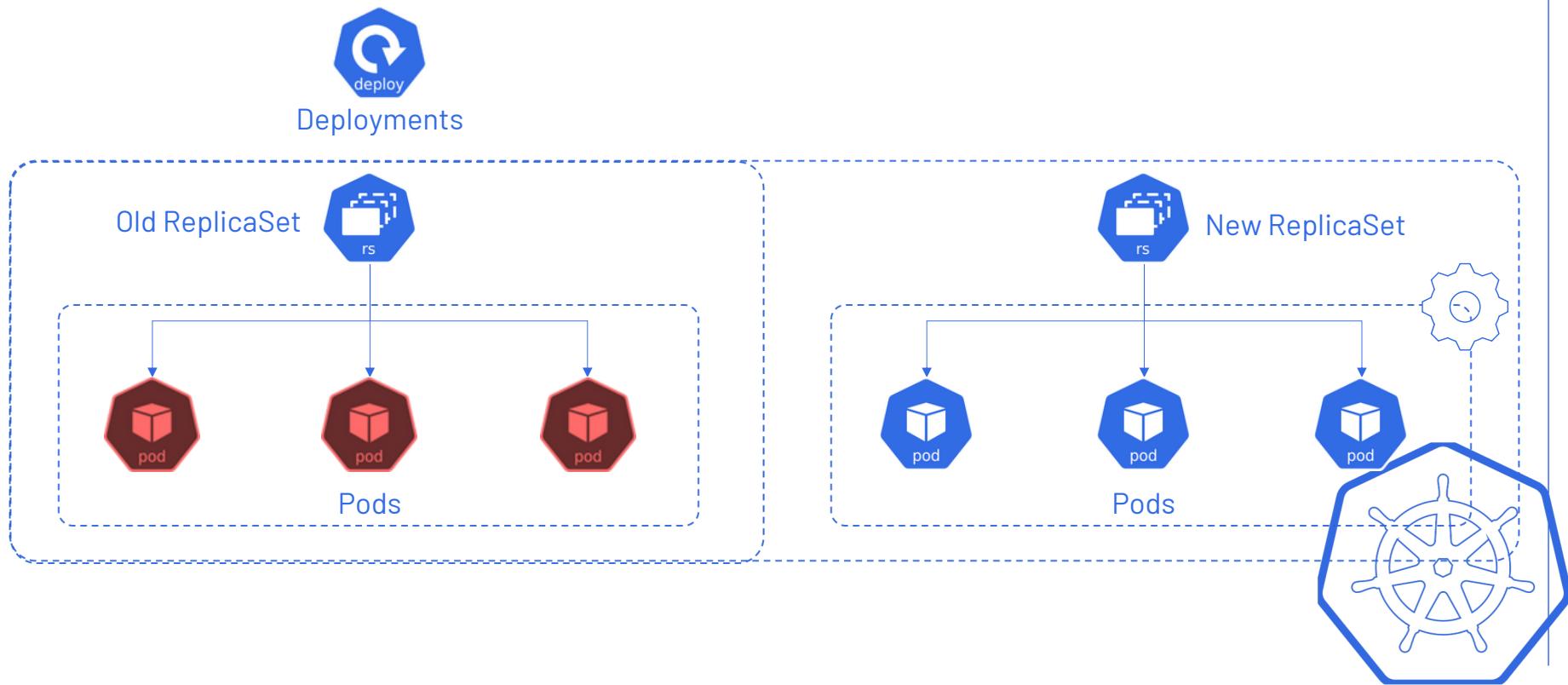


The Recreate strategy is one of the update strategies available for Kubernetes Deployments. With this strategy, Kubernetes will terminate all existing Pods before starting the new ones during an update. This approach contrasts with the default RollingUpdate strategy, which gradually replaces Pods, ensuring that some instances of the application remain available during the update.

Key Points about Recreate Strategy

- Behavior:
 1. When an update is triggered, the Recreate strategy shuts down all existing Pods first and then creates new Pods using the updated configuration.
- Use Cases:
 1. When your application cannot tolerate running two different versions simultaneously.
 2. When you need to ensure a clean start for the new version.
 3. When there are resource constraints and you prefer not to run multiple versions concurrently.
- Considerations:
 1. There is a downtime window as the old Pods are terminated before new ones start.
 2. It is simpler than a rolling update since you are not coordinating the overlap of old and new Pods.

Recreate Strategy



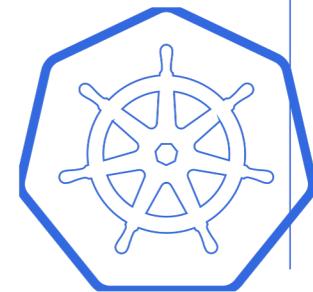
Recreate Strategy

Benefits

- ✓ Deployments are generally faster to implement
- ✓ Update process happens swiftly
- ✓ Rapid delivery of features or fixes

Drawback

- ✓ Downtime



Imperative Approach

Create an Initial Deployment

Create a Deployment using the default RollingUpdate strategy:

```
kubectl create deployment nginx-deployment --image=nginx:1.14.2 --replicas=3
```

Use kubectl patch to change the update strategy to Recreate:

```
kubectl patch deployment nginx-deployment --patch '{"spec": {"strategy": {"type": "Recreate"}}}'
```

Update the image to a new version. With the Recreate strategy, all Pods will be terminated before new ones are created:

```
kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1
```

Watch the rollout status:

```
kubectl rollout status deployment/nginx-deployment
```

```
kubectl get pods -w
```

Declarative Approach

Create a YAML Manifest with Recreate Strategy

Save the following content as nginx-recreate.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

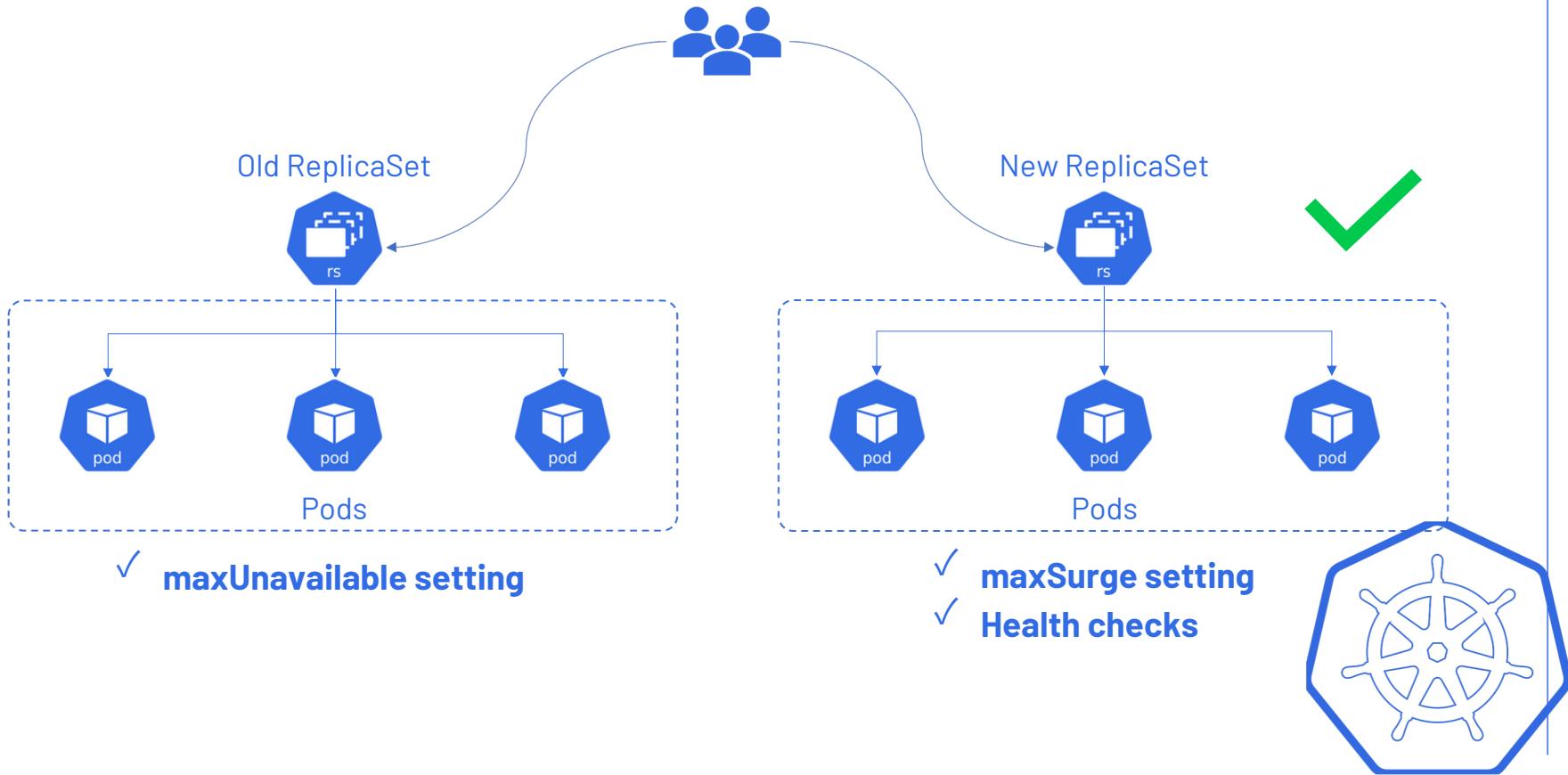
kubectl apply -f nginx-recreate.yaml

kubectl rollout status deployment/nginx-deployment

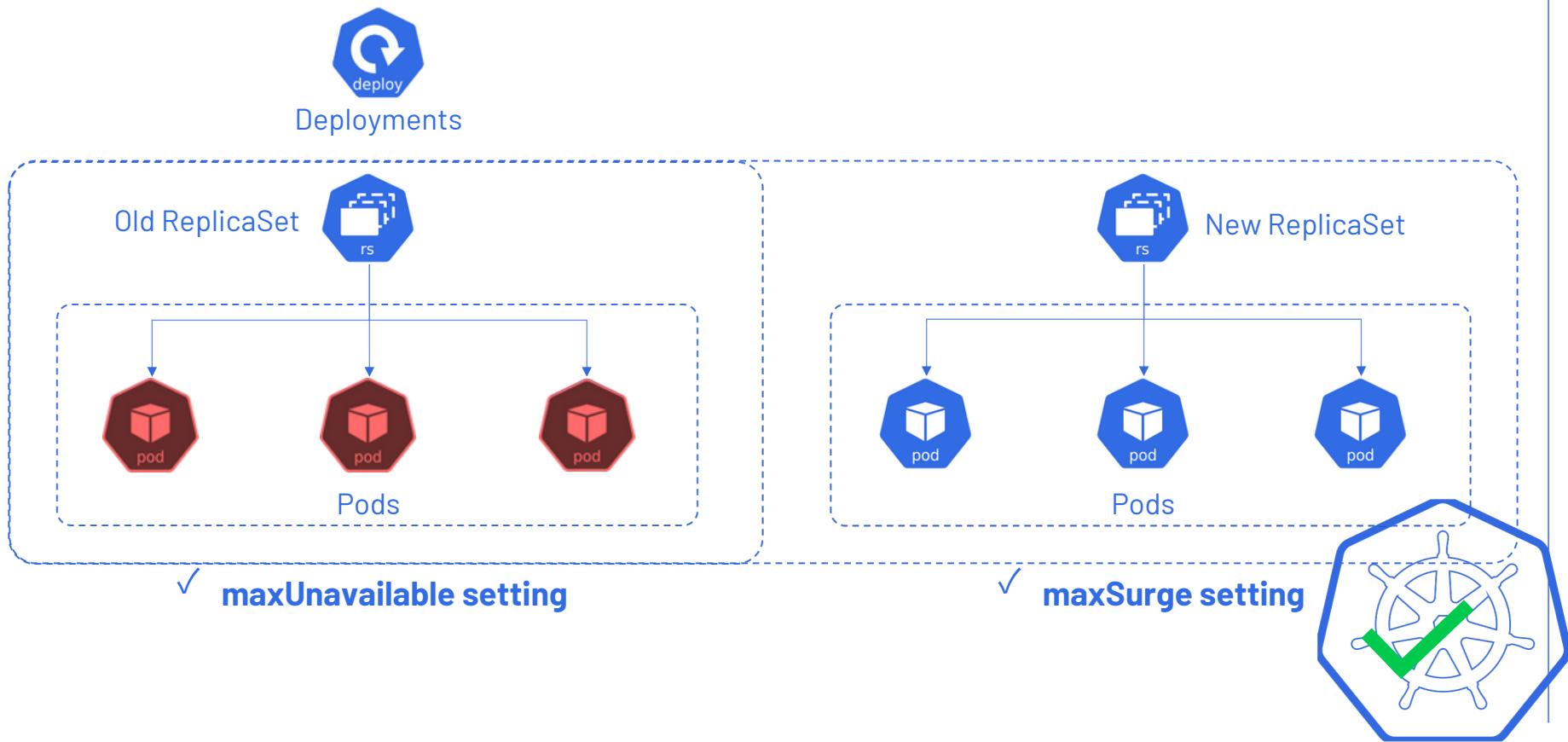
```
# Updated nginx-recreate.yaml snippet
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.16.1
          ports:
            - containerPort: 80
```

Rolling Update

Rolling Update Strategy



Rolling Update Strategy



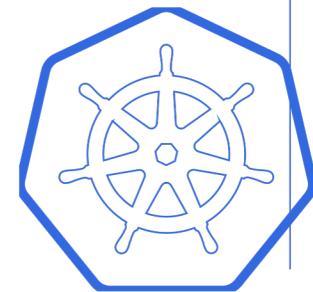
Rolling Update Strategy

Benefits

- ✓ Minimize service disruption

Drawback

- ✓ Gradual Update Process



The Rolling Update strategy is the default update mechanism in Kubernetes for Deployments. It allows you to update your application with minimal downtime by gradually replacing the old Pods with new ones. During a rolling update, Kubernetes creates new Pods using the updated configuration and terminates the old Pods incrementally while ensuring that a defined minimum number of Pods remains available.

Key Points About Rolling Update Strategy

- Gradual Update:

New Pods are started before the old ones are terminated, ensuring that some instances of your application remain available at all times.

- Control Over Update Process:

You can specify parameters like `maxUnavailable` (the maximum number of Pods that can be unavailable during the update) and `maxSurge` (the maximum number of extra Pods that can be created) to fine-tune the rollout.

- Default Strategy:

When you create a Deployment without specifying an update strategy, Kubernetes uses `RollingUpdate` by default.

Create a YAML Manifest with RollingUpdate Strategy

Create a file named `nginx-rollingupdate.yaml` with the following content. Even though `RollingUpdate` is the default, you can explicitly define it along with `maxUnavailable` and `maxSurge`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

maxUnavailable: 1

- During the update, at most 1 Pod can be taken down.
- This guarantees that at least 2 Pods are always running.

maxSurge: 1

- During the update, up to 1 extra Pod (beyond the desired count of 3) can be created.
- This means you might see 4 Pods running at one point, ensuring the new version is available faster.

These settings help ensure minimal downtime during a rolling update.

`kubectl apply -f nginx-rollingupdate.yaml`

`kubectl rollout status deployment/nginx-deployment`

```
# Updated nginx-rollingupdate.yaml snippet
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.16.1
          ports:
            - containerPort: 80
```

Choosing the Right Strategy

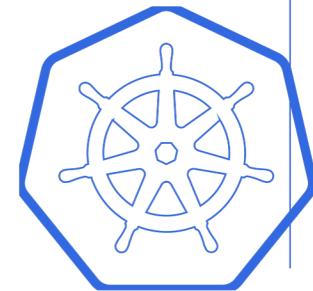
Choosing the Right Strategy

Recreate Strategy

Suitable for applications that can tolerate downtime

Rolling Update Strategy

Ideal for applications that need continuous availability during updates



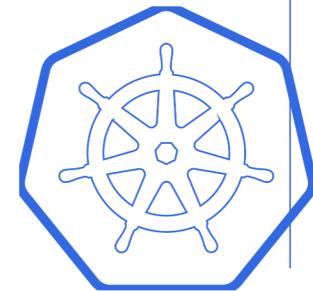
Rollback

Rollback

- Process of reverting application deployment to previous version

Undo current rollout & rollback to last deployment version

Undo current rollout & rollback to specific revision by specifying its revision number



Rollback in Kubernetes refers to reverting a Deployment to a previous stable state if an update causes issues. Here's a concise explanation along with imperative and declarative demonstration steps.

Update Deployment (e.g., problematic update):

```
kubectl set image deployment/nginx-deployment nginx=nginx:1.18.0
```

```
kubectl rollout status deployment/nginx-deployment
```

```
kubectl rollout undo deployment/nginx-deployment
```

```
kubectl rollout status deployment/nginx-deployment
```

```
kubectl get deployment nginx-deployment -o yaml | grep image:
```

To roll back to a specific revision number, you can use the kubectl rollout undo command with the --to-revision flag.

```
kubectl rollout history deployment/nginx-deployment
```

Rollback to a Specific Revision: kubectl rollout undo deployment/nginx-deployment --to-revision=2

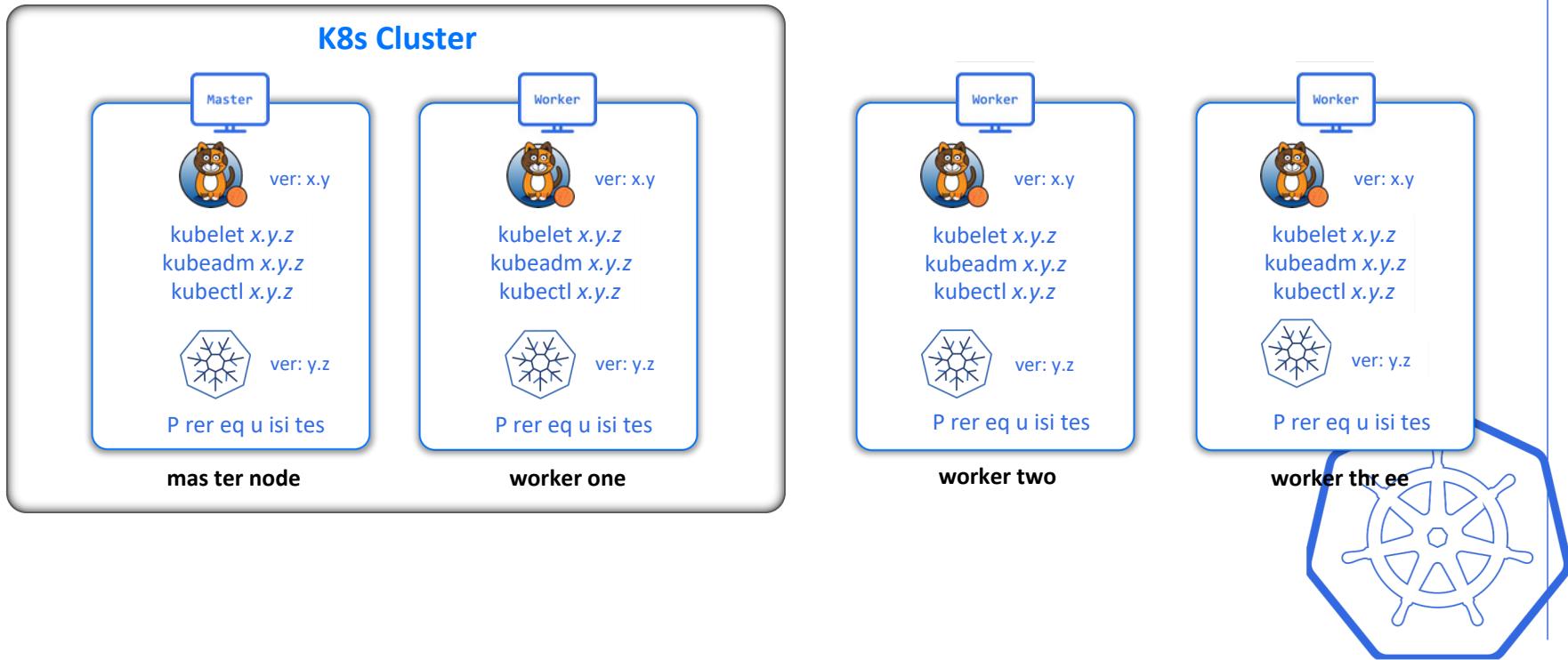
```
kubectl rollout status deployment/nginx-deployment
```

```
kubectl rollout history deployment/nginx-deployment
```

Kubernetes Cluster Maintenance

Scaling out a Cluster

Step 1: Provision New Worker Nodes



Scaling out a Cluster

Step 2: Generate a New Join Token

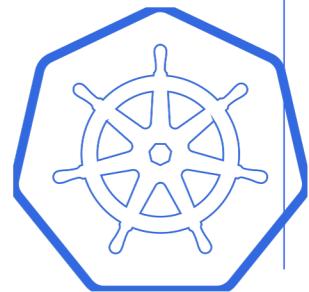
- K8s token expires after 24 hours

To add a new worker node, we need to generate a new token

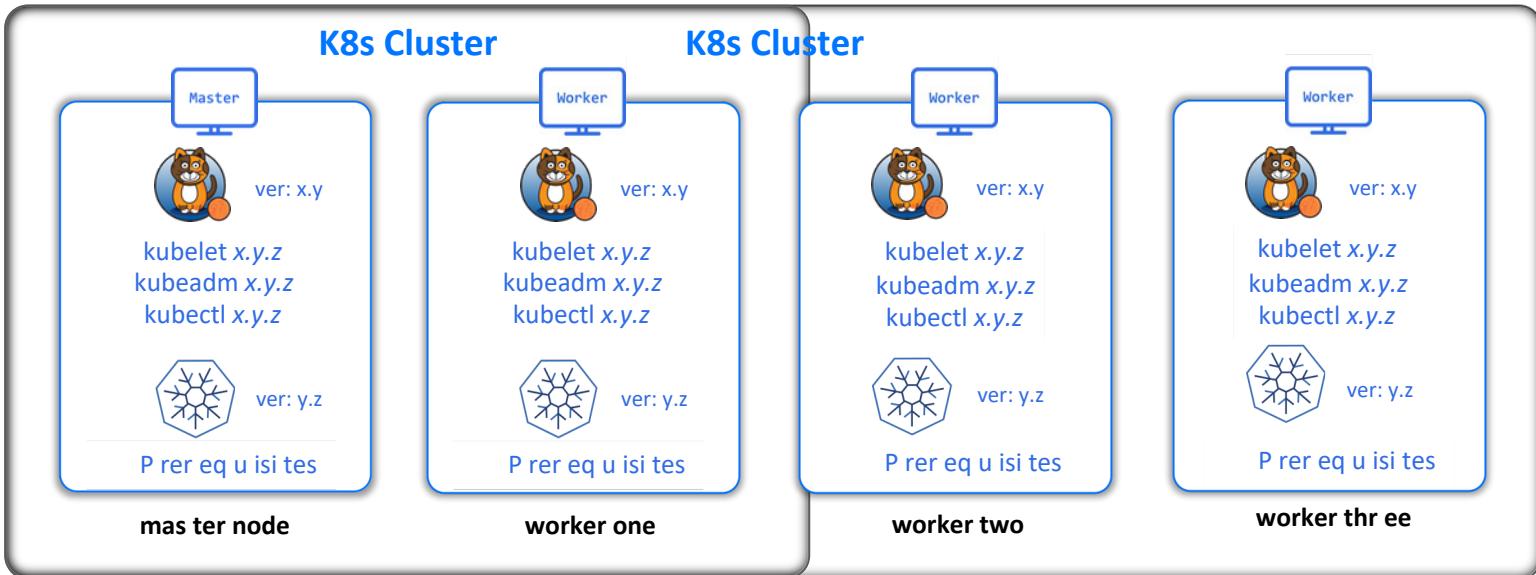


```
kubeadm token create --print-join-command
```

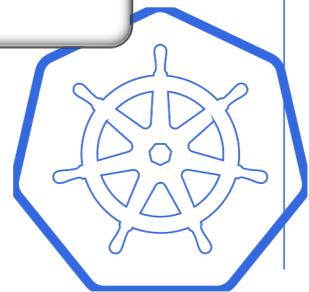
```
kubeadm join <master-node-ip>:6443 --token <token> --discovery-token-ca-cert-hash sha256:<hash>
```



Scaling out a Cluster



kubectl get nodes



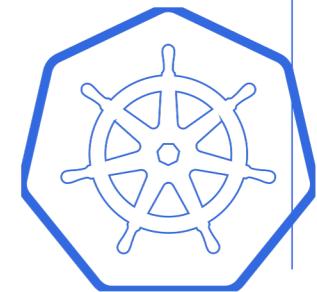
Cordon & Uncordon

Cordon & Uncordon

- Managing node availability in a Kubernetes cluster is key for planned maintenance or upgrades

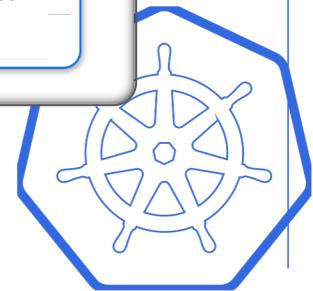
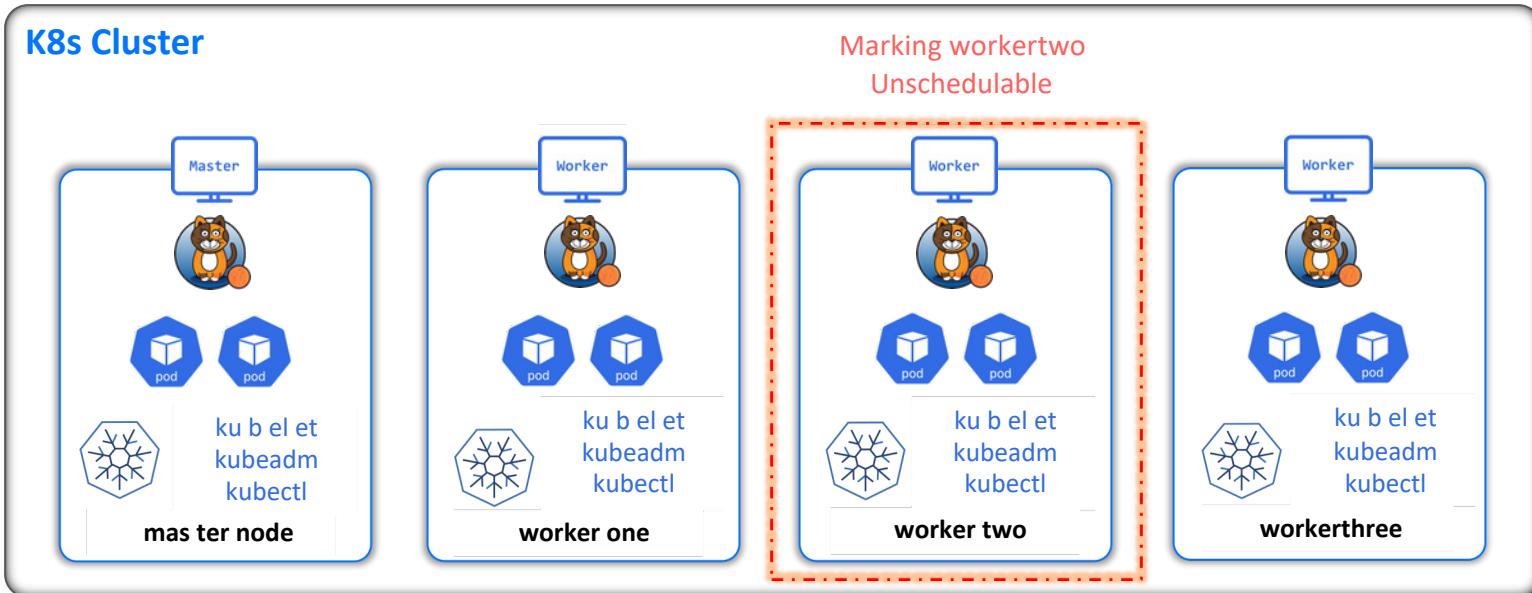
Cordon

Uncordon



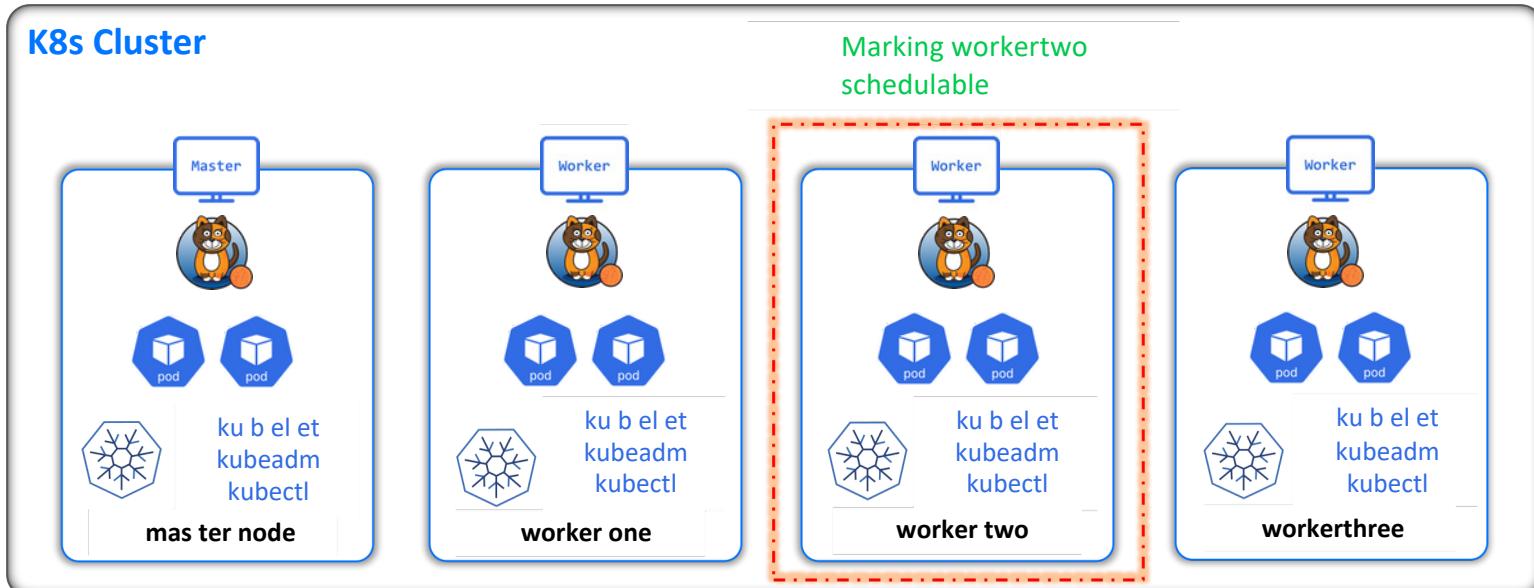
Cordon

- Cordoning a Kubernetes node stops new pods from being scheduled, allowing maintenance without disrupting current workloads

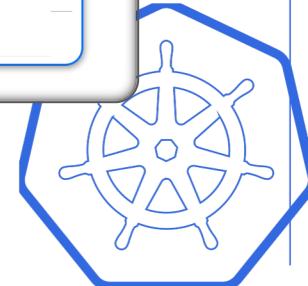


Uncordon

- After maintenance, uncordoning the node makes it schedulable again, allowing new pods to be scheduled



`kubectl uncordon workertwo`



Draining in Kubernetes

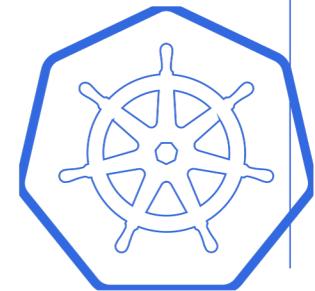
Draining in Kubernetes

- Cordon stops new pods from being scheduled but it doesn't evict existing ones
- Draining offers a mechanism to gracefully evicts existing pods for smooth maintenance or upgrades

```
kubectl drain <node-name>
```

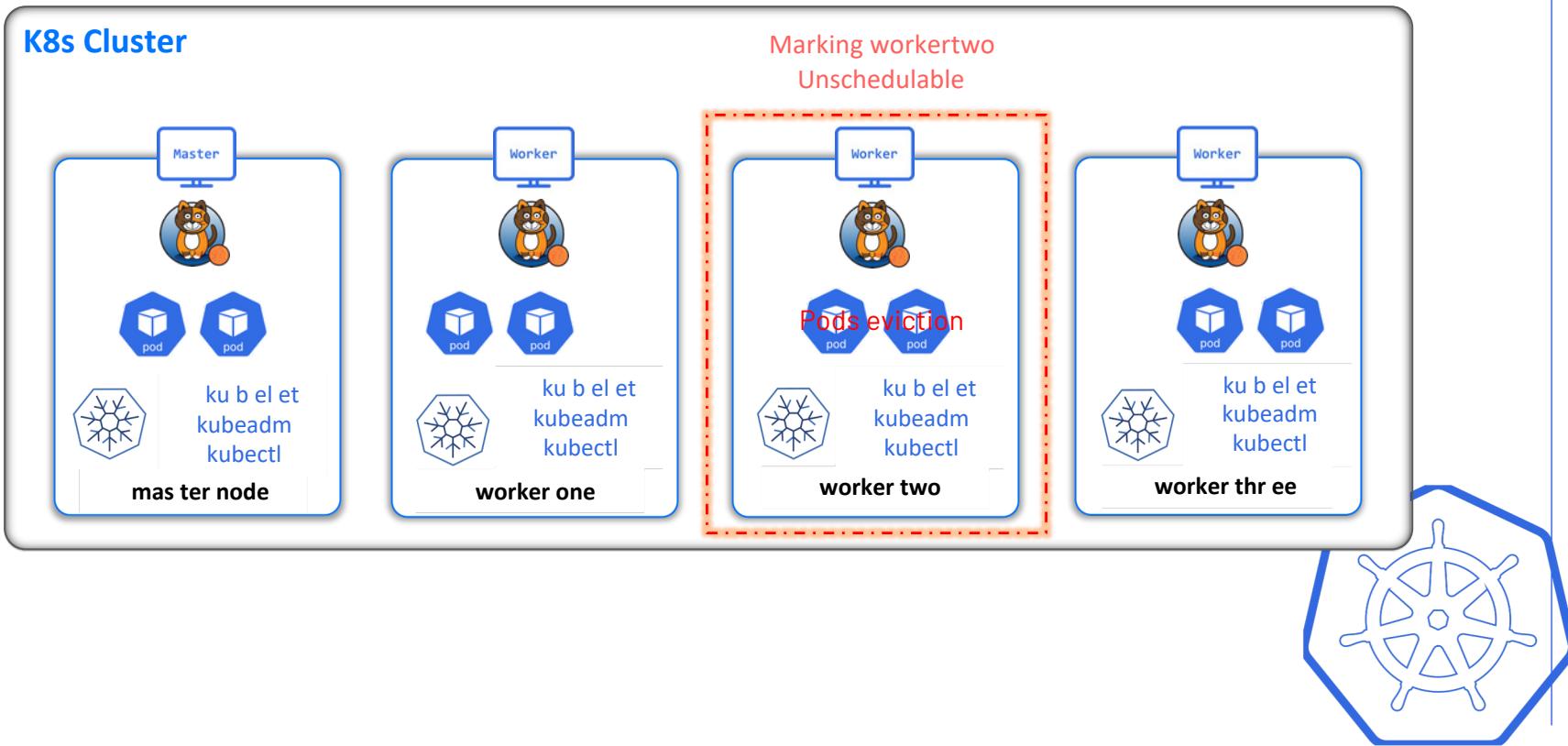
- **kubectl drain** cannot evict pods managed by DaemonSets

```
--ignore-daemonsets=true
```



Draining in Kubernetes

```
kubectl drain workertwo --ignore-daemonsets=true
```



- Create a pod and verify its scheduling on a worker node.
- Cordon the node (mark it unschedulable).
- Try deploying another pod (it should not be scheduled on the cordoned node).
- Drain the node (evict running pods).
- Uncordon the node (bring it back to normal operation).
- Deploy a new pod and check if it gets scheduled again.

kubectl get nodes

We'll create a simple Nginx pod and force it to be scheduled on worker-node-1.

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
spec:
  nodeSelector:           kubectl get pods -o wide
    kubernetes.io/hostname: worker-node-1
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
EOF
```

Cordon the Node (Mark it Unschedulable)

Now, we will prevent new pods from being scheduled on worker-node-1:

```
kubectl cordon worker-node-1
```

```
kubectl get nodes
```

```
worker-node-1 Ready,SchedulingDisabled worker 1d v1.26.0
```

Try Deploying Another Pod

Let's create another pod to see if it gets scheduled on worker-node-1

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: test-pod-2
spec:
  nodeSelector:
    kubernetes.io/hostname: worker-node-1
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
EOF
```

```
kubectl get pods
```

Since the node is cordoned, test-pod-2 is in the Pending state and won't be scheduled.

Drain the Node (Evict Pods)

Now, let's drain worker-node-1 to move all running workloads away from it.

```
kubectl drain worker-node-1 --ignore-daemonsets --delete-local-data --force
```

- `--ignore-daemonsets`: Ensures daemonset pods (e.g., logging or monitoring) are not evicted.
- `--delete-local-data`: Forces the deletion of pods using local storage.
- `--force`: Ensures pods without controllers (e.g., standalone pods) are also evicted.

```
kubectl get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	NODE	
test-pod	0/1	Terminating	0	6m	worker-node-1	test-pod is being terminated.
test-pod-2	0/1	Pending	0	1m	<none>	test-pod-2 is still pending because the node is unschedulable.

Uncordon the Node (Bring it Back Online)

Once maintenance is complete, bring worker-node-1 back into service:

```
kubectl uncordon worker-node-1
```

Deploy Another Pod and Verify Scheduling

```
kubectl delete pod test-pod-2
```

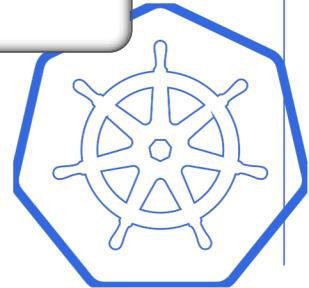
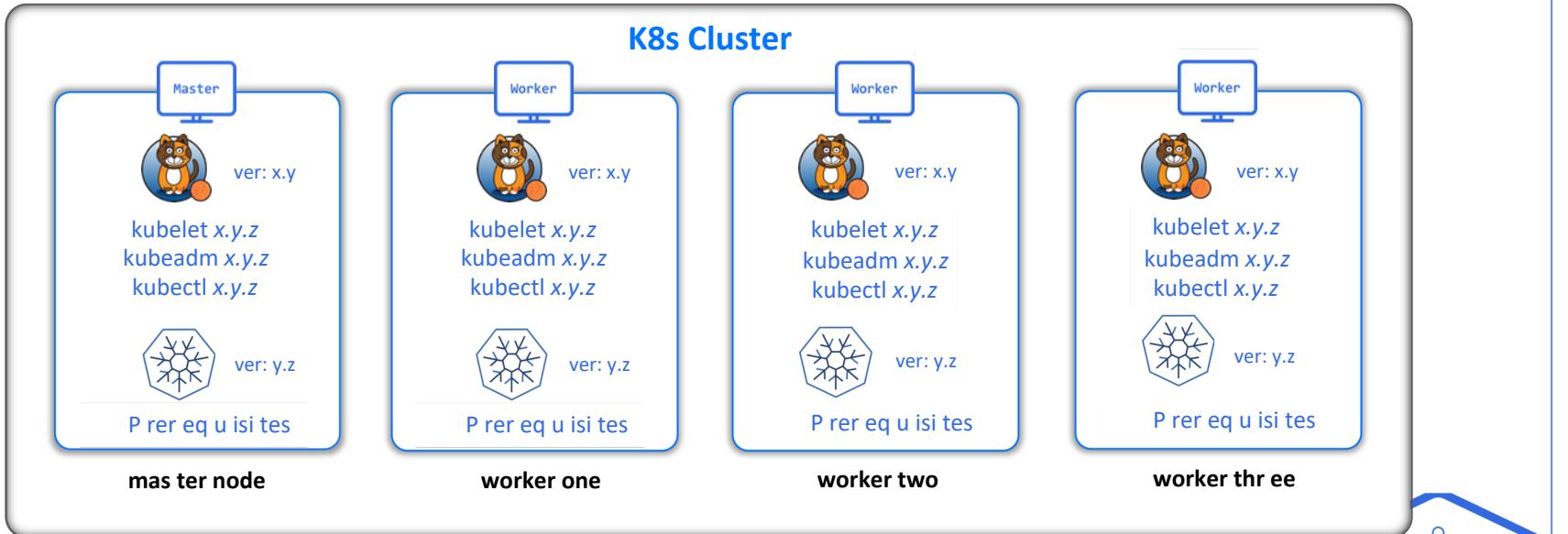
```
kubectl apply -f test-pod-2.yaml
```

- ✓ Cordon prevents new workloads from scheduling.
- ✓ Drain safely removes existing workloads.
- ✓ Uncordon restores scheduling to the node.
- ✓ Testing with a pod confirms behavior at each step.

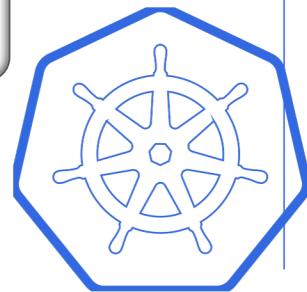
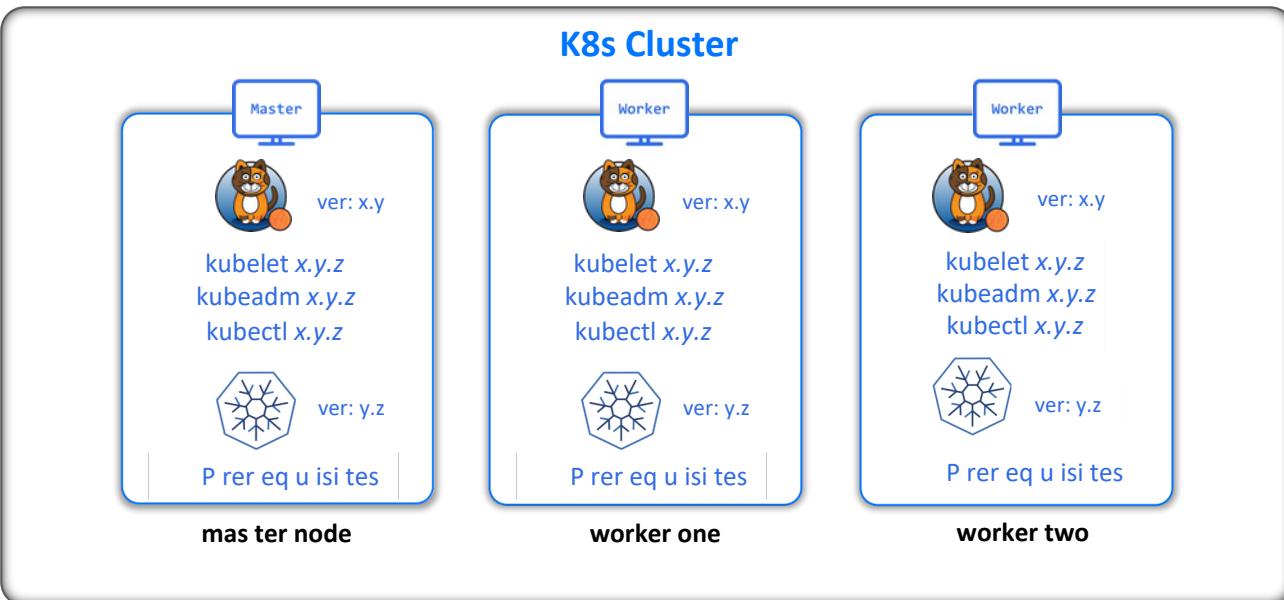
Using this process, you can perform maintenance safely in Kubernetes while ensuring minimal disruption to workloads.

Scaling in a Cluster

Scaling in a Cluster



Scaling in a Cluster



Scaling in a Cluster

Step 1: Drain a node (run below command from your master node)

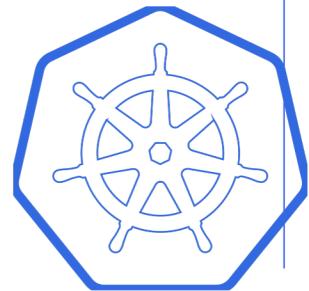
```
kubectl drain <node-name> --ignore-daemonsets=true
```

Step 2: Reset a node (run below command from your worker node)

```
kubeadm reset
```

Step 3: Delete a node (run below command from your master node)

```
kubectl delete node <nodename>
```



Useful Kubernetes Objects

DaemonSet

Jobs & CronJob

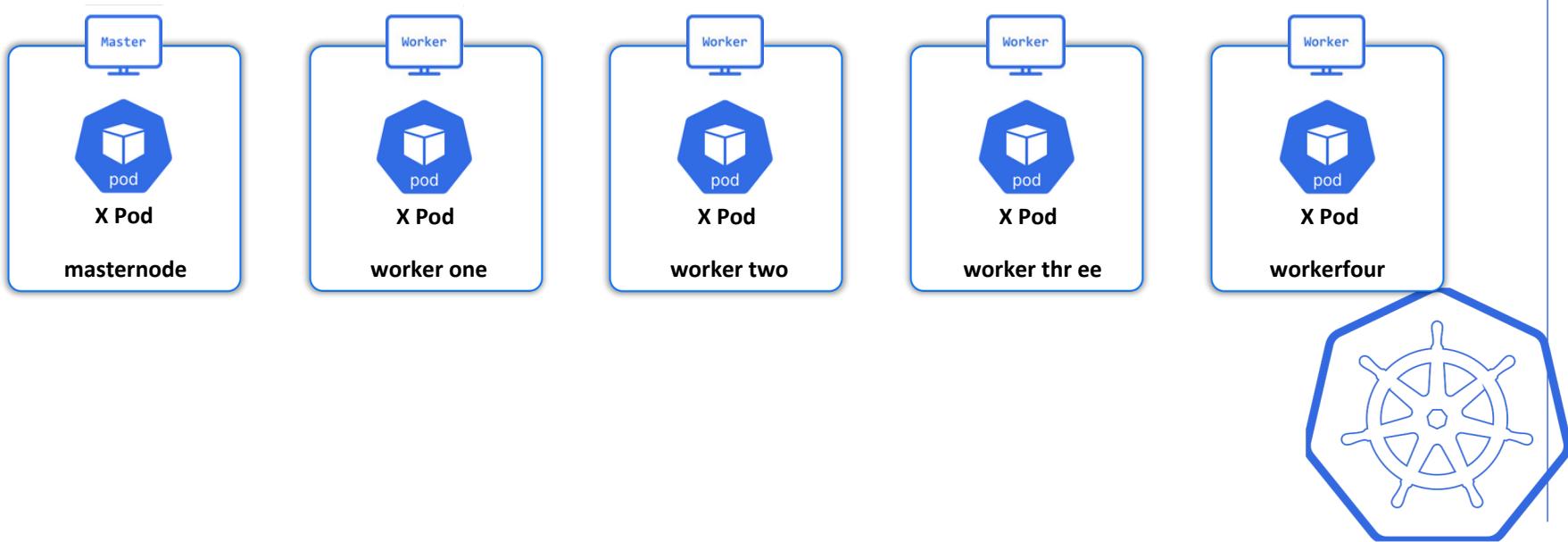
Static Pods

ConfigMaps & Secrets

DaemonSet

DaemonSet

- DaemonSet in Kubernetes ensures a specified pod runs on every node in the cluster, making it ideal for system-level tasks like log collection, monitoring, or networking



DaemonSet

- DaemonSet in Kubernetes ensures a specified pod runs on every node in the cluster, making it ideal for system-level tasks like log collection, monitoring, or networking

Key Features

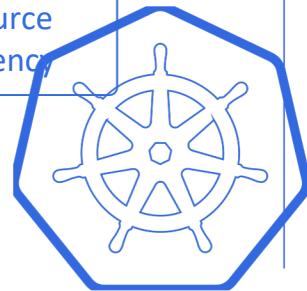
Uniform Deployment

Automatic Updates

Node Affinity>Selectors

Scalability

Resource Efficiency



DaemonSet



Logging



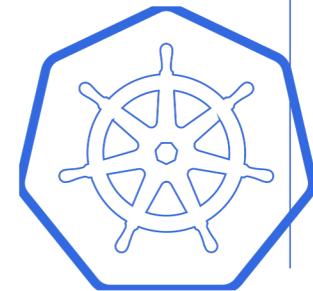
Monitoring



Networking



Security



A DaemonSet ensures that a copy of a Pod is automatically deployed on all (or specific) nodes in a Kubernetes cluster. It is commonly used for running cluster-wide system services such as log collection, monitoring, networking, and security tools.

Feature	Explanation
Ensures 1 Pod per Node	A DaemonSet ensures that a specific Pod runs on every node in the cluster.
Automatic Pod Scheduling	When a new node joins the cluster, a DaemonSet Pod is automatically created on it.
Pod Lifecycle Management	If a node is removed, the corresponding DaemonSet Pod is also removed.
Runs on All Nodes (or Select Nodes)	By default, runs on all nodes, but can be restricted using node selectors or taints and tolerations.
Common Use Cases	- Log collection (e.g., Fluentd, Filebeat) - Monitoring (e.g., Prometheus node exporter) - Networking (e.g., CNI plugins like Calico, Cilium)

- Create a DaemonSet that runs an nginx container on all nodes.
- Check how Pods are deployed across nodes.
- Test behavior when new nodes are added or removed.
- Apply node-specific scheduling.
- Demonstrate cordon, drain, and uncordon with DaemonSets.

```
kubectl get nodes
```

We will deploy an Nginx DaemonSet that ensures an Nginx pod runs on every node.

```
kubectl get daemonsets
```

```
cat <<EOF | kubectl apply -f -  
apiVersion: apps/v1  
kind: DaemonSet  
metadata:  
  name: nginx-daemonset  
  labels:  
    app: nginx  
spec:  
  selector:  
    matchLabels:  
      name: nginx  
  template:  
    metadata:  
      labels:  
        name: nginx  
    spec:  
      containers:  
      - name: nginx  
        image: nginx:latest  
        ports:  
        - containerPort: 80  
EOF
```

Restrict DaemonSet to Specific Nodes

DaemonSets can be restricted to specific nodes using node selectors.

Modify the existing DaemonSet:

```
kubectl patch daemonset nginx-daemonset --type='json' -p='[  
  {"op": "add", "path": "/spec/template/spec/nodeSelector", "value": {"kubernetes.io/hostname": "worker-node-1"}}  
]'
```

How DaemonSets Behave with Cordon, Drain, and Uncordon

DaemonSets do not get evicted when a node is drained unless you force it.

Cordon the Node

```
kubectl cordon worker-node-1
```

Now, new Pods cannot be scheduled on worker-node-1, but existing DaemonSet Pods keep running.

Drain the Node

```
kubectl drain worker-node-1 --ignore-daemonsets --delete-local-data --force
```

DaemonSet Pods remain because they are ignored by default during drain.

Uncordon the Node

```
kubectl uncordon worker-node-1
```

Now, the node is schedulable again, and new Pods can be assigned.

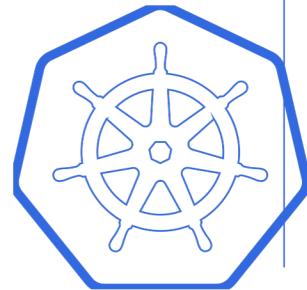
- ✓ DaemonSets ensure 1 Pod per node.
- ✓ Automatically schedules Pods on new nodes.
- ✓ DaemonSet Pods are NOT evicted by default during drain.
- ✓ Can be restricted to specific nodes using node selectors.
- ✓ Ideal for system services (logging, monitoring, networking).

Jobs

Jobs

- In Kubernetes, Jobs manage batch processing tasks by ensuring a set number of pods complete their work, ideal for one-time or finite tasks
- Typically used for tasks like data processing, backups, or sending emails

Remove “X Pod” after the task is completed



Jobs

- In Kubernetes, Jobs manage batch processing tasks by ensuring a set number of pods complete their work, ideal for one-time or finite tasks
- Typically used for tasks like data processing, backups, or sending emails

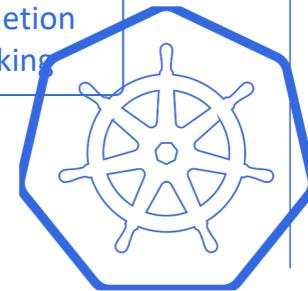
Key Features

Finite
Execution

Pod
Management

Parallelism

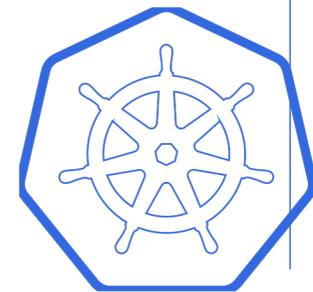
Completion
Tracking



Jobs

Types of Jobs

- ✓ Simple Jobs
- ✓ Parallel Jobs with a Fixed Completion Count
- ✓ Parallel Jobs with a Work Queue (Non-Parallel Jobs)



Jobs

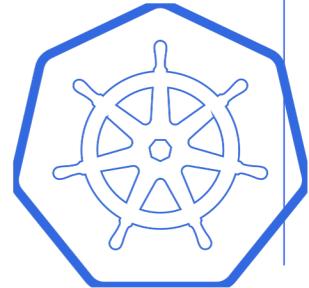
Use Cases

Data Processing

Database
Migrations

Scheduled
Maintenance

Event-Driven
Tasks



A Job in Kubernetes is used for running batch or one-time tasks. Unlike Deployments or DaemonSets, Jobs run Pods to completion and ensure the task finishes successfully. Jobs are often used for data processing, database migrations, and scheduled tasks.

Feature	Explanation
Run-to-Completion	Jobs run a Pod, complete execution, and exit when finished.
Automatic Retry on Failure	If a Job's Pod fails, Kubernetes will automatically restart it based on <code>restartPolicy</code> .
Parallel Execution	Jobs can run multiple Pods in parallel to speed up execution (controlled via <code>completions</code> and <code>parallelism</code>).
Pod Re-Creation	If a Pod is deleted before completion, Kubernetes will recreate it.
Common Use Cases	- Database migrations - Batch data processing - Scheduled tasks (when used with CronJobs)

- Create a simple Job that runs a task and exits.
- Test automatic re-creation on failure.
- Demonstrate a Job with parallel execution.
- Apply cordon, drain, and uncordon to see Job behavior.
- Delete a Job and verify its behavior.

Create a Simple Job

We will create a Job that runs an echo command and then exits.

```
cat <<EOF | kubectl apply -f -
apiVersion: batch/v1
kind: Job
metadata:
  name: simple-job
spec:
  template:
    spec:
      containers:
        - name: job-container
          image: busybox
          command: ["echo", "Hello from Kubernetes Job!"]
      restartPolicy: Never
EOF
```

Verify the Job Execution
Check if the Job is running

```
kubectl get jobs
kubectl get pods -o wide
kubectl logs simple-job-xyz
```

Test Job Failure Handling

Modify the Job to force a failure (simulate a crash).

```
cat <<EOF | kubectl apply -f -
apiVersion: batch/v1
kind: Job
metadata:
  name: fail-job
spec:
  template:
    spec:
      containers:
        - name: job-container
          image: busybox
          command: ["exit", "1"] # This will fail
      restartPolicy: Never
EOF
```

kubectl get jobs

kubectl get pods -o wide

Create a Parallel Job

We can run multiple Pods in parallel by setting completions and parallelism.

```
cat <<EOF | kubectl apply -f -
apiVersion: batch/v1
kind: Job
metadata:
  name: parallel-job
spec:
  completions: 3
  parallelism: 2
  template:
    spec:
      containers:
        - name: job-container
          image: busybox
          command: ["echo", "Running Parallel Job"]
    restartPolicy: Never
EOF
```

`kubectl get jobs`

completions: 3

The Job will run 3 Pods sequentially (if parallelism is not set).
The Job is finished only when all 3 Pods complete successfully.

parallelism: 2

A maximum of 2 Pods will run simultaneously.

If completions: 3, Kubernetes will run:

- 2 Pods first, wait for them to complete.
- Then 1 more Pod to reach the total of 3 completions.

Delete a Job

Since Jobs create Pods that do not restart, deleting a Job removes all associated Pods.

`kubectl delete job parallel-job`

CronJob

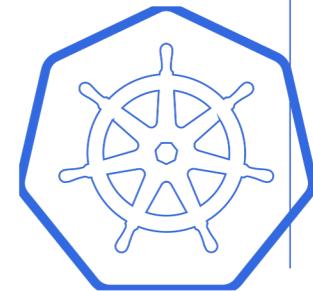
CronJob

- CronJob in Kubernetes schedules recurring tasks at specified intervals, similar to Unix cron, for regular operations like backups and report generation
- Ideal for regularly scheduled tasks like backups, report generation, or periodic data processing

Schedule and Remove “X Pod” after the task is completed to run periodically at fixed times



Monday - 9:00 AM
Tuesday - 9:00 AM
Wednesday - 9:00 AM



A CronJob in Kubernetes is used to schedule Jobs to run at specific times (similar to Linux cron). It is useful for recurring tasks, such as backups, report generation, or log rotation.

Feature	Explanation
Automated Scheduling	Runs Jobs at specified intervals using a cron schedule.
Handles Failures	Failed Jobs can be retried automatically.
Time-Based Execution	Uses standard cron syntax to define schedules.
Pod Lifecycle Management	Creates a new Pod per scheduled Job and deletes old ones based on history settings.
Parallel Execution Support	Can run multiple Jobs at the same time.
Common Use Cases	- Automated backups - Data processing - Log rotation - Database cleanups

- Create a simple CronJob that runs every minute.
- Check scheduled Jobs and their execution.
- Demonstrate automatic retries on failure.
- Test parallel execution with concurrency policy.
- Use completions and parallelism in CronJobs.
- Apply cordon, drain, and uncordon to test behavior.
- Delete a CronJob and verify cleanup.

Create a Simple CronJob

We will create a CronJob that runs every minute (`*/1 * * * *`) and prints "Hello from Kubernetes CronJob!".

```
cat <<EOF | kubectl apply -f -
apiVersion: batch/v1
kind: CronJob
metadata:
  name: simple-cronjob
spec:
  schedule: "*/1 * * * *" # Runs every minute
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: cron-container
              image: busybox
              command: ["echo", "Hello from Kubernetes CronJob!"]
          restartPolicy: Never
EOF
```

```
kubectl get cronjobs
```

```
kubectl get jobs
```

```
kubectl get pods -o wide
```

```
kubectl logs simple-cronjob-xyz-abc12
```

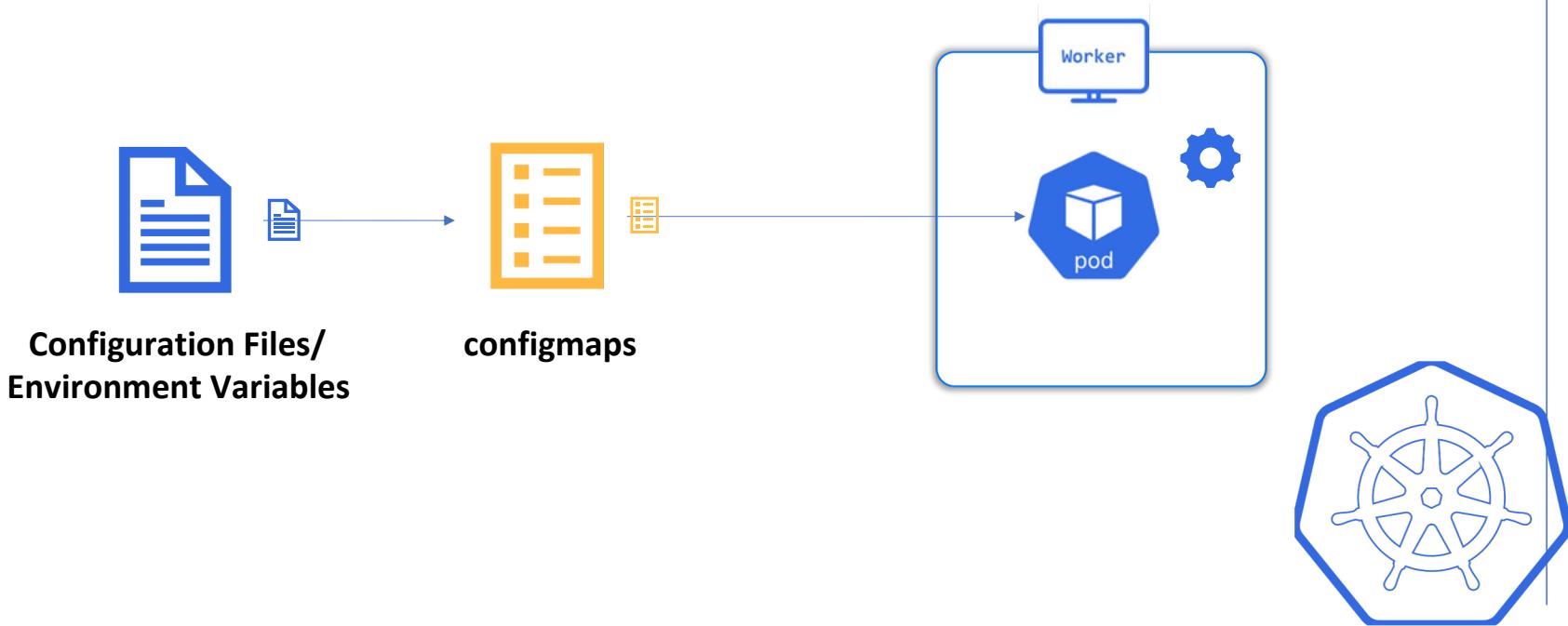
ConfigMaps

A ConfigMap in Kubernetes is used to store configuration data separately from application code. This allows decoupling of application logic from its configuration, making deployments more flexible and portable

Feature	Explanation
Decouples Configuration	Stores non-sensitive key-value pairs for applications.
Can Be Mounted as Files	ConfigMaps can be mounted as volumes inside Pods.
Can Be Used as Environment Variables	Allows injecting config values as env variables in containers.
Easier Config Updates	Allows modifying application behavior without changing container images.
Not for Sensitive Data	Use Secrets instead for passwords, API keys, and sensitive info.
Common Use Cases	- Application settings (e.g., URLs, timeouts) - Feature flags - Database connection details

ConfigMaps

- ConfigMaps in Kubernetes store configuration data as key-value pairs, allowing you to manage and update configurations independently from container images



ConfigMaps

- ConfigMaps in Kubernetes store configuration data as key-value pairs, allowing you to manage and update configurations independently from container images

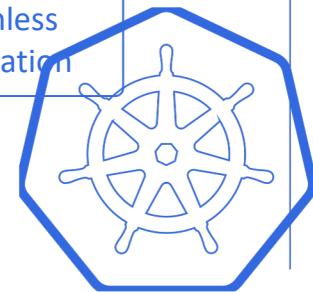
Key Features

Centralized Management

Dynamic Updates

Versatile Data Sources

Seamless Integration



We will:

- Create a ConfigMap using both imperative and declarative approaches.
- Use a ConfigMap in a Pod as environment variables.
- Use a ConfigMap as a mounted volume inside a Pod.
- Modify a ConfigMap and test dynamic updates.
- Apply cordon, drain, and uncordon to test behavior.
- Delete a ConfigMap and verify its impact.

Step 2: Create a ConfigMap

Method 1: Imperative Approach

Create a ConfigMap from CLI:

```
kubectl create configmap my-config --from-literal=APP_MODE=production --from-literal=APP_VERSION=v1.0  
kubectl get configmaps  
kubectl describe configmap my-config
```

Method 2: Declarative Approach

Create a ConfigMap using a YAML file:

```
cat <<EOF | kubectl apply -f -  
apiVersion: v1  
kind: ConfigMap  
metadata:  
  name: my-config  
data:  
  APP_MODE: "production"  
  APP_VERSION: "v1.0"  
EOF
```

```
kubectl get configmaps
```

Use ConfigMap as Environment Variables

Now, we will inject ConfigMap values into a Pod as environment variables.

```
cat <<EOF | kubectl apply -f -  
apiVersion: v1  
kind: Pod  
metadata:  
  name: env-pod  
spec:  
  containers:  
    - name: busybox  
      image: busybox  
      command: ["sh", "-c", "echo APP_MODE=$APP_MODE && echo  
APP_VERSION=$APP_VERSION && sleep 3600"]  
  envFrom:  
    - configMapRef:  
        name: my-config  
EOF
```

```
kubectl get pods -o wide
```

```
kubectl logs env-pod
```

Use ConfigMap as a Mounted Volume

Modify the ConfigMap to add multiple configuration values:

```
cat <<EOF | kubectl apply -f -  
apiVersion: v1  
kind: ConfigMap  
metadata:  
  name: config-volume  
data:  
  app.properties: |  
    APP_MODE=production  
    APP_VERSION=v1.0  
EOF
```

```
cat <<EOF | kubectl apply -f -  
apiVersion: v1  
kind: Pod  
metadata:  
  name: volume-pod  
spec:  
  containers:  
    - name: busybox  
      image: busybox  
      command: ["sleep", "3600"]  
  volumeMounts:  
    - name: config-volume  
      mountPath: "/etc/config"  
  volumes:  
    - name: config-volume  
      configMap:  
        name: config-volume  
EOF
```

```
kubectl get pods -o wide  
kubectl exec -it volume-pod -- ls /etc/config  
kubectl exec -it volume-pod -- cat /etc/config/app.properties
```

Modify ConfigMap and Test Dynamic Updates

Edit the ConfigMap:

```
kubectl edit configmap config-volume
```

Modify:

data:

```
app.properties: |  
  APP_MODE=development  
  APP_VERSION=v2.0
```

```
kubectl exec -it volume-pod -- cat /etc/config/app.properties
```

🚀 The update is reflected instantly without restarting the Pod!

Delete a ConfigMap

Deleting a ConfigMap will affect only new Pods that try to use it. Existing Pods using mounted ConfigMaps will retain their previous values.

```
kubectl delete configmap my-config
```

```
kubectl get configmaps
```

- ✓ ConfigMaps store non-sensitive configurations.
- ✓ Can be used as environment variables or mounted as files.
- ✓ Changes are dynamically reflected in mounted volumes.
- ✓ Draining a node can cause Pods to be rescheduled with new ConfigMap values.
- ✓ Deleting a ConfigMap affects only new Pods.

Secrets

A Secret in Kubernetes is a secure way to store sensitive information such as passwords, API keys, SSH keys, or TLS certificates. Unlike ConfigMaps, Secrets are stored in a base64-encoded format and are not visible in plain text when using kubectl describe.

Feature	Explanation
Stores Sensitive Data Securely	Secrets store confidential information like passwords and API keys.
Base64 Encoded	Secret values are encoded in base64, preventing casual visibility.
Can Be Used as Env Variables	Applications can access Secrets through environment variables.
Can Be Mounted as Volumes	Secrets can be mounted inside Pods as files for secure access.
Access Control	Kubernetes RBAC (Role-Based Access Control) can restrict access to Secrets.
Common Use Cases	- Storing database credentials - API keys for third-party services - TLS certificates - SSH private keys

Types of Secrets

Type	Description	Use Case
Opaque	Default type, used for arbitrary key-value pairs.	API keys, passwords, generic secrets.
TLS (kubernetes.io/tls)	Stores a TLS certificate and key.	HTTPS/TLS encryption.
Docker Registry (kubernetes.io/dockerconfigjson)	Stores credentials for private container registries.	Pulling images from private registries.
BasicAuth (kubernetes.io/basic-auth)	Stores a username and password.	Web service authentication.
SSH Key (kubernetes.io/ssh-auth)	Stores an SSH private key.	Secure SSH authentication.
Service Account Token (kubernetes.io/service-account-token)	Used for API authentication.	Accessing the Kubernetes API securely.

We will:

Create a Secret using both imperative and declarative approaches.

Use a Secret as environment variables in a Pod.

Use a Secret as a mounted volume.

Test modification of a Secret.

Delete a Secret and verify its impact.

Create a Secret

Method 1: Imperative Approach

Create a Secret from CLI using --from-literal

```
kubectl create secret generic my-secret --from-literal=DB_USER=admin --from-literal=DB_PASSWORD=SuperSecret123
```

```
kubectl get secrets
```

```
kubectl describe secret my-secret
```

Declarative Approach

Create a Secret using a YAML file:

```
cat <<EOF | kubectl apply -f -  
apiVersion: v1  
kind: Secret  
metadata:  
  name: my-secret  
type: Opaque  
data:  
  DB_USER: $(echo -n "admin" | base64)  
  DB_PASSWORD: $(echo -n "SuperSecret123" | base64)  
EOF
```

```
kubectl get secrets
```

```
kubectl get pods  
kubectl logs secret-env-pod
```

🚀 The application successfully retrieves secret values as environment variables!

Use a Secret as Environment Variables

Now, we will inject Secret values into a Pod as environment variables.

```
cat <<EOF | kubectl apply -f -  
apiVersion: v1  
kind: Pod  
metadata:  
  name: secret-env-pod  
spec:  
  containers:  
    - name: busybox  
      image: busybox  
      command: ["sh", "-c", "echo DB_USER=\$DB_USER && echo  
DB_PASSWORD=\$DB_PASSWORD && sleep 3600"]  
    env:  
      - name: DB_USER  
        valueFrom:  
          secretKeyRef:  
            name: my-secret  
            key: DB_USER  
      - name: DB_PASSWORD  
        valueFrom:  
          secretKeyRef:  
            name: my-secret  
            key: DB_PASSWORD
```

```
EOF
```

Use a Secret as a Mounted Volume

Modify the Secret to store multiple credentials in a single file:

```
cat <<EOF | kubectl apply -f -  
apiVersion: v1  
kind: Secret  
metadata:  
  name: secret-volume  
type: Opaque  
data:  
  db-config: $(echo -e "DB_USER=admin\nDB_PASSWORD=SuperSecret123" | base64)  
EOF
```

```
kubectl get pods  
kubectl exec -it secret-volume-pod -- ls /etc/secret
```

Expected Output: db-config

```
kubectl exec -it secret-volume-pod -- cat /etc/secret/db-config  
Expected Output:
```

```
DB_USER=admin  
DB_PASSWORD=SuperSecret123
```

Now, create a Pod that mounts this Secret as a file:

```
cat <<EOF | kubectl apply -f -  
apiVersion: v1  
kind: Pod  
metadata:  
  name: secret-volume-pod  
spec:  
  containers:  
    - name: busybox  
      image: busybox  
      command: ["sleep", "3600"]  
  volumeMounts:  
    - name: secret-volume  
      mountPath: "/etc/secret"  
  volumes:  
    - name: secret-volume  
      secret:  
        secretName: secret-volume  
EOF
```

Modify a Secret and Test Updates

Edit the Secret:

```
kubectl edit secret secret-volume
```

Modify:

data:

```
db-config: $(echo -e "DB_USER=newadmin\nDB_PASSWORD=NewSecret456" | base64)
```

```
kubectl exec -it secret-volume-pod -- cat /etc/secret/db-config
```

Delete a Secret

Deleting a Secret affects only new Pods that try to use it. Existing Pods will retain their previous values.

```
kubectl delete secret my-secret
```

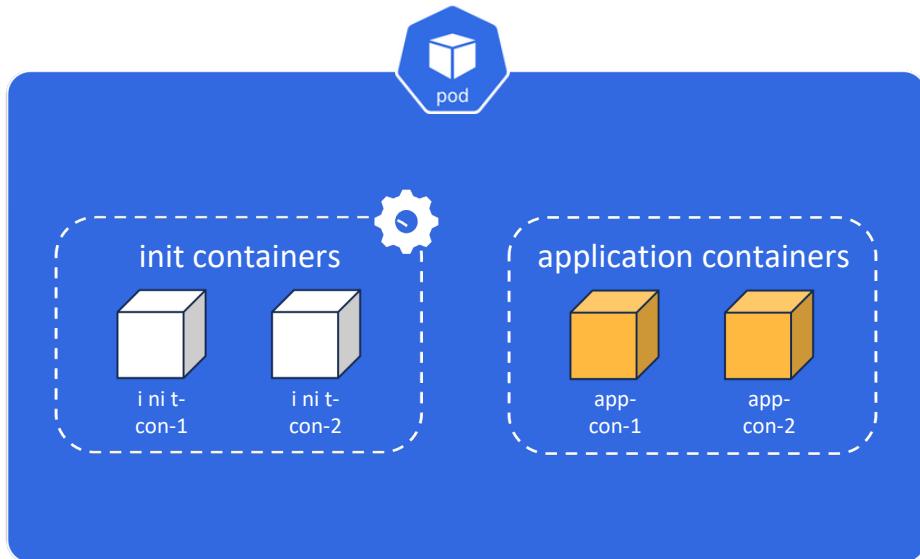
- ✓ Secrets store sensitive data securely.
- ✓ Base64-encoded values prevent casual visibility.
- ✓ Can be injected as environment variables or mounted as files.
- ✓ Modifying a Secret updates mounted values instantly.
- ✓ Deleting a Secret affects only new Pods.

Advanced Pod Tasks

Init Containers

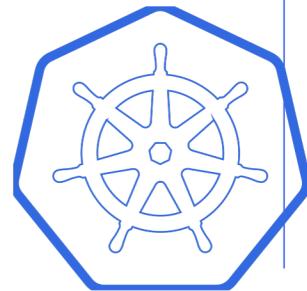
Init Containers

- Init containers are specialized containers that run before application containers in a Pod to perform setup scripts or utilities not included in the application image



Key characteristics:

- ✓ Run to Completion
- ✓ Sequential Execution
- ✓ Failure Handling



An Init Container is a specialized container that runs before the main application containers in a Pod. It is used for preparing the environment before the application starts, such as setting up configurations, waiting for dependencies, or performing database migrations.

Feature	Explanation
Runs Before Application Containers	Ensures pre-setup tasks are completed before the main application starts.
Sequential Execution	Multiple Init Containers execute in the order they are defined.
Prepares Application Environment	Useful for setting up files, waiting for services, or downloading dependencies.
Must Complete Successfully	A Pod does not start until all Init Containers have completed successfully.
Uses Different Images	Init Containers can use different container images than the main application.
Common Use Cases	- Waiting for database readiness - Configuring application settings - Fetching secrets before the app starts

We will:

- Create a Pod with an Init Container that prepares the environment.
- Verify Init Container execution.
- Modify an Init Container and observe behavior.
- Delete and recreate the Pod to see Init Containers in action.

Create a Pod with an Init Container

The following Pod uses an Init Container to simulate a pre-start task (sleep for 10 seconds to mimic a service dependency check). After it completes, the main application (Nginx) starts.

```
cat <<EOF | kubectl apply -f -  
apiVersion: v1  
kind: Pod  
metadata:  
  name: init-container-pod  
spec:  
  initContainers:  
    - name: init-setup  
      image: busybox  
      command: ["sh", "-c", "echo Initializing... && sleep 10"]  
  containers:  
    - name: nginx  
      image: nginx  
      ports:  
        - containerPort: 80  
EOF
```

kubectl get pods

Init:0/1 means the Init Container is still running.
Once it completes, the main application (nginx) will start.

kubectl describe pod init-container-pod

Modify an Init Container

Let's modify the Init Container to simulate a failure.

```
cat <<EOF | kubectl apply -f -  
apiVersion: v1  
kind: Pod  
metadata:  
  name: failing-init-container-pod  
spec:  
  initContainers:  
    - name: failing-init  
      image: busybox  
      command: ["sh", "-c", "echo Failing Init Container... && exit 1"]  
  containers:  
    - name: nginx  
      image: nginx  
      ports:  
        - containerPort: 80  
EOF
```

kubectl get pods

 The Pod does not start because the Init Container failed.

kubectl logs failing-init-container-pod -c failing-init

Use an Init Container to Prepare a Volume

Init Containers can write files that the main container uses. The following example:

The Init Container writes a config file inside a shared volume.

The main application reads the file and prints the contents.

```
cat <<EOF | kubectl apply -f -  
apiVersion: v1  
kind: Pod  
metadata:  
  name: init-volume-pod  
spec:  
  volumes:  
    - name: shared-data  
      emptyDir: {}  
  initContainers:  
    - name: write-config  
      image: busybox  
      command: ["sh", "-c", "echo 'APP_MODE=production' > /config/app.env"]  
  volumeMounts:  
    - name: shared-data  
      mountPath: /config  
  containers:  
    - name: nginx  
      image: busybox  
      command: ["sh", "-c", "cat /config/app.env && sleep 3600"]  
  volumeMounts:  
    - name: shared-data  
      mountPath: /config  
EOF
```

kubectl logs init-volume-pod

APP_MODE=production

- ✓ Init Containers run before main application containers.
- ✓ They execute sequentially and must complete before the main app starts.
- ✓ Used for preparing configurations, waiting for dependencies, or setting up files.
- ✓ Failures prevent the Pod from starting (Init:CrashLoop).
- ✓ Deleting a Pod restarts Init Containers from scratch.

Multicontainer Pod

A Multi-Container Pod is a Kubernetes Pod that runs multiple containers within the same Pod. These containers share resources like networking, storage, and memory but operate as separate processes. This setup is useful for building tightly coupled applications that need to communicate efficiently.

Feature	Explanation
Shared Network	All containers in a Pod share the same network namespace (localhost communication).
Shared Storage	Containers can mount shared volumes to exchange data.
Tightly Coupled Components	Useful when multiple processes need to work closely together within the same lifecycle.
Single Scheduling Unit	The entire Pod (with all containers) is scheduled as a single unit on a node.
Lifecycle Synchronization	If a Pod is restarted, all its containers are restarted together.
Common Use Cases	- Logging sidecars - Proxies and security agents - Data processors - Helper utilities

Types of Multi-Container Pods

There are three primary patterns for designing Multi-Container Pods:

Pattern	Description	Use Case
Sidecar	A helper container runs alongside the main application container.	Logging, monitoring, proxies.
Adapter	A lightweight container modifies or processes data from the main container.	Log parsing, data transformation.
Ambassador	Acts as a proxy container to handle communication inside or outside the Pod.	API gateways, network proxies.

We will:

- Create a Multi-Container Pod with a Sidecar for logging.
- Create an Adapter pattern Pod for transforming logs.
- Create an Ambassador pattern Pod for proxying network traffic.
- Verify inter-container communication.
- Delete and recreate Pods to see interactions.

Sidecar Pattern: Logging Sidecar

Scenario:

Main container runs an application (Nginx).

Sidecar container collects logs and prints them.

How It Works:

The Nginx container writes logs to /var/log/nginx/.

The Sidecar container reads these logs and prints them.

kubectl logs sidecar-pod -c logging-sidecar

🚀 The Sidecar continuously prints logs from the main container!

Force Nginx to Generate Logs

Manually send a request to Nginx from another Pod or the same Pod:

kubectl exec -it sidecar-pod -c app-container -- curl localhost

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: sidecar-pod
spec:
  volumes:
    - name: log-volume
      emptyDir: {}
  containers:
    - name: app-container
      image: nginx
      volumeMounts:
        - name: log-volume
          mountPath: /var/log/nginx
    - name: logging-sidecar
      image: busybox
      command: ["sh", "-c", "while true; do cat /var/log/nginx/access.log; sleep 5; done"]
      volumeMounts:
        - name: log-volume
          mountPath: /var/log/nginx
EOF
```

Adapter Pattern: Log Transformer

Scenario:

Main container generates logs.

Adapter container reads logs, modifies them, and saves them in uppercase format.

```
cat <<EOF | kubectl apply -f -  
apiVersion: v1  
kind: Pod  
metadata:  
  name: adapter-pod  
spec:  
  volumes:  
    - name: log-volume  
      emptyDir: {}  
  containers:  
    - name: app-container  
      image: busybox  
      command: ["sh", "-c", "echo 'error: something went wrong' > /log/output.log; sleep 3600"]  
      volumeMounts:  
        - name: log-volume  
          mountPath: /log  
    - name: log-adapter  
      image: busybox  
      command: ["sh", "-c", "while true; do cat /log/output.log | tr a-z A-Z > /log/processed.log; sleep 5; done"]  
      volumeMounts:  
        - name: log-volume  
          mountPath: /log  
EOF
```

How It Works:

Main container writes logs to /log/output.log.

Adapter container reads logs and converts them to uppercase in /log/processed.log.

kubectl exec adapter-pod -c log-adapter -- cat /log/processed.log

 The Adapter modifies log data dynamically!

Ambassador Pattern: Network Proxy

Scenario:

Main container connects to a mock database (app-container).

Ambassador container acts as a proxy for handling database communication.

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: ambassador-pod
spec:
  containers:
    - name: app-container
      image: busybox
      command: ["sh", "-c", "while true; do wget -qO- http://localhost:8080; sleep 5; done"]
    - name: network-proxy
      image: python:3.9
      command: ["sh", "-c", "echo 'Database connected!' > index.html && python3 -m http.server 8080"]
EOF
```

kubectl logs ambassador-pod -c network-proxy
kubectl exec ambassador-pod -c app-container -- wget -qO- http://localhost:8080

Resource Limits & Requests

Resource Limits & Requests in Kubernetes allow you to control the amount of CPU and memory (RAM) allocated to a container. These settings ensure that containers get the resources they need while preventing excessive resource usage.

Feature	Explanation
Resource Requests	Defines the minimum CPU and memory a container needs.
Resource Limits	Defines the maximum CPU and memory a container can use.
Ensures Fair Resource Allocation	Prevents a single container from consuming all available resources.
Affects Scheduling	Kubernetes schedules Pods based on Requests, ensuring enough resources are available.
Prevents Resource Starvation	Protects critical workloads by capping excessive resource usage.
Common Use Cases	- Ensuring fair CPU/memory allocation - Preventing Pods from crashing nodes - Managing workloads efficiently

Understanding Requests and Limits

Term	Definition	Example
CPU Request	The minimum CPU required for the container to be scheduled.	cpu: "500m" (500 millicores = 0.5 CPU core).
Memory Request	The minimum RAM required for the container to be scheduled.	memory: "256Mi" (256 Megabytes).
CPU Limit	The maximum CPU the container can use.	cpu: "1" (1 full CPU core).
Memory Limit	The maximum RAM the container can use.	memory: "512Mi" (512 Megabytes).

We will:

- Create a Pod with CPU & Memory Requests and Limits.
- Test CPU consumption with a stress test.
- Observe memory usage and out-of-memory kills (OOM).
- Modify Requests & Limits and redeploy.
- Delete and verify behavior.

Create a Pod with Resource Requests & Limits

The following Pod:

Requests 0.5 CPU and 256Mi RAM.

Limits CPU to 1 core and RAM to 512Mi.

kubectl get pods

kubectl describe pod resource-limited-pod

🚀 Pod is running with defined resource limits!

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: resource-limited-pod
spec:
  containers:
    - name: busybox
      image: busybox
      command: ["sh", "-c", "while true; do echo Running...; sleep 5; done"]
  resources:
    requests:
      cpu: "500m"    # Request 0.5 CPU core
      memory: "256Mi" # Request 256Mi RAM
    limits:
      cpu: "1"       # Max 1 CPU core
      memory: "512Mi" # Max 512Mi RAM
EOF
```

Test CPU Consumption

We will simulate high CPU usage inside the Pod to test resource limits.

```
kubectl exec -it resource-limited-pod -- sh
```

Run a CPU-intensive task:

```
yes > /dev/null &
```

Monitor CPU Usage: Open another terminal and run:

```
kubectl top pod resource-limited-pod
```

Test Memory Limits

Allocate Memory: **dd if=/dev/zero of=/dev/shm/test bs=1M count=600**
Inside the Pod:

Modify Resource Requests & Limits

Edit the Pod to increase CPU and memory limits:

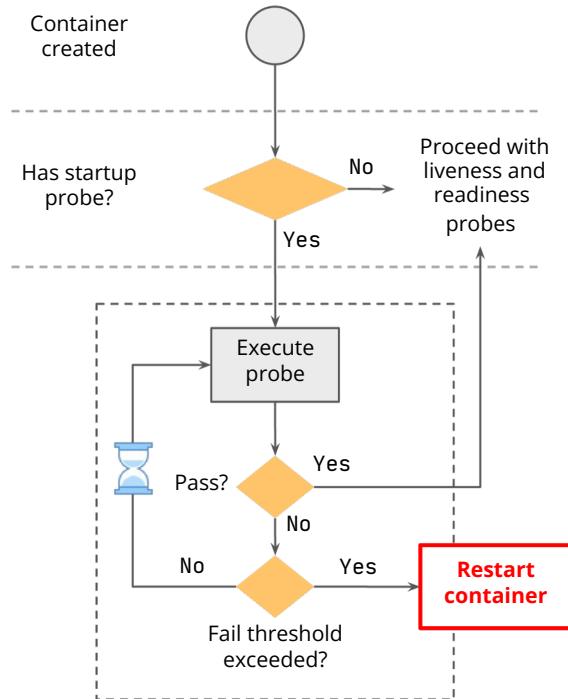
Kubernetes Liveness, Readiness & Startup Probes

Liveness, Readiness and Startup Probes

Continuously monitor container health

- Probes are periodic health checks performed by Kubernetes to determine the status of a container. They allow Kubernetes to manage the lifecycle of containers by checking if they are healthy and ready to serve traffic.
- Types of probes:
 - **Startup Probe:** Ensures that a container has started successfully. If the startup probe fails, Kubernetes will kill the container and try to restart it.
Liveness Probe: Checks if the container is still running. If the liveness probe fails, Kubernetes will restart the container. **Keeps executing throughout the container lifecycle.**
 - **Readiness Probe:** Checks if the container is ready to accept traffic. If the readiness probe fails, the container will be removed from the list of endpoints that receive traffic. **Keeps executing throughout the container lifecycle.**

Startup Probe



Startup Probe

Purpose: Ensures that an application has fully started before running Liveness and Readiness probes.

Process:

A container is created.

If a Startup Probe is defined, it runs the probe.

If the probe fails:

Kubernetes checks if the failure threshold is exceeded.

If exceeded, Kubernetes restarts the container.

If the probe passes, Kubernetes starts using Liveness & Readiness probes.

Use Case:

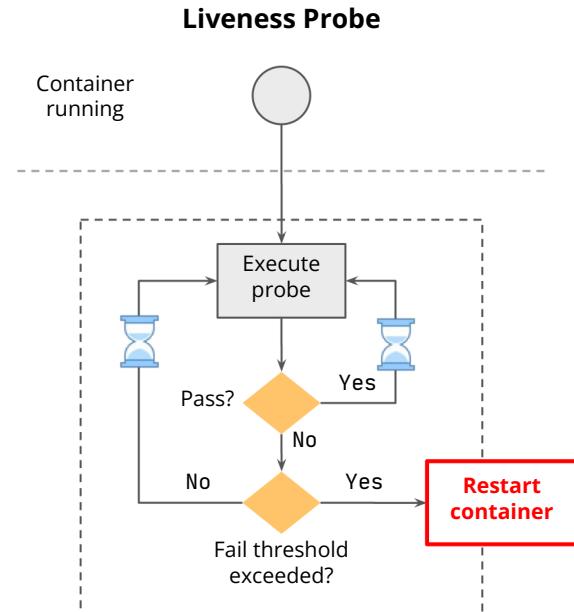
For slow-starting applications (e.g., Java Spring Boot apps, large ML models).

Prevents premature restarts caused by Liveness checks.

Liveness, Readiness and Startup Probes

Continuously monitor container health

- Probes are periodic health checks performed by Kubernetes to determine the status of a container. They allow Kubernetes to manage the lifecycle of containers by checking if they are healthy and ready to serve traffic.
- Types of probes:
 - **Startup Probe:** Ensures that a container has started successfully. If the startup probe fails, Kubernetes will kill the container and try to restart it.
 - **Liveness Probe:** Checks if the container is still running. If the liveness probe fails, Kubernetes will restart the container. **Keeps executing throughout the container lifecycle.**
 - **Readiness Probe:** Checks if the container is ready to accept traffic. If the readiness probe fails, the container will be removed from the list of endpoints that receive traffic. **Keeps executing throughout the container lifecycle.**



Liveness Probe

Key Parameters

`initialDelaySeconds`

Time to wait after container start before the first probe

`periodSeconds`

Interval between successive probes

`timeoutSeconds`

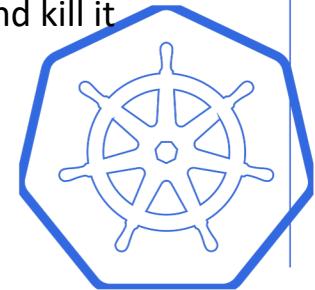
Time to wait for a probe to complete before timing out

`successThreshold`

Consecutive successes required to consider the probe successful after failure

`failureThreshold`

Consecutive failures required to mark the container as failed and kill it



Liveness Probe

Purpose: Checks whether the application is still alive (not stuck or crashed).

Process:

Kubernetes executes the probe at regular intervals.

If the probe passes, the container keeps running.

If the probe fails:

Kubernetes checks if the failure threshold is exceeded.

If exceeded, Kubernetes restarts the container.

Use Case:

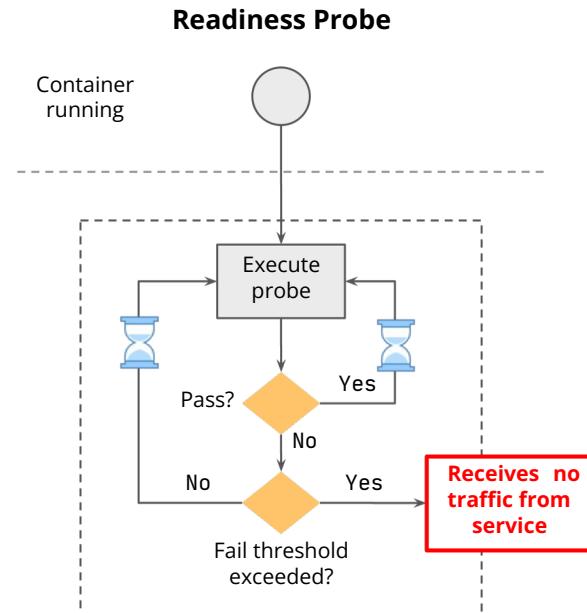
Detects application crashes, deadlocks, or infinite loops.

Ensures self-healing by automatically restarting broken containers.

Liveness, Readiness and Startup Probes

Continuously monitor container health

- Probes are periodic health checks performed by Kubernetes to determine the status of a container. They allow Kubernetes to manage the lifecycle of containers by checking if they are healthy and ready to serve traffic.
- Types of probes:
 - **Startup Probe:** Ensures that a container has started successfully. If the startup probe fails, Kubernetes will kill the container and try to restart it.
 - **Liveness Probe:** Checks if the container is still running. If the liveness probe fails, Kubernetes will restart the container. **Keeps executing throughout the container lifecycle.**
 - **Readiness Probe:** Checks if the container is ready to accept traffic. If the readiness probe fails, the container will be removed from the list of endpoints that receive traffic. **Keeps executing throughout the container lifecycle.**



Readiness Probe

Purpose: Determines if a container is ready to receive traffic.

Process:

Kubernetes executes the probe at regular intervals.

If the probe passes, the Pod remains in the Service load balancer.

If the probe fails:

Kubernetes checks if the failure threshold is exceeded.

If exceeded, the Pod stops receiving traffic but is NOT restarted.

📌 Use Case:

For apps that require initialization (e.g., database connections).

Removes unhealthy Pods from load balancers without restarting them.

We will:

- Create a Pod with Liveness, Readiness, and Startup Probes.
- Test failure conditions and observe behavior.
- Modify the probes and see how the application behaves.
- Delete and recreate the Pod to test probe execution.

Create a Pod with Liveness, Readiness, and Startup Probes

The following Pod:

Uses Liveness Probe to restart if the health check fails.

Uses Readiness Probe to delay traffic routing until it's ready.

Uses Startup Probe to wait before running Liveness checks.

```

cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: probe-demo
spec:
  containers:
  - name: my-app
    image: busybox
    command: ["/bin/sh", "-c", "sleep 30 && touch /tmp/ready && sleep 30 && rm -f /tmp/ready && sleep 600"]

    # Liveness Probe (checks if the app is alive)
    livenessProbe:
      exec:
        command: ["cat", "/tmp/ready"]
      initialDelaySeconds: 10
      periodSeconds: 5
      failureThreshold: 3

    # Readiness Probe (checks if the app is ready to receive traffic)
    readinessProbe:
      exec:
        command: ["cat", "/tmp/ready"]
      initialDelaySeconds: 5
      periodSeconds: 3
      failureThreshold: 2

    # Startup Probe (used for slow-starting applications)
    startupProbe:
      exec:
        command: ["cat", "/tmp/ready"]
      initialDelaySeconds: 10
      periodSeconds: 5
      failureThreshold: 10
EOF

```

Time	Event
0-30s	Pod is not ready (Startup Probe running).
30s	/tmp/ready is created → Readiness Probe passes, Pod becomes Ready.
60s	/tmp/ready is deleted → Liveness Probe fails.
75s	Kubernetes restarts the container due to Liveness failure.
120s	New container starts, the process repeats.

EOF

Here's a detailed explanation of each Kubernetes Probe Parameter:

Parameter	Definition
initialDelaySeconds	Time to wait after the container starts before the first probe runs. Prevents premature failures during startup.
periodSeconds	Time between successive probe executions. Defines how often the probe runs.
timeoutSeconds	Time to wait for a probe to complete before timing out. If the probe does not return success within this time, it is marked as failed.
successThreshold	Consecutive successful probe executions required to consider the probe healthy again after a failure.
failureThreshold	Consecutive failed probe executions required before marking the container as failed and taking action (restarting for Liveness, removing from traffic for Readiness).

```

apiVersion: v1
kind: Pod
metadata:
  name: probe-example
spec:
  containers:
    - name: my-app
      image: busybox
      command: ["./bin/sh", "-c", "sleep 30 && touch /tmp/ready && sleep 60 && rm -f /tmp/ready && sleep 600"]

```

Liveness Probe (Ensures the app is running)

```

livenessProbe:
  exec:
    command: ["cat", "/tmp/ready"]
  initialDelaySeconds: 10 # Wait 10s after container starts before the first probe
  periodSeconds: 5       # Run the probe every 5 seconds
  timeoutSeconds: 2      # If no response in 2 seconds, consider it failed
  successThreshold: 1    # 1 success is enough to mark as healthy
  failureThreshold: 3    # 3 consecutive failures will restart the container

```

Readiness Probe (Ensures the app is ready to receive traffic)

```

readinessProbe:
  exec:
    command: ["cat", "/tmp/ready"]
  initialDelaySeconds: 5 # Wait 5s before first probe
  periodSeconds: 3       # Run every 3 seconds
  timeoutSeconds: 1      # Probe timeout is 1 second
  successThreshold: 2    # Need 2 consecutive successes to be marked ready
  failureThreshold: 2    # 2 failures will mark the container as not ready

```

Startup Probe (For slow-starting apps)

```

startupProbe:
  exec:
    command: ["cat", "/tmp/ready"]
  initialDelaySeconds: 10 # Delay before the first probe runs
  periodSeconds: 5       # Check every 5 seconds
  timeoutSeconds: 2      # Timeout after 2 seconds if unresponsive
  successThreshold: 1    # 1 success is enough to proceed
  failureThreshold: 10   # Allow 10 failures before considering the startup failed

```

Scenario	Effect
initialDelaySeconds = 10	First probe starts 10 seconds after the container starts.
periodSeconds = 5	Probes are executed every 5 seconds after the initial delay.
timeoutSeconds = 2	If the probe does not respond within 2 seconds, it is marked as failed.
successThreshold = 2	If the container previously failed, it must succeed twice consecutively to be considered healthy again.
failureThreshold = 3	If the probe fails 3 times in a row, the container is restarted (Liveness) or removed from traffic (Readiness).

Kubernetes Node Scheduling Techniques

nodeName

Kubernetes provides multiple ways to control where Pods run inside a cluster. The Node Scheduler decides the placement of Pods based on node constraints, affinity rules, taints, and tolerations.

Node Schedulers in Kubernetes

How Scheduling Works

The Kubernetes Scheduler is responsible for assigning Pods to Nodes.

It considers resource availability, constraints, and affinity rules.

By default, the scheduler automatically places Pods where resources are available.

When to Override Scheduling?

When you want to pin a Pod to a specific Node.

When you want to spread Pods across different Nodes.

When some Nodes are dedicated for specific workloads.

Assigning Pods to Specific Nodes

nodeName (Direct Node Assignment)

Assigns a Pod directly to a specific Node.

If the Node is unavailable, the Pod stays in Pending state.

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: specific-node-pod
spec:
  nodeName: worker-node-1
  containers:
  - name: nginx
    image: nginx
EOF
```

- ◆ Use Case: Pinning a Pod to a specific node for debugging or special workloads.

nodeSelector (Simple Key-Value Match)

Assigns a Pod to a Node that matches a specific label.

Nodes must have labels defined.

```
kubectl label nodes worker-node-1 environment=production
```

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: node-selector-pod
spec:
  nodeSelector:
    environment: production
  containers:
  - name: nginx
    image: nginx
EOF
```

- ◆ Use Case: Assigning workloads to Nodes based on labels (e.g., gpu=true, storage:ssd).

Node Affinity & Anti-Affinity

Node Affinity & Anti-Affinity allow Kubernetes to control where a Pod is scheduled based on Node labels and conditions.

What is Node Affinity & Anti-Affinity?

Feature	Description
Node Affinity	Defines rules to schedule Pods on specific Nodes based on labels.
Node Anti-Affinity	Defines rules to avoid scheduling Pods on certain Nodes.

Example Use Cases:

- ✓ Running Pods on Nodes with SSD storage or GPUs.
- ✓ Ensuring database Pods do not run on testing Nodes.
- ✓ Spreading workloads only on specific groups of Nodes.

Types of Node Affinity & Anti-Affinity

Type	Description	Effect
requiredDuringSchedulingIgnoredDuringExecution	Mandatory: Pod must be scheduled on matching Nodes.	If no matching Node is found, the Pod stays Pending.
preferredDuringSchedulingIgnoredDuringExecution	Preferred: Kubernetes tries to place the Pod on matching Nodes.	If no matching Node is found, the Pod is still scheduled elsewhere.
requiredDuringSchedulingRequiredDuringExecution	Mandatory: If a Node label changes, the Pod is evicted.	Ensures strict compliance with affinity rules.

We will:

Label Nodes for scheduling.

Use Node Affinity to pin Pods to Nodes.

Use Node Anti-Affinity to prevent scheduling on certain Nodes.

Modify rules and test behavior.

```
kubectl label nodes worker-node-1 storage=ssd
```

```
kubectl get nodes --show-labels
```

Required Node Affinity (Strict Scheduling)

Deploy a Pod that must be scheduled on worker-node-1:

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: node-affinity-pod
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: storage
            operator: In
            values:
            - ssd
  containers:
  - name: nginx
    image: nginx
EOF
```



The Pod is scheduled on worker-node-1 because it matches the label (storage=ssd)!

Preferred Node Affinity (Soft Scheduling)

Now, let's deploy a Pod that prefers worker-node-1 but will run elsewhere if necessary.

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: preferred-node-affinity-pod
spec:
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 1
          preference:
            matchExpressions:
              - key: storage
                operator: In
                values:
                  - ssd
  containers:
    - name: nginx
      image: nginx
EOF
```

 If worker-node-1 had no space, Kubernetes scheduled it on worker-node-2!

Required Node Anti-Affinity (Avoid Specific Nodes)

Now, let's prevent a Pod from running on worker-node-1.

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: node-anti-affinity-pod
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: storage
            operator: NotIn
            values:
            - ssd
  containers:
  - name: nginx
    image: nginx
EOF
```

The Pod avoids worker-node-1 because of the anti-affinity rule.

kubectl label node worker-node-1 storage=standard --overwrite

If the Pod had the requiredDuringSchedulingRequiredDuringExecution rule, it would be evicted!

Inter-Pod Affinity & Anti-Affinity

Inter-Pod Affinity and Anti-Affinity allow Kubernetes to control how Pods are scheduled relative to each other. They help group related workloads together or distribute workloads for high availability.

Feature	Description
Pod Affinity	Ensures that Pods run on the same Node or in close proximity.
Pod Anti-Affinity	Ensures that Pods avoid running on the same Node to improve redundancy.

- ◆ Example Use Cases:
- ✓ Running frontend & backend together on the same Node for low latency.
- ✓ Ensuring database replicas do not run on the same Node to prevent failures.
- ✓ Distributing microservices across multiple Nodes.

Types of Inter-Pod Affinity & Anti-Affinity

Kubernetes supports two levels of affinity rules:

Type	Description	Effect
requiredDuringSchedulingIgnoredDuringExecution	Strict rule: Pod must be scheduled based on the affinity rule.	If no matching Node is found, the Pod stays Pending.
preferredDuringSchedulingIgnoredDuringExecution	Soft rule: Kubernetes tries to place the Pod based on the affinity rule.	If no matching Node is found, the Pod is scheduled elsewhere.

We will:

Label Nodes for scheduling.

Use Pod Affinity to group related Pods together.

Use Pod Anti-Affinity to spread Pods across different Nodes.

Modify rules and test behavior.

```
kubectl label nodes worker-node-1 zone=us-west
```

```
kubectl get nodes --show-labels
```

Pod Affinity (Pods Prefer to be Together)

Ensures that related Pods (e.g., frontend & backend) run on the same Node.

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: frontend-pod
  labels:
    app: frontend
spec:
  containers:
  - name: nginx
    image: nginx
---
apiVersion: v1
kind: Pod
metadata:
  name: backend-pod
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        labelSelector:
          matchLabels:
            app: frontend
        topologyKey: "kubernetes.io/hostname"
  containers:
  - name: nginx
    image: nginx
EOF
```

How It Works
The backend Pod requires the frontend Pod to be on the same Node.
Kubernetes schedules them together if possible.

 Both Pods are scheduled on worker-node-1 as expected!

Pod Anti-Affinity (Pods Should be on Different Nodes)

Ensures that multiple replicas of the same Pod do not run on the same Node.
Helps distribute workloads for high availability.

```
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: database-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: database
  template:
    metadata:
      labels:
        app: database
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchLabels:
                  app: database
              topologyKey: "kubernetes.io/hostname"
  containers:
    - name: postgres
      image: postgres
EOF
```

Expected Outcome

Database Pods will be distributed across different Nodes.
This ensures HA (High Availability) by preventing all replicas from being on the same Node.

 Database replicas are spread across different Nodes for fault tolerance!

✓ Deployments use Affinity & Anti-Affinity for automatic Pod scheduling.
✓ Pod Affinity keeps related Pods together (e.g., frontend & backend).
✓ Pod Anti-Affinity distributes Pods for high availability (e.g., database replicas).
✓ Scaling does not break Affinity/Anti-Affinity rules.
✓ Modifying Affinity preferences can change Pod placement dynamically.

Taints & Tolerations

Taints and Tolerations allow Kubernetes to control which Pods can be scheduled on which Nodes by marking Nodes as "unavailable" to specific workloads unless they have a Toleration.

Feature	Description
Taints (Node Level)	Prevents Pods from being scheduled unless they have a matching Toleration.
Tolerations (Pod Level)	Allows specific Pods to override a Node's Taint and get scheduled.

Taint Effects (Scheduling Behavior)

Taint Effect	Description
NoSchedule	Strict rule: New Pods without a matching Toleration will not be scheduled on the Node.
PreferNoSchedule	Soft rule: Kubernetes will try to avoid scheduling Pods on the Node, but may still do so if necessary.
NoExecute	Strictest rule: Existing Pods will be evicted if they do not have a Toleration.

We will:

Taint a Node to prevent normal workloads from being scheduled.

Deploy a normal Pod and observe its failure.

Deploy a Tolerant Pod and verify it runs successfully.

Modify Taints dynamically and test behavior.

Add a Taint to a Node

To prevent Pods from scheduling on worker-node-1, apply a Taint:

```
kubectl taint nodes worker-node-1 key=value:NoSchedule
```

```
kubectl describe node worker-node-1 | grep Taints
```



Now, worker-node-1 will reject all Pods unless they have a matching Toleration!

Deploy a Normal Pod (Without Toleration)

This Pod does not have a Toleration, so it should fail to schedule on worker-node-1.

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: normal-pod
spec:
  containers:
    - name: nginx
      image: nginx
EOF
```

```
kubectl describe pod normal-pod
```

The Pod is stuck in Pending because of the Node Taint!

Deploy a Pod with a Matching Toleration

Now, let's deploy a Pod that can tolerate the Node Taint.

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: tolerant-pod
spec:
  tolerations:
  - key: "key"
    operator: "Equal"
    value: "value"
    effect: "NoSchedule"
  containers:
  - name: nginx
    image: nginx
EOF
```

 The Tolerant Pod successfully bypasses the Taint and runs on worker-node-1!

Use NoExecute to Evict Running Pods

Now, let's modify the Taint so that existing Pods get evicted if they don't have a Toleration.

```
kubectl taint nodes worker-node-1 key=value:NoExecute --overwrite
```

 The normal Pod gets evicted because it doesn't tolerate NoExecute!

Remove a Taint

To remove the Taint from the Node:

```
kubectl taint nodes worker-node-1 key=value:NoSchedule-
```

 Now, normal Pods can be scheduled on worker-node-1 again!

Command	Description	
<code>kubectl taint nodes <node> key=value:NoSchedule</code>	Prevents new Pods from running on the Node unless they tolerate the Taint.	<ul style="list-style-type: none"> ✓ Taints prevent unwanted Pods from running on a Node.
<code>kubectl taint nodes <node> key=value:PreferNoSchedule</code>	Soft rule: Avoids scheduling Pods here unless necessary.	<ul style="list-style-type: none"> ✓ Tolerations allow specific Pods to override Taints.
<code>kubectl taint nodes <node> key=value:NoExecute</code>	Evicts existing Pods unless they tolerate the Taint.	<ul style="list-style-type: none"> ✓ NoSchedule prevents new Pods from being scheduled. ✓ PreferNoSchedule avoids scheduling Pods unless necessary. ✓ NoExecute immediately evicts existing Pods that do not tolerate the Taint. ✓ Removing a Taint makes the Node available for normal workloads again.
<code>kubectl describe node <node></code>	View Node Taints.	
<code>kubectl taint nodes <node> key=value:NoSchedule-</code>	Removes a Taint from a Node.	

Kubernetes

Storage and Persistence

Kubernetes

Introduction to Volumes

Kubernetes Volumes provide persistent storage for Pods, ensuring that data survives container restarts. Unlike container storage, which is ephemeral, Kubernetes Volumes allow data sharing between containers within a Pod or across multiple Pods.

Feature	Description
Persistent Storage	Ensures data is not lost when a container restarts.
Multiple Volume Types	Supports local storage, cloud storage, and distributed storage.
Pod-Scope Volumes	Volumes are defined at the Pod level, not per container.
Data Sharing	Enables data sharing between multiple containers in the same Pod.
Supports Persistent Volumes	Allows long-term storage beyond Pod lifetimes.

Types of Kubernetes Volumes

Volume Type	Description	Use Case
emptyDir	Temporary storage that is deleted when the Pod is deleted.	Caching, scratch space.
hostPath	Uses a directory on the Node's filesystem.	Logs, debug tools.
configMap/Secret	Stores configuration data as a file.	Application settings, passwords.
Persistent Volume (PV & PVC)	Persistent storage managed separately from the Pod lifecycle.	Databases, logs.
nfs, csi, awsEBS, azureDisk	Cloud-based or external network storage.	Cloud storage solutions.

Introduction to Volumes

Persist and share data in Kubernetes

Different types of Volumes

Volume Type	When to use
<code>emptyDir</code>	Ephemeral storage within the Pod. Follows the lifecycle of the Pod: when the Pod is terminated, its contents are also deleted.
<code>local*</code>	Durable storage within a specific node. Requires setting node affinity to correctly schedule Pods on the correct nodes. Preferred over <code>hostPath</code>
<code>Persistent Volume</code>	Durable storage backed by multiple technologies (e.g., cloud storage). Can be statically or dynamically provisioned.
<code>ConfigMap</code>	Inject configuration data into Pods, without having to hard-code the data into the Pod definition.
<code>Secret</code>	Inject sensitive data into Pods, without having to hard-code the data into the Pod definition.

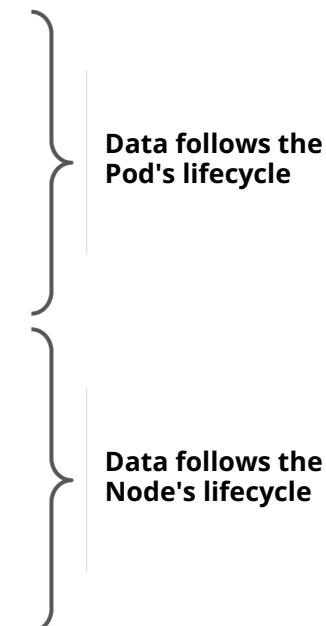
* `hostPath` may introduce many security vulnerabilities and is currently discouraged.

Kubernetes

Kubernetes **EmptyDir and Local**

EmptyDir and Local

Pod- and Node-level Storage

- `emptyDir` volumes are **ephemeral** and defined at the Pod level.
 - The volume is created when the Pod is assigned to a node.
 - The volume is initially empty.
 - All containers in the Pod can read and write the same files in the `emptyDir` volume.
 - Containers might mount the volume in different paths.
 - When a Pod is removed from a node, the data in the `emptyDir` is deleted permanently.
 - `local` volumes are **persistent** and defined at the Node level.
 - kube-scheduler knows how to assign Pods to Nodes based on the affinity constraints defined in the `PersistentVolume` configuration.
 - Setting a `PersistentVolume nodeAffinity` is mandatory when using local volumes.
 - Like other `PersistentVolumes`, it requires creating `PersistentVolumeClaims` so that Pods can use the storage.
 - Only supports static provisioning.
- 

Use emptyDir for Data Sharing Between Containers

emptyDir is a temporary directory shared between containers in a Pod.

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: shared-volume-pod
spec:
  volumes:
    - name: shared-data
      emptyDir: {}
  containers:
    - name: writer
      image: busybox
      command: ["sh", "-c", "echo 'Hello from Writer!' > /data/message && sleep 3600"]
      volumeMounts:
        - name: shared-data
          mountPath: /data
    - name: reader
      image: busybox
      command: ["sh", "-c", "cat /data/message && sleep 3600"]
      volumeMounts:
        - name: shared-data
          mountPath: /data
EOF
kubectl logs shared-volume-pod -c reader
```

Use hostPath to Store Logs on a Node

hostPath mounts a Node directory inside the Pod.

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
  name: hostpath-volume-pod
spec:
  volumes:
    - name: host-logs
      hostPath:
        path: /var/log/nginx
        type: DirectoryOrCreate
  containers:
    - name: nginx
      image: nginx
      volumeMounts:
        - name: host-logs
          mountPath: /var/log/nginx
EOF
```

kubectl exec -it hostpath-volume-pod -- ls /var/log/nginx

 Now, logs persist on the Node's filesystem even if the Pod is deleted!

Kubernetes

Persistent Volume Claims

Kubernetes StorageClass, PV, and PVC enable dynamic and persistent storage management. These components allow Kubernetes to provision storage dynamically, manage data persistence, and integrate with cloud-based storage solutions.

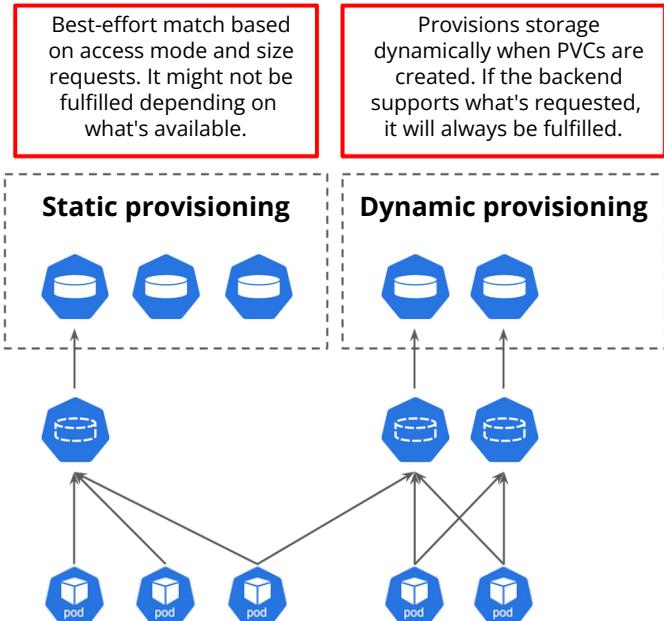
Concept	Description
StorageClass (SC)	Defines different storage types (e.g., SSD, HDD, Cloud Disks).
PersistentVolume (PV)	A physical storage unit available for use by Kubernetes.
PersistentVolumeClaim (PVC)	A request for storage by a Pod, which gets bound to a PV.

- ◆ Example Use Cases:
 - ✓ Running databases (MySQL, PostgreSQL) with persistent storage.
 - ✓ Storing logs, backups, and files beyond Pod lifetimes.
 - ✓ Using cloud-based block storage (AWS EBS, Azure Disk, GCP Persistent Disks).

Persistent Volume Claims

Claim durable storage for usage in Pods

- A `PersistentVolumeClaim` is what actually reserves a PV (when using static provisioning) or creates and reserves a PV (when using dynamic provisioning).
- Claims are bound to a single `PersistentVolume`, and a `PersistentVolume` can have **at most one** claim bound to it.
 - We can set specific criteria in a claim, so that only PVs that match these criteria are considered for binds.
 - Be mindful of the possibility for extra unused capacity!
 - Reclamation policies can be `Retain`, `Delete`, or `Recycle` (deprecated).
- Access modes:
 - `ReadWriteOnce`: the volume can be mounted as read-write by a single node, and can be used by any number of Pods within that node.
 - `ReadOnlyMany`: the volume can be mounted as read-only by many nodes.
 - `ReadWriteMany`: the volume can be mounted as read-write by many nodes.
 - `ReadWriteOncePod`: the volume can be mounted as read-write by a single Pod.



How PV, PVC, and StorageClass Work Together

StorageClass defines how storage is provisioned dynamically.

PersistentVolume (PV) provides actual storage space.

PersistentVolumeClaim (PVC) allows Pods to request storage.

Reclaim Policy	Effect
Retain	PV is not deleted after PVC removal (manual cleanup required).
Recycle	PV is wiped and reused.
Delete	PV is deleted automatically with PVC.

- ✓ StorageClass enables dynamic storage provisioning.
- ✓ PV is the actual storage resource, PVC is the request for storage.
- ✓ PVC binds to a PV based on storageClassName.
- ✓ Data persists across Pod restarts when using PVs/PVCs.
- ✓ Reclaim Policies (Retain, Recycle, Delete) define PV behavior after PVC deletion.

Kubernetes Deployment with PVC for Dynamic Storage Provisioning

```
kubectl get storageclass
```

```
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dynamic-storage-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      volumes:
        - name: storage-volume
          persistentVolumeClaim:
            claimName: dynamic-pvc
      containers:
        - name: nginx
          image: nginx
          volumeMounts:
            - name: storage-volume
              mountPath: /usr/share/nginx/html
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: dynamic-pvc
spec:
  storageClassName: standard # This triggers dynamic provisioning
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 500Mi
EOF
```

 Kubernetes will automatically create a PV and bind it to the PVC without manual intervention!

 A new PV was automatically created and bound to the PVC!

kubectl get pvc
kubectl get pv

Steps to Dynamically Provision Storage on Cloud

Check available StorageClasses in the cloud Kubernetes cluster.

Use the cloud provider's default StorageClass for automatic provisioning.

Deploy an application that dynamically provisions storage via Persistent Volume Claims (PVCs).

Verify that the cloud storage was automatically created and attached.

Step 1: Check Available StorageClasses in the Cloud

Before defining your PVC, check the available StorageClasses in your cloud environment.

```
kubectl get storageclass
```

Example Output for AWS, Azure, and GCP:

NAME	PROVISIONER	RECLAIMPOLICY	VOLUMEBINDINGMODE
gp2 (default)	kubernetes.io/aws-ebs	Delete	WaitForFirstConsumer
standard (default)	kubernetes.io/gce-pd	Delete	WaitForFirstConsumer
default (default)	kubernetes.io/azure-disk	Delete	WaitForFirstConsumer



The default StorageClass (gp2, standard, default) will automatically create Persistent Volumes (PVs).

Step 2: Deploy a Dynamic Storage Provisioning Application

💡 This Deployment dynamically provisions storage via PVC, using the default StorageClass of the cloud provider.

```
cat <<EOF | kubectl apply -f -  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: cloud-storage-deployment  
spec:  
  replicas: 2  
  selector:  
    matchLabels:  
      app: nginx  
  template:  
    metadata:  
      labels:  
        app: nginx  
    spec:  
      volumes:  
      - name: storage-volume  
        persistentVolumeClaim:  
          claimName: cloud-dynamic-pvc  
      containers:  
      - name: nginx  
        image: nginx  
        volumeMounts:  
        - name: storage-volume  
          mountPath: /usr/share/nginx/html  
---  
apiVersion: v1  
kind: PersistentVolumeClaim  
metadata:  
  name: cloud-dynamic-pvc  
spec:  
  storageClassName: gp2 # AWS (use "standard" for GCP or "default" for Azure)  
  accessModes:  
  - ReadWriteOnce  
resources:  
  requests:  
    storage: 5Gi  
EOF
```

📌 This YAML will:

Create a PVC (cloud-dynamic-pvc) using the default cloud StorageClass.
Dynamically provision a Persistent Volume (PV) in the cloud (EBS, Azure Disk, or GCE PD).
Deploy Nginx Pods, automatically mounting the dynamically provisioned storage.

Step 3: Verify Storage Provisioning

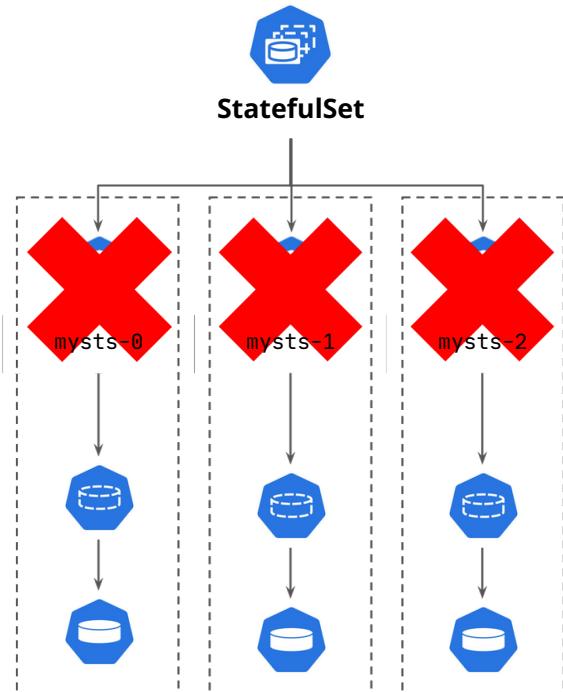
✓ Check if the PVC is successfully created and bound:

Kubernetes **StatefulSets**

StatefulSets

Manage stateful applications more effectively

- StatefulSets maintain stable identities for each pod, ensuring that even if a pod is restarted, it retains its unique identity and connection to persistent storage. More specifically, StatefulSets provide:
 - A stable, unique network identity for each pod created as part of the StatefulSet.
 - Consistent and stable persistent storage associated with each pod, also across restarts.
 - Ordered pod creation, scaling, and deletion, should that be necessary when working with databases or replicated services.
- We can specify a PersistentVolumeClaim template in the Pod definition, and as long as this stays the same, the Pod will use the same claim and have access to the same data.
 - Each replica in a StatefulSet will have its own PersistentVolumeClaim, so data is not shared across different replicas.
 - The PersistentVolumes are not deleted automatically when a replica is deleted!
- Pod names and networking identities are also stable, and will follow the following pattern: <statefulset name>-<ordinal id>
 - We can use headless services to expose the pods via more stable domains.



A StatefulSet in Kubernetes is used to manage stateful applications that require persistent storage, ordered deployment, and stable network identities. Unlike Deployments, StatefulSets ensure that Pods are created and deleted sequentially while maintaining a unique identity.

Feature	Description
Stable Pod Identity	Each Pod gets a unique, stable network identity (pod-0, pod-1).
Ordered Deployment & Scaling	Pods are created one at a time and scaled sequentially.
Persistent Storage with PVCs	Each Pod retains its own dedicated storage via Persistent Volume Claims (PVCs).
Ordered Updates & Deletion	Pods are updated and deleted in a controlled sequence.
Common Use Cases	- Databases (MySQL, PostgreSQL, MongoDB) - Distributed systems (Kafka, Zookeeper) - Stateful applications (Redis, Cassandra)

When to Use StatefulSets vs Deployments?

Feature	StatefulSet	Deployment
Stable Network Identity	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No
Persistent Storage (Unique PVCs)	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No
Ordered Deployment & Scaling	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No
Use Case	Databases, Zookeeper, Kafka	Stateless apps, Web servers

We will:

- Create a StorageClass for Minikube.
- Deploy a StatefulSet for a database (MySQL).
- Verify StatefulSet behavior (Pods, PVCs, Scaling).
- Test data persistence after Pod deletion.

Create a StorageClass

```
cat <<EOF | kubectl apply -f -
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: minikube-storage
provisioner: k8s.io/minikube-hostpath
volumeBindingMode: Immediate
EOF
```

```
kubectl get storageclass
```

Deploy MySQL StatefulSet

Now, we deploy a MySQL StatefulSet with Persistent Volumes.

Verify StatefulSet Deployment

1. Check StatefulSet Pods

```
kubectl get pods -o wide
```

```
kubectl get statefulset
```

```
kubectl get pvc
```

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
  name: mysql-service
spec:
  selector:
    app: mysql
  clusterIP: None
  ports:
    - name: mysql
      port: 3306
      targetPort: 3306
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  serviceName: "mysql-service"
  replicas: 2
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
        - name: mysql
          image: mysql:5.7
          env:
            - name: MYSQL_ROOT_PASSWORD
              value: "rootpassword"
          ports:
            - containerPort: 3306
          volumeMounts:
            - name: mysql-storage
              mountPath: /var/lib/mysql
  volumeClaimTemplates:
  - metadata:
      name: mysql-storage
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 1Gi
      storageClassName: minikube-storage
EOF
```

Test Data Persistence

1. Connect to MySQL Pod

```
kubectl exec -it mysql-0 -- mysql -u root -p  
CREATE DATABASE testdb; SHOW DATABASES;
```

Scale the StatefulSet

```
kubectl scale statefulset mysql --replicas=3
```

Delete a Pod and Verify Persistence

1. Delete mysql-0

```
kubectl delete pod mysql-0
```

 Pod mysql-0 has restarted with the same identity!

Verify Data Persistence

Reconnect to MySQL:

```
kubectl exec -it mysql-0 -- mysql -u root -p  
SHOW DATABASES;
```

Kubernetes

Role-Based Access Control

Kubernetes RBAC (Role-Based Access Control) is a security mechanism that restricts access to resources based on roles and permissions. It ensures that only authorized users, services, and applications can perform specific actions.

Component	Description
Role	Defines permissions (e.g., read Pods, update Deployments) within a specific namespace.
ClusterRole	Similar to a Role, but applies cluster-wide (e.g., managing Nodes, PersistentVolumes).
RoleBinding	Binds a Role to a specific user, group, or service account.
ClusterRoleBinding	Binds a ClusterRole to users, groups, or service accounts at the cluster level.
ServiceAccount	Provides an identity for Pods to interact with the API securely.

Creating a new user (jai) in Minikube.
Extracting the Kubernetes CA (ca.crt and ca.key).
Generating an SSL certificate for authentication.
Configuring Kubeconfig for the user.
Setting up RBAC permissions (Role & RoleBinding).
Verifying access with kubectl auth can-i.

SSH into Minikube and Extract CA Files

Run the following command to copy the Kubernetes CA certificate and key:

```
minikube ssh  
sudo cp /var/lib/minikube/certs/ca.crt /home/docker/  
sudo cp /var/lib/minikube/certs/ca.key /home/docker/  
sudo chmod 644 /home/docker/ca.crt /home/docker/ca.key  
exit
```

```
scp -i $(minikube ssh-key) docker@$(minikube ip):/home/docker/ca.crt .  
scp -i $(minikube ssh-key) docker@$(minikube ip):/home/docker/ca.key
```

Generate SSL Certificate for User jai

On your local machine, generate a private key for jai:

```
openssl genrsa -out jai.key 2048
```

```
openssl req -new -key jai.key -out jai.csr -subj "/CN=jai/O=developers"
```

```
openssl x509 -req -in jai.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out jai.crt -days 365
```

 Now, jai.crt and jai.key are ready for authentication!

Configure Kubeconfig for User jai

Now, we will add jai as a Kubernetes user in Kubeconfig.

Step 1: Add the User to Kubeconfig

```
kubectl config set-credentials jai --client-certificate=jai.crt --client-key=jai.key
```

Step 2: Create a Context for the User

```
kubectl config set-context jai-context --cluster=kubernetes --user=jai --namespace=default
```

```
kubectl config use-context jai-context
```

Step 3: Verify Authentication

```
kubectl auth can-i list pods --as=jai
```

✖ This is expected because jai does not have any permissions yet!

Now, let's create an RBAC Role for jai.

Configure RBAC Role for User jai

Now, we define RBAC permissions so jai can access Kubernetes resources.

Step 1: Create a Role That Allows Listing Pods

```
cat <<EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: []
  resources: ["pods"]
  verbs: ["get", "list"]
EOF
```

🚀 This Role allows jai to list Pods but not modify them.

Step 2: Bind the Role to jai

```
cat <<EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-reader-binding
  namespace: default
subjects:
- kind: User
  name: jai
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
EOF
```

🚀 Now, jai has access to list Pods in the default namespace!

Step 3: Verify Permissions

```
kubectl auth can-i list pods --as=jai
kubectl auth can-i delete pods --as=jai
```

🚀 RBAC is working correctly! jai can only
list Pods but cannot delete them.

4. Grant Additional Permissions (Optional)

If you want jai to manage Deployments (create, update, delete), modify the Role:

```
cat <<EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: deployment-manager
rules:
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["get", "list", "create", "update", "delete"]
EOF
```

Then, bind the Role to jai:

```
cat <<EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: deployment-manager-binding
  namespace: default
subjects:
- kind: User
  name: jai
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: deployment-manager
  apiGroup: rbac.authorization.k8s.io
EOF
```

🚀 Now, jai can fully manage Deployments but still cannot modify Pods!

5. Assign Cluster-Wide Permissions (ClusterRole)

If you want jai to list Pods across all namespaces, use a `ClusterRole` instead of a `Role`.

```
cat <<EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-pod-reader
rules:
- apiGroups: []
  resources: ["pods"]
  verbs: ["get", "list"]
EOF
```

Bind the `ClusterRole` to jai:

🚀 Now, jai can list Pods
across the entire cluster!

```
cat <<EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-pod-reader-binding
subjects:
- kind: User
  name: jai
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cluster-pod-reader
  apiGroup: rbac.authorization.k8s.io
EOF
```

6. Final Verification

```
kubectl auth can-i list pods --as=jai  
kubectl auth can-i delete pods --as=jai
```

- ✓ Kubeadm-based clusters store CA files in /etc/kubernetes/pki/.
- ✓ User jai was created with a client certificate and added to kubeconfig.
- ✓ RBAC Role & RoleBinding were created to grant limited permissions.
- ✓ ClusterRole allows cluster-wide access if needed.
- ✓ Use kubectl auth can-i to verify permissions.

For kubeadm

Extract the Kubernetes CA Certificate and Key

```
ls -l /etc/kubernetes/pki/
```

```
cp /etc/kubernetes/pki/ca.crt /root/      Copy them to a location accessible for downloading:  
cp /etc/kubernetes/pki/ca.key /root/
```

```
scp user@<master-node-ip>:/root/ca.crt .      Now, copy the CA files to your local machine:  
scp user@<master-node-ip>:/root/ca.key .
```

Continue from above minikube setup
from Generate SSL Certificate for User jai

Role-Based Access Control

Define permissions for users, groups, and service accounts

- **Role-Based Access Control (RBAC)** is a security model in Kubernetes that defines and enforces permissions for users, service accounts, and groups to interact with cluster resources.
- Permissions can target many different resources (pods, deployments, secrets, configmaps, and many others) and operations (read, create, update, delete, as well as specific operations such as exec) in the cluster, allowing for very fine-grained access control.
- **RBAC is a must** for any serious Kubernetes cluster.
- Five key components of **RBAC**:
 - **Users, groups, and service accounts:** the entities against which RBAC policies will be enforced
 - **Roles:** defines a set of permissions within a specific namespace. It specifies the actions (verbs), such as `get`, `list`, `create`, that can be performed on particular Kubernetes resources (like pods, services, secrets).
 - **RoleBindings:** assigns a `Role` to a user, group, or service account within a namespace.
 - **ClusterRoles:** similar to `Roles`, but has effect across the entire cluster. They can be used to grant permissions to resources that are not namespaced, or to resources across all namespaces.
 - **ClusterRoleBindings:** assigns a `ClusterRole` to a user, group, or service account.

Why Do We Need CA Certificates (ca.crt and ca.key) from the Master Node?

1Purpose of the Kubernetes CA (Certificate Authority)

In Kubernetes, the Control Plane (Master Node) is responsible for authenticating and securing communications between different components. The Kubernetes API server requires all users and service accounts to be authenticated before they can interact with the cluster.

The Kubernetes Certificate Authority (CA) is used to sign and validate certificates for:

API server communication

Worker nodes

Service accounts

Custom users (like jai)

File	Purpose
ca.crt	The public certificate of the Kubernetes Certificate Authority (CA). Used to verify that a client certificate is signed by Kubernetes.
ca.key	The private key of the Kubernetes CA. Used to sign new certificates for users and components.

Differences Between Cloud-Based and Self-Managed Kubernetes Authentication

Feature	Kubeadm (Self-Managed)	Cloud Kubernetes (EKS, AKS, GKE)
User Authentication	Uses TLS certificates (manual setup).	Uses IAM (Identity and Access Management) or OIDC (OpenID Connect).
RBAC Setup	Uses ca.crt and ca.key to sign certificates.	Uses IAM roles, Azure AD, Google IAM (no access to ca.key).
Managing Users	Manually create user certificates.	Users are authenticated via cloud IAM services.
Kubeconfig Setup	Manual certificate-based authentication.	<code>aws eks update-kubeconfig</code> , <code>gcloud container clusters get-credentials</code> , or <code>az aks get-credentials</code> .

Kubernetes Ingress

Ingress in Kubernetes is a resource that manages external access to services inside the cluster. It provides HTTP/HTTPS routing, load balancing, and TLS termination.

Method	Description	Use Case
ClusterIP	Internal access only (no external exposure).	Services communicating inside the cluster.
NodePort	Exposes a Service on a static port on all Nodes.	Direct access, no load balancing.
LoadBalancer	Uses a cloud provider's load balancer.	Expensive, best for cloud environments.
Ingress	Routes multiple services via a single entry point.	Best for web applications and APIs.

 Ingress is the preferred way to expose multiple services using a single Load Balancer!

How Ingress Works

A user requests example.com/api.

Ingress Controller receives the request.

Ingress Rules define which Service should handle the request.

Service forwards traffic to the correct Pod.

User → Ingress Controller → Ingress Rules → Service → Pods

Install Ingress Controller in Minikube

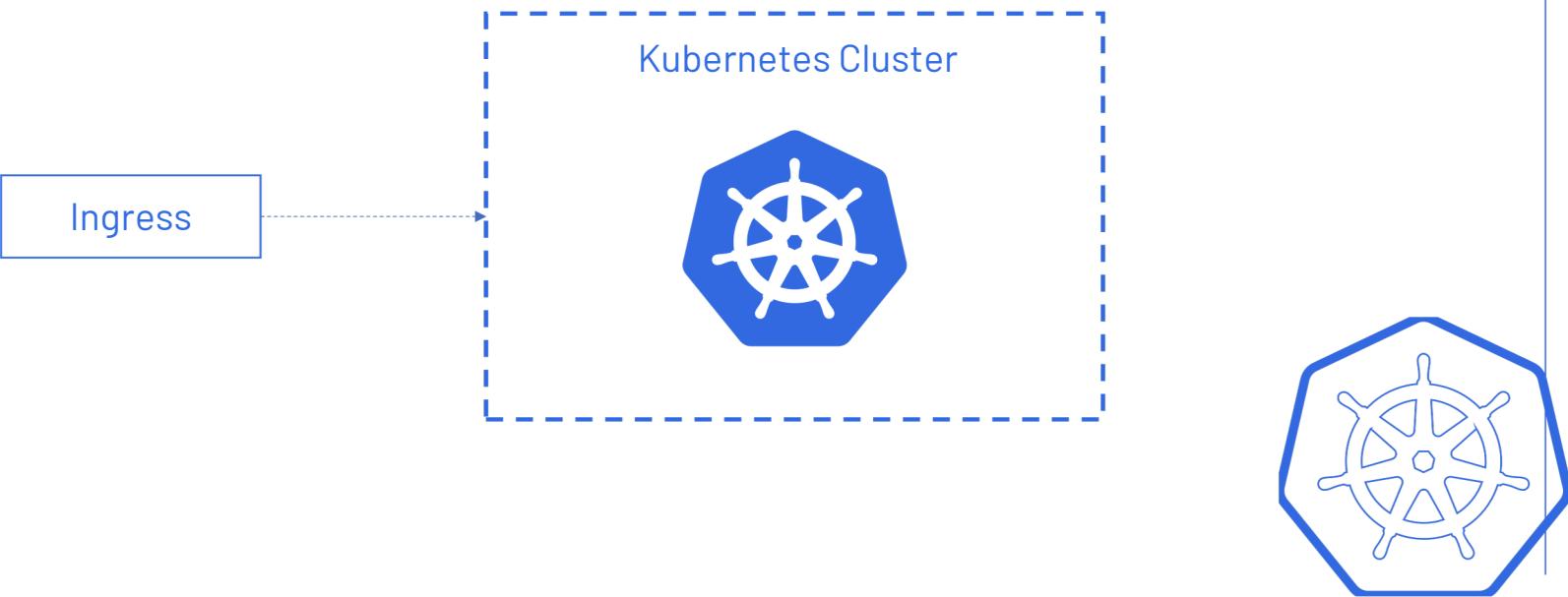
Since Minikube does not include an Ingress Controller by default, we must enable it.

Step 1: Enable Ingress in Minikube

minikube addons enable ingress

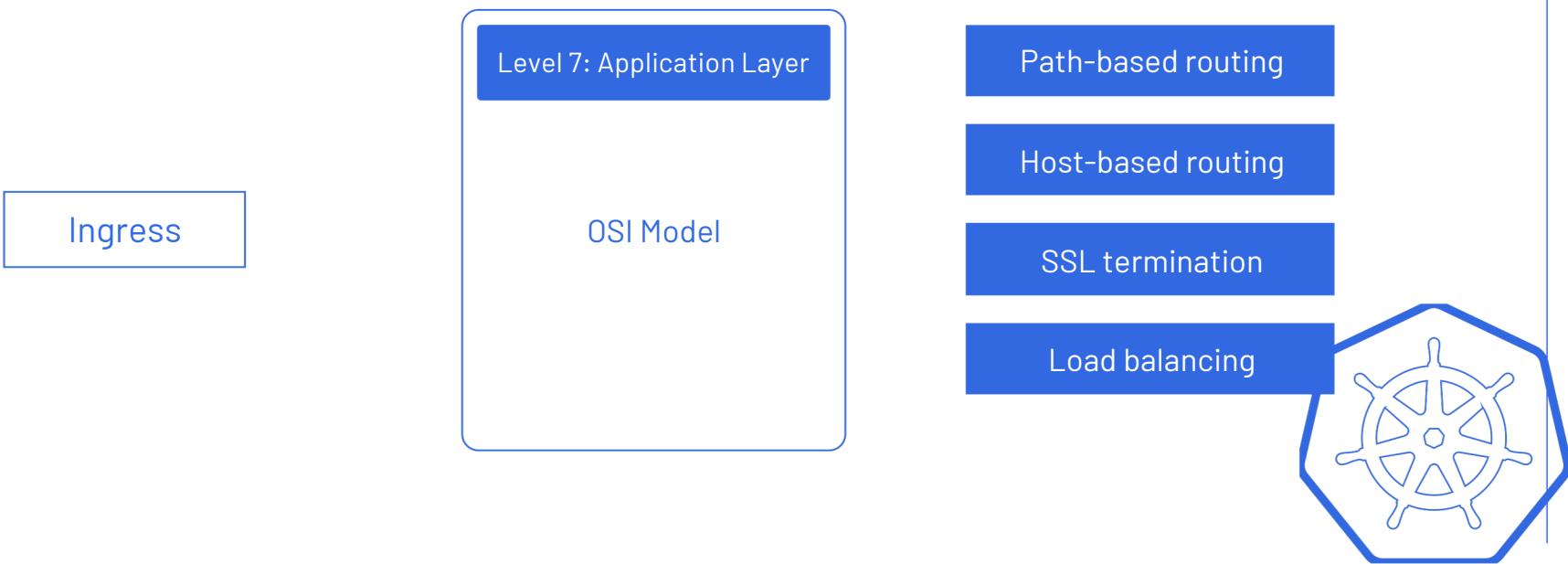
Ingress

- Ingress is a Kubernetes resource that manages external access to services within a cluster
- Acts as a router, defining rules to control how external users can access services running inside the cluster



Ingress

- Ingress defines rules for routing HTTP and HTTPS traffic to services within the cluster



Ingress

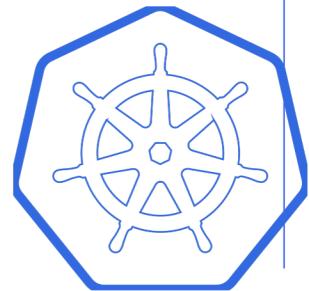
Components of Ingress

Defines routing rules to direct incoming requests to backend services

Ingress Resource

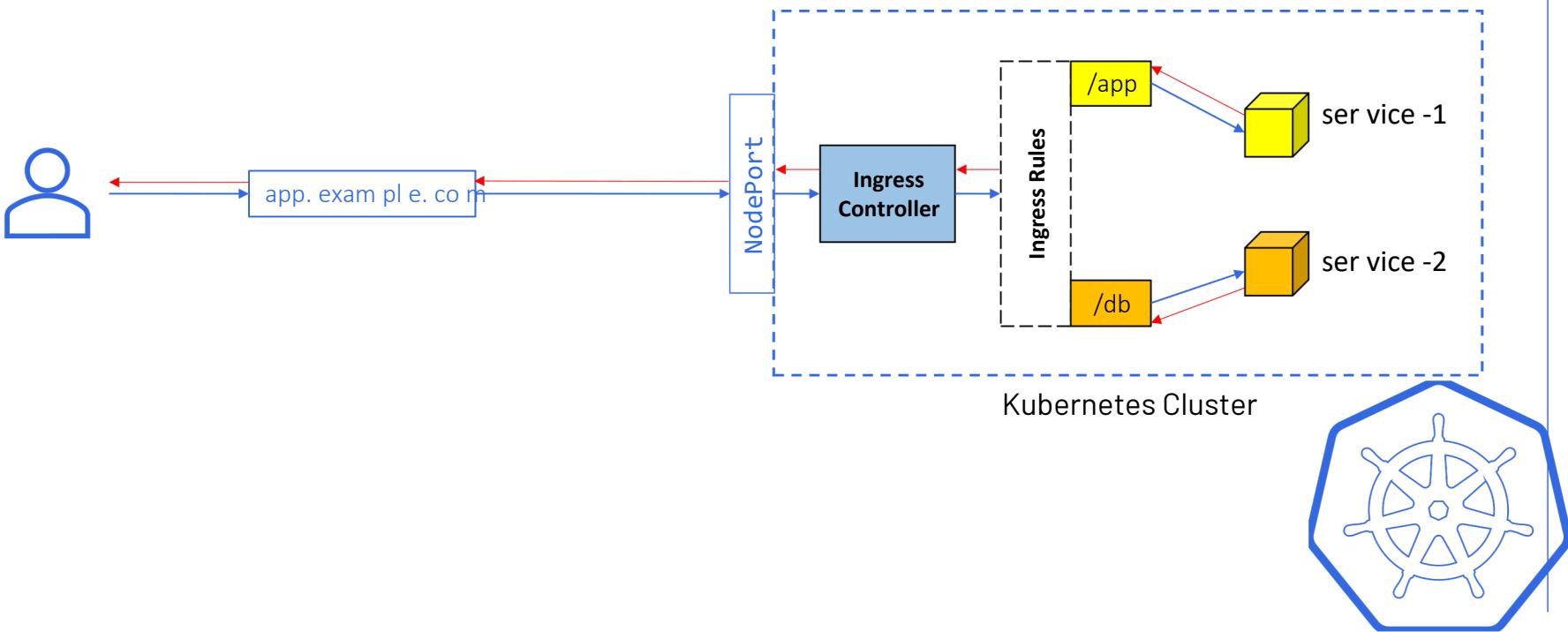
Ingress Controller

Enforces Ingress rules, but Kubernetes lacks a built-in Ingress controller



Ingress

Flow of Traffic in Ingress



Verify That the Ingress Controller is Running

```
kubectl get pods -n kube-system | grep ingress
```

 Now, the Ingress Controller is ready to handle requests!

Deploy an Example Application

We will:

Deploy two services (app1 and app2).

Create an Ingress to route traffic to them.

Step 1: Deploy Two Sample Applications

```
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app1
spec:
  replicas: 2
  selector:
    matchLabels:
      app: app1
  template:
    metadata:
      labels:
        app: app1
  spec:
    containers:
      - name: app1
        image: hashicorp/http-echo
        args:
          - "-text=Hello from App 1"
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: app1-service
spec:
  selector:
    app: app1
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5678
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app2
spec:
  replicas: 2
  selector:
    matchLabels:
      app: app2
  template:
    metadata:
      labels:
        app: app2
  spec:
    containers:
      - name: app2
        image: hashicorp/http-echo
        args:
          - "-text=Hello from App 2"
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: app2-service
spec:
  selector:
    app: app2
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5678
EOF
```

🚀 Now, we have two applications (app1 and app2) running inside Minikube!

```
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app1
spec:
  replicas: 2
  selector:
    matchLabels:
      app: app1
  template:
    metadata:
      labels:
        app: app1
    spec:
      containers:
        - name: app1
          image: hashicorp/http-echo
          args:
            - "-text=Hello from App 1"
---
apiVersion: v1
kind: Service
metadata:
  name: app1-service
spec:
  selector:
    app: app1
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5678
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app2
spec:
  replicas: 2
  selector:
    matchLabels:
      app: app2
  template:
    metadata:
      labels:
        app: app2
    spec:
      containers:
        - name: app2
          image: hashicorp/http-echo
          args:
            - "-text=Hello from App 2"
---
apiVersion: v1
kind: Service
metadata:
  name: app2-service
spec:
  selector:
    app: app2
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5678
EOF
```

5. Create an Ingress Resource

Now, we define an Ingress to route traffic to the correct Service.

```
cat <<EOF | kubectl apply -f -
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
spec:
  rules:
    - host: app1.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: app1-service
                port:
                  number: 80
    - host: app2.local
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: app2-service
                port:
                  number: 80
EOF
```



Now, requests to `app1.local` go to `app1-service`, and `app2.local` goes to `app2-service`.

6. Test Ingress in Minikube

Step 1: Get Minikube's IP

```
minikube ip
```

```
Add Host Entries      192.168.49.2      app1.local
Modify your /etc/hosts 192.168.49.2      app2.local
```

```
curl http://app1.local  Hello from App 1
```

```
curl http://app2.local  Hello from App 2
```

 Success! Requests are correctly routed to the respective applications.

Network Policies

Network Policies in Kubernetes control how Pods communicate with each other and with external networks. They allow you to restrict access between Pods based on labels, namespaces, and IP ranges.

Feature	Description
Pod-Level Security	Restricts communication between Pods.
Namespace Isolation	Prevents Pods from different namespaces from accessing each other.
Restrict External Traffic	Blocks unwanted external access to Pods.
Zero Trust Networking	By default, Pods can communicate freely; Network Policies enforce security rules.

 Without Network Policies, all Pods in a cluster can talk to each other freely!

2. How Network Policies Work

A NetworkPolicy defines rules for inbound and outbound traffic.

Default behavior → All Pods can communicate.

Apply NetworkPolicy → Restricts communication only to allowed Pods.

Network Policy Components

Component	Description
Pod Selector	Defines which Pods the policy applies to.
Ingress Rules	Controls incoming traffic to Pods.
Egress Rules	Controls outgoing traffic from Pods.
Namespace Selector	Defines which namespaces are allowed to communicate.
IP Block	Allows or denies traffic based on IP ranges.

We will:

Deploy two applications (frontend and backend).

Apply a Network Policy to restrict communication.

Step 1: Deploy Frontend and Backend Pods



Now, frontend and backend are deployed!

```
cat <<EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
        - name: nginx
          image: nginx
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
        - name: nginx
          image: nginx
---
apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
  selector:
    app: backend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
EOF
```

Step 2: Verify Default Connectivity

By default, Pods can communicate freely. Let's test this.

```
kubectl get pods -l app=frontend
```

```
kubectl exec -it <frontend-pod-name> -- curl backend-service
```

 The frontend can communicate with the backend! Now, let's restrict it.

4. Apply a Network Policy

Restrict Access to the Backend

Now, we only allow traffic from frontend Pods to backend Pods.

```
cat <<EOF | kubectl apply -f -  
apiVersion: networking.k8s.io/v1  
kind: NetworkPolicy  
metadata:  
  name: allow-frontend-to-backend  
  namespace: default  
spec:  
  podSelector:  
    matchLabels:  
      app: backend  
  policyTypes:  
  - Ingress  
  ingress:  
  - from:  
    - podSelector:  
        matchLabels:  
          app: frontend  
  ports:  
  - protocol: TCP  
    port: 80  
EOF
```

podSelector targets backend Pods (only Pods labeled with app=backend).

This means that this policy only applies to backend Pods.

🚀 Without a PodSelector, the policy would apply to all Pods in the namespace!

Defines that this policy only affects incoming traffic (Ingress).

No Egress is defined, meaning backend Pods can still send traffic anywhere.

🚀 Now, only frontend can access backend! All other traffic is blocked.

Test Network Policy

1. Verify That Frontend Can Still Access Backend

```
kubectl exec -it <frontend-pod-name> -- curl backend-service
```

✓ Frontend can still access the backend.

Try Accessing Backend From Another Pod

```
kubectl run test-pod --image=busybox --command -- sleep 3600
```

```
kubectl exec -it test-pod -- curl backend-service
```

 Access is blocked for unauthorized Pods!

Restrict Egress (Outgoing Traffic)

By default, Pods can send traffic anywhere. Let's restrict backend Pods so they cannot access the internet.

```
cat <<EOF | kubectl apply -f -
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: restrict-egress
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
  - Egress
  egress:
  - to:
    - podSelector:
        matchLabels:
          app: frontend
  ports:
  - protocol: TCP
    port: 80
EOF
```

 Now, backend Pods can only communicate with frontend, not external networks.

`kubectl exec -it <backend-pod-name> -- curl google.com`
 Egress to external networks is blocked!

Advanced Network Policies

Scenario	Example Network Policy
Allow traffic only from a namespace	namespaceSelector
Allow specific IP ranges	ipBlock
Restrict egress to specific domains	egress rules
Allow only internal cluster communication	Block all external traffic

Helm: Kubernetes Package Manager - Detailed Guide & Demonstration

Helm is the package manager for Kubernetes. It helps deploy, manage, and update applications using Helm Charts, which are pre-configured Kubernetes resource templates.

1. Why Use Helm?

Feature	Description
Simplifies Deployment	Deploys complex applications with a single command.
Reusability	Helm Charts can be reused across environments (dev, staging, prod).
Versioning & Rollback	Easily update and revert to previous versions.
Dependency Management	Automatically installs dependencies (e.g., databases, services).
Configuration Management	Allows overriding default values with values.yaml.

2. How Helm Works

Helm CLI interacts with the Helm Chart Repository.

Helm Charts are fetched and installed in Kubernetes.

Helm Templates generate Kubernetes manifests dynamically.

Helm Release is created, which can be managed and updated.

User → Helm CLI → Chart Repo → Kubernetes Cluster → Helm Release

3. Install Helm

```
curl -fsSL https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
```

```
helm version
```

4. Helm Chart Structure

A Helm Chart is a packaged Kubernetes application. It contains:

```
mychart/
├── charts/          # Dependencies (sub-charts)
├── templates/       # Kubernetes YAML templates
├── values.yaml      # Default configuration values
├── Chart.yaml        # Metadata about the Helm Chart
└── README.md        # Documentation
```

We will:

Deploy Nginx using Helm.

Modify the configuration using values.yaml.

Upgrade the Helm release.

Step 1: Add a Helm Chart Repository

```
helm repo add bitnami https://charts.bitnami.com/bitnami
```

```
helm repo update
```

Step 2: Install an Nginx Helm Chart

```
helm install mynginx bitnami/nginx
```

 Helm deploys Nginx as a Kubernetes application!

```
kubectl get all | grep mynginx
```

6. Customize Deployment with values.yaml

Instead of using default values, override configurations using values.yaml.

Step 1: Get Default Helm Values

```
helm show values bitnami/nginx > custom-values.yaml
```

Step 2: Edit custom-values.yaml

```
replicaCount: 3
service:
  type: LoadBalancer
```

Step 3: Upgrade the Helm Release

```
helm upgrade mynginx bitnami/nginx -f custom-values.yaml
kubectl get all | grep mynginx
```

 Now, Nginx runs with 3 replicas and a LoadBalancer Service!

7. Rollback to a Previous Version

```
helm history mynginx
```

```
helm rollback mynginx 1
```

 Helm makes it easy to revert changes!

8. Uninstall a Helm Release

```
helm uninstall mynginx
```

```
kubectl get all | grep mynginx
```

 The application is fully removed!

9. Create a Custom Helm Chart

We will create a Helm Chart for a simple web app.

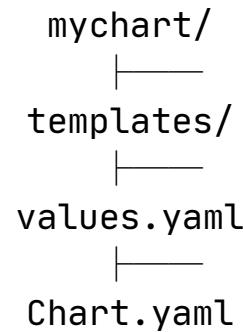
Step 1: Create a Helm Chart

```
helm create mychart  
cd mychart
```

This generates:

Step 2: Modify the Deployment Template

Edit templates/deployment.yaml:



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      app: {{ .Release.Name }}
  template:
    metadata:
      labels:
        app: {{ .Release.Name }}
  spec:
    containers:
      - name: myapp
        image: {{ .Values.image.repository }}:{{ .Values.image.tag }}
        ports:
          - containerPort: 80
```

Step 3: Modify values.yaml

```
replicaCount: 2
image:
  repository: nginx
  tag: latest
```

Step 4: Deploy the Custom Chart

```
helm install myapp ./mychart
```

```
kubectl get deployments
```

 Now, your custom Helm Chart is running!

Command	Description
<code>helm repo add <name> <url></code>	Add a new Helm Chart repository.
<code>helm repo update</code>	Update Helm repositories.
<code>helm search repo <name></code>	Search for charts in repositories.
<code>helm install <release-name> <chart-name></code>	Install a Helm Chart.
<code>helm upgrade <release-name> <chart-name> -f values.yaml</code>	Upgrade an existing release.
<code>helm rollback <release-name> <revision></code>	Rollback to a previous Helm release.
<code>helm uninstall <release-name></code>	Uninstall a Helm release.
<code>helm list</code>	List installed Helm releases.
<code>helm history <release-name></code>	View release history.

THE
END