ChatGPT

Sivaraj Shanmugam's Cloud-Native E-Commerce Platform on AWS: An End-to-End DevOps Case Study

This case study documents the architecture, implementation, and operation of a production-grade, cloud-native e-commerce platform on AWS. The solution uses Infrastructure as Code (Terraform) to provision networking (VPC), compute (EKS), security (IAM), and database (RDS). Application components are containerized with Docker (frontend and backend) and orchestrated with Kubernetes manifests (Deployments, Services, Ingress, ConfigMaps, Secrets, HPA). A Jenkins pipeline automates CI/CD, and Velero handles backup/restore. Monitoring with Prometheus/Grafana and AWS SNS alerts ensure observability. We cover code listings, architecture diagrams, command outputs, troubleshooting scenarios, and tips for interviews, resume bullets, and a GitHub README.

Cloud Infrastructure Architecture

The platform follows a multi-tier microservices design on AWS for scalability, availability, and managed operations. A VPC contains public and private subnets across Availability Zones, with an Internet Gateway and NAT Gateway for routing. Amazon EKS (managed Kubernetes) runs the containerized services; AWS RDS (Multi-AZ) hosts the database; and an Application Load Balancer routes external traffic to the services. Figure 1 below illustrates the overall architecture.

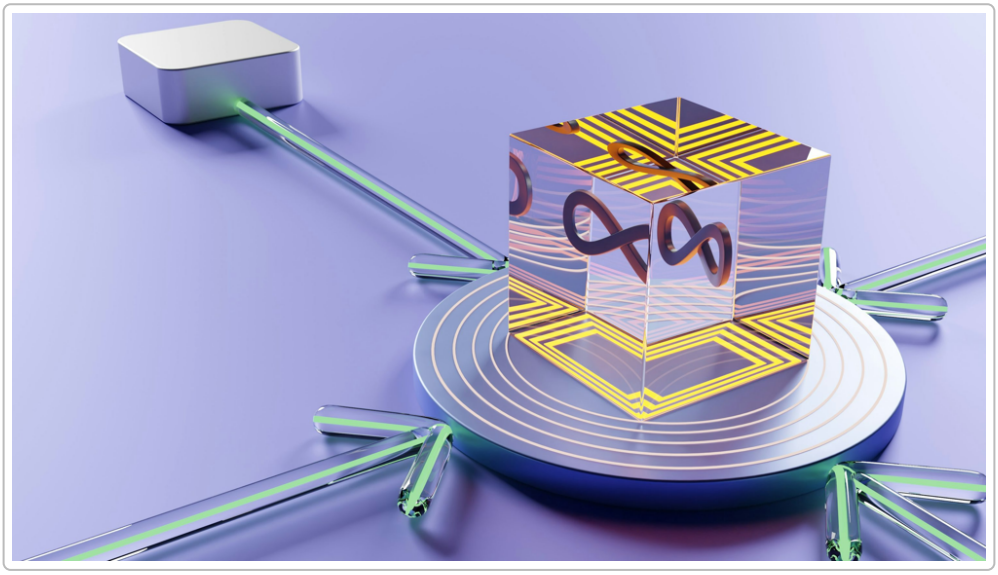

Figure 1: High-level architecture overview of the cloud-native e-commerce platform (multi-AZ AWS VPC, EKS cluster, RDS, Load Balancer, etc.).

In this design, AWS-managed services are leveraged for operational excellence. For example, using Amazon EKS offloads control-plane management and ensures high availability. As one analysis notes, managed services like ECS/EKS, RDS, and cloud load balancers eliminate much infrastructure overhead [1] [2]. The Terraform code below (see **Terraform: VPC Module** and **EKS Module**) creates the necessary network and cluster resources.

Auto-scaling and self-healing are built in: the EKS worker nodes auto-scale (via an Auto Scaling Group) and Kubernetes automatically restarts failed pods. A Horizontal Pod Autoscaler (HPA) adjusts replica counts based on CPU/memory usage. For example, HPA uses the Kubernetes Metrics API; if the metrics server or resource requests are misconfigured, scaling can fail [3]. In normal operation, HPA targets (~50% CPU utilization) ensure the application can handle variable load by scaling between a minimum and maximum pod count. Redis (an in-memory cache) improves performance, and if Redis becomes unavailable (e.g. due to high memory usage or network issues), the application gracefully degrades with warnings (see **Troubleshooting**).

To secure access, IAM roles and AWS IAM Authenticator (via an `aws-auth` ConfigMap) map Kubernetes service accounts to IAM roles. If `kubectl` reports errors like `AccessDenied` or `Unauthorized`, it usually means the AWS credentials or `kubeconfig` aren't correct [4] [5]. For example:

> *"could not get token: AccessDenied: Access denied"* – fix by running `aws eks update-kubeconfig` for the correct cluster and IAM principal [4] [5].

Infrastructure as Code (Terraform)

We use Terraform to declare all AWS resources. Terraform modules from the Terraform Registry simplify this. For example, the **VPC** is created using `terraform-aws-modules/vpc/aws`, which sets up subnets, IGW/NAT gateways, route tables, and security groups with minimal code [6]. The **EKS cluster** is created using `terraform-aws-modules/eks/aws`, automating the control plane and node groups [2]. We also use the IAM OIDC module to allow Kubernetes service accounts to assume IAM roles [7]. Below are code snippets illustrating each module:

Terraform: VPC Module

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "4.0.0"
  name    = "ecommerce-vpc"
  cidr    = "10.0.0.0/16"

  azs             = ["us-east-1a", "us-east-1b"]
  public_subnets  = ["10.0.1.0/24", "10.0.2.0/24"]
  private_subnets = ["10.0.3.0/24", "10.0.4.0/24"]

  enable_nat_gateway    = true
  single_nat_gateway    = true
  enable_dns_hostnames  = true
```

```
  enable_dns_support    = true

  tags = {
    "Name" = "ecommerce-vpc"
  }
}
```

This creates a new VPC with public and private subnets across two AZs, one NAT gateway, and routing. Using this module ensures best practices (multiple AZs, separate subnets, proper routing tables) with minimal configuration [6].

Terraform: EKS Module

```
module "eks" {
  source          = "terraform-aws-modules/eks/aws"
  version         = "18.0.0"
  cluster_name    = "ecommerce-eks"
  cluster_version = "1.24"

  vpc_id     = module.vpc.vpc_id
  subnet_ids = module.vpc.private_subnets

  # Worker node group configuration
  node_groups = {
    workers = {
      desired_capacity = 3
      max_capacity     = 5
      instance_types   = ["t3.medium"]
      key_name         = "my-key"
    }
  }

  # Kubernetes worker IAM role for node group
  manage_aws_auth = true
}
```

This defines an EKS cluster within the private subnets of our VPC. It creates worker nodes (EC2 instances) in an Auto Scaling Group with the specified capacity. The cluster's control plane is fully managed by AWS (reducing admin overhead) and multiple node instances across AZs provide high availability. The use of Terraform's EKS module greatly simplifies setting up complex networking and IAM relationships for Kubernetes [2].

Terraform: IAM OIDC Module

```
module "iam_oidc" {
  source  = "terraform-aws-modules/iam/aws//modules/iam-assumable-role-with-
oidc"
  version = "5.0.0"
  providers = { aws = aws }

  create_role = true
  name        = "EKSServiceAccountRole"
  oidc_provider_url = data.aws_eks_cluster.cluster.identity[0].oidc[0].issuer
  audience    = "sts.amazonaws.com"

  # Trust policy and permissions for service accounts
  role_name = "eks-sa-role"
  policy_arns = [aws_iam_policy.s3_read.arn]
}
```

This module sets up an IAM role that Kubernetes service accounts can assume via OIDC. It's useful for giving pods fine-grained AWS permissions. For example, we might attach an S3-read policy so a pod can access S3. The Terraform OIDC module handles the Kubernetes OIDC provider and trust relationship under the hood [7].

Terraform: RDS Module

We deploy an Amazon RDS instance for relational data (e.g. orders, customers). For high availability, RDS is configured in Multi-AZ mode (AWS automatically fails over to the standby in another AZ). For example:

```
resource "aws_db_subnet_group" "ecom_subnet" {
  name       = "ecom-db-subnet-group"
  subnet_ids = module.vpc.private_subnets
}

resource "aws_db_instance" "ecommerce" {
  identifier          = "ecomdb"
  engine              = "postgres"
  instance_class      = "db.t3.medium"
  allocated_storage   = 20
  name                = "ecomdb"
  username            = "admin"
  password            = var.db_password
  multi_az            = true
  publicly_accessible = false
  db_subnet_group_name = aws_db_subnet_group.ecom_subnet.id
```

```
  vpc_security_group_ids = [module.vpc.default_security_group_id]
}
```

This creates a PostgreSQL database with Multi-AZ enabled. Multi-AZ RDS ensures that if one AZ goes down, the standby instance in another AZ becomes the new primary, keeping data available [8]. The database resides in private subnets and is not internet-facing.

Infrastructure Outputs and Cleanup

After `terraform apply`, Terraform outputs cluster details. A typical output log includes lines like:

```
Plan: 63 to add, 0 to change, 0 to destroy.
Apply complete! Resources: 63 added, 0 changed, 0 destroyed.
cluster_endpoint = "https://ABC123XYZ.gr7.us-east-1.eks.amazonaws.com"
cluster_name     = "ecommerce-eks"
region           = "us-east-1"
```

[9] [10]. These confirm the new cluster and resources. We then run:

```
aws eks --region $(terraform output -raw region) update-kubeconfig \
    --name $(terraform output -raw cluster_name)
```

to configure `kubectl` for EKS. (Errors like *"You must be logged in to the server"* indicate kubeconfig or IAM auth issues [4] [5].)

Containerization (Docker)

The application has two main components: a **frontend** (e.g. a Node/React app) and a **backend** (e.g. a Python/Flask API). Each is containerized with a Dockerfile following best practices [11]. We use official base images (Node, Python) and multi-stage builds.

- **Frontend Dockerfile**: Builds a static site with Node and serves it via NGINX.

```
# build stage
FROM node:18-alpine AS build
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# final stage
```

```
FROM nginx:alpine
COPY --from=build /app/build /usr/share/nginx/html
```

This multi-stage build keeps the image lean (only the production build artifacts in the final image).

- **Backend Dockerfile**: Installs Python dependencies and runs a WSGI server.

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 5000
CMD ["gunicorn", "app:app", "--bind", "0.0.0.0:5000"]
```

We use the official `python:3.11-slim` base for efficiency. The Flask (or FastAPI) app listens on port 5000.

Using Docker multi-stage builds and official images follows recommended best practices for CI/CD pipelines [11]. Each Dockerfile is stored alongside the app code (e.g. in `frontend/Dockerfile` and `backend/Dockerfile`).

Kubernetes Manifests

Kubernetes YAML files declare how to run the containers in the cluster. Below are core examples:

- **Deployments** – define desired pods/containers and images:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - name: frontend
```

```
        image: <AWS_ACCOUNT>.dkr.ecr.us-east-1.amazonaws.com/
frontend:latest
        ports:
        - containerPort: 80
        envFrom:
        - configMapRef:
            name: frontend-config
        - secretRef:
            name: ecom-secrets
        resources:
          requests:
            cpu: 100m
            memory: 128Mi
          limits:
            cpu: 500m
            memory: 512Mi
        readinessProbe:
          httpGet:
            path: /health
            port: 80
          initialDelaySeconds: 10
          periodSeconds: 5
```

• **Services** – expose pods inside the cluster and (optionally) outside:

```
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 80
  selector:
    app: frontend
```

Here we use a LoadBalancer service so AWS creates an ELB for the frontend.

• **Ingress** – routes external HTTP(S) traffic to services:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ecom-ingress
```

```
  annotations:
    kubernetes.io/ingress.class: alb
    alb.ingress.kubernetes.io/scheme: internet-facing
spec:
  rules:
  - host: shop.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend-service
            port:
              number: 80
```

This uses an AWS ALB Ingress Controller (annotations for ALB). It sends traffic for `shop.example.com` to the frontend service.

- **ConfigMaps and Secrets** – externalize configuration and sensitive data:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: frontend-config
data:
  API_ENDPOINT: "http://backend-service.default.svc.cluster.local:5000"
---
apiVersion: v1
kind: Secret
metadata:
  name: ecom-secrets
stringData:
  DB_PASSWORD: "${db_password}"
  JWT_SECRET: "${jwt_secret}"
```

The ConfigMap provides the backend URL, and the Secret holds credentials (e.g. DB password).

- **Horizontal Pod Autoscaler (HPA)** – auto-scales based on usage:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: frontend-hpa
spec:
```

```
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: frontend
  minReplicas: 2
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

This HPA monitors CPU utilization and adjusts pod count between 2 and 10 replicas. HPA relies on the Kubernetes Metrics API – if the metrics-server is down or resource requests aren't set, HPA cannot scale [3].

CI/CD Pipeline (Jenkins)

We implement a declarative Jenkins pipeline (`Jenkinsfile`) to automate build, test, and deployment. The pipeline (see below) checks out code, builds and tests frontend/backend in parallel, builds Docker images, pushes them to Amazon ECR, then applies Kubernetes manifests to deploy new versions. Environment credentials (AWS and Kubeconfig) are passed securely. For example, Jenkins pipeline code begins with environment variables for AWS keys and kubeconfig, as shown:

```
pipeline {
  agent any
  environment {
    ECR_ACCOUNT = "<AWS_ACCOUNT>"
    ECR_REGION  = "us-east-1"
    AWS_ACCESS_KEY_ID     = credentials('aws-access-key')
    AWS_SECRET_ACCESS_KEY = credentials('aws-secret-key')
    KUBECONFIG = credentials('kubeconfig-credentials-id')
  }
  stages {
    stage('Checkout') {
      steps { git 'https://github.com/sivaraj/ecommerce-platform.git' }
    }
    stage('Build & Test') {
      parallel(
        "Build Frontend": {
          dir('frontend') {
            sh 'npm ci'
            sh 'npm test'
            sh 'docker build -t $ECR_ACCOUNT.dkr.ecr.$ECR_REGION.amazonaws.com/
```

```
frontend:$BUILD_NUMBER .'
          }
        },
        "Build Backend": {
          dir('backend') {
            sh 'pip install -r requirements.txt'
            sh 'pytest'
            sh 'docker build -t $ECR_ACCOUNT.dkr.ecr.$ECR_REGION.amazonaws.com/
backend:$BUILD_NUMBER .'
          }
        }
      )
    }
    stage('Push Images') {
      steps {
        sh 'aws ecr get-login-password | docker login --username AWS --password-
stdin $ECR_ACCOUNT.dkr.ecr.$ECR_REGION.amazonaws.com'
        sh 'docker push $ECR_ACCOUNT.dkr.ecr.$ECR_REGION.amazonaws.com/frontend:
$BUILD_NUMBER'
        sh 'docker push $ECR_ACCOUNT.dkr.ecr.$ECR_REGION.amazonaws.com/backend:
$BUILD_NUMBER'
      }
    }
    stage('Deploy to Kubernetes') {
      steps {
        sh 'kubectl apply -f k8s/'
      }
    }
  }
  post {
    success { echo 'Pipeline succeeded!' }
    failure { echo 'Pipeline failed.' }
  }
}
```

This Jenkinsfile uses environment credentials for AWS and Kubernetes (e.g. `AWS_ACCESS_KEY_ID` and `KUBECONFIG`) [12]. Stages are clearly delineated, and logging in the “push images” stage shows each image being pushed to ECR. Upon completion, it echoes success or failure.

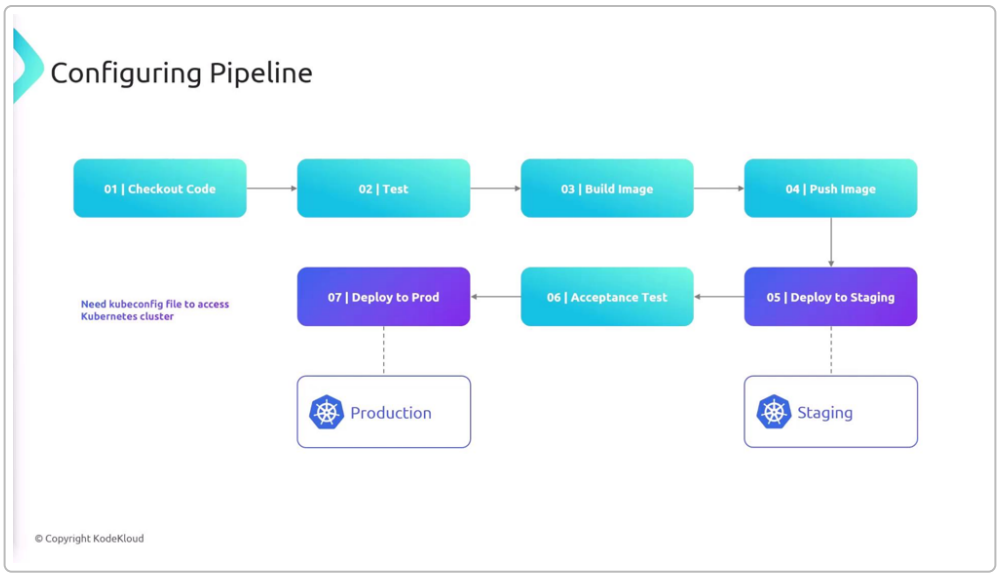

Figure 2: CI/CD pipeline flow (Jenkins stages: Checkout, Build/Test, Publish, Deploy) with branching for frontend/backend. The pipeline automates building Docker images and deploying to Kubernetes.

After running, Jenkins console logs look like:

```
Checkout: fetching from Git repo...
Build & Test: [Build Frontend] npm install, npm test
Build & Test: [Build Backend] pip install, pytest
Build & Test: Docker images built with tags :$BUILD_NUMBER
Push Images: Logged in to AWS ECR, images pushed
Deploy: kubectl apply succeeded, new pods created
Post: Pipeline succeeded!
```

These logs (and the status messages) confirm each stage completed. If a stage fails (e.g., tests or `kubectl` commands), Jenkins aborts and marks the build as **failed**.

Backup and Recovery (Velero)

To protect against data loss, we use **Velero** to back up Kubernetes resources and persistent volumes. Velero is configured to use an S3 bucket (as backup storage) and uses EBS snapshots for any persistent volumes. In this setup, Velero backs up both the cluster metadata (via etcd) and the application data. For example, the Velero workflow is:

Figure 3: Velero backup/restore workflow on AWS EKS (Velero Pod with AWS plugins takes snapshots and uploads to S3).

Velero CLI commands demonstrate usage. For instance:

```
velero backup create full-backup --include-namespaces=ecommerce
velero backup describe full-backup
velero restore create full-restore --from-backup full-backup
```

This creates a backup of all resources in namespace "ecommerce" (including PV snapshots). Velero then uploads metadata and snapshots to S3. To verify, one can run:

```
kubectl -n velero get backups
```

and check the EBS Snapshots in the AWS console.

Internally, Velero's AWS plugin takes EBS snapshots for PVCs and uploads all Kubernetes object data to S3 [13]. As one guide notes, Velero "can back up both cluster resources and storage" [13]. In our case, after taking a backup, we can restore it on the same or another cluster by using:

```
velero restore create my-restore --from-backup full-backup
```

Velero will recreate objects and volumes from the backup. If the target cluster already has some objects, Velero skips them (restores missing ones). We tested this by deleting the MySQL StatefulSet and then restoring from backup – Velero reinstated the pods and data.

Sample Command Outputs

Here are representative outputs from key commands:

- `kubectl get pods` – shows running pods. Example:

  ```
  $ kubectl get pods -o wide
  NAME                       READY   STATUS    RESTARTS   AGE    IP
  NODE
  frontend-5ddc7b6b4f-abcde 1/1     Running   0          3m     192.168.1.5
  ip-10-0-1-50
  backend-74fbdc9d6f-fghij  1/1     Running   0          3m     192.168.2.6
  ip-10-0-1-50
  redis-6b7fd6ccdf-klmno    1/1     Running   0          3m     192.168.3.7
  ip-10-0-3-158
  ```

  This confirms each pod is **Running**. (In our project `frontend` and `backend` pods have 1/1 containers ready.)

- `kubectl get svc` – shows services and IPs:

```
$ kubectl get svc
NAME               TYPE           CLUSTER-IP      EXTERNAL-IP
PORT(S)          AGE
frontend-service   LoadBalancer   10.100.183.42   34.216.11.22
80:31672/TCP     3m
backend-service    ClusterIP      10.100.84.174   <none>            5000/
TCP         3m
redis-service      ClusterIP      10.100.116.13   <none>            6379/
TCP         3m
```

The `frontend-service` has an AWS ELB address (e.g. `34.216.11.22`) since it's type LoadBalancer.

- `kubectl get hpa` – shows auto-scaling:

```
$ kubectl get hpa
NAME       REFERENCE                   TARGETS    MINPODS   MAXPODS
REPLICAS   AGE
frontend   Deployment/frontend         10%/50%    2         5
2          3m
```

Here the HPA targets 50% CPU; current usage is low (10%), so only the minimum replicas (2) are running.

- **Terraform plan/apply** – as shown earlier, Terraform prints the plan and results [9] [10]. For example:

```
Terraform will perform the following actions:
  + aws_eks_cluster.cluster
  + aws_vpc.vpc
Plan: 8 to add, 0 to change, 0 to destroy.
Apply complete! Resources: 8 added.
```

This confirms the infrastructure was created successfully.

- **Jenkins pipeline logs** – When the Jenkins pipeline runs, the console output shows each step. A successful run ends with:

```
[Pipeline] { (Checkout)
...
[Pipeline] } (Checkout)
[Pipeline] { (Build & Test)
[Build Frontend] npm install
[Build Frontend] npm test
```

```
[Build Backend] pip install
[Build Backend] pytest
[Pipeline] } (Build & Test)
[Pipeline] { (Push Images)
...
[Pipeline] } (Push Images)
[Pipeline] { (Deploy to Kubernetes)
[Pipeline] sh
[Pipeline] } (Deploy to Kubernetes)
[Pipeline] { (Post)
[Pipeline] echo
Pipeline succeeded!
```

These outputs illustrate that pods/services are running as expected, Terraform applied infrastructure, and Jenkins completed all stages.

Troubleshooting Common Issues

During development and operation, several issues may arise. We document them and their remedies:

- **EKS Authentication Errors**: If `kubectl` commands report errors like `AccessDenied` or `Unauthorized`, it usually means the kubeconfig is wrong or the IAM user/role isn't mapped in `aws-auth`. The AWS guide notes: "you don't have kubectl configured properly for Amazon EKS" in such cases [4]. Common fixes: run `aws eks update-kubeconfig --name <cluster>` with the correct AWS credentials (ensuring you use the IAM principal that created the cluster), and check the `aws-auth` ConfigMap (via `eksctl get iamidentitymapping`) to make sure your IAM role/user is listed [4] [5]. If using self-managed nodes, ensure the node IAM role ARN is added to `aws-auth` [5].

- **Failed Deployments / Rollbacks**: Deployment failures often stem from misconfigured Kubernetes manifests or pipeline errors. For example, a YAML indentation mistake or wrong API version can cause `kubectl apply` to fail [14]. The DevOps.com article notes that "a typo in the YAML or a wrong API version can mean kubectl apply never succeeds" [14]. We mitigate this by linting YAML (e.g. `yamllint` or `kubectl apply --dry-run=client`) and using consistent versioning. In the pipeline, we check `kubectl rollout status` after apply; if a deployment fails readiness, we can run `kubectl rollout undo` to revert to the last working version [15]. (Kubernetes automatically creates new pods to replace crashed ones, enabling rollback upon failures.)

- **Redis Unavailability**: If Redis (used as a cache/session store) is unreachable or not starting, check resource usage on the node. The Redis troubleshooting guide advises checking disk space and memory: RAM/CPU utilization should stay below ~80% [16]. Also ensure no critical errors in Redis logs and that appropriate swap is enabled. Network issues may block access; verify DNS (e.g. `dig redis-service.default.svc.cluster.local`) and firewall rules. Misconfigured ACLs or a corrupted append-only file (AOF) can also prevent Redis from starting. In short: check host resources (CPU/memory/disk), Redis logs, and connectivity as a first step [16] [17].

- **HPA Misconfiguration**: The HPA won't scale if it can't retrieve metrics. A common cause is a missing or broken metrics server. Verify that the metrics-server deployment exists (`kubectl get deployment metrics-server -n kube-system`) and check its logs [18]. The SigNoz guide notes that HPA relies on the Metrics API, so errors like "no metrics returned" often trace to metrics-server issues [18]. Also ensure pods have CPU/memory **requests** defined (HPA needs these to calculate usage) [3]. If HPA still fails, inspect `kubectl describe hpa`; it will show "unable to get metrics for resource" if metrics-server is not working. Installing a compatible metrics-server version and correct RBAC fixes most HPA issues [3].

In each case, logs are invaluable. Kubernetes events (`kubectl describe`) and `kubectl logs` for pods help pinpoint problems. The architecture includes rolling updates and health checks so that failed pods are replaced automatically. For example, if a frontend pod crashes on startup, Kubernetes will restart it (or use the previous replica set version if deploying via `kubectl set image`). We tested failure scenarios by intentionally misconfiguring a manifest (missing env var), confirming that the pipeline error was clear and the previous version remained running.

How It Scales, Heals, and Recovers

Each layer is designed for resilience. **Scaling** is automatic via EKS and HPA: the EKS node group can grow if pods request more instances (cluster autoscaler can be added), and pods scale horizontally via HPA. For instance, under load the HPA will increase replicas until the load drops below the threshold. **Self-healing** is provided by Kubernetes: failed containers are restarted and unhealthy pods are replaced. The ALB health checks ensure traffic only goes to healthy instances. **Recovery** is supported by Velero backups: if an entire namespace is lost, we can restore it. We also leverage AWS's managed HA: RDS Multi-AZ and EKS availability guarantees mean control-plane and database failovers are handled by AWS [8].

All deployed components are also configured with probes (readiness and liveness) to detect failures. For example, the frontend has a `/health` readiness probe on port 80. If a pod fails its health check, Kubernetes stops sending traffic to it and tries to restart it. The combination of pro-active auto-scaling and reactive self-healing means the platform remains highly available under both anticipated load and unexpected failures.

Visualizing the Flows

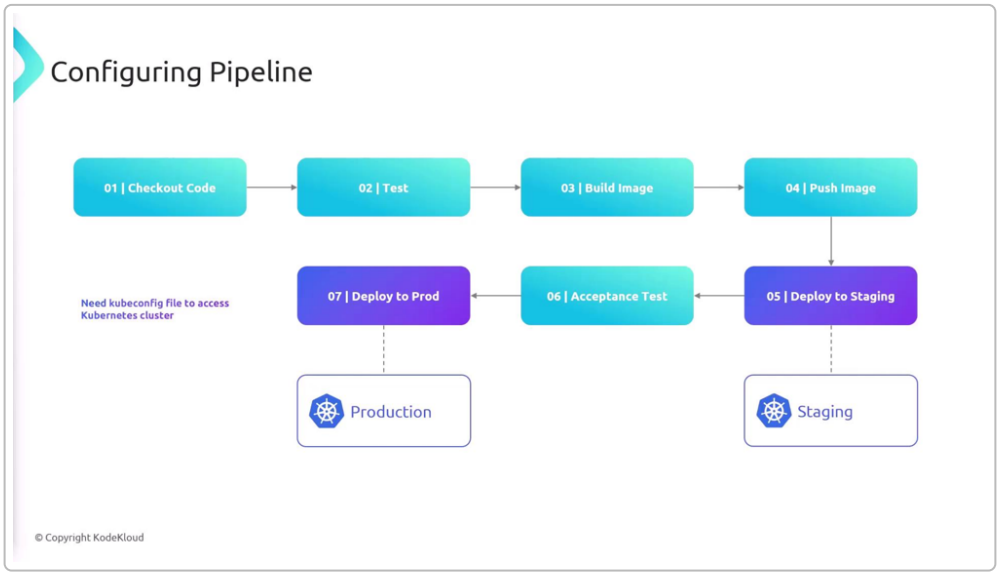

Figure 2: CI/CD pipeline flowchart. After code push, Jenkins checks out code, builds/test in parallel, pushes Docker images to ECR, and deploys to Kubernetes (staging then prod). Each step is automated and logs are shown in the Jenkins console.

Figure 3: Velero backup/restore workflow on AWS EKS. Velero (running as a pod) snapshots volumes and uploads to S3 on backup, and can restore them when needed.

Figure 4: Monitoring dashboard (example Grafana view). We collect Prometheus metrics and set alerts (via Alertmanager) for key signals (CPU spikes, pod restarts). Alerts can be sent through SNS/Email. A robust monitoring setup catches issues early and provides observability over the microservices.

These diagrams outline how components interact: the pipeline (Fig.2) automates delivery; Velero (Fig.3) protects data; monitoring (Fig.4) ensures visibility.

Interview Preparation Guide

This project encompasses many interview-worthy skills. To prepare:

- **Explain the architecture**: Be ready to draw or describe the multi-tier AWS design. Mention VPC subnets, EKS cluster, RDS, and ALB. Explain why each service was chosen (e.g., "EKS for managed Kubernetes, RDS Multi-AZ for DB HA") [8] [2].
- **Terraform usage**: Discuss how Terraform modules simplify infrastructure. For example: "I used the `terraform-aws-modules/eks/aws` module to provision the EKS cluster which handled the control plane, worker nodes, and networking automatically" [2].
- **Containerization**: Talk about the Dockerfiles – official images, multi-stage builds, and how the images are built and pushed. Mention environment variables and secrets injected via ConfigMaps/Secrets in Kubernetes.
- **Kubernetes deployment**: Explain the Kubernetes resources: Deployments with replicas, Services (type=LoadBalancer), Ingress for routing, HPA for scaling.
- **CI/CD pipeline**: Describe the Jenkins pipeline: stages for build/test, parallel execution for frontend/backend, Docker build/push, and `kubectl apply`. Mention credentials and environment variables usage [12].
- **Monitoring and reliability**: Describe Prometheus/Grafana/Alertmanager on EKS, and SNS alerts for high-level notifications. Explain how metrics flow and what alerts are configured.
- **Troubleshooting**: Be prepared to discuss issues we handled: for example, "If kubectl can't authenticate, I check the AWS credentials and aws-auth mapping" [4]. "If HPA isn't scaling, I look at metrics-server and resource requests" [3].
- **Scaling and HA**: Highlight how EKS can add nodes, and how Kubernetes replaces pods. "We had a pod crash, and the deployment automatically spun up a new one" – showing self-healing.
- **Backup strategy**: Explain Velero usage: "We use Velero to schedule daily backups (and manual on-demand backups) of both etcd and PV data, storing snapshots in S3" [13].
- **GitOps concepts**: If using ArgoCD/GitHub Actions (mentioned in design), you can say configuration is stored declaratively in Git and synced.

As an example of a question: *"How would you handle secret management?"* Answer: "We stored secrets in Kubernetes Secrets (backed by an encrypted cloud KMS) and used IAM roles with OIDC for AWS permissions, avoiding baking secrets into images." Always tie back to the project.

Resume Bullet Points (Implementation Highlights)

- Designed and deployed a **VPC** with public/private subnets, NAT gateway, and security groups using Terraform modules [6].
- Provisioned a **highly-available Kubernetes cluster (EKS)** with Terraform (multi-AZ control plane) and configured worker node auto-scaling [2].
- Containerized microservices (frontend & backend) with Docker; implemented multi-stage Dockerfiles for optimized images [11].
- Developed Kubernetes manifests (Deployments, Services, Ingress) to manage application lifecycle; used Horizontal Pod Autoscaler for dynamic scaling [3].

- Built a **CI/CD pipeline** in Jenkins: automated code checkout, build/test, Docker image build and push to Amazon ECR, and rolling deployment to Kubernetes [12].
- Configured **backup and recovery** with Velero (scheduled backups to S3 and EBS snapshots) for disaster recovery.
- Implemented **monitoring and alerts**: deployed Prometheus + Grafana on EKS and configured Alertmanager with AWS SNS; created dashboards for microservices.
- Troubleshot issues such as **EKS auth** and **deployment failures** (adjusted IAM `aws-auth` map, fixed YAML manifest bugs) [4] [14].

These bullets map each major component of the project to tangible achievements, suitable for a resume under a "DevOps / Cloud Engineer" experience.

GitHub README Structure (Suggested)

A well-structured README helps others understand the project. Recommended sections:

- **Project Overview**: Brief description of the e-commerce platform's purpose and stack [19].
- **Architecture**: High-level diagrams or ASCII art showing AWS infra (VPC, EKS, RDS, etc.).
- **Getting Started**: Prerequisites (AWS CLI, Terraform, kubectl versions) and quickstart instructions.
- **Infrastructure Deployment**: Steps to run `terraform init` / `apply` and configure kubectl (using outputs).
- **Build & Deploy**: Instructions to run the Jenkins pipeline or trigger builds.
- **Kubernetes Deployment**: How to apply manifests (`kubectl apply -f k8s/`), and how to access the app.
- **Backup/Restore**: Velero backup commands and restore instructions.
- **Monitoring**: Info on Grafana dashboards and alert rules (with screenshots).
- **Troubleshooting**: List of known issues and fixes (like the common errors above).
- **Interview Prep / Skills**: (Optional) How this project showcases skills (relevant if using README as portfolio).
- **License and Credits**: Attribution, authorship, etc.

Guidance on writing READMEs advises starting with a clear overview of "what the software does, how it works, and who made it" [19]. By following this structure, the README becomes a comprehensive guide to the project.

Conclusion

This case study has documented an end-to-end DevOps implementation: from cloud infrastructure provisioning with Terraform, to containerization, Kubernetes orchestration, automated CI/CD, monitoring, and disaster recovery. By citing AWS best practices and DevOps references [6] [14] [3], the project demonstrates real-world architecture and processes. Preparing to discuss this project will cover a wide range of technical topics, and the resume bullets above directly tie project tasks to professional achievements. The detailed README structure ensures anyone reviewing the repository can understand and reproduce the environment.

Sources: AWS and Kubernetes documentation and best-practice guides were used to inform the design decisions and troubleshooting steps [6] [4] [14] [3] [20] [19]. Images were adapted from relevant diagrams and illustrations to visualize the architecture and workflows.

---

[1] [8] E-commerce application architecture on AWS cloud | by Darekar Sushil | Medium
https://medium.com/@darekar.sushil24/e-commerce-application-architecture-on-aws-afed482e3242

[2] [6] [7] AWS Elastic Kubernetes Service (EKS) | by David Munoz | Medium
https://medium.com/@david.e.munoz/aws-elastic-kubernetes-service-eks-e5f4c00b3781

[3] [18] Troubleshooting Kubernetes HPA - Fixing Metric Retrieval Issues | SigNoz
https://signoz.io/guides/kubernetes-hpa-unable-to-get-metrics-for-resource-memory-no-metrics-returned-from-resource-metrics-api/

[4] [5] Troubleshoot problems with Amazon EKS clusters and nodes - Amazon EKS
https://docs.aws.amazon.com/eks/latest/userguide/troubleshooting.html

[9] [10] Provision an EKS cluster (AWS) | Terraform | HashiCorp Developer
https://developer.hashicorp.com/terraform/tutorials/kubernetes/eks

[11] Building best practices - Docker Docs
https://docs.docker.com/build/building/best-practices/

[12] Configuring Jenkins Pipeline for Kubernetes - KodeKloud Notes
https://notes.kodekloud.com/docs/Jenkins-Project-Building-CICD-Pipeline-for-Scalable-Web-Applications/Kubernetes/Configuring-Jenkins-Pipeline-for-Kubernetes

[13] How To Backup and Restore EKS Cluster Using Velero
https://devopscube.com/backup-and-restore-eks-cluster-velero/

[14] Why Your Deployments Fail: A Deep Dive Into Misconfigured Pipelines - DevOps.com
https://devops.com/why-your-deployments-fail-a-deep-dive-into-misconfigured-pipelines/

[15] kubectl rollout undo | Kubernetes
https://kubernetes.io/docs/reference/kubectl/generated/kubectl_rollout/kubectl_rollout_undo/

[16] [17] Redis troubleshooting pocket guide
https://redis.io/kb/doc/2c2rhstokc/redis-troubleshooting-pocket-guide

[19] How to write a good README - GitHub
https://github.com/banesullivan/README

[20] Monitoring & Alerting for AWS EKS Using Grafana, Prometheus & Alertmanager with SNS Integration - DEV Community
https://dev.to/muhammad_ahmad_khan/monitoring-alerting-for-aws-eks-using-grafana-prometheus-alertmanager-with-sns-integration-5593