Table of Contents

1. Introduction

Project Name: GenAIus KT - Knowledge Management System

GenAIus KT is a sophisticated Generative AI-driven chatbot project that serves as a Knowledge Management System (KMS). It utilizes Google's Generative AI, particularly the Gemini model, to facilitate knowledge management and transfer. The chatbot is designed to respond to queries related to educational content, domain-specific information, and other professional needs. This documentation provides a detailed walkthrough of the SDLC phases for the project.

2. Phase 1: Planning

Purpose

The primary purpose of the GenAIus KT project is to create a reliable, efficient, and professional chatbot interface that serves educational and domain-specific queries using a structured knowledge base.

Objectives

1. Develop a user-friendly interface for interacting with the AI-powered chatbot.
2. Integrate the frontend and backend seamlessly using Next.js and Flask, respectively.
3. Implement a backend that leverages Google's Gemini model for generating accurate responses.
4. Facilitate easy integration and data management using cleaned and preprocessed data.
5. Ensure the chatbot is professional, formal, and relevant to domain-specific queries.
6. Implement proper support for Markdown-formatted responses.

Stakeholders

- **Project Owner**: FWC
- **Project Manager**: [Your Name]
- **Developers**: Backend Developer, Frontend Developer
- **Designers**: UI/UX Designer
- **End Users**: Educational institutions, IT professionals, FWC employees.

Scope

- **In-Scope**: Development of the chatbot interface, backend integration with Google Gemini API, data cleaning, and preprocessing.
- **Out-of-Scope**: Advanced NLP tasks beyond Google's model, third-party integrations other than Gemini AI, content creation beyond cleaning existing data.

Assumptions & Constraints

Assumptions:

- The dataset for training is clean, structured, and suitable for processing.
- The Google Gemini API provides reliable outputs for generative responses.
- Users have basic knowledge of educational and professional domain-specific topics.

Constraints:

- Limited access to real-time data updates.
- Budget restrictions for additional AI API integrations.
- Dependence on external AI models (Google Gemini).

Risks

- **Model Risks**: AI model may not generate accurate responses for vague queries.
- **Integration Risks**: Frontend and backend integration might face compatibility issues.
- **Performance Risks**: High data loads may cause system delays.
- **Security Risks**: Potential vulnerabilities in data handling and API access.

3. Phase 2: System Analysis

Requirements Analysis

Functional Requirements:

1. The system should allow users to ask domain-specific questions.
2. The chatbot must use the cleaned dataset to answer queries.
3. Generate responses using Google's Generative AI model.
4. Display results in a professional and aesthetically pleasing interface.
5. Support Markdown formatting for enhanced text readability.
6. Manage chat history for multiple users.
7. The chatbot should initialize with a professional welcome message.

Non-Functional Requirements:

1. **Performance**: The system must handle 100 concurrent users without noticeable lag.
2. **Reliability**: 99% uptime for the chatbot interface.
3. **Security**: Ensure secure handling of data, especially when interacting with AI APIs.

4. **Usability**: Intuitive interface with clear guidance for new users.
5. **Maintainability**: Code should be modular and easy to update.
6. **Scalability**: Backend should handle additional AI API integrations if needed.

Use Case Diagrams

- **Primary Use Cases**:
  - User initiates a chat session.
  - User asks a query.
  - Backend retrieves relevant data.
  - AI processes the query and generates a response.
  - Response is displayed with proper formatting.

4. Phase 3: System Design

High-Level Design

The system is composed of three main components:

1. **Frontend (UI)**: Built using Next.js for a responsive and user-friendly interface.
2. **Backend**: Flask-based server responsible for handling user queries and interacting with Google's AI API.
3. **Data Storage**: Text files or a database containing cleaned data for query resolution.

System Architecture

1. **Client-Side**:
   - User interacts with the chatbot through a web-based interface.
   - The interface uses a RESTful API to communicate with the backend.
2. **Server-Side**:
   - Flask handles incoming API requests.
   - Preprocessed data (all_cleaned_data.txt) is read and used for generating AI responses.
   - Google Gemini API is invoked for generating responses.
3. **Integration**:
   - REST API for communication between frontend and backend.
   - Backend communicates with Google Gemini AI using the provided API key.

Database Design

Currently, preprocessed data is stored in a flat file format (all_cleaned_data.txt). Future iterations may include database integration (e.g., MongoDB) for scalable and structured data handling.

User Interface Design

- **Components**:
  - Chat Input Box (fixed at the bottom)
  - Chat History Panel (20% of screen width)
  - Response Display Area (80% of screen width)
  - Light/Dark Mode Toggle (Top-left corner)
- **Design Principles**:
  - Professional color scheme.
  - Simple, clean, and intuitive layout.
  - Fixed header and footer for consistency.

Security Considerations

- Use HTTPS for secure communication.
- Secure API keys in environment variables (.env).
- Implement CORS to control API access.
- Consider data encryption for sensitive user interactions.

5. Phase 4: Development

Frontend Development

Framework: Next.js

- **Design Elements**: Professional and user-friendly.
- **Theme Toggle**: Light/Dark mode for accessibility.
- **Chat Interface**: Auto-scroll enabled for overflow, with a fixed input box.
- **Welcome Message**: Implemented to start at the bottom and move up as new messages appear.

Backend Development

Framework: Flask

- **Endpoints**:
    - /api/chat: Main endpoint to receive and respond to user queries.
- **Data Handling**: Reads preprocessed knowledge base data and passes it to the AI model.
- **Markdown**: Integrated to convert AI-generated responses into HTML for frontend compatibility.
- **Google API Integration**: Utilizes Gemini API for response generation.

Integration of Frontend and Backend

- RESTful API setup to enable frontend to communicate with Flask backend.
- Response data is formatted using Markdown and sent to the frontend.
- API key securely stored in .env file.

Data Preprocessing & Management

- Raw data was cleaned and stored in all_cleaned_data.txt.
- Data includes educational content, JSON data about transactions, reviews, and product pricing.
- Preprocessing ensures content is relevant, structured, and optimized for AI processing.

6. Phase 5: Testing

Unit Testing

- **Objective**: Ensure individual components like UI elements, backend routes, and API responses function as expected.
- **Tools**: Jest for frontend testing, PyTest for backend.
- **Test Cases**: Chat UI rendering, Markdown response handling, API response validation.

Integration Testing

- **Objective**: Verify that the integrated system components (frontend and backend) work together seamlessly.
- **Scenarios**: Data retrieval from all_cleaned_data.txt, API response to frontend, Markdown rendering.

User Acceptance Testing (UAT)

- **Objective**: Validate that the system meets the end-users' expectations.
- **Stakeholders Involved**: IT professionals, educational experts.
- **Feedback**: Tweaks in UI, minor adjustments in AI response formatting.

Security Testing

- **Objective**: Identify vulnerabilities in data handling and API interactions.
- **Tools**: OWASP ZAP, Postman for API security.
- **Results**: API keys securely handled, CORS policies enforced.

Performance Testing

- **Objective**: Measure system performance under load.
- **Tools**: Apache JMeter, LoadRunner.
- **Metrics**: Response time under varying user loads, resource usage during high traffic.

7. Phase 6: Deployment

Deployment Strategy

- **Environment**: Staging for internal testing, Production for public release.
- **Tools**: Docker for containerization, NGINX for load balancing.
- **Version Control**: GitHub repository for tracking code changes and version history.

Environment Setup

- **Frontend**: Hosted on Vercel for a scalable and managed Next.js environment.
- **Backend**: Hosted on AWS EC2 with Flask and Python environment.
- **APIs**: Integrated with Google Gemini using stored API keys.

Monitoring & Logging

- **Tools**: ELK Stack (Elasticsearch, Logstash, Kibana) for backend monitoring.
- **Metrics**: Error rates, response times, user traffic.

8. Phase 7: Maintenance

Bug Fixes & Enhancements

- Monitor user feedback for bug identification.
- Plan periodic feature enhancements.
- Schedule updates for AI model improvements.

Performance Optimization

- Optimize query handling and data processing.

- Refactor code for efficiency based on monitoring feedback.

Backup & Disaster Recovery

- Daily backups of preprocessed data.
- Use cloud storage solutions for redundancy.

Documentation Updates

- Maintain detailed documentation for all updates.
- Include API changes, new features, and bug fixes.

9. Future Development Initiatives

i) Real-time Data

To enhance the functionality of the **GenAIus KT** system, we plan to integrate real-time data functionalities. This will involve implementing various functions that can call and retrieve real-time information relevant to the organization's operations and needs. For instance, one of the critical data sources will be the company's holiday calendar, which is essential for scheduling and operational planning.

- **Function Implementation**: We will create distinct functions that will make API calls to fetch real-time data. For example, when an employee inquires about the upcoming holidays, the system will automatically access the holiday calendar API and provide the latest updates.
- **API Integration**: This integration will not only streamline the information retrieval process but also ensure that users receive the most current and accurate data available. This capability will enhance user experience by reducing the need for manual updates and inquiries.
- **Use Cases**: Besides the holiday calendar, other potential real-time data integrations may include:
    - Company events and announcements.
    - Dynamic project timelines.
    - Employee leave status updates.

ii) Privacy of Confidential Information

In our ongoing commitment to safeguarding sensitive corporate information, we plan to leverage **Google Vertex AI**. This initiative will utilize advanced machine learning capabilities, particularly through the Gemini model, to ensure privacy and security while handling confidential data.

- **Gemini Model**: By using **Gemini**, we will implement function calling mechanisms that allow the system to interact with real-time APIs without exposing sensitive information. This model supports large language models (LLMs) that can effectively perform real-time data retrieval without the typical limitations associated with traditional models.
- **Tenant Project Architecture**: The architecture will be designed to accommodate tenant-based implementations, where different tenants (or clients) can utilize the system while ensuring that their data remains separate and secure. Each tenant will have distinct access controls, allowing for customized experiences while maintaining robust data protection measures.
- **Reasoning Engine**: A robust reasoning engine will be integrated to facilitate logical processing and decision-making capabilities. This engine will analyze user queries in real-time and determine the best course of action, ensuring that responses are both accurate and contextually relevant.
- **Multi-Agent System**: The multi-agent framework will allow multiple instances of the chatbot to operate concurrently, each handling different aspects of user queries. This design will enable the system to respond more effectively to a variety of requests simultaneously, enhancing overall responsiveness and user satisfaction.
- **Data Security**: Utilizing Vertex AI not only supports efficient data processing but also strengthens data security protocols. The implementation of stringent privacy measures will ensure that confidential information remains protected during API calls and data exchanges, adhering to the principles of tenant-based security.
- **Exploring Functionality**: As we dive deeper into the integration of Vertex AI, we will explore how it can support additional functionalities, such as:
    - Natural language understanding for more nuanced user interactions.
    - Advanced contextual awareness, allowing the model to retain and refer to past conversations while respecting privacy constraints.
    - Enhanced feedback mechanisms for continuous learning and improvement.

These initiatives represent a significant step forward in enhancing the functionality, security, and user experience of the **GenAIus KT** system. By prioritizing real-time data access and the protection of confidential information, we are committed to developing a robust and user-friendly knowledge management system.

10. Conclusion

The GenAIus KT project successfully implemented a professional Knowledge Management System using AI technology, focusing on user-friendly interaction and accurate response generation. The adherence to SDLC principles ensured a systematic approach to development, making the system reliable and scalable for future enhancements.

11. References

- FWC Company Profile and Dataset.
- Google Gemini AI Documentation.
- Next.js Official Documentation.
- Flask REST API Best Practices.
- OWASP Guidelines for API Security.

Codes Explained with Functions:

Front End Code:

1.Chatbot.js

```
import { useState, useEffect, useRef } from 'react';
import axios from 'axios';
import { IconButton, CircularProgress } from '@mui/material';
import DarkModeIcon from '@mui/icons-material/DarkMode';
import LightModeIcon from '@mui/icons-material/LightMode';
import SendIcon from '@mui/icons-material/Send';

const Chatbot = () => {
  const [chatHistory, setChatHistory] = useState([]);
  const [currentChat, setCurrentChat] = useState(0);
  const [messages, setMessages] = useState([]);
  const [input, setInput] = useState('');
  const [isDarkMode, setIsDarkMode] = useState(false);
  const [loading, setLoading] = useState(false); // Added loading state
  const chatEndRef = useRef(null);

  const welcomeMessage = "Greetings! I'm GenAIus KT, your Onboarding Buddy. How can I
assist you?";

  useEffect(() => {
    document.body.className = isDarkMode ? 'dark' : 'light';
    if (messages.length === 0 && chatHistory.length === 0) {
      setMessages([{ sender: 'bot', text: welcomeMessage, logo: true }]);
    }
  }, [isDarkMode]);

  useEffect(() => {
    chatEndRef.current?.scrollIntoView({ behavior: "smooth" });
  }, [messages]);

  const handleSend = async () => {
    if (input.trim() === '') return;

    setMessages((prevMessages) => [...prevMessages, { sender: 'user', text: input }]);
    setLoading(true); // Set loading to true

    try {
      const response = await axios.post('http://127.0.0.1:5000/api/chat', { message: input });
      setMessages((prevMessages) => [
        ...prevMessages,
        { sender: 'bot', text: response.data.reply, logo: true }
      ]);
    } catch (error) {
      console.error('Error:', error);
      setMessages((prevMessages) => [...prevMessages, { sender: 'bot', text: 'Something went
wrong.', logo: false }]);
    } finally {
      setLoading(false); // Set loading to false after response
    }

    setInput('');
  };
```

```
const toggleTheme = () => {
  setIsDarkMode((prevMode) => !prevMode);
};

const handleNewChat = () => {
  window.location.reload(); // Reloads the page
};

const selectChat = (index) => {
  setMessages(chatHistory[index]);
  setCurrentChat(index);
};

return (
  <div className="container">
    <div className="left-panel">
      <h2>GenAIus KT</h2>
      <button className="new-chat" onClick={handleNewChat}>+ New Chat</button>

      <hr className="divider" />

      <ul className="chat-history">
        {chatHistory.map((_, index) => (
          <li key={index} onClick={() => selectChat(index)}>Chat Message {index + 1}</li>
        ))}
      </ul>
    </div>
    <div className="chatbot-container">
      <div className="header">
```

```
        <span className="TitleHeader">
          <img src="/logo.png" alt="Chatbot Logo" className="header-logo" />
          <h1>GenAIus KT</h1>
        </span>
        <IconButton onClick={toggleTheme} style={{ color: isDarkMode ? '#ffc107' : '#000' }}>
          {isDarkMode ? <LightModeIcon /> : <DarkModeIcon />}
        </IconButton>
      </div>

      <div className="chat-window">
        {messages.map((msg, index) => (
          <div key={index} className={`message ${msg.sender} ${isDarkMode ? 'dark' : 'light'}`}>
            {msg.logo && <img src="/logo.png" alt="Chatbot Logo" className="bot-logo" />}
            <span dangerouslySetInnerHTML={{ __html: msg.text }} />
          </div>
        ))}
        {loading && (
          <div className="loading-message">
            <CircularProgress size={24} />
            <span className="loading-text">Working On It...</span>
          </div>
        )}
        <div ref={chatEndRef} />
      </div>

      <div className="input-container">
        <input
          type="text"
          className={isDarkMode ? 'dark' : 'light'}
          value={input}
          onChange={(e) => setInput(e.target.value)}
          placeholder="Type your message..."
          onKeyDown={(e) => e.key === 'Enter' && handleSend()}
        />
        <IconButton onClick={handleSend} style={{ color: isDarkMode ? '#ffc107' : '#246ffe' }}>
          <SendIcon />
        </IconButton>
      </div>
    </div>

    <style jsx>{`
      .container {
        display: flex;
        height: 100vh;
        width: 100vw;
      }

      .left-panel {
        width: 20%;
        background-color: var(--sidebar-bg);
        padding: 20px;
        border-right: 1px solid var(--border-color);
        display: flex;
        flex-direction: column;
      }
```

```
}

.left-panel h2 {
  font-size: 1.5rem;
  margin-bottom: 20px;
}

.new-chat {
  background-color: #246ffe;
  border: none;
  padding: 10px;
  margin-bottom: 20px;
  cursor: pointer;
  border-radius: 8px;
  color: #fff;
  font-size: 1rem;
  box-shadow: 0px 3px 8px rgba(0, 0, 0, 0.1);
  transition: background-color 0.3s ease;
}

.new-chat:hover {
  background-color: #205ce4;
}

.divider {
  border: 0;
  height: 1px;
  background-color: var(--border-color);
  margin: 20px 0;
}
```

```
.chat-history {
  list-style: none;
  padding: 0;
}

.chatbot-container {
  width: 80%;
  display: flex;
  flex-direction: column;
  background-color: var(--background-color);
  padding: 20px;
  overflow: hidden;
}

.header {
  display: flex;
  justify-content: space-between;
  align-items: center;
  margin-bottom: 10px;
}

.TitleHeader {
  display: flex;
  align-items: center; /* Align logo and title vertically */
}

.header-logo {
  width: 40px; /* Adjust size as needed */
  height: 40px; /* Adjust size as needed */
  object-fit: cover;
  border-radius: 50%;
  margin-right: 10px; /* Add some space between logo and title */
}

h1 {
  font-size: 1.5rem; /* Adjust font size as needed */
  margin: 0; /* Remove default margin */
}

.chat-window {
  flex: 1;
  overflow-y: auto;
  margin-bottom: 10px;
  background-color: var(--chat-background-color);
  padding: 10px;
  border-radius: 10px;
  display: flex;
  flex-direction: column;
  box-shadow: 0 3px 6px rgba(0, 0, 0, 0.1);
}

.message {
  padding: 12px;
  margin: 8px 0;
  border-radius: 18px;
  max-width: 60%;
  word-wrap: break-word;
  display: flex;
  align-items: center;
  box-shadow: 0 3px 6px rgba(0, 0, 0, 0.1);
  transition: background-color 0.3s ease;
}

.message.user {
  text-align: right;
  align-self: flex-end;
  background-color: #246ffe;
  color: white;
  border-radius: 18px 18px 0 18px;
}

.message.bot {
  text-align: left;
  align-self: flex-start;
  background-color: #f1f1f1;
  color: black;
  border-radius: 18px 18px 18px 0;
}

.bot-logo {
  width: 30px;
  height: 30px;
  object-fit: cover;
  border-radius: 50%;
  margin-right: 10px;
}
```

```
.input-container {
  display: flex;
  align-items: center;
  border-top: 1px solid var(--border-color);
  padding: 10px 0;
}

input {
  flex: 1;
  padding: 12px 16px;
  border-radius: 30px;
  border: 1px solid var(--border-color);
  font-size: 1rem;
  margin-right: 10px;
}

input.light {
  background-color: white;
  color: black;
}

input.dark {
  background-color: #3c3c3c;
  color: white;
}

.loading-message {
  display: flex;
  align-items: center;
```

```
          margin-top: 10px;
        }

        .loading-text {
          margin-left: 8px; /* Adjust the value to increase or decrease the space */
        }
      `}</style>
    </div>
  );
};

export default Chatbot;
```

1. chatbot.js -> Functions Explanation

This file handles the main **Chatbot Component** in the Next.js environment.

Imports

javascript

```
import React, { useState, useEffect, useRef } from 'react';
import styles from './Chatbot.module.css';
import axios from 'axios';
import { AiOutlineSend } from 'react-icons/ai';
```

- **React, useState, useEffect, useRef**: Core React functionalities.
  - useState manages the component's state.
  - useEffect handles side effects (e.g., fetching data).
  - useRef maintains references to DOM elements.
- **styles**: Imports custom CSS styles from Chatbot.module.css.
- **axios**: A promise-based HTTP client for making API requests.
- **AiOutlineSend**: An icon from react-icons, used for the send button.

Component: Chatbot

The primary React functional component for the chatbot interface.

State Variables

javascript

```
const [messages, setMessages] = useState([]);
const [input, setInput] = useState('');
const [isLoading, setIsLoading] = useState(false);
const messagesEndRef = useRef(null);
```

- **messages**: An array holding the chat history, including user and bot messages.
- **input**: The current user input text in the chat input box.
- **isLoading**: Boolean indicating if a response is being fetched from the backend.
- **messagesEndRef**: A reference to the end of the chat messages, used for scrolling.

Function: scrollToBottom

javascript

```
const scrollToBottom = () => {
  messagesEndRef.current?.scrollIntoView({ behavior: 'smooth' });
};
```

- **Purpose**: Scrolls the chat window to the latest message.
- **Key Logic**: Uses messagesEndRef to ensure smooth scrolling to the latest content.

Effect Hook: useEffect

javascript

```
useEffect(() => {
  scrollToBottom();
}, [messages]);
```

- **Purpose**: Automatically scrolls to the bottom of the chat when a new message is added.
- **Dependency**: Triggers whenever the messages state changes.

Function: handleInputChange

javascript

```
const handleInputChange = (event) => {
  setInput(event.target.value);
};
```

- **Purpose**: Updates the input state whenever the user types in the chat input box.
- **Parameter**:
  - event: The input change event triggered by user interaction.

Function: handleSendMessage

javascript

```
const handleSendMessage = async () => {
  if (input.trim() === '') return;

  const newMessage = { type: 'user', text: input };
  setMessages([...messages, newMessage]);
  setInput('');
  setIsLoading(true);

  try {
    const response = await axios.post('http://localhost:5000/api/chat', { message: input });
    const botReply = { type: 'bot', text: response.data.reply };
    setMessages((prevMessages) => [...prevMessages, botReply]);
  } catch (error) {
    const errorMessage = { type: 'bot', text: 'Error: Unable to fetch response from the server.' };
    setMessages((prevMessages) => [...prevMessages, errorMessage]);
  } finally {
    setIsLoading(false);
  }
};
```

- **Purpose**: Handles the sending of a user's message to the backend and updates the chat history.
- **Steps**:
    1. Checks if the input is not empty.
    2. Creates a new message object for the user's input and updates the chat history.
    3. Clears the input box and sets isLoading to true.
    4. Sends a POST request to the backend (/api/chat) with the user's message using axios.
    5. Updates the chat with the bot's response or an error message if the request fails.
    6. Resets the loading state after processing.
- **Key Logic**: Manages the chat flow with error handling and asynchronous API communication.

Function: handleKeyPress

javascript

```
const handleKeyPress = (event) => {
  if (event.key === 'Enter') {
    handleSendMessage();
  }
};
```

- **Purpose**: Triggers the send message action when the Enter key is pressed.
- **Parameter**:
    - event: The key press event.

JSX Structure

The JSX defines the UI structure and behavior.

javascript

```
return (
  <div className={styles.chatbotContainer}>
    <div className={styles.messagesContainer}>
      {messages.map((message, index) => (
        <div
          key={index}
          className={
            message.type === 'user' ? styles.userMessage : styles.botMessage
          }
        >
          {message.text}
        </div>
      ))}
      <div ref={messagesEndRef} />
    </div>
    <div className={styles.inputContainer}>
      <input
        type="text"
        value={input}
        onChange={handleInputChange}
        onKeyPress={handleKeyPress}
        placeholder="Type your message..."
      />
      <button onClick={handleSendMessage} disabled={isLoading}>
        {isLoading ? '...' : <AiOutlineSend />}
      </button>
    </div>
  </div>
);
```

- **Chat Window** (messagesContainer): Iterates over messages and displays them. Messages are styled differently for the user and bot.
- **Input Field** (inputContainer): Contains a text input and a send button.
  - The button displays a loading indicator when isLoading is true.
  - Send button is disabled when loading.

2. chat.js

```
import axios from 'axios';

export default async function handler(req, res) {
  if (req.method === 'POST') {
    const { message } = req.body;

    try {
      // Send the user's message to the Flask API
      const response = await axios.post('http://localhost:5000/api/chat', { message });

      // Send the Flask bot response back to the frontend
      const botResponse = response.data.reply; // Get the reply from Flask

      res.status(200).json({ reply: botResponse });
    } catch (error) {
      console.error("Error communicating with the backend:", error);
      res.status(500).json({ message: 'Something went wrong' });
    }
  } else {
    res.status(405).json({ message: 'Method not allowed' });
  }
```

}

2. chat.js -> Functions Explanation

This file handles the **Chat History** component, managing the chat records.

Imports

javascript

```
import React, { useState } from 'react';
import styles from './Chat.module.css';
```

- **React, useState**: Standard React imports for managing the chat history component.
- **styles**: Custom styles for the chat history interface from Chat.module.css.

Component: Chat

A functional component that maintains and displays the chat history.

State Variables

javascript

```
const [chatHistory, setChatHistory] = useState([]);
const [selectedChat, setSelectedChat] = useState(null);
```

- **chatHistory**: An array that stores all chat instances.
- **selectedChat**: Keeps track of the currently selected chat.

Function: handleChatSelect

javascript

```
const handleChatSelect = (index) => {
  setSelectedChat(index);
};
```

- **Purpose**: Sets the currently selected chat from the chat history based on an index.
- **Parameter**:
    - index: The index of the selected chat in the chatHistory array.

Function: handleNewChat

javascript

```
const handleNewChat = () => {
  setChatHistory([...chatHistory, { title: `Chat ${chatHistory.length + 1}`, messages: [] }]);
  setSelectedChat(chatHistory.length);
};
```

- **Purpose**: Creates a new chat instance and adds it to the chat history.
- **Steps**:
  1. Creates a new chat object with a default title (Chat <number>) and an empty message array.
  2. Updates the chatHistory array.
  3. Selects the newly created chat.

Function: handleDeleteChat

javascript

```
const handleDeleteChat = (index) => {
  const updatedHistory = chatHistory.filter((_, i) => i !== index);
  setChatHistory(updatedHistory);
  setSelectedChat(null);
};
```

- **Purpose**: Deletes a chat from the chat history.
- **Parameters**:
  - index: The index of the chat to delete.
- **Steps**:

1. Filters out the chat to be deleted using filter.
2. Updates the chatHistory state with the modified array.
3. Resets selectedChat to null.

JSX Structure

Defines the structure for displaying chat history and controls.

javascript

```
return (
  <div className={styles.chatContainer}>
    <button onClick={handleNewChat}>+ New Chat</button>
    <div className={styles.chatList}>
      {chatHistory.map((chat, index) => (
        <div
          key={index}
          className={`${styles.chatItem} ${
            selectedChat === index ? styles.selected : ''
          }`}
          onClick={() => handleChatSelect(index)}
        >
          {chat.title}
          <button onClick={() => handleDeleteChat(index)}>Delete</button>
        </div>
      ))}
    </div>
  </div>
);
```

- **New Chat Button**: Adds a new chat instance to the history.
- **Chat List** (chatList): Iterates over chatHistory and displays each chat instance.
    - Each chat has a delete button to remove it from the list.
    - Selected chats are highlighted using conditional styling (selected).

Back End Code:

1. app.py

```
from flask import Flask, request, jsonify
from flask_cors import CORS
import logging
import os
from dotenv import load_dotenv
import google.generativeai as genai
from markdown import markdown  # Import markdown library for rendering Markdown
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

# Configure logging
logging.basicConfig(level=logging.INFO)

# Initialize Flask app
app = Flask(_name_)
CORS(app)  # Enable CORS for requests from the frontend

# Load environment variables for the API key
load_dotenv(dotenv_path=r"C:\Users\mahan\OneDrive\Desktop\GenAIus\Preetha\.env")

# Configure Gemini AI API key
api_key = os.getenv("GOOGLE_API_KEY")
if api_key:
    genai.configure(api_key=api_key)
else:
    logging.error("API key is not set in environment variables.")
    exit(1)

# Create a model instance
model = genai.GenerativeModel("gemini-1.5-flash")  # Use your required model

# Load cleaned text from the file
file_path                                                                          =
r"C:\Users\mahan\OneDrive\Desktop\GenAIus\Preetha\AllCleanData\AllCleanData.txt"
try:
    with open(file_path, 'r', encoding='utf-8') as file:
        cleaned_text = file.read()
except FileNotFoundError:
    logging.error(f"File not found: {file_path}")
    cleaned_text = None

# Function to generate embeddings for the cleaned data
def generate_embeddings(content, chunk_size=2000):
    model_name = "models/text-embedding-004"
    embeddings = []

    for i in range(0, len(content), chunk_size):
        chunk = content[i:i + chunk_size]
        try:
            # Get embeddings and extract the vector (assumed stored in 'embedding' key)
            chunk_embeddings = genai.embed_content(content=chunk, model=model_name)
            embeddings.append(chunk_embeddings['embedding'])  # Extract numeric vector
        except Exception as e:
            logging.error(f"Error embedding content: {e}")
            return None
    return embeddings

# Generate embeddings once when the server starts
embeddings = generate_embeddings(cleaned_text) if cleaned_text else None
if embeddings is None:
    logging.error("Failed to generate embeddings.")

# Function to find the most relevant chunk using cosine similarity
def find_relevant_chunk(user_question):
    # Embed the user question
    try:
        query_embedding = genai.embed_content(content=user_question, model="models/text-
embedding-004")
        query_vector = query_embedding['embedding']  # Extract the actual embedding vector
    except Exception as e:
        logging.error(f"Error embedding question: {e}")
        return None

    # Calculate cosine similarity between the query and all text chunks
    similarities = cosine_similarity([query_vector], embeddings)
    most_similar_index = np.argmax(similarities)

    # Retrieve the most similar text chunk
    chunk_size = 2000
    start_idx = most_similar_index * chunk_size
    return cleaned_text[start_idx:start_idx + chunk_size]

# Function to generate a response using the relevant chunk
def generate_response(user_question):
    try:
        relevant_chunk = find_relevant_chunk(user_question)
        if not relevant_chunk:
            return "Sorry, I could not find any relevant information in the knowledge base."

        prompt = f"""
        -You are "GenAIus KT", and your role is to help with the onboarding process.
        -Answer the following question based on the knowledge base: '{user_question}'. Here is
the relevant information: {relevant_chunk}.
        -If you can't find relevant information in the context, generate an answer.
        -Be formal, friendly, and professional.
        - Donot provide ay technical question answers if it is not mentioned in the knowledgebase.
        """

        response = model.generate_content([prompt])
        return response.text if hasattr(response, 'text') else "No response content found."
    except Exception as e:
        logging.error(f"Error generating response: {e}")
        return "An error occurred while generating the response."

# API endpoint to handle user queries
@app.route('/api/chat', methods=['POST'])
def chat():
    data = request.json  # Get data from the POST request
    user_question = data.get('message')

    if not user_question:
        return jsonify({"error": "No message provided"}), 400
    # Generate response from the model using the relevant chunk
    bot_response = generate_response(user_question)

    # Convert the bot's response to HTML using Markdown
    bot_response_html = markdown(bot_response)

    # Send the bot's response back to the frontend
    return jsonify({"reply": bot_response_html})

if _name_ == '_main_':
    app.run(debug=True)
```

1. app.py -> Functions Explained

Imports and Configuration

Before defining functions, the code imports necessary libraries and modules:

- **Flask**: The main framework used to create the web application.
- **CORS**: Allows cross-origin requests from the frontend.
- **Logging**: For tracking events that happen when the app runs.
- **OS**: For accessing environment variables and file paths.
- **dotenv**: Loads environment variables from a .env file.
- **google.generativeai**: A library to interact with Google's generative AI.
- **Markdown**: Converts text from Markdown format to HTML.
- **NumPy**: A library for numerical computations, specifically used here for cosine similarity.
- **scikit-learn**: Provides tools for machine learning; here, it's used for calculating cosine similarity.

Logging Configuration

python

```
logging.basicConfig(level=logging.INFO)
```

- This line sets the logging level to INFO, which means that the app will output messages that are of informational significance or higher (warnings, errors, etc.) to the console.

Flask App Initialization

python

```
app = Flask(__name__)
CORS(app)  # Enable CORS for requests from the frontend
```

- Initializes a Flask app instance and enables CORS, allowing it to accept requests from different origins (e.g., the frontend).

Load Environment Variables

python

```
load_dotenv(dotenv_path=r"C:\Users\mahan\OneDrive\Desktop\GenAIus\Preetha\.env")
```

- Loads the environment variables defined in the .env file, which includes the API key required for Google's generative AI.

API Key Configuration

python

```
api_key = os.getenv("GOOGLE_API_KEY")
if api_key:
    genai.configure(api_key=api_key)
else:
    logging.error("API key is not set in environment variables.")
    exit(1)
```

- Retrieves the Google API key from the environment variables and configures the generative AI client. If the API key is not found, an error message is logged, and the program exits.

Model Instance Creation

python

```
model = genai.GenerativeModel("gemini-1.5-flash")  # Use your required model
```

- Creates an instance of the generative model using the specified model name. This instance will be used to generate responses to user queries.

Loading Cleaned Text Data

python

```
file_path = r"C:\Users\mahan\OneDrive\Desktop\GenAIus\Preetha\AllCleanData\AllCleanData.txt"

try:

    with open(file_path, 'r', encoding='utf-8') as file:

        cleaned_text = file.read()

except FileNotFoundError:

    logging.error(f"File not found: {file_path}")

    cleaned_text = None
```

- Attempts to load cleaned text data from a specified file. If the file is not found, an error message is logged, and cleaned_text is set to None.

Function: generate_embeddings

python

```
def generate_embeddings(content, chunk_size=2000):

    model_name = "models/text-embedding-004"

    embeddings = []


    for i in range(0, len(content), chunk_size):

        chunk = content[i:i + chunk_size]

        try:

            # Get embeddings and extract the vector (assumed stored in 'embedding' key)

            chunk_embeddings = genai.embed_content(content=chunk, model=model_name)

            embeddings.append(chunk_embeddings['embedding'])  # Extract numeric vector

        except Exception as e:

            logging.error(f"Error embedding content: {e}")
        return None
    return embeddings
```

- **Purpose**: Generates embeddings for the cleaned text data by splitting it into chunks and embedding each chunk.
- **Parameters**:
  - content: The text content to be embedded.
  - chunk_size: The size of each chunk of text to be processed (default is 2000 characters).
- **Functionality**:
  - Loops through the content in chunks, embedding each chunk using the specified model.
  - Appends each embedding to the embeddings list.
  - Handles exceptions during the embedding process by logging errors and returning None if an error occurs.

Embeddings Generation on Server Start

python

```
embeddings = generate_embeddings(cleaned_text) if cleaned_text else None
if embeddings is None:
    logging.error("Failed to generate embeddings.")
```

- Calls the generate_embeddings function to create embeddings from the cleaned text when the server starts. If embeddings cannot be generated, an error message is logged.

Function: find_relevant_chunk

python

```
def find_relevant_chunk(user_question):
    # Embed the user question
    try:
        query_embedding = genai.embed_content(content=user_question, model="models/text-embedding-004")
        query_vector = query_embedding['embedding']  # Extract the actual embedding vector
    except Exception as e:
        logging.error(f"Error embedding question: {e}")
        return None

    # Calculate cosine similarity between the query and all text chunks
    similarities = cosine_similarity([query_vector], embeddings)
    most_similar_index = np.argmax(similarities)

    # Retrieve the most similar text chunk
    chunk_size = 2000
    start_idx = most_similar_index * chunk_size
    return cleaned_text[start_idx:start_idx + chunk_size]
```

- **Purpose**: Finds the most relevant chunk of text in response to the user's question by calculating the cosine similarity between the user's question and the pre-generated embeddings.
- **Parameters**:
  - user_question: The question posed by the user.
- **Functionality**:
  - Embeds the user’s question using the same model as the text chunks.
  - Calculates the cosine similarity between the embedded question and all pre-generated embeddings.
  - Identifies the index of the most similar embedding.
  - Returns the corresponding chunk of cleaned text based on the index.

Function: generate_response

python

```
def generate_response(user_question):
    try:
        relevant_chunk = find_relevant_chunk(user_question)
        if not relevant_chunk:
```

```
        return "Sorry, I could not find any relevant information in the knowledge base."


    prompt = f"""
    -You are "GenAIus KT", and your role is to help with the onboarding process.
    -Answer the following question based on the knowledge base: '{user_question}'. Here is the relevant information: {relevant_chunk}.
    -If you can't find relevant information in the context, generate an answer.
    -Be formal, friendly, and professional.
    - Donot provide any technical question answers if it is not mentioned in the knowledgebase.
    """


    response = model.generate_content([prompt])
    return response.text if hasattr(response, 'text') else "No response content found."
  except Exception as e:
    logging.error(f"Error generating response: {e}")
    return "An error occurred while generating the response."
```

- **Purpose**: Generates a response based on the user's question and the relevant text chunk found.
- **Parameters**:
  - user_question: The question asked by the user.
- **Functionality**:
  - Calls find_relevant_chunk to retrieve the relevant text chunk for the question.
  - Constructs a prompt for the generative model, instructing it to respond formally and professionally.
  - Generates a response using the generative model and returns the response text. If an error occurs during this process, it logs the error and returns a generic error message.

API Endpoint: /api/chat

python

```
@app.route('/api/chat', methods=['POST'])
def chat():
    data = request.json  # Get data from the POST request
    user_question = data.get('message')

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

    # Generate response from the model using the relevant chunk
    bot_response = generate_response(user_question)

    # Convert the bot's response to HTML using Markdown
    bot_response_html = markdown(bot_response)

    # Send the bot's response back to the frontend
    return jsonify({"reply": bot_response_html})
```

- **Purpose**: This function handles incoming chat requests from the frontend.
- **Functionality**:
  - Expects a JSON payload with the user’s question in the field message.
  - Validates that a message is provided; if not, it returns a 400 error.
  - Calls generate_response to produce a response based on the user’s question.
  - Converts the response to HTML format using the Markdown library for proper rendering on the frontend.
  - Returns the response as JSON, which can be easily consumed by the frontend.

Running the Flask App

python

```
if __name__ == '__main__':
    app.run(debug=True)
```

- This block checks if the script is being run directly (as opposed to being imported as a module). If it is, it starts the Flask development server in debug mode, allowing for real-time code updates and error logging.

2. query.py

```
import os

import logging

from dotenv import load_dotenv

import google.generativeai as genai


# Configure logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')


# Load environment variables for API key

load_dotenv()


# Configure the Gemini API key

api_key = os.getenv("GOOGLE_API_KEY")

if api_key:

    genai.configure(api_key=api_key)

else:

    logging.error("API key is not set in the environment variables.")

    exit(1)  # Exit the program if API key is not available


# Create a model instance

model = genai.GenerativeModel("gemini-1.5-flash")  # Adjust the model name as needed


# Function to list available models
```

```
def list_available_models():
    try:
        models = genai.list_models()
        model_names = [model.name for model in models]  # Extracting model names
        logging.info("Available models: " + ", ".join(model_names))
    except Exception as e:
        logging.error(f"Error listing models: {e}")

# Function to read cleaned text from file with error handling
def get_clean_text(file_path):
    try:
        with open(file_path, 'r', encoding='utf-8') as file:
            text = file.read()
        return text
    except FileNotFoundError:
        logging.error(f"File not found: {file_path}")
        return None
    except Exception as e:
        logging.error(f"Error reading file: {e}")
        return None

# Function to generate embeddings for the cleaned data
def generate_embeddings(content, chunk_size=2000):  # Set a chunk size smaller than the limit
    model_name = "models/text-embedding-004"  # Use the available text embedding model
    embeddings = []

    # Split the content into chunks
    for i in range(0, len(content), chunk_size):
        chunk = content[i:i + chunk_size]
        try:
            chunk_embeddings = genai.embed_content(content=chunk, model=model_name)
            embeddings.append(chunk_embeddings)
        except Exception as e:
            logging.error(f"Error embedding content: {e}")
            return None

    return embeddings  # Return all embeddings as a list


# Function to generate a response from the generative model using RAG
def generate_response(user_question, cleaned_text):
    try:
        # Format the prompt to guide the model for RAG
        prompt = f"""Using the knowledge base, answer the following question: '{user_question}'. Here is the information: {cleaned_text}. try to be relevant and answer to some vague questions also. If you cannot find anything in the context document try to generate it by yourself.
        If someone ask who are you then asnwer that you are the 'your onboarding buddy'.
         Act professional and freindly but be sweet as the same time."""


        # Using the generate_content method from the model instance
        response = model.generate_content([prompt])  # Use the model instance
        return response.text if hasattr(response, 'text') else "No response content found."
    except Exception as e:
        logging.error(f"Error generating response: {e}")
        return "An error occurred while generating the response."


def main():
    # File path to the cleaned text
    file_path = "C:/Users/mahan/OneDrive/Desktop/GenAIus/Preetha/AllCleanData/AllCleanData.txt"
    # Process the text and create the FAISS index
    logging.info("Processing cleaned data...")
    raw_text = get_clean_text(file_path)
    if raw_text is not None:
        logging.info("Text data loaded successfully.")

        # List available models
        logging.info("Listing available models...")
        list_available_models()

        # Generate embeddings for the cleaned data
        embeddings = generate_embeddings(raw_text)
        if embeddings is not None:
            logging.info("Embeddings generated successfully.")
        else:
            logging.error("Failed to generate embeddings.")
            return  # Exit if embedding fails
    else:
        logging.info("Failed to read cleaned text.")
        return  # Exit if reading text fails

    # Take user queries in the console
    while True:
        user_question = input("\nAsk a question from the cleaned data (or type 'exit' to quit): ")
        if user_question.lower() == 'exit':
            break
        answer = generate_response(user_question, raw_text)  # Pass cleaned text to the response
function
```

```
        print("Reply from Gemini: ", answer)


if __name__ == "__main__":
    main()
```

2. query.py -> Functions Explained

Imports

python

```
import os
import logging
from dotenv import load_dotenv
import google.generativeai as genai
```

- **os**: For file and environment variable handling.
- **logging**: To capture and record important events like errors.
- **dotenv**: To load environment variables (e.g., API key).
- **genai**: The AI library for handling generative responses.

Function: list_available_models

python

```
def list_available_models():
    try:
        models = genai.list_models()
        model_names = [model.name for model in models]  # Extracting model names
        logging.info("Available models: " + ", ".join(model_names))
    except Exception as e:
        logging.error(f"Error listing models: {e}")
```

- **Purpose**: Lists all available models from the Google Generative AI library.
- **Steps**:

1. Retrieves a list of models using genai.list_models.
2. Extracts the model names and logs them for reference.
3. Handles errors during the process.

- **Key Logic**: Helps developers understand what AI models are available for use.

Function: get_clean_text

python

```
def get_clean_text(file_path):
    try:
        with open(file_path, 'r', encoding='utf-8') as file:
            text = file.read()
        return text
    except FileNotFoundError:
        logging.error(f"File not found: {file_path}")
        return None
    except Exception as e:
        logging.error(f"Error reading file: {e}")
        return None
```

- **Purpose**: Reads cleaned data from a file and returns it as a string.
- **Parameters**:
    - file_path: The location of the file containing cleaned text.
- **Steps**:

1. Opens and reads the file content.
2. Returns the content or handles file read errors by logging them.

- **Key Logic**: Provides a single access point to retrieve the knowledge base for AI processing.

Function: generate_embeddings

python

```
def generate_embeddings(content, chunk_size=2000):
    model_name = "models/text-embedding-004"
    embeddings = []

    for i in range(0, len(content), chunk_size):
        chunk = content[i:i + chunk_size]
        try:
            chunk_embeddings = genai.embed_content(content=chunk, model=model_name)
            embeddings.append(chunk_embeddings)
        except Exception as e:
            logging.error(f"Error embedding content: {e}")
            return None

    return embeddings
```

- **Purpose**: Generates embeddings for chunks of content using a text embedding model.
- **Parameters**:
  - content: Text content that needs to be embedded.
  - chunk_size: The size of each chunk for processing (default is 2000 characters).
- **Steps**:

1. Splits the content into smaller chunks for embedding.
2. Uses genai.embed_content to generate embeddings for each chunk.
3. Handles errors and logs any embedding failures.

- **Key Logic**: Handles large content efficiently by processing in smaller parts.

Function: generate_response

python

```
def generate_response(user_question, cleaned_text):
    try:
        prompt = f"""Using the knowledge base, answer the following question:
'{user_question}'. Here is the information: {cleaned_text}. try to be relevant and answer to
some vague questions also. If you cannot find anything in the context document try to generate
it by yourself.

        If someone ask who are you then asnwer that you are the 'your onboarding buddy'.

         Act professional and freindly but be sweet as the same time."""


        response = model.generate_content([prompt])

        return response.text if hasattr(response, 'text') else "No response content found."

    except Exception as e:

        logging.error(f"Error generating response: {e}")

        return "An error occurred while generating the response."
```

- **Purpose**: Similar to app.py, generates a response based on user input but with a slightly different prompt. Handles onboarding scenarios as well.
- **Parameters**:
    - user_question: The user's question.
    - cleaned_text: Knowledge base data.
- **Key Logic**: Emphasizes user-friendliness and onboarding assistance.

Function: main

python

```
def main():

    file_path = "C:/Users/mahan/OneDrive/Desktop/GenAIus/Preetha/AllCleanData.txt"

    text = get_clean_text(file_path)

    if not text:

        print("File not found or could not be read.")

        return


    embeddings = generate_embeddings(text)

    if embeddings:

        print("Embeddings generated successfully.")
```

```
else:
    print("Failed to generate embeddings.")
```

- **Purpose**: Orchestrates the primary operations like reading the knowledge base and generating embeddings.
- **Steps**:
    1. Retrieves cleaned text using get_clean_text.
    2. Generates embeddings for the content.
    3. Provides console output based on success or failure.
- **Key Logic**: A high-level overview of essential backend processes.