**How do you expose a Kubernetes application to external traffic?**

We use **load balancer service** / **NodePort**

Example configuration:

apiVersion: v1

kind: Service

metadata:

name: my-app-service

spec:

type: LoadBalancer

selector:

app: my-app

ports:

- port: 80

targetPort: 8080

we use **ingress controller** : traefik , haproxy ..

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: my-app-ingress

spec:

rules:

- host: myapp.example.com

http:

paths:

- path: /

pathType: Prefix

backend:

service:

name: my-app-service

port:

number: 80

or simply we use **kubectl port-forwad svc/my-app-service 8080:80**

**What is the purpose of a NAT Gateway?**

A NAT Gateway (Network Address Translation Gateway) is used to allow private resources inside a network (like a private subnet in AWS, or private nodes in Kubernetes) to access the internet or other external networks without exposing them directly to inbound connections.

* A NAT Gateway lets private instances talk out, but blocks the internet from talking in.

Use case in AWS:

* Public Subnet → contains Internet Gateway (for public resources like a load balancer).
* Private Subnet → contains app servers or databases.
* NAT Gateway → placed in the **public subnet**, routes outbound traffic from the private subnet to the internet.

**How do you check running processes in Linux?**

* **Quick snapshot → ps -ef**
* **Live monitoring → top or htop**
* **Find specific process → pgrep**

**What command would you use to find files larger than 100MB?**

find /path/to/search **-type** f **-size** +100M

**What is the difference between Deployment and StatefulSet in Kubernetes?**

|  |  |
| --- | --- |
| **Deployment** | **StatefulSet** |
| Stateless Applications (apps that don’t need to remember data between restarts) | Used for **stateful applications** (apps that need stable network identity and persistent storage). |
| Pods are **interchangeable** | Each Pod has a **stable, unique identity** (name & network). |
| Scaling up/down is straightforward — any Pod can replace another. | Scaling keeps identities consistent (my-db-0, my-db-1, my-db-2). |
| Web servers (e.g., Nginx, Apache)/ REST APIs/Frontend applications | Databases (MySQL, PostgreSQL, MongoDB)  /Message queues (Kafka, RabbitMQ) |

**What is ConfigMap, and how is it different from a Secret?**

| **Feature** | **ConfigMap** | **Secret** |
| --- | --- | --- |
| **Data type** | **Non-sensitive** | **Sensitive** |
| **Encoding** | **Plain text** | **Base64** |
| **Security** | **Not encrypted by default** | **Can be encrypted at rest** |
| **Use case** | **App config, flags, env vars** | **Passwords, tokens, keys** |

**How do you check network connectivity between two servers?**

* **Ping → is it alive?**
* **Telnet/nc/curl → is the port/service open?**
* **Traceroute → where does it fail?**
* **iperf3 → how fast is the connection?**

**My CI/CD pipeline**

1. **GitLab triggers Jenkins:**

* When a user pushes a commit or performs an action in GitLab, it sends a webhook to Jenkins.
* Jenkins detects this change automatically.

1. **Jenkins runs the pipeline:**

* It pulls the latest code from GitLab.
* Executes the defined steps: build, test, and package the application.
* In your setup, it’s running inside a container, which ensures a consistent environment.

[

1. **Build stage (CI):**

* Jenkins pulls the latest code from GitLab.
* Builds the application (e.g., compiles code).
* Runs automated tests.

1. **Package stage:**

* Application is packaged into a **container image** (e.g., Docker image).
* The image is tagged with a version or commit hash.

1. **Push stage:**

* Jenkins **pushes the container image to a registry** (e.g., Docker Hub, AWS ECR, GitLab Container Registry).
* At this point, the image is stored and ready for deployment.

1. **Deployment stage (CD):**

* The production environment (e.g., Kubernetes cluster, Docker Swarm, ECS) **pulls the container image** from the registry.
* Deployment can be triggered by Jenkins directly or automatically via Kubernetes manifests, Helm charts, or GitOps tools.

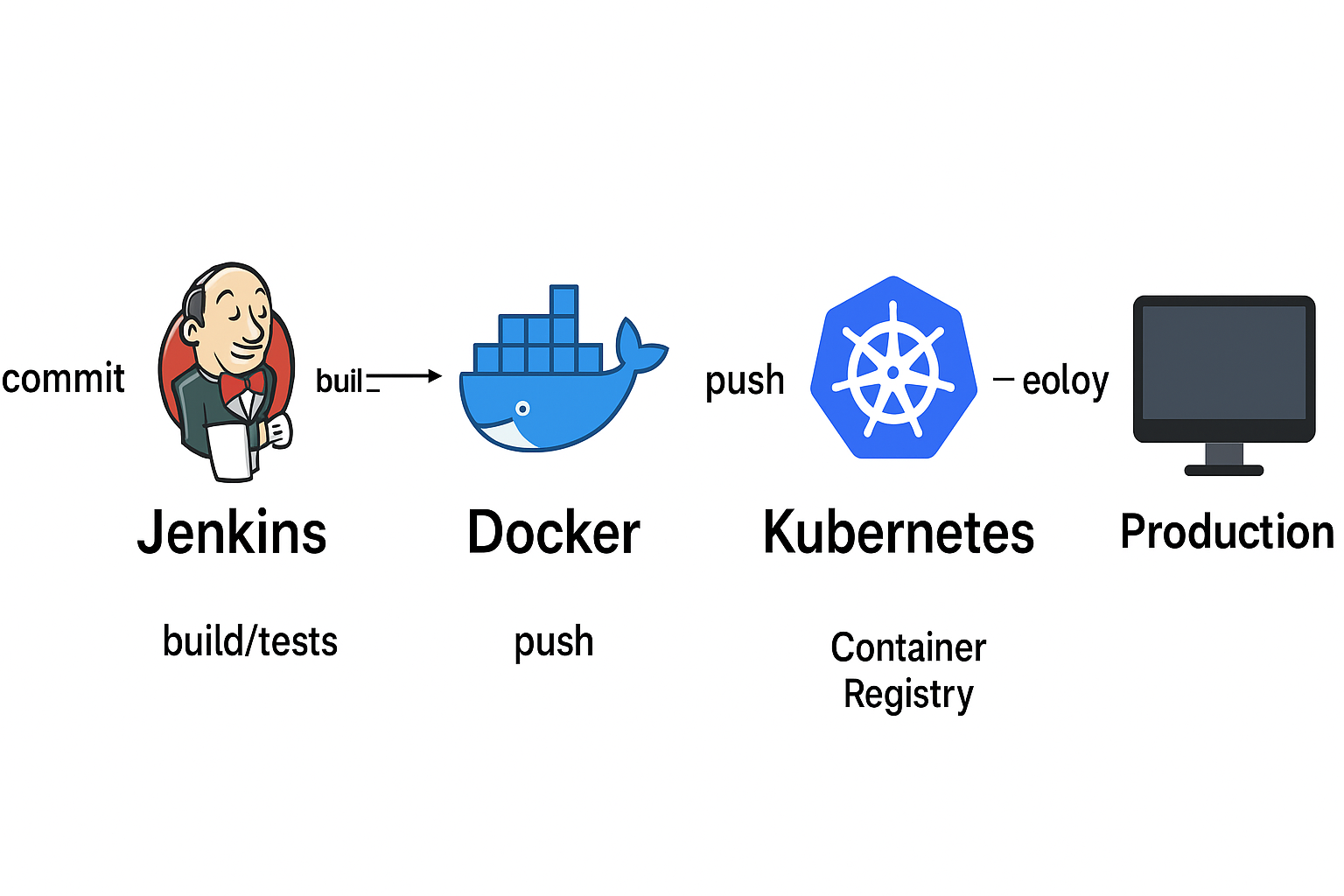
1. **Release/Prod:**

* The container is now running in the production environment.
* Jenkins can also run post-deployment steps like smoke tests, notifications, or monitoring checks.

]

1. **Delivery/Deployment:**

* After successful build and tests, Jenkins can deploy the application to a target environment or container platform.



**You have an application in Account A that needs to access an S3 bucket in Account B. How would you configure this?**

* 1. **Identify the S3 Bucket and Permissions**

 Determine what kind of access the application needs: Read, Write, or FullAccess.

 Decide if access should be to the entire bucket or just specific objects.

**Read-only:** Allow s3:GetObject

**Read/write:** Allow s3:GetObject, s3:PutObject, s3:DeleteObject

**Full bucket control:** Allow s3:\*

* 1. **Step 2: Create a Bucket Policy in Account B**

1. Go to S3 → select the bucket → Permissions → Bucket Policy.
2. Add a policy to allow Account A to access the bucket.

{

"Version": "2012-10-17",

"Statement": [

{

"Sid": "AllowAccountAFullAccess",

"Effect": "Allow",

"Principal": {

"AWS": "arn:aws:iam::111111111111:root" #Account A

},

"Action": [

"s3:GetObject",

"s3:PutObject",

"s3:DeleteObject"

],

"Resource": "arn:aws:s3:::my-bucket/\*" #should include both the **bucket** for listing and **objects (/\*)** for read/write.

},

{

"Sid": "AllowListBucket",

"Effect": "Allow",

"Principal": {

"AWS": "arn:aws:iam::111111111111:root"

},

"Action": "s3:ListBucket",

"Resource": "arn:aws:s3:::my-bucket"

}

]

}

**Step 3: Create an IAM Role in Account B**

1. Go to IAM → Roles → Create Role → Another AWS Account.
2. Enter Account A ID as the trusted account.
3. Attach a policy (like this for full access to the bucket):

**{**

**"Version": "2012-10-17",**

**"Statement": [**

**{**

**"Effect": "Allow",**

**"Action": "s3:\*",**

**"Resource": [**

**"arn:aws:s3:::my-bucket",**

**"arn:aws:s3:::my-bucket/\*"**

**]**

**}**

**]**

**}**

**Step 4: Assume Role from Account A**

**In Account A, your application IAM user or role must have permission to assume the cross-account role:**

**{**

**"Version": "2012-10-17",**

**"Statement": [**

**{**

**"Effect": "Allow",**

**"Action": "sts:AssumeRole",**

**"Resource": "arn:aws:iam::222222222222:role/AccountA-S3Access" #22.. : AccountID:role/name of role we give**

**}**

**]**

**}**

**Or we use boto3 (AWS SDK) more secure**

**- Write a Dockerfile for a Node.js application with multi-stage builds.**

A **multi-stage** build means you define multiple FROM instructions in a single Dockerfile.

# ----------------------

# Stage 1: Build Stage

# ----------------------

FROM node:20-alpine AS builder

# Set working directory

WORKDIR /app

# Copy package files first (for caching layer)

COPY package\*.json ./

# Install dependencies (dev + prod)

RUN npm install

# Copy application source code

COPY . .

# Build the application (if you have a build step, e.g., React frontend, TypeScript transpile)

RUN npm run build

# ----------------------

# Stage 2: Production Stage

# ----------------------

FROM node:20-alpine AS production

# Set working directory

WORKDIR /app

# Copy only package files

COPY package\*.json ./

# Install only production dependencies

RUN npm install --omit=dev

# Copy build artifacts from builder stage

COPY --from=builder /app/dist ./dist

# Expose app port

EXPOSE 3000

# Run the app

CMD ["node", "dist/index.js"]

**How do you handle Terraform state file corruption? 1st : terraform plan**

file corruption can happen due to : network failure , backend issues …

**Check for Local Backups**

**ls -l terraform.tfstate\* => terraform.tfstate / terraform.tfstate.backup**

then we copy the backup : **cp terraform.tfstate.backup terraform.tfstate**

**Try to Pull Current State**

**terraform state pull > terraform.tfstate**

**Validate the State File**

**terraform validate**

**terraform plan**

**Rebuild Missing State**

**terraform import aws\_instance.my\_ec2 i-1234567890abcdef**

**terraform import aws\_s3\_bucket.my\_bucket my-bucket-name**

**Refresh the State**

**terraform plan -refresh-only**

**Your EC2 instance in a private subnet needs to download packages without NAT Gateway. What alternatives exist?**

We use:

* ***VPC endpoint***
* ***Proxy, VPN:*** we launch an ec2 in public subnet with outbound internet , we configure it as a proxy and private the ec2 routes outbound traffic through the proxy.
* Mirror them in **S3 with VPC Endpoint : we copy the packages in s3 and we configure a s3 with vpc endpoint and pull them**

**How do you debug a container that exists?**

**docker ps -a**

**docker logs <container id>**

**docker inspect <container id> #get detailed info about the container**

**You need to import an existing AWS VPC into Terraform. What are the steps?**

* + 1. **We identify the VPC id that we want to import**
    2. **Terraform configuration:**

**Provider “aws”{**

**region = “ us-east-1”**

**}**

**Resource “aws\_vpc” “my\_vpc”{**

**Cidr\_block = “10.0.0.0/16”**

**}**

* + 1. **Terraform init**
    2. **Terraform import aws\_vpc.my\_vpc #idvpc**
    3. **Terraform show #display vpc attributes terraform**
    4. **Copy the output to the 2nd configuration**
    5. **Terraform plan**

**How would you implement blue-green deployment in Kubernetes?**

**Blue-Green deployment in Kubernetes means you run two versions of your application (Blue = current live version, Green = new version)**

**Blue deployment => current production app**

**Green deployment => new app version (test)**

**apiVersion:app/v1**

**kind:Deployment**

**metadata:**

**name:app-blue**

**spec:**

**replicas : 3**

**selector:**

**matchLabels:**

**app:myapp**

**version:blue**

**template:**

**metadata:**

**labels:**

**app:myapp**

**version: blue**

**spec:**

**containers:**

* + **name : myapp**

**image:myapp:v1**

**ports:**

* + **containerPort:80**

**apiVersion: apps/v1**

**kind: Deployment**

**metadata:**

**name: app-green**

**spec:**

**replicas: 3**

**selector:**

**matchLabels:**

**app: myapp**

**version: green**

**template:**

**metadata:**

**labels:**

**app: myapp**

**version: green**

**spec:**

**containers:**

**- name: myapp**

**image: myapp:v2 # new version**

**ports:**

**- containerPort: 80**

**Service**

**apiVersion: v1**

**kind:Service**

**metadata:**

**name: myapp-service**

**spec:**

**selector:**

**app:myapp**

**version:blue**

**ports:**

**-port: 80**

**targetPort:80 Deploy**

**kubectl apply -f app-blue.yaml**

**kubectl apply -f app-green.yaml**

**kubectl apply -f app-service.yaml**

**How do you manage secrets in Terraform without hardcoding them?**

**Variable “db\_password”{**

**type=string**

**sensitive=true**

**}**

**In secrets.tfvars**

**db\_password=”super123456”**

**What's the difference between COPY and ADD commands in Dockerfile?**

**COPY: copy files or directories from your local build context into the image**

**COPY <source><destination>**

**ADD: can copy local files (like copy)/download remote files using a URL/automatic extraction**

**ADD <source><destination>**

**ADD** [**https://example.com/file.tar.gz /tmp/**](https://example.com/file.tar.gz%20/tmp/)

**How would you implement cross-account resource provisioning using Terraform?**

* 1. **Create iam role in the target account +attach policies to allow the actions terraform needs**
  2. **Configure terraform to assume the role :**

**provider "aws" {**

**alias = "account\_b"**

**region = "us-east-1"**

**assume\_role {**

**role\_arn = "arn:aws:iam::222222222222:role/TerraformProvisionRole"**

**session\_name = "TerraformCrossAccountSession"**

**}**

**}**

* 1. **Create resource in the target account**

**resource "aws\_s3\_bucket" "example" {**

**provider = aws.account\_b**

**bucket = "my-cross-account-bucket"**

**acl = "private"**

**}**

**How would you handle secrets in a Docker container for a PHP application connecting to MySQL?**

**We won’t put the secret file hardcoded in docker image**

1. **Use env var**
2. **\*Use docker secret**

**echo "supersecret" | docker secret create db\_password -**

**echo "myuser" | docker secret create db\_user –**

**\*use them in docker-compose.yml**

**version: "3.8"**

**services:**

**app:**

**build: .**

**secrets:**

**- db\_password**

**- db\_user**

**environment:**

**DB\_HOST: db**

**DB\_USER\_FILE: /run/secrets/db\_user**

**DB\_PASSWORD\_FILE: /run/secrets/db\_password**

**db:**

**image: mysql:8**

**secrets:**

**- db\_password**

**- db\_user**

**environment:**

**MYSQL\_ROOT\_PASSWORD\_FILE: /run/secrets/db\_password**

**MYSQL\_USER\_FILE: /run/secrets/db\_user**

**MYSQL\_DATABASE: mydb**

**secrets:**

**db\_password:**

**external: true**

**db\_user:**

**external: true**

**\*in php**

**$db\_user = trim(file\_get\_contents($\_ENV['DB\_USER\_FILE']));**

**$db\_password = trim(file\_get\_contents($\_ENV['DB\_PASSWORD\_FILE']));**

**An S3 bucket was created via Terraform, but someone manually added a policy. How do you handle this drift?**

**Detect the drift by terraform plan**

**Update the .tf and reapply**

**How do you implement network policies to restrict pod-to-pod communication in Kubernetes?**

**Use podSelector and namespaceSelector to target pods**

**Define ingress/egress rules**

**Use default deny policy for security**

**Write a Python script to backup all files older than 30 days from a directory.**

**Import os,shutil,time**

**Source=”/path/to/source”**

**Backup=”path/to/backup”**

**Days=30**

**Cutoff = time.time() – Days\*86400**

**for file in os.listdir(source):**

**path=os.path.join(source,file)**

**if os.path.isfile(path) and os.path.getmtime(path) < cutoff:**

**shutil.copy2(path,os.path.join(Backup,file))**

**#print - Your company's cloud costs are increasing rapidly. - How would you approach cost optimization without impacting performance?**

**1-First we must check visibility and monitoring :generate monthly cost report, set up budget and alerts (to find where money is gone)**

**2. Eliminate Waste**

* **Identify and terminate idle instances (stopped >30 days).**
* **Delete orphaned resources (volumes, snapshots, IPs, load balancers).**
* **Check unused licenses (SQL Server, Oracle).**
* **Remove old test/dev resources no longer needed.**

**3. Rightsize Resources**

* **Review EC2/VM utilization (CPU, memory, network) → downsize overprovisioned instances.**
* **Optimize RDS/Databases (scale down, switch to serverless if workloads vary).**
* **Review Kubernetes clusters for over-allocated requests/limits.**
* **Enable auto-scaling for compute and containers.**

**5. Optimize Storage**

* **Enable S3/Blob/Cloud Storage lifecycle rules (Standard → Infrequent → Archive).**
* **Delete unused snapshots, AMIs, and old backups.**
* **Compress and deduplicate large data.**
* **Use EBS gp3 over gp2 (cheaper with same performance).**

**6. Optimize Networking**

* **Review data transfer costs (cross-AZ/region).**
* **Use CDN (CloudFront, Cloud CDN, Azure CDN) for static content.**
* **Replace NAT Gateway with VPC endpoints/PrivateLink if cheaper.**
* **Consolidate traffic to minimize inter-region data movement.**

**How would you set up geolocation-based routing using AWS services?**

**using Route 53 Geolocation Routing Policy. Step 1: Prepare resources in multiple regions**

* **Deploy your app in at least 2 AWS regions (e.g., us-east-1, eu-central-1).**
* **Each region should have its own load balancer, API endpoint, or S3 static site.**

**Step 2: Create a Route 53 Hosted Zone**

* **In Route 53, create or use an existing public hosted zone for your domain (e.g., example.com).**

**Step 3: Configure Geolocation Routing**

1. **Go to Route 53 → Hosted Zone → Create Record.**
2. **Choose Record type (A/AAAA for ALB/EC2, or CNAME for S3/CloudFront).**
3. **Select Routing Policy → Geolocation.**
4. **Choose the location (e.g., Europe, United States, or even France).**
5. **Enter the endpoint for that region (e.g., ALB DNS name in Frankfurt).**
6. **Repeat for other regions.**

**Step 4: Add a Default Policy**

* **Always add a Default record for users in locations you didn’t configure.**
* **Example: Route unknown countries → us-east-1.**

**A critical production Kubernetes cluster is experiencing multiple issues. Pods are stuck in ImagePullBackOff, some pods are being evicted, and users are reporting 503 errors from the application. What troubleshooting process will you follow, and how can to avoid this in the future?**

**First :**

**kubectl describe pod <pod> -n <ns> /node**

**kubectl get secret <image-pull-secret> -n <ns> -o yaml #verify secret and registry**

**kubectl run netcheck --rm -it --image=busybox -- nslookup <registry> && wget -qO- https://<registry>/v2/ #test DNS**

**503 (server)**

**Kubetctl get svc <svc>**

**Kubectl logs -n ingress-nginx**

**How to Prevent in Future**

* **Images: Use private registry, pin by digest, rotate secrets**
* **Capacity: Rightsize requests/limits, enable Cluster Autoscaler, use PriorityClasses**
* **Probes: Tune startup/readiness/liveness, set PodDisruptionBudgets**
* **Disk & Memory: Centralize logs, enable GC, monitor usage**
* **Observability: Dashboards + alerts on evictions, probe failures, registry errors**
* **Resilience: Use blue/green or canary deployments with rollback options**