**How to Run the Project**

1. **Download Project Files**  
   Download the project files (agents.py, crew.py, task.py, tools.py, app.py) from the provided drive link. Ensure all files are placed in the same directory.
2. **Install Python**  
   Make sure Python 3.7 or later is installed on your system.
3. **Create a Virtual Environment**  
   Open a terminal or command prompt and navigate to the project files' directory.  
   Run the following command to create a virtual environment:

python -m venv <env\_name>

Replace <env\_name> with your preferred environment name, e.g., venv.

1. **Activate the Virtual Environment**
   * **On Windows:**

<env\_name>\Scripts\activate

* + **On macOS/Linux:**

source <env\_name>/bin/activate

1. **Install Dependencies**  
   Use the following command to install the required libraries:

pip install -r requirements.txt

1. **Set Up Environment Variables**
   * Create a .env file in the project directory if it doesn’t already exist.
   * Add your OpenAI API key to the .env file:

OPENAI\_API\_KEY=your\_openai\_api\_key\_here

1. **Run the Application**  
   Start the Streamlit app by running the following command in the terminal:

streamlit run app.py

1. **Interact with the Chatbot**
   * Once the app starts, it will open in your default browser.
   * You can start interacting with the chatbot by asking questions about marketing with Gen Z insights.

This project is a modular, Streamlit-based chatbot system powered by LangChain, designed to answer user queries efficiently. It includes a mechanism to optimize embedding operations using a hashing algorithm to avoid redundant computations.

**1. Key Objectives**

* **Efficient Query Processing**: The chatbot uses LangChain agents and tools for dynamic query handling.
* **Optimized Embedding Workflow**: A hashing algorithm checks the metadata for changes, ensuring embeddings are updated only when necessary.
* **Streamlined Front-End**: Built with Streamlit for user-friendly interaction.
* **Extensible Architecture**: Modular components for easy updates and scalability.

**2. Project Components**

**a. agents.py**

* **Role**: Implements LangChain agents, which are responsible for routing queries and managing task execution.
* **Functionality**:
  + Each agent decides whether a query requires external tools, task delegation, or direct response generation.
  + Routes queries based on their nature, ensuring efficient and relevant responses.

**b. crew.py**

* **Role**: Manages a "crew" of agents/tools and their configurations.
* **Functionality**:
  + Centralized management of available tools and agents.
  + Ensures compatibility and optimal performance of all components.

**c. task.py**

* **Role**: Handles the execution of individual tasks delegated by agents.
* **Functionality**:
  + Breaks down user queries into smaller tasks.
  + Executes these tasks independently, allowing modular development and debugging.

**d. tools.py**

* **Role**: Implements utility functions and API integrations.
* **Functionality**:
  + Integrates external APIs like OpenAI for generating responses.
  + Includes the **hashing algorithm** to check embeddings metadata for changes.

**e. app.py**

* **Role**: The entry point for the chatbot application.
* **Functionality**:
  + Provides an interactive front-end using Streamlit.
  + Manages user input, displays chatbot responses, and maintains session state.

**3. How It Works: Step-by-Step**

**Step 1: Environment Setup**

1. The user creates and activates a Python virtual environment.
2. Necessary libraries are installed via pip install -r requirements.txt.
3. Sensitive information, like API keys, is loaded from a .env file.

**Step 2: Application Initialization**

1. The app.py file initializes the Streamlit interface.
2. LangChain agents, tools, and tasks are loaded and configured.
3. Metadata from previously generated embeddings is loaded, along with their hashes.

**Step 3: User Interaction**

1. Users input queries into the Streamlit interface.
2. The input is passed to the agent system, which evaluates the query.

**Step 4: Query Processing**

1. The primary agent determines:
   * Whether the query needs an external API or tool.
   * Whether embedding is required or the metadata remains unchanged.
2. If embedding is needed:
   * The metadata of the new input is hashed using a hashing algorithm (e.g., SHA256).
   * The hash is compared against stored hashes to determine if the embedding already exists.
   * If no change is detected, the system skips re-embedding, saving computational resources.

**Step 5: Task Execution**

1. If the query involves fetching external data:
   * Tools defined in tools.py are called to execute API requests or computations.
2. For general conversational queries:
   * OpenAI APIs are used to generate the response.

**Step 6: Response Generation**

1. The result of the task or API call is returned to the agent.
2. The agent formats the response and sends it back to the user.

**Step 7: Continuous Interaction**

1. The system maintains session context, allowing multi-turn conversations.
2. Metadata and hashes are updated dynamically for new embeddings.

**4. Technologies Used**

1. **Streamlit**: For creating the web-based front-end.
2. **LangChain**: For modular and flexible agent-based query handling.
3. **Hashing Algorithm**: Used to detect changes in embedding metadata, ensuring efficiency.
   * Example: **SHA256** to hash metadata.
   * Allows the system to skip re-embedding if no changes are detected.
4. **OpenAI API**: For natural language processing and response generation.
5. **Python Libraries**: Modular components for tasks, tools, and API integrations.
6. **Environment Variables**: Securely manage API keys and sensitive data.