**Documentation for Telegram Chatbot System**

**Overview**

This document outlines the approach and methodology followed in developing a chatbot system using the Telegram Bot API, MongoDB for database management, **Google Gemini API** for generating responses, and **Google Custom Search API** for web search functionality. The system was developed to meet specific requirements, including user registration, AI-powered chat, image/file analysis, and web search.

**Approach to Solving the Problem**

**Initial Approach with Crew AI (Agent-based)**

Initially, I used **Crew AI** to power the chatbot’s agent management and task automation. The idea was to leverage Crew AI’s agent system to handle user queries, image analysis, and web search. However, I encountered a significant roadblock with **Groq**, the server responsible for accessing Large Language Models (LLMs), which went down. This impacted the initial integration and limited the functionality I could use for agent-based interactions.

Despite these challenges, I proceeded with using **Google Gemini** for the AI responses and **Google Custom Search API** for web searches, ensuring the system was functional and met the required features.

**Features and Functionalities of the Chatbot**

**1. User Registration**

* **Registration Process**:
  + When a user first interacts with the bot, it registers their **first\_name**, **username**, and **chat\_id** in **MongoDB**.
  + The bot asks the user to share their **phone number** using Telegram’s contact-sharing feature.
  + Once the phone number is received, it is stored in **MongoDB**, and a confirmation message is sent to the user, confirming their number.
* **MongoDB Database Design**:
  + A **users collection** stores all user details: **chat\_id**, **first\_name**, **username**, and **phone\_number** (once shared).

**2. Gemini-Powered Chat**

* **AI Response System**:
  + The chatbot uses the **Google Gemini API** (**google.generativeai**) to generate responses for user queries.
  + When a user sends a message, it is forwarded to Gemini, which processes the query and provides a response.
  + The chatbot logs both the **user input** and the **bot response** in the **chat history collection** in **MongoDB** with a timestamp.
* **MongoDB Database Design**:
  + The **chat history collection** stores all interactions with the bot. Each document contains the **chat\_id**, **user\_message**, **bot\_response**, and **timestamp**.

**3. Image/File Analysis**

* **Image/Document Upload**:
  + The bot supports **images** and **documents** (such as **JPG**, **PNG**, **PDF**) uploaded by users.
  + Upon receiving an image or document, the bot uses **Google Gemini** to describe the content and generate a response.
  + For **PDFs**, the bot extracts images from the document and provides a description based on the extracted content.
* **File Metadata Management**:
  + All metadata related to files (e.g., file type, file name, description) is stored in **MongoDB** for historical tracking and analysis.
* **MongoDB Database Design**:
  + The **file\_metadata collection** stores metadata about all files sent by users, including **file\_name**, **file\_description**, and **timestamp**.

**4. Web Search**

* **Web Search Functionality**:
  + Users can perform a **web search** by typing **/websearch** followed by a query.
  + The chatbot uses the **Google Custom Search API** to perform the search and returns a summary of the top search results.
  + The chatbot summarizes the search results and provides the user with **3 top links**.
* **MongoDB Database Design**:
  + A **web\_search\_history collection** stores user queries and the corresponding **search results**, **timestamp**, and **chat\_id**.

**Trial and Error Process**

**1. Initial Attempt with Crew AI and Groq Server**

* My first approach was to integrate **Crew AI** for agent management and use **Groq** to access Large Language Models (LLMs). This setup aimed to streamline user queries, image analysis, and web searches using a central agent.
* However, due to downtime of **Groq**’s server, the system could not connect to LLMs, and the agent functionality was severely impacted.

**2. Use of Google Gemini for Chat Responses**

* Since **Gemini** was mandatory to use for generating AI responses, I continued with **Google Gemini API** for chat interactions.
* **Gemini** provided a reliable and functional way to generate responses for user queries, which proved to be efficient and accurate.

**3. Use of Google Custom Search API for Web Search**

* With the original AI agent approach not viable due to the Groq server issue, I utilized the **Google Custom Search API** for web searches.
* The API allowed the bot to perform searches and return concise summaries of top search results, which worked seamlessly with the chatbot.

**4. Image/File Processing via Gemini**

* The integration of **Google Gemini** for image and document analysis was relatively smooth.
* For documents like PDFs, additional handling was required to extract and analyze images before feeding them to Gemini for description generation.

**Conclusion**

The chatbot system effectively meets all required functionalities:

1. **User Registration** (with MongoDB integration),
2. **Gemini-Powered Chat** (for AI-driven responses),
3. **Image/File Analysis** (with metadata storage),
4. **Web Search** (via the Google Custom Search API).

Despite challenges related to server downtimes and integration issues, the system was able to pivot using available APIs to achieve the desired outcomes. This process demonstrates my problem-solving skills, adaptability in handling system failures, and ability to implement effective AI-powered solutions.

This project highlights my technical capabilities and my readiness to contribute to a team, showcasing my ability to navigate real-world challenges in AI and chatbot development.