

CS 816 – Software Production Engineering

SPE Project – Doctor Appointment Booking System with DevOps

Uttam Hamsaraj – IMT2022524
Ayush Arya Kashyap – IMT2022129
Pranav Laddhad – IMT2022074

December 2025

Important Links

GitHub Repo Link: [GitHub Repository](#)
DockerHub Frontend Image Repo Link: [DockerHub Repository](#)
DockerHub Backend Image Repo Link: [DockerHub Repository](#)

Introduction

This project is a Doctor Appointment Booking Application designed to streamline the scheduling and management of appointments between patients and doctors. To ensure reliability, scalability, and maintainability, the system incorporates DevOps best practices such as Jenkins, Docker, Ansible, Kubernetes and ELK-based logging.

Objectives

Objectives of this project are:

- Develop a full stack web application for doctor–patient appointment management using the MERN stack.
- Set up version control with Git and integrate with remote repository on GitHub.
- Containerized the frontend, backend, and database services using Docker.
- Push, store and manage all the docker images in docker hub registry.
- Implement CI/CD with Jenkins for automated builds and deployments.
- Use Ansible for configuration management and deployment automation.
- Orchestrate all services with Kubernetes using Deployments, StatefulSets, Services, and Horizontal Pod Autoscaling (HPA).
- Implement testing of backend to ensure code reliability
- Integrate centralized logging, ensuring backend logs flow into Elasticsearch via Logstash.

Workflow

Code Development

- Used Visual Studio Code for frontend and backend development.
- Built the frontend using React.js with Vite, located in the `frontend` directory.
- Developed the backend using Node.js and Express with Mongoose for MongoDB integration, located in the `backend` directory.
- Followed a modular architecture with separate folders for routes, middleware, and models.

Version Control

- All source code is maintained in a Git repository hosted on GitHub.
- Utilized branching and pull requests for collaboration and code review.
- The repository includes application code as well as infrastructure-as-code (Kubernetes manifests, Ansible roles).

Continuous Integration and Continuous Delivery (CI/CD)

- Jenkins is used for CI/CD automation with a pipeline defined in the `Jenkinsfile`.
- Jenkins pulls the latest code from GitHub, runs backend unit tests, builds Docker images for frontend and backend, and pushes them to the container registry.
- Successful builds trigger automated deployment to the Kubernetes cluster.

Containerization

- Docker is used to containerize the frontend, backend, and MongoDB services.
- Each service is packaged as an independent Docker image to ensure consistent deployment and scalability.

Kubernetes Deployment

- All services are deployed on a Kubernetes cluster using manifest files.
- Deployments are used for the frontend and backend, while MongoDB runs as a StatefulSet.
- Services provide internal and external traffic routing.
- ConfigMaps and Secrets manage environment-specific configurations and sensitive data.

Autoscaling Configuration (HPA)

- HPA is configured for both frontend and backend using `frontend-hpa.yaml` and `backend-hpa.yaml`.
- Autoscaling is based on CPU utilization to maintain performance during fluctuating workloads.

Configuration Management

- Ansible is used to automate configuration and deployment of Kubernetes resources.
- Playbooks and roles handle application lifecycle management, infrastructure setup, and configuration consistency.

Centralized Logging using ELK Stack

- Logstash and Elasticsearch are deployed using Kubernetes manifests for centralized logging.
- The backend is configured to forward logs to Logstash via TCP.
- Integrated Kibana dashboards to visualize the real time system insights, including traffic volume, API endpoint usage, etc..

Architecture of Application and Code Development

Application Overview

The Doctor Appointment Management System is a full stack web application designed to streamline appointment scheduling between patients and doctors. The platform supports secure registration and authentication for two user roles, patients and doctors with each role receiving dedicated functionality to ensure an efficient user experience.

Patient Workflow

- **Registration and Login:** Patients can create an account and authenticate securely.
- **Browse Doctors:** A directory of doctors is available, including specialties and available time slots.
- **Book Appointments:** Patients can view a doctor's availability and schedule appointments.
- **View Appointments:** Patients can access a list of their upcoming and past appointments along with appointment details such as doctor name, date, and status.

Doctor Workflow

- **Registration and Login:** Doctors can register and authenticate securely.
- **Manage Availability:** Doctors can create, update, or remove available slots for patient bookings.
- **View Bookings:** Doctors can view all scheduled patient appointments, including patient information and appointment timings.
- **Appointment Management:** Doctors can update the status of appointments (e.g., confirm, complete, or cancel).

The system enforces strict role-based access control to ensure that each user can only access features relevant to their role.

Code Structure and Main Components

The codebase is organized using a modular architecture to support scalability, maintainability, and clear separation of concerns across backend and frontend layers.

Backend (Node.js / Express):

API Routes

- Core functionalities are exposed through RESTful APIs.
- Dedicated route files manage authentication , doctor operations , and appointment handling.
- Middleware components enforce authentication and role-based authorization.

Models

- Mongoose models define schemas for Patients, Doctors, and Appointments.
- Models capture field structures and relationships, such as appointments referencing both patient and doctor entities.

Authentication

- Secure authentication is implemented using JWTs.
- Middleware protects restricted endpoints and enforces role permissions.

Logic

- Controllers and route handlers implement logic for registration, login, doctor slot management, appointment booking, and appointment retrieval.
- Input validation and structured error handling ensure robust API behavior.

Logging

- Application logs are forwarded to Logstash for centralized monitoring and downstream analysis in Elasticsearch.

Frontend (React + Vite):

Pages and Components

- The frontend includes pages for login, registration, patient dashboards, and doctor dashboards.
- Reusable components are implemented for doctor listings, appointment forms, and appointment tables.

Role Based User Interface

- The UI dynamically adapts to the authenticated user's role, rendering only the relevant functionalities and navigation options.

API Integration

- All backend communication is handled through a dedicated API client , ensuring a clean abstraction layer and maintainable code.

Version Control

All source code is maintained in a Git repository hosted on GitHub.

Containerization

Frontend Containerization

- The frontend is containerized using a multi-stage Dockerfile.

Backend Containerization

- The backend runs in a Node.js-based container built from a dedicated Dockerfile.

```
backend > 🐦 Dockerfile
1 # Backend Dockerfile
2 FROM node:18-alpine
3
4 WORKDIR /app
5
6 COPY package.json package-lock.json* ./
7
8 RUN npm install
9
10 COPY . .
11
12 EXPOSE 5000
13
14 CMD ["npm", "start"]
```

(a) Backend Dockerfile

```
frontend > 🐦 Dockerfile
1 # Frontend Dockerfile
2 FROM node:18-alpine AS builder
3
4 WORKDIR /app
5
6 COPY package.json package-lock.json* ./
7
8 RUN npm install
9
10 COPY . .
11
12 RUN npm run build
```

(b) Frontend Dockerfile

Figure 1: Dockerfiles

MongoDB Containerization

- MongoDB uses the official `mongo` image from Docker Hub.
- No custom Dockerfile is required, and data persistence is supported through mounted volumes if needed.

Docker Hub

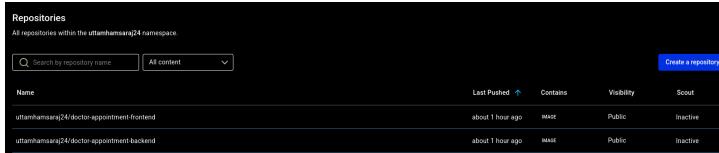


Figure 2: Docker Hub Repository

CI/CD Pipeline Overview

Pipeline Initialization

The pipeline begins by defining the execution environment and securely injecting required credentials. This includes authentication details for the container registry and configuration of image metadata used during the build and deployment process.

Stage 1: Source Code Retrieval

This stage fetches the latest version of the application code from the remote repository. The commit history is logged for traceability, ensuring the build is linked to a specific revision. This guarantees that each pipeline run works on a consistent and verifiable codebase.

```
stage('Step 1 : Git Clone') {
    steps {
        script {
            echo " Step 1 : Git Clone "
            checkout scm
            sh 'git log -1 --oneline'
        }
    }
}
```

Figure 3: Stage 1: Source Code Retrieval

Stage 2: Backend Dependency Installation

The backend service dependencies are installed to prepare the application for testing and packaging. Package installation ensures that all required libraries and modules are available for the upcoming testing and build phases. This step validates that the backend environment is correctly set up.

```

stage(' Step 2 : Install Requirements') {
    steps {
        script {
            echo "Setup Backend Stage"
            sh 'cd backend && npm install'
            echo "Backend dependencies installed successfully"
        }
    }
}

```

Figure 4: Stage 2: Backend Dependency Installation

Stage 3: Backend Testing

This test verifies the availability and responsiveness of the backend service by checking a simple health-check endpoint. This ensures continuous feedback on code quality.

```

stage(' Step 3 : Test Backend') {
    steps {
        script {
            echo "Test Backend Stage"
            catchError(buildResult: 'SUCCESS', stageResult: 'FAILURE') {
                sh 'cd backend && npm test'
            }
            echo "Backend tests completed"
        }
    }
}

```

Figure 5: Stage 3: Backend Testing

Stage 4: Backend Image Build and Registry Push

A container image for the backend service is built using the defined build instructions. The pipeline authenticates with the container registry and uploads the newly built image.

```

stage(' Step 4 : Build and Push Backend Docker Image') {
    steps {
        script {
            echo "Build and Push Backend Docker Image Stage"
            // Login to Docker Hub
            sh '''
... 
echo $DOCKER_HUB_CREDENTIALS_PSW | docker login -u $DOCKER_HUB_CREDENTIALS_USR
...
// Build Backend Image
sh '''
cd backend
docker build -t ${BACKEND_IMAGE} .
echo "Backend Docker image built successfully"
...
// Push Backend Image
sh '''
docker push ${BACKEND_IMAGE}
echo "Backend Docker image pushed to Docker Hub"
...
}
}

```

Figure 6: Stage 4: Backend Image Build and Registry Push

Stage 5: Frontend Image Build and Registry Push

The frontend application is containerized using a similar process. The image is built, tagged, and pushed to the registry for consumption by the deployment environment.

```

stage(' Step 5 : Build and Push Frontend Docker Image') {
    steps {
        script []
            echo "Build and Push Frontend Docker Image Stage"
            // Build Frontend Image
            sh '''
                cd frontend
                docker build -t ${FRONTEND_IMAGE} .
                echo "Frontend Docker image built successfully"
            '''
            // Push Frontend Image
            sh '''
                docker push ${FRONTEND_IMAGE}
                echo "Frontend Docker image pushed to Docker Hub"
            '''
    }
}

```

Figure 7: Stage 5: Frontend Image Build and Registry Push

Stage 6: Container Cleanup

The pipeline performs cleanup operations by removing unused containers and images on the agent. This prevents excessive disk usage and maintains a clean execution environment for subsequent runs. Regular pruning ensures healthy and predictable pipeline performance.

```

stage(' Step 6 : Clean Docker Images') {
    steps {
        script {
            echo "Clean Docker Images Stage"
            sh '''
                echo "Removing unused Docker containers..."
                docker container prune -f || true
                echo "Docker containers pruned"

                echo "Removing unused Docker images..."
                docker image prune -f || true
                echo "Docker images pruned"
            '''
        }
    }
}

```

Figure 8: Stage 6: Container Cleanup

Stage 7: Automated Deployment

The deployment phase uses an automation tool to apply the updated application configuration to the target infrastructure. This step orchestrates the rollout of the new application versions, typically to a container orchestration platform. The deployment is performed in a controlled and repeatable manner, ensuring consistency across environments.

```

stage(' Step 7 : Ansible Deployment') {
    steps {
        script {
            echo " Ansible Deployment Stage"
            echo "Triggering Ansible Playbook for Kubernetes Deployment"
            // Execute Ansible Playbook using Ansible Plugin
            ansiblePlaybook(
                inventory: 'Deployment/inventory',
                playbook: 'Deployment/deploy.yml',
                verbosity: 1,
                colorized: true,
                disableHostKeyChecking: true,
                extras: '-e ansible_python_interpreter=/usr/bin/python3'
            )
            echo "Ansible playbook executed successfully"
        }
    }
}

```

Figure 9: Stage 7: Automated Deployment

Complete Pipeline



Figure 10: Complete Jenkins Pipeline

Configuration Management using Ansible Inventory

- **inventory:** Defines all target hosts and host groups used by the Ansible playbooks.

```

Deployment > [ ] inventory
1   [local]
2     localhost ansible_connection=local
3
4
5

```

Figure 11: Ansible Inventory Configuration

Playbook

- **deploy.yml:** Primary orchestration playbook. This file is executed automatically via Jenkins. It drives the overall deployment workflow by invoking required roles, applying configuration changes, and coordinating deployment steps.

```

Deployment > deploy.yml
1  ---
2  - name: Deploy Doctor Appointment Application to Kubernetes
3  hosts: localhost
4  gather_facts: yes
5  vars:
6    | kubeconfig_path: "{{ ansible_env.HOME }}/.kube/config"
7
8  roles:
9    - kubernetes
10

```

Figure 12: Ansible Playbook

Roles Directory

- **roles/**: Contains modular and reusable Ansible roles, each encapsulating a specific unit of functionality. Roles improve maintainability, readability, and separation of concerns.

Active Roles

1. kubernetes

- Responsible for applying all the Kubernetes manifests for deployments.
- Handles deploying Deployments, StatefulSets, Services, ConfigMaps, and related K8s components.

2. docker

- Manages Docker Compose-based deployments.
- Not triggered by the Jenkins pipeline, since in jenkinsfile we have separate stages for containerization

Kubernetes

Kubernetes orchestrates all containerized components of the Doctor Appointment Management System. It provides automated deployment, scaling, load distribution, and life-cycle management.

Cluster and Access

- **Cluster Type:** Minikube is used.
- **Access:** The control plane is managed via kubectl from the CI/CD node (Jenkins/Ansible) using a configured kubeconfig.
- **Automation:** Kubernetes manifests are applied through the **kubernetes** Ansible role using the main playbook.

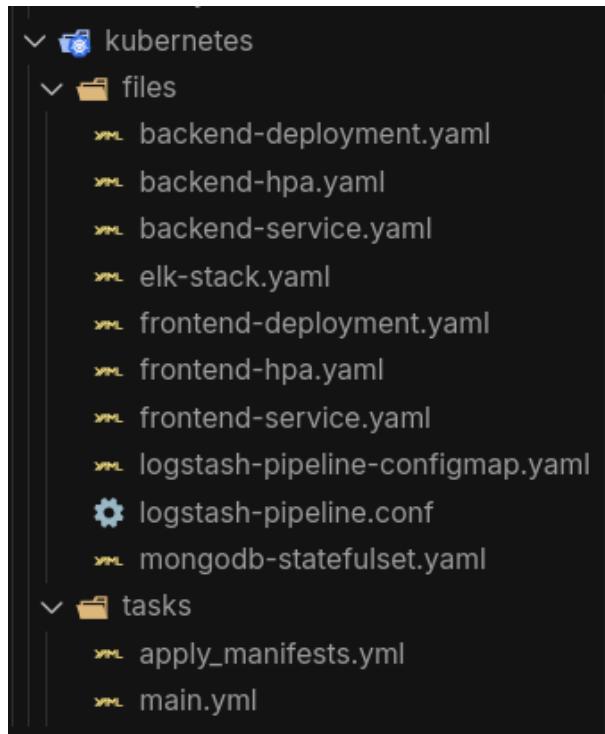


Figure 13: Kubernetes files

Container Images

- **Backend:** uttamhamsaraj24/doctor-appointment-backend:latest.
- **Frontend:** uttamhamsaraj24/doctor-appointment-frontend:latest.
- Images are pushed via the Jenkins CI pipeline and pulled by Kubernetes during pod creation.
- Tag management and registry authentication are handled outside of Kubernetes manifests.

Kubernetes Manifests

Deployments

- **backend-deployment.yaml:**
 - Declares backend Deployment, replicas, environment variables, container ports.
 - Defines resource limits and leverages rolling update strategy.
- **frontend-deployment.yaml:**
 - Defines frontend Deployment with React build image and replica count.
 - Includes essential runtime configuration for serving the UI.

Services

- **backend-service.yaml:**
 - Exposes backend pods via ClusterIP for internal access.
- **frontend-service.yaml:**
 - Provides external access to the frontend (NodePort).

Horizontal Pod Autoscalers (HPA)

- **backend-hpa.yaml** and **frontend-hpa.yaml:**
 - Configures autoscaling for both backend and frontend Deployments with a replica range of 2–5 pods.
 - Uses multiple resource metrics, including CPU utilization (50% target) and memory utilization (70% target), to drive scaling decisions.

StatefulSet and Persistent Storage

- **mongodb-statefulset.yaml:**
 - Deploys MongoDB using a StatefulSet for stable network identity.
 - Defines PersistentVolumeClaims (PVCs) for durable data storage.
 - Ensures data persistence across restarts and rescheduling.

```
uttam@uttams-Laptop:~/Documents/SPE-Project$ kubectl get pods -n doctor-appointment
NAME                               READY   STATUS    RESTARTS   AGE
backend-deployment-85889d6576-58nw2  1/1    Running   0          30m
backend-deployment-85889d6576-plk4c  1/1    Running   0          29m
elasticsearch-79465db66f-98wwd     1/1    Running   0          4h6m
frontend-deployment-6855fc0ffc-cqlxb 1/1    Running   0          30m
frontend-deployment-6855fc0ffc-lfkhc 1/1    Running   0          29m
kibana-544df88b9d-g68cg           1/1    Running   0          60m
logstash-665476dd55-7cz2x         1/1    Running   0          4h6m
mongodb-0                          1/1    Running   0          4h9m
uttam@uttams-Laptop:~/Documents/SPE-Project$ kubectl get hpa -n doctor-appointment
NAME      REFERENCE          TARGETS          MINPODS
backend-hpa Deployment/backend-deployment  cpu: 0%/50%, memory: 24%/70%  2
frontend-hpa Deployment/frontend-deployment  cpu: 1%/50%, memory: 12%/70%  2
```

Figure 14: Kubernetes Pods and HPA Status

Monitoring logs using ELK stack

- **Log Ingestion (Logstash):**
 - Logstash is deployed as the primary log aggregator.
 - The Node.js backend streams JSON-formatted logs directly to Logstash over TCP on port 5000, rather than using standard output (stdout).
 - This decoupling ensures that high-volume logging does not block the main event loop or interfere with application performance.
- **Elasticsearch:**

- Logstash processes incoming logs and indexes structured data into Elasticsearch, a distributed search and analytics engine.
- Stored metadata includes timestamps, HTTP methods (GET/POST), response durations, and status codes.
- Enables complex historical queries and analysis of application behavior.

• Visualization using Kibana:

- Kibana is deployed as the visualization frontend.
- Connects to Elasticsearch to provide interactive dashboards and insights.

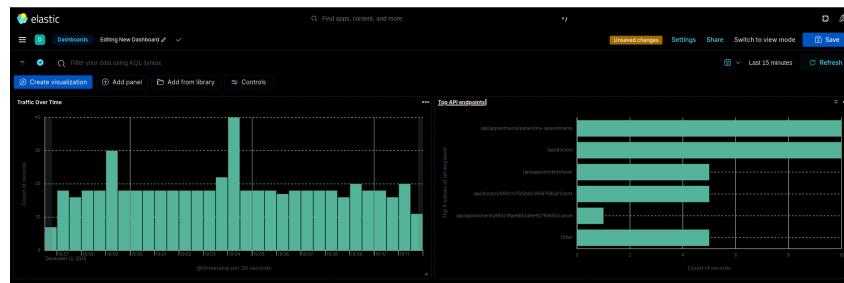


Figure 15: Kibana Dashboard Overview



Figure 16: Application Logs in Kibana