A MINI PROJECT REPORT On

"INSURANCE ADVISOR"

Submitted to

OSMANIA UNIVERSITY

In partial fulfillment of the requirements for the award

of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING (AI & ML)

BY

BHEEMANPALLY ABHIJITH	245522748073
PANKAJ DESHMUKH	245522748075
PPC KALYAN	245522748115

Under the esteemed guidance of

Mrs . Bhavani Vanama

Assistant Professor

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (AI & ML)

KESHAV MEMORIAL ENGINEERING COLLEGE

(Approved by AICTE, New Delhi & Affiliated to Osmania University, Hyderabad) D.No. 10 TC-111, Kachavanisingaram (V), Ghatkesar (M), Medchal-Malkajgiri, Telangana – 500088

(2024-2025)

DECLARATION

This is to certify that the mini project titled "**INSURANCE ADVISOR**" is a bonafide work done by us in fulfillment of the requirements for the award of the degree Bachelor of Engineering in Department of Computer Science and Engineering (AI & ML), and submitted to the Department of CSE (AI & ML), Keshav Memorial Engineering College, Hyderabad.

We also declare that this project is a result of our own effort and has not been copied or intimated from any source. Citations from any websites are mentioned in the bibliography. This work was not submitted earlier at any other university for the award of any degree.

BHEEMANPALLY ABHIJITH (245522748073)
PANKAJ DESHMUKH (245522748075)
PPC KALYAN (245522748115)

Place:
Date:

Signature of the Candidate

KESHAV MEMORIAL ENGINEERING COLLEGE

Department of Computer Science and Engineering (AI & ML)

CERTIFICATE

This is to certify that the project report entitled **"INSURANCE ADVISOR"** that is being submitted by **DESHMUKH PANKAJ (245522748075), BHEEMANPALLY ABHIJITH (245522748073), PPC KALYAN (245522748115)** under the guidance of **Mrs . Bhavani Vanama** with fulfillment for the award of the degree of **Bachelor of Engineering** in **Computer Science and Engineering (AI & ML)** to the **Osmania University** is a record of bonafide work carried out by his under my guidance and supervision. The results embodied in this project report have not been submitted to any other University or Institute for the award of any graduation degree.

Mrs . Bhavani Vanama
Assistant Professor,
Internal Guide,
CSE (AI & ML) Dept.

Mr. P. Naresh Kumar
Assistant Professor,
Head of the Department,
CSE (AI & ML) Dept.

EXTERNAL EXAMINER

Submitted for Viva Voce Examination held on _____

INDEX :

ACKNOWLEDGEMENT

This is to place on our record my appreciation and deep gratitude to the persons without whose support this project would never been this successful. We are grateful to **Mr. Neil Gogte, Founder Director**, for facilitating all the amenities required for carrying out this project. It is with immense please that we would like to express our indebted gratitude to the respected **Prof. P.V.N Prasad, Principal, Keshav Memorial Engineering College,** for providing a great support and for giving us the opportunity of doing the project. We express our sincere gratitude to **Mrs. Deepa Ganu, Director Academics**, for providing an excellent environment in the college. We would like to take this opportunity to specially thank to **Mr.P.Naresh kumar, Assistant Professor & HoD, Department of CSE (AI & ML), Keshav Memorial Engineering College**, for inspiring us all the way and for arranging all the facilities and resources needed for our project. We would like to take this opportunity to thank our internal guide **Mrs . Bhavani Vanama , Assistant Professor, Department of CSE (AI & ML), Keshav Memorial Engineering College,** who has guided us a lot and encouraged us in every step of the project work. Her valuable moral support and guidance throughout the project helped us to a greater extent. We would like to take this opportunity to specially thank our Project Coordinator, **Mrs. Harini, Assistant Professor, Department of CSE (AI & ML), Keshav Memorial Engineering College**, who guided us in successful completion of our project. Finally, we express our sincere gratitude to all the members of the faculty of Department of CSE (AI & ML), our friends and our families who contributed their valuable advice and helped us to complete the project successfully.

BHEEMANPALLY ABHIJITH (245522748073)
Place: PANKAJ DESHMUKH (245522748075)
Date: PPC KALYAN (245522748115)

ABSTRACT

Insurance Advisor is an advanced virtual advising bot that streamlines the process of finding relevant insurance policies for users. Built with the MERN stack comprising MongoDB for database management, Express.js and Node.js for backend operations, and React.js for the user interface—the system ensures a seamless experience across platforms. The project integrates a Retrieval-Augmented Generation (RAG) model that enhances the bot's ability to provide accurate and contextual recommendations by blending both retrieval and generative techniques. This approach is particularly effective in complex domains like insurance, where users require precise information tailored to their needs.

A key innovation in Insurance Advisor is its use of the Gemini API for natural language processing (NLP) and text conversion, allowing the system to process and understand user queries in a more human-like manner. By converting raw text input into structured data, the Gemini API facilitates smooth interactions between users and the bot, making the conversation feel natural and intuitive.

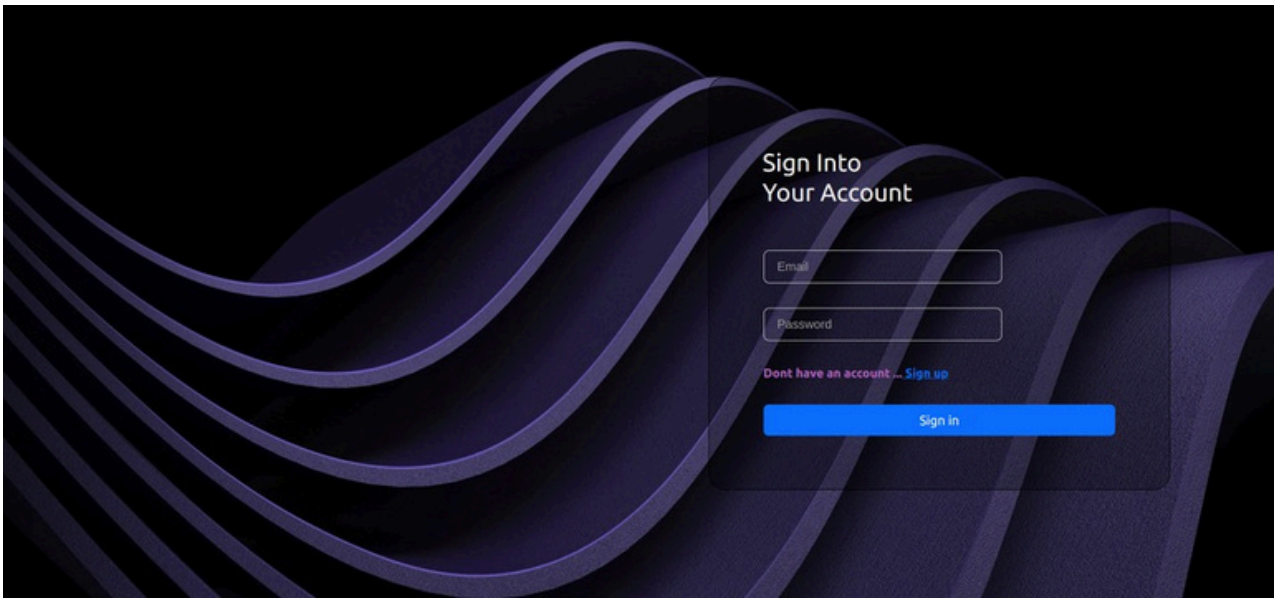
To further enhance the accuracy of policy recommendations, the project implements a sentence transformer for embedding user queries. This embedding technique transforms user queries into vector representations, enabling more precise searching within the policy database. By mapping the semantic meaning of queries to relevant entries in the database, Insurance Advisor significantly improves the relevance of the policies it retrieves, minimizing the time users spend searching for suitable options.

The bot’s architecture is designed to be both robust and scalable. It ensures fast query processing and delivers real-time results, even as the database of policies grows. Additionally, the modular design of the MERN stack allows for easy updates and integrations, making the system adaptable to future changes in the insurance market or user needs. This flexibility is crucial, given the evolving nature of insurance products and the increasing demand for personalized customer service.

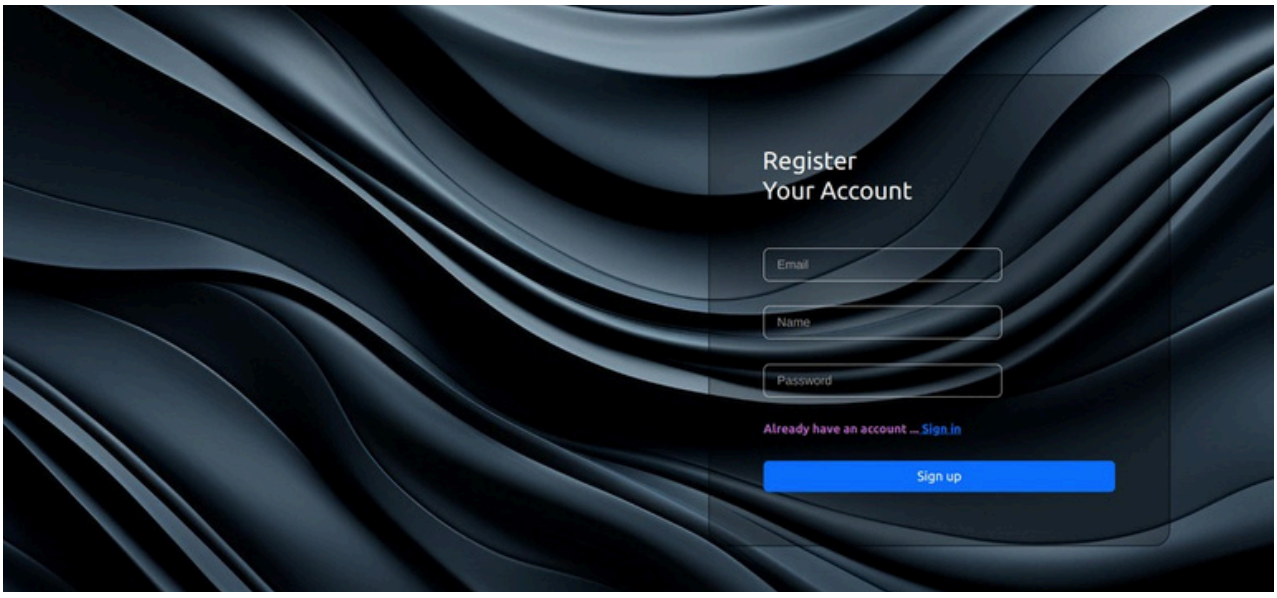
Insurance Advisor not only simplifies the user experience but also helps demystify the complexities of insurance policies. Whether users are searching for life, health, auto, or other types of insurance, the system provides clear, well-matched suggestions based on their specific requirements. By leveraging cutting-edge machine learning models and modern development frameworks, Insurance Advisor stands as an efficient and reliable tool for navigating the vast landscape of insurance options.

Keywords: Insurance Advisor, AI, MERN stack, NLP, FAISS, RAG model, Personalized Recommendations, Insurance Technology.

LIST OF FIGURES

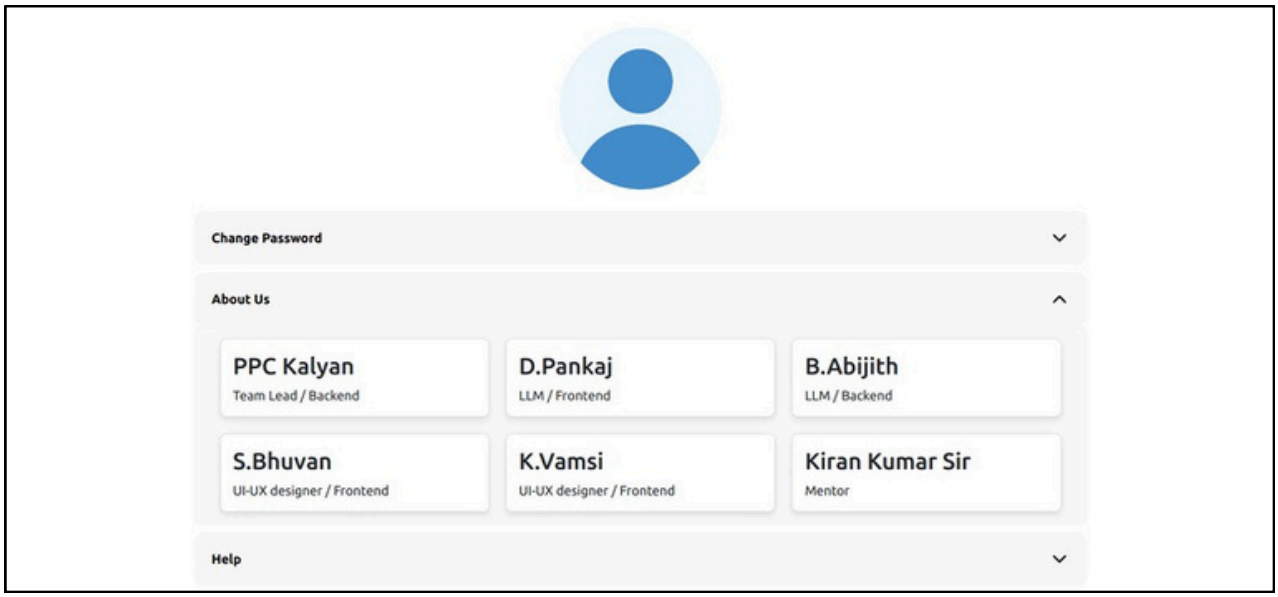


SIGN IN

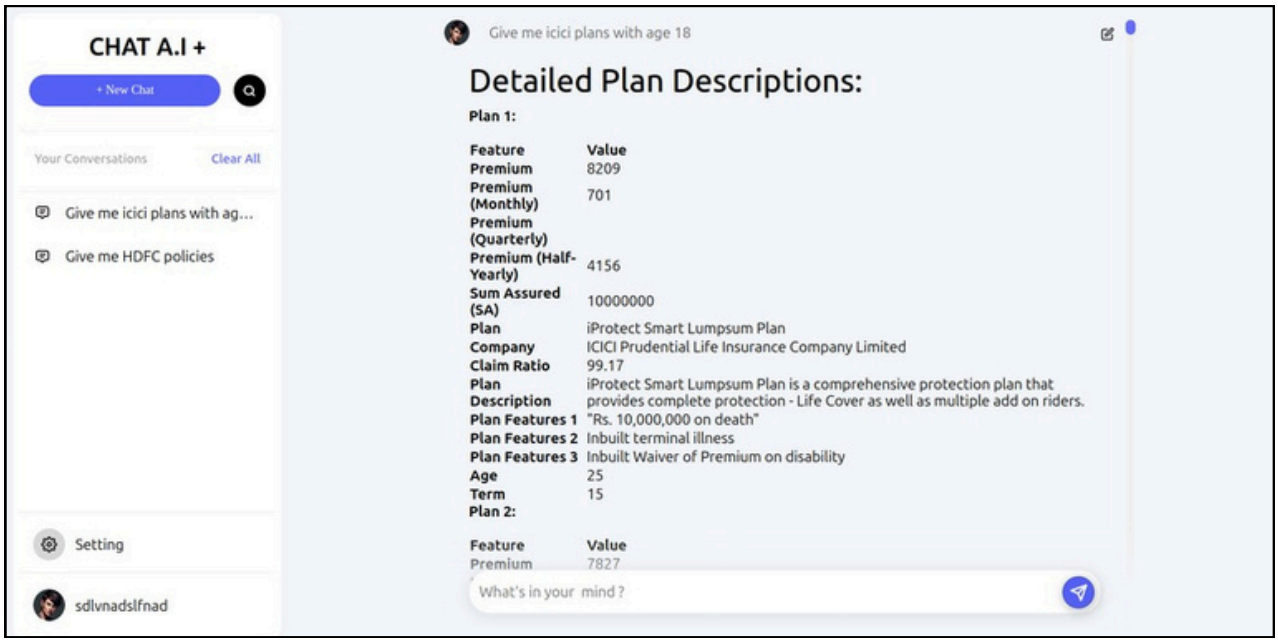


SIGN UP

LIST OF FIGURES



SETTINGS



HOME

INTRODUCTION

The evolution of digital technology has transformed how people access and process information, especially in complex domains like insurance. With a growing and diverse market, consumers face an overwhelming array of options—life, health, auto, and home insurance—each with varying coverage levels, premiums, and terms. This complexity makes it challenging for individuals to make informed decisions. The Insurance Advisor project addresses this by offering an AI-powered virtual advising bot that simplifies the insurance selection process. Leveraging modern web technologies and machine learning, it provides personalized recommendations, real-time information, and user-friendly guidance. By integrating natural language processing (NLP) and web scraping, the system gathers data from insurance websites to help users find policies tailored to their needs. Built on the MERN stack (MongoDB, Express.js, React.js, Node.js), it incorporates advanced tools like the Gemini API, sentence transformers, and FAISS for efficient and accurate results.

Aim and Objectives:

The Insurance Advisor aims to create a large language model (LLM)-based system that delivers personalized insurance recommendations and insights. Unlike traditional platforms, it enhances user experience by offering targeted suggestions based on preferences, queries, and specific needs. The system addresses the challenge of navigating complex policy options by providing dynamic, real-time, and adaptable recommendations. It scrapes data from sources like PolicyBazaar.com, cleans and processes it, and fine-tunes the LLM to power the recommendation engine. By analyzing user queries, it identifies relevant policies and delivers personalized suggestions.

Objectives include:

- Using web scraping to gather real-time data on insurance policies, coverage, and premiums.
- Preprocessing and cleaning scraped data for use in machine learning models.
- Developing a user-friendly chatbot interface for interacting with the LLM and inputting criteria.
- Fine-tuning the LLM on insurance-specific data for accurate, contextually relevant responses.
- Testing the system for stability, performance, and accuracy to ensure high-quality recommendations.

Technology Stack and Architecture : The Insurance Advisor is built on a flexible and powerful technology stack that ensures both performance and scalability. The project leverages the MERN stack, which consists of MongoDB for the database, Express.js and Node.js for backend logic, and React.js for the frontend user interface. This combination of technologies allows for a full-stack JavaScript solution, simplifying development and ensuring consistency

across the application. On the backend, the system runs on Nodemon, which facilitates hot-reloading during development, making it easier to update the server-side code. The frontend is developed using React.js, offering a smooth and responsive user experience that guides users through the process of finding the right insurance policy. The backend logic is further supported by a Flask server, which hosts the LLM and handles communication between the chatbot interface and the machine learning models.

One of the most critical components of the Insurance Advisor is its use of web scraping technologies like BeautifulSoup and Scrapy to gather insurance-related data. By targeting specific insurance websites, such as PolicyBazaar.com, the system retrieves detailed information about policies, including coverage options, terms, and premium rates. This data is then cleaned and preprocessed to remove noise, standardize text formats, and handle missing values. The cleaned data is stored in a structured format, such as CSV files, and used to fine-tune the large language model that powers the virtual advisor. Another key technology integrated into the system is natural language processing (NLP). The Insurance Advisor utilizes the Gemini API, which allows the bot to process and understand user queries in a conversational manner. By converting user input into structured data, the system is able to interpret and respond to inquiries effectively. Additionally, sentence transformers are used to embed user queries into high-dimensional vectors, allowing the system to find the most relevant matches within the database. To further optimize the retrieval process, the Insurance Advisor employs FAISS (Facebook AI Similarity Search). This tool allows for efficient similarity searching by calculating cosine similarities between the user's query and the available policies in the database. By comparing the semantic meaning of the user's input with the data, FAISS ensures that the most relevant policies are retrieved and recommended to the user.

Workflow and User Interaction The user experience with the Insurance Advisor begins with a simple interaction through a chatbot interface. Users can input their criteria or ask questions related to different types of insurance policies, such as life, health, auto, or home insurance. The system processes these queries using the Gemini API and generates embeddings using sentence transformers. The query embeddings are passed to the RAG model (Retrieval-Augmented Generation model), which retrieves relevant policies from the database.
To ensure that the recommendations are accurate and personalized, FAISS is used to search for policies that closely match the user's query. This process results in the retrieval of insurance policies that are ranked by relevance. The bot not only provides the relevant policy suggestions but also offers additional context or explanations to help users understand the different options available to them.

LITERATURE SURVEY

1.Existing System

Insurance advisory and recommendation systems have evolved over the past decade, largely driven by the rapid growth of digital platforms and advancements in artificial intelligence. Existing systems primarily fall into the following categories:

1.1 Insurance Aggregators

Popular insurance aggregators such as PolicyBazaar, CompareTheMarket, and GoCompare act as intermediaries, allowing users to compare various insurance policies across different providers. These platforms gather data from multiple insurers and present them in a structured format, enabling users to compare premiums, coverage, and policy terms. However, these systems focus mainly on price comparison and do not provide personalized, in-depth guidance. They lack the intelligence to understand specific user needs beyond filtering for certain types of insurance or budget constraints.

Limitations:

- **Generic Comparisons:** Recommendations are primarily based on premiums and policy features, without considering individual needs and circumstances.
- **Limited Personalization:** These platforms typically offer filtering options but do not offer personalized guidance tailored to the user's situation.
- **Static Data:** Most aggregator platforms rely on static data feeds from insurers, limiting their ability to provide real-time insights and updates.

1.2 Insurance Chatbots

Several insurers and third-party platforms have integrated chatbots into their services to automate customer queries and facilitate the purchase of insurance policies. Examples include **GEICO's virtual assistant, Allstate's chatbot, and Lemonade's AI Maya**. These bots can answer basic queries, recommend insurance products, and guide users through policy purchase processes. However, their recommendation capabilities are limited to the specific products offered by the provider or affiliated insurers.

Limitations:

- Product-Centric: Recommendations are confined to the insurer's portfolio, making the advice less objective or comprehensive.
- Limited Understanding: Basic chatbots lack the ability to deeply understand complex user queries and cannot offer highly personalized recommendations.

1.3 Recommendation Systems in Insurance

Some advanced insurers have begun integrating machine learning-based recommendation systems into their platforms. These systems analyze user data, including demographics, past behavior, and financial status, to suggest suitable insurance products. For instance, AI-driven recommendation engines use historical customer data to recommend policies with

high conversion rates. However, these systems are often siloed within specific companies and suffer from data privacy limitations, as they depend heavily on internal data sets without tapping into broader, real-time market data.

Limitations:

- **Data Constraints:** These systems often rely on historical internal data, which may not be sufficient for generating real-time, comprehensive insights.
- **Limited Scope:** Recommendations are usually constrained to a small subset of policies available through the specific insurance provider, rather than considering a broad spectrum of market options.

2. Proposed System: Insurance Advisor

The **Insurance Advisor** project is designed to overcome the limitations of existing systems by combining advanced AI, machine learning, and natural language processing technologies to create a more dynamic, personalized, and user-friendly insurance recommendation system. This proposed system integrates **web scraping, LLMs (large language models)**, and **similarity search algorithms** to offer real-time data, personalized recommendations, and more insightful guidance to users.

2.1 System Overview

The proposed system, built on the **MERN stack**, leverages web scraping techniques to gather real-time data from various insurance-related websites, such as **PolicyBazaar.com**, to extract policy details, coverage options, and premiums. Using the **Gemini API** for natural language processing and **sentence transformers** for embedding user queries, the system searches for relevant policies in its database. To enhance precision, the system uses **FAISS** for cosine similarity calculations to match user queries with the most relevant policies.

In addition to retrieving relevant insurance policies, the proposed system provides contextual explanations and recommendations, ensuring users fully understand their options and can make well-informed decisions.

2.2 Benefits of the Proposed System

1. **Personalization**
   - **Tailored Recommendations:** Unlike existing systems that primarily focus on price comparison, the Insurance Advisor offers personalized recommendations based on the user's specific needs, preferences, and criteria. By embedding user queries into high- dimensional vectors and using similarity search algorithms, the system ensures that users receive recommendations that are highly relevant to their unique circumstances.
   - **Natural Interaction:** Through NLP powered by the Gemini API, the system enables more human-like interactions, making it easier for users to communicate their needs and receive customized advice.

2. Real-Time Data

- **Up-to-Date Information:** By utilizing web scraping to gather real-time data from insurance websites, the system ensures that users always have access to the latest policies, premiums, and coverage details. This is a significant improvement over existing aggregators that often rely on static data feeds.
- **Dynamic Updates:** The system continuously updates its database, ensuring that the recommendations are not only personalized but also based on the most recent market information.

3. Advanced Search Capabilities

- **Semantic Matching:** Using sentence transformers and FAISS, the system can semantically match user queries with insurance policies, ensuring that the recommendations are based on the meaning of the query rather than just keyword matching. This deep understanding of user intent results in more accurate and relevant policy suggestions.
- **Cosine Similarity Search:** FAISS allows for fast and efficient searching across large datasets, improving the response time of the system and ensuring that users receive recommendations without unnecessary delays.

4. Comprehensive Coverage

- **Variety of Policies:** The proposed system is designed to cover a wide range of insurance products, from life and health insurance to auto and property insurance, ensuring that users can find policies that suit all their needs in one place.
- **Cross-Provider Insights:** Unlike insurance chatbots that are tied to specific companies, the Insurance Advisor aggregates data across multiple providers, offering more objective and diverse recommendations.

5. User-Friendly Interface

- **Interactive Chatbot**: The system features an interactive chatbot built using React.js, offering a user-friendly experience that guides users through the process of finding the best insurance policies. The chatbot interface simplifies the user's experience by providing clear and concise answers to queries.
- **Seamless Integration**: Built on the MERN stack, the system is highly scalable and can be easily integrated with other platforms or extended with new features.

6. Scalability and Flexibility

- **Modular Design:** The modular design of the MERN stack allows for easy updates and future expansions, making the system highly adaptable to evolving market trends and user needs.
- **Machine Learning Expansion:** The system's LLM can be further fine-tuned and expanded as more data is collected, ensuring that it continues to improve in terms of accuracy and performance.

SOFTWARE REQUIREMENTS SPECIFICATION

3.1 Overall Description

The **Insurance Advisor** is a virtual advising bot designed to assist users in selecting the best insurance policies based on their unique needs. It leverages real-time data scraped from various insurance websites and uses advanced natural language processing (NLP) to interpret user queries and offer personalized recommendations. The system's core functionalities are built using the MERN stack (MongoDB, Express.js, React.js, Node.js) for scalability and responsiveness, while the AI models are hosted on Flask. The system also employs web scraping technologies, the Gemini API for NLP, sentence transformers for query embedding, and FAISS for fast similarity search. The **Insurance Advisor** is designed to:

- Scrape real-time data from selected insurance websites.
- Process and store this data in structured formats for further analysis.
- Use AI-driven models to provide personalized policy recommendations.
- Provide an intuitive user interface through a chatbot for easy user interaction.

Key Features:

- **Real-Time Data Integration:** The system updates insurance policy information regularly through web scraping.
- **Personalized Recommendations:** AI models provide tailored suggestions based on the user's query.
- **User-Friendly Interface:** A chatbot guides users through the process of selecting insurance policies.
- **Efficient Search:** FAISS ensures fast and accurate retrieval of relevant policies based on user input.

3.2 Operating Environment

Hardware Requirements:

- **Server Requirements:**
  - Processor: Minimum Intel i5 or equivalent.
  - RAM: Minimum 8 GB (recommended 16 GB or higher for large-scale operations).
  - Storage: SSD with a minimum of 250 GB for fast read/write operations.
  - Network: Reliable internet connection for real-time web scraping and API integration.
- **Client-Side Requirements:**
  - Device: PC, tablet, or smartphone.
  - Processor: Minimum dual-core processor.
  - RAM: Minimum 2 GB for basic operation.
  - Storage: 500 MB for installation and data storage.
  - Browser Compatibility: Google Chrome, Firefox, Safari, or Microsoft Edge.

Software Requirements:

- **Server-Side:**
  - Operating System: Ubuntu 18.04+ or Windows Server 2016+.
  - Frameworks: Node.js (for backend), Express.js (for REST APIs), Flask (for AI model hosting).
  - Database: MongoDB (NoSQL database).
  - Machine Learning Libraries: Sentence Transformers, FAISS, and the Gemini API.
  - Package Manager: npm for JavaScript dependencies.
- **Client-Side:**
  - Frontend Framework: React.js (for building the user interface).
  - Browser: Latest version of Google Chrome, Firefox, Safari, or Microsoft Edge.
  - Compatibility: Compatible with modern web browsers on desktop and mobile devices.

User Characteristics

The Insurance Advisor system is designed for a wide range of users who seek personalized guidance in selecting insurance policies. These users are likely to vary in technical skill and insurance knowledge.

- **Primary Users:**
  - Consumers Seeking Insurance: Users looking for information about different insurance policies (life, health, auto, home, etc.).
  - Non-Technical Users: The system is designed to be user-friendly, allowing individuals with no technical background to easily interact with the chatbot interface.
- **Secondary Users:**
  - Insurance Professionals: Users in the insurance industry may use the system to analyze competitors or to get a quick overview of various policies on the market.
  - Developers and System Admins: These users may interact with the backend system for maintenance, updates, or troubleshooting.
- **User Characteristics:**
  - Varied Technical Expertise: Users may range from individuals with little technical knowledge to more tech-savvy individuals.
  - Insurance Knowledge: Some users may have detailed knowledge of insurance policies, while others are novices looking for guidance.
  - Mobile-Friendly Users: A significant portion of users may prefer accessing the system on mobile devices, requiring a responsive design.

3.3 Functional Requirements

The following are the primary functional requirements of the Insurance Advisor system:

3.3.1 User Interaction

- **Chatbot Interface:** Users will interact with the system via a chatbot, which will guide

them through the insurance selection process. The chatbot will understand and respond to natural language queries.

- **Query Processing:** The system should be able to process user queries through the chatbot and provide personalized insurance recommendations.

3.3.2 Real-Time Data Scraping

- **Web Scraping:** The system should continuously scrape data from selected insurance websites (e.g., PolicyBazaar.com), extracting relevant details such as policy types, premiums, and coverage options.
- **Data Storage:** All scraped data should be cleaned, preprocessed, and stored in MongoDB in a structured format (e.g., CSV or JSON) for further analysis.

3.3.3 Policy Recommendations

- **NLP Integration:** The system should use the Gemini API for NLP to process and understand user queries.
- **Query Embedding:** User queries should be converted into embeddings using sentence transformers for accurate searching.
- **Similarity Search:** The system should use FAISS to conduct cosine similarity searches between the embedded query and the stored policy data, ensuring the most relevant results are returned.
- **Policy Ranking:** Recommended policies should be ranked by relevance to the user's query, with the option for users to filter based on additional criteria (e.g., premium amount, coverage type).

3.3.4 User Feedback

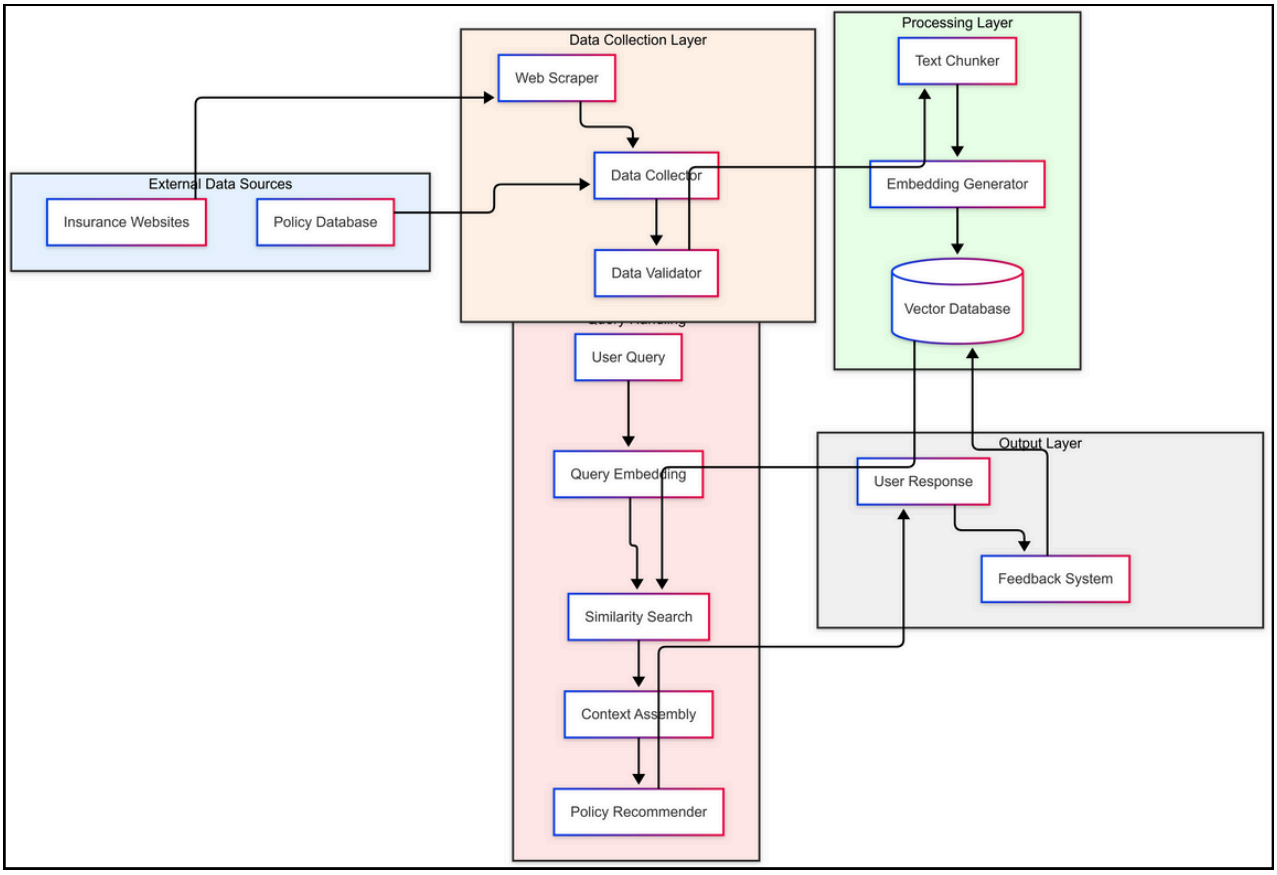
- **Feedback Mechanism:** The system should allow users to provide feedback on the recommendations provided. This feedback will help refine the model over time for better performance.

3.3.5 System Maintenance

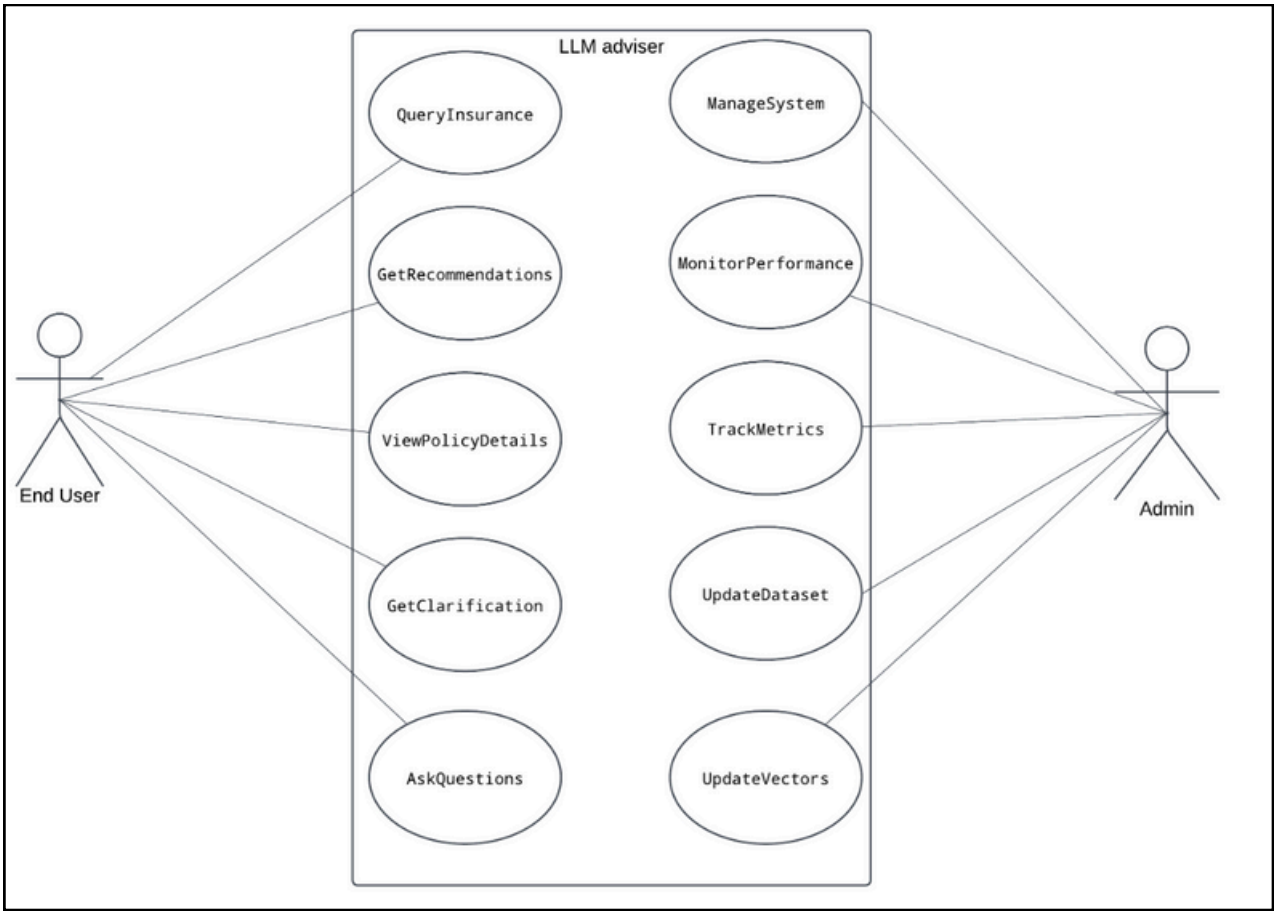
- **Scalability:** The system should be designed to scale as user demand increases, especially as more data is scraped and processed.
- **Error Logging and Handling:** The system should log any errors encountered during operation and notify system administrators for timely resolution.

In summary, the Insurance Advisor aims to combine state-of-the-art web technologies, AI, and NLP to deliver a comprehensive, real-time insurance advisory service that is both user-friendly and highly functional.

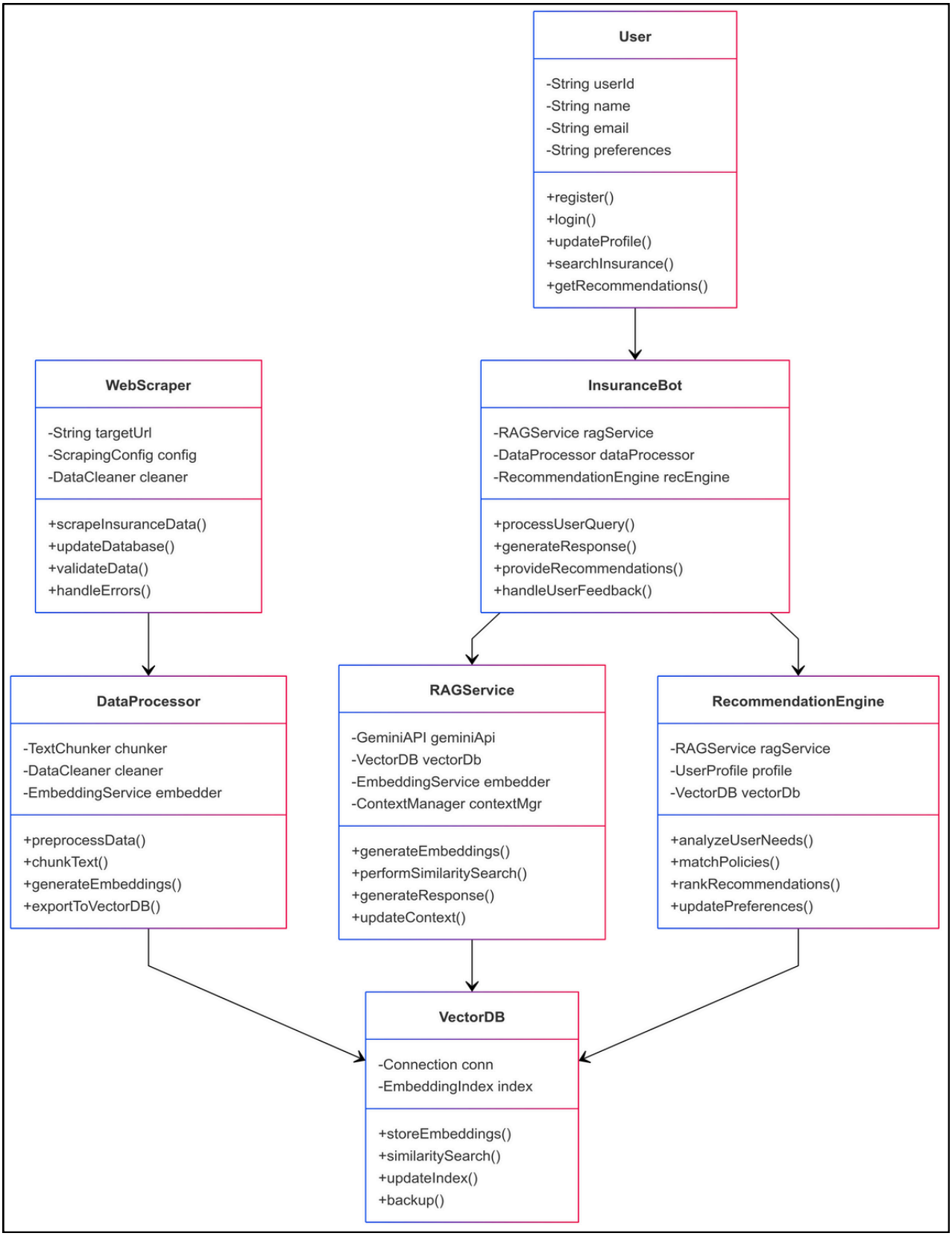
SYSTEM DESIGN



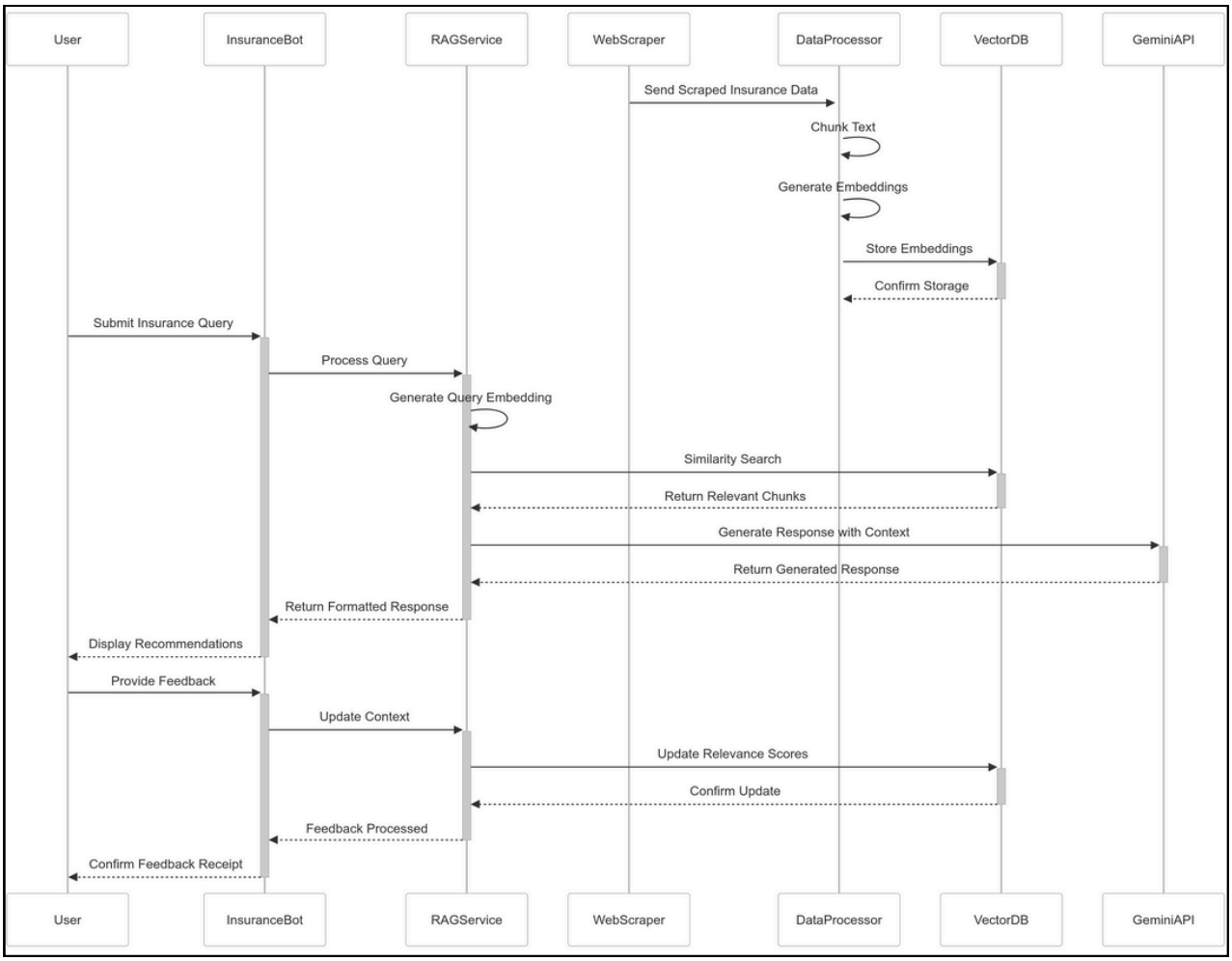
DATA FLOW DIAGRAM



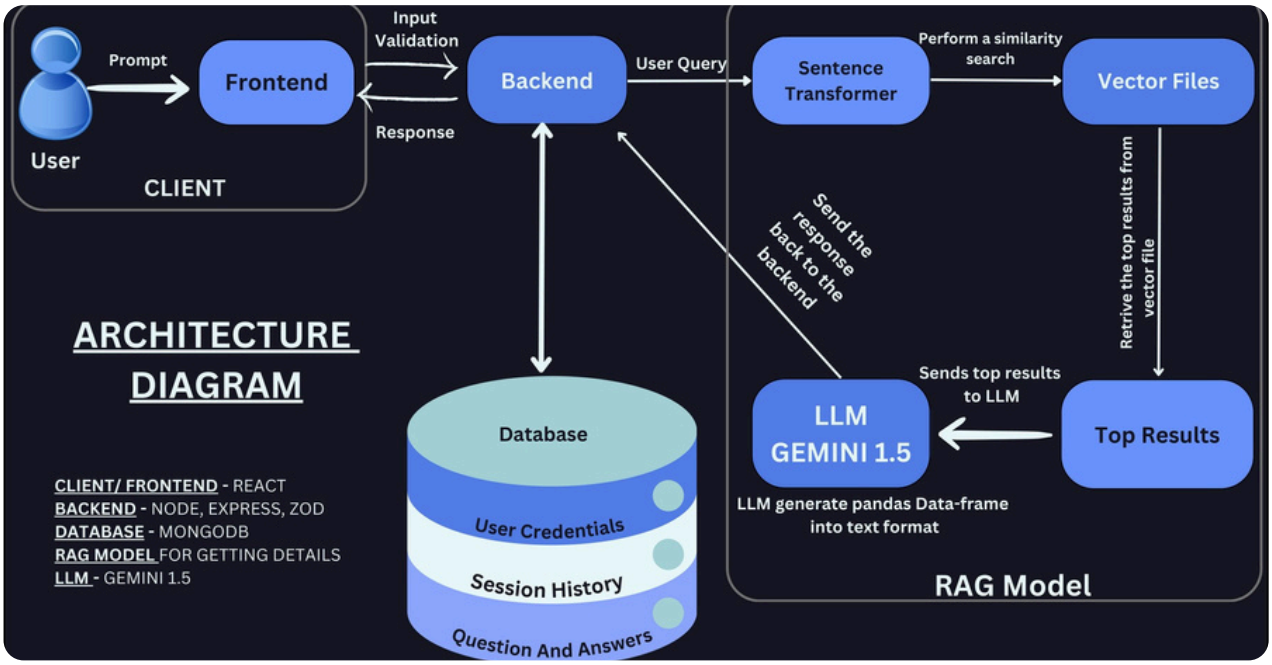
USE-CASE DIAGRAM



CLASS DIAGRAM



SEQUENCE DIAGRAM



ARCHITECTURE

4.1 Components of Architecture

- **User:** Provides input (prompt) via the chatbot interface.
- **Frontend:** Collects the user's prompt and displays the response. Built with React.js.
- **Backend:** Validates the user's prompt and coordinates interaction between the frontend, embedding component, LLM, and database. Developed using Node.js and Express.js.
- **Embedding:** Converts the user's text into vector embeddings to prepare for similarity search using Sentence Transformers.
- **Vector Database:** Stores insurance policy data as vector embeddings Uses FAISS for efficient similarity search based on cosine similarity.
- **LLM (Large Language Model):** Generates responses based on the retrieved data from the vector database and user input. Uses Gemini API with RAG for enhanced text generation.
- **Database:** Stores user credentials and persistent data using MongoDB.

Flow of the System (Explanation)

- **User Interaction:** The user interacts with the Frontend (React.js) by submitting a prompt, such as a query for insurance recommendations.
- **Prompt Validation & Backend Processing:** The Frontend sends the user's prompt to the Backend (Node.js/Express.js) for validation. The backend ensures that the prompt is correctly formatted and ready for further processing.
- **Embedding Process:** Once validated, the Backend forwards the prompt to the Embedding component, which converts the text into vector embeddings. These embeddings represent the query in a numerical format, making it suitable for similarity comparison.
- **Vector Search:** The Embedding is then matched against pre-stored vector embeddings in the Vector Database (FAISS). Using cosine similarity, the system retrieves the top insurance policies relevant to the user's query.
- **Response Generation via LLM:** The Vector Database returns the most similar insurance policies to the LLM. The LLM (hosted on Flask using the Gemini API) processes these results and generates a coherent response, which might include summaries or further insights into the policies.
- **Returning the Response:** The Backend receives the generated text from the LLM and sends it back to the Frontend, where it is displayed to the user as a natural language response.
- **User Database Interaction:** During the interaction, if necessary (e.g., login or account management), the Backend interacts with the Database (MongoDB) to retrieve or store user credentials and data.

This flow ensures that user queries are processed efficiently, with personalized recommendations generated based on real-time data. The architecture ensures smooth interaction between the user, embedding search, and LLM, all while maintaining scalability and speed.

4.2 Technology Stack

The Insurance Advisor system is built using a combination of modern web development frameworks and machine learning technologies, optimized for scalability, real-time data retrieval, and AI-powered recommendations.

- **Frontend:**
  - Technology: React.js
  - Purpose: Provides a responsive and interactive user interface, handling user inputs and displaying responses from the LLM.
- **Backend:**
  - Technology: Node.js with Express.js
  - Purpose: Acts as the bridge between the frontend, embeddings, LLM, and vector database, validating user inputs, orchestrating data retrieval, and managing responses.
  - Secondary Stack: Flask is used to host the LLM scripts that handle advanced machine learning tasks.
- **Embedding:**
  - Technology: Sentence Transformers
  - Purpose: Converts user queries into vector embeddings to enable similarity searches within the vector database.
- **Vector Database:**
  - Technology: FAISS (Facebook AI Similarity Search)
  - Purpose: Stores vectorized data and efficiently retrieves top results based on cosine similarity searches.
- **LLM (Large Language Model):**
  - Technology: Gemini API with Retrieval-Augmented Generation (RAG)
  - Purpose: Processes and generates natural language text based on the retrieved data and user input.
- **Database:**
  - Technology: MongoDB
  - Purpose: Stores persistent user data, including user credentials and potentially user-specific insurance data.

This architecture and technology stack ensure that the Insurance Advisor system is both scalable and capable of delivering real-time personalized insurance recommendations.

IMPLEMENTATION

Implementation of the Project The implementation of our project involves a carefully orchestrated process that integrates multiple technologies and frameworks to build a robust, scalable, and secure web application. The following sections detail the steps and considerations involved in bringing this project from concept to reality.

1. Backend Development

The backend of our project is developed using Node.js and Express.js. Node.js is selected for its non-blocking, event-driven architecture, which is ideal for handling multiple concurrent requests efficiently. Express.js serves as the framework for building our RESTful API, providing a structured way to define routes and handle HTTP requests.

Database Integration Our application utilizes MongoDB as the primary data store. MongoDB is a NoSQL database that stores data in a flexible, JSON-like format, making it well-suited for applications that require scalability and the ability to handle varying data structures. The database is connected to the Node.js application using Mongoose, an Object Data Modeling (ODM) library. Mongoose simplifies data interactions by providing a schema-based solution, ensuring data consistency, integrity through validation and middleware.

User Authentication User authentication is implemented using JWT (JSON Web Tokens). JWTs are generated upon successful user login and are included in subsequent requests to verify the user's identity. This stateless authentication mechanism is highly scalable, as it does not require the server to maintain session data between requests. Passwords are securely handled using Bcrypt, which hashes passwords before storing them in the database, protecting user credentials from potential breaches.

Data Validation To ensure data integrity, Zod is used for schema validation. Zod provides a straightforward way to declare and validate the shape of incoming data, preventing invalid or malicious data from being processed by the backend. This validation step is integrated into the request handling pipeline of Express.js, ensuring that all data conforms to the expected structure before any business logic is applied.

2. Frontend Development

The frontend is built using React with TypeScript. React is chosen for its component-based architecture, which promotes reusability and simplifies the management of complex user interfaces. TypeScript adds type safety to the application, reducing the risk of runtime errors and improving the overall quality of the codebase.

User Interface React components are used to build the user interface, ensuring a dynamic and responsive experience. The frontend communicates with the backend through the RESTful API, sending requests and rendering the data received from the server.

The use of TypeScript enhances development efficiency by providing better autocompletion, refactoring tools, and inline documentation, all of which contribute to a more maintainable and scalable frontend codebase.

3. API Design and Implementation

The communication between the frontend and backend is facilitated through a well-designed RESTful API. The API endpoints are defined in Express.js and handle all data transactions between the client and server. Each endpoint is carefully designed to be stateless, ensuring that each request is independent and contains all the necessary information for processing. API Testing To ensure the reliability of the API, we use Thunder Client for testing. Thunder Client, integrated within Visual Studio Code, allows us to send HTTP requests to our API endpoints and validate the responses. This testing process is essential for identifying issues and verifying that the API behaves as expected under various scenarios.

4. Security Measures

Security is a critical aspect of our application, with several layers of protection implemented throughout the stack. JWT tokens secure user sessions, while Bcrypt ensures that passwords are stored securely. Data validation using Zod further protects the backend by filtering out potentially harmful inputs. Additionally, all API communications are secured with HTTPS, ensuring that data exchanged between the client and server is encrypted and safe from interception.

5. Version Control and Development Workflow

Throughout the development process, Git is used as the version control system. Git enables us to track changes, manage branches, and collaborate effectively within the development team. We maintain a structured Git workflow, where feature development, bug fixes, and updates are managed through separate branches and merged into the main codebase after thorough testing.

Development Environment

Visual Studio Code (VS Code) is our chosen Integrated Development Environment (IDE). VS Code's rich feature set, including extensions for Node.js, React, TypeScript, and Git, makes it an ideal tool for our development needs. The integrated terminal, debugger, and Git tools allow us to streamline our workflow, from writing and testing code to managing version control.

6. Large Language Model (LLM) Integration

To enhance the capabilities of our application, we have integrated Gemini 1.5, a Large Language Model (LLM), into our system. This model is deployed as a microservice, interacting with the backend through secure API endpoints. It is fine-tuned to handle specific language processing tasks within our application, such as chatbot interactions and content generation, providing a more intelligent and responsive user experience.

5.1 CODE:

// App.js

```
import { useEffect } from "react";
import { Routes, Route, useNavigationType, useLocation } from "react-router-dom";
import SignIn from "./pages/SignIn";
import SignUp from "./pages/SignUp";
import Start from "./pages/Start";
import Body from "./pages/Body";
import Settings from "./pages/Settings";
function App() {
  const action = useNavigationType();
  const location = useLocation();
  const pathname = location.pathname;
  useEffect(() => {
    if (action !== "POP") {
      window.scrollTo(0, 0);
    }
  }, [action, pathname]);
  useEffect(() => {
    let title = "";
    let metaDescription = "";
    switch (pathname) {
      case "/":
        title = "";
        metaDescription = "";
        break;
    }
    if (title) {
      document.title = title;
    }
    if (metaDescription)
      const metaDescriptionTag: HTMLMetaElement | null = document.querySelector(
        'head > meta[name="description"]'
      );
    if (metaDescriptionTag) {
      metaDescriptionTag.content = metaDescription;
    }
  }, [pathname]);
  return (
    <Routes>
```

```
          <Route path="/" element={<Start />} />
          <Route path="/signin" element={<SignIn />} />
          <Route path="/signup" element={<SignUp />} />
          <Route path="/home" element={<Body />} />
          <Route path="/settings" element={<Settings />} />
        </Routes>
    );
}
export default App;
```

// Index.js

```
const express = require("express");
const cors = require("cors");
const AuthRouter = require("./routes/AuthRouter");
const QueryRouter = require("./routes/QueryRouter");
const app = express();

// Middleware
app.use(cors());
app.use(express.json());

// Routes
app.use("/api/auth", AuthRouter);
app.use("/api/query", QueryRouter);

// Start Server
app.listen(8081, () => {
    console.log("Backend Started on Port 8081");
});
```

// QueryRouter.js

```
require("dotenv").config();
const { Router } = require("express");
const z = require("zod");
const { User, Question } = require("../db/index"); // Adjust the path as needed
const authMiddleware = require("../middlewares/authMiddleware");
const { default: mongoose } = require("mongoose");
const axios = require("axios");
const router = Router();
```

```
// Zod schemas for input validation
const llmQuerySchema = z.object({
  query: z.string().nonempty(),
  current_session_id: z.string().nonempty(),
});
const createNewSessionSchema = z.object({});
const getSessionDetailsSchema = z.object({
  session_to_change_id: z.string().nonempty(),
});
const getAllSessionsSchema = z.object({});

// Function to update session questions
const updateSessionQuestions = (user, sessionId, question) => {
  let sessionUpdated = false;
  for (let session of user.sessionQuestions) {
    if (session.sessionId === sessionId && session.question === "") {
      session.question = question;
      sessionUpdated = true;
      break;
    }
  }
};
// Endpoint to handle LLM queries and store the result in the database
router.post("/llmquery", authMiddleware, async (req, res) => {
  const validation = llmQuerySchema.safeParse(req.body);
  if (!validation.success) {
    return res.status(400).json({ errors: validation.error.errors });
  }
  const { query, current_session_id } = validation.data;
  try {
    const userId = req.id;
    // Send query to LLM service
    let answer = await axios.post("http://localhost:5000/get_policies", {
      question: query,
    });
    answer = answer.data.response;
    // Find the user and the session
    const user = await User.findById(userId);
    if (!user) {
      return res.status(404).json({ message: "User not found" });
    }
```

```
        // Create a new Question entry
        const newQuestion = new Question({
            question: query,
            answer: answer,
            userId: userId,
            sessionId: current_session_id,
        });
        await newQuestion.save();
        // Update user's session data
        if (!user.sessionData.has(current_session_id)) {
            user.sessionData.set(current_session_id, []);
        }
        user.sessionData.get(current_session_id).push(newQuestion);
        // Update session questions
        updateSessionQuestions(user, current_session_id, query);
        await user.save();
        res.status(200).json(Array.from(user.sessionData.get(current_session_id)));
    } catch (error) {
        console.error(error);
        res.status(500).json({ message: "Server error" });
    }
});
// Endpoint to create a new session
router.post("/createNewSession", authMiddleware, async (req, res) => {
    const validation = createNewSessionSchema.safeParse(req.body);
    if (!validation.success) {
        return res.status(400).json({ errors: validation.error.errors });
    }
    try {
        const userId = req.id;
        const newSessionId = new mongoose.Types.ObjectId().toString();
        const user = await User.findById(userId);
        if (!user) {
            return res.status(404).json({ message: "User not found" });
        }
        // Add new session
        user.sessionQuestions.push({ sessionId: newSessionId, question: "" });
        await user.save();
        res.status(200).json({ newSessionId });
    } catch (error) {
        console.error(error);
```

```
        res.status(500).json({ message: "Server error" });
    }
});
// Endpoint to get session details
router.post("/getSessionDetails", authMiddleware, async (req, res) => {
    const validation = getSessionDetailsSchema.safeParse(req.body);
    if (!validation.success) {
        return res.status(400).json({ errors: validation.error.errors });
    }
    const { session_to_change_id } = validation.data;
    try {
        const userId = req.id;
        const user = await User.findById(userId);
        if (!user) {
            return res.status(404).json({ message: "User not found" });
        }
        const sessionData = user.sessionData.get(session_to_change_id);
        res.status(200).json(sessionData ? Array.from(sessionData) : []);
    } catch (error) {
        console.error(error);
        res.status(500).json({ message: "Server error" });
    }
});
// Endpoint to get all sessions
router.post("/getAllSessions", authMiddleware, async (req, res) => {
    const validation = getAllSessionsSchema.safeParse(req.body);
    if (!validation.success) {
        return res.status(400).json({ errors: validation.error.errors });
    }
    try {
        const userId = req.id;
        const user = await User.findById(userId);
        if (!user) {
            return res.status(404).json({ message: "User not found" });
        }
        // Format session details
        const sessionDetails = Array.from(user.sessionData).map(([sessionId, questions]) => ({
            sessionId,
            questions,
        }));
        res.status(200).json(sessionDetails);
```

```
    } catch (error) {
        console.error(error);
        res.status(500).json({ message: "Server error" });
    }
});
module.exports = router;
```

Flask Code

```
import pickle
import pandas as pd
import torch
import google.generativeai as genai
import os
from transformers import AutoTokenizer, AutoModel
from flask import Flask, request, jsonify

# Load saved objects
with open("embeddings_dataset.pkl", "rb") as f:
    embeddings_dataset = pickle.load(f)

with open("tokenizer.pkl", "rb") as f:
    tokenizer = pickle.load(f)

with open("model.pkl", "rb") as f:
    model = pickle.load(f)

# Configure Gemini API
api_key = "AIzaSyAt7ISPqMfg_D8Kj6y13uVS4RoQVN VkcUA"
genai.configure(api_key=api_key)
gemini_model = genai.GenerativeModel('gemini-1.5-flash')

app = Flask(__name__)

# Function to perform CLS pooling
def cls_pooling(model_output):
    """Extracts the first token's hidden state as the embedding."""
    return model_output.last_hidden_state[:, 0]

# Function to generate embeddings
def get_embeddings(text_list):
```

```
    """Tokenizes input text and returns embeddings."""
    encoded_input = tokenizer(text_list, padding = True, truncation = True, return_tensors
="pt")
    model_output = model(**encoded_input)
    return cls_pooling(model_output).detach().numpy()

@app.route('/get_policies', methods=['POST'])
def get_policies():
    """Handles policy retrieval based on user query."""
    data = request.json
    question = data.get('question', '')

    if not question:
        return jsonify({"error": "No question provided"}), 400

    question_embedding = get_embeddings([question])

    if question_embedding is None:
        return jsonify({"error": "Failed to generate embeddings"}), 500

    # Retrieve nearest examples
    scores, samples = embeddings_dataset.get_nearest_examples("embeddings",
question_embedding, k=5)
    samples_df = pd.DataFrame.from_dict(samples)
    samples_df["scores"] = scores
    samples_df.sort_values("scores", ascending=False, inplace=True)

    # Create prompt for Gemini API
    prompt = samples_df.iloc[:, :-3].to_csv(index=False)
    response = gemini_model.generate_content("Generate each row (Plan) in a detailed way\n"
+ prompt)

    return jsonify({"response": response.text})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True)
```

RESULTS AND DISCUSSIONS

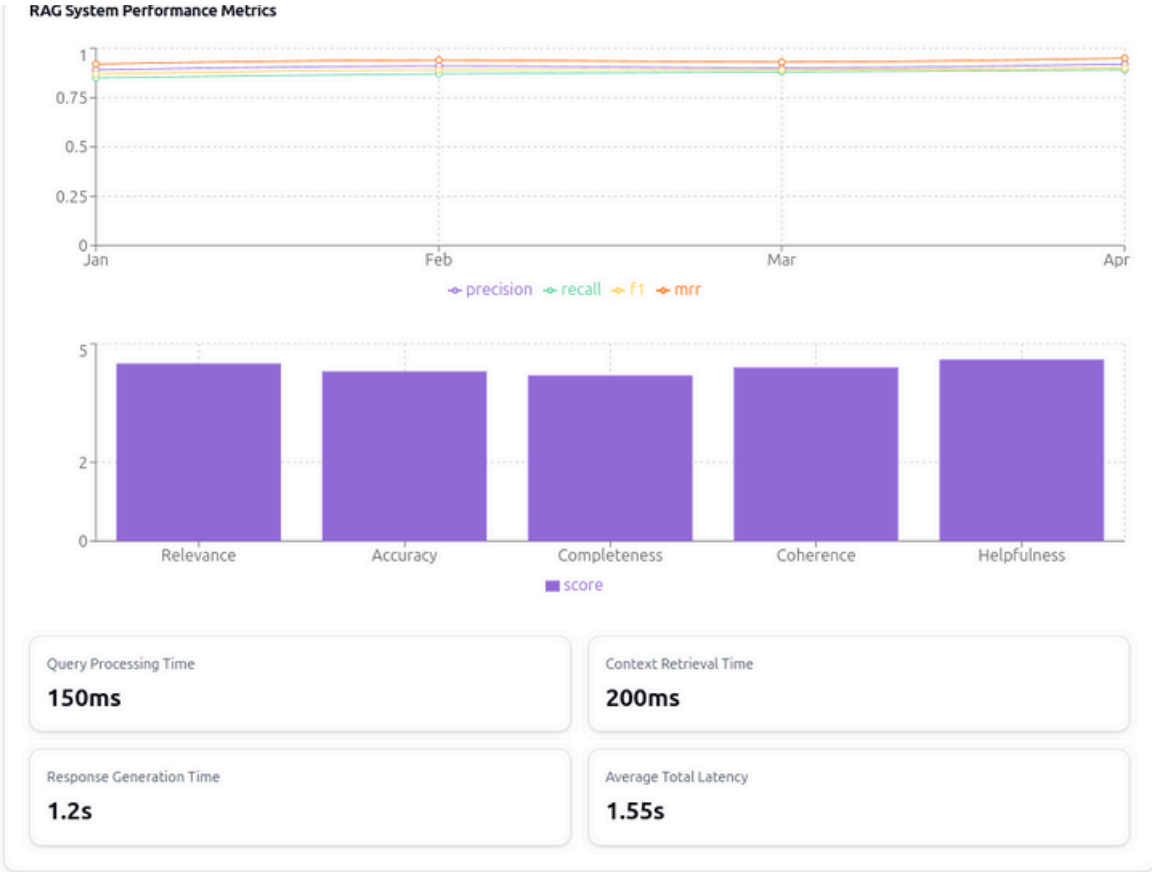
The Insurance Advisor successfully achieved its primary goal of providing personalized insurance recommendations to users. By leveraging web scraping, the system gathered real-time data from insurance websites and processed this data using advanced machine learning models.

Results:

- Accurate Recommendations: Through the use of FAISS and cosine similarity, the system efficiently retrieved relevant insurance policies based on user queries.
- Natural Language Responses: The integration of the Gemini API with Retrieval-Augmented Generation (RAG) enabled the LLM to generate coherent and contextually appropriate responses for users.
- Scalability: The MERN stack (MongoDB, Express.js, React.js, and Node.js) ensured that the system is scalable, with the backend handling user requests efficiently and the frontend providing a smooth user experience.

Discussions:

- Effectiveness: The combination of NLP and vector search technologies provided accurate and relevant results, enhancing the user experience in navigating complex insurance policies.
- Challenges: Fine-tuning the LLM for insurance-specific data presented challenges in training and inference times, but careful optimization allowed for improved performance.
- Future Improvements: There is room for expanding the range of policies and improving response times through further optimizations in data handling and LLM performance.



CONCLUSION AND FUTURE SCOPE

Conclusion

The Insurance Advisor project successfully developed a virtual advising bot capable of providing personalized insurance recommendations based on user queries. By integrating technologies such as the MERN stack, Retrieval-Augmented Generation (RAG), FAISS, and Gemini API, the system demonstrated efficient real-time processing and accurate delivery of relevant insurance policies. The use of NLP for query processing and cosine similarity for policy retrieval significantly enhanced the user experience, offering clear, tailored advice. The project met its goal of simplifying the often-complex task of comparing and understanding various insurance policies for users.

Future Scope

There are several areas for future improvement and expansion of the Insurance Advisor:

1. Expanded Data Sources: Incorporating more insurance providers and data sources to offer users a wider variety of policies and coverage options.
2. Enhanced NLP Capabilities: Improving the LLM's fine-tuning to handle more nuanced insurance-specific terminology and queries.
3. Real-Time Data Updates: Implementing real-time data updates from insurance providers to ensure the most current information is always available to users.
4. Multilingual Support: Expanding the system to support multiple languages, thereby reaching a broader audience.
5. Mobile Application: Developing a mobile app to provide users with easier access to the insurance advisor on the go.
6. Advanced User Analytics: Introducing advanced analytics to offer users more personalized suggestions based on their preferences, past searches, and behavior.

These enhancements will allow the system to better serve users and adapt to the evolving insurance market, ensuring continued relevance and effectiveness.

REFERENCES / BIBLIOGRAPHY :

1. **MERN Stack Documentation:** MongoDB, Express.js, React.js, and Node.js stack documentation, explaining the technology and its implementation in modern web applications.
   - URL:https://www.mongodb.com/mern-stack
2. **Retrieval-Augmented Generation (RAG):** Facebook AI research paper explaining the concept of RAG for improving NLP models with external data retrieval.
   - URL: https://arxiv.org/abs/2005.11401
3. **FAISS (Facebook AI Similarity Search):** Documentation for FAISS, the library used for efficient similarity search and clustering of dense vectors.
   - URL: https://github.com/facebookresearch/faiss
4. **Sentence Transformers:** Documentation and tutorials for Sentence Transformers, used for embedding user queries in vector form.
   - URL: https://www.sbert.net
5. **Gemini API for NLP:** API documentation for Gemini, which was used for natural language processing and conversion in the Insurance Advisor system.
   - URL: https://geminiapi.com
6. **React.js Official Documentation:** Official React.js documentation providing insights into building user interfaces for modern web applications.
   - URL: https://reactjs.org
7. **Node.js Official Documentation:** Official documentation for Node.js, detailing its use in backend development and server-side scripting.
   - URL: https://nodejs.org
8. **Web Scraping with BeautifulSoup and Scrapy:** Tutorials and documentation on web scraping using BeautifulSoup and Scrapy, utilized for collecting real-time insurance data.
   - URL: https://docs.scrapy.org/en/latest/
   - URL: https://www.thebeautifulscoop.com/
9. **MongoDB Official Documentation:** MongoDB's documentation on database management, used for storing user data and policy information in the system.
   - URL: https://docs.mongodb.com
10. **Flask Framework Documentation:** Flask official documentation, detailing its use in hosting machine learning models for the backend.
    - URL: https://flask.palletsprojects.com

These references offer comprehensive knowledge of the technologies and concepts used in the Insurance Advisor project.

APPENDIX: TOOLS AND TECHNOLOGY

React: A robust JavaScript library for creating dynamic and interactive user interfaces is called React.js. A component-based design, which is its mainstay, allows developers to create reusable user interface pieces that encompass structure, appearance, and behavior. Smoother performance is achieved because of React's virtual DOM, which refreshes the real DOM efficiently. Developers express the required UI state using declarative programming, while React takes care of the updates. This makes it easier to create complicated applications while keeping readability and order in the code, especially when combined with features like state management and props for data flow. A React application's primary focus is on frontend development, but the ecosystem frequently includes additional components that can encompass backend and styling in CSS.

CSS (Cascading Style Sheets) is a style sheet language that controls the appearance and layout of web pages. It works with HTML to improve the visual presentation of material by using styles including colors, fonts, spacing, and positioning. CSS enables developers to create visually appealing designs while maintaining consistent styling across various pages of a website. CSS allows you to manage responsive designs that adjust to different screen sizes and devices, making it an indispensable tool for developing current, beautiful, and user-friendly web interfaces.

HTML (HyperText Markup Language) is the essential building element of web development, defining the structure of web pages. It is used to describe the content and layout of a web page using a collection of elements and tags. HTML organizes content into headings, paragraphs, lists, links, and other components, allowing browsers to display the information in a consistent manner. HTML guarantees that content is organized and accessible by building the skeletal framework of a web page, laying the groundwork for other technologies to build on.

JavaScript(JS) is a computer language that incorporates interactivity and dynamic behavior into web pages. It allows developers to construct interactive features like forms, animations, and real-time updates, which improve the user experience. JavaScript may leverage the DOM (Document Object Model) to dynamically change content, structure, and style in response to user inputs. It also enables the development of complicated capabilities like data validation, asynchronous data loading, and client-side processing, which improves the interaction and responsiveness of online applications. JavaScript plays an important part in modern web development because to its rich libraries and frameworks, which allow for the production of sophisticated and engaging web applications

React.js: A popular JavaScript library for building user interfaces. A well-liked JavaScript package called React.js is used to create reusable UI components, which are useful for

constructing user interfaces, especially single-page applications. Designed by Facebook, it enables a component-based design that maximizes rendering efficiency by enabling developers to effectively manage and update user interfaces with a virtual DOM. Because of its declarative methodology and strong ecosystem which includes resources like Redux and React Router. React is a great option for building dynamic and interactive web applications. Common programming languages for application logic and backend servers are as follows:

Python: Known for its ease of use and readability, Python is frequently used for server-side scripting and automation, as well as web development with frameworks like Django and Flask.

JavaScript (Node.js): With frameworks like Express.js, Node.js makes it possible to use JavaScript for server-side scripting, which facilitates the creation of scalable network applications.

Frameworks:
Express.js: A web framework for Node.js. A simple and adaptable Node.js web application framework, Express.js makes server-side application development easier. It offers powerful functionality for mobile and online apps, such as effective routing for managing several HTTP protocols and producing dynamic webpages. Requests, answers, and other middle-tier actions can be processed thanks to the middleware functions supported by the framework. Express.js is a great option for creating scalable and maintainable online apps since it also makes it easier to integrate other tools

Machine learning (ML) and natural language processing (NLP) Applications are made more capable through the integration of machine learning (ML) and natural language processing (NLP) technologies, which allow for sophisticated data analysis and wise decision-making. While machine learning (ML) algorithms can detect patterns and make predictions, natural language processing (NLP) makes it easier to interpret and process human language. This helps to create systems that are more intelligent and responsive.

NLP Libraries and Frameworks: NLP libraries and frameworks provide essential tools for processing and understanding human language in a computer-readable format. They encompass a wide range of functionalities including text preprocessing (tokenization, stemming, lemmatization), syntactic analysis (part-of-speech tagging, dependency parsing), semantic analysis (sentiment analysis, named entity recognition, topic modeling), and advanced techniques like machine translation and text generation. These libraries accelerate NLP development by offering pre-trained models, efficient algorithms, and standardized APIs, enabling developers and researchers to focus on building sophisticated language-based applications.

Transformers by Hugging Face: Hugging face provides more Large language model. For leveraging state-of-the-art language models like BERT, GPT-3, etc.

Machine Learning Frameworks: Software platforms known as machine learning frameworks make it easier to create and use machine learning models. They speed up the machine learning lifecycle by offering pre-built algorithms and tools for data preparation, model training, assessment, and deployment. These frameworks save data scientists and engineers from the burden of low-level implementation details and enable them to concentrate on problem-solving and model optimization by abstracting away difficult mathematical computations and offering intuitive interfaces.

TensorFlow: The main applications of the open-source TensorFlow platform are in artificial intelligence and machine learning. It is quite good at creating and training intricate neural networks. Efficient tensor calculations, extensive model training, platform deployment (cloud, mobile, embedded), and a robust tool and library ecosystem are among the salient features. With low-level APIs for fine-grained control and high-level APIs like Keras for quick experimentation, TensorFlow provides versatility. It is extensively used for tasks including time series analysis, natural language processing, and picture recognition.

PyTorch: An open-source machine learning library for Python. Scikit-learn: A library for machine learning in Python.

Vector Search Engines:

FAISS (Facebook Al Similarity Search): For efficient similarity search and clustering of dense vectors. Annoy: Approximate Nearest Neighbors in C++ for large datasets

Gemini 1.5 Api Key: The Gemini API key is a crucial credential for authenticating and securing access to Gemini's services. By integrating this key into your API requests, your application can interact with Gemini's functionalities, such as natural language processing and data retrieval, while ensuring that only authorized users gain access. To obtain a key, you'll need to register your application on Gemini's platform, where you can generate and manage your credentials. It's essential to keep this key confidential to prevent unauthorized access and misuse. Additionally, be mindful of any rate limits or usage quotas associated with the API to ensure smooth and compliant operation