1. Deploying 3 Tier App in to AWS with HA & Scalable

➢ 🧱 Architecture Overview

Tier 1: Presentation Layer (Web Layer)

- **Component**: EC2 instances or AWS Elastic Load Balancer with Auto Scaling group.
- **Function**: Hosts frontend (React, Angular, etc.).
- **Location**: In public subnets across multiple Availability Zones (AZs).

Tier 2: Application Layer (Logic Layer)

- **Component**: EC2 Auto Scaling group (or AWS Elastic Beanstalk).
- **Function**: Processes business logic (Node.js, Java, PHP etc.).
- **Location**: In private subnets across AZs.

Tier 3: Database Layer

- **Component**: Amazon RDS (e.g., MySQL, PostgreSQL) with Multi-AZ enabled.
- **Function**: Stores application data.
- **Location**: Private subnets, isolated.

➢ 📶 Step-by-Step Deployment Plan

1. VPC Setup

- Create a custom **VPC** (e.g., 10.0.0.0/16)
- Create **public** and **private subnets** in **2+ AZs**
- Create **Internet Gateway** and attach to the VPC
- Create **NAT Gateway** for private subnets' internet access

2. Security Groups

- Web Layer: Allow HTTP/HTTPS (from 0.0.0.0/0)
- App Layer: Allow traffic only from Web Layer
- DB Layer: Allow traffic only from App Layer

3. Web Tier (Frontend)

- Launch EC2 instances (or use Elastic Load Balancer + Auto Scaling)
- Host your frontend (HTML/JS or web server)
- Place instances in **public subnets**
- Attach to **Application Load Balancer (ALB)**

4. Application Tier

- Launch EC2 instances in private subnets with Auto Scaling
- Install backend (Node.js, Django, etc.)
- Connect ALB from Web Tier to these instances
- These are not exposed to the internet directly

5. Database Tier

- Launch **Amazon RDS** (Multi-AZ enabled)
- Choose appropriate instance class
- Place it in private subnets
- Ensure it can be accessed only by App Layer

➢ 🔁 High Availability (HA)

- **Multi-AZ Deployment** for RDS
- **EC2 Auto Scaling Groups** in **multiple AZs**
- **Elastic Load Balancer** to route traffic across AZs

➢ 📈 Scalability

- **Auto Scaling Groups** for Web and App tiers
- **Elastic Load Balancer** automatically handles traffic
- Use **Amazon CloudWatch** to scale based on CPU/Memory usage

➢ 🔒 Security Features

- Use **Security Groups** and **Network ACLs**
- **IAM Roles** for EC2 instances
- **SSM** to connect securely instead of public IP
- Use **Secrets Manager** for DB credentials

➢ 🌐 Optional Enhancements

- Use **Route 53** for custom domain & failover routing
- Use **CloudFront** for CDN caching
- Add **WAF** for Layer 7 protection
- Use **S3 + CloudFront** to host frontend statically

➢ 🧪 Testing & Monitoring

- Setup **CloudWatch** for logs and metrics
- Enable **AWS X-Ray** for tracing
- Perform failover and load testing

➢ 📦 Final Outcome

- A fully functional, secure, and scalable 3-tier app that:
- **Scales automatically**
- **Is highly available across AZs**
- **Follows AWS best practices**

2. Desigining DC DR Startegy in AWS

➢ ☑ What is DC-DR in AWS?

- **DC (Data Center)** = Main AWS region where your production runs.
- **DR (Disaster Recovery)** = Backup AWS region used when DC fails.
- Purpose: Ensure **business continuity**, **minimal downtime**, and **data protection**.

➢ 🎯 DR Goals

- **RTO** (Recovery Time Objective): How fast to recover?
- **RPO** (Recovery Point Objective): How much data loss is ok?

➢ 🔄 DR Strategy Types in AWS

Strategy	RTO/RPO	Cost
Backup & Restore	High	Low
Pilot Light	Medium	Low-Mid
Warm Standby	Low	Mid-High
Multi-Site Active-Active	Very Low	High

➢ 🛠️ Key AWS Services for DR

- **S3 Cross-Region Replication**: Auto copy data to DR region.
- **RDS Read Replica (Cross-region)**: Sync DB.
- **Route 53**: DNS-based automatic failover.
- **EC2 AMIs**: Backup machine images ready to launch.
- **CloudFormation**: Quick infra deployment in DR.
- **CloudWatch + SNS**: Monitoring & alerts.

🔐 Security & Testing

- Enable **encryption**, **IAM replication**, and **VPC peering**.
- Test DR setup regularly (mock failovers).
- Use **AWS Resilience Hub** to review fault tolerance.

📦 Example: Pilot Light Model

- **Primary Region**: Runs full app.
- **DR Region**: Only DB replica + app AMIs + S3 replicated.
- On disaster: Launch EC2, scale up, Route 53 switches traffic.

3. Automating Infra Setup by using Jenkins with Terraform modules

➢ 🔧 Tools Used

- **Jenkins** – Automation
- **Terraform** – Infra as Code
- **Terraform Modules** – Reusable infra blocks
- **AWS** – Cloud platform

➢ 🪜 Steps Overview

- **Create Terraform Modules**
  (e.g., VPC, EC2, RDS) inside `modules/`
- **Configure Remote Backend**
  Use **S3 + DynamoDB** for state management.
- **Install Jenkins & Plugins**
  Terraform, Git, Pipeline, etc.
- **Write Jenkins Pipeline**

  ```
  stage('Terraform Apply') {

    steps {

      sh 'terraform apply -auto-approve'

    }

  }
  ```

- **Run & Monitor Pipeline**
  Infra gets auto-created on AWS.

➢ ☑ Best Practices

- Use workspaces (`dev`, `prod`)
- Secure state and secrets

4. Deploying micro services in to AWS EKS

➢ 🧰 Tools Needed

- **AWS EKS** – Managed Kubernetes
- **kubectl** – CLI for Kubernetes
- **eksctl** – EKS cluster creation
- **Helm** – Package manager for K8s
- **Docker** – Build container images
- **GitHub + CI/CD** – Automate deployments

➢ 🪜 Steps Overview

- Create EKS Cluster
  - ✓ eksctl create cluster --name greens-cluster --region ap-south-1 --nodes 3
- Build & Push Docker Images
  - ✓ docker build -t <your-image>
  - ✓ docker push <your-ECR-repo>
- Write Kubernetes Manifests
  - ✓ `deployment.yaml` – Defines pods
  - ✓ `service.yaml` – Exposes app
  - ✓ `ingress.yaml` – (optional) HTTP routing
- Deploy to EKS
  - ✓ kubectl apply -f deployment.yaml
  - ✓ kubectl apply -f service.yaml
- Access the App
  - ✓ Use **LoadBalancer** or **Ingress Controller**
  - ✓ kubectl get svc

- Automate with CI/CD
    - ✓ Use **Jenkins/GitHub Actions**
    - ✓ Auto-deploy on every commit

➢ ☑ Pro Tips

- Use **Helm** for templating
- Enable **auto-scaling**
- Monitor with **Prometheus + Grafana**
- Secure with **IAM roles for service accounts (IRSA)**