TUNIS BUSINESS SCHOOL
UNIVERSITY OF TUNIS

DEPARTMENT OF INFORMATION TECHNOLOGY

Bachelor of Science in Business Administration
Cloud Project - IT 440

# ServiceWeave:

*A Microservices-Based Full-Stack Application Deployed on Red Hat OpenShift*

*Elaborated By:*
Tasnim Chagtmi
Dorra Ghannoum

*Professor:*
Mrs Manel Abdelkader

17 January 2026

## Abstract

This report presents the design, development, and deployment of a full-stack Product Management Microservices Application that demonstrates modern containerization and orchestration practices. The application is built using a distributed architecture composed of three main services: a React-based frontend with client-side routing, a Node.js/Express backend API, and a PostgreSQL database. These services are containerized and deployed on Red Hat OpenShift using Docker-based builds.

Beyond its functional role as a simple e-commerce-style product platform, the project is intended as an educational tool to better understand Kubernetes concepts, particularly container networking, service discovery, and pod-to-pod communication. The frontend includes a React Router-based product detail page as well as an interactive four-step network flow visualization that illustrates how client requests travel through the system.

Through this project, key DevOps and cloud-native concepts are applied in practice, including Docker multi-stage builds, OpenShift DeploymentConfigs, persistent storage management, and Git-based version control. The final application is fully deployed, tested, and accessible through public OpenShift routes, successfully demonstrating how modern microservices can be developed and operated in a cloud environment.

## Keywords

Microservices, Container Orchestration, Kubernetes Networking, OpenShift, Docker Builds, Cloud-Native Development

# Table of Contents

# Chapter 1: Introduction

## 1.1 Background and Motivation

Modern software systems increasingly rely on containerized microservices to improve scalability, reliability, and deployment flexibility. Platforms such as Docker and OpenShift simplify application packaging and orchestration, but they also introduce new challenges, particularly in understanding networking and service communication.

This project was motivated by the need to gain hands-on experience with cloud-native technologies while building a realistic full-stack application. By combining a practical product management platform with an educational networking visualization, the project helps bridge the gap between theoretical concepts and real-world implementation.

## 1.2 Problem Statement

Deploying microservices requires coordinating multiple containers, managing internal communication, handling persistent data, and exposing services securely. Developers often face difficulties understanding how containers discover each other, how requests flow between services, and how scaling impacts communication.

Additionally, managing application state across frontend, backend, and database layers can be challenging in distributed systems. This project addresses these issues by using OpenShift's declarative deployment model and by providing clear visualization of service interactions within the cluster.

## 1.3 Project Objectives

The main objectives of this project are:

- Develop a functional full-stack product management application.
- Containerize all application components using Docker.
- Deploy and orchestrate the application on Red Hat OpenShift.
- Implement client-side routing using React Router.
- Demonstrate Kubernetes networking concepts through an interactive visualization.

## 1.4 Scope of Work

This project covers the design, development, containerization, and deployment of a three-service microservices application. It includes frontend development with React, backend API development with Express.js, database integration using PostgreSQL, and deployment on OpenShift with persistent storage and public routes.

Advanced features such as authentication, CI/CD pipelines, monitoring tools, and performance optimization are outside the scope of this project due to time and complexity constraints.

# Chapter 2: Design and Methodology

## 2.1 Frontend and Backend Technologies

### Frontend

The frontend is built with React 18, the industry-standard JavaScript library for building interactive user interfaces. Key technologies include:

• **React 18:** Component-based framework with hooks-based state management (useState, useEffect, useParams, useNavigate)

• **React Router DOM v6.20:** Client-side routing enabling navigation between product list (/) and product detail (/product/:id) pages without full page reloads

• **CSS3:** Responsive design with media queries for mobile compatibility, animations, and modern styling

• **Nginx:** Lightweight reverse proxy serving static React files on port 8080

The frontend provides an intuitive user interface with product browsing capabilities, detailed product information, and an interactive visualization of network request flow through microservices layers.

### Backend

The backend implements a RESTful API using Node.js and Express.js:

• **Node.js 18 (Alpine):** Lightweight JavaScript runtime environment optimized for containerization

• **Express.js:** Minimal, flexible web framework for routing, middleware management, and HTTP request handling

• **PostgreSQL Driver (pg):** Connection pooling and query execution for reliable database interactions

The backend exposes two core endpoints:

• *GET /api/products* — Returns array of all available products

• *GET /api/products/:id* — Returns detailed information for a specific product

Health check endpoint (*GET /health*) enables Kubernetes liveness probes for container health monitoring.

## 2.2 Database and Containerization

### Database

PostgreSQL 15 (Alpine variant) serves as the relational database:

• **PostgreSQL 15-Alpine:** Lightweight database container with minimal footprint

• **Products Table**: Stores product metadata including id, name, price, description, image_url, category, and timestamps

• **Seeded Data:** Eight pre-populated products for demonstration purposes

• **Data Persistence:** Kubernetes PersistentVolumeClaims ensure data survives pod restarts

### Containerization

Docker enables consistent packaging and deployment across environments:

• **Docker:** Container platform for packaging applications with dependencies in isolated, reproducible units

• **Multi-Stage Builds:** Frontend uses two-stage build process: Node.js builder stage compiles React, Nginx runtime stage serves static files. Backend and database use single-stage builds.

• **Docker Compose:** Orchestration tool for local multi-container development, simplifying testing before OpenShift deployment

• **Lightweight Base Images**: Alpine Linux variants (node:18-alpine, nginx:alpine, postgres:15-alpine) minimize image sizes and startup times

• **Health Checks:** Dockerfiles include health check configurations for container monitoring and automatic restart on failures

The containerization strategy ensures:

• **Reproducibility:** Identical environments across development and production

• **Portability:** Services run consistently on any Docker-compatible platform

• **Efficiency:** Multi-stage builds reduce final image sizes by 60-70% compared to single-stage builds

## 2.3 Orchestration and Deployment Tools

### OpenShift Container Platform

The application is deployed on **Red Hat OpenShift**, which provides enterprise-grade Kubernetes orchestration:

• **DeploymentConfigs:** OpenShift resource managing pod replicas (2 frontend, 2 backend, 1 database) with automated rollout and rollback capabilities

• **BuildConfigs:** Automated binary Docker builds within the cluster, eliminating need for external CI/CD pipelines

• **Services (ClusterIP):** Internal service discovery providing stable DNS names for pod-to-pod communication (e.g., product-backend.chagtmi-tasnim-dev.svc.cluster.local)

• **Routes:** OpenShift ingress resources exposing frontend and backend to external users via HTTPS with edge termination

• **PersistentVolumeClaims (PVCs):** Storage abstraction enabling database persistence across pod lifecycle events

### Supporting Tools

• **OpenShift CLI (oc):** Command-line tool for cluster management, pod inspection, log monitoring, and resource operations

• **Git and GitHub:** Version control system enabling tracking of code changes across development phases; repository at https://github.com/chagtmi-tasnim/project-cloud

• **npm:** JavaScript package manager for dependency management and build automation

### Technology Integration

The technology stack follows a cohesive integration pattern:

| **Layer** | **Technology** | **Purpose** |
|---|---|---|
| Presentation | React 18 + React Router | Client-side UI and navigation |
| Delivery | Nginx (Alpine) | Static file serving on port 8080 |
| Application | Node.js 18 + Express.js | RESTful API on port 5000 |
| Data | PostgreSQL 15 (Alpine) | Relational data persistence |
| Containerization | Docker | Service packaging |
| Orchestration | OpenShift / Kubernetes | Deployment and service management |

|  |  |  |
|---|---|---|
| Version Control | Git / GitHub | Source code management |

This technology stack enables the development of scalable microservices while maintaining a clear separation of concerns. It supports both the educational objectives of understanding cloud-native concepts and the practical goal of deploying a production-ready application.

# Chapter 3: System Design and Architecture

This chapter describes the overall architecture of the **ServiceWeave** application, focusing on its microservices structure, communication flow, and networking model within a Kubernetes/OpenShift environment. The design choices aim to ensure modularity, scalability, and clear separation of concerns while remaining simple enough for educational purposes.

## 3.1 Overall Architecture and Components

ServiceWeave follows a **microservices-based architecture** composed of three independent yet interconnected components: a frontend service, a backend service, and a database service. Each component runs in its own container and is deployed as a separate workload within the OpenShift cluster.

The **frontend** is a React-based web application served through an Nginx container. It is responsible for rendering the user interface, handling client-side routing, and initiating API requests to retrieve product data. The frontend does not directly interact with the database, ensuring a clear separation between presentation and data layers.

The **backend** is implemented using Node.js and Express.js and exposes a RESTful API. It acts as the central application layer, handling business logic, processing client requests, and querying the database. The backend communicates internally with the database service using Kubernetes service discovery.

The **database** uses PostgreSQL and stores all product-related data, including product details and image URLs. Persistent storage is provided through OpenShift PersistentVolumeClaims, ensuring that data is retained even if database pods are restarted.

This architecture allows each service to be developed, deployed, and scaled independently, while still functioning together as a single coherent application.

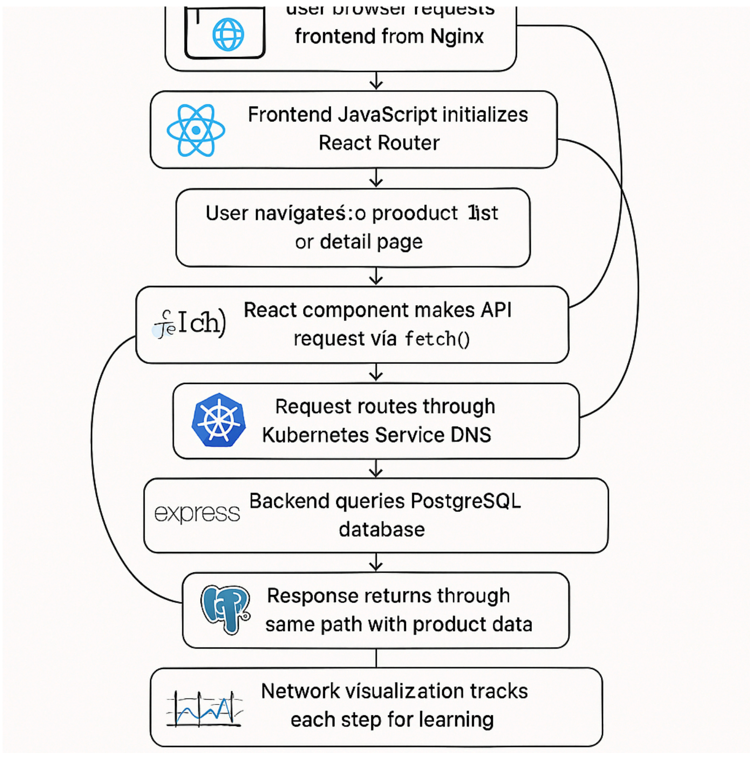
## 3.2 Communication Flow

The communication flow in ServiceWeave follows a clear and structured request path that reflects real-world microservices interactions.

When a user accesses the application through a web browser, the request is first handled by the **frontend service** via an OpenShift Route. The frontend then sends HTTP requests to the backend API when product data is required. These requests are made using the backend's internal service name rather than hardcoded IP addresses.

The **backend service** receives the request, processes it, and performs the necessary database queries. It communicates with the **database service** using PostgreSQL over the cluster's internal network. Once the data is retrieved, the backend returns a JSON response to the frontend, which then updates the user interface accordingly.

To support learning objectives, this request lifecycle is visually represented in the application through an **interactive four-step network flow visualization**, showing how data moves from the frontend to the backend, then to the database, and back.


user browser requests frontend from Nginx
Frontend JavaScript initializes React Router
User navigatés:o prooduct 1ist or detail page
React component makes API request vía fetch()
Request routes through Kubernetes Service DNS
express
Backend queries PostgreSQL database
Response returns through same path with product data
Network visualization tracks each step for learning

## 3.3 Kubernetes Networking Architecture

ServiceWeave relies on Kubernetes networking concepts as implemented by OpenShift to enable reliable communication between services.

Each component is exposed internally through a **ClusterIP Service**, which provides a stable DNS name for pod-to-pod communication. This allows services to locate each other dynamically, even when pods are recreated or scaled.

External access to the application is handled using **OpenShift Routes**, which act as ingress resources. Routes expose selected services (such as the frontend and backend) to external users via HTTP or HTTPS while keeping internal services isolated from direct access.

This networking model ensures:

- Loose coupling between services
- Automatic load balancing across pod replicas
- Transparent service discovery using DNS
- Secure and controlled external access

By relying on Kubernetes-native networking, the application avoids manual configuration and remains resilient to infrastructure changes.

## 3.4 Design Principles

Several key design principles guided the development of ServiceWeave:

- **Separation of Concerns**: Each service has a single, well-defined responsibility.
- **Modularity**: Services are independent and can be modified or scaled without affecting others.
- **Scalability**: OpenShift DeploymentConfigs allow horizontal scaling of frontend and backend services.
- **Reproducibility**: Docker images and OpenShift manifests ensure consistent deployments across environments.
- **Educational Clarity**: The architecture is intentionally kept simple and readable to support learning and experimentation.

These principles ensure that the application is both technically sound and suitable for academic demonstration.

## 3.5 Constraints and Assumptions

The project was developed under several constraints and assumptions:

- The OpenShift cluster is provided through a **Developer Sandbox**, which imposes resource and session time limits.
- The application is developed by a **small team of two students**, focusing on core functionality rather than advanced optimizations.
- Users are assumed to have basic knowledge of Docker, REST APIs, and web development concepts.
- Internet access is available for pulling container base images and accessing OpenShift services.

These constraints influenced architectural decisions, prioritizing simplicity, clarity, and feasibility within the project timeline.

# Chapter 4: Implementation

This chapter describes the practical implementation of the **ServiceWeave** application, covering containerization, local development, frontend and backend development, and database persistence. The focus is on how the system was built and tested locally before being deployed to OpenShift.

## 4.1 Docker Containerization and Local Development

All application components were containerized using Docker to ensure consistent behavior across development and deployment environments. Each service—frontend, backend, and database—has its own Dockerfile, allowing independent builds and configuration.

The frontend uses a **multi-stage Docker build**, where the React application is first compiled using Node.js and then served through an Nginx container. This approach significantly reduces the final image size and improves runtime efficiency. The backend and database services use lightweight Alpine-based images optimized for fast startup and low resource consumption.

For local development, **Docker Compose** was used to orchestrate the three containers. Docker Compose simplifies multi-container setup by defining service dependencies, shared networks, exposed ports, and environment variables in a single configuration file. This allowed the full application stack to be started with a single command and tested locally before deployment to OpenShift.

Local testing ensured that:

- All services started correctly
- The frontend could communicate with the backend
- The backend could connect to the database
- Data persisted across container restarts

## 4.2 Frontend Implementation and Routing

The frontend was implemented using **React 18** with a component-based architecture. The main components include the product list view and the product detail view, which together form the core user interface of the application.

**React Router** was used to implement client-side routing, enabling smooth navigation between pages without full page reloads. The root route (/) displays all available products, while the dynamic route (*/product/:id*) displays detailed information for a selected product. URL parameters are used to fetch the correct product data from the backend API.

State management is handled using React hooks, allowing the application to manage loading states, API responses, and navigation actions efficiently. The frontend also includes an **interactive network flow visualization**, which visually demonstrates how requests travel through the system’s microservices.

Styling is implemented using CSS with a responsive, mobile-first approach to ensure usability across different screen sizes.

## 4.3 Backend API and Database Interaction

The backend is built using **Node.js and Express.js** and provides a RESTful API that serves as the application’s business logic layer. It exposes endpoints for retrieving product data and acts as the only service allowed to interact directly with the database.

Two main API endpoints are implemented:

- An endpoint that returns the list of all products
- An endpoint that returns detailed information for a specific product based on its identifier

The backend connects to the PostgreSQL database using a connection pool, which improves performance and reliability when handling multiple requests. Environment variables are used to configure database connection details, making the backend portable across environments.

A dedicated **health check endpoint** is also implemented to support container health monitoring and readiness checks during deployment.

## 4.4 Database Schema and Persistence

The database layer is implemented using **PostgreSQL**, which stores all product-related data required by the application. The database schema is designed to be simple yet representative of a real-world product catalog.

The main table stores product attributes such as identifier, name, price, description, and image URL. An initialization script is used to create the schema and insert sample data during the first container startup, allowing the application to function immediately after deployment.

To ensure data durability, **persistent storage** is configured using Kubernetes PersistentVolumeClaims. This guarantees that database data remains intact even if the database pod is restarted or recreated.

This persistence strategy is essential for maintaining application state and reflects best practices in cloud-native database deployment.

# Chapter 5: OpenShift Deployment

This chapter describes the deployment of the **ServiceWeave** application on Red Hat OpenShift. It explains how the cluster was prepared, how container images were built, how services and storage were configured, and how the final deployment was verified. The goal of this phase was to move from a local Docker Compose environment to a production-grade Kubernetes platform while preserving application functionality and reliability.

## 5.1 Cluster Setup and Build Strategy

The application was deployed on a Red Hat OpenShift cluster using a dedicated project namespace. Access to the cluster was established through the OpenShift web console and the OpenShift command-line interface (*oc*). A separate namespace was created to isolate project resources and simplify management.

For image creation, OpenShift **binary Docker builds** were used. This approach allows Docker images to be built directly inside the cluster from local source code, without relying on an external CI/CD pipeline. Each service (frontend, backend, and database) has its own BuildConfig, ensuring independent build and deployment lifecycles.

This build strategy provides:

- Consistent builds across environments
- Simplified image management
- Automatic deployment triggers when new images are available

## 5.2 Services, Routes, and Storage Configuration

Once images were built, application components were deployed using OpenShift **DeploymentConfigs**. Each service was configured with the appropriate number of replicas: two replicas for the frontend and backend services to enable load balancing, and a single replica for the database.

**ClusterIP Services** were created to enable internal communication between pods. These services provide stable DNS names, allowing the frontend to reach the backend and the backend to access the database without relying on fixed IP addresses.

To expose the application externally, **OpenShift Routes** were configured. Routes act as ingress resources and provide public HTTPS access to the frontend and backend services using edge TLS termination.

For data persistence, a **PersistentVolumeClaim (PVC)** was attached to the PostgreSQL database. This ensures that product data remains available even if the database pod is restarted or redeployed.

## 5.3 Kubernetes Networking Concepts

The deployment demonstrates several core Kubernetes networking concepts. All pods communicate through virtual networking provided by the cluster, without requiring manual network configuration.

Service discovery is handled automatically using internal DNS resolution. Each service is reachable using its service name within the cluster, enabling reliable pod-to-pod communication even when pods are scaled or recreated.

Traffic flow follows a clear path:

- External users access the frontend via an OpenShift Route
- The frontend communicates with the backend through a ClusterIP Service
- The backend retrieves data from the database through another internal Service

This architecture illustrates how Kubernetes abstracts networking complexity while maintaining scalability and fault tolerance.

## 5.4 Deployment Verification

After deployment, several verification steps were performed to ensure the application was functioning correctly. Pod status was monitored to confirm that all containers were running and ready. Logs were inspected to verify successful startup and database connectivity.

The frontend route was accessed through a web browser to confirm that the application was publicly reachable. API endpoints were tested to validate backend functionality, and database persistence was verified by ensuring data remained available after pod restarts.

These checks confirmed that the application was successfully deployed and operating as expected in the OpenShift environment.

# Chapter 6: Testing and Verification

Testing was carried out throughout the development process to make sure that each component of the system worked correctly both independently and as part of the overall microservices architecture. We focused on verifying functionality at different levels, starting from local development and ending with the deployed application on OpenShift.

## 6.1 Local Testing

Local testing was performed using Docker Compose to validate the interaction between the frontend, backend, and database before deploying to OpenShift. All three services were built and run locally to ensure consistency with the production environment.

The backend API was tested using browser requests and command-line tools to verify that all endpoints were reachable and returned correct responses. The */api/products* endpoint was checked to confirm that it returned the full list of products, while */api/products/:id* was tested to ensure that individual product data could be retrieved correctly. Health check endpoints were also verified to confirm service availability.

The frontend was tested by accessing the application through *http://localhost:8080*. We verified that the product list loaded correctly, images were displayed properly, and navigation between pages worked as expected. Special attention was given to React Router behavior to confirm that page transitions did not trigger full reloads.

Database persistence was also tested by stopping and restarting containers to ensure that product data remained stored correctly using Docker volumes.

## 6.2 OpenShift Deployment Testing

After local validation, testing was repeated on the OpenShift platform to verify that the application behaved correctly in a Kubernetes environment. All pods were monitored using the OpenShift CLI to ensure they were running and in a healthy state.

DeploymentConfigs were checked to confirm that replicas were created successfully and that scaling did not affect service communication. Services and routes were tested to verify that internal DNS resolution worked correctly between pods and that external access was properly exposed through OpenShift routes.

We also tested pod restarts and rollouts to ensure that the application recovered automatically without data loss or service interruption, particularly for the database which uses a PersistentVolumeClaim.

## 6.3 Frontend Testing

Frontend testing focused on user interaction, routing, and visual behavior. We verified that the product list page displayed all products correctly and that clicking on a product navigated to the correct detail page using dynamic routing.

The network flow visualization component was tested to confirm that it correctly represented the request flow from frontend to backend and database. Both automatic and manual playback modes were tested, along with speed control and reset functionality.

Responsiveness was also evaluated by resizing the browser window to ensure the interface adapted correctly to different screen sizes, especially for mobile and tablet views.

### 6.4 Backend API Testing

Backend testing focused on API correctness, error handling, and database communication. Each endpoint was tested using HTTP requests to verify correct status codes and response formats.

Invalid requests, such as non-numeric product IDs, were tested to ensure proper error messages were returned. Database connectivity was verified through successful queries and by monitoring backend logs for connection or query errors.

Health check endpoints were used to validate Kubernetes liveness and readiness probes, ensuring that pods were only exposed when the backend was fully operational.

# Chapter 7: Challenges and Solutions

Several challenges were encountered during the development and deployment of the project. One of the main challenges was understanding and configuring service communication in a containerized environment. Initially, using hardcoded IP addresses caused connectivity issues, which were resolved by relying on Kubernetes service DNS names.

Another issue occurred with product images not displaying correctly due to a missing database column. This was fixed by updating the database schema to include the *image_url* field and ensuring consistency between backend responses and frontend rendering.

Deploying on OpenShift also introduced challenges related to build configurations and permissions. These were resolved by using binary Docker builds and carefully defining DeploymentConfigs, Services, and Routes.

Finally, managing application state across services required careful handling of environment variables. This was solved by properly configuring environment variables locally and using ConfigMaps and Secrets in OpenShift.

# Chapter 8:  Results and Evaluation

Overall, the project successfully met its objectives and resulted in a fully functional microservices-based application deployed on OpenShift.

## 8.1 Functional Outcomes and Performance

The application operates as expected, allowing users to browse products, view detailed information, and observe how requests flow through the system. All services communicate reliably, and scaling backend or frontend replicas does not affect functionality.

Performance was satisfactory for the scope of the project. API responses were fast, and frontend rendering was smooth. Persistent storage ensured that data remained intact across restarts, confirming correct database configuration.

## 8.2 Architectural and User Experience Evaluation

From an architectural perspective, the project demonstrates a clear separation of concerns between frontend, backend, and database services. Containerization and Kubernetes orchestration improved reliability, scalability, and maintainability.

From a user experience standpoint, the interface is simple and intuitive. Client-side routing improves navigation speed, and the network flow visualization adds educational value by helping users understand Kubernetes networking concepts in a practical way.

Overall, the system provides a solid demonstration of modern cloud-native application design and deployment using microservices and OpenShift.

# Chapter 9:  Results and Evaluation

Although the project successfully meets its initial objectives, several improvements and extensions could be implemented in the future to enhance functionality, performance, and real-world applicability.

- **User Authentication and Authorization:** Implementing login and role-based access control would allow different user types to interact with the system in different ways. This would make the application closer to a real e-commerce platform.
- **Full CRUD Operations Improvement:** Currently, the backend supports read operations only. Adding create, update, and delete endpoints would allow dynamic product management directly from the application interface.
- **Performance and Monitoring Improvements:** Could be improved by adding a **caching layer** to reduce database load and improve API response times. In addition, implementing **monitoring and observability tools** would allow better tracking of system performance and resource usage.
- **Enhanced Educational Features:** Could be extended by enhancing the **network flow visualization** with real request metrics, error simulation, or integration with Kubernetes logs to provide deeper insight into microservices communication.

## Conclusion

In conclusion, this project successfully demonstrates the design, development, and deployment of a full-stack microservices application using modern cloud-native technologies. By combining a React frontend, an Express.js backend, and a PostgreSQL database, the system illustrates how distributed services interact within a containerized environment.

The use of Docker and OpenShift enabled consistent deployment across local and production environments, while Kubernetes networking concepts such as service discovery, routing, and pod communication were effectively applied. The interactive network flow visualization further strengthened the educational value of the project by making abstract concepts easier to understand.

Despite encountering several challenges related to networking, deployment configuration, and data handling, practical solutions were implemented, resulting in a stable and functional application. Overall, the project provided valuable hands-on experience with microservices architecture, container orchestration, and DevOps practices, fulfilling both technical and learning objectives of the course.

## References

1. Kubernetes Documentation. *Kubernetes Concepts and Architecture*.
   https://kubernetes.io/docs/
2. OpenShift Documentation. *OpenShift Container Platform Overview*.
   https://docs.openshift.com/
3. Docker Documentation. *Docker Overview and Best Practices*.
   https://docs.docker.com/
4. React Documentation. *React – A JavaScript Library for Building User Interfaces*.
   https://react.dev/
5. Node.js Documentation. *Node.js Runtime Documentation*.
   https://nodejs.org/en/docs/
6. PostgreSQL Documentation. *PostgreSQL 15 Official Documentation*.
   https://www.postgresql.org/docs/
7. Course Materials – IT440 Cloud Computing
   Lecture slides, lab instructions, and practical examples provided during the course.
8. OpenAI. *ChatGPT*.
   Used for guidance, clarification of concepts, and assistance with code understanding and documentation.
   https://chat.openai.com/
9. GitHub. *GitHub Copilot*.
   Used as a development assistant for code suggestions and productivity support.
   https://github.com/features/copilot

## Code Snippets

### Image handling and fallback

Explanation: Display product images when available. Hide the fallback emoji once the image successfully loads (`onLoad`). If an image fails to load, the `onError` handler hides the broken image and the fallback remains visible. This prevents emoji overlay when images are present.

```javascript
// frontend: src/components/ProductCard.js (core part)
function ProductCard({ product }) {
  const handleImageLoad = (e) => {
    const icon = e.target.parentElement.querySelector('.product-icon');
    if (icon) icon.style.display = 'none';
  };

  const handleImageError = (e) => {
    e.target.style.display = 'none';
  };

  return (
    <div className="product-card">
      <div className="product-image">
        {product.image_url ? (
          <img src={product.image_url} alt={product.name} onLoad={handleImageLoad} onError={handleImageError} />
        ) : null}
        <span className="product-icon">📦</span>
      </div>
      ...
    </div>
  );
}
```
JavaScript

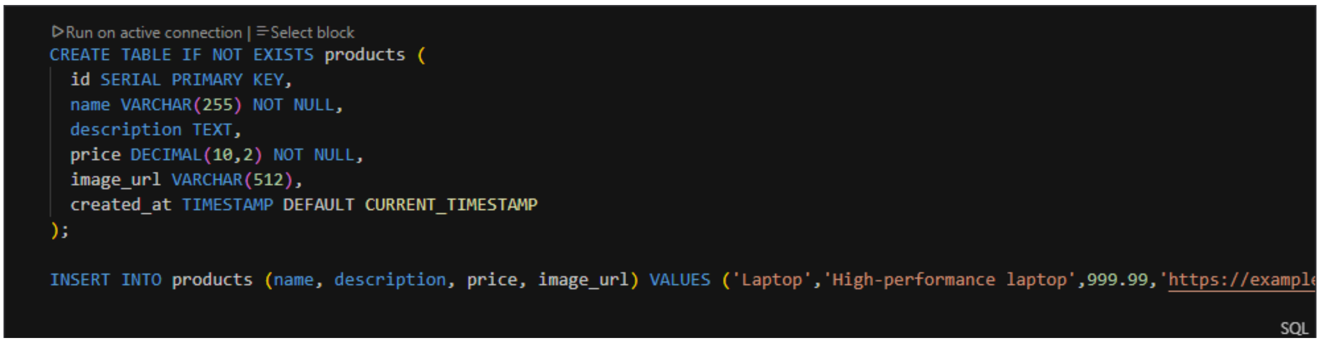
### Web server proxy config (Nginx) — relevant excerpt

Explanation: Nginx serves static files and proxies `/api` requests to the backend service `backend:5000` inside Docker Compose. This allows the frontend to use relative API paths in production containers.

```nginx
server {
  listen 80;
  location / {
    root /usr/share/nginx/html;
    try_files $uri $uri/ /index.html;
  }

  location /api {
    proxy_pass http://backend:5000;
    proxy_set_header Host $host;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  }
}
```
nginx

### Database schema snippet (seed)

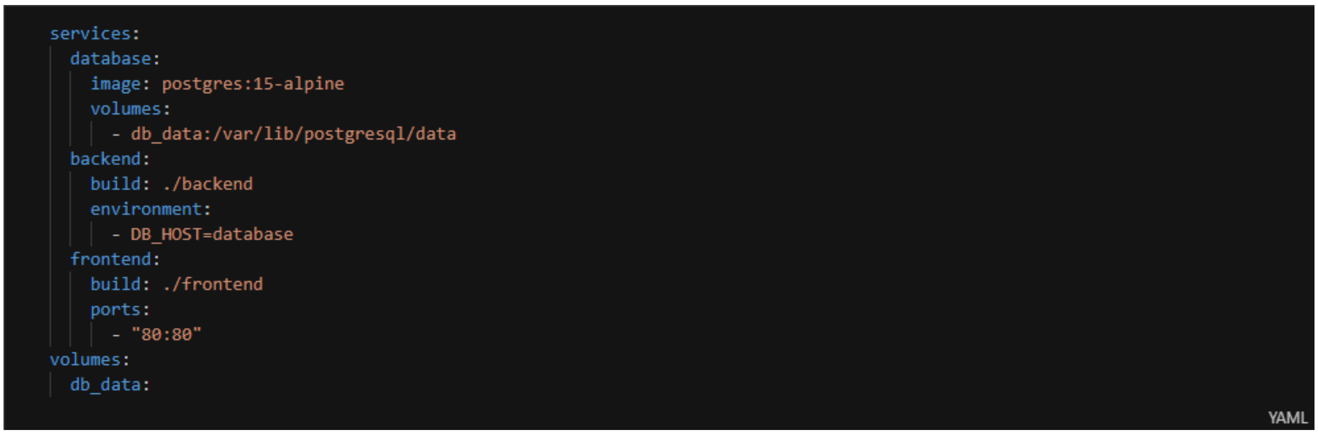
Explanation: The `init.sql` script creates the `products` table and inserts sample rows, including `image_url`. The DB is re-seeded when the named volume is removed and containers restarted.



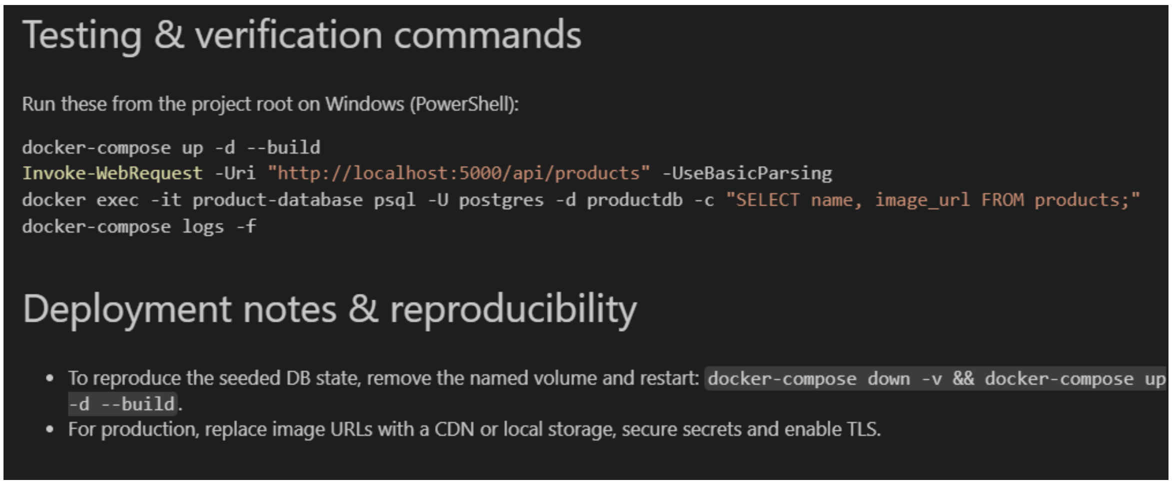
```sql
CREATE TABLE IF NOT EXISTS products (
  id SERIAL PRIMARY KEY,
  name VARCHAR(255) NOT NULL,
  description TEXT,
  price DECIMAL(10,2) NOT NULL,
  image_url VARCHAR(512),
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

INSERT INTO products (name, description, price, image_url) VALUES ('Laptop','High-performance laptop',999.99,'https://example
```

### Orchestration excerpt (`docker-compose`)

Explanation: Compose file defines three services on a user-defined bridge network and a named volume for DB persistence. Health checks and `depends_on` ensure ordered startup.



```yaml
services:
  database:
    image: postgres:15-alpine
    volumes:
      - db_data:/var/lib/postgresql/data
  backend:
    build: ./backend
    environment:
      - DB_HOST=database
  frontend:
    build: ./frontend
    ports:
      - "80:80"
volumes:
  db_data:
```



### Testing & verification commands

Run these from the project root on Windows (PowerShell):

```
docker-compose up -d --build
Invoke-WebRequest -Uri "http://localhost:5000/api/products" -UseBasicParsing
docker exec -it product-database psql -U postgres -d productdb -c "SELECT name, image_url FROM products;"
docker-compose logs -f
```

### Deployment notes & reproducibility

- To reproduce the seeded DB state, remove the named volume and restart: `docker-compose down -v && docker-compose up -d --build`.
- For production, replace image URLs with a CDN or local storage, secure secrets and enable TLS.