Project - High Level Design

on

Automotive Python Application Deployment on Kubernetes with Rolling Updates

Course Name:

Institution Name: Medicaps University – Datagami Skill Based Course

Student Name(s) & Enrolment Number(s):

Sr no	Student Name	Enrolment Number
1	KANCHI TIWARI	EN22CS304032
2	ARYAN PAWAR	EN22CS304017
3	AMAN RAI	EN22CS304009
4	AAKRATI JAIN	EN22CS304001
5	PRATHAM BHAWAR	EN22CS304047
6	MOHAMMAD FARAZ ABBASI	EN22CS303033

Group Name: Group 10D11

Project Number: DO-36

Industry Mentor Name:

University Mentor Name: Prof. Shyam Patel

Academic Year: 2026

Table of Contents

1. Introduction

The Automotive DevOps System is a containerized web application built using Python that enables users to manage and browse automotive-related data such as vehicles and related services.

The system is designed following modern DevOps principles including:

- Containerization using Docker
- Continuous Integration and Continuous Deployment (CI/CD) using Jenkins or GitHub Actions
- Orchestration using Kubernetes (Minikube)
- Zero-downtime Rolling Updates

The primary objective of this system is to demonstrate scalable, automated, and reliable deployment of a Python-based application on Kubernetes while ensuring uninterrupted service availability during version upgrades.

1.1 Scope of the Document

This document defines the high-level architecture, deployment workflow, infrastructure design, and operational processes of the Automotive Kubernetes Deployment System.

It outlines:

- Application architecture
- Container architecture
- CI/CD workflow
- Kubernetes deployment strategy
- Rolling update mechanism
- Data design
- Security and performance considerations

This document does not include low-level source code implementation details

1.2 Intended Audience

The document assumes readers have basic knowledge of:

- DevOps practices

- Containerization
- Kubernetes fundamentals
- Web application architecture

This document is intended for:

- DevOps Engineers
- Cloud Engineers
- Backend Developers
- Kubernetes Administrators
- Technical Reviewers
- Project Evaluators

1.3 System Overview

The system follows a modular and layered architecture separating:

- Application Layer
- Container Layer
- Orchestration Layer
- CI/CD Automation Layer

The system consists of:

- Python backend application (Flask / FastAPI)
- Jinja2-based frontend templates (if Flask)
- MySQL / PostgreSQL database
- Docker containerization
- Git for version control
- Jenkins or GitHub Actions for CI/CD
- Docker Registry (Docker Hub)
- Kubernetes cluster (Minikube)

The system ensures automated build and deployment along with zero-downtime rolling updates during application upgrades.

2. System Design

2.1 Application Design

The application follows a layered architecture:

Presentation Layer

- HTML/CSS
- Jinja2 Templates (Flask)

Application Layer

- Python Controllers / Routes
- REST APIs
- Business Logic
- Input Validation

Data Layer

- MySQL / PostgreSQL database
- SQLAlchemy ORM

Infrastructure Layer

- Docker container
- Kubernetes Deployment and Service

The application follows a request-response lifecycle:

1. Client sends HTTP request
2. Kubernetes Service routes traffic
3. Pod receives request
4. Python application processes request
5. Database interaction occurs
6. Response returned to client

2.2 Process Flow

User Request Flow

1. User accesses application via browser
2. Request reaches Kubernetes Service
3. Service forwards request to one of the running Pods
4. Laravel application processes the request
5. Response is sent back to user

CI/CD Deployment Flow

1. Developer pushes code to Git repository
2. CI/CD pipeline triggers automatically
3. Docker image is built
4. Image is pushed to Docker Registry
5. Kubernetes Deployment is updated
6. Rolling Update replaces old Pods with new Pods gradually

This ensures zero downtime during version upgrades.

2.3 Information Flow

User Browser

↓

Kubernetes Service

↓

Kubernetes Pod

↓

Docker Container

↓

Laravel Application

↓

Database

↓

Response to User

Deployment Information Flow:

$$\begin{tikzcd}
\text{Git Repository} \arrow[d] \\
\text{CI/CD Pipeline} \arrow[d] \\
\text{Docker Image} \arrow[d] \\
\text{Docker Registry} \arrow[d] \\
\text{Kubernetes Deployment}
\end{tikzcd}$$

2.4 Components Design

Git Repository

Stores Python application source code
Maintains version history

CI/CD Tool (Jenkins / GitHub Actions)

Automates build process
Builds Docker image
Pushes image to Docker Hub
Triggers Kubernetes deployment

Docker

Packages Python application
Ensures environment consistency

Docker Hub

Stores versioned container images

Kubernetes (Minikube)

Manages pods and replicas
Handles rolling updates
Provides service abstraction
Ensures high availability

Database (MySQL / PostgreSQL)

Stores automotive data
Connected via environment variables

2.5 Key Design Considerations

- Zero Downtime Rolling Updates
- Stateless application design
- Environment configuration using ConfigMaps and Secrets
- Resource limits for CPU and memory
- Liveness and Readiness Probes
- Image version tagging strategy
- Horizontal scalability support

2.6 API Catalogue

All APIs follow RESTful standards.

Endpoint	Method	Description
/	GET	Home page
/vehicles	GET	List all vehicles
/vehicles/{id}	GET	Get vehicle details
/vehicles	POST	Add new vehicle
/vehicles/{id}	PUT	Update vehicle
/vehicles/{id}	DELETE	Delete vehicle

Future enhancements may include:

- Authentication APIs
- Role-based access APIs
- API versioning

3. Data Design

3.1 Data Model

Main Entities:

Vehicle

- id
- name
- model
- price
- description
- created_at
- updated_at

User (optional)

- id
- name
- email
- password

3.2 Data Access Mechanism

Data is accessed using:

- SQLAlchemy ORM
- MySQL / PostgreSQL database
- Secure database connection via environment variables

All database interactions are handled at backend level.

3.3 Data Retention Policies

- Database data retained until manual deletion
- Docker images retained based on versioning policy
- Kubernetes logs retained as per cluster configuration
- Backup strategy can be implemented for production

3.4 Data Migration

Database migration can be handled using:

Flask + Alembic migration commands (example):

alembic upgrade head

Migrations can be triggered during CI/CD deployment to ensure schema consistency.

4. Interfaces

External Interfaces:

- Web Browser
- Docker Hub
- Git Repository
- CI/CD Tool

Internal Interfaces:

- Kubernetes Service to Pod
- Pod to Container
- Container to Database

5. State and Session Management

The application is designed as stateless for Kubernetes compatibility.

Session handling options:

- Client-side sessions
- Database-based sessions
- Redis-based session storage (recommended for scaling)

Kubernetes ensures pod restarts without affecting overall system availability.

6. Caching

Currently basic Laravel caching can be used.

Future enhancements:

- Redis-based distributed caching

- In-memory cache for improved performance
- Query result caching

7. Non-Functional Requirements

7.1 Security Aspects

- Kubernetes Secrets for sensitive data
- Environment variables for DB credentials
- Secure image storage in Docker Hub
- RBAC policies in Kubernetes
- Image vulnerability scanning
- CI/CD pipeline access control

7.2 Performance Aspects

- Rolling Updates ensure zero downtime
- Multiple replicas for load distribution
- Liveness and Readiness probes for health checks
- Optimized Docker image size
- Horizontal scaling capability

8. References

Kubernetes Documentation

Docker Documentation

Python Documentation

Flask / FastAPI Documentation

Jenkins Documentation

GitHub Actions Documentation