AWS EKS Deployment Guide for Ecommerce Microservices

This guide covers the complete process for deploying the ecommerce microservices application to AWS EKS using Terraform. It demonstrates enterprise-level cloud architecture and DevOps practices essential for a senior role in cloud engineering.

> **Architectural Note:** This deployment uses containerized solutions for all services (PostgreSQL, Kafka, Zookeeper) rather than AWS managed services (RDS, MSK), offering greater cost optimization and configuration flexibility.

Prerequisites

- AWS CLI configured with appropriate permissions
- Terraform (v1.0.0+)
- Docker
- kubectl
- AWS account with permissions to create:
  - VPC and networking resources
  - EKS clusters
  - ECR repositories
  - IAM roles and policies

Deployment Process Overview

1. Build and push Docker images to AWS ECR
2. Provision AWS infrastructure with Terraform
3. Configure kubectl to connect to the EKS cluster
4. Deploy Kubernetes resources to the EKS cluster
5. Verify the deployment

1. Building and Pushing Docker Images

1.1 Build All Docker Images

Build all the required Docker images for your microservices architecture:

1.1.1 Build the Spring Boot Application

```
# Navigate to the product-service directory
cd <SOURCE_DIR>\twelve-factor\ecommerce-microservices\product-service

# Run Maven build
./mvnw clean package -DskipTests

# Build the Docker image
docker build -t ecommerce-product-service:latest .
```

1.1.2 Build or Pull Kafka-Related Images

You can either build custom Kafka images with your configurations or pull and tag official images:

```
# Pull Kafka, Zookeeper, and Kafka UI images
docker pull confluentinc/cp-kafka:7.3.0
docker pull confluentinc/cp-zookeeper:7.3.0
docker pull provectuslabs/kafka-ui:latest

# Tag them for your project
docker tag confluentinc/cp-kafka:7.3.0 ecommerce-kafka:latest
docker tag confluentinc/cp-zookeeper:7.3.0 ecommerce-zookeeper:latest
docker tag provectuslabs/kafka-ui:latest ecommerce-kafka-ui:latest
```

1.2 Create ECR Repositories via Terraform

The ECR repositories for all services are defined in our Terraform configuration:

```
# Navigate to the Terraform production environment directory
cd <SOURCE_DIR>\twelve-factor\ecommerce-microservices\terraform\environments\prod

# Initialize Terraform
terraform init

# Create ECR repositories for all services
```

```
terraform apply -target=module.product_service_ecr -target=module.kafka_ecr -
target=module.zookeeper_ecr -target=module.kafka_ui_ecr
```

1.3 Push Docker Images to ECR

```
# Get the ECR repository URLs from Terraform output
PRODUCT_ECR=$(terraform output -raw product_service_ecr_repository_url)
KAFKA_ECR=$(terraform output -raw kafka_ecr_repository_url)
ZOOKEEPER_ECR=$(terraform output -raw zookeeper_ecr_repository_url)
KAFKA_UI_ECR=$(terraform output -raw kafka_ui_ecr_repository_url)

# Log in to ECR
aws ecr get-login-password --region us-west-2 | docker login --username AWS --
password-stdin $(echo $PRODUCT_ECR | cut -d'/' -f1)

# Tag and push the Product Service image
docker tag ecommerce-product-service:latest $PRODUCT_ECR:latest
docker push $PRODUCT_ECR:latest

# Tag and push Kafka image
docker tag ecommerce-kafka:latest $KAFKA_ECR:latest
docker push $KAFKA_ECR:latest

# Tag and push Zookeeper image
docker tag ecommerce-zookeeper:latest $ZOOKEEPER_ECR:latest
docker push $ZOOKEEPER_ECR:latest

# Tag and push Kafka UI image
docker tag ecommerce-kafka-ui:latest $KAFKA_UI_ECR:latest
docker push $KAFKA_UI_ECR:latest
```

2. Provisioning AWS Infrastructure

2.1 Create terraform.tfvars File

Create a `terraform.tfvars` file to customize your deployment:

```
# Create terraform.tfvars in the prod environment directory
cat > terraform.tfvars << EOF
aws_region       = "us-west-2"
project_name     = "ecommerce"
environment      = "prod"
eks_cluster_name = "ecommerce-eks-cluster"
db_password      = "YourSecurePasswordHere"  # Use AWS Secrets Manager in
```

```
production
eks_instance_types = ["t3.large"]
eks_desired_size = 3
eks_min_size     = 3
eks_max_size     = 6
EOF
```

2.2 Deploy AWS Infrastructure

```
# Apply the Terraform configuration to create all infrastructure
terraform apply
```

This will create:

- VPC with public and private subnets
- EKS cluster with worker nodes
- ECR repository (if not created in step 1.2)
- Kubernetes namespace, ConfigMaps, and Secrets
- Storage classes and persistent volume claims

3. Configure kubectl for EKS

After the infrastructure deployment completes, configure kubectl to communicate with your new EKS cluster:

```
# Use the command provided in Terraform output
$(terraform output -raw kubernetes_config_command)

# Verify the connection
kubectl get nodes
```

4. Deploying Kubernetes Resources

4.1 Create Kubernetes Deployment YAML Files

Create Kubernetes deployment files for each component of your microservices architecture:

4.1.1 PostgreSQL Deployment

```
cat > postgres-deployment.yaml << EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
  namespace: ecommerce
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
      - name: postgres
        image: postgres:14
        ports:
        - containerPort: 5432
        env:
        - name: POSTGRES_USER
          valueFrom:
            secretKeyRef:
              name: product-service-secrets
              key: SPRING_DATASOURCE_USERNAME
        - name: POSTGRES_PASSWORD
          valueFrom:
            secretKeyRef:
              name: product-service-secrets
              key: SPRING_DATASOURCE_PASSWORD
        - name: POSTGRES_DB
          value: product_db
        volumeMounts:
        - name: postgres-data
          mountPath: /var/lib/postgresql/data
        resources:
          requests:
            memory: "512Mi"
            cpu: "500m"
          limits:
            memory: "1Gi"
            cpu: "1000m"
        livenessProbe:
          exec:
            command:
            - pg_isready
            - -U
```

```
            - postgres
          initialDelaySeconds: 30
          periodSeconds: 10
      volumes:
      - name: postgres-data
        persistentVolumeClaim:
          claimName: postgres-data
---
apiVersion: v1
kind: Service
metadata:
  name: postgres-service
  namespace: ecommerce
spec:
  selector:
    app: postgres
  ports:
  - port: 5432
    targetPort: 5432
  type: ClusterIP
EOF
```

4.1.2 Zookeeper Deployment

```
cat > zookeeper-deployment.yaml << EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: zookeeper
  namespace: ecommerce
spec:
  replicas: 1
  selector:
    matchLabels:
      app: zookeeper
  template:
    metadata:
      labels:
        app: zookeeper
    spec:
      containers:
      - name: zookeeper
        image: confluentinc/cp-zookeeper:7.3.0
        ports:
        - containerPort: 2181
        env:
        - name: ZOOKEEPER_CLIENT_PORT
          value: "2181"
        - name: ZOOKEEPER_TICK_TIME
          value: "2000"
        resources:
          requests:
            memory: "512Mi"
            cpu: "500m"
```

```
          limits:
            memory: "1Gi"
            cpu: "1000m"
---
apiVersion: v1
kind: Service
metadata:
  name: zookeeper-service
  namespace: ecommerce
spec:
  selector:
    app: zookeeper
  ports:
  - port: 2181
    targetPort: 2181
  type: ClusterIP
EOF
```

4.1.3 Kafka Deployment

```
cat > kafka-deployment.yaml << EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kafka
  namespace: ecommerce
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kafka
  template:
    metadata:
      labels:
        app: kafka
    spec:
      containers:
      - name: kafka
        image: confluentinc/cp-kafka:7.3.0
        ports:
        - containerPort: 9092
        env:
        - name: KAFKA_BROKER_ID
          value: "1"
        - name: KAFKA_ZOOKEEPER_CONNECT
          value: "zookeeper-service:2181"
        - name: KAFKA_LISTENER_SECURITY_PROTOCOL_MAP
          value: "PLAINTEXT:PLAINTEXT,PLAINTEXT_INTERNAL:PLAINTEXT"
        - name: KAFKA_ADVERTISED_LISTENERS
          value: "PLAINTEXT://kafka-service:9092,PLAINTEXT_INTERNAL://kafka:29092"
        - name: KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR
          value: "1"
        - name: KAFKA_TRANSACTION_STATE_LOG_MIN_ISR
          value: "1"
```

```
        - name: KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR
          value: "1"
        resources:
          requests:
            memory: "1Gi"
            cpu: "500m"
          limits:
            memory: "2Gi"
            cpu: "1000m"
---
apiVersion: v1
kind: Service
metadata:
  name: kafka-service
  namespace: ecommerce
spec:
  selector:
    app: kafka
  ports:
  - port: 9092
    targetPort: 9092
  type: ClusterIP
EOF
```

4.1.4 Kafka UI Deployment

```
cat > kafka-ui-deployment.yaml << EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kafka-ui
  namespace: ecommerce
spec:
  replicas: 1
  selector:
    matchLabels:
      app: kafka-ui
  template:
    metadata:
      labels:
        app: kafka-ui
    spec:
      containers:
      - name: kafka-ui
        image: provectuslabs/kafka-ui:latest
        ports:
        - containerPort: 8080
        env:
        - name: KAFKA_CLUSTERS_0_NAME
          value: "local"
        - name: KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS
          value: "kafka-service:9092"
        - name: KAFKA_CLUSTERS_0_ZOOKEEPER
          value: "zookeeper-service:2181"
```

```
        resources:
          requests:
            memory: "256Mi"
            cpu: "200m"
          limits:
            memory: "512Mi"
            cpu: "500m"
---
apiVersion: v1
kind: Service
metadata:
  name: kafka-ui-service
  namespace: ecommerce
spec:
  selector:
    app: kafka-ui
  ports:
  - port: 80
    targetPort: 8080
  type: ClusterIP
EOF
```

4.1.5 Product Service Deployment

```
cat > product-service-deployment.yaml << EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: product-service
  namespace: ecommerce
spec:
  replicas: 2
  selector:
    matchLabels:
      app: product-service
  template:
    metadata:
      labels:
        app: product-service
    spec:
      containers:
      - name: product-service
        image: ${ECR_REPO}:latest
        ports:
        - containerPort: 8080
        envFrom:
        - configMapRef:
            name: product-service-config
        - secretRef:
            name: product-service-secrets
        resources:
          requests:
            memory: "512Mi"
            cpu: "500m"
```

```
          limits:
            memory: "1Gi"
            cpu: "1000m"
        readinessProbe:
          httpGet:
            path: /actuator/health
            port: 8080
          initialDelaySeconds: 60
          periodSeconds: 10
        livenessProbe:
          httpGet:
            path: /actuator/health
            port: 8080
          initialDelaySeconds: 120
          periodSeconds: 30
---
apiVersion: v1
kind: Service
metadata:
  name: product-service-service
  namespace: ecommerce
spec:
  selector:
    app: product-service
  ports:
  - port: 80
    targetPort: 8080
  type: ClusterIP
EOF
```

4.1.6 Ingress for External Access

```
cat > ingress.yaml << EOF
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ecommerce-ingress
  namespace: ecommerce
  annotations:
    kubernetes.io/ingress.class: alb
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/target-type: ip
spec:
  rules:
  - http:
      paths:
      - path: /api/v1/products
        pathType: Prefix
        backend:
          service:
            name: product-service-service
            port:
              number: 80
      - path: /kafka-ui
```

```
        pathType: Prefix
        backend:
          service:
            name: kafka-ui-service
            port:
              number: 80
EOF
```

4.2 Deploy AWS Load Balancer Controller

For the Ingress to work with AWS ALB, deploy the AWS Load Balancer Controller:

```
# Add the EKS chart repo
helm repo add eks https://aws.github.io/eks-charts

# Install the AWS Load Balancer Controller
helm install aws-load-balancer-controller eks/aws-load-balancer-controller \
  -n kube-system \
  --set clusterName=ecommerce-eks-cluster \
  --set serviceAccount.create=false \
  --set serviceAccount.name=aws-load-balancer-controller
```

4.3 Deploy Kubernetes Resources

```
# Deploy all Kubernetes resources
kubectl apply -f postgres-deployment.yaml
kubectl apply -f zookeeper-deployment.yaml
kubectl apply -f kafka-deployment.yaml
kubectl apply -f kafka-ui-deployment.yaml
kubectl apply -f product-service-deployment.yaml
kubectl apply -f ingress.yaml
```

5. Verify the Deployment

5.1 Check Pod Status

```
kubectl get pods -n ecommerce
```

5.2 Check Service Status

```
kubectl get svc -n ecommerce
```

5.3 Check Ingress Status

```
kubectl get ingress -n ecommerce
```

5.4 Access the Application

Once the AWS ALB provisioning is complete (may take a few minutes), you can access:

- Product Service API: `http://<ALB-DNS-NAME>/api/v1/products`
- Kafka UI: `http://<ALB-DNS-NAME>/kafka-ui`

Get the ALB DNS name with:

```
kubectl get ingress -n ecommerce -o
jsonpath='{.items[0].status.loadBalancer.ingress[0].hostname}'
```

6. Monitoring and Troubleshooting

6.1 View Application Logs

```
# View Product Service logs
kubectl logs -f deployment/product-service -n ecommerce

# View PostgreSQL logs
kubectl logs -f deployment/postgres -n ecommerce

# View Kafka logs
kubectl logs -f deployment/kafka -n ecommerce
```

6.2 Access Kubernetes Dashboard (Optional)

You can deploy the Kubernetes Dashboard for a graphical interface:

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/recommende
d.yaml

# Create a service account with admin permissions
kubectl apply -f - <<EOF
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin-user
  namespace: kubernetes-dashboard
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: admin-user
  namespace: kubernetes-dashboard
EOF

# Get the token for logging in
kubectl -n kubernetes-dashboard create token admin-user

# Start the kubectl proxy
kubectl proxy
```

The Kubernetes Dashboard is accessible at: http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/

7. Cleaning Up Resources

To avoid incurring unnecessary AWS costs, clean up resources when they're not needed:

```
# Navigate to Terraform directory
cd <SOURCE_DIR>\twelve-factor\ecommerce-microservices\terraform\environments\prod

# Destroy all resources created by Terraform
terraform destroy
```

Conclusion

This deployment guide demonstrates a complete enterprise-grade approach to deploying Spring Boot microservices to AWS EKS. It leverages infrastructure as code with Terraform and follows cloud-native best practices, which are key skills required for a Senior DevOps/Cloud Engineer position.

The approach ensures:

- Infrastructure reproducibility through Terraform
- Container image management with ECR
- Proper Kubernetes resource configuration
- Secure database credential management
- Load balancing and external access configuration
- Complete monitoring and troubleshooting capabilities

By following these steps, we implement a production-ready Kubernetes deployment on AWS, meeting industry best practices for cloud-native applications.

Benefits of Your Containerized Approach:

1. **Cost Optimization** : Running containerized services can be 40-60% cheaper than using AWS managed services - highlighting your financial awareness for enterprise environments
2. **Configuration Flexibility** : Containers allow more granular configuration than managed services - demonstrating your technical depth
3. **Portability** : Your architecture can now move between cloud providers with minimal changes - showing your strategic thinking

4. **Unified Management** : All services are managed through Kubernetes - streamlining operations