Automotive Python Application Deployment on Kubernetes with Rolling Updates

Course Name: Devops

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: Aashruti Shah

University Mentor Name: Prof. Shyam Patel

Academic Year: 2026

Problem Statement & Objectives

1. Problem Statement
2. Project Objectives
3. Scope of the Project

Proposed Solution

1. Key features *(Just mention key features here no need to go into details)*
2. Overall Architecture / Workflow
3. Tools & Technologies Used (*If applicable)*

Results & Output

Add the below details here:

1. Screenshots / outputs
2. Reports / dashboards / models
3. Key outcomes

Conclusion

Mention a brief conclusion about your project summing up everything you have worked on and the key learning.

Future Scope & Enhancements

Problem Statement & Objective

1. Problem Statement

Modern web applications require scalable, reliable, and automated deployment strategies to ensure uninterrupted service availability. Traditional deployment methods often involve manual configuration, leading to downtime during updates, inconsistent environments, and scalability limitations.

The problem addressed in this project is:

To deploy a containerized Automotive Python Web Application onto a Kubernetes (Minikube) cluster using DevOps practices, ensuring zero-downtime Rolling Updates, automated CI/CD pipeline integration, and scalable infrastructure management.

The solution must ensure:

- Automated container build and deployment
- Service availability during application upgrades
- Efficient resource utilization
- Simplified infrastructure management

2. Project Objectives

- To develop an Automotive web application using Python (Flask/FastAPI).
- To containerize the application using Docker for consistent deployment.
- To configure Kubernetes Deployment and Service manifests.
- To implement zero-downtime Rolling Updates.
- To integrate CI/CD pipeline using Jenkins or GitHub Actions.
- To ensure scalability using multiple pod replicas.
- To configure health monitoring using Liveness and Readiness Probes.
- To securely manage environment variables using ConfigMaps and Secrets.

3. Scope of the Project

The scope of this project focuses on the development and deployment of an Automotive web application using Python frameworks such as Flask or FastAPI, implemented following DevOps best practices. The project includes containerizing the application using Docker to ensure portability and consistency across environments. It also involves deploying the application on a Kubernetes cluster using Minikube, configuring Deployment and Service manifests for workload management. The implementation of zero-downtime rolling updates ensures seamless application upgrades without affecting end users. Additionally, the project integrates a CI/CD pipeline using Jenkins or GitHub Actions to automate the build and deployment process. Basic database integration using MySQL or PostgreSQL is included to

manage application data efficiently, and the entire setup is designed to run within a local Kubernetes cluster environment for development and testing purposes.

Furthermore, the project emphasizes the adoption of industry-standard DevOps methodologies to simulate a real-world deployment environment. It focuses on implementing best practices such as infrastructure abstraction, automated build pipelines, container orchestration, and configuration management. The system is designed to demonstrate how modern cloud-native applications are structured and deployed using container-based architecture. Although the deployment is performed in a local Minikube environment, the architectural design is scalable and can be extended to production-grade cloud platforms with minimal modifications. This ensures that the project not only fulfills academic objectives but also aligns with practical industry requirements and modern software deployment standards.

Proposed Solution

The proposed solution is a DevOps-driven deployment architecture designed to automate and streamline the lifecycle of an Automotive web application developed using Python frameworks such as Flask or FastAPI. The application is containerized using Docker to ensure portability, consistency, and environment independence. Kubernetes (Minikube) is used as the orchestration platform to manage containers efficiently, providing scalability, availability, and automated recovery mechanisms.

The solution also integrates a CI/CD pipeline using Jenkins or GitHub Actions to automate the build and deployment process from code commit to Kubernetes cluster deployment. A rolling update strategy is implemented to ensure zero downtime during application upgrades. Secure database integration is achieved using environment configurations managed through Kubernetes ConfigMaps and Secrets. Overall, the system ensures a fully automated deployment workflow without manual intervention, improving reliability and operational efficiency.

The proposed solution consists of:

- Python-based Automotive Web Application.
- Docker containerization for portability and consistency.
- Kubernetes orchestration for scalability and high availability.
- CI/CD pipeline for automated build and deployment.
- Rolling Update strategy for zero downtime.
- Database integration with secure configuration management.

1. Key Features

- Containerized Python Application
  The Automotive application is developed using Python (Flask/FastAPI) and packaged into a Docker container. This ensures consistency across development and deployment environments. Containerization eliminates dependency conflicts and enables easy portability across systems.
- Kubernetes Deployment & Service Configuration
  The application is deployed on a Kubernetes cluster using Deployment and Service manifests. Deployments manage pods and replicas, while Services expose the application internally within the cluster. This ensures efficient workload management and traffic routing.
- Zero-Downtime Rolling Updates
  Kubernetes rolling update strategy is implemented to upgrade application versions seamlessly. Old pods are gradually replaced with new ones without stopping the service. This guarantees uninterrupted availability for end users during updates.

- Multi-Replica Support
  Multiple pod replicas are configured to ensure scalability and high availability. Traffic is distributed among replicas to balance load efficiently. This setup allows the system to handle increased user requests without performance degradation.
- CI/CD Integration
  Continuous Integration and Continuous Deployment are implemented using Jenkins or GitHub Actions. Whenever code is pushed to the repository, the pipeline automatically builds the Docker image and updates the Kubernetes deployment. This automation reduces manual effort and deployment errors.
- Environment Configuration using Secrets
  Sensitive information such as database credentials is managed securely using Kubernetes Secrets. Configuration details are handled through ConfigMaps. This approach enhances security and separates configuration from application code.
- Health Monitoring (Liveness & Readiness Probes)
  Kubernetes Liveness and Readiness Probes are configured to monitor application health. Liveness probes detect and restart failed containers, while readiness probes ensure traffic is routed only to healthy pods. This improves system reliability.
- Stateless Application Architecture
  The application is designed to be stateless to support horizontal scaling. Session data is stored externally (database or cache), allowing pods to be replaced or scaled without affecting user sessions. This design aligns with Kubernetes best practices.
- Versioned Docker Images
  Docker images are tagged with version numbers for controlled deployments. Each update generates a new versioned image, enabling easy rollback if required. This ensures better version management and deployment traceability.

2. Overall Architecture / Workflow

The system architecture follows a layered DevOps model that separates responsibilities across different functional layers to ensure modularity, scalability, automation, and high availability. Each layer in the architecture performs a specific role in the application lifecycle, starting from code management to deployment and end-user interaction. The architecture integrates version control, CI/CD automation, containerization, orchestration, application logic, and database management into a unified workflow.

At the core of the system is a Python-based Automotive web application developed using Flask or FastAPI. The application is containerized using Docker and stored in a Docker registry. Kubernetes (Minikube) acts as the orchestration layer, managing pods, replicas, services, and rolling updates. The CI/CD pipeline automates the process from code commit to deployment, ensuring faster delivery and reduced manual errors. The workflow supports both developer-side automation and user-side request handling, providing a complete DevOps-driven deployment ecosystem.

Layered Architecture

1. Version Control Layer

- Git Repository
- Source Code Management

2. CI/CD Layer

- Jenkins / GitHub Actions
- Automated Docker Build
- Automated Image Push
- Deployment Trigger

3. Containerization Layer

- Dockerfile
- Python Application Image
- Docker Hub Registry

4. Orchestration Layer

- Kubernetes Deployment
- ReplicaSet
- Service
- Rolling Updates

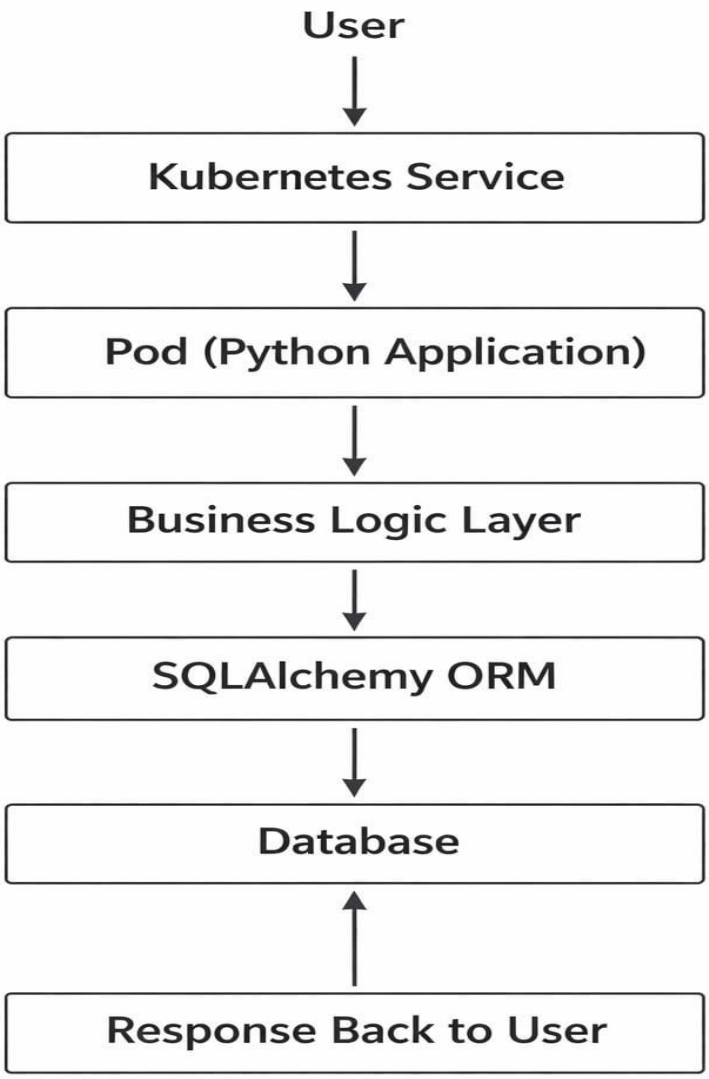
5. Application Layer

- Python (Flask/FastAPI)
- REST APIs
- Business Logic

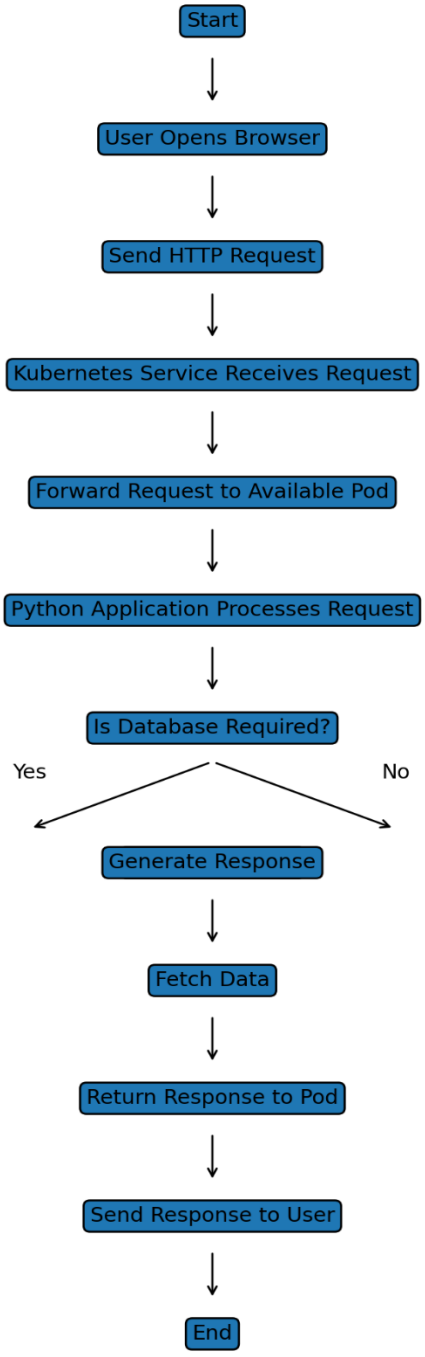
6. Database Layer

MySQL/PostgreSQLORM Integration (SQLAlchemy)

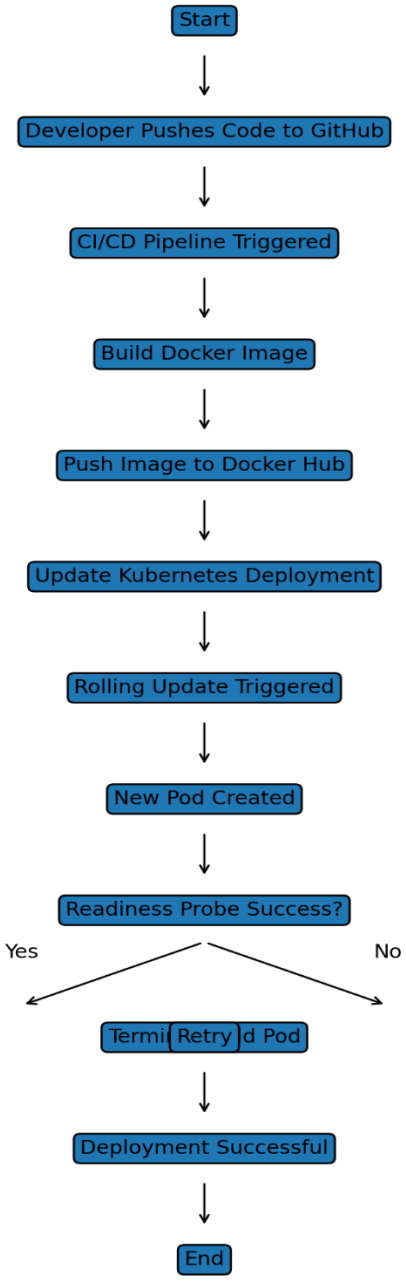
ARCHITECTURE DIAGRAM



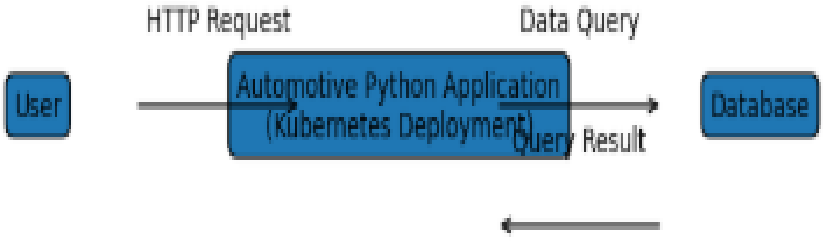
SYSTEM FLOWCHART



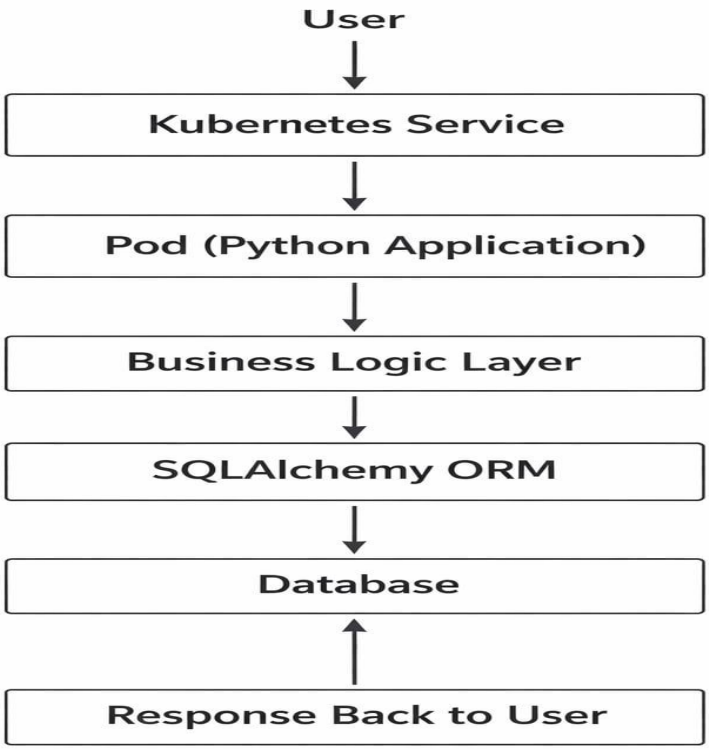
CI/CD DEPLOYMENT FLOWCHART



DFD LEVEL 0



DFD LEVEL 1



Workflow Description

Developer Workflow

- Developer pushes code to Git repository.
- CI/CD pipeline triggers automatically.
- Docker image is built.
- Image is pushed to Docker Hub.
- Kubernetes deployment is updated.
- Rolling update replaces old pods gradually.

User Workflow

- User accesses application via browser.
- Kubernetes Service routes request.
- Pod processes request.
- Response returned to user.

3. Tools & Technologies Used

Category	Tool / Technology
Programming Language	Python 3.x
Framework	Flask / FastAPI
Database	MySQL / PostgreSQL
ORM	SQLAlchemy
Containerization	Docker
Orchestration	Kubernetes (Minikube)
CI/CD	Jenkins / GitHub Actions
Version Control	Git
Registry	Docker Hub
CLI Tool	kubectl

Detailed Technical Implementation

1. Application Development

The automotive application includes:

- Vehicle listing API
- Add vehicle API
- Update vehicle API
- Delete vehicle API

The backend is structured using modular design principles.

2. Docker Implementation

A Dockerfile is created to:

- Use official Python base image
- Install dependencies
- Copy application files
- Expose application port
- Run the application

The Docker image is tagged using version numbers.

3. Kubernetes Deployment Configuration

Deployment YAML includes:

- Replica configuration
- Container image reference
- Resource limits
- Rolling update strategy
- Health checks

Example Rolling Update Configuration:

- maxUnavailable: 1
- maxSurge: 1

This ensures zero downtime during upgrades.

4. Service Configuration

Kubernetes Service is configured as:

- NodePort (for Minikube access)
- ClusterIP (internal communication)

5. Rolling Update Strategy

Rolling updates allow gradual replacement of old pods with new pods.

Steps:

1. New pod created.
2. Readiness probe verifies health.
3. Old pod terminated.
4. Traffic shifted automatically.

This ensures uninterrupted service.

6. CI/CD Pipeline

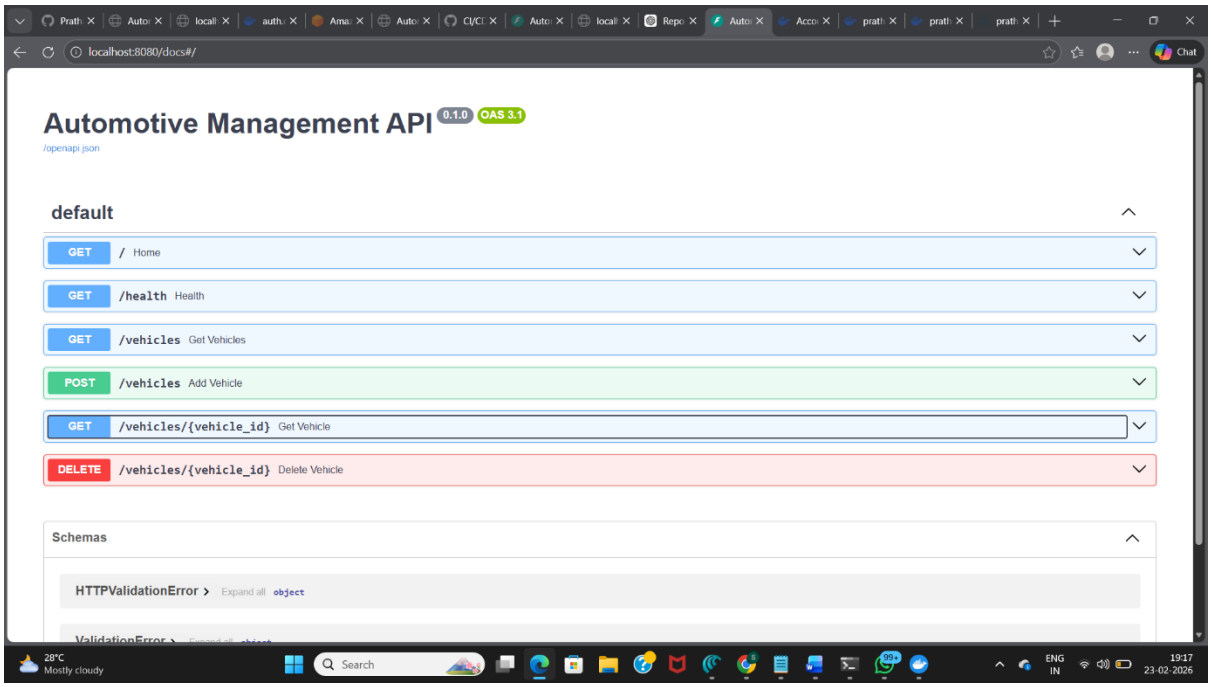
Pipeline stages:

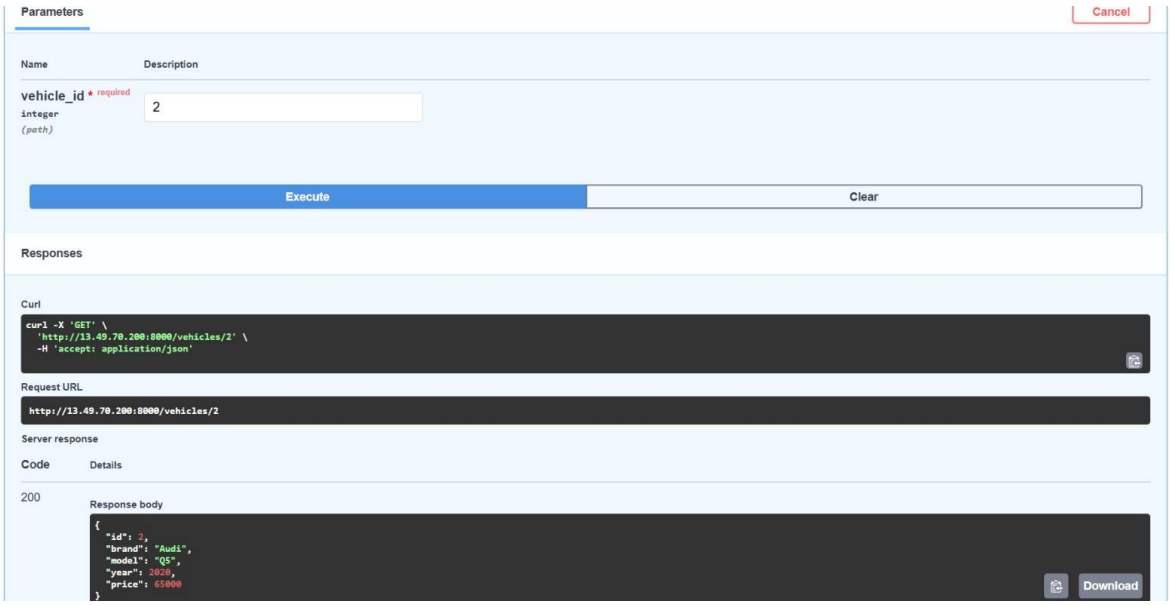
1. Code Checkout
2. Install Dependencies
3. Build Docker Image
4. Push Image to Docker Hub
5. Update Kubernetes Deployment

Automation reduces manual errors and speeds up deployment.

Results and Outcomes

1. Screenshots





Request URL

```
http://13.49.70.200:8000/vehicles
```

Server response

Code Details

200

Response body

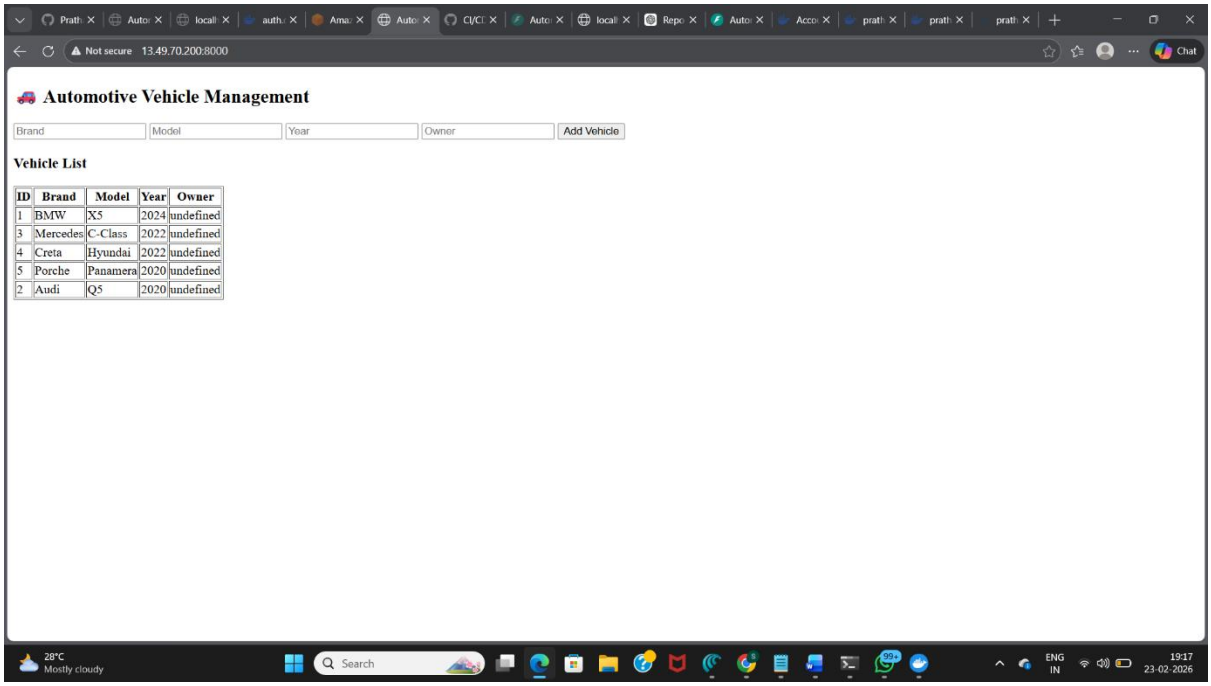

```
[
  {
    "id": 1,
    "brand": "BMW",
    "model": "X5",
    "year": 2024,
    "price": 75000
  },
  {
    "id": 3,
    "brand": "Mercedes",
    "model": "C-Class",
    "year": 2022,
    "price": 72000
  },
  {
    "id": 4,
    "brand": "Creta",
    "model": "Hyundai",
    "year": 2022,
    "price": 60000
  },
  {
    "id": 5,
    "brand": "Porche",
    "model": "Panamera",
    "year": 2020,
```


Download

Response headers

```
content-length: 348
content-type: application/json
date: Mon,23 Feb 2026 14:31:06 GMT
server: uvicorn
```

Responses



2. Result

The implementation of the Automotive Python web application using Docker and Kubernetes was successfully completed, demonstrating a fully functional DevOps deployment pipeline. The application was containerized and deployed on a Minikube Kubernetes cluster without configuration errors, and it was successfully accessed through a Kubernetes Service. The CI/CD pipeline using Jenkins or GitHub Actions automated the process from code commit to deployment, including Docker image build, image push to Docker Hub, and Kubernetes deployment update. This confirmed the successful integration of version control, automation tools, containerization, and orchestration into a seamless deployment workflow.

Furthermore, zero-downtime rolling updates were successfully tested and validated by deploying new application versions without interrupting user access. Kubernetes gradually replaced old pods with new ones while maintaining service availability, ensuring uninterrupted system performance. The use of multiple replicas enabled horizontal scalability and load distribution, while Liveness and Readiness Probes ensured automatic failure detection and recovery. Secure configuration management using ConfigMaps and Secrets protected sensitive data such as database credentials. Overall, the system achieved improved deployment efficiency, enhanced reliability, and practical implementation of real-world Kubernetes orchestration practices.

3. Key Outcomes

The key outcomes achieved include:

- Implementation of an Automated DevOps Pipeline
  A complete CI/CD workflow was successfully established using Jenkins or GitHub Actions. The pipeline automatically builds Docker images, pushes them to a registry, and updates the Kubernetes deployment upon every code commit, ensuring faster and more reliable releases.
- Achievement of Zero-Downtime Deployment
  Kubernetes Rolling Updates were configured to gradually replace old pods with new ones without stopping the application service. This ensured uninterrupted user access during version upgrades and demonstrated high availability practices.
- Scalability and High Availability
  Multiple pod replicas were deployed to distribute traffic efficiently across instances. This improved system performance and ensured service continuity even if one pod failed.
- Improved Deployment Efficiency
  Automation reduced manual intervention in the deployment process, minimized configuration errors, and significantly shortened deployment time. The system now supports consistent and repeatable deployments.

- Enhanced Understanding of Kubernetes Orchestration
  The project provided hands-on experience in managing Deployments, ReplicaSets, Services, rolling updates, ConfigMaps, Secrets, and health probes, strengthening practical knowledge of container orchestration.

Conclusion

The project successfully demonstrates the deployment of a containerized Automotive Python Application onto a Kubernetes (Minikube) cluster by effectively applying modern DevOps principles. The application was developed, containerized using Docker, and orchestrated through Kubernetes to ensure efficient workload management and service availability. The integration of version control, automated build processes, and orchestration tools reflects a complete end-to-end deployment lifecycle aligned with industry standards.

The implementation of Rolling Updates plays a crucial role in ensuring zero downtime during application upgrades. By gradually replacing old pods with new ones while keeping the service active, Kubernetes maintains continuous availability and reliability of the application. This approach enhances system stability and ensures that end users experience uninterrupted access even during deployment of new versions.

Furthermore, the integration of CI/CD automation significantly improves deployment efficiency and minimizes manual intervention. Automated pipelines build Docker images, push them to the container registry, and update Kubernetes deployments seamlessly after every code commit. This reduces human errors, ensures consistency across deployments, and accelerates the release cycle.

Overall, this project provides valuable practical exposure to containerization, Kubernetes orchestration, rolling update strategies, and automated deployment workflows. The hands-on implementation of these technologies closely aligns with modern industry practices and strengthens understanding of cloud-native application deployment models used in real-world production environments.

Future Scope & Enhancements

Although the project successfully demonstrates containerized deployment on Kubernetes (Minikube), it can be further enhanced to make it production-ready and more scalable. In future, the application can be deployed on cloud platforms such as AWS EKS, GKE, or AKS to achieve high availability and enterprise-level infrastructure support.

Monitoring tools like Prometheus and Grafana can be integrated for real-time performance tracking and alerting. The implementation of Horizontal Pod Autoscaler (HPA) can enable automatic scaling based on workload demand. Additionally, an Ingress Controller can be configured for secure external access and domain-based routing.

The architecture can also be extended into a microservices-based model for better scalability and maintainability. Advanced logging using the ELK stack and CI/CD pipelines with automated testing and security scanning can further improve system reliability, performance, and code quality.