# Step-By-Step AWS EKS Deployment Guide for Ecommerce Microservices

This enterprise-grade deployment guide outlines the complete process for deploying containerized ecommerce microservices to AWS EKS using Terraform. This approach demonstrates production-ready best practices and cloud architecture expertise.

# Phase 1: AWS Authentication & Environment Setup

## Step 1: Configure AWS CLI

```
# Install AWS CLI if not already installed
# https://aws.amazon.com/cli/

# Configure AWS credentials with appropriate permissions
aws configure

# Enter your:
# - AWS Access Key ID
# - AWS Secret Access Key
# - Default region (recommend using us-west-2 as in our config)
# - Output format (json recommended)
```

## Step 2: Set Up Terraform Remote State with S3 and DynamoDB

```
# Create an S3 bucket for Terraform state
aws s3 mb s3://ecommerce-terraform-state-YOUR-ACCOUNT-ID

# Enable versioning on the bucket
aws s3api put-bucket-versioning --bucket ecommerce-terraform-state-YOUR-ACCOUNT-ID
--versioning-configuration Status=Enabled
```

```
# Create a DynamoDB table for state locking
aws dynamodb create-table --table-name ecommerce-terraform-locks \
    --attribute-definitions AttributeName=LockID,AttributeType=S \
    --key-schema AttributeName=LockID,KeyType=HASH \
    --billing-mode PAY_PER_REQUEST
```

## Step 3: Update Backend Configuration

Create a new file in your prod environment directory:

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

# Create backend.tf file
```

Add the following content to `backend.tf`:

```
terraform {
  backend "s3" {
    bucket         = "ecommerce-terraform-state-YOUR-ACCOUNT-ID"
    key            = "ecommerce/prod/terraform.tfstate"
    region         = "us-west-2"
    dynamodb_table = "ecommerce-terraform-locks"
    encrypt        = true
  }
}
```

# Phase 2: Pre-Deployment Security & Configuration

## Step 1: Create a Secure terraform.tfvars File

```
# Create terraform.tfvars in the prod environment
```

Contents of `terraform.tfvars`:

```
aws_region       = "us-west-2"
project_name     = "ecommerce"
environment      = "prod"
eks_cluster_name = "ecommerce-eks"
db_password      = "YOUR-SECURE-PASSWORD"  # In production, use AWS Secrets Manager
eks_instance_types = ["t3.medium"]  # Adjust based on workload needs
eks_desired_size = 3
eks_min_size     = 2
eks_max_size     = 5
```

## Step 2: Create AWS KMS Key for EKS Secrets Encryption

```
# Create a KMS key for encrypting EKS secrets
aws kms create-key --description "EKS Secret Encryption Key for Ecommerce"

# Note the KeyId from the output
KMS_KEY_ID="YOUR-KMS-KEY-ID"
```

## Step 3: Update EKS Module Configuration for KMS Encryption

Add the following to your `environments/prod/main.tf`:

```
# Add this to the eks module call
encryption_config = [{
  provider_key_arn = "arn:aws:kms:us-west-2:YOUR-ACCOUNT-ID:key/$KMS_KEY_ID"
  resources        = ["secrets"]
}]
```

# Phase 3: Incremental Deployment

## Step 1: Initialize Terraform with the Remote Backend

```
# Initialize Terraform with the new backend configuration
terraform init
```

# Step 2: Create Network Infrastructure First

```
# Validate your VPC configuration
terraform validate

# Create network infrastructure first
terraform apply -target=module.vpc
```

# Step 3: Create ECR Repositories and Push Images

```
# Create ECR repositories
terraform apply -target=module.product_service_ecr -target=module.kafka_ecr -
target=module.zookeeper_ecr -target=module.kafka_ui_ecr -target=module.postgres_ecr
-auto-approve

# Get ECR repository URLs and store them in a more compatible way for different
shells
# For Bash/Linux environments
if [ -z "$OSTYPE" ] || [[ "$OSTYPE" == "linux-gnu"* ]] || [[ "$OSTYPE" == "darwin"*
]]; then
  PRODUCT_ECR=$(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=$(terraform output -raw kafka_ui_ecr_repository_url)
  POSTGRES_ECR=$(terraform output -raw postgres_ecr_repository_url)

  # Get ECR registry URL for login
  ECR_REGISTRY=$(echo $PRODUCT_ECR | cut -d'/' -f1)
else
  # For Windows PowerShell environments
  $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
  $POSTGRES_ECR = terraform output -raw postgres_ecr_repository_url

  # Get ECR registry URL for login
  $ECR_REGISTRY = $PRODUCT_ECR.Split('/')[0]
fi

# The following commands differ based on shell environment
```

```
# For Bash/Linux environments
if [ -z "$OSTYPE" ] || [[ "$OSTYPE" == "linux-gnu"* ]] || [[ "$OSTYPE" == "darwin"*
]]; then
  # Login to ECR
  aws ecr get-login-password --region us-west-2 | docker login --username AWS --
password-stdin $ECR_REGISTRY

  # Build and push the Spring Boot application image
  cd <SOURCE_DIR>/twelve-factor/ecommerce-microservices/product-service
  ./mvnw clean package -DskipTests
  docker build -t ecommerce-product-service:latest .
  docker tag ecommerce-product-service:latest $PRODUCT_ECR:latest
  docker push $PRODUCT_ECR:latest

  # Pull infrastructure images
  docker pull confluentinc/cp-kafka:7.3.0
  docker pull confluentinc/cp-zookeeper:7.3.0
  docker pull provectuslabs/kafka-ui:latest
  docker pull postgres:14

  # Tag all images with ECR repository URLs
  docker tag confluentinc/cp-kafka:7.3.0 $KAFKA_ECR:latest
  docker tag confluentinc/cp-zookeeper:7.3.0 $ZOOKEEPER_ECR:latest
  docker tag provectuslabs/kafka-ui:latest $KAFKA_UI_ECR:latest
  docker tag postgres:14 $POSTGRES_ECR:latest

  # Push all images to ECR
  docker push $KAFKA_ECR:latest
  docker push $ZOOKEEPER_ECR:latest
  docker push $KAFKA_UI_ECR:latest
  docker push $POSTGRES_ECR:latest

else
  # For Windows PowerShell environments
  # Login to ECR
  aws ecr get-login-password --region us-west-2 | docker login --username AWS --
password-stdin $ECR_REGISTRY

  # Build and push the Spring Boot application image
  cd <SOURCE_DIR>\twelve-factor\ecommerce-microservices\product-service
  .\mvnw clean package -DskipTests
  docker build -t ecommerce-product-service:latest .
  docker tag ecommerce-product-service:latest "$PRODUCT_ECR`:latest"
  docker push "$PRODUCT_ECR`:latest"

  # Pull infrastructure images
  docker pull confluentinc/cp-kafka:7.3.0
  docker pull confluentinc/cp-zookeeper:7.3.0
  docker pull provectuslabs/kafka-ui:latest
  docker pull postgres:14

  # Tag all images with ECR repository URLs - with proper variable expansion for
PowerShell
  docker tag confluentinc/cp-kafka:7.3.0 "$KAFKA_ECR`:latest"
  docker tag confluentinc/cp-zookeeper:7.3.0 "$ZOOKEEPER_ECR`:latest"
  docker tag provectuslabs/kafka-ui:latest "$KAFKA_UI_ECR`:latest"
```

```
  docker tag postgres:14 "$POSTGRES_ECR`:latest"

  # Push all images to ECR
  docker push "$KAFKA_ECR`:latest"
  docker push "$ZOOKEEPER_ECR`:latest"
  docker push "$KAFKA_UI_ECR`:latest"
  docker push "$POSTGRES_ECR`:latest"
fi
```

# Step 4: Deploy EKS Cluster

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

# Create the EKS cluster
terraform apply -target=module.eks
```

# Step 5: Configure kubectl for EKS Access

```
# Configure kubectl to access your EKS cluster
aws eks update-kubeconfig --name ecommerce-eks --region us-west-2

# Verify connection
kubectl get nodes
```

# Step 6: Deploy Storage Requirements First

```
# Apply the complete Terraform configuration for Kubernetes resources
terraform apply

# Wait for resources to be created
kubectl get namespace ecommerce
kubectl get pvc -n ecommerce
```

# Phase 4: Application Deployment

## Step 1: Create Kubernetes Deployment Files

Create deployment files for your applications using the ECR image URLs:

```
# Navigate to a directory for your K8s manifests
cd <SOURCE_DIR>\twelve-factor\ecommerce-microservices\kubernetes

# Retrieve ECR URLs again if needed
cd <SOURCE_DIR>\twelve-factor\ecommerce-microservices\terraform\environments\prod
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)
POSTGRES_ECR=$(terraform output -raw postgres_ecr_repository_url)

# Go back to Kubernetes directory
cd <SOURCE_DIR>\twelve-factor\ecommerce-microservices\kubernetes
```

# Step 2: Update Kubernetes Manifests with ECR Image URLs

### Option 1: Manual Update

Edit your deployment files to use ECR URLs in these files:

- `postgres-deployment.yaml`
- `zookeeper-deployment.yaml`
- `kafka-deployment.yaml`
- `kafka-ui-deployment.yaml`
- `product-service-deployment.yaml`

For example:

In `postgres-deployment.yaml`:

```
containers:
- name: postgres
  image: ${POSTGRES_ECR}:latest   # Replace with actual ECR URL
```

In `product-service-deployment.yaml`:

```
containers:
- name: product-service
```

```
    image: ${PRODUCT_ECR}:latest    # Replace with actual ECR URL
```

### Option 2: Automated Update (PowerShell)

Create a script to automatically update all Kubernetes deployment files with the correct ECR URLs:

```
# PowerShell script to update Kubernetes deployment files with ECR repository URLs

# Change to the Terraform directory to get outputs
Set-Location <SOURCE_DIR>\twelve-factor\ecommerce-
microservices\terraform\environments\prod

# Get ECR repository URLs
$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
$POSTGRES_ECR = terraform output -raw postgres_ecr_repository_url

Write-Host "ECR URLs retrieved:" -ForegroundColor Cyan
Write-Host "Product Service: $PRODUCT_ECR"
Write-Host "Kafka: $KAFKA_ECR"
Write-Host "Zookeeper: $ZOOKEEPER_ECR"
Write-Host "Kafka UI: $KAFKA_UI_ECR"
Write-Host "PostgreSQL: $POSTGRES_ECR"

# Change to the Kubernetes directory
Set-Location <SOURCE_DIR>\twelve-factor\ecommerce-microservices\kubernetes

# Update the deployment files with the correct image URLs
$files = @{
    "zookeeper.yaml" = @{
        searchString = "image: "
        replaceString = "image: $ZOOKEEPER_ECR`:latest"
    }
    "kafka.yaml" = @{
        searchString = "image: "
        replaceString = "image: $KAFKA_ECR`:latest"
    }
    "kafka-ui.yaml" = @{
        searchString = "image: "
        replaceString = "image: $KAFKA_UI_ECR`:latest"
    }
    "product-service.yaml" = @{
        searchString = "image: "
        replaceString = "image: $PRODUCT_ECR`:latest"
    }
    "postgres.yaml" = @{
        searchString = "image: "
        replaceString = "image: $POSTGRES_ECR`:latest"
    }
}
```

```
foreach ($file in $files.Keys) {
    if (Test-Path $file) {
        Write-Host "Updating $file..." -ForegroundColor Yellow
        $content = Get-Content $file
        $updated = $content -replace $files[$file].searchString,
$files[$file].replaceString
        Set-Content -Path $file -Value $updated
        Write-Host "Updated $file successfully" -ForegroundColor Green
    } else {
        Write-Host "File $file not found" -ForegroundColor Red
    }
}

Write-Host "All deployment files updated with ECR repository URLs!" -
ForegroundColor Green
```

Save this script as `update-k8s-images.ps1` in your Kubernetes directory and run it:

```
.\update-k8s-images.ps1
```

# Step 3: Deploy Services in Order

```
# Apply Kubernetes manifests in order
kubectl apply -f postgres-deployment.yaml
kubectl apply -f zookeeper-deployment.yaml

# Wait for Zookeeper to be ready
kubectl rollout status deployment/zookeeper -n ecommerce

kubectl apply -f kafka-deployment.yaml
kubectl rollout status deployment/kafka -n ecommerce

kubectl apply -f kafka-ui-deployment.yaml
kubectl apply -f product-service-deployment.yaml
```

# Step 4: Set Up AWS Load Balancer Controller for Ingress

### Option 1: Using Helm (if installed)

```
# Install AWS Load Balancer Controller via Helm
kubectl apply -k "github.com/aws/eks-charts/stable/aws-load-balancer-
controller//crds?ref=master"

helm repo add eks https://aws.github.io/eks-charts
helm repo update

helm install aws-load-balancer-controller eks/aws-load-balancer-controller \
  -n kube-system \
  --set clusterName=ecommerce-eks \
  --set serviceAccount.create=true \
  --set region=us-west-2 \
  --set vpcId=$(terraform output -raw vpc_id)
```

### Option 2: Direct Installation (PowerShell)

```
# PowerShell script to install AWS Load Balancer Controller using YAML directly

# Get cluster name from Terraform output
Set-Location <SOURCE_DIR>\twelve-factor\ecommerce-
microservices\terraform\environments\prod
$CLUSTER_NAME = terraform output -raw eks_cluster_name

# Download and modify the controller YAML
Invoke-WebRequest -Uri "https://github.com/kubernetes-sigs/aws-load-balancer-
controller/releases/download/v2.5.4/v2_5_4_full.yaml" -OutFile "aws-lb-
controller.yaml"

# Replace the cluster name placeholder
(Get-Content -Path "aws-lb-controller.yaml") -replace "your-cluster-name",
$CLUSTER_NAME | Set-Content -Path "aws-lb-controller.yaml"

# Apply the modified YAML file
kubectl apply -f aws-lb-controller.yaml
```

# Deploy ingress

kubectl apply -f ingress.yaml

# Wait for the ALB to be provisioned

kubectl get ingress -n ecommerce --watch

```
## Phase 5: Verification and Monitoring

### Step 1: Verify All Components

```bash
# Check pod status
kubectl get pods -n ecommerce

# Check services
kubectl get svc -n ecommerce

# Check ingress
kubectl get ingress -n ecommerce
```

# Step 2: Set Up CloudWatch for Monitoring

```
# Deploy AWS CloudWatch agent to EKS
kubectl apply -f https://raw.githubusercontent.com/aws-samples/amazon-cloudwatch-container-insights/latest/k8s-deployment-manifest-templates/deployment-mode/daemonset/container-insights-monitoring/quickstart/cwagent-fluentd-quickstart.yaml
```

# Step 3: Set Up Prometheus/Grafana (Optional)

```
# Add Prometheus Helm repo
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update

# Install Prometheus and Grafana
helm install prometheus prometheus-community/kube-prometheus-stack \
  --namespace monitoring \
  --create-namespace
```

# Phase 6: Post-Deployment Security Auditing
```

```
# Run kube-bench for security auditing
kubectl apply -f https://raw.githubusercontent.com/aquasecurity/kube-
bench/main/job-eks.yaml

# Check results
kubectl logs -l app=kube-bench -n default
```

# Phase 7: Disaster Recovery Setup

## Step 1: Set Up Automated Backups

```
# Create an S3 bucket for backups
aws s3 mb s3://ecommerce-eks-backups-YOUR-ACCOUNT-ID

# Set up Velero for Kubernetes backup
helm repo add vmware-tanzu https://vmware-tanzu.github.io/helm-charts
helm repo update

helm install velero vmware-tanzu/velero \
  --namespace velero \
  --create-namespace \
  --set-file credentials.secretContents.cloud=./credentials-velero \
  --set configuration.provider=aws \
  --set configuration.backupStorageLocation.bucket=ecommerce-eks-backups-YOUR-
ACCOUNT-ID \
  --set configuration.backupStorageLocation.config.region=us-west-2

# Create a scheduled backup
kubectl create -f - <<EOF
apiVersion: velero.io/v1
kind: Schedule
metadata:
  name: daily-backup
  namespace: velero
spec:
  schedule: "0 1 * * *"
  template:
    includedNamespaces:
    - ecommerce
    ttl: 720h
EOF
```

## Step 2: Create Restore Documentation

Document the restore procedure:

```
# To restore from backup:
velero restore create --from-backup [backup-name] --include-namespaces ecommerce
```

# Phase 8: Ongoing Maintenance

## Configuration Updates

When making configuration changes:

```
# Make changes to Terraform files
terraform validate
terraform plan
terraform apply
```

## Managing Kubernetes Resources

For Kubernetes-specific updates:

```
# Apply configuration changes
kubectl apply -f updated-deployment.yaml

# Monitor rollout
kubectl rollout status deployment/product-service -n ecommerce
```

# Phase 9: Cost Optimization

## Set Up AWS Cost Explorer Tags

```
# Ensure all resources have appropriate tags
aws resourcegroupstaggingapi tag-resources \
  --resource-arn-list [your-cluster-arn] \
  --tags Environment=prod,Project=ecommerce,CostCenter=devops
```

```
# Enable Cost Explorer if not already enabled
aws ce enable-cost-explorer
```

## Set Up AWS Budgets

```
# Create a budget for the ecommerce project
aws budgets create-budget \
  --account-id YOUR-ACCOUNT-ID \
  --budget file://budget.json \
  --notifications-with-subscribers file://notifications.json
```

# 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, including:

- Infrastructure reproducibility through Terraform
- State management with S3 and DynamoDB
- Container image management with ECR
- Proper Kubernetes resource configuration
- Security with KMS encryption
- Load balancing and external access configuration
- Complete monitoring and troubleshooting capabilities
- Backup and disaster recovery planning
- Cost management and optimization

By following these steps, you've implemented a production-ready Kubernetes deployment on AWS that meets enterprise standards for cloud-native applications.