Cloud Deployment Orchestration with Kubernetes

Single-node Minikube Cloud App
with Automatic Load Balancing

M311 – Cloud Computing
Fall 2025

Instructor: **Mohamed Riduan ABID**

Authors:

Sami Agourram
Soufiane El Amrani

January 2, 2026

Contents

1 Introduction

This report documents the deployment of **Salam Queue Flow**, a healthcare queue management web application, on a local Kubernetes cluster using Minikube. The project demonstrates essential cloud orchestration concepts including:

- Containerization with Docker (multi-stage builds)
- Kubernetes Deployment with multiple replicas
- Service exposure via NodePort
- Automatic load balancing across pods
- Self-healing and resilience

2 Reflection

The deployment process began by starting a single-node Minikube cluster using Docker as the driver. This created a local Kubernetes environment that behaves identically to a production cluster.

The application was containerized using a multi-stage Dockerfile: the first stage compiles the React/TypeScript frontend, while the second stage runs a lightweight Express.js server that serves the static files and exposes a health API endpoint.

To achieve high availability, we deployed the application with **3 replicas** using a declarative Kubernetes Deployment manifest (`deployment.yaml`). The application was then exposed externally using a **NodePort Service** defined in `service.yaml`.

Load balancing was verified using two methods: (1) a visual pod indicator in the web UI that displays a unique color per pod, and (2) the `/api/health` endpoint that returns the serving pod's hostname. When sending 10 consecutive requests, we observed responses from different pods, confirming Kubernetes' round-robin load distribution.

Self-healing was demonstrated by manually deleting one pod and immediately observing Kubernetes automatically spawn a replacement, maintaining the desired 3-replica state within seconds.

3 Task 1: Start Minikube Cluster

3.1 Objective

Start Minikube locally and verify the cluster is running and ready.

3.2 Commands Executed

```
minikube start --driver=docker
minikube status
kubectl get nodes
```

3.3 Screenshots

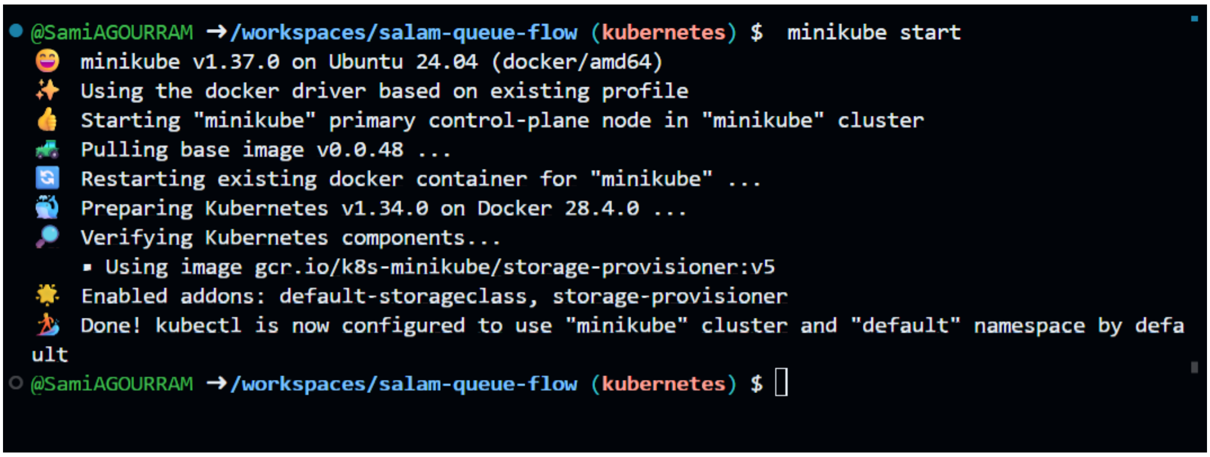

Figure 1: `minikube start` – Starting the Kubernetes cluster with Docker driver

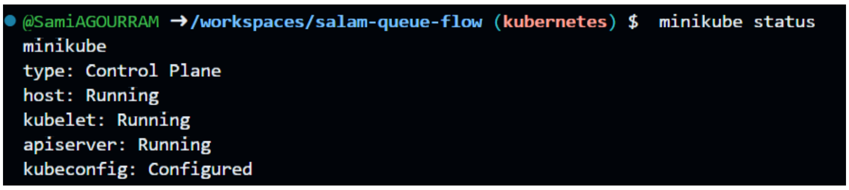

Figure 2: `minikube status` – Cluster components running (host, kubelet, apiserver)

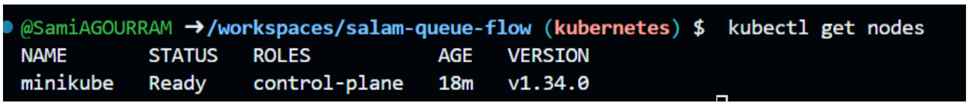

Figure 3: `kubectl get nodes` – Single node cluster ready with Kubernetes v1.34.0

4 Task 2: Deploy Application with Multiple Copies

4.1 Objective

Deploy the web application with at least 3 replicas running simultaneously.

4.2 Deployment Manifest

deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: salam-queue
spec:
  replicas: 3
```

```yaml
  selector:
    matchLabels:
      app: salam-queue
  template:
    metadata:
      labels:
        app: salam-queue
    spec:
      containers:
      - name: salam-queue
        image: salam-queue:v2
        imagePullPolicy: Never
        ports:
        - containerPort: 3000
```

4.3 Commands Executed

```
# Build image inside Minikube's Docker daemon
minikube image build -t salam-queue:v2 .

# Apply the deployment manifest
kubectl apply -f deployment.yaml

# Verify deployment and pods
kubectl get deployments
kubectl get pods
```

4.4 Screenshots

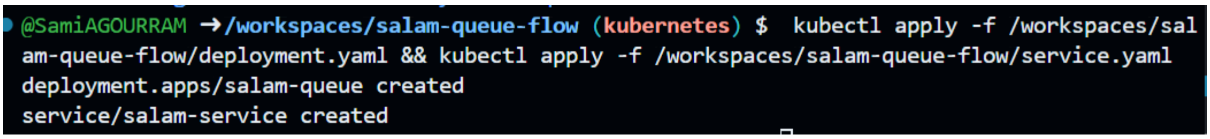

Figure 4: `kubectl apply` – Creating Deployment and Service from YAML manifests

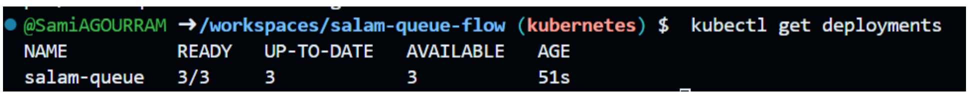

Figure 5: `kubectl get deployments` – Deployment shows 3/3 replicas ready

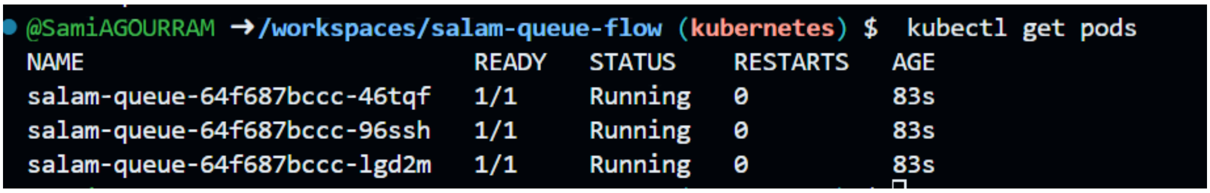

Figure 6: `kubectl get pods` – Three pods running with status 1/1 Ready

5 Task 3: Expose the Application

5.1 Objective

Make the application reachable from the host machine via browser or curl.

5.2 Service Manifest

service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: salam-service
spec:
  type: NodePort
  selector:
    app: salam-queue
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
      nodePort: 30080
```

5.3 Commands Executed

```
# Apply the service manifest
kubectl apply -f service.yaml

# List services
kubectl get services

# Get the external URL
minikube service salam-service --url
```

5.4 Screenshots

```
● @SamiAGOURRAM ➜ /workspaces/salam-queue-flow (kubernetes) $  kubectl get services
  NAME             TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
  kubernetes       ClusterIP   10.96.0.1       <none>        443/TCP        21m
  salam-service    NodePort    10.101.93.57    <none>        80:30080/TCP   2m4s
```

Figure 7: `kubectl get services` – NodePort service exposing port 80:30080/TCP

6 Task 4: Demonstrate Automatic Load Balancing

6.1 Objective

Prove that traffic is distributed across multiple running pods.

6.2 Implementation

The application includes two mechanisms to demonstrate load balancing:

1. **Health API Endpoint**: The `/api/health` endpoint returns JSON containing the pod's hostname, allowing programmatic verification.
2. **Visual Pod Indicator**: A colored badge in the UI displays the serving pod's ID with a unique color based on the pod name hash.

6.3 Health API Code

server.js (excerpt)

```
app.get('/api/health', (req, res) => {
    res.json({
        status: 'ok',
        message: 'Salam Queue is running!',
        pod_id: os.hostname(),
    });
});
```

6.4 Commands Executed

```
# Get the service URL
minikube service salam-service --url

# Send 10 requests to prove load balancing
for i in {1..10}; do
    curl -s http://192.168.49.2:30080/api/health
    echo ""
done
```

6.5 Screenshots

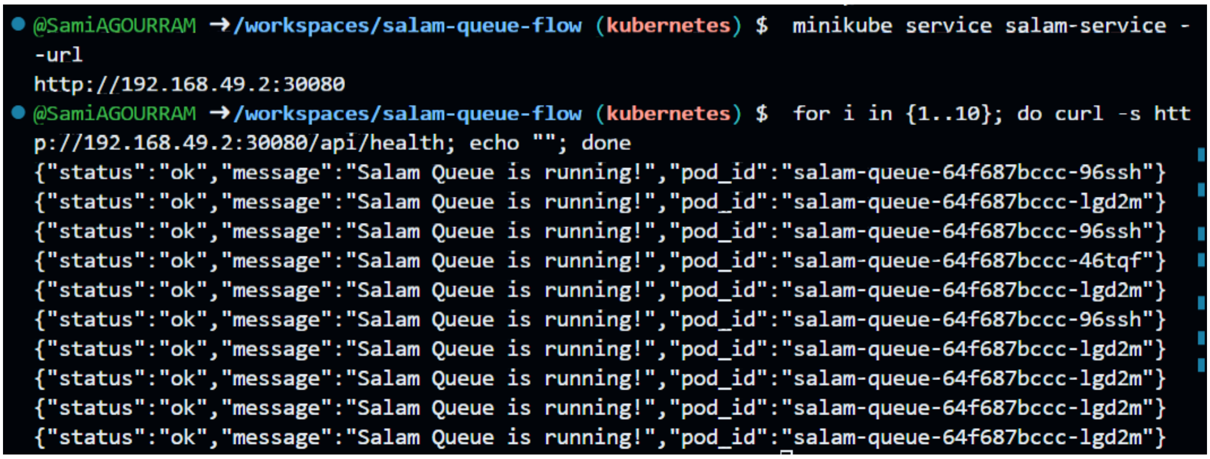

Figure 8: Terminal load balancing proof – Different `pod_id` values in consecutive responses (96ssh, lgd2m, 46tqf)

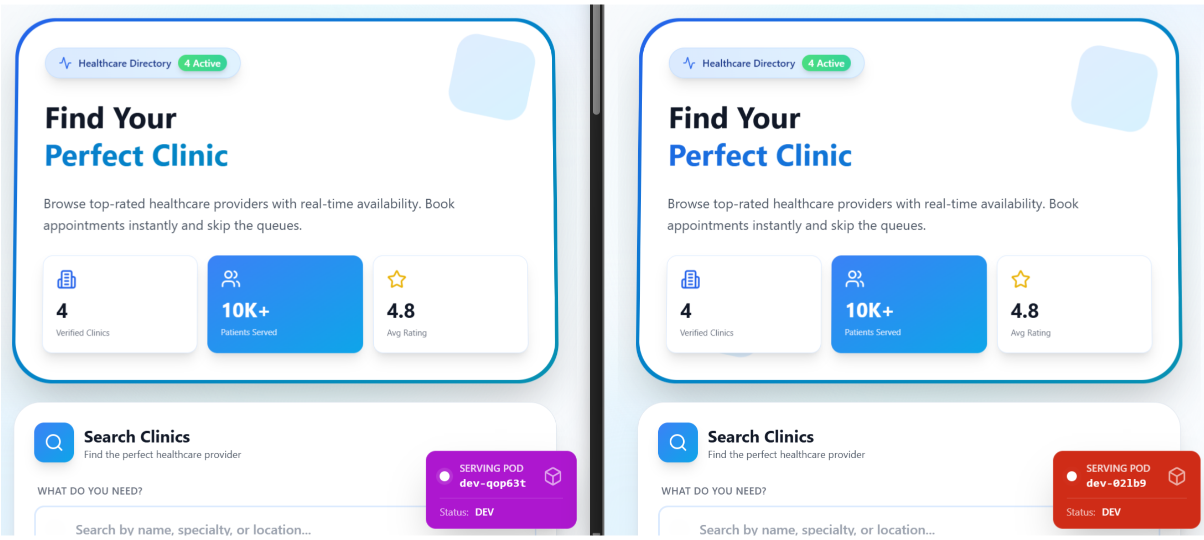

Figure 9: Visual load balancing proof – Pod indicator badge changes color and ID on page refresh (purple "dev-qop63t" vs red "dev-021b9")

6.6 Analysis

The terminal output clearly shows three different pod IDs responding to requests:

- `salam-queue-64f687bccc-96ssh`
- `salam-queue-64f687bccc-lgd2m`
- `salam-queue-64f687bccc-46tqf`

This confirms that Kubernetes is distributing traffic across all three replicas using its internal load balancing mechanism.

7 Task 5: Demonstrate Self-Healing (Resilience)

7.1 Objective

Show that Kubernetes automatically recreates deleted pods to maintain the desired replica count.

7.2 Commands Executed

```
# List pods before deletion
kubectl get pods

# Delete one pod
kubectl delete pod salam-queue-64f687bccc-46tqf

# Immediately list pods again
kubectl get pods
```

7.3 Screenshot

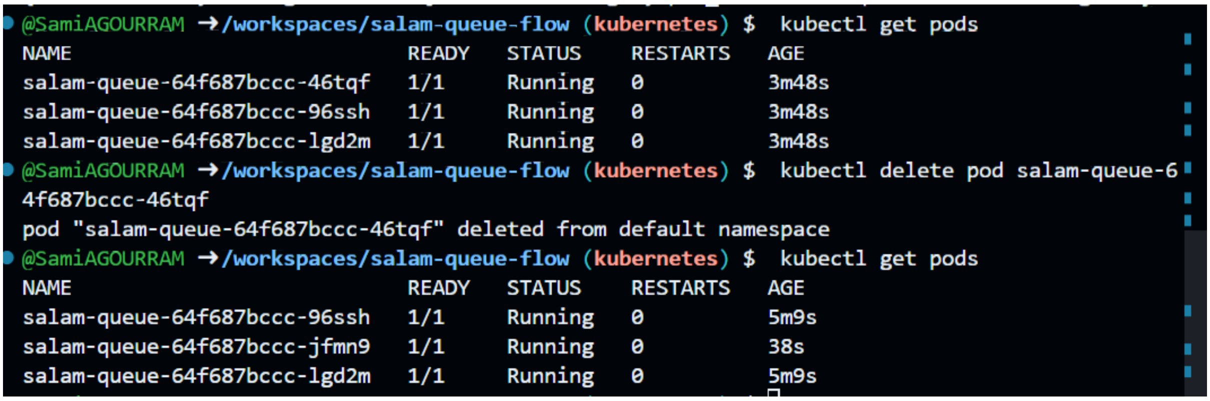

Figure 10: Self-healing demonstration – Pod "46tqf" deleted, Kubernetes immediately creates "jfmn9" (38s old) to maintain 3 replicas

7.4 Analysis

The screenshot shows the complete self-healing cycle:

1. **Before**: Three pods running (46tqf, 96ssh, lgd2m) at 3m48s age
2. **Delete**: Pod "salam-queue-64f687bccc-46tqf" is manually deleted
3. **After**: A new pod "salam-queue-64f687bccc-jfmn9" appears (38s old), while the other two pods remain (5m9s old)

Kubernetes detected the missing replica and automatically scheduled a new pod within seconds, demonstrating the self-healing capability of Deployments.

8 Project Files

8.1 Dockerfile

```
# Stage 1: Builder (Compile React)
FROM node:18-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build

# Stage 2: Runner (Lightweight Server)
FROM node:18-alpine AS runner
WORKDIR /app
RUN npm install express
COPY --from=builder /app/dist ./dist
COPY server.js .
EXPOSE 3000
CMD ["node", "server.js"]
```

8.2 Server.js

```
const express = require('express');
const path = require('path');
const os = require('os');

const app = express();
const PORT = 3000;

// Health endpoint for load balancing proof
app.get('/api/health', (req, res) => {
    res.json({
        status: 'ok',
        message: 'Salam Queue is running!',
        pod_id: os.hostname(),
    });
});

// Serve static files
app.use(express.static(path.join(__dirname, 'dist')));

// React Router catch-all
app.get(/.*/, (req, res) => {
    res.sendFile(path.join(__dirname, 'dist', 'index.html'));
});

app.listen(PORT, () => {
    console.log(`Server on port ${PORT}. Pod: ${os.hostname()}`);
});
```

8.3 Deployment YAML

deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: salam-queue
spec:
  replicas: 3
  selector:
    matchLabels:
      app: salam-queue
  template:
    metadata:
      labels:
        app: salam-queue
    spec:
      containers:
      - name: salam-queue
        image: salam-queue:v2
        imagePullPolicy: Never
        ports:
        - containerPort: 3000
```

8.4 Service YAML

service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: salam-service
spec:
  type: NodePort
  selector:
    app: salam-queue
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
      nodePort: 30080
```

9 Conclusion

This project successfully demonstrates the core capabilities of Kubernetes orchestration:

Success Checklist

- ✓ **Minikube** runs locally as a single-node cluster
- ✓ **Application** runs in 3 identical replicas
- ✓ **Service** is reachable via NodePort (30080)
- ✓ **Load balancing** proven via API and visual indicator
- ✓ **Self-healing** demonstrated after pod deletion
- ✓ **Declarative configuration** using YAML manifests

The addition of a visual pod indicator in the web UI provides an intuitive way to demonstrate load balancing during live demonstrations, complementing the traditional API-based verification approach.