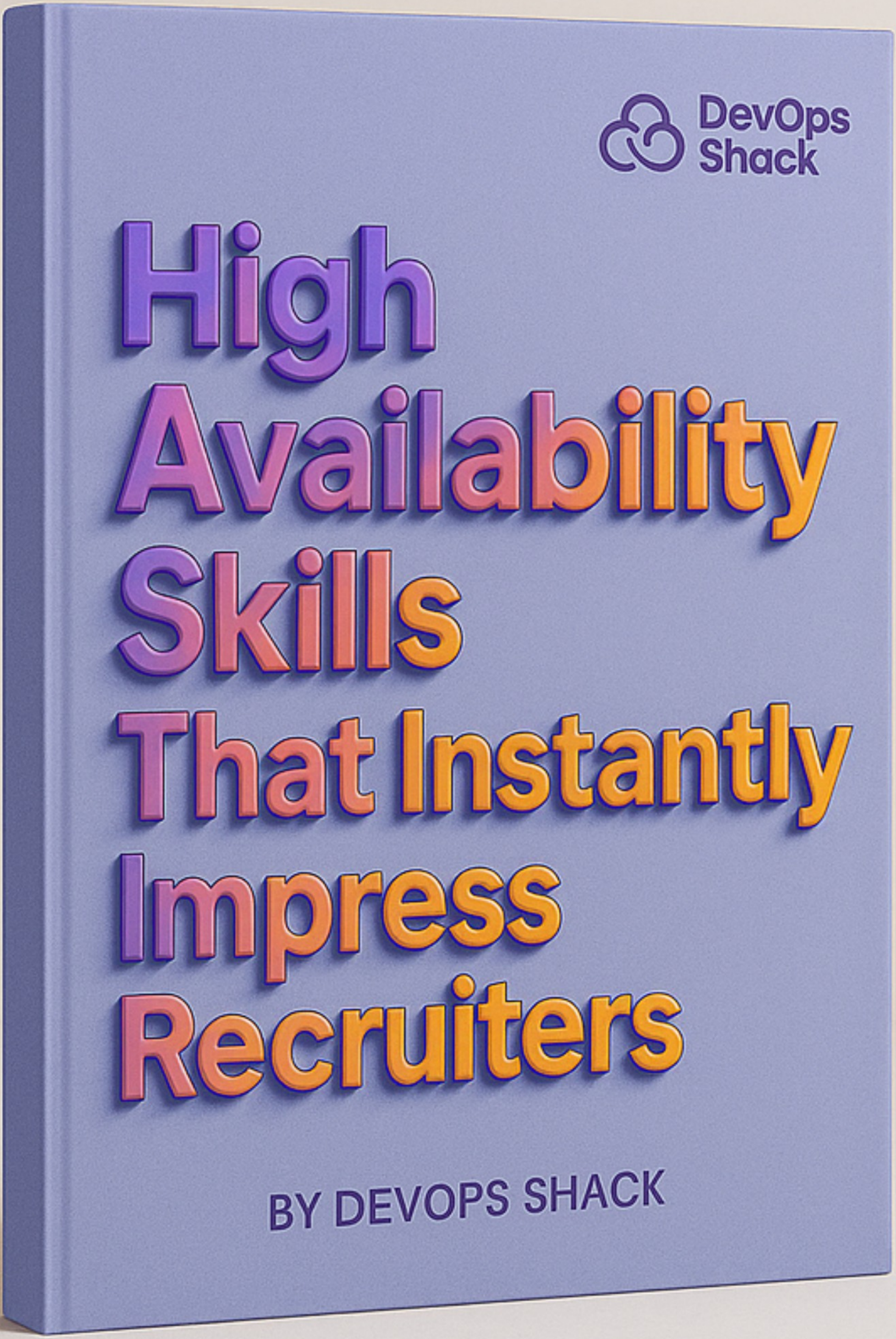


DevOps Shack

# High Availability Skills That Instantly Impress Recruiters

BY DEVOPS SHACK

# DevOps Shack

## *High Availability Skills That Instantly Impress Recruiters*

### ◍ Basic – *"HA Web Server with NGINX & Keepalived on AWS EC2"*

- Build a basic two-node web server cluster with automatic failover using Keepalived for VIP management.

### ◍ Intermediate – *"Highly Available Jenkins CI/CD with NFS Shared Storage on Kubernetes"*

- Deploy Jenkins in HA mode with multiple replicas, shared NFS volume for /var/jenkins_home, and Kubernetes Horizontal Pod Autoscaling.

### ◍ Advanced – *"Production-Grade HA Microservices Stack with Istio, Vault, MongoDB, and ArgoCD on EKS"*

- Full-scale project deploying multi-tier apps with HA MongoDB StatefulSet, HA Vault with Raft, Canary rollout using ArgoCD, and Istio- managed routing.

# ▧ High Availability in DevOps Projects

**What Recruiters Love to Hear — From Basics to Production-Grade Architectures**

In today's digital-first world, **downtime is expensive** — every second of unavailability can cost businesses users, revenue, and trust. That's why **High Availability (HA)** is no longer optional. It's a **core skill** every serious DevOps engineer must master.

But here's the twist: while many talk about HA, **few know how to design and implement it practically**. This guide fills that gap with **real-world DevOps projects** that showcase your ability to build resilient, fault-tolerant, self-healing systems.

Whether you're a beginner just getting started, an intermediate learner scaling your skills, or an advanced engineer prepping for top-tier interviews — this guide walks you through **3 real HA projects**, each one designed to:

☑ Deepen your technical architecture knowledge
☑ Help you stand out to recruiters with real proof of HA thinking
☑ Build confidence in designing, deploying, and explaining fault-tolerant systems

## 🎯 Why It Matters:

High Availability (HA) isn't just a buzzword — it's **business-critical**. Recruiters **love candidates** who:

☑ Think in terms of **fault tolerance**, not just features
☑ Architect for **99.99% uptime** or better
☑ Proactively **design recovery**, not just deployment

Whether you're applying for a DevOps, SRE, or Platform Engineer role — the **ability to think and build for resilience** is a skill that **sets you apart**.

Let's break it down into what you can **showcase in interviews, resumes, or projects** to highlight HA experience:

## 🧠 What Is High Availability (HA)?

High Availability means your systems remain **operational and accessible** even when:

A pod or container crashes

- A node goes down
- A region becomes unavailable
- A disk fails or IOPS drops
- A service gets overwhelmed with

traffic In technical terms, HA requires:

- **Redundancy:** Multiple instances of critical components
- **Load Balancing:** Traffic distribution to healthy nodes
- **Failover Mechanisms:** Automated switching to backup systems
- **Health Checks:** Probes to detect and react to failures
- **State Management:** Persistent data storage that survives failure

Recruiters don’t just want to know that you can *run* apps—they want to see that you can keep them running **no matter what**.

## ☑ Project 1: High Availability Web Server with NGINX & Keepalived on AWS EC2

## 📘 Level: Basic

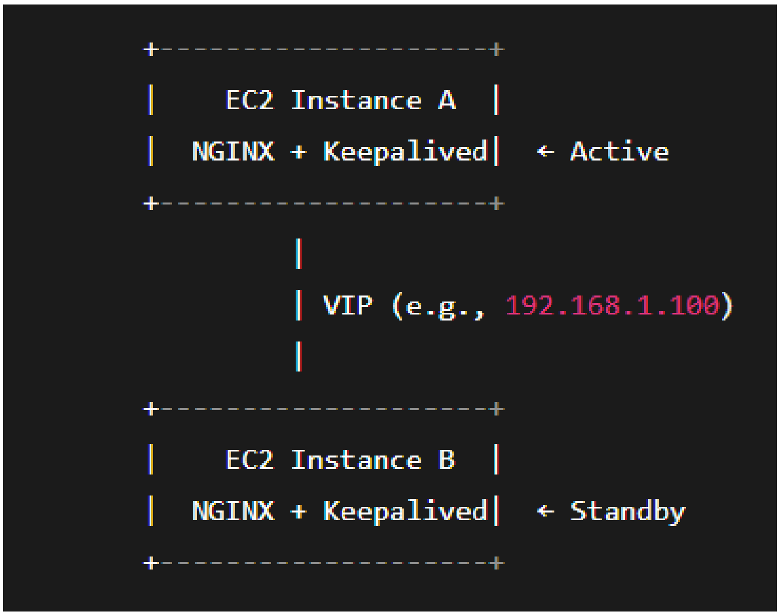
## 📌 Objective:

Deploy two NGINX web servers behind a **Virtual IP (VIP)** using **Keepalived** to ensure **automatic failover** when one EC2 instance goes down.

## 🧠 What You'll Learn

- Basic High Availability concepts using *active-passive setup*
- How to install and configure **NGINX**
- How to configure **Keepalived** to manage a **floating IP**
- Testing automatic failover when a node goes down

## 🏗 Architecture Overview



```
        +--------------------+
        |    EC2 Instance A  |
        |  NGINX + Keepalived|  ← Active
        +--------------------+
                 |
                 | VIP (e.g., 192.168.1.100)
                 |
        +--------------------+
        |    EC2 Instance B  |
        |  NGINX + Keepalived|  ← Standby
        +--------------------+
```

**Note:** Only one instance holds the VIP at a time. If the active one fails, the passive node becomes active.

## 🔧 Prerequisites

- 2 AWS EC2 instances (Ubuntu 22.04 recommended) in the same VPC and subnet

A **secondary private IP address** in the subnet to act as a floating VIP

- SSH access with sudo privileges to both instances
- Security Group allowing:
  - TCP port 80 (HTTP)
  - Protocol 112 (VRRP)
  - ICMP (for health check pings)

- **Step 1: Allocate a Secondary Private IP (VIP)**

1. Go to **EC2 Dashboard → Network Interfaces**
2. Find the network interface attached to **Instance A**
3. Click **Actions → Manage IP Addresses → Assign new IP**
4. Note down the new private IP (e.g., 192.168.1.100)
5. **Do NOT associate this with any instance manually** – Keepalived will do this dynamically.

- **Step 2: Install NGINX & Keepalived**

Run the following on **both instances**:

```
sudo apt update
sudo apt install -y nginx keepalived net-tools
```

Check installation:

```
nginx -v keepalived -
v
```

- **Step 3: Configure NGINX**

On **both nodes**, replace the default NGINX index page:

```
echo "Welcome to Instance A" | sudo tee /var/www/html/index.html #
```

Or on B:

```
echo "Welcome to Instance B" | sudo tee /var/www/html/index.html
```

Restart NGINX:

```
sudo systemctl restart nginx
```

Check on browser: http://<EC2_IP>

- **Step 4: Configure Keepalived on Instance A (MASTER)**

Create the config file:

```
sudo nano /etc/keepalived/keepalived.conf
```

Paste:

```
vrrp_instance VI_1
  { state MASTER
  interface eth0
  virtual_router_id 51
  priority 101
  advert_int 1
  authentication {
    auth_type PASS
    auth_pass devops123
  }
  virtual_ipaddress {
    192.168.1.100
  }
}
```

Save and restart:

```
sudo systemctl restart
keepalived
```

Check:

```
ip a | grep 192.168.1.100
```

You should see the VIP bound to eth0 on **Instance A**.

---

- **Step 5: Configure Keepalived on Instance B (BACKUP)**

Edit config:

```
sudo nano /etc/keepalived/keepalived.conf
```

Paste:

```
vrrp_instance VI_1
  { state BACKUP
  interface eth0
  virtual_router_id 51
  priority 100
  advert_int 1
  authentication {
    auth_type PASS
    auth_pass devops123
  }
  virtual_ipaddress {
    192.168.1.100
  }
}
```

Restart service:

```
sudo systemctl restart keepalived
```

Instance B won’t show the VIP yet (expected).

- **Step 6: Test High Availability (Failover)**

☑ **Check active instance:**

```
curl
```

# Should show: Welcome to Instance A

**Simulate failure:**

On Instance A:

```
sudo systemctl stop keepalived
```

On Instance B:

```
ip a | grep 192.168.1.100
```

# Should now see VIP assigned to eth0

Test again:

```
curl http://192.168.1.100
```

# Output: Welcome to Instance B

🎉 Success! HA failover works.

🔐 **Security Consideration**

- Ensure VRRP (protocol 112) is **allowed in the security group**
- Use stronger auth_pass for production
- Disable root SSH if not required
- Consider health check scripts and notifications for production

### 📝 What to Mention in Resume or Interview

☑ **Built an HA web server cluster using NGINX & Keepalived**
☑ **Configured VIP failover between EC2 instances via VRRP protocol**
☑ **Designed Active-Passive topology to simulate real-world disaster recovery**
☑ **Validated system resilience via simulated node failure tests**

## 🚀 Project 2: Highly Available Jenkins CI/CD with NFS Shared Storage on Kubernetes

**Level:**

**Intermediate** 📌

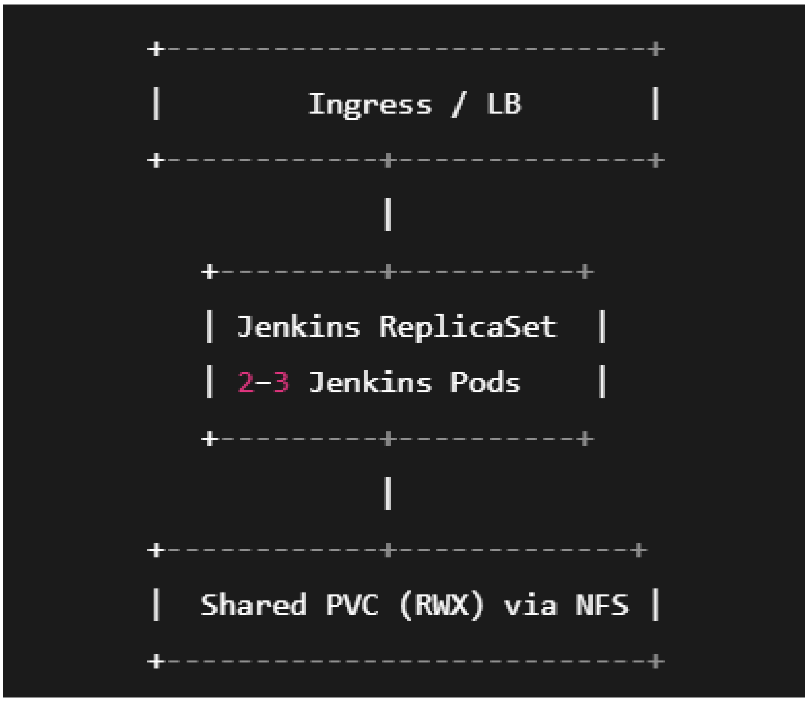
**Objective:**

Deploy **Jenkins in HA mode** on Kubernetes using **multiple replicas** backed by a **shared NFS volume**, ensuring that CI/CD pipelines stay available even if a pod crashes or gets rescheduled.

### 🧠 What You'll Learn

- Deploy Jenkins in **High Availability mode** on Kubernetes
- Configure **Persistent Shared Storage** using NFS
- Use **ReadWriteMany (RWX)** volumes to support multiple replicas
- Leverage **Horizontal Pod Autoscaler (HPA)** and **liveness probes**
- Test pod crash recovery and job state persistence

### 🏗 Architecture Overview



```
+--------------------------+
|        Ingress / LB      |
+------------+-------------+
             |
   +---------+----------+
   | Jenkins ReplicaSet |
   | 2-3 Jenkins Pods   |
   +---------+----------+
             |
+------------+------------+
|  Shared PVC (RWX) via NFS |
+--------------------------+
```

### 🔧 Prerequisites

- Kubernetes cluster (minikube or EKS/GKE/AKS)
- NFS Server deployed (or use NFS provisioner)
- Helm installed

kubectl configured

- Ingress controller (NGINX or ALB Ingress)

- **Step 1: Setup a Shared NFS Volume (Using NFS Provisioner)**

If you don’t already have NFS:

```
helm repo add nfs-subdir-external-provisioner https://kubernetes-sigs.github.io/nfs-subdir-external-provisioner/

helm repo update


helm install nfs-server nfs-subdir-external-provisioner/nfs-subdir-external-provisioner \

  --set nfs.server=<NFS_SERVER_IP> \

  --set nfs.path=/exported/path \

  --set storageClass.name=nfs-rwx
```

- **Step 2: Add Bitnami Jenkins Helm Repo**

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

   helm repo update
```

- **Step 3: Prepare a jenkins-values.yaml File**

```
   controller:

     replicaCount: 2

 strategyType: RollingUpdate adminPassword:

     "admin123"


     persistence:

       enabled: true
```

```
  storageClass: "nfs-rwx"
  accessMode: ReadWriteMany
  size: 10Gi

healthProbes:
  startupProbe:
    enabled: true
    initialDelaySeconds: 30
    periodSeconds: 10
    failureThreshold: 12
  livenessProbe:
    enabled: true
    initialDelaySeconds: 90
    periodSeconds: 10
    failureThreshold: 6

service:
  type: ClusterIP

ingress:
  enabled: true
  ingressClassName:
  "nginx" hostname:
  jenkins.local tls: false
```

- **Step 4: Deploy Jenkins with Helm**

```
helm install jenkins bitnami/jenkins -f jenkins-values.yaml -n jenkins --create-namespace
```

- **Step 5: Verify Jenkins Is Running in HA**

```
kubectl get pods -n jenkins
```

You should see **two Jenkins pods** running.
Check logs for successful startup:

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

- **Step 6: Access Jenkins UI**

Update /etc/hosts:

```
<INGRESS_IP> jenkins.local Then
access: http://jenkins.local
```

Login with:

```
User: user  Password:
admin123
```

- **Step 7: Create a Sample Pipeline and Validate HA**

1. Create a simple **Freestyle Job** or **Pipeline Job**
2. Trigger it
3. While the job is running, **delete the active pod**:

```
kubectl delete pod <jenkins-0> -n jenkins
```

☑ Job should continue in the next pod
☑ UI remains accessible
☑ Pipeline state persists (thanks to shared NFS volume)

- **Step 8: Enable Horizontal Pod Autoscaling (Optional)**

```
kubectl autoscale deployment jenkins -n jenkins --cpu-percent=60 --min=2 --max=4
```

You can simulate CPU load to see HPA in action.

### ▧ What to Mention in Resume or Interview

☑ **Deployed Jenkins in High Availability mode using Kubernetes & NFS**
☑ **Ensured zero downtime CI/CD using multiple controller replicas and RWX PVC**
☑ **Configured liveness/startup probes for fault detection and HPA for scaling**
☑ **Simulated pod failure to validate pipeline continuity and HA setup**

## 🔥 Project 3: Production-Grade HA Microservices Stack with Istio, Vault, MongoDB & ArgoCD on EKS

◍ **Level: Advanced**

📌 **Objective:**

Design and deploy a **highly available microservices platform** on **AWS EKS**, implementing:

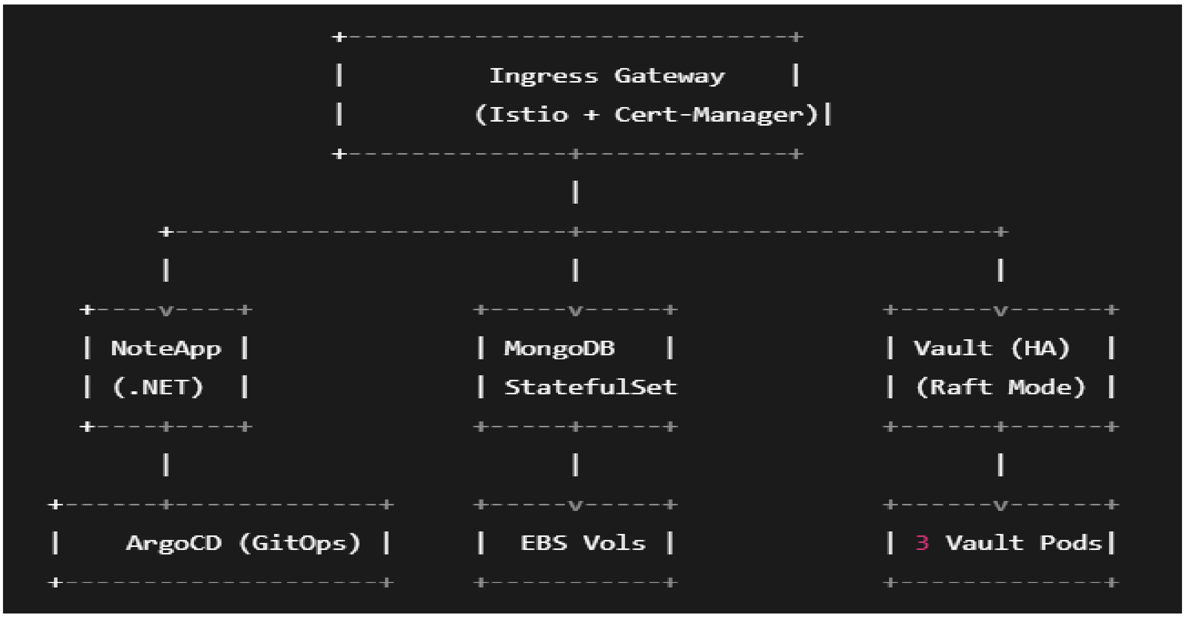
HA MongoDB StatefulSets with data replication

- HA Vault setup with Integrated Raft storage
- Progressive Delivery via ArgoCD (with Canary rollouts)
- Traffic management and observability via Istio

## 🧠 What You'll Learn

- Build a real-world **microservices architecture** on EKS
- Configure HA for databases (MongoDB), secrets (Vault), and workloads
- Automate GitOps deployments using ArgoCD
- Enable **secure service-to-service communication** via Istio mTLS
- Validate high availability via failover simulations and health checks

## 🏗 High-Level Architecture



```
                      +-----------------------------+
                      |       Ingress Gateway       |
                      |    (Istio + Cert-Manager)|
                      +--------------+--------------+
                                     |
         +---------------------------+---------------------------+
         |                           |                           |
    +----v----+                +-----v-----+              +------v------+
    | NoteApp |                | MongoDB   |              | Vault (HA)  |
    | (.NET)  |                | StatefulSet              | (Raft Mode) |
    +----+----+                +-----+-----+              +------+------+
         |                           |                           |
  +------+-------------+       +-----v-----+              +------v------+
  |    ArgoCD (GitOps) |       |  EBS Vols |              | 3 Vault Pods|
  +--------------------+       +-----------+              +-------------+
```

## 🔧 Prerequisites

- EKS Cluster (3 or more nodes across multiple AZs)
- Helm & kubectl installed
- Domain name & Route53 (for Ingress DNS)
- SSL certificate via cert-manager
- ArgoCD, Istio, and Vault Helm charts

- **Step 1: Deploy Istio for Ingress & mTLS**

```
istioctl install --set profile=demo -y
kubectl label namespace default istio-injection=enabled
```

Deploy Istio Gateway and VirtualService to expose your app:

```
    # ingress-gateway.yaml
apiVersion: networking.istio.io/v1beta1 kind:
    Gateway
    ...
```

Enable **mTLS and DestinationRules** to handle traffic securely.

- **Step 2: Deploy MongoDB as HA StatefulSet**

To ensure **High Availability**, we will deploy MongoDB as a **StatefulSet** with:

- **3 replicas**
- **Pod-specific persistent volumes** using EBS
- **Headless service** for internal DNS
- **Readiness and liveness probes**
- **ReplicaSet configuration** for automatic failover

📁 **File 1: mongo-headless-service.yaml**

```
apiVersion: v1
    kind: Service
    metadata:
    name: mongo
      namespace: app
    spec:
```

```
  ports:
    - port: 27017
  clusterIP: None  # headless
  selector:
    app: mongo
```

### 📁 File 2: mongo-statefulset.yaml

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mongo
namespace: app
spec:
  serviceName: "mongo"
  replicas: 3
  selector:
  matchLabels:
  app: mongo
  template:
  metadata:
      labels:
        app: mongo
    spec:
    containers:
        - name: mongo
          image: mongo:4.4
      ports:
       - containerPort: 27017
      volumeMounts:
       - name: mongo-persistent-storage
        mountPath: /data/db
      livenessProbe:
       exec:
        command:
         - mongo
         - --eval
         - db.adminCommand('ping')
       initialDelaySeconds: 30
       periodSeconds: 10
      readinessProbe:
       exec:
        command:
         - mongo
         - --eval
         - db.adminCommand('ping')
       initialDelaySeconds: 10
       periodSeconds: 10
  volumeClaimTemplates:
   - metadata:
     name: mongo-persistent-storage
    spec:
     accessModes: ["ReadWriteOnce"]
    storageClassName: ebs-sc
        resources:
          requests:
            storage: 5Gi
```

### 📁 File 3: mongo-init-config.yaml (for ReplicaSet Setup)

You need to **initiate the ReplicaSet** after the pods are running:

```
kubectl exec -it mongo-0 -n app -- mongosh
```

```
Then run:
js
rs.initiate({
  _id: "rs0",
  members: [
    { _id: 0, host: "mongo-0.mongo.app.svc.cluster.local:27017" },
    { _id: 1, host: "mongo-1.mongo.app.svc.cluster.local:27017" },
    { _id: 2, host: "mongo-2.mongo.app.svc.cluster.local:27017" }
  ]
})
```

### ☑ Verify ReplicaSet Health

```
kubectl exec -it mongo-0 -n app -- mongosh
rs.status()
```

Look for:

- One **PRIMARY**
- Two **SECONDARY**

health: 1 for all members

### G HA Testing

1. **Kill the PRIMARY pod:**

```
kubectl delete pod mongo-0 -n app
```

2. Run rs.status() again from mongo-1 or mongo-

2. Also create:

- mongo-headless service (ClusterIP: None)
- Readiness & Liveness probes
- Affinity rules to spread pods across

AZs Use:

```
helm install mongodb percona/psmdb-db --namespace app --values
    values.yaml
```

### o Step 3: Deploy Vault in HA Mode (Raft)

```
helm repo add hashicorp https://helm.releases.hashicorp.com helm

    repo update
```

helm install vault hashicorp/vault -n vault --create-namespace -f vault-values.yaml

Vault config (vault-values.yaml):

```
    server:

  ha:

    enabled: true raft:

        enabled: true

      dataStorage:
```

```
    enabled: true
    storageClass: ebs-sc
    size: 10Gi
injector:
  enabled: true
```

Initialize and unseal Vault:

```
kubectl exec -it vault-0 -n vault -- vault operator init
```

- **Step 4: Deploy Microservice (e.g., NoteApp)**

```
# Deployment uses annotations for Vault injector annotations:
  vault.hashicorp.com/agent-inject: "true"
  vault.hashicorp.com/role: "noteapp"
  vault.hashicorp.com/agent-inject-secret-MongoDB__ConnectionString: "secret/data/noteapp"
```

ServiceAccount, Vault policy, and Kubernetes auth setup required.

Deploy app with:

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

- **Step 5: Setup ArgoCD for GitOps + Canary**

```
kubectl create namespace argocd
helm install argocd argo/argo-cd -n argocd
```

Create Application YAML pointing to GitHub repo.

Set up **Progressive Delivery**:

```
canary:
  steps:
    - setWeight: 20
    - pause: {}
    - setWeight: 50
    - pause: { duration: 30s }
```

### Step 6: Configure Ingress + TLS

☑ **Objective:**

Secure external access to your NoteApp with:

- **Istio Ingress Gateway**
- **DNS domain mapped to LoadBalancer**
- **TLS certificates managed by cert-manager**

🔧 **Prerequisites:**

- **Istio installed and configured**
- **cert-manager installed**
  Install it via Helm if not already:

```
helm repo add jetstack https://charts.jetstack.io
helm repo update
helm install cert-manager jetstack/cert-manager \
  --namespace cert-manager --create-namespace \
  --set installCRDs=true
```

- A **domain name** (e.g., noteapp.example.com)
- A valid **email address** (used by Let's Encrypt)

📁 **Step A: Create a ClusterIssuer (Let’s Encrypt)**

```
# cluster-issuer.yaml
apiVersion:
cert-manager.io/v1 kind:
```

## ClusterIssuer

```
metadata:
  name: letsencrypt-prod
spec:
  acme:
    email: your-email@example.com
    server: https://acme-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      name: letsencrypt-prod
    solvers:
      - http01:
          ingress:
            class: istio
```

Apply it:

```
kubectl apply -f cluster-issuer.yaml
```

### 📁 Step B: Create Certificate Resource

```
# certificate.yaml
apiVersion:
cert-manager.io/v1 kind:
Certificate
metadata:
  name: noteapp-cert
namespace: app
spec:
  secretName: noteapp-tls
  issuerRef:
    name: letsencrypt-prod
```

```
kind: ClusterIssuer
  commonName: noteapp.example.com
  dnsNames:
    - noteapp.example.com
```

Apply it:

```
kubectl apply -f certificate.yaml
```

This will automatically generate a TLS certificate and store it in the noteapp-tls Kubernetes secret.

### Step C: Configure Istio Gateway & VirtualService

```
# istio-gateway.yaml
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: noteapp-gateway
  namespace: app
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 443
      name: https
      protocol: HTTPS
      tls:
        mode: SIMPLE
        credentialName: noteapp-tls
  hosts:
    - noteapp.example.com
```

-----------------------------------------------------------------------

```
# istio-virtualservice.yaml
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: noteapp-vs
namespace: app
spec:
  hosts:
    - noteapp.example.com
  gateways:
    - noteapp-gateway
  http:
    - match:
        - uri:
            prefix: /
      route:
        - destination:
            host:
            noteapp port:
            number: 80
```

Apply them:

```
kubectl apply -f istio-gateway.yaml
kubectl apply -f istio-virtualservice.yaml
```

### 🌐 Step D: DNS Configuration

1. Get **LoadBalancer IP** of Istio Gateway:

```
kubectl get svc istio-ingressgateway -n istio-system
```

2. Add an **A record** in your DNS provider (Route53, GoDaddy, etc.): noteapp.example.com → <LoadBalancer IP>

Wait for DNS propagation (usually <5 min).

### ☑ Test Secure Access

Visit:

https://noteapp.example.com

✔ Should load the NoteApp securely
✔ TLS padlock visible
✔ Certificate issued by Let’s Encrypt

### ▧ What to Mention in Resume or Interview

☑ **Designed and implemented a production-grade HA microservices platform on AWS EKS**
☑ **Enabled HA for Vault (Raft), MongoDB (ReplicaSet), and ArgoCD (GitOps)**
☑ **Secured service-to-service communication via Istio mTLS**
☑ **Used StatefulSets, PVs, and Helm charts to ensure self-healing and fault tolerance**
☑ **Applied Progressive Delivery strategy to minimize release risks**

### ☑ Summary of Tools Used

| Component | Tool |
|---|---|
| App Platform | Kubernetes on EKS |
| HA Ingress & Traffic | Istio |
| GitOps | ArgoCD |
| Secrets Management | Vault (HA Raft) |
| DB with Replication | MongoDB (StatefulSet + PVC) |
| Storage | AWS EBS |
| TLS Certs | cert-manager |
| CI/CD | Jenkins (Optional extension) |

## ○ Conclusion: Building Resilience

High Availability (HA) isn't just a technical checkbox — it's a **philosophy** that defines how you think, design, and operate systems at scale.

In this guide, you've walked through **three progressively challenging projects** that demonstrate your ability to:

☑ Architect resilient systems that **don't break under pressure**
☑ Handle failures gracefully using **active-passive setups**, **replication**, and **self-healing deployments**
☑ Integrate the latest in **GitOps, Service Mesh, StatefulSets, Vault HA**, and more
☑ Think like an **SRE, DevOps, or Platform Engineer** who owns not just uptime, but user trust

Whether you're prepping for your first job, switching roles, or trying to stand out in a crowded job market, showing that you've actually **implemented HA in real projects** is a **career-defining advantage**.

### 🔓 What to Do Next

✨ Add these projects to your GitHub portfolio
📄 Mention HA highlights clearly in your resume
💬 Be ready to **explain these architectures** in interviews
🎥 Record short video demos or blog posts showcasing failover scenarios
🚀 Start treating *availability* as a core design principle in all your future projects