Interview Questions and Answers on Retrieval Augmented Generation (RAG)

Q1: What is Retrieval Augmented Generation (RAG), and why is it crucial in the context of large language models (LLMs)?

A1: Retrieval Augmented Generation (RAG) is an advanced technique that significantly enhances the capabilities of large language models (LLMs). While LLMs are incredibly powerful in generating human-like text, they are limited to the information they were trained on, which often cuts off at a certain point in time. This limitation means that LLMs might not have access to the most recent data or private datasets that are essential for specific tasks. RAG addresses this issue by allowing the model to retrieve relevant information from external data sources,

including databases, documents, or APIs, and integrate this information into the model's prompt during the generation process. This integration enables LLMs to generate responses that are informed by the latest data or proprietary information, making RAG indispensable for applications that require reasoning over up-to-date or specialized datasets.

Q2: What are the two primary components of a Retrieval Augmented Generation (RAG) application, and how do they function together?

A2: A typical RAG application is composed of two primary components: indexing and retrieval-generation.

1. **Indexing:** This component is responsible for the initial preparation of data, which involves ingesting and organizing it for efficient retrieval. During this phase, data from various sources is loaded into the system, often using document loaders. The data is then split into

manageable chunks, a process that is crucial because it makes large documents easier to search and ensures they fit within the context window of LLMs. These chunks are then stored in a format optimized for retrieval, typically using a VectorStore alongside an Embeddings model. This step is usually performed offline.

2. **Retrieval and Generation:** This component operates at runtime. When a user submits a query, the system retrieves the most relevant data chunks from the indexed storage. The retrieval process is powered by a retriever that matches the user's query with the pre-indexed data. Once the relevant data is retrieved, it is combined with the user's query and passed to a language model (such as a ChatModel or LLM). The model then generates a response that is informed by both the query and the retrieved data. This real-time

interaction between retrieval and generation allows the RAG application to produce accurate and contextually appropriate answers.

Q3: How does the indexing process work in a RAG application, and why is it essential for the system's performance?

A3: The indexing process in a RAG application involves several crucial steps that prepare the data for efficient retrieval and subsequent use by the language model:

1. **Loading:** Data is first loaded into the system using document loaders, which can handle various data formats and sources.
2. **Splitting:** The loaded data is then split into smaller, more manageable chunks. This step is critical because large documents are difficult to search through and may

exceed the context window limitations of LLMs, making it hard for the model to process them effectively.

3. **Storing:** These chunks are stored in a VectorStore, where they are indexed using an Embeddings model. The Embeddings model converts the text data into high-dimensional vectors that capture the semantic meaning of the text, allowing for efficient similarity searches during the retrieval phase.

Indexing is essential because it organizes the data in a way that allows for quick and accurate retrieval when a user query is processed. Without a well-structured index, the system would struggle to provide relevant and timely answers, as it would be difficult to sift through large volumes of data in real-time.

Q4: What is the role of the retrieval component in a RAG application, and how does it impact the overall effectiveness of the system?

A4: The retrieval component plays a pivotal role in the RAG application by identifying and returning the most relevant pieces of data from the indexed storage in response to a user query. When a query is received, the retriever searches through the indexed data to find the chunks that best match the query's content. These relevant chunks are then passed to the language model as part of the prompt, which allows the model to generate a response that is both well-informed and contextually accurate. The effectiveness of the retrieval process is crucial because it directly influences the relevance and accuracy of the model's responses. If the retrieval component fails to find the most pertinent data, the model's generated response may lack context or contain inaccuracies, which can undermine the application's utility.

Q5: Why is it necessary to split documents into smaller chunks during the indexing process in a

RAG system, and what benefits does this provide?

A5: Splitting documents into smaller chunks during the indexing process is necessary for several reasons:

1. **Improved Search Efficiency:** Smaller chunks are easier and faster to search through, which enhances the overall retrieval speed and accuracy. This is particularly important when dealing with large datasets or when quick responses are required.
2. **Context Window Limitations:** Large language models have a finite context window, meaning they can only process a limited amount of text at one time. Splitting documents ensures that each chunk fits within the model's context window, allowing the model to process and understand the text more effectively.

3. **Increased Relevance:** By dividing documents into smaller chunks, the likelihood of retrieving highly relevant information for a given query increases. Each chunk represents a more focused piece of content, which can be more precisely matched to the user's query.

This process ensures that the system can efficiently retrieve and process the most relevant sections of a document, leading to more accurate and contextually appropriate responses.

Q6: Why is configuring LangSmith crucial when building complex applications with LangChain, and how does it contribute to the development process?

A6: Configuring LangSmith is crucial in the development of complex applications with LangChain because it provides essential tools for tracing and logging. As LangChain applications

become more sophisticated, they often involve multiple steps and calls to language models (LLMs), which can make the debugging and optimization processes challenging. LangSmith allows developers to inspect and monitor the internal workings of the application, providing visibility into each step of the chain or agent. This capability is essential for identifying and resolving issues, optimizing performance, and ensuring that the application behaves as expected. Without LangSmith, developers would have a much harder time understanding what is happening inside the application, which could lead to inefficiencies and errors.

Q7: What are the essential dependencies for working with LangChain, and how can they be installed to ensure smooth operation?

A7: To work with LangChain effectively, several essential dependencies are required. These include:

- **langchain:** The core LangChain package, which provides the foundational tools and interfaces for building RAG applications.
- **langchain_community:** A package that includes additional community-contributed tools and extensions that can enhance LangChain's functionality.
- **langchain_chroma:** A package that supports the integration of LangChain with Chroma, a popular vector store used for managing embeddings and performing similarity searches.

These dependencies can be installed using the following `pip` commands:

```
bashCopy codepip install langchain
```

```
pip install langchain_community
```

```
pip install langchain_chroma
```

Additionally, if you are using specific language models, you may need to install other packages like `langchain-openai`. Ensuring that all required dependencies are installed and up-to-date is critical for the smooth operation of LangChain applications.

Q8: How do the WebBaseLoader and RecursiveCharacterTextSplitter contribute to the LangChain RAG setup, and what are their specific roles?

A8: The WebBaseLoader and RecursiveCharacterTextSplitter are key components in the LangChain RAG setup, each serving a specific purpose:

- **WebBaseLoader:** This tool is used to load the content of specific webpages into the application. It parses the webpage and extracts the relevant text, making it accessible for further processing and indexing within the RAG system. This loader is particularly useful for

applications that need to integrate web-based data into their knowledge base.

- **RecursiveCharacterTextSplitter:** After the content is loaded, the RecursiveCharacterTextSplitter takes over to split this content into smaller chunks based on specified parameters, such as chunk size and overlap. This splitter works recursively, ensuring that each chunk is appropriately sized for efficient retrieval and processing within the context window of the LLM. The resulting chunks are then indexed and used by the application to retrieve and generate answers to user queries.

Together, these tools enable the seamless integration and processing of web content in a RAG application, facilitating more informed and accurate responses.

Q9: Why is setting environment variables like LANGCHAIN_TRACING_V2 and LANGCHAIN_API_KEY important in LangChain, and what do they achieve?

A9: Setting environment variables like `LANGCHAIN_TRACING_V2` and `LANGCHAIN_API_KEY` is important because they enable critical features in LangChain applications:

- **LANGCHAIN_TRACING_V2:** This variable activates the tracing feature within LangSmith, which is essential for monitoring the internal operations of a LangChain application. Tracing allows developers to track each step of the application's execution, helping them identify bottlenecks, errors, or inefficiencies that need to be addressed.
- **LANGCHAIN_API_KEY:** This variable is used to authenticate and connect to the LangSmith service. It ensures that the tracing and logging features are

securely enabled, allowing for proper tracking and monitoring of the application's performance.

These environment variables are integral to the development and maintenance of robust LangChain applications, providing the necessary tools for debugging and optimization.

Q10: What is the primary purpose of using DocumentLoaders in the LangChain Retrieval Augmented Generation (RAG) framework, and how do they function?

A10: DocumentLoaders play a critical role in the LangChain RAG framework by fetching data from various sources and returning it as a list of Document objects. Each Document object typically consists of two main components:

- **Page Content:** A string that contains the text content of the document.

- **Metadata:** A dictionary that includes additional information about the document, such as the source, date, or any other relevant attributes.

The DocumentLoader is responsible for gathering and structuring the raw data, making it ready for indexing and subsequent retrieval in the RAG application. This functionality is essential for creating question-answering systems, where the quality and structure of the data directly impact the accuracy and relevance of the generated responses.

Q11: How does the WebBaseLoader function within the LangChain ecosystem, and what benefits does it provide for processing web content?

A11: The WebBaseLoader functions as a tool for loading HTML content from web URLs into the LangChain ecosystem. It works by fetching the HTML content using `urllib` and parsing it with

`BeautifulSoup`, a powerful Python library for web scraping and content parsing. The parsed data is then converted into a list of Document objects, each containing the text content and associated metadata. This loader is particularly beneficial for applications that need to extract and utilize specific parts of a webpage, such as articles, blog posts, or structured data. By converting web content into Document objects, the WebBaseLoader enables seamless integration of online information into RAG systems, facilitating more comprehensive and up-to-date responses.

Q12: How can the HTML parsing process be customized when using WebBaseLoader, and why might this be necessary?

A12: The HTML parsing process in WebBaseLoader can be customized by passing specific parameters to the BeautifulSoup parser via the `bs_kwargs` argument. For instance, you can use a

`SoupStrainer` to filter and retain only specific HTML tags or classes, such as "post-title," "post-header," and "post-content." This customization is necessary when you want to focus on particular sections of a webpage while ignoring others. For example, if you're interested only in the main content of a blog post and want to exclude sidebars, footers, or advertisements, customizing the parsing process ensures that only the relevant information is extracted and indexed for retrieval in the RAG system.

Q13: What role does BeautifulSoup play in the WebBaseLoader, and how does it enhance the document loading process?

A13: BeautifulSoup is a critical component in the WebBaseLoader, responsible for parsing the fetched HTML content. It allows developers to filter and extract specific parts of the HTML based on tags, classes, or other attributes, making the

data more relevant and structured for further processing. By leveraging BeautifulSoup, the WebBaseLoader can convert raw HTML into a list of Document objects that are well-suited for indexing and retrieval in a LangChain application. This functionality is essential for extracting meaningful and targeted information from web pages, which can then be used to generate accurate and contextually relevant answers in a RAG system.

Q14: What is a SoupStrainer, and how does it improve the efficiency of HTML parsing in WebBaseLoader?

A14: A SoupStrainer is a filter used in conjunction with BeautifulSoup to selectively parse HTML content. It allows developers to specify which HTML tags or classes should be retained while discarding the rest. For example, a SoupStrainer can be configured to keep only the “post-title,” “post-header,” and “post-content” tags, ignoring other elements like

advertisements or navigation menus. This selective parsing improves the efficiency of the document loading process by ensuring that only the most relevant content is extracted and stored as Document objects. The use of a SoupStrainer is particularly useful in applications where the focus is on specific sections of a webpage, enabling more targeted and efficient retrieval and generation processes.

Q15: Why is it important to customize the HTML parsing when using WebBaseLoader, and what impact does this have on the RAG system?

A15: Customizing the HTML parsing process when using WebBaseLoader is important because it allows you to focus on and retain only the relevant sections of a webpage that are needed for your application. By filtering out unnecessary tags and content, you can ensure that the resulting Documents are concise and focused. This customization not only reduces the

amount of data that needs to be processed and indexed but also enhances the accuracy and relevance of the retrieval process. In a RAG system, where the quality of the retrieved data directly impacts the generated responses, this level of customization is crucial for optimizing both performance and output quality.

Q16: Why is it necessary to split long documents before embedding them in a Retrieval-Augmented Generation (RAG) system, and how does this practice improve system performance?

A16: Splitting long documents before embedding them in a RAG system is essential for several reasons:

- **Context Window Limitations:** Most language models have limited context windows, meaning they can only process a certain amount of text at one time. If a document is too long, it may exceed the model's context

window, leading to inefficient processing and potential loss of relevant information.

- **Improved Retrieval Efficiency:** By splitting documents into smaller, more manageable chunks, the retrieval process becomes more efficient. Smaller chunks are easier to search through and allow for more precise matching with the user's query.
- **Focused Attention:** Splitting documents ensures that the model's attention is focused on the most relevant sections during question-answering. This practice helps the model generate more accurate and contextually appropriate responses.

Overall, splitting long documents enhances the system's ability to retrieve and process relevant information, leading to better performance and more reliable outputs.

Q17: What is the purpose of using character overlap when splitting a document into chunks, and how does it preserve context?

A17: Character overlap between chunks is used to preserve the continuity and context of the text across splits. For example, if a statement at the end of one chunk is closely related to the beginning of the next, the overlap ensures that this connection is maintained. A typical implementation might involve a 200-character overlap, which helps to mitigate the risk of losing important context or information that might be split between chunks. This practice is especially important in a RAG system, where maintaining context is crucial for generating accurate and coherent responses. The overlap allows the model to consider related information from adjacent chunks, thereby improving the overall quality of the generated output.

Q18: What is the RecursiveCharacterTextSplitter, and why is it recommended for splitting generic text in LangChain?

A18: The RecursiveCharacterTextSplitter is a tool in LangChain designed to split documents into smaller chunks based on common separators such as new lines, sentences, or paragraphs. It works recursively, dividing the document until each chunk reaches the desired size. This splitter is recommended for generic text use cases because it ensures that chunks are logically coherent, preserving the structure of the text, such as paragraphs or sentences. This preservation is crucial for maintaining the context and meaning of the text, which is important for accurate retrieval and generation in a RAG system. By ensuring that each chunk is a self-contained and meaningful unit of text, the RecursiveCharacterTextSplitter helps improve the quality and relevance of the information retrieved and used in the generation process.

Q19: How does the `add_start_index=True` parameter benefit the document splitting process, and why is this feature useful in a RAG system?

A19: The `add_start_index=True` parameter is a feature that ensures the starting character index of each chunk is preserved as a metadata attribute called `start_index`. This feature is useful because it allows you to track the original location of each chunk within the full document. This metadata can be crucial for tasks that require mapping back to the original text or for understanding the context of the chunk within the larger document. For instance, if you need to reference the original document during retrieval or generation, knowing the exact position of a chunk can help in reconstructing the broader context. This feature enhances the traceability and interpretability of the data within a RAG system, contributing to more accurate and contextually grounded outputs.

Q20: What are the expected outputs when using the `split_documents()` method with RecursiveCharacterTextSplitter in LangChain, and how can these outputs be utilized?

A20: When using the `split_documents()` method with RecursiveCharacterTextSplitter in LangChain, the expected outputs are a list of smaller document chunks, each containing a portion of the original document's content. Additionally, the method provides metadata for each chunk, including the `start_index`, which indicates where the chunk begins in the original document. These outputs can be utilized in various ways:

- **Indexing:** The chunks can be indexed in a VectorStore for efficient retrieval.
- **Retrieval:** During a query, the most relevant chunks can be retrieved based on their content and metadata.

- **Context Preservation:** The metadata, such as `start_index`, allows for preserving the context and mapping back to the original document if needed.

These outputs are essential for ensuring that the document splitting process is effective and that the chunks are ready for subsequent embedding, retrieval, and generation tasks in the RAG system.

Q21: What is the purpose of embedding and storing document splits in a vector store, and how does this process facilitate efficient information retrieval?

A21: The purpose of embedding and storing document splits in a vector store is to enable efficient search and retrieval of information at runtime. Here's how the process works:

- **Embedding:** Each text chunk is converted into a high-dimensional vector (embedding) that captures the semantic meaning of the text. This embedding process is typically performed using an Embedding Model, such as OpenAIEmbeddings.
- **Storing:** The resulting vectors are stored in a vector store, a specialized database designed to handle and search through high-dimensional data efficiently.

When a query is made, it is also converted into an embedding, and a similarity search (such as cosine similarity) is performed to identify and retrieve the most relevant document splits based on their embeddings. This process facilitates quick and accurate retrieval of information, ensuring that the system can provide contextually appropriate answers to user queries.

Q22: Can you explain how cosine similarity is used in the context of a vector store, and why is it

an effective measure for retrieving relevant information?

A22: Cosine similarity is a measure used to determine the similarity between two vectors by calculating the cosine of the angle between them. In the context of a vector store, each document split is embedded into a high-dimensional vector. When a query is made, it is also embedded into a vector. Cosine similarity is then used to compare the query vector with the stored document vectors, identifying those with the smallest angles (i.e., most similar vectors). The most similar vectors correspond to the document splits that are most relevant to the query. Cosine similarity is an effective measure for retrieving relevant information because it focuses on the orientation of the vectors rather than their magnitude, making it particularly well-suited for capturing the semantic similarity between text representations. This allows the RAG system to retrieve

document chunks that are highly relevant to the user's query, even if the exact wording differs.

Q23: What are the key components involved in creating and querying a vector store in LangChain, and how do they interact?

A23: The key components involved in creating and querying a vector store in LangChain are:

- **Documents:** These are the text chunks that are converted into embeddings and stored in the vector store. Each document is typically a small, manageable portion of a larger dataset or document.
- **Embedding Model:** This model, such as OpenAIEmbeddings, is used to convert the text chunks into high-dimensional vector embeddings that capture the semantic meaning of the text.

- **Vector Store:** A storage mechanism like Chroma that holds the embeddings and allows for similarity searches. The vector store is designed to efficiently manage and search through large volumes of high-dimensional data.
- **Similarity Search:** The process of querying the vector store using an embedded query to retrieve the most relevant document splits based on cosine similarity or other distance measures.

These components interact to enable the RAG system to efficiently retrieve and process relevant information in response to user queries. The documents are first embedded and stored, and then when a query is made, the system uses similarity search to find and retrieve the most relevant documents, which are then used in the generation process.

Q24: What challenges might you encounter when using the Chroma vector store with LangChain, and how can these challenges be addressed?

A24: Several challenges might arise when using the Chroma vector store with LangChain:

- **Handling Document Splits:** Document splits must be converted into Document objects before being passed to the Chroma vector store. Ensuring the correct format and structure of these objects is essential for proper indexing and retrieval.
- **API Key Management:** Proper handling of API keys, especially for embedding models like OpenAIEmbeddings, is crucial. Errors can occur if the API key is not correctly set or passed, leading to issues in embedding and retrieval processes.

- **Understanding Methods and Parameters:** It's important to use the correct methods and arguments when interacting with Chroma. For example, understanding the difference between `n_results` and `top_k` for retrieving search results is vital for obtaining the desired output.

Addressing these challenges involves careful attention to data formats, API management, and thorough understanding of the available methods and their proper usage. Referring to the official documentation and examples can also help in navigating these challenges effectively.

Q25: How do you inspect the contents of a vector store in LangChain, and why is this inspection necessary?

A25: To inspect the contents of a vector store in LangChain, you can use the `_collection.count()` method to retrieve the number of

stored documents. Additionally, you can perform a dummy query using `similarity_search` to retrieve and inspect the content of the stored document chunks. Since there isn't a direct method to access documents by index in the Chroma vector store, these techniques help in understanding what is stored in the vector database. This inspection is necessary to ensure that the data has been correctly embedded and stored and to verify that the retrieval process will function as intended. By inspecting the contents, developers can confirm that the correct documents are being indexed and that they contain the relevant information needed for accurate and contextually appropriate responses in the RAG system.

Q26: What steps should you follow to embed and store document splits in a vector store using LangChain, and why is each step important?

A26: To embed and store document splits in a vector store using LangChain, follow these steps:

1. **Prepare the Document Splits:** Convert the text chunks into Document objects, where each object contains the text of a chunk. This step is important for structuring the data in a way that is compatible with the vector store.
2. **Set Up the Embedding Model:** Use an embedding model like OpenAIEmbeddings, ensuring that the API key is correctly set or passed. This step is crucial for generating the vector embeddings that will be stored in the vector store.
3. **Embed and Store:** Use the `Chroma.from_documents` method to embed the document splits and store them in the Chroma vector store. This step ensures that the data is properly indexed and can be efficiently retrieved based on its semantic content.

4. **Query the Store:** Use `similarity_search` to perform a similarity search on the stored embeddings with a given query to retrieve the most relevant document chunks. This step is essential for validating the effectiveness of the embedding and storage process and for enabling the RAG system to generate accurate and contextually relevant responses.

Each step is important because it ensures that the document splits are correctly processed, embedded, and stored, enabling the RAG system to function efficiently and effectively.

Q27: Can you explain the role of a Retriever in the LangChain RAG pipeline, and how does it contribute to the system's ability to generate relevant responses?

A27: A Retriever in the LangChain RAG pipeline is an object responsible for retrieving relevant documents based on a given

text query. It wraps an index, such as a VectorStore, which stores document embeddings. When a query is made, the Retriever uses similarity search or other techniques to identify and return documents that are most relevant to the query. This step is crucial in a RAG pipeline because it determines which documents will be passed to the language model for generating a final answer. The effectiveness of the Retriever directly impacts the quality of the generated response, as it ensures that the model has access to the most relevant and contextually appropriate information when formulating its answer.

Q28: How does the VectorStoreRetriever work in LangChain, and what is its primary function within the RAG pipeline?

A28: The VectorStoreRetriever in LangChain is a specific type of retriever that leverages the similarity search capabilities of a VectorStore. A VectorStore contains document embeddings,

which represent the documents in a high-dimensional space. The VectorStoreRetriever searches through these embeddings to find the documents most similar to the input query. Its primary function within the RAG pipeline is to retrieve the top relevant documents that will then be passed to a language model for further processing, such as generating answers to questions. This retriever is essential for ensuring that the model receives the most relevant information, enabling it to produce accurate and contextually appropriate responses.

Q29: Why is it important to inspect the content of retrieved documents in a LangChain RAG pipeline, and what benefits does this practice offer?

A29: Inspecting the content of retrieved documents is crucial in a LangChain RAG pipeline to ensure that the retrieval process is functioning as intended. By examining the documents returned by the retriever, you can verify that they are relevant to the input

query and contain the information needed for the model to generate accurate and contextually appropriate answers. This practice helps in fine-tuning the retrieval process and improving the overall performance of the application. It also allows developers to identify and address any issues with the retrieval process, such as retrieving irrelevant or incomplete data, which could negatively impact the quality of the generated responses.

Q30: What is the purpose of integrating retrieval and generation in a LangChain application, and how does this integration enhance the system's capabilities?

A30: The purpose of integrating retrieval and generation in a LangChain application is to build a pipeline that retrieves relevant documents based on a query and then generates an answer or output using a language model. This integration allows for creating sophisticated question-answering systems where the generation of responses is informed by specific, relevant content

retrieved from a knowledge base or document store. By combining these two processes, the system can generate responses that are both accurate and contextually grounded, making it more effective at handling complex queries and providing detailed, reliable answers.

Q31: How does LangChain's Runnable protocol contribute to building a retrieval and generation chain, and what advantages does it offer for developers?

A31: LangChain's Runnable protocol provides a flexible and standardized interface for creating custom chains that integrate various components, such as retrieval and generation. By leveraging this protocol, developers can easily build a pipeline where the output of one step (e.g., document retrieval) becomes the input for the next step (e.g., prompt construction and generation). This modular approach simplifies the creation of complex workflows and allows for the seamless integration of

retrieval and generation within a single chain. The Runnable protocol offers developers the advantage of flexibility and customization, enabling them to design and implement tailored solutions that meet the specific needs of their applications.

Q32: What is the role of the `gpt-3.5-turbo` model in the retrieval and generation chain, and why is it a suitable choice for this task?

A32: The `gpt-3.5-turbo` model is used as the language model responsible for generating answers or outputs based on the prompt constructed from the retrieved documents. After relevant documents are retrieved and processed, the `gpt-3.5-turbo` model takes the prompt and generates a coherent and contextually appropriate response. This model is known for its efficiency and effectiveness in generating natural language outputs, making it suitable for tasks like question-answering, where high-quality and relevant responses are required. Its ability to produce fluent

and contextually aware text makes it an ideal choice for integration in a RAG pipeline, where the quality of the generated output is paramount.

Q33: How can you customize a RAG chain using LangChain's built-in and custom components, and what benefits does this customization provide?

A33: Customizing a RAG (Retrieval-Augmented Generation) chain in LangChain involves combining built-in components like retrievers and generators with custom logic. Developers can use the Runnable protocol to define the sequence of operations, specify how data flows between steps, and incorporate custom components for specific tasks. For instance, a custom retriever could be used to query a specific database, and a custom prompt constructor could be created to format the retrieved data in a particular way before passing it to the generation model. This customization provides several benefits, including the ability to

tailor the RAG chain to the specific needs of the application, optimize performance, and enhance the relevance and accuracy of the generated responses.

Q34: Why is it important to use context from retrieved documents in the generation process, and how does this practice improve the quality of the generated responses?

A34: Using context from retrieved documents in the generation process is important because it ensures that the generated responses are relevant, accurate, and grounded in specific information. This approach prevents the generation model from producing generic or uninformed answers by anchoring its output in real, retrieved content. By incorporating document context, the model can provide more precise and useful answers, especially in scenarios requiring detailed or specialized knowledge. This practice improves the quality of the generated responses by making them more specific, informed, and aligned

with the user's query, ultimately enhancing the effectiveness and reliability of the RAG system.