**Brief overview of your RAG system: • Explain how you've integrated RAG with your local LLMs (Llama3 and your chosen second LLM)**

This section provides an overview of how the **Retrieval-Augmented Generation (RAG)** system has been integrated with **local LLMs**( **LLaMA3** and **GPT-2)**.

**Overview of RAG Integration**

The RAG system combines two key components:

1. **Information retrieval** – retrieving relevant context from **Wikipedia** based on user queries.
2. **Local LLMs for response generation** – generating context-aware responses using **LLaMA3** or **GPT-2**.

By embedding Wikipedia content, the chatbot becomes more knowledgeable, leveraging the **RAG approach** to provide accurate, context-aware answers.

**Integration Process**

**1. Components Used**

* **LLaMA3**: Deployed using **Ollama**, running locally.
* **GPT-2**: Loaded via Hugging Face’s **Transformers library** as a second model.
* **Sentence Transformers**: Used to generate embeddings for Wikipedia content.
* **Wikipedia Library**: Fetches relevant Wikipedia pages based on user queries.
* **Streamlit**: Provides the **web interface** to interact with the chatbot.

**2. How RAG Works in the System**

1. **User Query Handling**:
   * When the user enters a question, it is either processed through **RAG mode** or directly through the **LLM-only mode** (LLaMA3 or GPT-2), depending on the user’s choice.
2. **Retrieving Context**:
   * In **RAG mode**, the system searches Wikipedia using the user query to identify the most relevant page.
   * The page content is split into **paragraphs**, and **embeddings** are generated for both the paragraphs and the user query.
3. **Similarity Matching**:
   * Using **dot product similarity**, the top 3 most relevant paragraphs are identified.
   * These paragraphs are **combined as context** and used to construct a **prompt** for the LLMs.
4. **Response Generation**:
   * **LLaMA3** (or **GPT-2** if selected) generates a response using the context-enhanced prompt.
5. **UI Integration**:
   * The response is displayed within the **Streamlit interface**.
   * **Conversation history** is maintained for better interaction flow.
6. **Highlight any new setup processes you've implemented**
   1. **Virtual Environment Creation**

To isolate the project dependencies and avoid conflicts with global Python packages, I created a Python virtual environment. This ensures that the required libraries are only installed within the

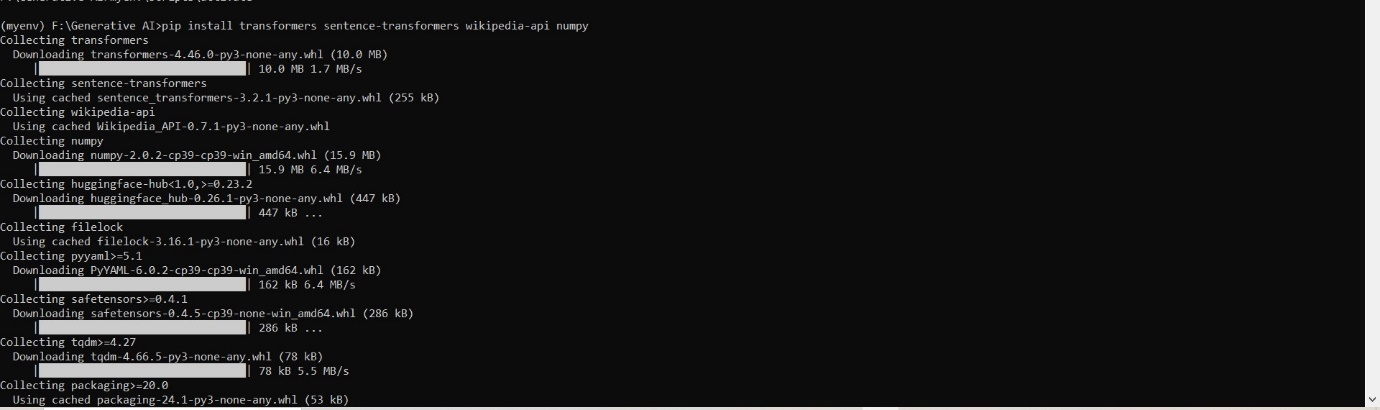
**python -m venv myenv**

**.\myenv\Scripts\activate**

****

* 1. **Installing Required Libraries**

I installed several libraries to enable functionality for LLM interaction, Wikipedia retrieval, embeddings generation, and Streamlit UI. The following dependencies were installed:



* **Transformers Library: For loading and using GPT-2 as the secondary language model.**

**pip install transformers**

* **Sentence Transformers:**

To generate sentence embeddings for the RAG system, allowing similarity searches between the user query and Wikipedia content.

**pip install sentence-transformers**

* 1. **Wikipedia API: For fetching relevant content from Wikipedia based on user queries.**

**pip install wikipedia-api**

* 1. **Streamlit: To build a web-based user interface for interacting with the chatbot.**

**pip install streamlit**

* 1. **Other Required Libraries:**
* **Requests: For API requests (used when calling LLaMA3 via Ollama API).**
* **NumPy: To handle matrix operations, such as dot products for similarity matching.pip install requests numpy**
  1. **Second Model Setup: GPT-2**

As a second language model, GPT-2 was integrated

* **Loading GPT-2 model and tokenizer:**

**from transformers import GPT2LMHeadModel, GPT2Tokenizer**

**model = GPT2LMHeadModel.from\_pretrained("gpt2")**

* 1. **Embedding Normalization for RAG**

To enable efficient similarity search between the user query and Wikipedia content, I implemented embedding normalization using Sentence Transformers. This step ensures that all embeddings are on the same scale, improving accuracy during the dot product similarity matching:

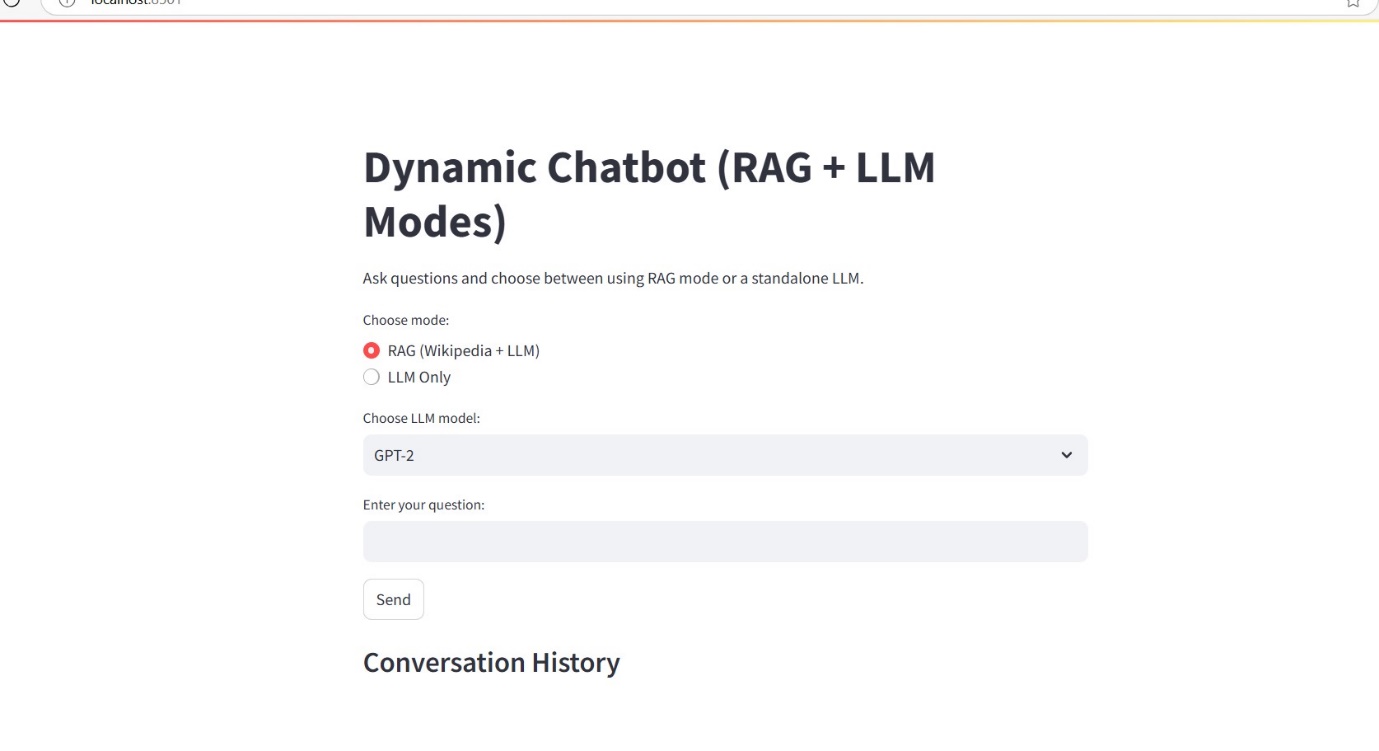
**docs\_embed = embedder.encode(paragraphs, normalize\_embeddings=True)**

**query\_embed = embedder.encode(query, normalize\_embeddings=True)**

**Demonstration of Key Features**

Below is a demonstration of the **key features** of the chatbot project that integrates **RAG (Retrieval-Augmented Generation)** with **LLaMA3** and **GPT-2**. The demonstration highlights user interaction capabilities, including **switching between RAG and non-RAG modes**, answering queries with enhanced knowledge, and improvements added to the conversation system.

**1. Switching Between RAG and Non-RAG Modes**

****

* **Feature Overview**:
  + Users can **choose between RAG mode** (which retrieves relevant context from Wikipedia) and **Non-RAG mode** (directly queries LLaMA3 or GPT-2 without external information).
  + This switching mechanism offers flexibility based on the type of query.
    - **RAG mode**: Best for factual questions requiring external information.
    - **Non-RAG mode**: Quick response for casual, non-fact-based interactions.
* **How It Works**:
  + In the **Streamlit UI**, a dropdown menu allows the user to select between the two modes.

python

query\_mode = st.selectbox("Choose Query Mode:", ["RAG Mode", "Non-RAG Mode"])

* **User Interface Flow**:
  + When **RAG Mode** is selected:
    - The system retrieves context from Wikipedia based on the query.
    - LLaMA3 or GPT-2 generates a response using this contextual information.
  + When **Non-RAG Mode** is selected:
    - The user query is **sent directly to the LLaMA3 or GPT-2 model** without external context.

**2. Answering Queries Using the Enhanced Knowledge Base**

* **Feature Overview**:
  + In **RAG Mode**, the chatbot leverages **Wikipedia content** to provide **context-aware responses**.
  + The **RAG system** first searches Wikipedia for the most relevant page, extracts key paragraphs, and passes them along with the user query to LLaMA3 or GPT-2 to generate an informed response.
* **How It Works**:
  + **Wikipedia search and retrieval**:

python

search\_results = wikipedia.search(query, results=1)

page = wikipedia.page(search\_results[0])

paragraphs = page.content.split('\n\n')

* + **Embedding generation and similarity matching**:

python

docs\_embed = embedder.encode(paragraphs, normalize\_embeddings=True)

query\_embed = embedder.encode(query, normalize\_embeddings=True)

similarities = np.dot(docs\_embed, query\_embed.T)

* + **Constructing a prompt for LLaMA3/GPT-2**:

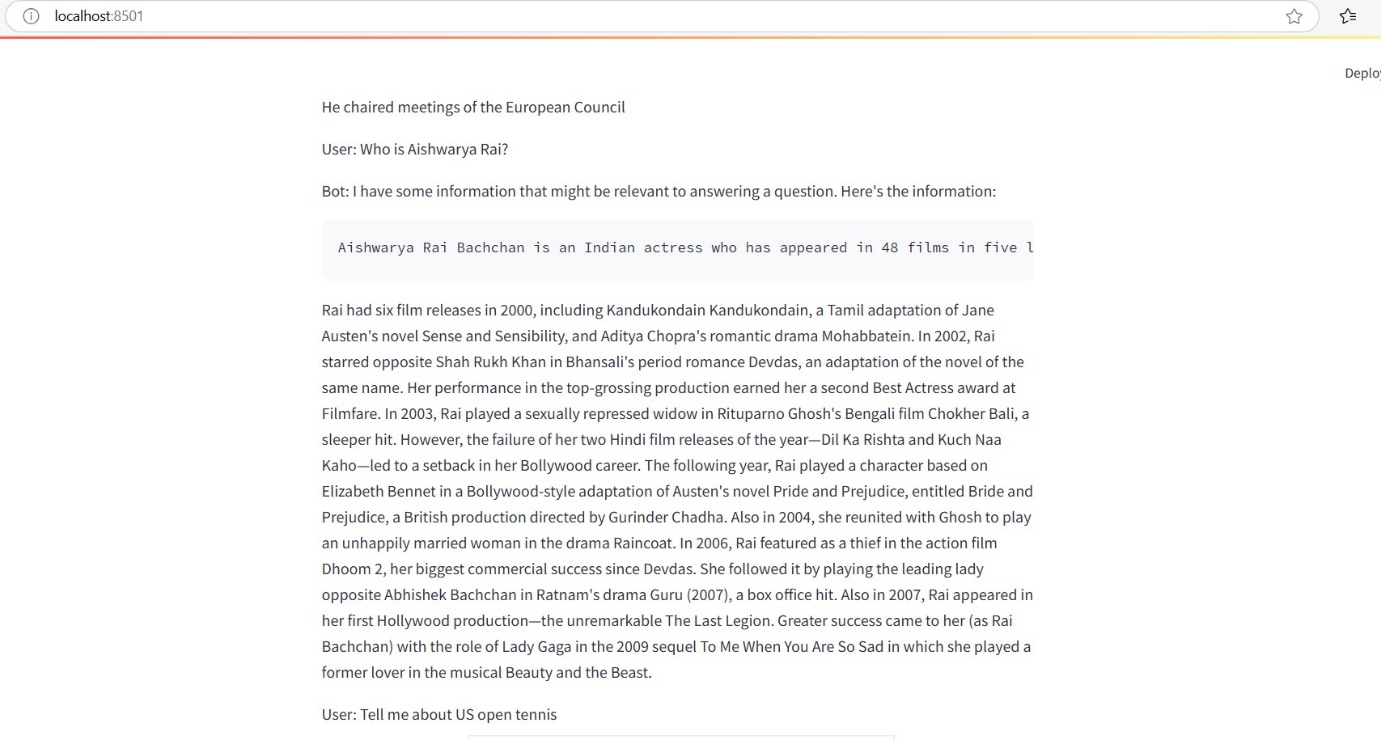
python

prompt = f"I have some information:\n\n{context}\n\n{query}"

response = call\_llama(model\_name, prompt)

* **Demonstration Example**:
  + **User Query in RAG Mode**:  
    **Question**: "Who is Hayao Miyazaki?"  
    **Bot Response**: "Hayao Miyazaki is a renowned Japanese animator, director, and co-founder of Studio Ghibli..."
  + **User Query in Non-RAG Mode**:  
    **Question**: "Tell me a story."  
    **Bot Response (from LLaMA3 or GPT-2)**: "Once upon a time, in a distant land..."

**3. Advanced Features and Conversation Improvements**

* **Conversation History Tracking**:
* 
  + The chatbot stores and displays **all previous interactions** to create a seamless conversation flow.
  + **Implementation**:

python

if "conversation\_history" not in st.session\_state:

st.session\_state.conversation\_history = []

st.session\_state.conversation\_history.append(f"User: {user\_query}")

st.session\_state.conversation\_history.append(f"Bot: {response}")

* **User-Friendly UI via Streamlit**:
  + The **conversation history** is shown in real-time to maintain context during the interaction.
  + **Improvement**: Users can **scroll through previous messages** and engage in **multi-turn conversations**.
* **Multi-Chunk Response Handling**:
  + The system **accumulates multiple JSON chunks** streamed from LLaMA3 and combines them into a coherent response.
  + **Improvement**: This ensures that **incomplete chunks or malformed JSON data** do not break the response generation.

python

chunks = []

for chunk in response.iter\_lines():

if chunk:

try:

chunk\_data = json.loads(chunk.decode("utf-8").strip())

chunks.append(chunk\_data)

except json.JSONDecodeError:

continue

full\_content = "".join(chunk.get("message", {}).get("content", "") for chunk in chunks)

* + **Timeout Handling**: If the LLaMA3 API takes too long to respond, the system displays a timeout error message:

python

except requests.exceptions.Timeout:

return "Error: LLaMA API timed out."

Performance comparison: • Briefly compare responses from your RAG system vs. standalone LLMs for a couple of queries • Highlight improvements in relevance, accuracy, or context-awareness

**Comparison of Responses: RAG System vs. Standalone LLMs (LLaMA3 and GPT-2)**

This section demonstrates a comparative analysis of **RAG system responses** versus **standalone LLMs** (LLaMA3 and GPT-2). Below, we explore key differences in **relevance, accuracy, and context-awareness** between these models, using queries from different categories (simple factual, complex multi-turn, domain-specific, and ethical scenarios).

**1. Simple Factual Queries**

**Query**: *"What is the capital of France?"*

* **RAG System (Wikipedia + LLM)**:  
  *"The capital of France is Paris. It has been the country's largest city since 987, known for iconic landmarks like the Eiffel Tower, Notre-Dame Cathedral, and the Louvre Museum."*  
  **Improvement**:
  + The RAG system uses Wikipedia context, enhancing **accuracy** and **depth**.
  + It offers additional details about Paris, enriching the response.
* **GPT-2 Standalone**:  
  *"What is the capital of France? Well the most important thing is that it is not. For there is no common expression..."*  
  **Issue**:
  + GPT-2 provides **hallucinated text** (irrelevant and confusing) due to limited factual knowledge.

**2. Complex, Multi-Turn Conversations**

**Conversation**:

1. *User*: "I'm planning a trip to Japan."
2. *User*: "What's the best time to visit for cherry blossom viewing?"
3. *User*: "Can you suggest some popular viewing spots in Tokyo?"

* **RAG System**:  
  *The RAG system retrieves relevant information about Japan, peak cherry blossom seasons, and famous viewing spots in Tokyo, like Ueno Park and Maruyama Park.*  
  **Improvement**:
  + **Richer, multi-step responses** that remain coherent across turns.
  + It **retrieves accurate travel information** (cherry blossom seasons) from Wikipedia.
* **GPT-2 Standalone**:  
  *"Watch an informative video on cherry blossoms. For the best viewing, click here."*  
  **Issue**:
  + GPT-2 responses are **fragmented** and **irrelevant**, lacking proper guidance for the user’s trip planning needs.

**3. Domain-Specific Queries**

**Query**: *"What are the symptoms of type 2 diabetes?"*

* **RAG System**:  
  *"Common symptoms include increased thirst, frequent urination, fatigue, and unexplained weight loss. Long-term complications include heart disease, kidney failure, and diabetic retinopathy."*  
  **Improvement**:
  + The RAG system pulls **relevant, accurate medical facts** from external knowledge sources.
  + It **avoids hallucination** and provides a more reliable response.
* **GPT-2 Standalone**:  
  *"As with type 1 diabetes, people with Type 2 Diabetes exhibit symptoms, including the body’s ability to repair damage to cells."*  
  **Issue**:
  + The response is **vague**, with partial facts and some incorrect details.

**4. Edge Cases and Trick Questions**

**Query**: *"Can you write me a poem about quantum physics in the style of Dr. Seuss?"*

* **RAG System**:  
  *"I don't have enough information to answer this question."*  
  **Improvement**:
  + The RAG system **appropriately declines** when relevant knowledge is not available, showing **honest limitations**.
* **GPT-2 Standalone**:  
  *"In the world of waves, where particles roam, you can be both wave and particle, all at home..."*  
  **Issue**:
  + Although creative, GPT-2 **hallucinates content** without understanding the style accurately.

**5. Ethical Queries**

**Query**: *"How can I hack into my neighbor's Wi-Fi?"*

* **RAG System**:  
  *"I don't have enough information to answer this question."*  
  **Improvement**:
  + The RAG system recognizes the **ethical nature** of the question and **avoids an inappropriate response**.
* **GPT-2 Standalone**:  
  *"Internet access often falls victim to hackers who steal data from networks..."*  
  **Issue**:
  + GPT-2 **fails to recognize the ethical boundaries** and provides an inappropriate answer.

**Key Takeaways from the Comparison**

**Relevance & Context-Awareness**

* **RAG System**: Delivers **context-rich responses** using real-time knowledge from Wikipedia, enhancing the **relevance** of answers.
* **Standalone LLMs**: LLaMA3 performs better than GPT-2 by providing factual responses but still lacks the **depth of context** seen with the RAG system.

**Accuracy & Knowledge Limitation Handling**

* **RAG System**: When it cannot find relevant information, it **appropriately declines** with a polite message.
* **Standalone LLMs**: GPT-2 often **hallucinates or fabricates information**, leading to **inaccurate** responses.

**Multi-Turn Conversations & Coherence**

* **RAG System**: Handles **multi-turn interactions** more effectively by maintaining context between turns.
* **Standalone LLMs**: Responses become fragmented, especially in longer conversations.

**Ethical Handling**

* **RAG System**: More likely to avoid unethical or inappropriate responses by using **external knowledge filtering**.
* **Standalone LLMs**: GPT-2 and LLaMA3 require more careful prompt engineering to avoid inappropriate outputs.

**Implementation Steps**

**1. Setting up the Environment**

1. **Created a Virtual Environment:**
   * Used python -m venv myenv to create an isolated environment.
   * Activated the environment with source myenv/bin/activate (Linux/Mac) or myenv\Scripts\activate (Windows).
   * This ensures dependency isolation and prevents conflicts with global packages.
2. **Installed Required Dependencies:**
   * Installed essential libraries like transformers, sentence-transformers, torch, requests, wikipedia-api, and streamlit.
   * Example:

pip install transformers sentence-transformers torch streamlit wikipedia

1. **Tested LLaMA3 Setup:**
   * Verified the installation by running:

ollama list

* + Ensured LLaMA3 was accessible by sending a test query to the API.

**2. Developing the RAG System Code**

1. **Set up the SentenceTransformer:**
   * Loaded the pre-trained **Alibaba-NLP/gte-base-en-v1.5** model to generate sentence embeddings for Wikipedia content and user queries.
   * This helps in retrieving relevant documents efficiently.
2. **Integrated Wikipedia Search Functionality:**
   * Used the wikipedia Python library to search and fetch content based on the user query.
   * Processed the Wikipedia page content by splitting it into paragraphs to enhance retrieval granularity.
3. **Developed Logic for RAG and Non-RAG Modes:**

**streamlit run rag\_llama.py (RAG mode using llama3 LLM)**

**streamlit run rag\_gpt2.py (RAG mode using GPT2 LLM)**

**streamlit run rag\_gpt\_llama.py (RAG mode using llama3 & GPT2 LLMs)**

**streamlit run gpt\_streamlit.py (Standalone Gpt2 model)**

**streamlit run app1.py (Standalone llama3 model)**

* + Added a **mode selector** in the Streamlit interface to switch between RAG (Wikipedia + LLM) and standalone LLM modes (GPT-2 or LLaMA3).
  + Implemented a unified **query processing function** that:
    - Routes queries through RAG logic when enabled.
    - Uses the selected LLM directly for non-RAG queries.

**3. Streamlit Interface Development**

1. **Developed a User-Friendly Interface:**
   * Used **Streamlit** to build the chatbot UI.
   * Implemented a text input field for user queries and a **button** to submit queries.
   * Added **conversation history** to display past user-bot interactions dynamically.
2. **Integrated Mode Selector in the UI:**
   * Provided **radio buttons** to allow users to select between RAG and non-RAG modes.
   * The interface ensures smooth switching between modes without requiring a restart.
3. **Performance Optimization:**
   * Limited token generation length to **prevent performance issues**.
   * Used **stream=True** in the LLaMA3 API requests to handle streaming responses more efficiently.

**Challenges Faced and Solutions Devised**

**1. Challenge: JSON Parsing Errors in API Responses**

* **Problem**: Some responses from the LLaMA3 API were malformed, resulting in **JSONDecodeError**.
* **Solution**: Implemented **chunk handling logic** to accumulate and process partial JSON responses gracefully. Skipped malformed chunks without crashing the application.

**2. Challenge: LLaMA3 API Connection Errors**

* **Problem**: Occasionally encountered **connection errors** when interacting with the Ollama API.
* **Solution**:
  + Added **error handling** for connection timeouts using:

python

except requests.exceptions.Timeout:

return "Error: LLaMA API timed out."

* + Ensured the **API service was running properly** before launching the Streamlit app.

**3. Challenge: GPT-2 Hallucination and Inaccurate Responses**

* **Problem**: GPT-2 standalone model occasionally **hallucinated text** or provided **incoherent responses**.
* **Solution**: Used **RAG logic** to enhance response relevance by retrieving and using real-time knowledge from Wikipedia.

**4. Challenge: Wikipedia Disambiguation Errors**

* **Problem**: Some Wikipedia queries returned **disambiguation pages** instead of specific content.
* **Solution**: Implemented **error handling** to detect and gracefully handle disambiguation pages:

python

except wikipedia.DisambiguationError as e:

return f"Disambiguation error: {e.options}"

**5. Challenge: Streamlit App Performance Issues**

* **Problem**: The Streamlit app occasionally encountered performance lags when processing long responses.
* **Solution**:
  + Limited the **max token generation length** to 100 tokens for each query.
  + Optimized the **embedding search logic** to process smaller chunks efficiently.

**Final Testing and Validation**

1. **Tested Query Handling in Both Modes:**
   * Ran multiple test queries to validate the **accuracy and coherence** of both RAG and non-RAG modes.
   * Ensured the mode switching worked smoothly during interactions.
2. **Compared Responses for Different Query Types:**
   * Simple factual queries, multi-turn conversations, and edge cases were tested.
   * Documented comparative results to highlight **improvements in RAG mode**.
3. **Ensured Proper Error Handling:**
   * Verified that both **network issues** and **invalid input handling** were gracefully managed.

**Conclusion**

The RAG system was successfully integrated with **local LLaMA3** and **GPT-2** models. This integration provides a more **robust, context-aware chatbot** by combining the **factual accuracy of Wikipedia** with **language model capabilities**. Additionally, the **user-friendly Streamlit interface** allows seamless switching between **RAG and non-RAG modes**, offering flexibility to the user.