####################################
Docker & Kubernetes Overview
####################################

What is Docker?

Docker is a free and open-source containerization software that allows you to package applications along with their dependencies into a single unit called a Docker Image. This image can run on any system that has Docker installed, making deployment easy and consistent across different environments.

Why use Docker?

✅ Portability – Run the same application on any machine, regardless of OS configuration.
✅ Dependency Management – Ensures that all required software (e.g., libraries, databases, and runtimes) is included within the image.
✅ Fast Deployment – No need to manually install dependencies every time you set up a new environment.
✅ Resource Efficiency – Uses fewer resources compared to traditional virtual machines.

#######################
How Docker Works?
#######################
Create a Docker Image – Package the app code + dependencies into a lightweight container image.

Run the Docker Container – Deploy this image as a container using the docker run command.

Execute Anywhere – The same image runs on any machine with Docker installed.

Once the image is built, it can run on any machine without requiring additional software setup.

##
Kubernetes (Container Orchestration Software)
##

What is Kubernetes?

Kubernetes (K8s) is a free and open-source orchestration tool developed by Google to manage containerized applications.

💡 Orchestration = Managing multiple containers efficiently

Kubernetes automates key tasks like:

1. Creating, starting, and stopping containers.

2. Scaling up/down based on demand.

3. Handling failures automatically.

#######################
Why use Kubernetes?
#######################

✅ Orchestration – Efficiently manages multiple containers across a cluster of machines.
✅ Self-Healing – If a container crashes, Kubernetes automatically replaces it.
✅ Load Balancing – Distributes traffic across multiple containers to avoid overloading.
✅ Auto Scaling – Increases or decreases the number of running containers based on traffic load.
✅ Automated Deployments – Supports CI/CD for rolling updates and version control.

Kubernetes Advantages (Detailed Explanation)
1) Orchestration – Managing Containers
Kubernetes helps manage multiple Docker containers across different machines (nodes) efficiently.

🔹 Instead of running docker run manually for each container, Kubernetes automates deployment.
🔹 It ensures that all containers are running smoothly and adjusts their status as needed.

Note: Kubernetes ensures all these containers are running, healthy, and communicating with each other properly.

2) Self-Healing – Automatic Recovery
If a container crashes due to an error or system failure, Kubernetes automatically restarts a new instance.

📌 Example:

A web server container (Apache, Nginx) stops unexpectedly.

Kubernetes detects the failure and starts a new container to replace it.

Users never notice downtime.

3) Load Balancing – Distributes Traffic Efficiently
Kubernetes distributes incoming user requests across multiple containers to avoid overloading any single instance.

📌 Example:

A shopping website experiences high traffic during a sale.

Kubernetes ensures that requests are evenly distributed across available backend servers.

Prevents server crashes and ensures smooth performance.

4) Auto Scaling – Adjusting Resources Dynamically
Kubernetes can increase or decrease the number of containers automatically based on traffic load.

📌 Example:

If website traffic increases, Kubernetes adds more containers to handle the load.

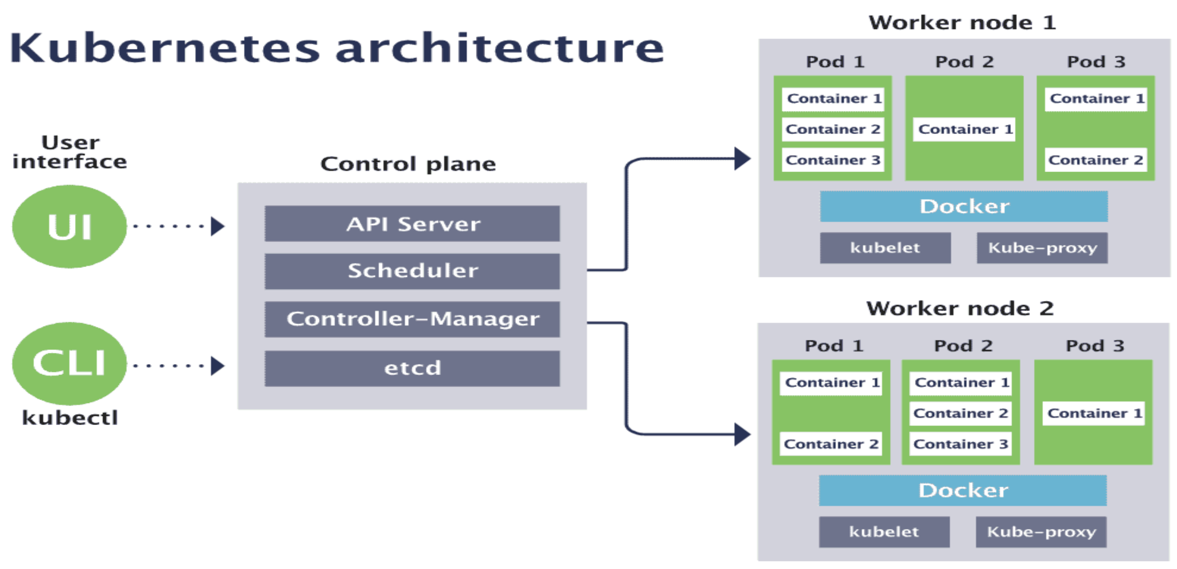
If traffic reduces, Kubernetes removes extra containers to save resources.

Works similarly to cloud-based Auto Scaling Groups (ASG).

Conclusion
🚀 Docker simplifies packaging applications into portable containers.
🚀 Kubernetes ensures these containers are orchestrated, scalable, and reliable.

Together, Docker & Kubernetes enable modern cloud-native application deployment—making applications highly available, efficient, and automated.

##
Kubernetes (K8s) Architecture - Explained in Detail
##



1) Control Plane (Master Node/Control Node)
-> The control plane is responsible for managing the Kubernetes cluster. It includes the following components:

1. API Server: Receives requests from kubectl and manages cluster operations.

2. Scheduler: Identifies pending tasks in ETCD and assigns them to worker nodes.

3. Controller Manager: Ensures the cluster’s desired state matches the actual state.

4. ETCD: A distributed key-value store that acts as Kubernetes' internal database.

2) Worker Nodes (Slave Nodes)
-> Worker nodes run application workloads. They include the following components:

1. Kubelet: A node agent that communicates with the control plane and manages containers.

2. Kube Proxy: Manages networking and ensures communication within the cluster.

3. Docker Engine: Runs and manages containerized applications.

4. Pod: The smallest deployable unit in Kubernetes, housing one or more containers.

5. Container: Runs inside a Pod and contains the application code.

######################################
Explanation k8S working
######################################

Step 1: To deploy an application, we interact with the control plane using the kubectl CLI.

Step 2: The API Server receives the request and stores it in ETCD with a pending status.

Step 3: The Scheduler identifies an available worker node to execute the task, using Kubelet for node management.

Step 4: The Kubelet ensures the worker node is running the assigned workload.

Step 5: The Kube Proxy manages networking for seamless cluster communication.

Step 6: The Controller Manager continuously monitors the cluster to ensure tasks run correctly.

##########
Note:
##########

-> A cluster in Kubernetes (K8s) refers to a group of servers (nodes) that work together to run containerized applications. It consists of:

a. Control Plane (Master Node) – Manages and controls the cluster.

b. Worker Nodes (Slave Nodes) – Run application workloads inside containers.

#####################################
Kubernetes (K8s) Cluster Setup
######################################

A Kubernetes Cluster is a group of servers working together to run containerized applications. It can be set up in two main ways:

A Kubernetes Cluster = Control Plane + Worker Nodes + Pods + Resources + Networking + Storage

1) Self-Managed Cluster
In this setup, we manually install and manage Kubernetes on our own infrastructure.

a) MiniKube (Single Node)
-> Runs a single-node cluster on a local machine.
-> Best for learning and practicing Kubernetes concepts.
-> Not suitable for production as it lacks high availability and scalability.

b) Kubeadm (Multi-Node)
-> A tool for setting up a multi-node cluster manually.
-> Requires configuring the control plane, worker nodes, and networking.
-> Gives full control over the cluster but requires deep Kubernetes expertise.
-> Used for on-premise or customized Kubernetes deployments.

2) Cloud Provider-Managed Cluster (Pre-configured, ready-to-use)
Cloud providers offer fully managed Kubernetes services, where they handle cluster maintenance, updates, and availability.

a) AWS EKS (Elastic Kubernetes Service)
-> A managed Kubernetes service on Amazon Web Services.

b) Azure AKS (Azure Kubernetes Service)
-> Microsoft Azure's managed Kubernetes offering.

c) GCP GKE (Google Kubernetes Engine)
-> Google Cloud's fully managed Kubernetes solution.

########################
MiniKube Setup
######################

Step-1 : Setup Linux VM

Login into AWS Cloud account
Create Linux VM with Ubuntu AMI - t2.medium
Select Storage as 50 GB or more with 2 vCPU required minimum(Default is 8 GB only for Linux)
Create Linux VM and connect to it using SSH Client

Step-2 : Install Docker In Ubuntu VM

```
sudo apt update
curl -fsSL get.docker.com | /bin/bash
sudo usermod -aG docker ubuntu
exit
```

Step-3 : Updating system packages before installing Minikube dependencies

```
sudo apt update
sudo apt install -y curl wget apt-transport-https
```

Step-4 : Installing Minikube

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
minikube version
```

Step-5 : Install Kubectl (Kubernetes Client)

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s
https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl
chmod +x kubectl
sudo mv kubectl /usr/local/bin/
kubectl version -o yaml
```

Step-6 : Start MiniKube Server

```
minikube start — driver=docker
```

Step-7 : Check MiniKube status

```
minikube status
```

Step-8 : Access K8S Cluster

```
kubectl cluster-info
```

Step-9 : Access K8S Nodes

```
kubectl get nodes
```

$$ Setup Completed
$$

######################################
🚀 What is a POD in Kubernetes?
######################################

✅ Key Concepts Explained:

----
1. If you deploy an app, it will ultimately run inside one or more Pods. It is a building block to run app that we deploy in K8S

2. "Applications will be deployed as PODS in k8s."
Your app (e.g., a Spring Boot API) will be containerized using Docker. This container will then be wrapped inside a Pod and deployed on the cluster.

3. "To create PODS we will use Docker images."
A Pod runs one or more containers (usually one), and each container uses a Docker image. You can build a Docker image of your app and then deploy it inside a Pod.

4. "To create PODS we will use Manifest YML file."
A YAML manifest file defines the configuration for the Pod (or other resources like Deployments).

It includes:

a. The name of the Pod

b. The image to use

c. Ports to expose

c. Environment variables, etc.

5. "Create multiple PODS."
The same image (e.g., myapp:latest) can be used to create many Pods. This is how you scale your application—running multiple copies (Pods) to handle more traffic.

6. "If we run application with multiple pods then Load Balancing can be performed resulting in 99.9% uptime of the application."
High Availability: If one Pod crashes, others are still running, so your app stays available.Load Balancing: Kubernetes distributes traffic across Pods using a Service (like a load balancer).

7. "PODS count will be increased and decreased based on the demand (scalability)."
Kubernetes supports auto-scaling. You can scale Pods manually or automatically using a Horizontal Pod Autoscaler (HPA). Based on CPU/memory usage or custom metrics, Kubernetes will increase/decrease the number of Pods.

################################
🚀 Kubernetes Services

##################################

-> A Kubernetes Service is used to expose a group of Pods so that they can be accessed reliably. Since Pods can be created and destroyed at any time (with changing IPs), a Service gives them a stable network identity.

🧭 Why Do We Use Services?

-> Pods are short-lived and can crash or restart.

-> Each time a Pod is created, it gets a new IP address.

-> Directly accessing Pods via IP is not reliable.

-> A Service gives a static IP to a group of Pods.

🌐 Types of Kubernetes Services

Kubernetes offers different types of services depending on how and where you want to expose your Pods:

🔹 1. ClusterIP (Default)
🔸 2. NodePort
🔹 3. LoadBalancer

🔐 ClusterIP Service (Internal Access Only)

📌 Key Points:
-> Pods are short-lived objects; if one crashes, Kubernetes replaces it with a new Pod.
-> Every new Pod gets a different IP address.

Note: 🛑 Never rely on Pod IPs to access an application.

-> A ClusterIP Service groups multiple Pods and assigns them a single static IP.

-> This static IP allows other components inside the cluster to access the group of Pods reliably, even when individual Pods change.

🚫 Access Scope:

-> Only accessible within the Kubernetes cluster.

-> Not reachable from the outside world (internet or external clients).

💡 Use Case:

-> Internal services such as databases, backend APIs, authentication services, etc.

Example: You don't want to expose a database Pod to the internet, so you use a ClusterIP service to allow access only from other internal Pods.

🌐 What is a NodePort Service in Kubernetes?

-> A NodePort service is a type of Kubernetes Service that exposes your Pods outside the cluster using a port on each worker node (called a "NodePort").

🧭 Why Use NodePort?

By default, Pods and ClusterIP services are only accessible within the cluster.

NodePort makes them accessible externally by opening a static port (from 30000 to 32767) on each worker node.

It allows you to access your application using:

http://<NodeIP>:<NodePort>

Note: Here all traffic is routed to one worker node. Means load balancing cannot happen here.

🌐 What is a LoadBalancer Service in Kubernetes?

-> It not only provides external access to your app but also handles automatic traffic distribution across the backend Pods running on different worker nodes.

###
📄 What is a Kubernetes Manifest YAML?
###

-> Think of it as an instruction manual for Kubernetes to create and manage resources.

🧱 Main Sections of a Manifest YAML

Here are the 4 main parts:

```
apiVersion: <version-number>   # API version to use
kind: <resource-type>          # Type of resource (Pod, Service, Deployment, etc.)
metadata:                      # Metadata like name, labels
spec:                          # Specification of what the resource should do
```

🧪 Example: Pod Manifest YAML
Let’s look at a simple Pod definition:

```
---
apiVersion: v1
kind: Pod
metadata:
  name: testpod
  labels:
    app: dempapp
spec:
  containers:
    - name: test
      image: psait/pankajsiracademy:latest
      ports:
        - containerPort: 9090
...
```

Explanation
----

apiVersion: v1
Tells Kubernetes to use version v1 of the API.

Since you are creating a Pod, this is the correct API version.

kind: Pod
Defines the type of resource you want to create.

In this case, it’s a Pod, which is the smallest and simplest unit in Kubernetes.

metadata:
Metadata gives Kubernetes basic info about your Pod.

name: testpod
This is the name of your Pod.

You’ll use this name to check logs or status (e.g., kubectl get pod testpod).

labels:
Labels are key-value pairs to categorize and group Kubernetes objects.

app: dempapp is a label to help identify this Pod as part of the "dempapp" application.

spec:
This section defines what’s inside the Pod.

containers:
A Pod can contain one or more containers. You're defining one container here.

- name: test
This is the name of the container inside the Pod (not the Pod itself).

image: psait/pankajsiracademy:latest
This is the Docker image used to create the container.

It will pull the latest version of psait/pankajsiracademy from Docker Hub or another registry.

⚠️ Make sure the image exists and is accessible (public or with correct credentials).

ports:
This tells Kubernetes the container is listening on port 8080 inside the Pod.

containerPort: 8080
This is the internal port your application is running on.

Kubernetes can use this for things like service routing, health checks, etc.

#############
Commands:

######################

Note: Save above content in .yml file

```
# execute manifest yml
kubectl apply -f <manifest-yml-file>

# check pods
kubectl get pods

# check pod logs
kubectl logs <pod-name>

# Describe pods
kubectl describe pod <pod-name>

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

K8s Service Manifest YAML (for your Pod)

```
---
apiVersion: v1
kind: Service
metadata:
  name: testpod-service
spec:
  type: NodePort
  selector:
    app: dempapp          # This must match the Pod's label
  ports:
    - port: 80            # Exposed port for external access
      targetPort: 9090    # Port on which the app is running inside the container
      nodePort: 30080     # External port exposed on each node
```

💡 Explanation:
----
name: testpod-service – The name of the service.

type: NodePort – Exposes the Pod outside the cluster.

selector.app: dempapp – This matches the label of your Pod, so the service knows which Pod(s) to route to.

port: 80 – The port used when calling the service.Port 80 is the default port for HTTP traffic

You're using a web server like Nginx, Apache, or similar.

targetPort: 8080 – The port your container app actually listens on.

nodePort: 30080 – External port accessible via http://<NodeIP>:30080

Commands

```
# 🔍 Check existing services
kubectl get svc

# 📦 Create the service using the YAML
kubectl apply -f testpod-service.yml

# 🔁 Verify that the service is created
kubectl get svc

# 🚪 Open service in browser (Minikube only)
minikube service testpod-service

# Test
#Get minikube ip address

Test this in same local network

curl http://<mini-kube-ip>:3080/
curl http://192.168.49.2:30080/
```

Part	Meaning
curl	A tool to make HTTP requests from the command line. It's often used to test whether a URL is reachable and what it returns.
http://192.168.49.2	This is the IP address of the Minikube VM. It's the entry point into your Kubernetes cluster from your host machine. You found this IP using minikube ip.
:30080	This is the NodePort exposed by your Kubernetes service (testpod-service). It forwards external requests to the internal Pod's port (8080 in your case).

/ This is the path of the URL. Since it's just a /, it hits the root endpoint of your Spring Boot app.

###
note: How to delete pod and services

kubectl delete pod testpod
kubectl delete svc testpod-service

kubectl apply -f pod-01.yml
kubectl apply -f service-01.yml
##

#############
Stop complete minikube
#################

-> minikube stop
-> minikube delete
-> minikube status

#####################
To see all resources running
####################

-> kubectl get all

########################
To delete all resources use
########################
-> kubectl delete all --all

#######################################
What are name spaces in k8s?
#######################################

-> They help logically group and isolate resources. Just like how we create folders to isolate our work in computers.

Example:
----

database-ns = all database-related stuff

backend-ns = for backend applications

Note: If we donot specifiy name space they k8S will automatically provide default name - space

```
####################
Commands
##################
```

list all name spaces

-> kubectl get ns

list all pods in given name space

-> kubectl get pod -n <name-space>

```
###########################################
How to create name space in k8s
###########################################
```

1. Using kubectl command-
kubectl create namespace backend-ns

2.using manifest yml file

```
---
apiVersion: v1
kind: Namespace
metadata:
 name: backend-ns
...
```

```
# execute manifest yml
kubeclt apply -f <yml-file-name>

# get all resources belongs to backend-ns namespace
kubectl get all -n backend-ns

#get all pods in kube-system
kubectly get pods -n kube-system
```

```
#get all worker nodes
kubectl get nodes

#delete name space - All resource related to that will be deleted
kubectl delete ns backend-ns

#Open tunnel
minikube service <service-name>
```

```
############################
Namespace with POD with Service creation yml file
##############################
---
apiVersion: v1
kind: Namespace
metadata:
 name: backend-ns
---
apiVersion: v1
kind: Pod
metadata:
  name: testpod
  namespace: backend-ns
  labels:
    app: dempapp
spec:
  containers:
    - name: test
      image: psait/pankajsiracademy:latest
      ports:
        - containerPort: 9090
---
apiVersion: v1
kind: Service
metadata:
  name: testpod-service
  namespace: backend-ns
spec:
  type: NodePort
  selector:
    app: dempapp           # This must match the Pod's label
  ports:
    - port: 80             # Exposed port for external access
      targetPort: 9090     # Port on which the app is running inside the container
      nodePort: 30080      # External port exposed on each node
...
```

##
k8S Resources
###

-> When you create a Pod directly using kind: Pod, Kubernetes does not manage its lifecycle — if it crashes or is deleted, it's gone forever unless recreated manually.

-> K8S resources manages POD lifecycle

-> To let Kubernetes manage, restart, and scale Pods, we use higher-level controllers like the ones you listed.

🔁 1) ReplicationController (RC)
🔁 2) ReplicaSet (RS)
🚀 3) Deployment
🛰️ 4) DaemonSet
💾 5) StatefulSet

######################################
📦 What is ReplicationController (RC)?
A Kubernetes resource used to manage the lifecycle of Pods.

Ensures a specified number of Pods are always running.

Provides self-healing — if a Pod crashes or is deleted, RC will recreate it.

manifest yml file

```
---
apiVersion: v1
kind: ReplicationController
metadata:
 name: webapp
spec:
 replicas: 3
 selector:
  app: dempapp
 template:
  metadata:
   name: testpod
   labels:
    app: dempapp
  spec:
   containers:
    - name: webappcontainer
      image: psait/pankajsiracademy:latest
      ports:
      - containerPort: 9090
...
```

```
kubectl apply -f rc.yml
```

##

```
kubectl get all

kubectl get pods

kubectl delete pod <pod-name>

kubectl get pods

kubectl scale rc dempapp --replicas=5

kubectl scale rc dempapp --replicas=1  Explain in short
```

##