Container Design & Deployment Project #2 2025/26

Book Management Microservice

Docker Compose | Docker Swarm | Kubernetes

Student Name: Ahmed Wahba

Student ID: A00336722

Module: Container Design & Deployment

Submission Date: December 2025

Repository: https://github.com/ahmedwahba47/cdd-assignment-2

Docker Hub: https://hub.docker.com/r/wahba87/bookservice

Table of Contents

1. Introduction

1.1 Microservices and Containerisation

Microservices architecture structures applications as a collection of loosely coupled, independently deployable services. Each service handles a specific business capability, can be deployed without affecting others, and may use different technology stacks.

Containerisation packages applications with their dependencies into isolated, portable units. Key benefits include:

Benefit	Description
Consistency	Identical environment from development to production
Isolation	Applications run without conflicts with other processes
Portability	Runs on any platform supporting container runtime
Efficiency	Lightweight compared to VMs (shared OS kernel)
Speed	Containers start in seconds, enabling rapid deployment

In this project, I developed a **Book Management Microservice** using Java Spring Boot (Java 25 LTS)that provides RESTful APIs for CRUD operations on a book inventory, backed by MySQL 8.4.

1.2 The Role of Container Orchestration

Container orchestration automates deployment, scaling, and management of containerised applications. As microservice architectures grow, orchestration provides:

Function	Description
Scheduling	Assigns containers to nodes based on resource availability
Service Discovery	Enables containers to find each other dynamically
Load Balancing	Distributes traffic across container instances
Scaling	Adjusts container count based on demand
Self-healing	Replaces failed containers automatically
Rolling Updates	Deploys updates without service downtime

1.3 Logging and Monitoring in Distributed Systems

Observability is critical in microservice systems. The **ELK Stack** provides comprehensive logging:

Component	Role
Elasticsearch	Distributed search engine for storing and querying logs
Logstash	Pipeline for collecting, parsing, and forwarding logs
Kibana	Dashboard for visualising and analysing log data

2. Docker Compose Deployment

2.1 What is Docker Compose?

Docker Compose is a tool for defining and running multi-container applications using declarative YAML configuration.

Feature	Description
Declarative Config	Services, networks, volumes defined in YAML
Dependency Management	Control startup order with depends_on
Environment Variables	Configuration via .env files
Volume Persistence	Data persists across container restarts
Network Isolation	Services communicate via custom networks

2.2 Microservice Architecture

The Book Management application consists of two containers communicating over a bridge network:
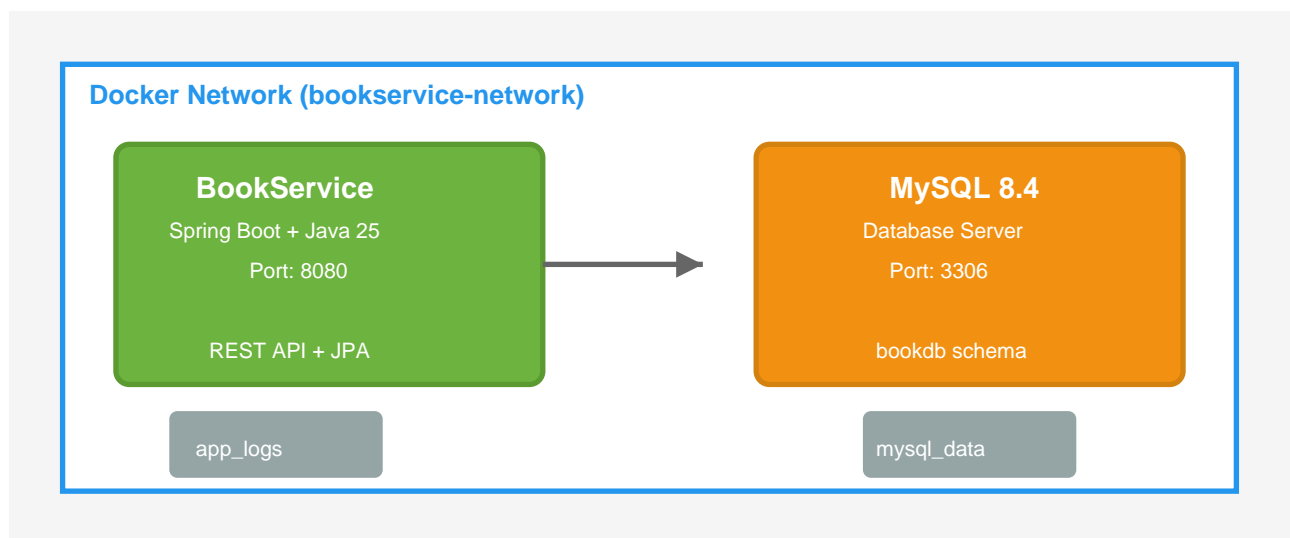

Figure 1: Docker Compose Architecture - BookService and MySQL containers

2.3 Dockerfile (Multi-stage Build)

The Dockerfile uses multi-stage build to optimise the image size:

```
# Stage 1: Build
FROM maven:3.9-eclipse-temurin-25 AS builder
WORKDIR /app
COPY pom.xml .
RUN mvn dependency:go-offline -B
COPY src ./src
```

```
RUN mvn clean package -DskipTests -B


# Stage 2: Runtime

FROM eclipse-temurin:25-jre-alpine

RUN addgroup -S appgroup && adduser -S appuser -G appgroup

WORKDIR /app

COPY --from=builder /app/target/*.jar app.jar

USER appuser

EXPOSE 8080

HEALTHCHECK --interval=30s --timeout=10s CMD wget -q --spider http://l
   ocalhost:8080/actuator/health

ENTRYPOINT ["java", "-jar", "app.jar"]
```

Figure 2: Multi-stage Dockerfile for BookService

2.4 Deployment Steps

Build and Start Services:

```
$ cd docker-compose
$ docker-compose up -d --build
```

Verify Containers:

```
$ docker-compose ps
$ docker-compose logs -f bookservice
```

Test API Endpoints:

Method	Endpoint	Description
GET	/api/books	Retrieve all books
GET	/api/books/{id}	Get book by ID
POST	/api/books	Create new book
PUT	/api/books/{id}	Update book
DELETE	/api/books/{id}	Delete book
GET	/actuator/health	Health check

Push to Docker Hub:

```
$ docker login
$ docker tag bookservice:1.0.0 wahba87/bookservice:1.0.0
$ docker push wahba87/bookservice:1.0.0
```
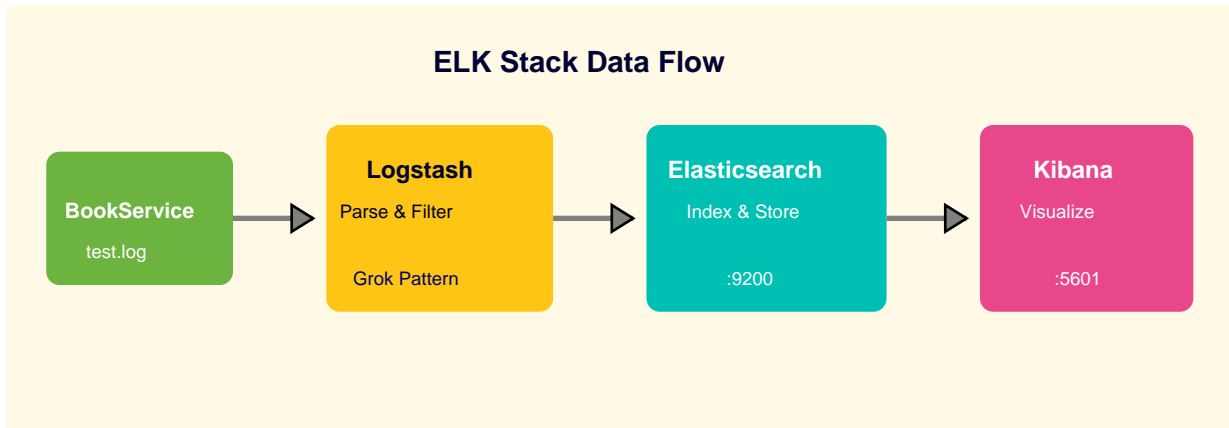
2.5 ELK Stack Integration

ELK Stack Data Flow

BookService	Logstash	Elasticsearch	Kibana
test.log	Parse & Filter Grok Pattern	Index & Store :9200	Visualize :5601

Figure 3: ELK Stack Data Flow for Log Processing

Logstash Configuration:

```
input {
  file {
    path => "/app/logs/test.log"
    start_position => "beginning"
  }
}


filter {
  grok {
    match => { "message" => "%{TIMESTAMP_ISO8601:timestamp} | %{WORD:m
  ethod} %{URIPATH:endpoint}" }
  }
}


output {
  elasticsearch {
    hosts => ["elasticsearch:9200"]
    index => "bookservice-logs-%{+YYYY.MM.dd}"
  }
}
```

3. Docker Swarm Deployment

3.1 What is Docker Swarm?

Docker Swarm is Docker's native clustering and orchestration tool.

Component	Description
Manager Nodes	Control cluster state, schedule services
Worker Nodes	Execute container tasks
Services	Declarative definition of desired state
Tasks	Individual container instances
Overlay Network	Multi-host networking
Ingress	Built-in load balancing

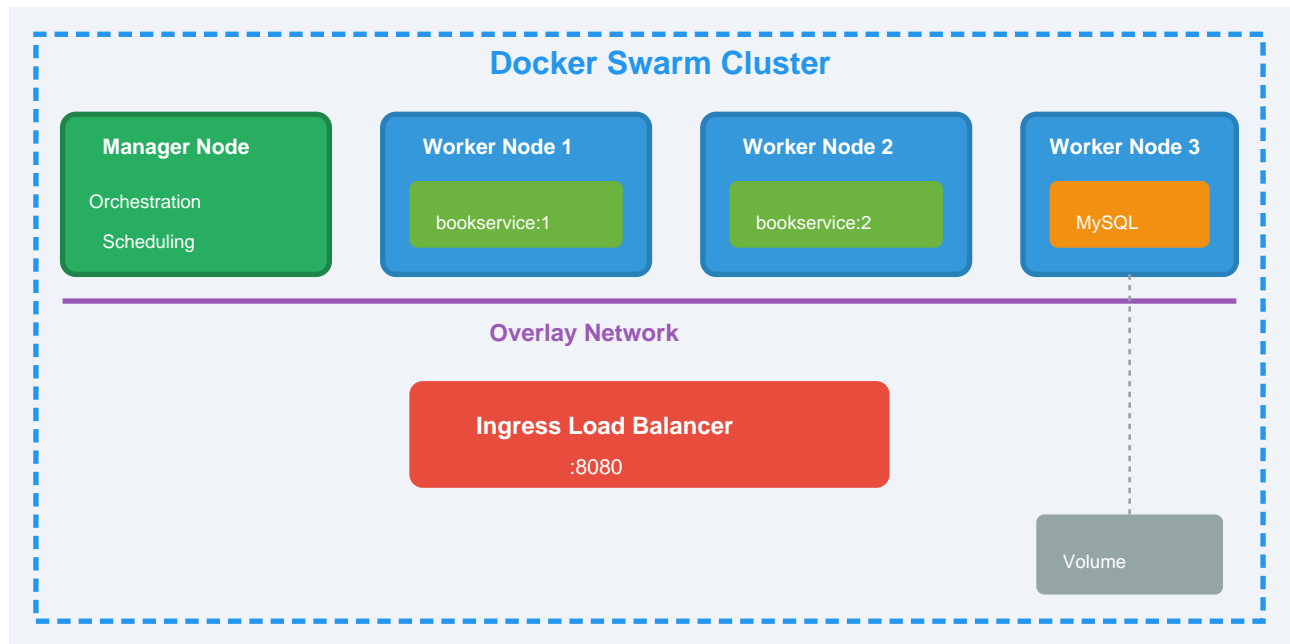## 3.2 Swarm Architecture

Figure 4: Docker Swarm Cluster with Manager and Worker Nodes

3.3 Stack Configuration

```
version: '3.8'
services:
  bookservice:
    image: bookservice:1.0.0
    deploy:
      mode: replicated
```

```
      replicas: 3

      update_config:

        parallelism: 1

        delay: 10s

        failure_action: rollback

      resources:

        limits:

          cpus: '0.5'

          memory: 512M

    networks:

      - bookservice-network


networks:

  bookservice-network:

    driver: overlay
```

Figure 5: Docker Stack YAML Configuration

3.4 Scaling, Upgrades and Rollback

Initialize and Deploy:

```
$ docker swarm init

$ docker stack deploy -c docker-stack.yml bookstack
```

Scale Services:

```
$ docker service scale bookstack_bookservice=5
```

Rolling Update (Upgrade):

Swarm performs rolling updates by replacing containers one at a time, ensuring zero downtime:

```
$ docker service update --image bookservice:1.0.1 \

$   --env-add APP_VERSION=1.0.1 bookstack_bookservice
```

Rollback:

If the new version has issues, instantly revert to the previous version:

```
$ docker service rollback bookstack_bookservice
```

3.5 Advantages and Limitations

Advantages	Limitations
Native Docker integration	Smaller ecosystem than K8s
Simple setup	No built-in auto-scaling

Low learning curve	Limited networking options
Built-in load balancing	Fewer integrations

4. Kubernetes Deployment

4.1 What is Kubernetes?

Kubernetes (K8s) is an open-source container orchestration platform providing automated deployment, scaling, and management of containerised applications.

Note: For this project, I used **k3s** instead of Minikube. Minikube caused persistent system freezes on my VPS requiring hard reboots - a documented issue (GitHub #11124) related to disk I/O operations. k3s is a lightweight, production-ready Kubernetes distribution from Rancher that runs natively without virtualisation overhead, making it ideal for VPS deployments.

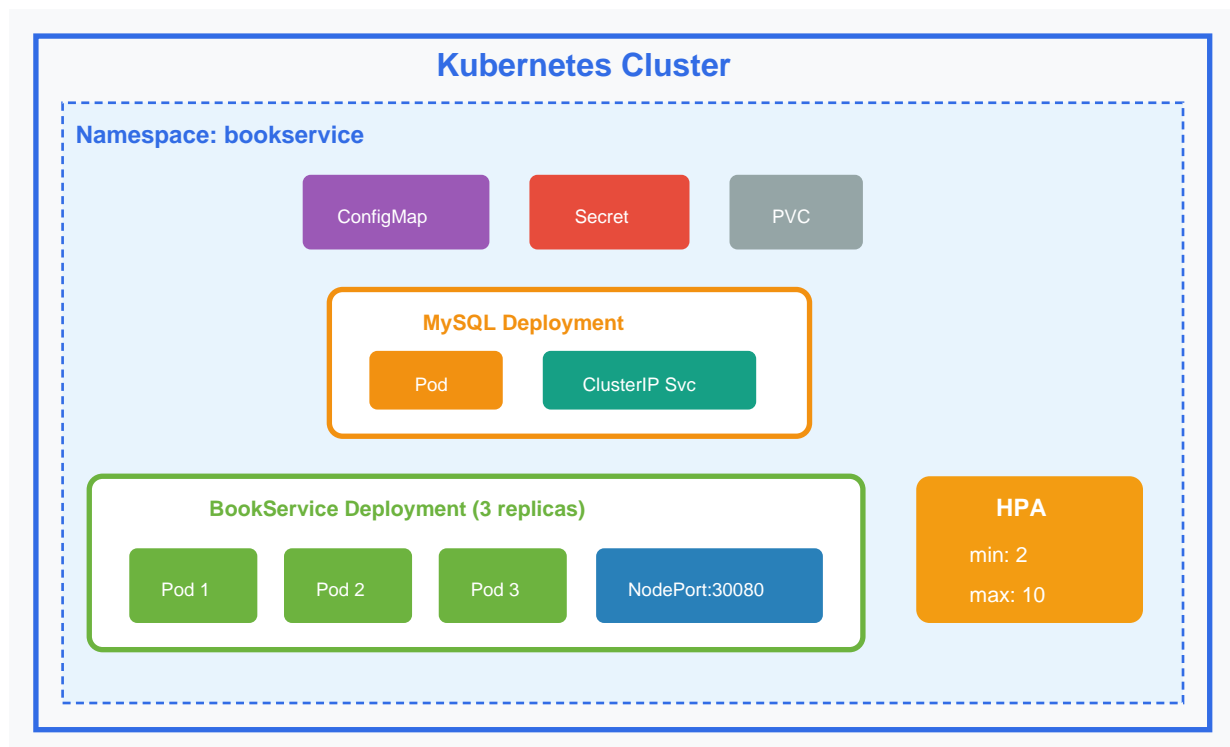| Component	Description
Pod	Smallest deployable unit; containers sharing network
Deployment	Manages ReplicaSets and declarative updates
Service	Stable network endpoint for pods
ConfigMap	Non-sensitive configuration data
Secret	Sensitive data (passwords, tokens)
PVC	Persistent storage request
HPA	Automatic scaling based on metrics

4.2 Architecture

Figure 6: Kubernetes Cluster Architecture with HPA

4.3 Deployment Configuration

```yaml
apiVersion: apps/v1

kind: Deployment

metadata:

  name: bookservice

  namespace: bookservice

spec:

  replicas: 3

  strategy:

    type: RollingUpdate

    rollingUpdate:

      maxUnavailable: 1

      maxSurge: 1

  template:

    spec:

      containers:

        - name: bookservice

          image: wahba87/bookservice:1.0.0

          resources:

            limits:

              memory: "512Mi"

              cpu: "500m"

          livenessProbe:

            httpGet:

              path: /actuator/health

              port: 8080
```

Figure 7: Kubernetes Deployment Manifest

4.4 Scaling, Upgrades and Rollback

Deploy and Scale:

```
$ kubectl apply -f kubernetes/

$ kubectl scale deployment bookservice -n bookservice --replicas=5
```

Rolling Update (Upgrade):

Kubernetes performs rolling updates with zero downtime - new pods start before old ones terminate:

```
$ kubectl apply -f kubernetes/bookservice-configmap-v1.0.1.yaml

$ kubectl rollout restart deployment/bookservice -n bookservice

$ kubectl rollout status deployment/bookservice -n bookservice
```

Rollback:

Instantly revert to the previous deployment revision:

```
$ kubectl apply -f kubernetes/bookservice-configmap.yaml

$ kubectl rollout undo deployment/bookservice -n bookservice
```

4.5 Helm Charts

Helm is the package manager for Kubernetes, enabling templated deployments:

```
$ helm install bookservice ./bookservice-chart -n bookservice

$ helm upgrade bookservice ./bookservice-chart --set replicaCount=5

$ helm rollback bookservice -n bookservice
```

- Templating - Parameterised configs for environments

- Versioning - Track deployment versions

- Dependencies - Manage chart dependencies

- Rollbacks - Easy rollback to previous release

5. Evaluation and Conclusion
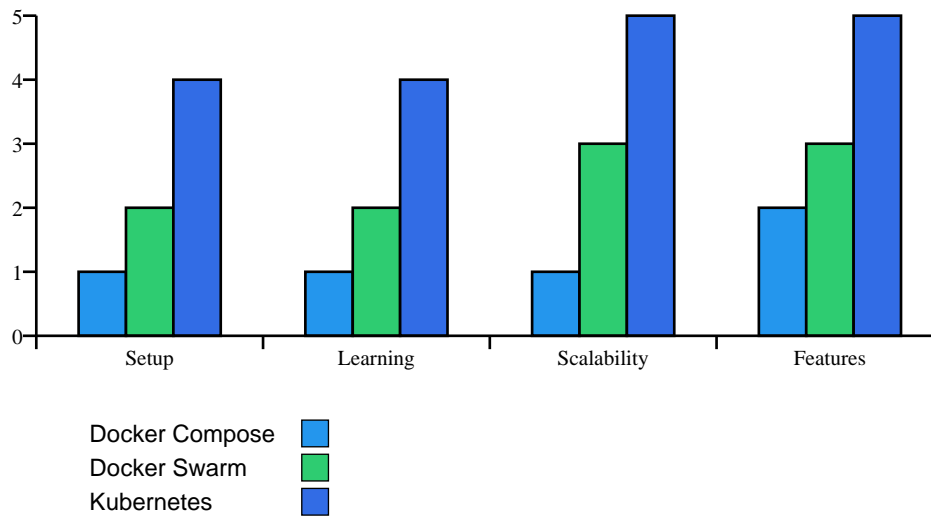
5.1 Comparative Analysis

Figure 8: Orchestration Tools Comparison (1=Low, 5=High)

Feature	Docker Compose	Docker Swarm	Kubernetes
Setup	Low	Medium	High
Learning Curve	Easy	Moderate	Steep
Scalability	Single host	Multi-host	Enterprise
Auto-scaling	No	No	Yes (HPA)
Rolling Updates	Basic	Yes	Advanced
Best For	Development	Small prod	Enterprise

5.2 Reflection

What Worked Well:

- Docker Compose enabled quick local iteration - I could rebuild and test changes in seconds

- Multi-stage Dockerfile reduced image size from ~800MB to ~350MB (over 50% reduction)

- Health checks with depends_on condition ensured MySQL was ready before the app started

- Helm charts made it easy to deploy the same app with different configurations

- Using non-root user in containers improved security posture

Challenges and Solutions:

- MySQL startup timing - initially the app crashed when MySQL wasn't ready. Solved by adding health checks and depends_on with condition: service_healthy

- Kubernetes networking - understanding Services, ClusterIP vs NodePort took time. Reading official docs and hands-on practice helped

- **VPS freezing with disk I/O spikes** - experienced system freezes every 15-30 minutes requiring hard reboots. Root cause: Hetzner CX series VPS uses network-attached storage with limited IOPS (~3,000-5,000 baseline). Elasticsearch segment merging, MySQL InnoDB flushing, and k3s etcd operations all hitting disk simultaneously caused IOPS exhaustion. Fixed by: (1) adding mem_limit to all containers, (2) tuning Elasticsearch with async translog and 30s sync intervals, (3) migrating from Minikube to k3s which has less I/O overhead

- Logstash duplicate logs - sincedb_path=/dev/null caused Logstash to re-read entire log file on restart, creating I/O bursts and duplicate entries. Fixed by using persistent sincedb path

- ELK resource consumption - Elasticsearch uses memory-mapped files beyond JVM heap, consuming more RAM than expected. Added memory limits and I/O throttling (20MB/s) to prevent resource exhaustion

- Port conflicts - running all 3 orchestrators simultaneously required different ports. Configured Compose on 8080, Swarm on 8081, and K8s on 30080

- **Security alert from hosting provider** - received warning that Elasticsearch port 9200 was publicly accessible without authentication. Fixed by: (1) removing public port bindings for Elasticsearch and Logstash, (2) binding Kibana to localhost only (access via SSH tunnel), (3) adding automated port security tests to prevent regression. Key lesson: always review which ports are exposed to the internet

Key Learnings:

- Infrastructure as Code is powerful - all configs in version control means reproducible deployments

- Start simple, add complexity as needed - Compose for dev, K8s for production

- Observability is not optional - without logs, debugging distributed systems is nearly impossible

5.3 Conclusion

This project gave me hands-on experience with container orchestration at three different levels. I built a complete microservice from scratch with Spring Boot, containerised it with Docker, and deployed it using three orchestration tools.

Docker Compose is my choice for local development. It's simple, fast, and matches how I think about multi-container applications. The YAML syntax is intuitive, and I can go from code change to running container in under a minute.

Docker Swarm surprised me with its simplicity. For smaller production deployments where you need scaling and high availability but don't want the complexity of Kubernetes, Swarm is a solid choice. The transition from Compose to Swarm is nearly seamless.

Kubernetes is clearly the most powerful option. Features like HPA (auto-scaling), rolling updates with zero downtime, and declarative configuration make it ideal for enterprise production. The learning curve is steep, but the payoff is worth it for complex systems.

Production Recommendation: For a real production system, I would use Docker Compose during development and Kubernetes for deployment. I'd add a CI/CD pipeline (GitHub Actions or GitLab CI) to automate the build, test, and deploy process. Helm charts would manage environment-specific

configurations. The ELK stack or a managed logging service would provide observability.

6. AI Assistance Acknowledgment

Claude (Anthropic, 2025) was used for assistance with:

- Docker Compose configuration

- Kubernetes manifest templates

- Helm chart structure and templates

- ELK stack configuration

- Migration from Minikube to k3s

- Preparing this report