Microservices Transformation Project at Tech Mahindra

During my time at Tech Mahindra from November 2020 to February 2023, I led a major project to transform our monolithic e-commerce application into a microservices architecture. Here's a detailed account of what we did, including specific implementations and challenges we faced.

Project Overview

We called it "Project Nexus." Our main goal was to break down our monolithic e-commerce app into microservices, containerize them, and set up a robust CI/CD pipeline. This would help us scale better and deploy faster.

Breaking Down the Monolith

We identified six core services to start with:

1. User Service
2. Product Catalog Service
3. Inventory Service
4. Order Service
5. Payment Service
6. Recommendation Service

For each service, we had to carefully extract the relevant code from the monolith. Here's an example of how we structured the Product Catalog Service:

```
product-catalog-service/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/techmahindra/productcatalog/
│   │   │       ├── controller/
│   │   │       ├── model/
│   │   │       ├── repository/
│   │   │       ├── service/
│   │   │       └── ProductCatalogApplication.java
│   │   └── resources/
│   │       └── application.properties
│   └── test/
│       └── java/
│           └── com/techmahindra/productcatalog/
│               └── ProductCatalogApplicationTests.java
├── Dockerfile
├── pom.xml
└── README.md
```

Containerization with Docker

For each service, I created a Dockerfile. Here's the one I wrote for the Product Catalog Service:

dockerfile

```
FROM openjdk:11-jre-slim
WORKDIR /app
COPY target/product-catalog-service.jar .
EXPOSE 8080
CMD ["java", "-jar", "product-catalog-service.jar"]
```

To build and push the image, I used these commands:

bash

```
docker build -t product-catalog-service:v1 .
docker tag product-catalog-service:v1 123456789012.dkr.ecr.ap-south-1.amazonaws.com/product-catalog-service:v1
docker push 123456789012.dkr.ecr.ap-south-1.amazonaws.com/product-catalog-service:v1
```

Kubernetes Setup on AWS EKS

Setting up EKS was a challenge, but I learned a lot. I used Terraform to provision the EKS cluster:

hcl

```
module "eks" {
  source          = "terraform-aws-modules/eks/aws"
  cluster_name    = "nexus-cluster"
  cluster_version = "1.21"
  subnets         = module.vpc.private_subnets

  node_groups = {
    eks_nodes = {
      desired_capacity = 3
      max_capacity     = 7
      min_capacity     = 2
      instance_type    = "m5.large"
    }
  }
}
```

After setting up the cluster, I created Kubernetes deployments for each service. Here's an example for the Product Catalog Service:

yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: product-catalog
spec:
  replicas: 3
  selector:
    matchLabels:
      app: product-catalog
  template:
    metadata:
      labels:
        app: product-catalog
    spec:
      containers:
      - name: product-catalog
        image: 123456789012.dkr.ecr.ap-south-1.amazonaws.com/product-catalog-service:v1
        ports:
        - containerPort: 8080
```

CI/CD Pipeline

I set up a Jenkins pipeline to automate our build and deployment process. Here's a simplified version of the Jenkinsfile I created:

groovy

```
pipeline {
    agent any
    environment {
        AWS_ACCOUNT_ID="123456789012"
        AWS_DEFAULT_REGION="ap-south-1"
        IMAGE_REPO_NAME="product-catalog-service"
        IMAGE_TAG="${BUILD_NUMBER}"
        REPOSITORY_URI =
"${AWS_ACCOUNT_ID}.dkr.ecr.${AWS_DEFAULT_REGION}.amazonaws.com/${IMAGE_REPO_NAME}"
    }
    stages {
        stage('Checkout') {
            steps {
                git 'https://github.com/techmahindra/product-catalog-service.git'
            }
        }
        stage('Build') {
            steps {
                sh 'mvn clean package'
            }
        }
        stage('Docker Build and Push') {
            steps {
                script {
                    sh "aws ecr get-login-password --region ${AWS_DEFAULT_REGION} | docker login --username AWS --
password-stdin ${AWS_ACCOUNT_ID}.dkr.ecr.${AWS_DEFAULT_REGION}.amazonaws.com"
                    sh "docker build -t ${IMAGE_REPO_NAME}:${IMAGE_TAG} ."
                    sh "docker tag ${IMAGE_REPO_NAME}:${IMAGE_TAG} ${REPOSITORY_URI}:${IMAGE_TAG}"
                    sh "docker push ${REPOSITORY_URI}:${IMAGE_TAG}"
                }
            }
        }
        stage('Deploy to EKS') {
            steps {
                script {
                    sh "kubectl set image deployment/product-catalog product-catalog=${REPOSITORY_URI}:${IMAGE_TAG}"
                }
            }
        }
    }
}
```

Monitoring and Logging

For monitoring, I set up Prometheus and Grafana. Here's a snippet of the Prometheus configuration I used:

yaml

```
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'kubernetes-pods'
    kubernetes_sd_configs:
```

```
  - role: pod
  relabel_configs:
  - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_scrape]
    action: keep
    regex: true
```

I also created custom Grafana dashboards to visualize our service metrics. Here's a screenshot of one of the dashboards I made:

Show Image

Challenges and Solutions

7. **Service Communication**: Initially, services were failing to communicate properly. I solved this by implementing service discovery using Kubernetes DNS and creating proper network policies.
8. **Data Consistency**: Maintaining data consistency across services was tricky. I implemented event-driven architecture using Apache Kafka to ensure eventual consistency.
9. **Deployment Rollbacks**: We had a few bad deployments that were hard to rollback. I implemented a blue-green deployment strategy using Kubernetes deployments, which made rollbacks much easier.

Results and Learnings

By the end of the project, we had successfully transformed our monolithic application into a scalable, maintainable microservices architecture. Our deployment time decreased from hours to minutes, and we were able to scale individual services based on demand.

I learned a ton about Kubernetes, CI/CD practices, and cloud-native architectures. This project really deepened my understanding of modern DevOps practices and I'm proud of what we accomplished.