RAG Chatbot Project Report

1. Description of Document Structure and Chunking Logic

The core documentation for this RAG chatbot is assumed to be a PDF file (initial_data/AI Training Document.pdf). To effectively use this document with a vector database and an LLM, it undergoes a crucial preprocessing pipeline managed in cleaning_data.ipynb.

The process involves:

- **Text Extraction:** The extract_clean_text function uses PyMuPDF (fitz) to read the PDF document page by page and extract all textual content into a single string. This ensures all information from the original document is captured.
- **Text Cleaning:** The clean_text function performs regular expression-based cleaning. It removes common artifacts like page numbers (\nPage \d+\n) and normalizes whitespace by replacing multiple spaces or newlines with a single space (\s+). This step is essential to reduce noise and improve the quality of the text fed into the chunking process, preventing irrelevant characters from affecting embeddings.
- **Sentence-Aware Chunking:** The chunk_text_spacy function is responsible for dividing the large, cleaned text into smaller, manageable chunks. This is a critical step for RAG systems because:
    - **Relevance:** Smaller chunks are more likely to contain a focused piece of information, leading to more precise retrieval from the vector database.
    - **Context Window:** LLMs have a limited context window. By providing smaller, relevant chunks, we ensure that the most pertinent information fits within the model's input limit, maximizing its ability to generate accurate responses.
    - **Efficiency:** Processing smaller chunks is generally faster for both embedding generation and LLM inference.

The chunk_text_spacy function utilizes spaCy, a powerful NLP library, specifically its en_core_web_sm model, for robust sentence tokenization. It iterates through sentences and aggregates them into chunks, aiming for a max_words limit (default 300 words) while ensuring each chunk meets a min_words threshold (default 100 words). If adding a new sentence exceeds max_words, and the current chunk already meets min_words, the current chunk is finalized, and a new chunk begins with the current sentence. This approach ensures that chunks maintain semantic coherence by respecting sentence boundaries and are of an optimal size for embedding and retrieval. The final chunks are saved as a JSON file (chunks.json).

2. Explanation of Embedding Model and Vector DB Used

Embedding Model: text-embedding-004

- **Choice:** The project leverages Google's text-embedding-004 model. This model is part of the latest generation of embedding models designed for high-quality semantic understanding.
- **Purpose:** Its primary role is to transform human-readable text into dense numerical vectors (embeddings). These vectors capture the semantic meaning of the text, meaning that texts with similar meanings will have vectors that are numerically "close" to each other in the high-dimensional space.
- **Dimensionality:** The text-embedding-004 model outputs embeddings with a dimensionality of 768. This fixed size is a common characteristic of embedding models and dictates the number of features used to represent each piece of text.
- **Implementation:** The get_embeddings function in utils.py handles the API calls to generate these embeddings.

Vector Database: Pinecone

- **Choice:** Pinecone is utilized as the vector database for storing and retrieving the generated embeddings. Pinecone is a popular choice for production-grade RAG systems due to its scalability, low latency, and efficient similarity search capabilities.
- **Purpose:** Its main function in this architecture is to rapidly find the most relevant text chunks (based on semantic similarity) to a given user query. When a user asks a question, the question is also converted into an embedding. This query embedding is then sent to Pinecone, which performs a nearest-neighbor search to identify the top_k (default 2) most similar document embeddings.
- **Configuration:**
  - **Index Name:** Configured via the PINECONE_INDEX_NAME environment variable.
  - **Dimension:** Set to 768, matching the output dimensionality of the text-embedding-004 model.
  - **Metric:** cosine similarity is used. Cosine similarity measures the cosine of the angle between two vectors, ranging from -1 (opposite) to 1 (identical). A higher cosine similarity score indicates greater semantic resemblance between the query and the document chunk.
  - **Serverless Spec:** The index is created with a ServerlessSpec on AWS us-east-1, indicating a managed, serverless deployment for ease of use and scalability.
- **Implementation:** The build_vector_database.ipynb notebook handles the creation of the Pinecone index and the "upsert" (update or insert) of all document embeddings. The DetailsAgent uses the Pinecone client to perform query

operations to retrieve relevant documents.

3. Prompt Format and Generation Logic

The chatbot employs a two-agent system, each with its own prompt format and generation logic:

a) Guard Agent (gaurd_agent.py)

- **Purpose:** To act as a first line of defense, filtering out irrelevant or inappropriate queries. This prevents the more resource-intensive RAG pipeline from being triggered unnecessarily and maintains the chatbot's focus on eBay-related topics.
- **Prompt Format:** The GaurdAgent receives the user's latest messages (specifically, the last 5 messages in the conversation history as a string). It is given a strict system_instruction that defines its role and rules for allowed and disallowed questions.
  You are a helpful AI assistant for a ebay.
  Your task is to determine whether a customer is asking something relevant to ebay or not.
  The user is allowed to:
  1. Ask questions about ebay, it's documentation, it's policies related questions.
  2. The user can greet

  The user is not allowed to:
  1. Ask questions anythings else than the ebay related questions
  2. Ask questions about the staff or prices of the products
  3. Ask questions about the competitors of ebay
  4. Ask questions about the personal information of the staff or customers

  Your output should be in JSON format with the following structure, each key is a string and each value is a string,
  make sure to follow the format correctly:
  {
  "chain of thought": "go over each of the points above and see if the message lies under this point or not. Then you write some thought about as what point is the input relevant to."
  "decision": "allowed" or "not allowed", pick one of these choice and only write the word
  "message": leave the message empty "" if allowed otherwise write "sorry I am not allowed to help you with that, is there something else i can help you with?"
  }

- **Generation Logic:** The Guard Agent sends this prompt and the user's message to the gemini-1.5-flash model. It expects a JSON output with a decision key ("allowed" or "not allowed") and a message key. Based on the decision, it either allows the message to proceed or returns a blocked message.

b) Details Agent (details_agent.py)

- **Purpose:** To provide accurate and informative responses to eBay-related queries by retrieving relevant information from the documentation.
- **Prompt Format:** The DetailsAgent constructs a prompt that includes:
  - **Context:** The source_knowledge retrieved from the Pinecone vector database. This is a crucial element of RAG, providing the LLM with factual grounding.
  - **Query:** The user's original question.
  - **System Instruction:** A clear instruction guiding the LLM to act as a customer support agent for eBay and focus on answering the latest query using the provided context.

  Using the context below try to answer the query

  Context:
  {source_knowledge}

  Query:
  {user_messages}
  System Instruction: You are a customer support agent for a ebay, you should answer latest question as a support agent,
  focus on the latest query
  and provide the necessary information to the user regarding the latest query

- **Generation Logic:**
  1. The user's query is embedded using text-embedding-004.
  2. Pinecone is queried to retrieve the most relevant source_knowledge (text chunks).
  3. A final prompt is constructed, incorporating the retrieved source_knowledge and the user's query.
  4. This comprehensive prompt is sent to the gemini-1.5-flash model.
  5. The model generates a response based on the provided context and instructions, which is then returned to the user.

4. Example Queries with Responses

Here are a few example queries demonstrating the chatbot's behavior, highlighting both successful and failure cases (based on the provided Guard Agent rules and assumed documentation content):

Scenario 1: Successful Retrieval and Response

- **Query:** "What is eBay?"
- **Guard Agent Decision:** allowed (relevant to eBay documentation)
- **Details Agent Source Knowledge (Example Snippet):** "eBay is a marketplace that allows users to offer, sell, and buy goods and services in various geographic locations using a variety of pricing formats."
- **Chatbot Response:** "eBay is an online marketplace where users can buy and sell various goods and services globally using different pricing formats."
- **Success Case:** The chatbot accurately identified the core question about eBay, retrieved relevant information, and provided a concise summary.

Scenario 2: Successful Retrieval and Response (Specific Policy)

- **Query:** "Can eBay help resolve disputes between buyers and sellers?"
- **Guard Agent Decision:** allowed (relevant to eBay policies)
- **Details Agent Source Knowledge (Example Snippet):** "We may help facilitate the resolution of disputes between buyers and sellers through various programs."
- **Chatbot Response:** "Yes, eBay may help facilitate the resolution of disputes between buyers and sellers through various programs."
- **Success Case:** The chatbot correctly answered a specific policy-related question by retrieving the exact relevant sentence.

Scenario 3: Guard Agent Blocking (Irrelevant Topic)

- **Query:** "What is the capital of France?"
- **Guard Agent Decision:** not allowed (not related to eBay documentation)
- **Chatbot Response:** "Sorry, I am not allowed to help you with that. Is there something else I can help you with?"
- **Failure Case (Intentional):** The Guard Agent successfully identified the query as out-of-scope and blocked it, demonstrating its protective function.

Scenario 4: Guard Agent Blocking (Prohibited Question Type - Competitors)

- **Query:** "Is Amazon a competitor of eBay?"
- **Guard Agent Decision:** not allowed (asking about competitors is disallowed)
- **Chatbot Response:** "Sorry, I am not allowed to help you with that. Is there something else I can help you with?"
- **Failure Case (Intentional):** The Guard Agent correctly identified this as a

prohibited question type and prevented the DetailsAgent from engaging.

Scenario 5: Potential Hallucination/Limited Context (Hypothetical)

- **Query:** "What is eBay's policy on cryptocurrency payments for vehicle sales?"
- **Guard Agent Decision:** allowed (policy question related to eBay)
- **Details Agent Source Knowledge (Hypothetical):** The retrieved documents might talk about vehicle sales and general payment methods but *not* specifically about cryptocurrency for vehicle sales.
- **Chatbot Response (Potential Hallucination/Limitation):** "While eBay allows various payment methods for vehicles, specific policies on cryptocurrency payments for vehicle sales are not explicitly detailed. Please refer to the general payment terms."
- **Failure Case (Subtle):** This is a hypothetical scenario where the model attempts to answer based on partial information or general knowledge, leading to a somewhat vague or potentially misleading answer if the documentation *doesn't* cover it. This highlights the importance of the completeness of the source documentation. The model tries to be helpful but might not have the precise answer.

5. Notes on Hallucinations, Model Limitations, or Slow Responses

Hallucinations and Model Limitations:

- **Reliance on Source Knowledge:** The primary defense against hallucinations in a RAG system is the quality and completeness of the source_knowledge. If the retrieved documents do not contain the answer to a user's query, the DetailsAgent (powered by gemini-1.5-flash) might:
  - **Confabulate:** Generate information that sounds plausible but is not present in the provided context. This is a true hallucination.
  - **Generalize/Infer:** Attempt to infer an answer based on general knowledge or partially related information from the context, as seen in Scenario 5. While not a direct hallucination, it can still lead to less precise or inaccurate responses.
  - **Admit Lack of Knowledge:** Ideally, the model should be able to state that it cannot find the answer in the provided context. The current prompt encourages answering based on context, but a more explicit "cannot answer from context" instruction could be beneficial for robust responses.
- **Guard Agent Limitations:** While effective, the GaurdAgent's effectiveness is tied to the clarity and exhaustiveness of its system_instruction. Ambiguous queries or novel ways of asking disallowed questions might occasionally bypass the guard or be incorrectly flagged. Regular review and refinement of these rules are necessary.

- **Nuance and Interpretation:** LLMs, even with RAG, can sometimes struggle with highly nuanced questions or those requiring complex logical deductions across multiple, disparate document sections. The top_k retrieval might not always capture all necessary interlinked information.
- **Out-of-Distribution Queries:** For questions far removed from the training data of gemini-1.5-flash or the domain of eBay documentation, the model's performance will naturally degrade.

Slow Responses:

- **API Latency:** The primary source of response time in this architecture comes from API calls:
  - **Embedding API (text-embedding-004):** Generating embeddings for the user query adds a small but noticeable latency.
  - **Pinecone Query:** Searching the vector database introduces latency, though Pinecone is designed for low-latency retrieval. The time taken depends on index size and complexity of the query.
  - **LLM Inference (gemini-1.5-flash):** The gemini-1.5-flash model, while being optimized for speed ("flash" model), still introduces the most significant portion of the latency. The complexity of the query and the desired output length directly impact inference time.
- **Network Latency:** The round-trip time for API calls to Google's and Pinecone's servers can contribute to perceived slowness, especially for users geographically distant from the data centers.
- **Sequential Agent Calls:** The sequential nature of the GaurdAgent followed by the DetailsAgent (if allowed) means that for relevant queries, two LLM calls and one embedding/Pinecone query are made, cumulatively adding to the response time.
- **Streamlit Reruns:** The st.rerun() call after each interaction in development_code.py causes the entire Streamlit script to re-execute, which can introduce a slight delay compared to a purely event-driven web application, although for simple chatbots, it's often acceptable.

To mitigate slow responses, techniques like caching (st.cache_data, st.cache_resource in Streamlit for agent initialization), asynchronous processing (if the framework allowed), or optimizing the top_k value for Pinecone could be considered in a more performance-critical deployment. However, for interactive chat, gemini-1.5-flash generally provides a responsive experience.