Docker & Kubernetes Implementation Project

Containerization and Orchestration of Node.js Applications

Author: Aman Singh
Date: January 2026
Technologies: Docker, Kubernetes (Minikube), Node.js, Express.js, MySQL

Executive Summary

This project demonstrates the complete lifecycle of containerizing and orchestrating Node.js applications using Docker and Kubernetes. The work was divided into two major phases: deploying a single monolithic application and implementing a microservices architecture.

Key Technologies Used:

- **Containerization:** Docker, Docker Compose
- **Orchestration:** Kubernetes (Minikube)
- **Application:** Node.js, Express.js
- **Database:** MySQL 8.0
- **Platform:** Windows with Minikube

Key Achievements:

1. Successfully containerized a Node.js application with Docker
2. Deployed application to Kubernetes cluster with load balancing and self-healing
3. Implemented persistent data storage using PersistentVolumeClaims
4. Built a three-tier microservices architecture (Frontend, Middleware, Backend)
5. Utilized ConfigMaps for configuration management
6. Implemented LoadBalancer service for external access
7. Demonstrated Docker networking (bridge networks) and Kubernetes networking

(ClusterIP, NodePort, LoadBalancer)

---

1. Introduction

1.1 Project Objectives

The primary objective of this project was to gain hands-on experience with modern containerization and orchestration technologies. Specific goals included:

- Understanding Docker containerization and image creation
- Learning Kubernetes orchestration concepts
- Implementing microservices architecture patterns
- Managing configuration and secrets in containerized environments
- Understanding container networking at both Docker and Kubernetes levels

1.2 Scope of Work

The project was structured in two phases:

Phase 1: Single application deployment covering Docker containerization and Kubernetes orchestration with MySQL integration.

Phase 2: Microservices architecture implementation with separate frontend, middleware, and backend services, deployed using both Docker Compose and Kubernetes.

1.3 Technology Overview

Docker provides containerization, packaging applications with their dependencies into isolated containers. **Kubernetes** orchestrates these containers across clusters, providing features like auto-scaling, self-healing, and load balancing. **Docker Compose** simplifies multi-container development environments.

---

2. Part A: Single Application Deployment

2.1 Application Overview

A Node.js application was built using Express.js framework to demonstrate containerization and orchestration concepts. The application featured:

- **Server information endpoint** displaying hostname, platform, memory, and uptime
- **Health check endpoint** for Kubernetes probes
- **Static file serving** for the web interface
- **MySQL integration** for logging requests

Technology Stack:

- Node.js 18 (Alpine Linux base)
- Express.js 4.18
- MySQL 8.0

2.2 Local Development

Project Structure:

```
nodejs-k8s-app/
├── app/
│   ├── package.json
│   ├── server.js
│   └── public/
│       └── index.html
├── Dockerfile
├── .dockerignore
└── k8s/
    ├── deployment.yaml
    └── service.yaml
```

The application was first developed and tested locally on Windows using:

```
cd app
npm install
npm start
```

The server ran on `http://localhost:3000`, displaying system information dynamically fetched from the backend.

2.3 Docker Implementation

2.3.1 Dockerfile Design

A multi-stage Dockerfile was created to optimize the final image size and security:

```
# Stage 1: Builder
FROM node:18-alpine AS builder
WORKDIR /usr/src/app
COPY ./app/package*.json ./
RUN npm ci --only=production

# Stage 2: Production
FROM node:18-alpine
WORKDIR /usr/src/app
RUN addgroup -g 1001 -S nodejs && adduser -S nodejs -u 1001
COPY --from=builder /usr/src/app/node_modules ./node_modules
COPY --chown=nodejs:nodejs ./app/ .
USER nodejs
EXPOSE 3000
CMD ["node", "server.js"]
```

Key Features:

- **Multi-stage build:** Separates build dependencies from runtime
- **Non-root user:** Enhances security by running as unprivileged user
- **Alpine Linux:** Minimal base image (~5MB vs 900MB for Ubuntu)
- **Health checks:** Automatic container health monitoring

2.3.2 Docker Networking

Default Bridge Network:

Containers on the default bridge network can communicate via IP addresses but not by name:

```
docker run -d -p 3000:3000 --name nodejs-app nodejs-k8s-app
```

Custom Bridge Network:

Created a custom bridge network for better DNS resolution:

```
docker network create --driver bridge my-app-network
docker run -d -p 3001:3000 --name app1 --network my-app-network nodejs-k8s-app
docker run -d -p 3002:3000 --name app2 --network my-app-network nodejs-k8s-app
```

Containers on custom bridge networks can ping each other by name:

```
docker exec app1 ping app2
```

Commands Used:

```
# Build image
docker build -t nodejs-k8s-app:latest .

# Run container
docker run -d -p 3000:3000 --name nodejs-app nodejs-k8s-app

# View logs
docker logs nodejs-app

# Execute into container
docker exec -it nodejs-app sh

# Create network
docker network create my-app-network

# Inspect network
docker network inspect my-app-network
```

Achievement: Successfully containerized the application with proper networking configuration.

2.4 Kubernetes Deployment

2.4.1 Minikube Setup

Minikube was installed to create a local Kubernetes cluster:

```
minikube start --driver=docker
minikube status
```

Images were built in Minikube's Docker environment:

```
& minikube docker-env --shell powershell | Invoke-Expression
docker build -t nodejs-k8s-app:latest .
```

2.4.2 Deployment Configuration

deployment.yaml - Defined 3 replicas with health checks:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nodejs-app-deployment
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: nodejs-app
        image: nodejs-k8s-app:latest
        ports:
        - containerPort: 3000
        livenessProbe:
          httpGet:
            path: /health
            port: 3000
          initialDelaySeconds: 10
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /health
            port: 3000
          initialDelaySeconds: 5
          periodSeconds: 5
```

service.yaml - Exposed the application via NodePort:

```
apiVersion: v1
kind: Service
metadata:
  name: nodejs-app-service
spec:
  type: NodePort
  selector:
    app: nodejs-app
  ports:
  - port: 3000
    targetPort: 3000
    nodePort: 30080
```

Deployment Commands:

```
kubectl apply -f k8s/deployment.yaml
kubectl apply -f k8s/service.yaml
```

```
kubectl get pods
kubectl get services
minikube service nodejs-app-service --url
```

2.4.3 Key Features Demonstrated

Load Balancing:

With 3 replicas, the service automatically distributed traffic across pods. Refreshing the webpage showed different hostnames, proving round-robin load balancing.

```
kubectl get pods -o wide
# Shows 3 pods with different IPs
```

Self-Healing:

When a pod was deleted, Kubernetes automatically created a replacement:

```
kubectl delete pod <pod-name>
kubectl get pods -w  # Watch new pod being created
```

Scaling:

Easily scaled replicas up or down:

```
kubectl scale deployment nodejs-app-deployment --replicas=5
kubectl get pods  # Shows 5 pods running
```

Health Checks:

- **Liveness Probe:** Kubernetes restarted pods that failed health checks
- **Readiness Probe:** Pods not ready were removed from service endpoints

Resource Management:

```
resources:
  requests:
    memory: "128Mi"
    cpu: "100m"
  limits:
    memory: "256Mi"
    cpu: "200m"
```

Ensured pods didn’t consume excessive resources.

Achievement: Successfully deployed a highly available, self-healing application on Kubernetes with automatic load balancing.

2.5 MySQL Integration

2.5.1 Database Addition

MySQL was integrated to store request logs, demonstrating stateful applications in Kubernetes.

What was added:

- MySQL 8.0 deployment
- PersistentVolumeClaim for data persistence
- Kubernetes Secret for password management
- Modified Node.js app to log requests to MySQL

2.5.2 Persistent Storage

mysql-pvc.yaml:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

This ensured data survived pod restarts.

2.5.3 Secrets Management

mysql-secret.yaml:

```
apiVersion: v1
kind: Secret
metadata:
```

```
  name: mysql-secret
type: Opaque
data:
  mysql-root-password: cm9vdHBhc3N3b3JkMTIz  # base64 encoded
  mysql-password: YXBwcGFzc3dvcmQ=
```

Passwords were stored securely and injected as environment variables.

2.5.4 Data Stored

The application logged every HTTP request to MySQL:

Database Schema:

- `id` : Auto-increment primary key
- `timestamp` : Request time
- `hostname` : Which pod served the request
- `client_ip` : Client IP address
- `request_path` : URL path
- `status_code` : HTTP response code
- `platform` : OS platform
- `environment` : Development/production

Accessing Data:

```
kubectl exec -it <mysql-pod-name> -- mysql -u appuser -p
mysql> SELECT * FROM request_logs;
mysql> SELECT hostname, COUNT(*) FROM request_logs GROUP BY hostname;
```

This showed request distribution across pods, proving load balancing.

2.5.5 Testing Data Persistence

```
# Check data count
kubectl exec -it <mysql-pod> -- mysql -u appuser -p -e "SELECT COUNT(*) FROM appd

# Delete MySQL pod
kubectl delete pod <mysql-pod-name>
```

```
# Wait for new pod to come up
kubectl get pods -w

# Check data is still there
kubectl exec -it <new-mysql-pod> -- mysql -u appuser -p -e "SELECT COUNT(*) FROM
```

Data persisted across pod restarts, validating PersistentVolume implementation.

Achievement: Successfully integrated stateful MySQL database with persistent storage and secure credential management.

---

3. Part B: Microservices Architecture

3.1 Microservices Concept

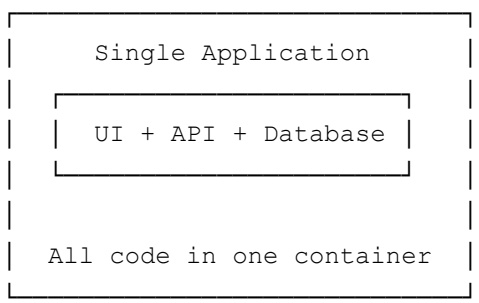
3.1.1 What are Microservices?

Microservices architecture breaks an application into independent, loosely coupled services. Each service:

- Runs in its own process
- Communicates via well-defined APIs
- Can be deployed independently
- Owns its own data
- Can use different technologies

3.1.2 Monolithic vs Microservices

Monolithic Architecture:



Drawbacks:

- Tight coupling between components
- Difficult to scale specific parts
- One bug can crash entire application
- Technology lock-in

Microservices Architecture:

Benefits:

- Independent scaling
- Fault isolation
- Technology diversity
- Easier maintenance
- Faster development

3.1.3 Our Architecture

Three-Tier Microservices:

1. **Frontend Service** (Presentation Layer)
    - Serves static HTML/CSS/JavaScript
    - Makes API calls to middleware
    - Port: 3000
2. **Middleware Service** (Business Logic Layer)
    - REST API endpoints
    - Business logic processing
    - Database queries
    - Port: 5000
3. **Backend Service** (Data Layer)

- MySQL database
- Data persistence
- Port: 3306

3.2 Architecture Design

3.2.1 Service Communication Flow

```
User Browser
    ↓ HTTP Request
Frontend Service (Port 3000)
    ↓ API Call (http://middleware:5000/api/info)
Middleware Service (Port 5000)
    ↓ MySQL Query
MySQL Service (Port 3306)
    ↓ Data Response
Middleware Service
    ↓ JSON Response
Frontend Service
    ↓ Render UI
User Browser
```

3.2.2 Technology Stack

Service	Technology	Image	Purpose
Frontend	Node.js + Express	frontend:latest	Static file serving
Middleware	Node.js + Express + MySQL2	middleware:latest	REST API
Backend	MySQL 8.0	mysql:8.0	Database

Frontend Features:

- Beautiful dashboard UI
- Real-time data display
- Service status monitoring
- Request logs visualization

Middleware Features:

- `/health` - Health check endpoint
- `/api/info` - Server information
- `/api/stats` - Request statistics
- `/api/logs` - Recent request logs
- CORS enabled for cross-origin requests

Backend Features:

- Request logs table
- Automatic data persistence
- Connection pooling

3.3 Docker Compose Implementation

3.3.1 What is Docker Compose?

Docker Compose is a tool for defining and running multi-container applications. Instead of running multiple `docker run` commands, a single YAML file defines all services.

Benefits:

- Single command to start all services
- Automatic network creation
- Service dependency management
- Environment variable management
- Volume management

3.3.2 Services Defined

docker-compose.yml:

```
version: '3.8'

services:
  mysql:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: rootpassword123
```

```
      MYSQL_DATABASE: appdb
      MYSQL_USER: appuser
      MYSQL_PASSWORD: apppassword
    ports:
      - "3306:3306"
    volumes:
      - mysql_data:/var/lib/mysql
    networks:
      - microservices-network
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
      interval: 10s
      timeout: 5s
      retries: 5

  middleware:
    build: ./middleware
    environment:
      DB_HOST: mysql
      DB_USER: appuser
      DB_PASSWORD: apppassword
      DB_NAME: appdb
    ports:
      - "5000:5000"
    depends_on:
      mysql:
        condition: service_healthy
    networks:
      - microservices-network

  frontend:
    build: ./frontend
    environment:
      API_URL: http://middleware:5000
    ports:
      - "3000:3000"
    depends_on:
      - middleware
    networks:
      - microservices-network

networks:
  microservices-network:
    driver: bridge

volumes:
  mysql_data:
```

Key Features:

1. **Service Dependencies:**
    - Frontend depends on middleware
    - Middleware depends on MySQL
    - MySQL must be healthy before middleware starts
2. **Network Configuration:**
    - Custom bridge network: `microservices-network`
    - Services communicate by name (e.g., `http://middleware:5000`)
    - DNS resolution built-in
3. **Volume Management:**
    - Named volume `mysql_data` for database persistence
    - Data survives container restarts

3.3.3 Key Commands

```
# Start all services
docker-compose up -d

# View logs
docker-compose logs -f

# Check service status
docker-compose ps

# Stop services
docker-compose stop

# Remove containers
docker-compose down

# Remove containers and volumes
docker-compose down -v

# Restart specific service
docker-compose restart middleware
```

```
# View resource usage
docker stats
```

3.3.4 Development Workflow

Day-to-day development:

```
# Make code changes in frontend
docker-compose restart frontend

# View middleware logs
docker-compose logs -f middleware

# Access MySQL
docker-compose exec mysql mysql -u appuser -p

# Clean rebuild
docker-compose down
docker-compose up --build
```

Achievement: Created a complete local development environment with three interconnected services, managed through a single YAML file.

3.4 Kubernetes Implementation

3.4.1 Namespace Creation

Created a dedicated namespace for better organization:

```
kubectl create namespace microservices
kubectl get namespaces
```

All resources were deployed in the `microservices` namespace.

3.4.2 ConfigMap Implementation

What is ConfigMap?

ConfigMap is a Kubernetes object that stores configuration data as key-value pairs. It separates configuration from application code.

Why Use ConfigMap?

1. **Separation of Concerns:** Code doesn't contain configuration

2. **Environment-Specific Configs:** Same image, different configs for dev/staging/prod
3. **Easy Updates:** Change config without rebuilding images
4. **Centralized Management:** All configuration in one place

ConfigMaps Created:

```
# frontend-config
apiVersion: v1
kind: ConfigMap
metadata:
  name: frontend-config
  namespace: microservices
data:
  API_URL: "http://middleware-service:5000"
  PORT: "3000"

---
# middleware-config
apiVersion: v1
kind: ConfigMap
metadata:
  name: middleware-config
  namespace: microservices
data:
  PORT: "5000"
  NODE_ENV: "production"
  DB_HOST: "mysql-service"
  DB_PORT: "3306"
  DB_NAME: "appdb"
  DB_USER: "appuser"
```

Usage in Deployment:

```
spec:
  containers:
  - name: middleware
    envFrom:
    - configMapRef:
        name: middleware-config
```

All ConfigMap values are automatically injected as environment variables.

Testing ConfigMap:

```
# View ConfigMap
kubectl describe configmap middleware-config -n microservices

# Check environment variables in pod
kubectl exec -it <middleware-pod> -n microservices -- env | findstr DB
```

3.4.3 Secrets Management

Passwords stored in Kubernetes Secrets (base64 encoded):

```
apiVersion: v1
kind: Secret
metadata:
  name: mysql-secret
  namespace: microservices
type: Opaque
data:
  mysql-root-password: cm9vdHBhc3N3b3JkMTIz
  mysql-password: YXBwcGFzc3dvcmQ=
```

3.4.4 Service Deployment Strategy

MySQL Deployment (Backend):

- **Replicas:** 1 (databases need special handling for multi-replica)
- **Service Type:** ClusterIP (internal only)
- **Storage:** PersistentVolumeClaim (2Gi)
- **Secrets:** Passwords injected from Secret

Middleware Deployment (API):

- **Replicas:** 2 (for load balancing)
- **Service Type:** ClusterIP (internal only)
- **ConfigMap:** Database connection details
- **Secret:** Database password

Frontend Deployment (UI):

- **Replicas:** 3 (for high availability)
- **Service Type:** LoadBalancer (external access)

- **ConfigMap:** API URL

Deployment Commands:

```
# Deploy in order
kubectl apply -f k8s/namespace.yaml
kubectl apply -f k8s/configmap.yaml
kubectl apply -f k8s/secrets.yaml
kubectl apply -f k8s/mysql-pvc.yaml
kubectl apply -f k8s/mysql-deployment.yaml
kubectl apply -f k8s/mysql-service.yaml
kubectl apply -f k8s/middleware-deployment.yaml
kubectl apply -f k8s/middleware-service.yaml
kubectl apply -f k8s/frontend-deployment.yaml
kubectl apply -f k8s/frontend-service.yaml

# Verify all resources
kubectl get all -n microservices
```

3.4.5 LoadBalancer vs NodePort

NodePort:

```
Browser → http://minikube-ip:30080
          ↓
    Node Port 30080
          ↓
      Service
          ↓
       Pods
```

Limitations:

- Must use node IP address
- Port range: 30000-32767
- Not production-ready

LoadBalancer:

```
Browser → http://external-ip
          ↓
  Cloud Load Balancer
          ↓
    Node Port
```

```
    ↓
Service
    ↓
 Pods
```

Benefits:

- Clean external IP/domain
- Standard HTTP ports (80/443)
- Production-ready
- Automatic health checks

3.4.6 Minikube Tunnel

Minikube doesn't have a real cloud load balancer, so we use `minikube tunnel`:

```
# In separate terminal (keep running)
minikube tunnel

# Get external IP
kubectl get svc frontend-service -n microservices
```

The tunnel creates a route from Windows to Minikube, simulating a LoadBalancer.

3.4.7 Scaling Strategies

Manual Scaling:

```
# Scale frontend
kubectl scale deployment frontend-deployment --replicas=5 -n microservices

# Scale middleware
kubectl scale deployment middleware-deployment --replicas=3 -n microservices

# Verify
kubectl get pods -n microservices
```

Achievement: Deployed a complete three-tier microservices application on Kubernetes with proper configuration management, load balancing, and scaling capabilities.

3.5 Inter-Service Communication

3.5.1 DNS-Based Service Discovery

Kubernetes provides built-in DNS for services. Format: `<service-name>.<namespace>.svc.cluster.local`

Examples:

- Full: `middleware-service.microservices.svc.cluster.local`
- Short (same namespace): `middleware-service`
- With port: `middleware-service:5000`

Testing DNS:

```
kubectl run test-pod --image=busybox -n microservices -it --rm -- nslookup middle
```

3.5.2 Internal vs External Services

Internal Services (ClusterIP):

- `mysql-service` - Only accessible within cluster
- `middleware-service` - Only accessible within cluster

External Service (LoadBalancer):

- `frontend-service` - Accessible from outside cluster

This provides security by not exposing internal services.

3.5.3 Communication Flow

Frontend → Middleware:

```
// In frontend app.js
const API_URL = 'http://middleware-service:5000';
fetch(`${API_URL}/api/info`)
```

Middleware → MySQL:

```
// In middleware server.js
const dbConfig = {
  host: process.env.DB_HOST, // 'mysql-service'
  port: 3306
};
```

Testing Connectivity:

```
# From frontend pod to middleware
kubectl exec -it <frontend-pod> -n microservices -- wget -qO- http://middleware-s

# From middleware pod to MySQL
kubectl exec -it <middleware-pod> -n microservices -- wget -qO- http://mysql-serv
```

Achievement: Established secure, reliable communication between microservices using Kubernetes DNS and service discovery.

---

4. Key Concepts Learned

4.1 Docker Concepts

4.1.1 Containerization Benefits

- **Consistency:** Same environment across dev, test, prod
- **Isolation:** Each container has its own filesystem and processes
- **Portability:** Run anywhere Docker runs
- **Efficiency:** Lighter than virtual machines
- **Version Control:** Image tags for version management

4.1.2 Multi-Stage Builds

```
FROM node:18-alpine AS builder
# Build dependencies

FROM node:18-alpine
# Runtime only
```

Benefits:

- Smaller final image
- Faster deployments
- Reduced attack surface

- Separation of build and runtime dependencies

4.1.3 Networking Modes

Mode	Use Case	Connectivity
Bridge (default)	Most apps	Containers via IP
Custom Bridge	Better isolation	Containers via name
Host	Performance critical	Direct host network
None	Maximum security	No network

4.2 Kubernetes Concepts

4.2.1 Core Resources

Pod:

- Smallest deployable unit
- One or more containers
- Shared network and storage
- Ephemeral (can be deleted/recreated)

Deployment:

- Manages multiple pod replicas
- Ensures desired state
- Rolling updates
- Rollback capability

Service:

- Stable network endpoint
- Load balances to pods
- Service discovery via DNS
- Types: ClusterIP, NodePort, LoadBalancer

ConfigMap:

- Non-sensitive configuration
- Key-value pairs
- Injected as environment variables or files

Secret:

- Sensitive data (passwords, tokens)
- Base64 encoded
- Injected securely into pods

4.2.2 Declarative vs Imperative

Imperative (commands):

```
kubectl run nginx --image=nginx
kubectl scale deployment nginx --replicas=3
```

Declarative (YAML files):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
```

Benefit of Declarative: Infrastructure as Code, version control, reproducibility

4.2.3 Self-Healing

Kubernetes automatically:

- Restarts failed containers
- Replaces crashed pods
- Reschedules pods from failed nodes
- Kills pods that don’t respond to health checks

Example:

```
kubectl delete pod <pod-name>
# Kubernetes immediately creates replacement
```

4.2.4 Load Balancing

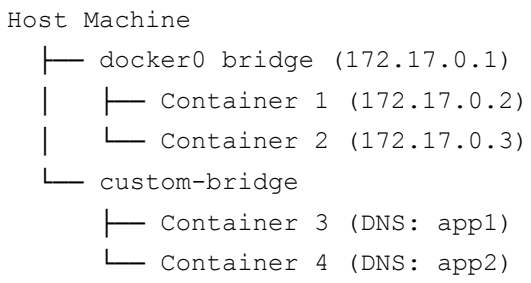
Service distributes traffic across healthy pods using round-robin algorithm.

Verification:

- Refresh webpage multiple times
- Different hostname each time
- Proves load balancing working

4.3 Networking Deep Dive

4.3.1 Docker Bridge Networking



Custom bridges provide DNS resolution.

4.3.2 Kubernetes Pod Networking

Every pod gets a unique IP from the pod network (10.244.0.0/16). Pods can communicate directly without NAT.

4.3.3 Service Networking

ClusterIP:

- Virtual IP within cluster
- Load balances to pod IPs
- Internal only

NodePort:

- Opens port on every node
- Forwards to ClusterIP
- External access via node IP

LoadBalancer:

- Creates external load balancer
- Forwards to NodePort
- Production external access

4.3.4 Traffic Flow End-to-End

```
1. Browser: http://external-ip
2. LoadBalancer: Routes to Node
3. Node: Port forwards to Service (ClusterIP)
4. Service: Selects pod via load balancing
5. iptables: Routes to pod IP
6. Pod: Container receives request
7. Response: Reverse path back to browser
```

---

5. Comparison: Single App vs Microservices

Aspect	Single Application	Microservices
Architecture	Monolithic (all-in-one)	Distributed (separate services)
Deployment	Single container/pod	Multiple containers/pods
Scaling	Scale entire app	Scale services independently
Development	One team, one codebase	Multiple teams, multiple codebases
Technology	Single stack	Different stacks per service
Failure Impact	Entire app down	Isolated to one service
Complexity	Lower	Higher
Resource Usage	Efficient for small apps	Overhead for small apps
Updates	Update entire app	Update individual services

Database	Single database	Database per service

When to Use Single Application:

- Small applications
- Simple requirements
- Small team
- Fast development needed
- Low traffic

When to Use Microservices:

- Large, complex applications
- Need to scale specific components
- Multiple teams
- Different technologies required
- High availability requirements

---

6. Testing & Validation

6.1 Load Balancing Test

Method:

1. Deployed 3 frontend replicas
2. Accessed application via LoadBalancer IP
3. Clicked “Test Load Balancing” button (makes 10 rapid requests)
4. Checked request logs

Result:

```
kubectl logs -l app=middleware -n microservices --tail=20
```

Logs showed requests distributed across 2 middleware pods, confirming load balancing.

6.2 Self-Healing Test

Method:

```
# Delete a pod
kubectl delete pod <pod-name> -n microservices

# Watch replacement creation
kubectl get pods -n microservices -w
```

Result: Kubernetes immediately created a new pod to maintain desired replica count. Application remained accessible throughout.

6.3 ConfigMap Update Test

Method:

```
# Update ConfigMap
kubectl edit configmap middleware-config -n microservices
# Changed NODE_ENV from "production" to "staging"

# Restart deployment
kubectl rollout restart deployment middleware-deployment -n microservices

# Verify new config
kubectl exec -it <middleware-pod> -n microservices -- env | findstr NODE_ENV
```

Result: Environment variable updated without rebuilding Docker image.

---

7. Challenges & Solutions

7.1 Technical Challenges

Challenge 1: Folder Structure Mismatch

- **Issue:** Dockerfile couldn't find package.json
- **Solution:** Updated COPY paths to `./app/package*.json`

Challenge 2: LoadBalancer Pending State

- **Issue:** LoadBalancer external IP stuck in "Pending"
- **Solution:** Ran `minikube tunnel` in separate terminal

Challenge 3: MySQL Connection Failure

- **Issue:** Middleware couldn't connect to MySQL
- **Solution:** Ensured MySQL pod was ready before middleware started using health checks

Challenge 4: ConfigMap Changes Not Reflected

- **Issue:** Updated ConfigMap but pods still had old values
- **Solution:** Learned that pods must be restarted to pick up ConfigMap changes

7.2 Lessons Learned

1. **Always use health checks:** Prevents routing traffic to unhealthy pods
2. **Leverage ConfigMaps:** Makes applications configurable without rebuilding
3. **Use Secrets for passwords:** Never hardcode credentials
4. **Test increment