# Project - High Level Design

On

# Deploy Insurance App To Kubernetes

## Course Name: Devops Foundation

**Institution Name:** Medicaps University – Datagami Skill Based Course

| Sr no | Student Name | Enrolment Number |
|---|---|---|
| 1 | S. JEZREEL | EN22IT301085 |
| 2 | KRITAGYA TIWARI | EN22IT301048 |
| 3 | SADHANA YADAV | EN24CA5030146 |
| 4 | CHANDRESH BHATI | EN24CA50040 |
| 5 | SHIVKUMAR SOLANKI | EN24CA5030157 |

Project Number:DO-40

Industry Mentor Name: Mr. Vaibhav

University Mentor Name: Prof. Akshay Saxena

Academic Year:2025-26

Group Name: 02D12

# Table of Contents

# CHAPTER 1: INTRODUCTION

## 1.1 Introduction:

The deployment architecture follows a cloud-native approach using AWS EC2 as the primary infrastructure. The application code is first uploaded to GitHub using GitHub Desktop from the local system. The EC2 instance is launched with Ubuntu OS and accessed securely via SSH. Docker is installed on the EC2 instance to build and run the application container. Kubernetes is configured on the same instance to manage the container lifecycle, handle pod creation, and ensure application availability. This setup enables smooth deployment, easy updates, and better resource management. The architecture ensures scalability, portability, and fault tolerance while maintaining a simple and cost-effective deployment strategy suitable for real-world applications.

## 1.2 Scope of the document:

This document provides a high-level technical design of the Insurance application deployment architecture. It explains the system components, application structure, deployment workflow, data flow, security considerations, and non-functional requirements.

The document does not focus on detailed coding but instead highlights architectural decisions, tools used, and overall system behaviour.

## 1.3 Intended Audience:

The intended audience for this project includes organizations and users interested in deploying scalable and highly available web-based insurance applications using cloud-native technologies.

This project is suitable for development teams and DevOps engineers who want to understand how containerization and orchestration can be applied to real-world applications. It is also designed for Customers and admins for managing Insurance policies.

## 1.4 System overview:

The system is designed to deploy a cloud-native Insurance web application using modern DevOps practices. The application is developed using Python with the Flask framework and provides a web-based interface for displaying insurance plans and policy details. The frontend is implemented using HTML, CSS, and JavaScript to ensure a simple and user-friendly experience.

The application is containerized using Docker, which packages the application and its dependencies into a single container image for consistent execution across environments. These containers are managed using Kubernetes, which handles deployment, scaling, load balancing, and health monitoring.

The Kubernetes cluster is configured using Minikube and hosted on an Amazon Web Services EC2 instance. Deployment and Service manifests are used to manage containerized workloads and expose the application through a NodePort service.

# CHAPTER 2: SYSTEM DESIGN

## 2.1. Application Design:

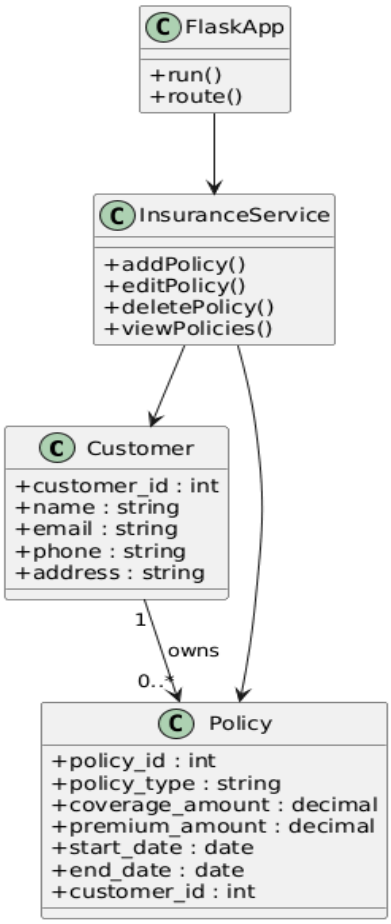


Fig:2.1(Application diagram)

The application design of the Insurance Management System using a class-based architecture. The FlaskApp class acts as the main application layer, handling routing and execution of the web application. It interacts with the InsuranceService class, which contains the core business logic responsible for managing insurance operations such as adding, editing, deleting, and viewing policies. The Customer and Policy classes represent the data entities of the system, storing customer details and insurance policy information respectively. A one-to-many relationship exists between Customer and Policy, indicating that one customer can own multiple policies. Overall, the diagram illustrates the structured interaction between the application layer, service layer, and data layer, ensuring organized and maintainable system design.

## 2.2. Process Flow:

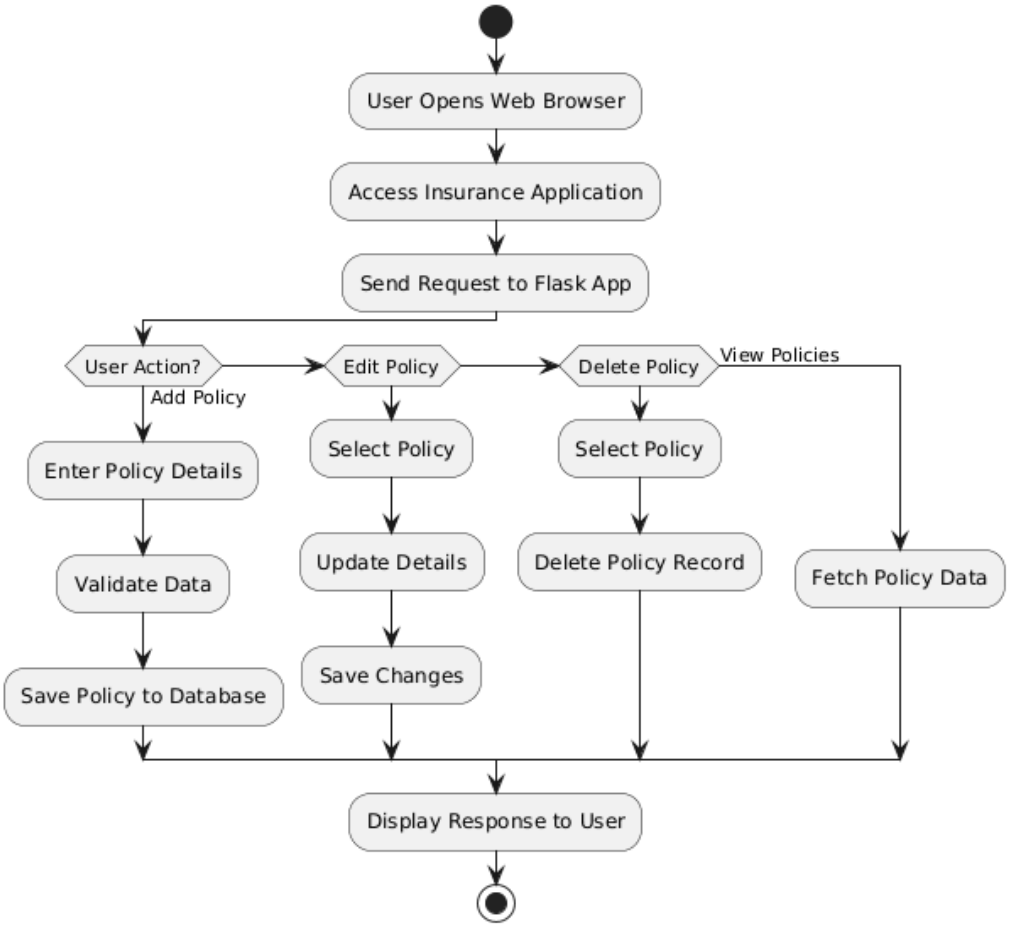


Fig:2.2 (Process diagram)

The process flow of the Insurance Management System, illustrating the sequence of operations performed from user interaction to system response. The process begins when the user opens a web browser and accesses the insurance application, which sends a request to the Flask application server. The system then evaluates the user's action, where the user can choose to add, edit, delete, or view insurance policies. For adding a policy, the user enters policy details, the system validates the data, and the information is stored in the database. In the edit operation, the user selects an existing policy, updates the details, and saves the changes, while the delete operation involves selecting a policy and removing its record from the database. If the user chooses to view policies, the system retrieves policy data for display. After completing any selected operation, the application processes the request and displays the appropriate response back to the user, completing the workflow.

## 2.3. Information Flow:

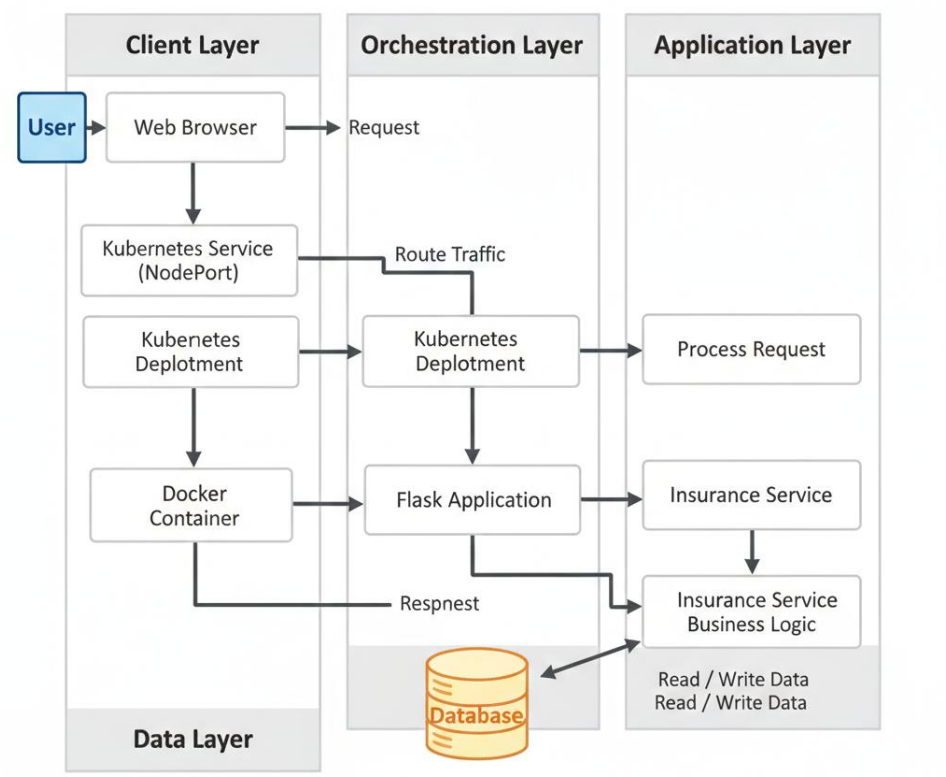


Fig:2.3 (Information flow)

The information and processing flow of the Insurance Web Application across multiple architectural layers, demonstrating how user requests are handled within a containerized Kubernetes environment. The process begins in the **Client Layer**, where the user interacts with the system through a web browser, generating an HTTP request. This request is forwarded to the **Orchestration Layer**, where the Kubernetes Service (NodePort) receives and routes traffic to the Kubernetes Deployment. The deployment manages application instances running inside Docker containers and forwards the request to the Flask application. In the **Application Layer**, the Flask application processes the request and communicates with the Insurance Service, which executes the business logic required for policy management operations. The Insurance Service then interacts with the **Data Layer**, where the database performs read and write operations to store or retrieve information. After processing, the response is sent back through the same layers—from the application to the container, through Kubernetes, and finally to the user's browser—ensuring a structured, scalable, and efficient flow of data within the cloud-native system architecture.

## 2.4. Components Design:

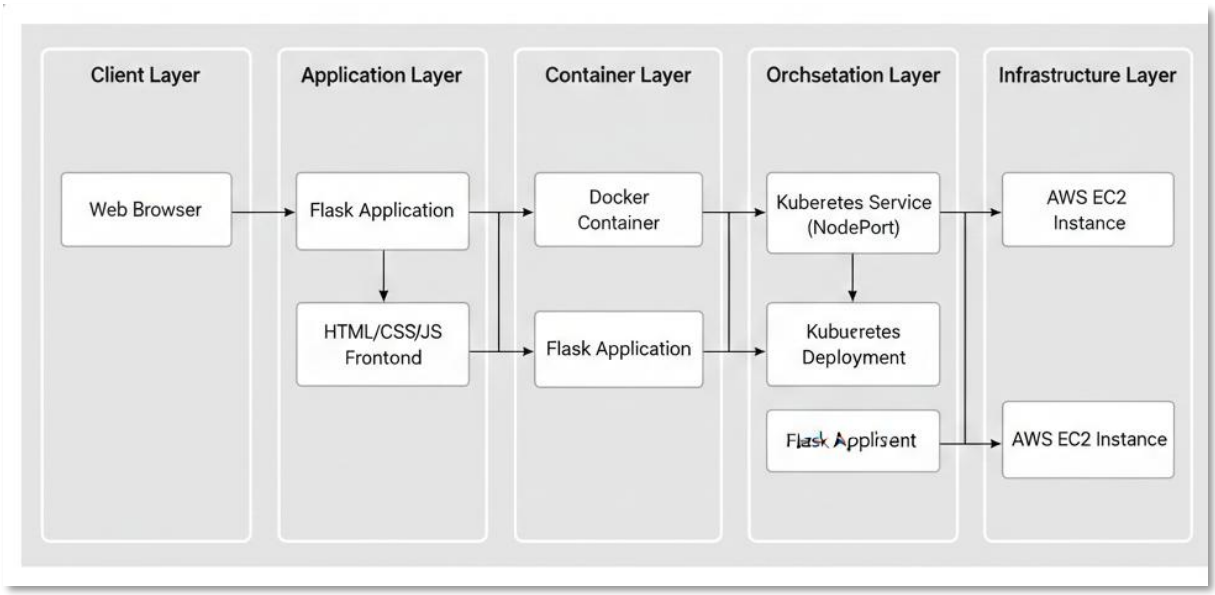


Fig:2.4(Components Desgin)

The high-level architecture of the Insurance Web Application, showing how different system layers interact to deliver a containerized and cloud-deployed solution. The process begins in the **Client Layer**, where users access the application through a web browser. The request is then handled in the **Application Layer**, where the Flask application processes user interactions and communicates with the HTML, CSS, and JavaScript frontend to generate dynamic responses. The application is packaged within a Docker container in the **Container Layer**, ensuring portability and consistent execution across environments. In the **Orchestration Layer**, Kubernetes manages the containerized application through a NodePort Service that exposes the application externally and a Deployment that maintains and controls application instances. Finally, the entire system runs on an **AWS EC2 Instance** within the **Infrastructure Layer**, providing scalable cloud hosting. Overall, the diagram demonstrates a layered cloud-native architecture that integrates web technologies, containerization, orchestration, and cloud infrastructure to ensure scalability, reliability, and efficient application deployment.

## 2.5. Key Design Considerations:

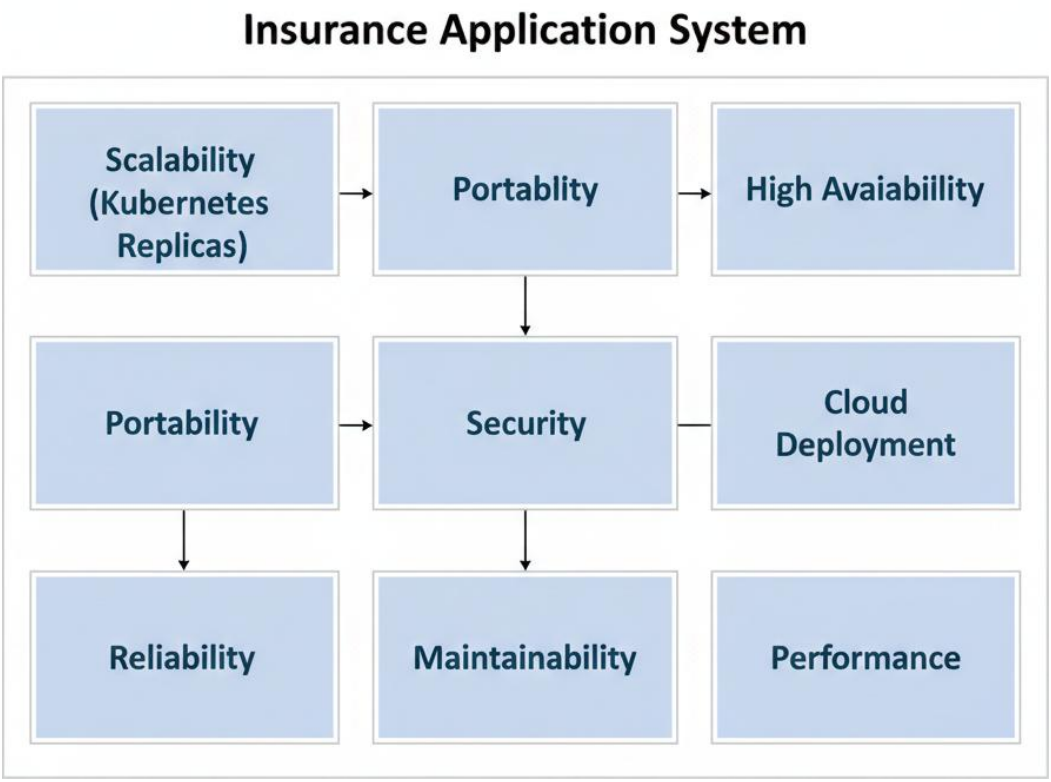


Fig:2.5(key design consideration)

The key design considerations of the Insurance Application System, highlighting the fundamental architectural principles followed during system development and deployment. The design focuses on **scalability**, achieved through Kubernetes replicas that allow the application to handle increasing user demand efficiently. **Portability** is ensured by containerization, enabling the application to run consistently across different environments, which further contributes to **high availability** by allowing seamless deployment and replication of services. The diagram also emphasizes **security**, which protects application data and controls access within the cloud environment, and **cloud deployment**, where the system is hosted on scalable infrastructure such as AWS EC2. Additionally, the architecture prioritizes **reliability** through Kubernetes self-healing mechanisms, **maintainability** via modular application design, and improved **performance** through efficient resource utilization and load distribution. Overall, the diagram represents a balanced and robust system design that ensures efficiency, stability, and scalability in a cloud-native application environment.

## 2.6. API Catalogue:

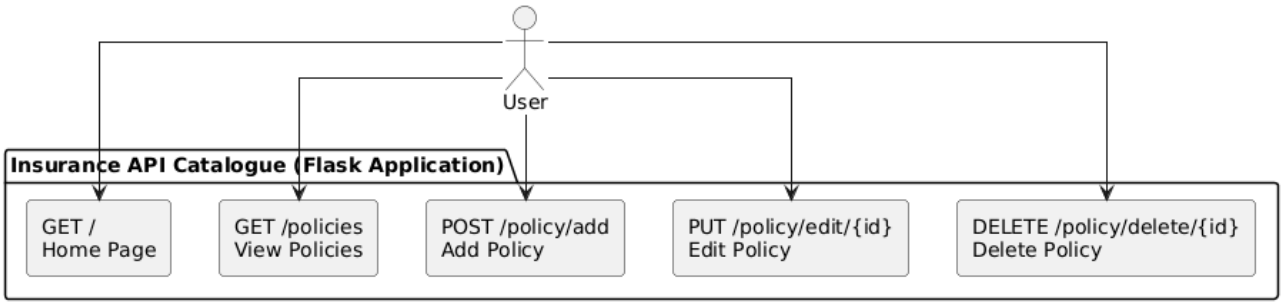


Fig:2.6(API Catalogue)

The **API Catalogue of the Insurance Application**, illustrating the different API endpoints provided by the Flask-based system and how the user interacts with them. The user acts as the primary actor who sends requests to the application through various HTTP methods. The catalogue includes multiple endpoints such as **GET /** for accessing the home page, **GET /policies** for viewing all insurance policies, **POST /policy/add** for adding a new policy, **PUT /policy/edit/{id}** for updating an existing policy, and **DELETE /policy/delete/{id}** for removing a policy.

Each endpoint corresponds to a specific operation within the Insurance Management System and represents the communication interface between the user and the backend application. Overall, the diagram provides a structured overview of the system's API design, demonstrating how different user actions are mapped to specific HTTP requests for managing insurance policies efficiently.

# CHAPTER 3: DATA DESIGN

## 3.1. Data Model:

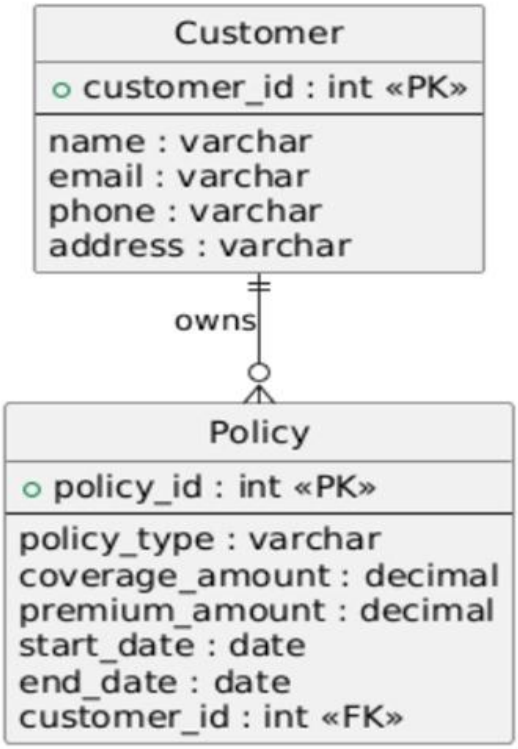


Fig:3.1(Data Model)

The system follows a relational database design consisting of two main entities: Customer and Policy. The Customer table stores personal details such as name, email, phone number, and address. The Policy table stores insurance-related details including policy type, premium amount, start date, end date, and status. A one-to-many relationship exists where one customer can have multiple policies.

The Policy table includes a foreign key that connects each policy to its respective customer. This design ensures data consistency, reduces redundancy, and maintains proper relationships between records.

## 3.2. Data Access Mechanism:

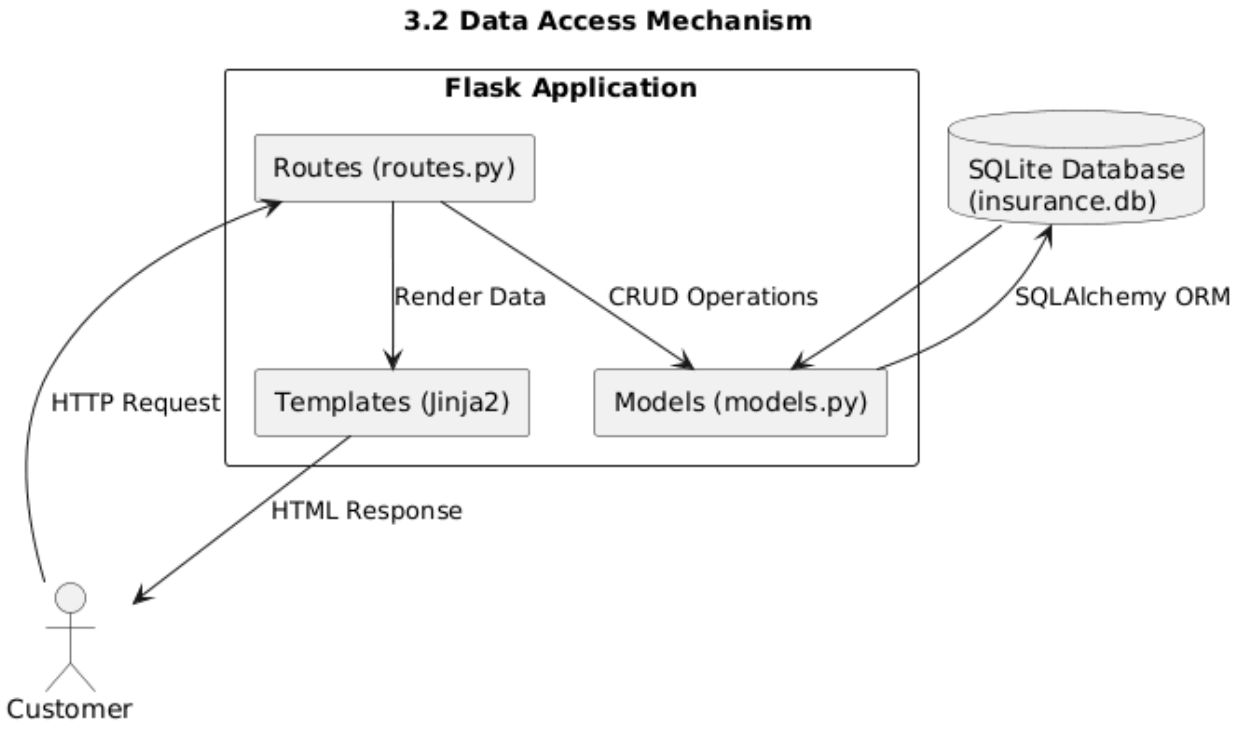


Fig:3.2(Data Access Mechanism)

The application uses Flask as the backend framework and SQLAlchemy as the Object Relational Mapper (ORM) to manage database interactions. User requests are processed through defined routes, which communicate with the database models. SQLAlchemy converts Python objects into SQL queries, allowing smooth execution of create, read, update, and delete operations. This layered approach separates application logic from database logic, improving maintainability, security, and scalability of the system.

## 3.3. Data Retention Policies:

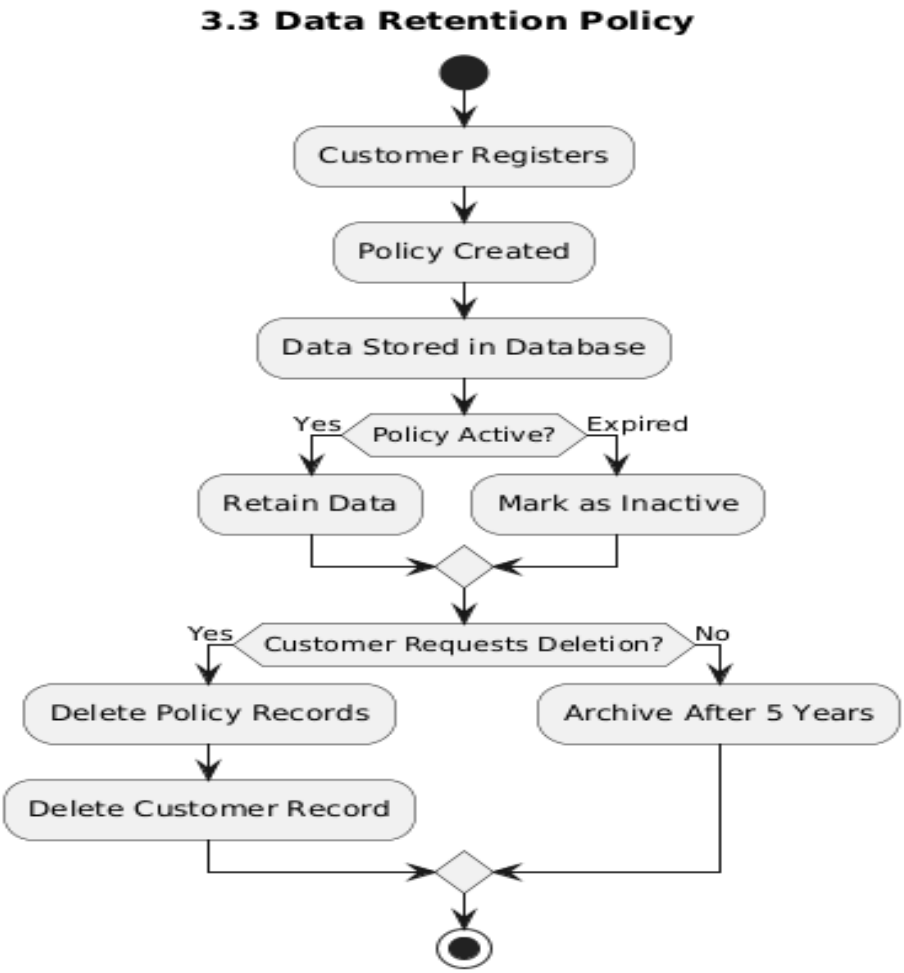


Fig:2.3(Data Retention policies)

Customer and policy information is stored in the database throughout the policy lifecycle. Active policies remain fully accessible, while expired policies are retained for record-keeping and reporting purposes. If a user deletes a policy, the record is permanently removed from the database. Regular database backups are maintained to prevent accidental data loss and to ensure data reliability over time.

## 3.4. Data Migration:

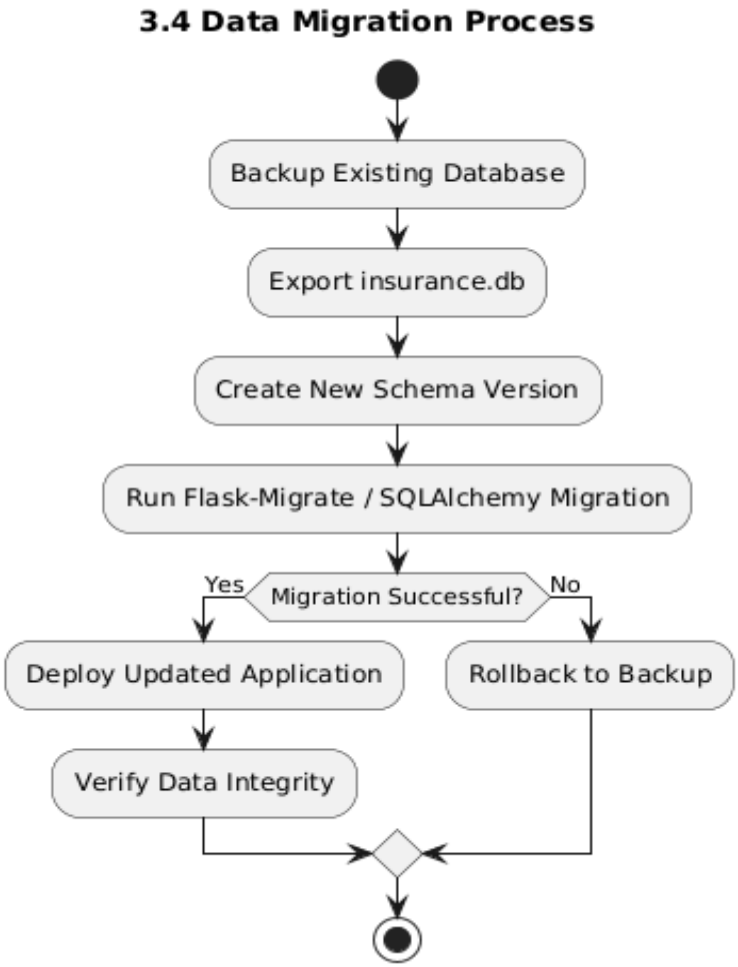


Fig:2.4(Data Migration)

Data migration is performed when updates or structural changes are required in the database. Before applying any modifications, a full database backup is created to avoid data loss. Migration tools such as Flask-Migrate can be used to apply schema changes safely. After migration, data validation is performed to ensure that all existing records remain intact and consistent. This process allows the system to evolve without compromising stored customer and policy information.

# CHAPTER 4: INTERFACES

## 4.1 Interfaces:

The system exposes multiple interfaces that enable interaction between users, application components, and infrastructure services. These interfaces ensure smooth communication, automation, and accessibility throughout the deployment lifecycle.

## 4.2 User Interface:

The User Interface is a web-based interface developed using HTML, CSS, and JavaScript. It allows end users to access the Insurance application through a standard web browser. The interface enables users to view insurance plans, policy details, and other related information in a simple and user-friendly manner. The frontend focuses on clarity, responsiveness, and ease of navigation to provide a smooth user experience.

## 4.3 Application Interface (Frontend–Backend Interface):

The application interface manages communication between the frontend and backend components of the system. User requests generated from the web interface are sent to the Flask backend using HTTP protocols. The backend processes these requests, applies the required business logic, and returns appropriate responses in the form of rendered HTML pages. This interface ensures proper separation between presentation logic and application processing.

## 4.4 Container Interface:

The container interface defines how the application is packaged and executed within a containerized environment. The Insurance application runs inside Docker containers, which provide an isolated runtime environment containing the application code and its dependencies. This interface ensures consistent behavior of the application across development, testing, and production environments.

## 4.5 Kubernetes Interface:

The Kubernetes interface manages container orchestration and service exposure. Kubernetes uses Deployment manifests to define pod replicas, container image versions, and rolling update strategies. Service manifests control internal networking and external access to

the application through a NodePort service. This interface ensures high availability, load balancing, and automated management of containerized workloads.

## 4.6 Source Control Interface:

The source control interface is managed using GitHub, which stores the complete application source code and configuration files. It enables version control, change tracking, and collaboration. GitHub also acts as the trigger point for the CI/CD pipeline, allowing automated deployment workflows.

## 4.7 Cloud Infrastructure Interface:

The cloud infrastructure interface provides the underlying computing and networking resources required to host the application. The system is deployed on an AWS EC2 instance, which offers scalable virtual server infrastructure. Network access is managed using security groups to ensure secure and controlled exposure of application services.

## 4.8 Administrative Interface:

The administrative interface is used for system monitoring, configuration, and troubleshooting. Administrators access the system through secure SSH connections to the EC2 instance and use Kubernetes command-line tools to monitor pod status, view logs, and manage deployments. This interface ensures efficient operational control of the application environment.

# CHAPTER 5: State and Session Management

## 5.1 Application State Management:

The Insurance application is designed to be stateless in nature. The Flask backend does not store any client-specific data or application state on the server side. Each user request is treated as an independent transaction, allowing the application to scale easily across multiple container instances. This stateless design aligns with Kubernetes deployment best practices and enables smooth horizontal scaling and rolling updates without affecting active users.

## 5.2 Session Handling:

The application does not implement server-side session management such as login sessions or user authentication. No session data is stored in memory or persistent storage. All interactions are handled through simple HTTP requests and responses. This approach eliminates the need for session synchronization across pods and avoids complexity related to session persistence.

## 5.3 Impact on Containerization:

Since the application is stateless, Docker containers can be created, destroyed, or replaced at any time without affecting application behavior. This allows Kubernetes to freely manage pods during scaling operations or rolling updates. Containers do not depend on any locally stored data, making deployments lightweight and reliable.

## 5.4 Impact on Kubernetes Deployment:

Stateless application design plays a critical role in Kubernetes orchestration. During rolling updates, Kubernetes gradually replaces old pods with new ones while continuing to route traffic to healthy pods. Because no session data is tied to any specific pod, users experience uninterrupted service during application updates. This directly supports the zero-downtime rolling update strategy implemented in the project.

## 5.5 Future Enhancements for State Management

If future versions of the Insurance application require user authentication or personalized data, external state management solutions such as databases, distributed caches, or session stores can be integrated. These services would allow persistent session handling while maintaining stateless application pods. This enhancement can be implemented without major architectural changes to the existing deployment.

# CHAPTER 6: CACHING

## 6.1 Overview of Caching Strategy:

In the current implementation of the Insurance application, no explicit caching mechanism is used at the application or infrastructure level. The system is designed to serve requests dynamically through the Flask backend running inside containerized pods.

This approach keeps the application simple and avoids additional complexity related to cache management during development and deployment.

## 6.2 Reason for Not Using Caching:

The application mainly serves static and lightweight dynamic content such as insurance plans and policy information. Since the data volume is small and response times are fast, caching is not mandatory at this stage.

Avoiding caching also ensures that users always receive the most up-to-date information without the risk of stale data being served.

## 6.3 Impact on Kubernetes Deployment:

Even without caching, Kubernetes ensures efficient request handling through load balancing across multiple pods.Requests are distributed evenly among healthy containers, which helps maintain performance and availability.

The stateless nature of the application ensures that each pod can independently handle requests without relying on cached data stored locally.

# CHAPTER 7: Non-Functional Requirements

The Insurance application is designed to meet essential non-functional requirements that ensure system reliability, scalability, and maintainability. High availability is achieved through Kubernetes pod replication, allowing multiple instances of the application to run simultaneously. Scalability is supported by Kubernetes, which enables horizontal scaling by increasing or decreasing the number of pods based on demand. The system ensures reliability by automatically restarting failed containers and routing traffic only to healthy pods. Portability is achieved through Docker containerization, allowing the application to run consistently across different environments. Maintainability is supported by modular application design, version-controlled source code, and automated deployments using Jenkins. The system also emphasizes fault tolerance, ensuring that failures in individual containers do not impact overall service availability.

## 7.1. Security Aspects:

Security is addressed at multiple levels in the system architecture. The application runs inside Docker containers,providing isolation from the host system and other applications. Access to the cloud server is secured using SSH with key-based authentication, reducing the risk of unauthorized access.

AWS EC2 security groups are configured to allow only required ports for application access and administrative operations. Kubernetes manages secure internal communication between pods and services, ensuring controlled network access.

Sensitive information such as credentials and configuration values is not hard-coded into the application. The stateless nature of the application further enhances security by avoiding server-side storage of user session data. Overall, the system follows basic security best practices suitable for a prototype-level deployment.

## 7.2. Performance Aspects:

The performance of the Insurance application is optimized through lightweight application design and efficient container orchestration. The Flask backend is minimal and processes requests quickly with low resource consumption. Docker containers ensure fast startup times, enabling rapid scaling when required. Kubernetes improves performance by distributing incoming requests across multiple pods using built-in load balancing.

Rolling updates allow application upgrades without impacting active users, maintaining consistent response times. Since the application is stateless, any pod can handle any request, reducing bottlenecks and improving throughput. The absence of heavy backend processing or database interactions further contributes to low latency and stable performance under normal load conditions.

# CHAPTER 8: REFERENCES


Kubernetes Documentation – Official documentation for Kubernetes architecture, deployments, services, and rolling update strategies.
Available at: https://kubernetes.io/docs/

Docker Documentation – Official Docker documentation for containerization concepts, Dockerfiles, and image management.
Available at: https://docs.docker.com/

Flask Documentation – Official Flask framework documentation for Python-based web application development.
Available at: https://flask.palletsprojects.com/

AWS EC2 Documentation – Amazon Web Services documentation for Elastic Compute Cloud (EC2) setup and security.

Available at: https://docs.aws.amazon.com/ec2/

OpenAI ChatGPT – Used as an AI-based assistant for guidance in system design, DevOps workflow understanding, Kubernetes deployment strategies, and documentation preparation.

Available at: https://chat.openai.com/