**DOCKER**

**1. Introduction to Docker**

Docker is an open-source centralized platform designed to create, deploy, and run applications using containers. It was first released in March 2013 by Solomon Hykes and Sebastien Pahl. Docker allows applications to use the same Linux kernel as the host system, rather than creating a full virtual OS, enabling OS-level virtualization (containerization). This solves issues where code runs on a developer's system but fails on a user's due to environmental differences. Docker is written in Go and runs natively on Linux distributions, though it can be installed on any OS.

**Advantages**

* No pre-allocation of RAM.
* Improves CI efficiency by using the same container image across deployment steps.
* Cost-effective and lightweight.
* Can run on physical hardware, virtual hardware, or cloud.
* Reusable images.
* Quick container creation.

**Disadvantages**

* Not ideal for applications requiring rich GUIs.
* Challenging to manage large numbers of containers.
* No cross-platform compatibility (e.g., Windows-designed containers won't run on Linux without adjustments).
* Use virtual machines if development and testing OS differ.
* Lacks built-in data recovery and backup solutions.

**Note on Images vs. Containers**

A running image is a container; a stopped or non-runnable container is an image.

**All Docker Commands for Introduction/Setup**

* docker --version: Check Docker version.
* docker info: Get system-wide information.

**2. Docker Architecture**

Docker's architecture includes:

* **Docker Client**: Users interact via commands and REST API. Sends commands to the Docker Daemon (e.g., docker build, docker pull, docker run). Can communicate with multiple daemons.
* **Docker Daemon**: Runs on the host OS, manages containers, images, and services. Communicates with other daemons.
* **Docker Host**: Provides the environment for execution, containing the daemon, images, containers, networks, and storage.
* **Docker Registry**: Stores and manages images. Types: Public (Docker Hub) for sharing publicly; Private for enterprise sharing.
* **Docker Images**: Read-only templates with dependencies and configurations to run programs.
* **Docker Containers**: Runtime instances of images, like virtualized environments on the Docker Engine.

The client communicates with the daemon on the host, which pulls images from registries to create containers.

**All Docker Commands for Architecture**

* No direct commands, but use docker system info for daemon/host details.
* docker version: Client and server version info.

**3. Installing Docker**

Create an AWS machine with a Docker-installed AMI or install via:

* yum install docker (on yum-based systems like Amazon Linux).
* apt update && apt install docker.io (on Debian/Ubuntu).
* For macOS/Windows: Download Docker Desktop from official site.

Key commands after installation:

* service docker status or systemctl status docker: Check if the service is running.
* service docker start or systemctl start docker: Start the service.
* docker run hello-world: Test installation by running a test container.

**All Docker Commands for Installation**

* sudo systemctl enable docker: Enable Docker to start on boot.
* sudo usermod -aG docker $USER: Add user to docker group (logout/login required).

**4. Docker Images**

Images are read-only binary templates for creating containers, containing all dependencies.

**Ways to Create Images**

1. Pull from Docker Hub: docker pull <image\_name>.
2. From a Dockerfile (see section on Dockerfile).
3. From an existing container: Commit changes to a new image.

Commands:

* docker images or docker image ls: List local images.
* docker search <image\_name>: Search Docker Hub.
* docker pull <image\_name>: Download from Docker Hub.
* docker rmi <image\_name> or docker image rm <image\_name>: Delete an image.
* docker diff <container\_name>: See differences between base image and changes.
* docker commit <container\_name> <new\_image\_name>: Create image from container.
* docker image inspect <image\_name>: Inspect image details.
* docker image tag <source\_image> <target\_image>: Tag an image.
* docker image push <image\_name>: Push to registry (after login).
* docker image history <image\_name>: Show image layers history.
* docker image prune: Remove unused images.

Example: Create container from base image, make changes (e.g., create file), commit to new image.

**5. Docker Containers**

Containers hold the full package needed to run an application. An image is a template; a container is a running copy.

Commands:

* docker run -it --name <container\_name> <image\_name> /bin/bash: Run interactively with name.
* docker start <container\_name>: Start a stopped container.
* docker stop <container\_name>: Stop a running container.
* docker attach <container\_name>: Enter running container (attach to main process).
* docker exec -it <container\_name> /bin/bash: Execute command in running container (new process).
* docker ps -a: List all containers.
* docker ps: List running containers.
* docker rm <container\_name>: Delete a stopped container.
* docker rm -f <container\_name>: Force delete running container.
* exit: Exit container.
* docker container ls: Alias for docker ps.
* docker container inspect <container\_name>: Inspect container details.
* docker container logs <container\_name>: View logs.
* docker container top <container\_name>: See processes inside.
* docker container stats <container\_name>: Resource usage stats.
* docker container prune: Remove stopped containers.
* docker kill <container\_name>: Kill running container.
* docker restart <container\_name>: Restart container.
* docker run -d <image\_name>: Run detached (background).
* docker run --rm <image\_name>: Auto-remove on exit.

Difference between docker attach and docker exec:

* attach: Connects to the main process's STDIN/STDOUT/STDERR.
* exec: Runs a new process in the container (e.g., for shells or commands).

Port Exposure:

* --publish or -p: Map ports (e.g., -p 8080:80 maps host 8080 to container 80).
* EXPOSE in Dockerfile exposes ports internally.
* If neither: Service accessible only inside container.
* Only EXPOSE: Accessible from other containers.
* EXPOSE + -p: Accessible externally.
* -p 8080:80/tcp: Specify protocol.
* -p 127.0.0.1:8080:80: Bind to localhost only.

**6. Docker Networking**

Docker networking allows containers to communicate with each other and external workloads. Containers see a network interface with IP, gateway, routing table, and DNS.

**Types of Networks**

* **Bridge**: Default; for same-host container communication.
* **Host**: Uses host's network directly (no isolation). Use: docker run --network host ....
* **Overlay**: For multi-host (swarm) communication.
* **Macvlan**: Assigns MAC address, appears as physical device.
* **IPvlan**: Controls IPv4/IPv6 addressing.
* **None**: No networking. Use: docker run --network none ....

**Creating and Managing**

* docker network create -d bridge my-net: Create a bridge network.
* docker run --network=my-net <image\_name>: Run in specific network.
* docker network connect <network> <container>: Connect running container to network.
* docker network disconnect <network> <container>: Disconnect.
* docker network ls: List networks.
* docker network inspect <network>: Inspect network.
* docker network rm <network>: Remove network.
* docker network prune: Remove unused networks.
* --hostname <name>: Set container hostname.
* --dns <ip>: Set DNS server.
* --add-host <host:ip>: Add host entry.

**Communication**

Containers on same user-defined network use names/IPs. Use --network container:<id> for shared stack.

**Exposing Ports**

* -p 8080:80: Map TCP.
* -p 192.168.1.100:8080:80: Specific IP.
* -p 127.0.0.1:8080:80: Restrict to localhost.
* -p 8080:80/udp: UDP protocol.

**Best Practices**

* Use user-defined bridges for isolation.
* Restrict to localhost for security.
* Avoid unnecessary port publishing.

**7. Docker Volumes and Storage**

Volumes are directories in containers for persistent data, decoupled from containers.

**Benefits**

* Decouple from storage.
* Share across containers.
* Attach to containers.
* Persist after container deletion.

**Creating**

* In Dockerfile: VOLUME ["/myvolume1"].
* Command: docker volume create myvol.
* When running: docker run -v /host/path:/container/path ... (bind mount).
* -v myvol:/container/path: Named volume.
* --mount source=myvol,target=/container/path: Alternative mount syntax.

**Mapping**

* Container-to-Container: Share via same volume name.
* Host-to-Container: -v /host/dir:/container/dir.

Commands:

* docker volume ls: List volumes.
* docker volume create <name>: Create volume.
* docker volume rm <name>: Delete volume.
* docker volume prune: Remove unused volumes.
* docker volume inspect <name>: Details.
* docker inspect <container>: Container details (includes mounts).
* docker run -v /dev/null:/tmp: Temporary discard.

Example: Create image with VOLUME, run, share with another container.

**8. Docker Compose**

Docker Compose defines and runs multi-container apps via YAML file (docker-compose.yml).

**Overview**

Manages services, networks, volumes. Suitable for dev, test, prod. Install via pip install docker-compose or package managers.

**docker-compose.yml Structure**

Defines:

* **Services**: Containers (e.g., image, ports, volumes).
* **Networks**: For communication.
* **Volumes**: For persistence.

Basic Example:

version: '3'

services:

web:

image: nginx

ports:

- "80:80"

db:

image: mongo

volumes:

- db\_data:/data/db

volumes:

db\_data:

Key Commands:

* docker-compose up: Start services (add -d for detached).
* docker-compose down: Stop and remove (add -v to remove volumes).
* docker-compose build: Build or rebuild services.
* docker-compose ps: List running services.
* docker-compose logs <service>: View logs.
* docker-compose exec <service> <command>: Run command in service.
* docker-compose restart <service>: Restart service.
* docker-compose stop <service>: Stop service.
* docker-compose start <service>: Start service.
* docker-compose run <service> <command>: Run one-off command.
* docker-compose config: Validate compose file.
* docker-compose pull: Pull images.
* docker-compose images: List images.

**9. Docker Registry**

Registries store images. Docker Hub is public; private for enterprises.

**Docker Hub**

Public registry for sharing.

**Private Registries**

Deploy via open-source "Distribution" (CNCF project, GitHub: <https://github.com/distribution/distribution>).

**Pushing/Pulling**

* docker push <repo/image:tag>: Push.
* docker pull <repo/image:tag>: Pull.
* docker login: Authenticate.
* docker logout: Logout.
* docker tag <image> <registry/image:tag>: Tag for push.

**Best Practices**

Use private for sensitive images; tag versions.

(Note: Registry is open-source, donated to CNCF in 2019.)

**10. Multi-stage Docker Builds**

Multi-stage builds use multiple FROM in Dockerfile to optimize, separating build and runtime.

**Why Use**

Reduce image size by discarding build tools.

**Syntax**

* Multiple FROM AS <stage>.
* COPY --from=<stage> /path /dest.
* Build: docker build --target <stage> -t <image> . (for specific stage).

Example (for Angular):

# Build stage

FROM node:20-alpine AS build

WORKDIR /app

COPY package\*.json ./

RUN npm install

COPY . .

RUN npm run build --prod

# Runtime stage

FROM nginx:alpine

COPY --from=build /app/dist/angular-app /usr/share/nginx/html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]

Example (for Spring Boot):

# Build stage

FROM maven:3.8.6-eclipse-temurin-17 AS build

WORKDIR /app

COPY pom.xml .

COPY src ./src

RUN mvn clean package -DskipTests

# Runtime stage

FROM eclipse-temurin:17-jre-alpine

WORKDIR /app

COPY --from=build /app/target/\*.jar app.jar

EXPOSE 8080

ENTRYPOINT ["java", "-jar", "app.jar"]

**Optimizing**

Copy only artifacts; use --target <stage> to build specific stage.

**Use Cases**

Compile binaries, debug stages.

**Legacy vs. BuildKit**

BuildKit skips unrelated stages for efficiency. Enable: DOCKER\_BUILDKIT=1 docker build ....

**All Docker Commands for Builds**

* docker build -t <image> .: Build image.
* docker build --no-cache -t <image> .: No cache.
* docker build --squash -t <image> .: Squash layers (experimental).

**11. Monitoring and Logging in Docker**

**Logging**

* Drivers: Send to files, hosts, DBs.
* Configure: Symlinks (e.g., Nginx logs to /dev/stdout).
* docker logs <container>: View STDOUT/STDERR.
* docker logs -f <container>: Follow logs.
* docker logs --tail 100 <container>: Last N lines.
* --log-driver json-file: Default driver.

**Monitoring**

* docker stats <container>: Container metrics (CPU, memory).
* docker stats --no-stream: One-shot stats.
* docker top <container>: Processes.

**Integration**

Use drivers for external systems (e.g., ELK stack). Configure: docker run --log-driver syslog ....

**All Docker Commands for Monitoring/Logging**

* docker events: Stream events.
* docker system df: Disk usage.
* docker system prune: Clean up everything.

**Dockerfile Creation**

Dockerfile is a text file with instructions for automating image creation.

**Key Instructions**

* FROM <image>: Base image (must be first).
* RUN <command>: Execute command (creates layer).
* MAINTAINER <name>: Deprecated; use LABEL.
* LABEL <key>=<value>: Metadata.
* COPY <src> <dest>: Copy from local (no remote).
* ADD <src> <dest>: Like COPY, but adds URL download and tar extraction.
* EXPOSE <port>: Expose port (documentation).
* WORKDIR <path>: Set working directory.
* CMD <command>: Default command (JSON array or exec form).
* ENTRYPOINT <command>: Entry point (higher priority than CMD).
* ENV <key>=<value>: Set environment variable.
* ARG <key>=<value>: Build-time variable (use --build-arg).
* VOLUME <path>: Declare volume.
* USER <user>: Run as user.
* ONBUILD <command>: For child images.

**Creation Steps**

1. Create Dockerfile.
2. Add instructions.
3. Build: docker build -t <image> ..
4. Run: docker run <image>.

Example Basic Dockerfile:

FROM ubuntu

RUN apt update && apt install -y curl

COPY testfile /tmp/

EXPOSE 80

CMD ["echo", "Hello"]

Update: docker build -t new-image . (rebuilds).

**All Docker Commands for Dockerfile**

* docker buildx build -t <image> .: Advanced build.
* docker history <image>: Layers from Dockerfile.

**Projects**

**Project 1: 3-Tier Application with Docker Compose (Angular Frontend, Spring Boot Backend, MongoDB, Postgres)**

This project demonstrates a 3-tier full-stack application: Angular for frontend, Spring Boot for backend, using MongoDB (for NoSQL data) and Postgres (for relational data). The backend connects to both databases. Use Docker Compose to orchestrate.

**Assumptions**:

* Backend: Simple CRUD APIs (e.g., /items for MongoDB, /users for Postgres).
* Frontend: Angular app consuming the APIs.
* Place Dockerfiles in respective directories (frontend/, backend/).

**Project Codes (Dockerfiles and Compose Only)**

**Backend (Spring Boot) Dockerfile** (multi-stage):

# Build stage

FROM maven:3.8.6-eclipse-temurin-17 AS build

WORKDIR /app

COPY pom.xml .

COPY src ./src

RUN mvn clean package -DskipTests

# Runtime stage

FROM eclipse-temurin:17-jre-alpine

WORKDIR /app

COPY --from=build /app/target/\*.jar app.jar

EXPOSE 8080

ENTRYPOINT ["java", "-jar", "app.jar"]

**Frontend (Angular) Dockerfile** (multi-stage with Nginx):

# Build stage

FROM node:20-alpine AS build

WORKDIR /app

COPY package\*.json ./

RUN npm install

COPY . .

RUN npm run build --prod

# Runtime stage

FROM nginx:alpine

COPY --from=build /app/dist/angular-app /usr/share/nginx/html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]

**docker-compose.yml**:

version: '3.8'

services:

mongodb:

image: mongo:latest

container\_name: mongodb

ports:

- "27017:27017"

volumes:

- mongo-data:/data/db

networks:

- app-network

postgres:

image: postgres:latest

container\_name: postgres

environment:

POSTGRES\_DB: usersdb

POSTGRES\_USER: postgres

POSTGRES\_PASSWORD: password

ports:

- "5432:5432"

volumes:

- pg-data:/var/lib/postgresql/data

networks:

- app-network

backend:

build: ./backend

container\_name: spring-backend

ports:

- "8080:8080"

depends\_on:

- mongodb

- postgres

environment:

SPRING\_DATA\_MONGODB\_URI: mongodb://mongodb:27017/itemsdb

SPRING\_DATASOURCE\_URL: jdbc:postgresql://postgres:5432/usersdb

SPRING\_DATASOURCE\_USERNAME: postgres

SPRING\_DATASOURCE\_PASSWORD: password

networks:

- app-network

frontend:

build: ./frontend

container\_name: angular-frontend

ports:

- "80:80"

depends\_on:

- backend

networks:

- app-network

volumes:

mongo-data:

pg-data:

networks:

app-network:

driver: bridge

**Usage**:

* Build and run: docker-compose up --build.
* Access frontend at <http://localhost>.
* Backend APIs at <http://localhost:8080/items> (Mongo) or /users (Postgres).
* Commands: docker-compose logs backend for logs; docker-compose down -v to clean up.

**Project 2: Deploying a Web Application with Angular, Spring Boot, MongoDB, and Postgres**

Similar to Project 1, but optimized for deployment (e.g., production profiles, health checks). Use the same Dockerfiles, but update compose for production.

**Updated docker-compose.yml** (with health checks):

version: '3.8'

services:

mongodb:

image: mongo:latest

container\_name: mongodb

ports:

- "27017:27017"

volumes:

- mongo-data:/data/db

healthcheck:

test: ["CMD", "mongosh", "--eval", "db.adminCommand('ping')"]

interval: 10s

timeout: 5s

retries: 5

networks:

- app-network

postgres:

image: postgres:latest

container\_name: postgres

environment:

POSTGRES\_DB: usersdb

POSTGRES\_USER: postgres

POSTGRES\_PASSWORD: password

ports:

- "5432:5432"

volumes:

- pg-data:/var/lib/postgresql/data

healthcheck:

test: ["CMD-SHELL", "pg\_isready -U postgres"]

interval: 10s

timeout: 5s

retries: 5

networks:

- app-network

backend:

build: ./backend

container\_name: spring-backend

ports:

- "8080:8080"

depends\_on:

mongodb:

condition: service\_healthy

postgres:

condition: service\_healthy

environment:

SPRING\_PROFILES\_ACTIVE: prod

SPRING\_DATA\_MONGODB\_URI: mongodb://mongodb:27017/itemsdb

SPRING\_DATASOURCE\_URL: jdbc:postgresql://postgres:5432/usersdb

SPRING\_DATASOURCE\_USERNAME: postgres

SPRING\_DATASOURCE\_PASSWORD: password

networks:

- app-network

frontend:

build: ./frontend

container\_name: angular-frontend

ports:

- "80:80"

depends\_on:

- backend

networks:

- app-network

volumes:

mongo-data:

pg-data:

networks:

app-network:

driver: bridge

**Usage**:

* Build and run: docker-compose up --build -d.
* Health checks ensure DBs are ready before backend starts.
* Scale: docker-compose up --scale backend=2.
* For production: Add restart policies (restart: always) and secrets.

**KUBERNETES**

**Introduction & Overview of K8s Concept**

Kubernetes (K8s) is an open-source container orchestration platform for automating deployment, scaling, and management of containerized applications. It abstracts away the underlying infrastructure, allowing developers to focus on application logic. K8s manages clusters of nodes, scheduling containers (via Pods) across them, handling failures, scaling, and networking. Key concepts include declarative configuration (define desired state in YAML), self-healing (restarts failed containers), and service discovery. It supports microservices architectures and integrates with tools like Docker for container runtime.

**All K8s Commands for Introduction/Setup**

* kubectl version: Check K8s client and server version.
* kubectl cluster-info: Display cluster information.
* kubectl get nodes: List nodes in the cluster.

**K8s History**

Kubernetes was originally developed by Google as an internal project called Borg, later reimplemented as Kubernetes in Go language. It was announced in June 2014 and donated to the Cloud Native Computing Foundation (CNCF) in 2015. The first stable release (v1.0) came in July 2015. It evolved from Google's 15 years of experience running production workloads at scale. Key milestones: v1.2 (2016) added federation, v1.10 (2018) stabilized core APIs, and recent versions (e.g., v1.28 in 2024) focus on security and extensibility. As of 2025, it's the de facto standard for container orchestration.

**Why Should We Learn K8s**

K8s is essential for modern DevOps as it solves scalability, portability, and management challenges in cloud-native applications. It enables efficient resource utilization, auto-scaling, rolling updates, and multi-cloud deployments. Learning K8s boosts career prospects in cloud engineering, as it's used by 96% of organizations (CNCF survey 2024). It reduces downtime through self-healing and supports CI/CD pipelines. For microservices, it provides service discovery, load balancing, and secrets management out-of-the-box.

**Monolithic vs Microservices**

* **Monolithic**: Single, large application with all components (UI, business logic, DB) tightly coupled. Pros: Simple development, easy deployment. Cons: Hard to scale, any change requires full redeployment, single failure point. Example: Traditional web app in one JAR/WAR file.
* **Microservices**: Application broken into small, independent services communicating via APIs (e.g., REST, gRPC). Pros: Independent scaling, faster deployments, technology diversity. Cons: Increased complexity in networking, monitoring, data consistency. K8s excels in managing microservices by orchestrating services as Pods, handling traffic with Services, and scaling with HPA.

**K8s Architecture, Kubectl**

K8s architecture consists of a Control Plane (master) and Worker Nodes.

* **Control Plane Components**:
  + **ETCD**: Key-value datastore for cluster data (nodes, pods, configs, secrets, roles, bindings). It's a distributed, consistent store. Commands: etcdctl get /registry/pods/default/my-pod (using etcdctl CLI).
  + **Kube-apiserver**: Central API for authentication, validation, and communication with other components. It updates ETCD and serves as the frontend for the cluster.
  + **Kube-controller-manager**: Monitors cluster state and works toward desired state. Includes node controller (manages node health, checks every 5s, marks unreachable after 40s, replaces after 5min), replication controller (ensures desired pods for ReplicaSets), cronjob/deployment controllers, persistent volume protection, binder, etc.
  + **Kube-scheduler**: Assigns pods to nodes based on resources, affinity, taints.
  + **Cloud-controller-manager**: Integrates with cloud providers (e.g., AWS, GCP).
* **Worker Node Components**:
  + **Kubelet**: Agent on each node, communicates with API server, manages pods. Checks node health every 5s.
  + **Kube-proxy**: Manages network rules for service communication.
  + **Container Runtime**: e.g., containerd (via CRI - Container Runtime Interface). Docker shim was used but deprecated; alternatives like CRI-O.
* **Kubectl**: CLI for interacting with the cluster. It's the primary tool for commands like create, get, describe.

**All K8s Commands for Architecture**

* kubectl get componentstatuses: Check control plane health (deprecated in newer versions, use kubectl get cs).
* kubectl describe node <node-name>: Node details.
* kubectl get events: Cluster events.
* kubectl config view: See current config.

**Create K8s Architecture by Own**

To create a custom K8s cluster, use tools like Kubeadm for production setups. This involves setting up master and worker nodes manually. Steps: Install CRI (containerd), kubelet, kubeadm on all nodes; initialize master with kubeadm init; join workers with kubeadm join. For hands-on, use VMs or cloud instances.

**AWS Setup for Hands-On**

Use AWS EC2 for nodes. Create t2.medium instances with Ubuntu, open ports (6443 for API, 10250 for kubelet). Install Docker/containerd, then K8s components. Use EKS for managed clusters.

Commands:

* aws ec2 run-instances --image-id ami-xxx --count 3 --instance-type t2.medium: Create instances.
* SSH to instances, then install K8s.

**What is KIND Cluster & Installation of It**

KIND (Kubernetes IN Docker) is a tool for running local K8s clusters using Docker containers as nodes. Ideal for testing. Installation: Download from <https://kind.sigs.k8s.io/dl/v0.23.0/kind-linux-amd64>, make executable, move to /usr/local/bin/kind.

**KIND Setup on Local**

Setup: Install Docker, then KIND. Create config YAML for multi-node if needed.

**KIND Cluster Creation**

Commands:

* kind create cluster --name tws-cluster --config config.yml: Create cluster.
* kubectl config use-context kind-tws-cluster: Switch context.
* kind delete cluster --name tws-cluster: Delete.

Example config.yml for multi-node:

text

kind: Cluster

apiVersion: kind.x-k8s.io/v1alpha4

nodes:

- role: control-plane

- role: worker

- role: worker

**What is Minikube Cluster, Installation, Cluster Creation**

Minikube is a tool for running single-node K8s cluster locally for development. Installation: Download minikube binary, install driver (Docker/VirtualBox).

Commands:

* minikube start --driver=docker: Start cluster.
* minikube status: Check status.
* minikube delete: Delete.
* minikube dashboard: Open dashboard.

**Kubeadm, Setup & Installation, Cluster Creation**

Kubeadm is for bootstrapping clusters. Setup: Install container runtime, kubelet, kubeadm, kubectl on all nodes.

Steps for master:

* kubeadm init --pod-network-cidr=192.168.0.0/16: Initialize.
* Copy join command for workers.

For workers: kubeadm join <token>.

Install CNI (e.g., Calico): kubectl apply -f calico.yaml.

Commands:

* kubeadm reset: Reset node.
* kubeadm upgrade plan: Check upgrades.

**Namespaces**

Namespaces divide cluster resources for multi-tenancy.

Commands:

* kubectl create namespace monitoring: Create.
* kubectl get namespace: List.
* kubectl label namespace monitoring team=devops: Label.
* kubectl describe namespace monitoring: Describe.
* kubectl delete namespace monitoring: Delete.

**Pods**

Pods are the smallest deployable units, containing one or more containers.

Commands:

* kubectl run nginx --image=nginx -n nginx: Create pod.
* kubectl describe pod nginx -n nginx: Describe.
* kubectl logs pod-name -n namespace: Logs.
* kubectl exec -it pod-name -n namespace -- bash: Exec.
* kubectl get pods -o wide: List with IP.

Example YAML:

text

apiVersion: v1

kind: Pod

metadata:

name: nginx

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

**Deployments**

Deployments manage ReplicaSets for stateless apps, supporting rolling updates.

Commands:

* kubectl apply -f deployment.yml: Apply.
* kubectl scale deployment nginx-deployment --replicas=3 -n nginx: Scale.
* kubectl rollout status deployment/nginx-deployment: Status.
* kubectl rollout undo deployment/nginx-deployment: Rollback.

Example YAML:

text

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

**ReplicaSet vs StatefulSet vs Deployments**

* **ReplicaSet**: Ensures specified number of pod replicas. Managed by Deployments.
* **StatefulSet**: For stateful apps, provides stable identity, ordered deployment/scaling, persistent storage.
* **Deployment**: Manages ReplicaSets for stateless apps, handles updates.

**Labels and Selectors**

Labels are key-value pairs for organizing resources. Selectors match labels for targeting.

Commands:

* kubectl label pod my-pod app=web: Label.
* kubectl get pods -l app=web: Get by label.

Example in YAML:

text

selector:

matchLabels:

app: web

**Rolling Update in Deployment**

Updates pods gradually without downtime.

Commands:

* kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1: Update image.
* kubectl rollout history deployment/nginx-deployment: History.

In YAML, set strategy: type: RollingUpdate.

**ReplicaSets**

Commands:

* kubectl get rs: List.
* kubectl describe rs <name>: Describe.

**DaemonSets**

DaemonSets run one pod per node, for daemon-like tasks (e.g., monitoring).

Commands:

* kubectl apply -f daemonset.yml: Apply.
* kubectl get daemonsets: List.
* kubectl scale daemonset <name> --replicas=0: Scale (but DaemonSets ignore replicas).

Example YAML:

text

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: fluentd-elasticsearch

labels:

k8s-app: fluentd-logging

spec:

selector:

matchLabels:

name: fluentd-elasticsearch

template:

metadata:

labels:

name: fluentd-elasticsearch

spec:

containers:

- name: fluentd-elasticsearch

image: quay.io/fluentd\_elasticsearch/fluentd:v2.5.2

**Jobs**

Jobs run finite tasks to completion.

Commands:

* kubectl apply -f job.yaml: Apply.
* kubectl get jobs: List.
* kubectl logs job/<name>: Logs.

Example YAML:

text

apiVersion: batch/v1

kind: Job

metadata:

name: pi

spec:

template:

spec:

containers:

- name: pi

image: perl:5.34.0

command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]

restartPolicy: Never

backoffLimit: 4

**CronJobs**

CronJobs run Jobs on schedule.

Commands:

* kubectl apply -f cronjob.yaml: Apply.
* kubectl get cronjobs: List.

Example YAML:

text

apiVersion: batch/v1

kind: CronJob

metadata:

name: hello

spec:

schedule: "\*/1 \* \* \* \*"

jobTemplate:

spec:

template:

spec:

containers:

- name: hello

image: busybox:1.28

args:

- /bin/sh

- -c

- date; echo Hello from the Kubernetes cluster

restartPolicy: OnFailure

**Storage**

K8s storage abstracts persistent data.

**Persistent Volume (PV)**

PV is cluster-wide storage.

Commands:

* kubectl apply -f pv.yaml: Apply.
* kubectl get pv: List.

Example YAML:

text

apiVersion: v1

kind: PersistentVolume

metadata:

name: pv0001

spec:

capacity:

storage: 1Gi

volumeMode: Filesystem

accessModes:

- ReadWriteOnce

persistentVolumeReclaimPolicy: Retain

storageClassName: manual

hostPath:

path: /data/pv0001/

**StorageClasses**

StorageClasses define storage provisioners.

Commands:

* kubectl get storageclass: List.

Example YAML:

text

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: standard

provisioner: kubernetes.io/aws-ebs

parameters:

type: gp2

reclaimPolicy: Retain

mountOptions:

- debug

**Persistent Volume Claim (PVC)**

PVC requests storage from PV.

Commands:

* kubectl apply -f pvc.yaml: Apply.
* kubectl get pvc: List.

Example YAML:

text

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: pvc0001

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 1Gi

storageClassName: manual

**Services**

Services expose pods.

Commands:

* kubectl apply -f service.yml: Apply.
* kubectl describe svc nginx-service -n nginx: Describe.
* kubectl get svc -A: List.

Example YAML:

text

apiVersion: v1

kind: Service

metadata:

name: my-service

spec:

selector:

app: MyApp

ports:

- protocol: TCP

port: 80

targetPort: 9376

**Small Project (django-notes-app)**

For a small Django app, containerize with Docker, then deploy as Deployment with Service.

Example Deployment YAML for Django:

text

apiVersion: apps/v1

kind: Deployment

metadata:

name: django-app

spec:

replicas: 2

selector:

matchLabels:

app: django

template:

metadata:

labels:

app: django

spec:

containers:

- name: django

image: your-dockerhub/django-notes-app

ports:

- containerPort: 8000

Service:

text

apiVersion: v1

kind: Service

metadata:

name: django-service

spec:

type: LoadBalancer

selector:

app: django

ports:

- port: 80

targetPort: 8000

Commands:

* kubectl apply -f django-deployment.yaml
* kubectl apply -f django-service.yaml

**Ingress**

Ingress manages external access to services.

Commands:

* kubectl apply -f ingress.yml: Apply.
* kubectl describe ingress nginx-ingress -n nginx: Describe.

Example YAML:

text

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: minimal-ingress

spec:

rules:

- http:

paths:

- path: /

pathType: Prefix

backend:

service:

name: web

port:

number: 8080

**Annotations**

Annotations are metadata for resources, e.g., in Ingress for custom configs.

Example in YAML:

text

metadata:

annotations:

nginx.ingress.kubernetes.io/rewrite-target: /

**StatefulSets**

For stateful apps.

Commands:

* kubectl apply -f statefulset.yaml: Apply.
* kubectl get statefulsets: List.

Example YAML:

text

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: web

spec:

selector:

matchLabels:

app: nginx

serviceName: "nginx"

replicas: 3

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: registry.k8s.io/nginx-slim:0.8

ports:

- containerPort: 80

name: web

volumeMounts:

- name: www

mountPath: /usr/share/nginx/html

volumeClaimTemplates:

- metadata:

name: www

spec:

accessModes: [ "ReadWriteOnce" ]

resources:

requests:

storage: 1Gi

**ConfigMaps**

For non-sensitive config.

Commands:

* kubectl create configmap app-config --from-file=config.properties: Create.
* kubectl get configmaps: List.

Example YAML:

text

apiVersion: v1

kind: ConfigMap

metadata:

name: game-demo

data:

player\_initial\_lives: "3"

ui\_properties\_file\_name: "user-interface.properties"

**Secrets**

For sensitive data.

Commands:

* kubectl create secret generic db-credentials --from-literal=username=admin --from-literal=password=admin123: Create.
* kubectl get secrets: List.

Example YAML:

text

apiVersion: v1

kind: Secret

metadata:

name: mysecret

type: Opaque

data:

password: dmFsdWUtMg0K

username: dmFsdWUtMQ0K

**Resource Quotas and Limits**

Quotas limit resources per namespace; limits per pod/container.

Commands:

* kubectl apply -f quota.yaml: Apply quota.
* kubectl describe quota: Describe.

Example Quota YAML:

text

apiVersion: v1

kind: ResourceQuota

metadata:

name: mem-cpu-quota

spec:

hard:

requests.cpu: "1"

requests.memory: 1Gi

limits.cpu: "2"

limits.memory: 2Gi

Limits in Pod YAML:

text

spec:

containers:

- name: db

resources:

limits:

memory: "200Mi"

cpu: "700m"

requests:

memory: "200Mi"

cpu: "700m"

**Probes**

Probes check container health.

* Liveness: Restart if failed.
* Readiness: Remove from service if not ready.
* Startup: Delay other probes.

Commands:

* kubectl describe pod <pod>: See probe status.

Example YAML:

text

spec:

containers:

- name: liveness

image: registry.k8s.io/busybox

args:

- /bin/sh

- -c

- touch /tmp/healthy; sleep 30; rm -f /tmp/healthy; sleep 600

livenessProbe:

exec:

command:

- cat

- /tmp/healthy

initialDelaySeconds: 5

periodSeconds: 5

**Taints/Tolerations**

Taints repel pods from nodes; tolerations allow pods to tolerate taints.

Commands:

* kubectl taint nodes node1 key1=value1:NoSchedule: Add taint.
* kubectl describe node node1: See taints.

Example Toleration in YAML:

text

spec:

tolerations:

- key: "key1"

operator: "Equal"

value: "value1"

effect: "NoSchedule"

**HPA**

Horizontal Pod Autoscaler scales pods based on metrics.

Commands:

* kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10: Create.
* kubectl get hpa: List.

Example YAML:

text

apiVersion: autoscaling/v2

kind: HorizontalPodAutoscaler

metadata:

name: php-apache

spec:

scaleTargetRef:

apiVersion: apps/v1

kind: Deployment

name: php-apache

minReplicas: 1

maxReplicas: 10

metrics:

- type: Resource

resource:

name: cpu

target:

type: Utilization

averageUtilization: 50

**VPA**

Vertical Pod Autoscaler adjusts pod resources.

Commands:

* kubectl apply -f vpa.yaml: Apply.
* kubectl get vpa: List.

Example YAML:

text

apiVersion: autoscaling.k8s.io/v1

kind: VerticalPodAutoscaler

metadata:

name: my-vpa

spec:

targetRef:

apiVersion: "apps/v1"

kind: Deployment

name: my-deployment

updatePolicy:

updateMode: "Auto"

**Node Affinity**

Affinity attracts pods to nodes.

Commands:

* kubectl label nodes node1 disktype=ssd: Label node.

Example YAML:

text

spec:

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: kubernetes.io/os

operator: In

values:

- linux

**Role Based Access Control (RBAC)**

RBAC controls access via roles.

Commands:

* kubectl create role pod-reader --verb=get --verb=list --verb=watch --resource=pods: Create role.
* kubectl create rolebinding pod-reader-binding --role=pod-reader --user=user1: Bind.

Example YAML:

text

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

namespace: default

name: pod-reader

rules:

- apiGroups: [""]

resources: ["pods"]

verbs: ["get", "watch", "list"]

---

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: read-pods

namespace: default

subjects:

- kind: User

name: jane

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: Role

name: pod-reader

apiGroup: rbac.authorization.k8s.io

**Monitoring and Logging Using Kubernetes Dashboard**

Dashboard is a web UI for monitoring.

Commands:

* kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/recommended.yaml: Install.
* kubectl proxy: Access at <http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/>.
* kubectl logs <pod> -n kubernetes-dashboard: Logs.

For logging, use tools like ELK; dashboard shows metrics if metrics-server installed.

**Custom Resource Definitions (CRDs)**

CRDs extend K8s API.

Commands:

* kubectl apply -f crd.yaml: Apply.
* kubectl get crds: List.

Example YAML:

text

apiVersion: apiextensions.k8s.io/v1

kind: CustomResourceDefinition

metadata:

name: crontabs.stable.example.com

spec:

group: stable.example.com

versions:

- name: v1

served: true

storage: true

scope: Namespaced

names:

plural: crontabs

singular: crontab

kind: CronTab

shortNames:

- ct

**Operators, Helm, Kubernetes API**

* **Operators**: Automate app management using CRDs.
* **Helm**: Package manager for K8s charts.
* **K8s API**: REST API for resources.

Commands for Helm:

* helm install my-release bitnami/mysql: Install chart.
* helm list: List releases.

For Operators, use operator-sdk to build.

**SideCar / Init Containers**

* **Sidecar**: Additional container in pod for support (e.g., logging).
* **Init**: Runs before main containers for setup.

Commands:

* kubectl apply -f pod.yaml: Apply.

Example YAML for Init:

text

spec:

initContainers:

- name: init-myservice

image: busybox:1.28

command: ['sh', '-c', "until nslookup myservice.$(cat /var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local; do echo waiting for myservice; sleep 2; done"]

Sidecar:

text

spec:

containers:

- name: main-app

image: main-image

- name: sidecar

image: sidecar-image

**Istio Service Mesh**

Istio is a service mesh for traffic management, security, observability.

Installation:

* Download istio-1.23.0, add bin/ to PATH.
* istioctl install --set profile=default -y: Install.
* kubectl label namespace default istio-injection=enabled: Enable injection.

Commands:

* istioctl dashboard kiali: Open dashboard.
* istioctl uninstall -y --purge: Uninstall.

**Project 1: Chat Application (3 Tier) - on Minikube**

3-tier: Frontend (React), Backend (Node.js), DB (MongoDB).

From the repo, Docker-based setup, K8s planned.

Steps:

* Start Minikube: minikube start.
* Build images: docker build -t frontend . (frontend dir).
* Load to Minikube: minikube image load frontend.
* Apply deployment and service YAML (similar to examples).
* Access with minikube service frontend-service.

Example Deployment for Backend:

text

apiVersion: apps/v1

kind: Deployment

metadata:

name: backend

spec:

replicas: 1

selector:

matchLabels:

app: backend

template:

spec:

containers:

- name: backend

image: backend-image

envFrom:

- secretRef:

name: db-credentials

**Project 2: Angular, Springboot, Three-tier App - on KIND**

3-tier: Frontend (Angular), Backend (Spring Boot), DB (Postgres or Mongo).

From the medium article, use Dockerfiles and K8s YAML.

Dockerfile for Spring Boot (Backend):

text

FROM openjdk:11

COPY target/contact-backend-app.jar /usr/app/

WORKDIR /usr/app/

ENTRYPOINT ["java","-jar","contact-backend-app.jar"]

EXPOSE 8080

Dockerfile for Angular (Frontend):

text

FROM nginx:latest

COPY /dist/contact-ui /usr/share/nginx/html/

EXPOSE 80

K8s YAML for Backend:

text

apiVersion: apps/v1

kind: Deployment

metadata:

name: contactbackendappdeployment

spec:

replicas: 2

selector:

matchLabels:

app: contactbackend

template:

metadata:

labels:

app: contactbackend

spec:

containers:

- name: contactbackendcontainer

image: jayadeep3/contact\_backend\_app

ports:

- containerPort: 8080

---

apiVersion: v1

kind: Service

metadata:

name: contactbackendsvc

spec:

type: NodePort

selector:

app: contactbackend

ports:

- port: 80

targetPort: 8080

nodePort: 30001

For Frontend similar.

On KIND:

* kind create cluster.
* kind load docker-image contact\_backend\_app.
* kubectl apply -f backend-deployment.yaml.
* Access via node port.

**DevOps Mega Project using EKS**

From the video, it's Kubernetes to Production with EKS, ArgoCD, GitOps.

Description: Deploy Wanderlust app on EKS using ArgoCD for GitOps.

Repo: <https://github.com/LondheShubham153/Wanderlust-Mega-Project>

Steps:

* Create EKS cluster: eksctl create cluster --name my-cluster --region us-east-1.
* Install ArgoCD: kubectl create namespace argocd; kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml.
* Set up Git repo with YAML.
* Use ArgoCD to sync apps.

Commands:

* argocd login <server>: Login.
* argocd app create my-app --repo <git> --path . --dest-server https://kubernetes.default.svc --dest-namespace default: Create app.
* argocd app sync my-app: Sync.### Introduction & Overview of K8s Concept

Kubernetes (K8s) is an open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts. It provides container-centric infrastructure, allowing developers to deploy applications consistently across environments. Key concepts include Pods (basic execution unit), Services (networking abstraction), Deployments (scaling and updates), and Namespaces (virtual clusters). K8s handles load balancing, storage orchestration, automated rollouts/rollbacks, self-healing, and secret/configuration management. It's designed for microservices but supports monolithic apps too. As of 2025, it supports advanced features like serverless (Knative) and AI workloads.

**All K8s Commands for Introduction/Setup**

* kubectl version: Check client and server versions.
* kubectl cluster-info: Get cluster details.
* kubectl get nodes: List nodes.
* kubectl api-resources: List available resources.
* kubectl config current-context: Current context.

**K8s History**

Kubernetes originated from Google's Borg system. Announced in 2014, v1.0 released in 2015, donated to CNCF. Key milestones: Federation in 2016, RBAC in 2017, CRDs in 2018, Windows support in 2019, Gateway API in 2022, and enhanced security in v1.30 (2025). It's now the standard for orchestration, with over 100 certified distributions.

**Why Should We Learn K8s**

K8s is crucial for DevOps as it enables scalable, resilient apps. Benefits: Portability across clouds, auto-scaling, zero-downtime deployments, efficient resource use. Demand is high—CNCF reports 7 million developers in 2024. It reduces ops overhead, integrates with CI/CD, and supports hybrid/multi-cloud.

**Monolithic vs Microservices**

* **Monolithic**: Single codebase, easy initial development but hard to scale, update, or maintain. Failures affect entire app.
* **Microservices**: Independent services, scalable individually, fault-isolated, but complex networking/logging. K8s excels in microservices with Services for discovery, HPA for scaling.

**K8s Architecture, Kubectl**

* **Cluster Architecture**: Control Plane (master) and Nodes.
  + ETCD: Key-value store for cluster state.
  + Kube-apiserver: Handles authentication, validation, ETCD updates.
  + Kube-controller-manager: Monitors and maintains desired state (e.g., replication, node controller checks every 5s, marks unreachable after 40s, replaces after 5min).
  + Node controller: Manages node health.
  + Replication controller: Ensures pod replicas.
  + Kube-scheduler: Assigns pods.
  + CRI: Interface for runtimes (containerd, CRI-O; Docker deprecated).
  + Kubelet: Node agent.
  + Kube-proxy: Network proxy.
* **Kubectl**: CLI for K8s API.

**All K8s Commands for Architecture**

* kubectl get componentstatuses: Control plane health (deprecated; use kubectl get cs).
* kubectl describe node <node>: Node details.
* etcdctl get / --prefix --keys-only: ETCD keys (requires etcdctl).

**Create K8s Architecture by Own**

Setup manual cluster with master/workers. Use VMs, install CRI, K8s binaries.

**AWS Setup for Handson**

Use EC2 for nodes or EKS for managed.

Commands:

* aws ec2 create-security-group --group-name k8s --description "K8s ports".
* Launch instances, install K8s.

**What is KIND Cluster & Installation of It**

KIND runs K8s in Docker for testing.

Installation:

* curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.23.0/kind-linux-amd64
* chmod +x ./kind
* mv ./kind /usr/local/bin/kind

**KIND Setup on Local**

Require Docker. Configure config.yml for nodes.

**KIND Cluster Creation**

Commands:

* kind create cluster --name tws-cluster --config config.yml
* kind get clusters: List.
* kind delete cluster --name tws-cluster

config.yml example:

text

kind: Cluster

apiVersion: kind.x-k8s.io/v1alpha4

nodes:

- role: control-plane

- role: worker

**What is Minikube Cluster, Installation, Cluster Creation**

Minikube: Local single-node cluster.

Installation:

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

Commands:

* minikube start --driver=docker
* minikube stop
* minikube delete

**Kubeadm, Setup & Installation, Cluster Creation**

Kubeadm bootstraps clusters.

Setup:

* Install containerd, kubelet, kubeadm, kubectl.
* Master: kubeadm init --pod-network-cidr=192.168.0.0/16
* Worker: kubeadm join <token>
* Install CNI: kubectl apply -f https://raw.githubusercontent.com/projectcalico/calico/v3.27.3/manifests/calico.yaml

Commands:

* kubeadm reset
* kubeadm token create --print-join-command

**Namespaces**

Commands:

* kubectl create namespace monitoring
* kubectl get namespace
* kubectl label namespace monitoring team=devops
* kubectl describe namespace monitoring

**Pods**

Commands:

* kubectl run nginx --image=nginx -n nginx
* kubectl describe pod nginx -n nginx
* kubectl logs nginx -n nginx
* kubectl exec -it nginx -n nginx -- bash

YAML example:

text

apiVersion: v1

kind: Pod

metadata:

name: nginx

spec:

containers:

- name: nginx

image: nginx

**Deployments**

Commands:

* kubectl apply -f deployment.yml
* kubectl scale deployment nginx-deployment --replicas=3 -n nginx
* kubectl rollout status deployment/nginx-deployment

YAML example:

text

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

spec:

containers:

- name: nginx

image: nginx

**ReplicaSet vs StatefulSet vs Deployments**

ReplicaSet: Basic replica management. Deployment: Manages ReplicaSets for updates. StatefulSet: For stateful, ordered pods.

**Labels and Selectors**

Commands:

* kubectl label pod nginx app=v1
* kubectl get pods -l app=v1

**Rolling Update in Deployment**

Commands:

* kubectl set image deployment/nginx-deployment nginx=nginx:1.16.1
* kubectl rollout undo deployment/nginx-deployment

**ReplicaSets**

Commands:

* kubectl get rs
* kubectl describe rs <name>

**DaemonSets**

Commands:

* kubectl apply -f daemonset.yaml
* kubectl get ds

YAML example:

text

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: fluentd

spec:

selector:

matchLabels:

app: fluentd

template:

spec:

containers:

- name: fluentd

image: fluentd

**Jobs**

Commands:

* kubectl apply -f job.yaml
* kubectl get jobs

YAML example:

text

apiVersion: batch/v1

kind: Job

metadata:

name: pi

spec:

template:

spec:

containers:

- name: pi

image: perl

command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]

restartPolicy: Never

**CronJobs**

Commands:

* kubectl apply -f cronjob.yaml
* kubectl get cronjobs

YAML example:

text

apiVersion: batch/v1

kind: CronJob

metadata:

name: hello

spec:

schedule: "\* \* \* \* \*"

jobTemplate:

spec:

template:

spec:

containers:

- name: hello

image: busybox

args:

- /bin/sh

- -c

- date; echo Hello

**Storage**

Abstracts storage for pods.

**Persistent Volume (PV)**

Commands:

* kubectl get pv
* kubectl apply -f pv.yaml

YAML example:

text

apiVersion: v1

kind: PersistentVolume

metadata:

name: pv-example

spec:

capacity:

storage: 5Gi

accessModes:

- ReadWriteOnce

hostPath:

path: "/data/pv"

**StorageClasses**

Commands:

* kubectl get sc

YAML example:

text

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: fast

provisioner: kubernetes.io/aws-ebs

parameters:

type: gp2

**Persistent Volume Claim (PVC)**

Commands:

* kubectl get pvc
* kubectl apply -f pvc.yaml

YAML example:

text

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: pvc-example

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 5Gi

storageClassName: fast

**Services**

Commands:

* kubectl apply -f service.yaml
* kubectl get svc

YAML example:

text

apiVersion: v1

kind: Service

metadata:

name: my-service

spec:

selector:

app: my-app

ports:

- protocol: TCP

port: 80

targetPort: 80

type: LoadBalancer

**Small Project (django-notes-app)**

Deploy Django app with Deployment, Service, PVC for DB.

YAML for Deployment:

text

apiVersion: apps/v1

kind: Deployment

metadata:

name: django-notes

spec:

replicas: 1

selector:

matchLabels:

app: django

template:

spec:

containers:

- name: django

image: django-notes-app

volumeMounts:

- mountPath: /data

name: data

volumes:

- name: data

persistentVolumeClaim:

claimName: django-pvc

Commands:

* kubectl apply -f django.yaml

**Ingress**

Commands:

* kubectl apply -f ingress.yaml
* kubectl get ingress

YAML example:

text

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: ingress-example

spec:

rules:

- host: example.com

http:

paths:

- path: /

pathType: Prefix

backend:

service:

name: my-service

port:

number: 80

**Annotations**

In Ingress YAML:

text

metadata:

annotations:

kubernetes.io/ingress.class: "nginx"

**StatefulSets**

Commands:

* kubectl apply -f statefulset.yaml
* kubectl get sts

YAML example:

text

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: mysql

spec:

selector:

matchLabels:

app: mysql

serviceName: "mysql"

replicas: 3

template:

spec:

containers:

- name: mysql

image: mysql

volumeMounts:

- name: data

mountPath: /var/lib/mysql

volumeClaimTemplates:

- metadata:

name: data

spec:

accessModes: [ "ReadWriteOnce" ]

resources:

requests:

storage: 10Gi

**ConfigMaps**

Commands:

* kubectl create configmap app-config --from-file=config.properties
* kubectl get cm

YAML example:

text

apiVersion: v1

kind: ConfigMap

metadata:

name: app-config

data:

config.properties: |

key=value

**Secrets**

Commands:

* kubectl create secret generic db-cred --from-literal=user=admin --from-literal=pass=secret
* kubectl get secrets

YAML example:

text

apiVersion: v1

kind: Secret

metadata:

name: db-cred

type: Opaque

data:

user: YWRtaW4=

pass: c2VjcmV0

**Resource Quotas and Limits**

Commands:

* kubectl apply -f quota.yaml
* kubectl describe quota

YAML example (Quota):

text

apiVersion: v1

kind: ResourceQuota

metadata:

name: quota

spec:

hard:

pods: "4"

requests.cpu: "1"

requests.memory: 1Gi

Limits in Pod:

text

spec:

containers:

- name: app

resources:

limits:

cpu: "500m"

memory: "128Mi"

requests:

cpu: "250m"

memory: "64Mi"

**Probes**

Commands:

* kubectl apply -f pod.yaml

YAML example:

text

spec:

containers:

- name: app

livenessProbe:

httpGet:

path: /healthz

port: 8080

initialDelaySeconds: 3

periodSeconds: 3

readinessProbe:

httpGet:

path: /ready

port: 8080

initialDelaySeconds: 3

periodSeconds: 3

**Taints/Tolerations**

Commands:

* kubectl taint nodes node1 key=value:NoSchedule
* kubectl taint nodes node1 key=value:NoSchedule-: Remove.

YAML example (Toleration):

text

spec:

tolerations:

- key: "key"

operator: "Exists"

effect: "NoSchedule"

**HPA**

Commands:

* kubectl autoscale deployment app --cpu-percent=50 --min=1 --max=10
* kubectl get hpa

YAML example:

text

apiVersion: autoscaling/v2

kind: HorizontalPodAutoscaler

metadata:

name: app-hpa

spec:

scaleTargetRef:

apiVersion: apps/v1

kind: Deployment

name: app

minReplicas: 1

maxReplicas: 10

metrics:

- type: Resource

resource:

name: cpu

target:

type: Utilization

averageUtilization: 50

**VPA**

Commands:

* kubectl apply -f vpa.yaml

YAML example:

text

apiVersion: autoscaling.k8s.io/v1

kind: VerticalPodAutoscaler

metadata:

name: app-vpa

spec:

targetRef:

apiVersion: "apps/v1"

kind: Deployment

name: app

updatePolicy:

updateMode: "Auto"

**Node Affinity**

Commands:

* kubectl label nodes node1 zone=east

YAML example:

text

spec:

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: zone

operator: In

values:

- east

**Role Based Access Control (RBAC)**

Commands:

* kubectl create role pod-reader --verb=get,list --resource=pods
* kubectl create rolebinding reader --role=pod-reader --user=jane

YAML example:

text

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

namespace: default

name: pod-reader

rules:

- apiGroups: [""]

resources: ["pods"]

verbs: ["get", "list"]

---

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: read-pods

namespace: default

subjects:

- kind: User

name: jane

roleRef:

kind: Role

name: pod-reader

**Monitoring and Logging Using Kubernetes Dashboard**

Commands:

* kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/recommended.yaml
* kubectl proxy (access at <http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/>)
* kubectl port-forward svc/kubernetes-dashboard -n kubernetes-dashboard 8443:443
* kubectl logs -n kubernetes-dashboard <pod-name>

For logging, integrate with ELK or Fluentd.

**Custom Resource Definitions (CRDs)**

Commands:

* kubectl apply -f crd.yaml
* kubectl get crd

YAML example:

text

apiVersion: apiextensions.k8s.io/v1

kind: CustomResourceDefinition

metadata:

name: crontabs.stable.example.com

spec:

group: stable.example.com

versions:

- name: v1

served: true

storage: true

scope: Namespaced

names:

plural: crontabs

kind: CronTab

**Operators, Helm, Kubernetes API**

* **Operators**: Use CRDs for app management.
* **Helm**: helm install my-mysql bitnami/mysql
* helm list
* **API**: Use kubectl proxy to access /api.

Helm chart example (values.yaml):

text

replicaCount: 1

image:

repository: nginx

**SideCar / Init Containers**

Commands:

* kubectl apply -f pod.yaml

YAML example (Init):

text

spec:

initContainers:

- name: init

image: busybox

command: ['sh', '-c', 'echo Init done']

containers:

- name: main

image: nginx

Sidecar:

text

spec:

containers:

- name: main

image: nginx

- name: sidecar

image: fluentd

**Istio Service Mesh**

Installation:

* curl -L https://istio.io/downloadIstio | sh -
* cd istio-1.23.0
* istioctl install --set profile=default -y

Commands:

* istioctl analyze
* kubectl label namespace default istio-injection=enabled

Project 1: Chat Application (3 Tier) - on Minikube

Frontend (React) Deployment and Service

apiVersion: apps/v1

kind: Deployment

metadata:

  name: chat-frontend

spec:

  replicas: 2

  selector:

    matchLabels:

      app: chat-frontend

  template:

    metadata:

      labels:

        app: chat-frontend

    spec:

      containers:

      - name: chat-frontend

        image: chat-frontend-image:latest

        ports:

        - containerPort: 3000

---

apiVersion: v1

kind: Service

metadata:

  name: chat-frontend-service

spec:

  selector:

    app: chat-frontend

  ports:

    - protocol: TCP

      port: 80

      targetPort: 3000

  type: LoadBalancer

Backend (Node.js) Deployment and Service

apiVersion: apps/v1

kind: Deployment

metadata:

  name: chat-backend

spec:

  replicas: 2

  selector:

    matchLabels:

      app: chat-backend

  template:

    metadata:

      labels:

        app: chat-backend

    spec:

      containers:

      - name: chat-backend

        image: chat-backend-image:latest

        ports:

        - containerPort: 4000

        env:

        - name: MONGO\_URI

          value: "mongodb://mongodb-service:27017/chatdb"

---

apiVersion: v1

kind: Service

metadata:

  name: chat-backend-service

spec:

  selector:

    app: chat-backend

  ports:

    - protocol: TCP

      port: 80

      targetPort: 4000

  type: ClusterIP

MongoDB Deployment and Service

apiVersion: apps/v1

kind: Deployment

metadata:

  name: mongodb

spec:

  replicas: 1

  selector:

    matchLabels:

      app: mongodb

  template:

    metadata:

      labels:

        app: mongodb

    spec:

      containers:

      - name: mongodb

        image: mongo:latest

        ports:

        - containerPort: 27017

        volumeMounts:

        - name: mongo-data

          mountPath: /data/db

      volumes:

      - name: mongo-data

        emptyDir: {}

---

apiVersion: v1

kind: Service

metadata:

  name: mongodb-service

spec:

  selector:

    app: mongodb

  ports:

    - protocol: TCP

      port: 27017

      targetPort: 27017

  type: ClusterIP

**Usage Instructions**

* Build Docker images for frontend, backend, and MongoDB locally (e.g., docker build -t chat-frontend-image:latest . in frontend directory).
* Load images into Minikube: minikube image load chat-frontend-image:latest, etc.
* Apply YAML files: kubectl apply -f frontend-deployment.yaml, kubectl apply -f backend-deployment.yaml, kubectl apply -f mongodb-deployment.yaml.
* Access the frontend via minikube service chat-frontend-service --url.

Project 2: Angular, Springboot, Three-tier App - on KIND

Frontend (Angular) Deployment and Service

apiVersion: apps/v1

kind: Deployment

metadata:

  name: angular-frontend

spec:

  replicas: 2

  selector:

    matchLabels:

      app: angular-frontend

  template:

    metadata:

      labels:

        app: angular-frontend

    spec:

      containers:

      - name: angular-frontend

        image: angular-frontend-image:latest

        ports:

        - containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

  name: angular-frontend-service

spec:

  selector:

    app: angular-frontend

  ports:

    - protocol: TCP

      port: 80

      targetPort: 80

  type: LoadBalancer

Backend (Spring Boot) Deployment and Service

apiVersion: apps/v1

kind: Deployment

metadata:

  name: springboot-backend

spec:

  replicas: 2

  selector:

    matchLabels:

      app: springboot-backend

  template:

    metadata:

      labels:

        app: springboot-backend

    spec:

      containers:

      - name: springboot-backend

        image: springboot-backend-image:latest

        ports:

        - containerPort: 8080

        env:

        - name: SPRING\_DATASOURCE\_URL

          value: "jdbc:postgresql://postgres-service:5432/mydb"

        - name: SPRING\_DATASOURCE\_USERNAME

          value: "postgres"

        - name: SPRING\_DATASOURCE\_PASSWORD

          value: "password"

---

apiVersion: v1

kind: Service

metadata:

  name: springboot-backend-service

spec:

  selector:

    app: springboot-backend

  ports:

    - protocol: TCP

      port: 80

      targetPort: 8080

  type: ClusterIP

PostgreSQL Deployment and Service with PVC

apiVersion: apps/v1

kind: Deployment

metadata:

  name: postgres

spec:

  replicas: 1

  selector:

    matchLabels:

      app: postgres

  template:

    metadata:

      labels:

        app: postgres

    spec:

      containers:

      - name: postgres

        image: postgres:latest

        ports:

        - containerPort: 5432

        env:

        - name: POSTGRES\_DB

          value: "mydb"

        - name: POSTGRES\_USER

          value: "postgres"

        - name: POSTGRES\_PASSWORD

          value: "password"

        volumeMounts:

        - name: postgres-data

          mountPath: /var/lib/postgresql/data

      volumes:

      - name: postgres-data

        persistentVolumeClaim:

          claimName: postgres-pvc

---

apiVersion: v1

kind: Service

metadata:

  name: postgres-service

spec:

  selector:

    app: postgres

  ports:

    - protocol: TCP

      port: 5432

      targetPort: 5432

  type: ClusterIP

---

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

  name: postgres-pvc

spec:

  accessModes:

    - ReadWriteOnce

  resources:

    requests:

      storage: 1Gi

**Usage Instructions**

* Build Docker images for Angular and Spring Boot using their respective Dockerfiles (e.g., docker build -t angular-frontend-image:latest . in Angular directory).
* Load images into KIND: kind load docker-image angular-frontend-image:latest, etc.
* Create a KIND cluster: kind create cluster --name three-tier.
* Apply YAML files: kubectl apply -f angular-deployment.yaml, kubectl apply -f springboot-deployment.yaml, kubectl apply -f postgres-deployment.yaml.
* Access the frontend via kubectl port-forward svc/angular-frontend-service 80:80 or use KIND's node port.

**1. Docker Container Exits Immediately, how will you troubleshoot?**

When a Docker container exits immediately after starting, it typically indicates that the main process inside the container has terminated. Here’s how I would troubleshoot this step-by-step:

* **Check Container Logs**: First, I’d use docker logs <container\_id> or docker logs <container\_name> to inspect the output for error messages or reasons for the exit (e.g., application crash, missing dependencies, or configuration issues).
* **Verify the Entry Point/Command**: I’d review the Dockerfile or docker run command to ensure the entry point or command is correctly defined and points to a long-running process (e.g., a web server like Nginx or a script that doesn’t exit immediately).
* **Run in Interactive Mode**: I’d start the container with docker run -it <image\_name> /bin/bash (or /bin/sh if bash isn’t available) to access it interactively and manually run the entry point command to observe the behavior.
* **Check Exit Code**: Using docker ps -a to list all containers and docker inspect <container\_id> to check the State.ExitCode, I’d determine if it’s a specific error (e.g., 0 for success, 1 for general error).
* **Resource Issues**: I’d ensure the container has sufficient CPU/memory limits and that the host system isn’t resource-constrained.
* **Common Fixes**: If it’s a missing dependency or configuration, I’d update the Dockerfile to install it (e.g., apt-get update && apt-get install -y <package>) and rebuild the image.

This systematic approach helps identify whether it’s a code issue, misconfiguration, or environment problem.

**2. [90 Percent get this wrong] What is the purpose of EXPOSE in Dockerfile?**

The EXPOSE instruction in a Dockerfile is often misunderstood. Its primary purpose is to **document** the ports that the container intends to use for communication. It acts as a hint to the user or other tools (e.g., orchestration systems like Kubernetes) about which ports the application inside the container listens on. However, it does **not** actually publish or open those ports to the host or external network.

* **Key Points**:
  + To make a port accessible, you must use -p or --publish with docker run (e.g., docker run -p 8080:80 <image\_name>).
  + EXPOSE is metadata and can be overridden or ignored at runtime.
  + It’s useful for documentation and can be used by Docker Compose or Kubernetes to configure networking automatically.

For example:

language-Dockerfile

EXPOSE 80

This tells others the container expects to handle HTTP traffic on port 80, but without -p 80:80, the port remains internal to the container. The misconception that EXPOSE automatically opens ports is why many get this wrong.

**3. Port is Not Accessible on localhost even after Port mapping in Docker**

If a port isn’t accessible on localhost despite port mapping, I’d troubleshoot as follows:

* **Verify Port Mapping**: Check the mapping with docker ps to ensure the correct host:container port pair (e.g., -p 8080:80) is active. If incorrect, I’d stop and restart with the right mapping.
* **Check Container Status**: Use docker ps -a to confirm the container is running. If it’s exited, I’d investigate logs (docker logs <container\_id>) for the root cause.
* **Application Listening**: Inside the container, I’d use docker exec -it <container\_id> netstat -tuln or ss -tuln to verify the application is listening on the mapped port (e.g., 80). If not, it might be a misconfiguration in the app.
* **Firewall/Security Groups**: On the host, I’d check if a firewall (e.g., ufw, iptables) or cloud security group is blocking the port. I’d allow it with ufw allow 8080 or adjust the security group.
* **Docker Network**: Ensure the container is on the correct network (e.g., bridge by default). I might use docker network ls and docker network inspect <network> to verify, or try --network host to bypass Docker networking.
* **Host Binding**: If the app binds to 127.0.0.1 inside the container, it won’t be accessible externally. I’d configure it to bind to 0.0.0.0.

Testing with curl localhost:8080 or a browser would confirm resolution after these steps.

**4. Data lost when container stops and restarts, How will you fix it**

Data loss on container restart occurs because containers are stateless by default, and their filesystem is ephemeral unless persisted. Here’s how I’d fix it:

* **Use Volumes**: I’d mount a Docker volume with -v <volume\_name>:/app/data or a host path with -v /host/path:/container/path in docker run. Volumes persist data independently of the container lifecycle.
* **Bind Mounts**: For specific cases, I’d use a bind mount to a host directory (e.g., -v /data:/app/data), ensuring the host path exists and has proper permissions.
* **Update Dockerfile**: I’d avoid storing data in the container’s writable layer by designing the app to use a volume or external storage.
* **Verify Persistence**: After restarting the container (docker stop <container\_id> && docker start <container\_id>), I’d check data integrity inside the container with docker exec.

For example:

bash

docker run -d -v mydata:/app/data myimage

This ensures data persists across restarts. For production, I’d consider managed storage solutions like AWS EBS or Kubernetes PVs if using an orchestrator.

**5. You made change in your code, rebuilt the image, but the change isn't reflected**

If changes aren’t reflected after rebuilding, it’s likely a caching or build issue. Here’s my approach:

* **Clear Cache**: I’d rebuild with --no-cache to force a fresh build: docker build --no-cache -t myimage:latest ..
* **Check Tag**: I’d ensure I’m using the correct tag (e.g., latest) and not an old image. I’d run docker images to verify the image ID and docker rmi <old\_image\_id> to remove outdated images.
* **Layer Caching**: If the Dockerfile uses cached layers (e.g., COPY after RUN), I’d reorder commands to minimize caching impact or add a unique identifier (e.g., ARG CACHEBUST=$(date +%s)).
* **Run New Image**: I’d stop the old container (docker stop <container\_id>) and run a new one with docker run -d myimage:latest to ensure the updated image is used.
* **Inspect Container**: I’d use docker exec to confirm the changes inside the container.

This ensures the latest code is built and deployed.

**6. App Crashes with Permission Denied in Container but works fine on localhost**

A "Permission Denied" crash in a container, despite working locally, suggests a user or file permission mismatch. Here’s how I’d troubleshoot:

* **Check User Context**: I’d inspect the Dockerfile for a USER instruction. If it runs as a non-root user (e.g., USER appuser), the app might lack permissions. I’d either adjust the user with USER root temporarily or fix file ownership.
* **File Permissions**: Inside the container, I’d use docker exec -it <container\_id> ls -l to check file permissions. If needed, I’d update the Dockerfile with RUN chown appuser:appuser /app or copy files with the correct owner.
* **Volume Mounts**: If using a volume (-v), I’d ensure the host directory permissions match the container user (e.g., chmod 755 /host/path).
* **SELinux/AppArmor**: On the host, I’d check if SELinux or AppArmor is enforcing restrictions. I’d disable it temporarily (setenforce 0) to test, then configure proper policies.
* **Logs**: I’d review docker logs for specific error details to pinpoint the denied resource.

Fixing ownership or adjusting the user context in the Dockerfile usually resolves this.

**7. Docker host is running out of disk space. How do you clean up**

When a Docker host runs out of disk space, it’s often due to unused images, containers, or volumes. Here’s how I’d clean up:

* **Remove Stopped Containers**: docker container prune deletes all stopped containers after confirmation.
* **Delete Unused Images**: docker image prune -a removes dangling and unused images (use with caution, requires -f to force).
* **Clean Up Volumes**: docker volume prune removes unused volumes.
* **Check Disk Usage**: Use docker system df to see space usage and df -h on the host to confirm.
* **Manual Cleanup**: If needed, I’d identify large images with docker images and remove specific ones with docker rmi <image\_id>.
* **Set Limits**: For prevention, I’d configure storage driver options or use a cleanup script.

This approach reclaims space while minimizing disruption.

**8. How will you Debug a Live Container**

To debug a live container, I’d follow these steps:

* **Access Container**: Use docker exec -it <container\_id> /bin/bash (or /bin/sh) to get an interactive shell. If no shell is available, I’d use nsenter or attach with docker attach.
* **Check Logs**: Run docker logs <container\_id> -f to stream logs and identify issues in real-time.
* **Inspect Process**: Use ps aux inside the container to see running processes and resource usage.
* **Network Debugging**: Check network with netstat -tuln or curl to test connectivity.
* **Resource Monitoring**: Use top or docker stats <container\_id> to monitor CPU/memory.
* **Add Debugging Tools**: If needed, I’d modify the Dockerfile to include tools (e.g., apt-get install -y net-tools) and restart the container.
* **Sidecar Approach**: For production, I might deploy a debug sidecar container sharing the network namespace.

This allows real-time diagnosis without downtime.

**9. Which container registry do you use in your organization**

In my current/previous organization, we primarily use **Docker Hub** for public images and a private **Amazon Elastic Container Registry (ECR)** for proprietary applications. We chose ECR for its integration with AWS ecosystems, secure access control via IAM, and automatic cleanup of old images. However, the specific registry depends on the organization’s infrastructure—alternatives like Google Container Registry (GCR), Azure Container Registry (ACR), or a self-hosted Harbor could be used based on needs for security, cost, or compliance.

**10. Explain the difference between CMD and Entrypoint in Docker**

Both CMD and ENTRYPOINT define the default executable in a container, but they differ in behavior:

* **CMD**:
  + Specifies the default command or parameters that can be overridden at runtime with docker run arguments.
  + Can be defined as a string (e.g., CMD "nginx") or an exec array (e.g., CMD ["nginx", "-g", "daemon off;"]).
  + If multiple CMD instructions exist, the last one takes effect.
  + Example: CMD ["python", "app.py"] can be overridden with docker run myimage bash.
* **ENTRYPOINT**:
  + Sets a fixed executable that cannot be overridden unless --entrypoint is used.
  + Defined as an exec array (e.g., ENTRYPOINT ["nginx", "-g", "daemon off;"]).
  + CMD can provide default arguments to ENTRYPOINT if CMD is in array form.
  + Example: ENTRYPOINT ["python"] with CMD ["app.py"] runs python app.py, but docker run myimage -m "new.py" uses python -m new.py.
* **Key Difference**: ENTRYPOINT is the unchangeable base command, while CMD is flexible and defaults to arguments. Combining them (e.g., ENTRYPOINT as executable, CMD as args) is common for extensible images.

**11. What docker commands you use on day to day basis**

On a day-to-day basis, I use the following Docker commands frequently:

* docker ps -a: List all containers to check status.
* docker run: Start a new container (e.g., docker run -d -p 80:80 nginx).
* docker stop and docker start: Manage container lifecycle.
* docker logs: View container output for debugging.
* docker exec -it: Access a running container interactively.
* docker build: Build images from Dockerfile (e.g., docker build -t myimage .).
* docker images: List images to manage storage.
* docker rm and docker rmi: Remove containers and images.
* docker network ls and docker network inspect: Manage networking.
* docker system prune: Clean up unused resources.

These cover deployment, monitoring, and maintenance tasks.

**12. When will you forcefully remove a container and how**

I’d forcefully remove a container in these scenarios:

* **Stuck Container**: If a container is in a dead or unresponsive state and docker stop fails.
* **Resource Lock**: When a container holds resources (e.g., ports, volumes) needed urgently, and graceful shutdown isn’t possible.
* **Corruption**: If the container’s filesystem is corrupted, requiring a fresh start.

**How to Forcefully Remove**:

* Use docker kill <container\_id> to stop it immediately, then docker rm -f <container\_id> to remove it. The -f flag forces removal even if it’s running.
* Example: docker rm -f mycontainer.
* **Precaution**: I’d first check with docker ps -a to confirm the container and back up any critical data (e.g., via volumes) before proceeding.

This ensures minimal disruption while resolving the issue.

**1. Explain Kubernetes Cluster Architecture**

Kubernetes cluster architecture consists of a **Control Plane** and **Worker Nodes**, working together to manage and orchestrate containerized applications.

* **Control Plane Components**:
  + **kube-apiserver**: The front-end that exposes the Kubernetes API, handling all communication and authentication.
  + **etcd**: A distributed key-value store that holds the cluster’s configuration and state data.
  + **kube-scheduler**: Assigns pods to nodes based on resource availability, policies, and constraints.
  + **kube-controller-manager**: Runs controllers (e.g., Node Controller, Replication Controller) to maintain the desired state.
  + **cloud-controller-manager** (optional): Integrates with cloud providers for load balancers or storage.
* **Worker Node Components**:
  + **kubelet**: An agent that ensures containers in pods are running and healthy, communicating with the Control Plane.
  + **kube-proxy**: Manages network rules (e.g., iptables) to enable communication between pods and services.
  + **Container Runtime**: Software (e.g., Docker, containerd) that runs containers.

The Control Plane manages the cluster state, while Worker Nodes execute the workloads, with components interacting via the API server to maintain scalability and resilience.

**2. How various components of Kubernetes interact when you run kubectl apply (Pod)**

When you run kubectl apply -f pod.yaml, the following interaction occurs:

* **kubectl**: Sends the pod configuration (in YAML) to the **kube-apiserver** via REST API calls.
* **kube-apiserver**: Validates the request, authenticates the user, and updates the desired state in **etcd**.
* **kube-scheduler**: Detects the new pod in etcd, evaluates node resources (CPU, memory) and constraints (e.g., taints, affinity), and assigns it to a suitable node.
* **kubelet** (on the assigned node): Receives the pod specification from the API server, interacts with the **container runtime** (e.g., Docker) to pull the image, create containers, and start the pod.
* **kube-proxy**: Updates network rules to ensure the pod can communicate with other pods or services.

This process ensures the cluster converges to the desired state, with each component playing a specialized role.

**3. What is the purpose of services in Kubernetes**

Services in Kubernetes provide a stable networking abstraction to enable communication with a set of pods, regardless of their lifecycle (e.g., restarts, scaling). Their purposes include:

* **Load Balancing**: Distributes traffic across multiple pod instances using a virtual IP (ClusterIP).
* **Service Discovery**: Assigns a DNS name and IP to access pods, abstracting their dynamic IPs.
* **Decoupling**: Allows applications to communicate without hardcoding pod IPs, supporting rolling updates and scaling.
* **External Access**: Types like NodePort or LoadBalancer expose services outside the cluster.

For example, a service with selector: app=web ensures traffic to web-service reaches all pods labeled app=web.

**4. Why is hardcoding Pod IP communication a bad practice**

Hardcoding pod IP addresses for communication is problematic because:

* **Dynamic IPs**: Pods get new IPs on restart or rescheduling, breaking hardcoded references.
* **Scalability**: Manual IP management doesn’t scale with auto-scaling or rolling updates.
* **Resilience**: If a pod fails, its IP becomes invalid, disrupting communication.
* **Maintenance**: Updates require code changes, increasing operational overhead.

Instead, using services or DNS names (e.g., my-service.namespace.svc.cluster.local) ensures stable, automated communication.

**5. What are the types of services in Kubernetes**

Kubernetes supports the following service types:

* **ClusterIP** (default): Exposes the service on a cluster-internal IP, accessible only within the cluster.
* **NodePort**: Exposes the service on each node’s IP at a specific port (e.g., 30000-32767), allowing external access.
* **LoadBalancer**: Provisions a cloud provider’s load balancer to expose the service externally, routing traffic to NodePort.
* **ExternalName**: Maps the service to an external DNS name (e.g., my.database.example.com) without proxies, useful for external dependencies.

Each type serves different use cases, from internal communication to public exposure.

**6. What are labels and selectors in Kubernetes**

* **Labels**: Key-value pairs attached to objects (e.g., pods, nodes) for identification and organization. Example: app=web, env=prod.
* **Selectors**: Queries that match labels to filter or target objects. Two types:
  + **Equality-based**: app=web matches pods with app=web.
  + **Set-based**: env in (prod, staging) matches pods with env=prod or env=staging.

Labels and selectors enable flexible grouping (e.g., for services, deployments) and are fundamental to Kubernetes’ declarative model.

**7. What would you recommend: NodePort service or Load balancer type service and why**

I’d recommend a **LoadBalancer** service over NodePort for production environments because:

* **Scalability**: LoadBalancer integrates with cloud providers (e.g., AWS ELB, GCP LB) for advanced load balancing and auto-scaling, while NodePort relies on node-level ports, which can become a bottleneck.
* **Security**: LoadBalancer can use SSL termination and WAF, offering better external access control than NodePort’s direct node exposure.
* **Ease of Use**: LoadBalancer provides a single external IP, simplifying client access, whereas NodePort requires knowing node IPs and ports.
* **Use Case**: NodePort is suitable for development or small clusters without cloud integration.

However, for a local Minikube setup or cost-sensitive scenarios, NodePort might be a practical choice.

**8. How Kubernetes Services are related to Kube Proxy**

Kubernetes Services rely on **kube-proxy**, a daemon on each node, to manage network traffic:

* **IP Tables Mode**: Kube-proxy configures iptables rules to route traffic from the service’s ClusterIP to the pods’ IPs based on selectors.
* **IPVS Mode**: Uses a more efficient virtual server to balance traffic, an alternative to iptables.
* **Userspace Mode** (deprecated): Historically proxied traffic in userspace.
* **Function**: When a service is created, kube-proxy ensures pods are reachable via the service IP, handling load balancing and session persistence.

This abstraction allows services to remain stable despite pod changes.

**9. What is the disadvantage of LoadBalancer service type**

The main disadvantages of a LoadBalancer service include:

* **Cost**: Cloud providers charge for load balancer instances, making it expensive for small setups or multiple services.
* **Dependency**: Relies on cloud provider support, limiting portability to on-premises or hybrid environments.
* **Configuration Overhead**: Requires additional setup (e.g., annotations for provider-specific options), increasing complexity.
* **Single Point**: A misconfigured load balancer can affect all traffic, necessitating robust monitoring.

For non-cloud or cost-sensitive cases, alternatives like Ingress or NodePort might be preferred.

**10. What is a Headless Service in Kubernetes and when did you use it**

A **Headless Service** is a service with .spec.clusterIP set to None, which disables the ClusterIP and allows direct access to pod IPs via DNS.

* **Behavior**: Instead of load balancing, it returns multiple A records (one per pod) or SRV records for stateful applications.
* **Use Case**: I’ve used it with **StatefulSets** (e.g., databases like MySQL or Cassandra) where each pod needs a stable, unique identity and direct communication. For example, in a Cassandra cluster, I configured a Headless Service to enable peer discovery among nodes.

This is ideal for applications requiring pod-level control rather than load-balanced access.

**11. Can a Pod access Service in different namespace, if Yes, how**

Yes, a pod can access a service in a different namespace by using the fully qualified domain name (FQDN).

* **How**: The service is addressed as <service-name>.<namespace>.svc.cluster.local. For example, if a service db-service is in db-namespace, a pod in app-namespace can reach it at db-service.db-namespace.svc.cluster.local.
* **Mechanism**: Kubernetes DNS (CoreDNS) resolves this across namespaces, provided network policies allow it.
* **Example**: I’ve used this to connect a frontend pod to a backend service in a separate namespace for isolation.

Ensure proper network policies or CIDR configurations if restrictions are in place.

**12. Explain how you can restrict access to db pod to only one app in the namespace**

To restrict access to a db pod to only one app in the namespace, I’d use **Network Policies**:

* **Create a Network Policy**: Define a policy that allows traffic only from the app’s pod label.

yaml

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: db-access-policy

namespace: app-namespace

spec:

podSelector:

matchLabels:

app: db

policyTypes:

- Ingress

ingress:

- from:

- podSelector:

matchLabels:

app: myapp

ports:

- protocol: TCP

port: 5432

* **Label Pods**: Ensure the db pod has app: db and the app pod has app: myapp.
* **Apply Policy**: Use kubectl apply -f policy.yaml.
* **Effect**: This restricts ingress traffic to the db pod to only pods labeled app: myapp, leveraging the CNI plugin (e.g., Calico) to enforce it.

This provides fine-grained security without modifying application code.

**13. Explain the deployment strategy that you follow in your organization**

In my organization, we follow a **Rolling Update** strategy for deployments:

* **Process**: We increment the image tag in the Deployment YAML (e.g., from v1 to v2), and Kubernetes gradually replaces old pods with new ones, maintaining availability.
* **Configuration**: We set maxSurge: 25% and maxUnavailable: 0% to control the rollout pace and ensure no downtime.
* **Validation**: We use readiness probes to ensure new pods are healthy before traffic is routed.
* **Rollback**: If issues arise, we revert to the previous version using kubectl rollout undo.

This balances stability and zero-downtime updates, tailored to our e-commerce platform’s needs.

**14. Explain the rollback strategy that you follow in your organization**

Our rollback strategy leverages Kubernetes’ built-in rollout history:

* **Monitoring**: We monitor application metrics (e.g., latency, error rate) post-deployment using Prometheus and Grafana.
* **Trigger**: If issues are detected (e.g., >5% error rate), we initiate a rollback within 5 minutes.
* **Command**: We use kubectl rollout undo deployment/<deployment-name> to revert to the previous revision, stored in the Deployment’s history.
* **Validation**: Post-rollback, we verify stability with smoke tests and logs (kubectl logs).
* **Automation**: We’ve integrated this into a CI/CD pipeline with ArgoCD, automating rollbacks on failed health checks.

This ensures rapid recovery with minimal manual intervention.

**15. Design a solution to avoid rollbacks**

To minimize rollbacks, I’d design a robust deployment pipeline:

* **Canary Deployment**: Roll out changes to a small subset (e.g., 10%) of pods, monitored with custom metrics (e.g., user success rate). If successful, scale to 100%.
* **Pre-Deployment Testing**: Use staging environments with production-like data and chaos engineering (e.g., Netflix’s Chaos Monkey) to simulate failures.
* **Blue-Green Deployment**: Maintain two identical environments (blue and green); switch traffic only after green passes all tests.
* **Automated Validation**: Implement liveness/readiness probes and integration tests in the pipeline.
* **Feature Flags**: Deploy code with flags to enable features post-validation, avoiding immediate impact.
* **Monitoring**: Use real-time alerts (e.g., Datadog) to catch issues early.

This reduces risks, making rollbacks rare.

**16. Explain the deployment strategies that you used in the past**

I’ve used the following strategies in past projects:

* **Rolling Update**: Gradually updated pods in a web app, ensuring 100% uptime with maxUnavailable: 0%, used for a customer-facing portal.
* **Canary Deployment**: Tested new features on 5% of traffic for a mobile app, scaling up after user feedback, using Istio for traffic splitting.
* **Blue-Green Deployment**: Switched environments for a payment system, validating green with load tests before cutting over.
* **Recreate**: Replaced all pods at once for a batch job, acceptable due to no downtime requirement.

The choice depended on the application’s criticality and user impact.

**17. Explain the role of CoreDNS in Kubernetes**

CoreDNS is the default DNS server in Kubernetes, providing service discovery and name resolution:

* **Function**: Resolves service names (e.g., my-service.namespace.svc.cluster.local) to ClusterIPs, enabling pod-to-service communication.
* **Integration**: Runs as a Deployment in the kube-system namespace, managed by the kube-dns Service.
* **Features**: Supports custom records, federation, and health checks, ensuring high availability.
* **Use Case**: I’ve relied on it to connect microservices across namespaces without hardcoding IPs.

It’s critical for the cluster’s networking layer.

**18. A DevOps engineer tainted a node as Noschedule. Can you still schedule a pod**

No, a pod cannot be scheduled on a node tainted with NoSchedule unless it has a matching toleration.

* **Taint**: kubectl taint nodes <node-name> key=value:NoSchedule marks the node as unschedulable.
* **Toleration**: You can allow scheduling by adding a toleration to the pod spec:

yaml

tolerations:

- key: "key"

operator: "Equal"

value: "value"

effect: "NoSchedule"

* **Workaround**: If intentional, I’d respect the taint; otherwise, I’d coordinate with the DevOps engineer to remove it with kubectl taint nodes <node-name> key:NoSchedule-.

This enforces node isolation for maintenance or special workloads.

**19. Pod is Stuck in CrashLoopBackOff, What steps will you take**

A CrashLoopBackOff indicates a pod keeps crashing and restarting. Here’s my troubleshooting:

* **Check Logs**: kubectl logs <pod-name> to identify errors (e.g., missing files, config issues).
* **Describe Pod**: kubectl describe pod <pod-name> to see events and exit codes.
* **Exec Inside**: If running, kubectl exec -it <pod-name> -- /bin/sh to debug manually.
* **Image Issues**: Verify the container image with docker pull <image> on the node and check tags.
* **Resource Limits**: Ensure CPU/memory requests/limits in the pod spec aren’t exceeded (kubectl get pod -o yaml).
* **Probes**: Check liveness/readiness probes for misconfiguration causing restarts.
* **Fix**: Update the Deployment with corrected config and rollout (kubectl rollout restart deployment/<name>).

This isolates whether it’s a code, config, or resource problem.

**20. What is the difference between liveness and readiness probes**

* **Liveness Probe**: Checks if a pod is alive; if it fails, Kubernetes restarts the pod. Example: HTTP GET to /health every 10 seconds.
  + Use Case: Detects deadlocks or crashes.
* **Readiness Probe**: Checks if a pod is ready to serve traffic; if it fails, the pod is removed from service endpoints. Example: TCP socket check every 5 seconds.
  + Use Case: Ensures a pod is initialized (e.g., database connection established).

Both are defined in the pod spec (e.g., livenessProbe: httpGet: path: /health), but liveness triggers restarts, while readiness affects traffic routing.

**21. Explain the difference between Ingress and LB service type**

* **Ingress**: A Kubernetes resource that manages external HTTP/HTTPS traffic routing to services, typically using an Ingress Controller (e.g., NGINX, Traefik). It supports host-based routing, path-based routing, and TLS termination.
* **LoadBalancer**: A service type that provisions a cloud load balancer to expose a service externally, routing traffic to NodePorts without advanced routing logic.
* **Difference**: Ingress is more flexible for complex routing (e.g., multiple services under one IP), while LoadBalancer is simpler but limited to one service per instance. Ingress requires a controller; LoadBalancer relies on cloud provider integration.

**22. Your app works with ClusterIP but fails with Ingress. How do you troubleshoot it**

If an app works with ClusterIP but fails with Ingress, I’d troubleshoot:

* **Ingress Controller**: Verify the controller (e.g., NGINX) is running with kubectl get pods -n ingress-nginx.
* **Ingress Resource**: Check the Ingress YAML (kubectl get ingress) for correct annotations, hosts, and paths (e.g., nginx.ingress.kubernetes.io/rewrite-target).
* **Service Selector**: Ensure the Ingress targets the correct service and pod labels match.
* **Logs**: Review controller logs (kubectl logs -n ingress-nginx) for errors (e.g., 404, 503).
* **Network**: Test connectivity with curl <ingress-ip>/<path> and check firewall rules.
* **TLS**: If using HTTPS, validate certificates and annotations (e.g., kubernetes.io/tls-acme).

This isolates configuration or networking issues.

**23. Why do I need to setup Ingress controller after creating ingress**

An Ingress resource defines routing rules but doesn’t handle traffic itself; it requires an **Ingress Controller** to implement those rules. The controller (e.g., NGINX, HAProxy) runs as a pod, watches the API server for Ingress resources, and configures itself to route traffic accordingly. Without it, the Ingress is inactive, as Kubernetes doesn’t include a default controller. Setup involves deploying the controller (e.g., via Helm) and ensuring it’s in the same namespace or accessible.

**24. We have an inhouse load balancer, Can we use Ingress with our load balancer**

Yes, you can use Ingress with an inhouse load balancer by configuring the Ingress Controller to work with it:

* **Setup**: Deploy an Ingress Controller (e.g., NGINX) and configure it to use your load balancer as the external endpoint.
* **Annotation**: Use annotations like nginx.ingress.kubernetes.io/upstream-hash-by to integrate with your LB’s capabilities.
* **Custom Implementation**: If your LB supports APIs, you might need a custom controller (e.g., using the Kubernetes Ingress API) to sync rules.
* **Validation**: Test with curl <lb-ip>/<path> to ensure traffic routes correctly.

This requires aligning the LB’s features with the controller’s requirements.

**25. Your Deployment has replicas 3, but only 1 pod is running. What could be wrong**

Possible issues include:

* **Resource Constraints**: Nodes lack CPU/memory (check kubectl describe nodes for allocatable resources).
* **Taints/Tolerations**: Nodes are tainted with NoSchedule, and pods lack tolerations (check kubectl get nodes -o wide).
* **Pod Disruption**: A PDB or manual eviction limits pod creation.
* **Image Pull Error**: Check kubectl describe pod for ImagePullBackOff or incorrect image tags.
* **Scheduler Issues**: Verify with kubectl get events for scheduling failures.
* **Config Issue**: Review the Deployment YAML for misconfigured selectors or limits.

I’d start with kubectl describe deployment and logs to pinpoint the cause.

**26. Your pod mounts a ConfigMap, but changes to the ConfigMap are not reflected**

If ConfigMap changes aren’t reflected, it’s due to how mounts are handled:

* **Volume Mount**: ConfigMaps mounted as volumes are static; updates require pod restart. I’d use kubectl rollout restart deployment/<name> to apply changes.
* **Environment Variables**: If injected as env vars, they’re also static unless the pod restarts.
* **Dynamic Updates**: Use the configmap.reload sidecar or a custom controller to watch and reload ConfigMaps without restarting.
* **Verify**: Check kubectl describe pod to ensure the mount path matches the ConfigMap data.

Restarting the pod is the simplest fix for volume mounts.

**27. Explain how Node Affinity works and when will you use it**

* **Node Affinity**: A pod specification that influences scheduling based on node attributes (e.g., labels). Two types:
  + **requiredDuringSchedulingIgnoredDuringExecution**: Must match (hard constraint).
  + **preferredDuringSchedulingIgnoredDuringExecution**: Should match (soft preference).
  + Example: nodeAffinity: required: nodeSelectorTerms: - matchExpressions: - key: disktype value: ssd operator: In.
* **How It Works**: The kube-scheduler evaluates node labels against the affinity rules during pod placement.
* **Use Case**: I’ve used it to schedule compute-intensive pods on nodes with SSDs or GPU nodes for ML workloads, ensuring optimal performance.

**28. What is the difference between Node Affinity and Node Label Selector**

* **Node Affinity**: A pod-level policy in the spec (nodeAffinity) that uses complex match expressions (e.g., In, NotIn) to define scheduling preferences or requirements.
* **Node Label Selector**: A simpler node selection mechanism using nodeSelector in the pod spec, matching exact label key-value pairs (e.g., disktype: ssd).
* **Difference**: Node Affinity is more flexible with logical operators, while nodeSelector is a basic, exact-match filter. Affinity is preferred for advanced scheduling needs.

**29. What is container runtime in Kubernetes**

The **container runtime** is the software responsible for running containers on Kubernetes nodes. It manages container lifecycle (create, start, stop) and interacts with the OS.

* **Examples**: Docker, containerd, CRI-O.
* **Role**: Kubelet uses the Container Runtime Interface (CRI) to communicate with the runtime, pulling images and executing containers.
* **Evolution**: Kubernetes shifted from Docker to CRI-compliant runtimes (e.g., containerd) for modularity.
* **Use Case**: I’ve configured containerd on nodes to optimize performance in a large cluster.

It’s the foundational layer for container execution in Kubernetes.