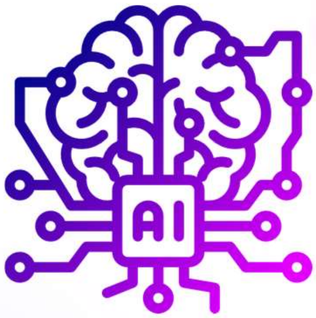
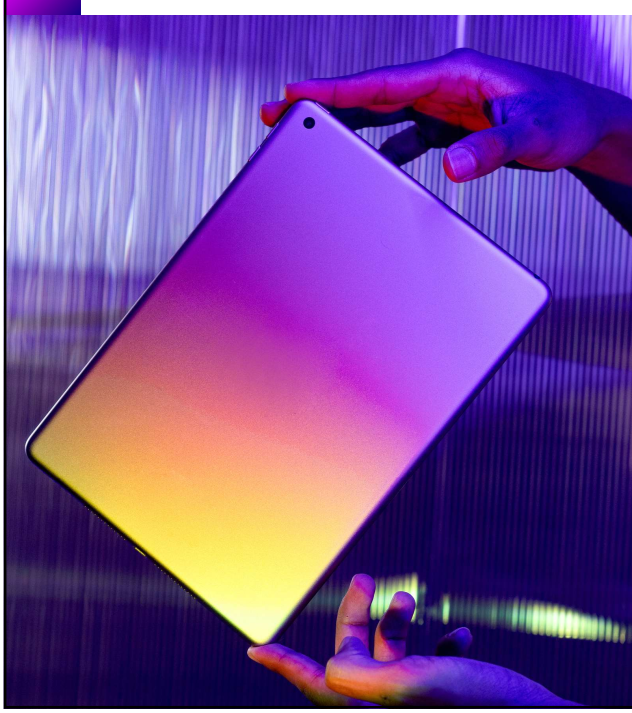


RETRIEVEL-AUGMENTED GENERATION

RAG

01



What is Retrieval-Augmented Generation (RAG)

- RAG is an approach that combines the power of retrieval-based methods and generation-based models to improve language model performance.
- The core idea is to retrieve relevant external information from a knowledge base (e.g., a document corpus) and use it to augment the response generation process.

02

Components of RAG

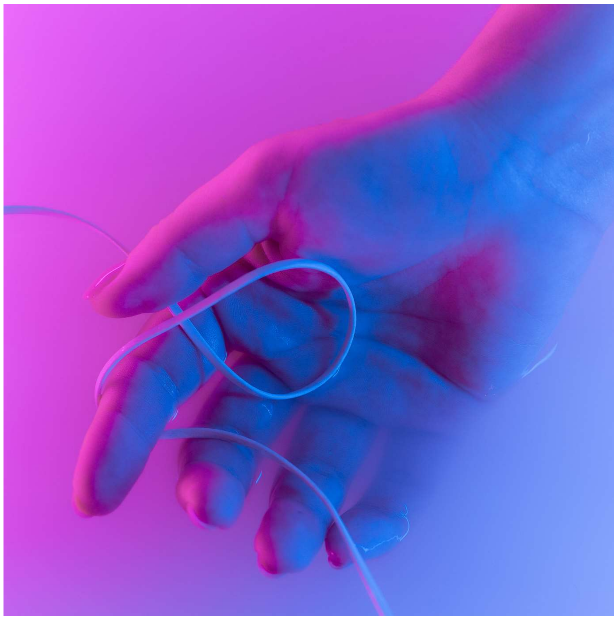
- **Retriever**

  A component responsible for retrieving relevant documents or information from an external knowledge base, based on the input query.

- **Technology**

  A generative model (e.g., GPT-like model) that generates responses using the retrieved information.

Key Idea: Instead of relying only on the training data of the language model, RAG uses an external dataset to enhance the model's output.



03

RAG Workflow

01 Input Query

The user provides a query.

02 Retrieval

The retriever searches for relevant documents or passages related to the query.

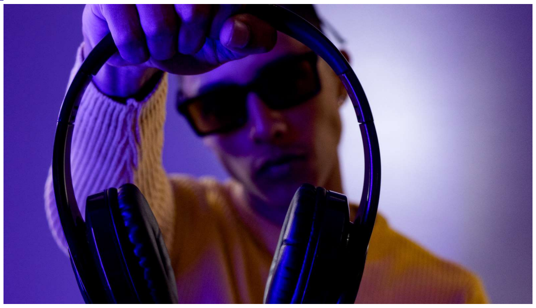
03 Augmented Input

Retrieved information is provided to the generative model.

04 Response Generation

The model generates a response based on the augmented input.

06



RAG Use Cases

Question Answering
Provide better answers by retrieving information from external databases

Text Summarization
Summarize content more accurately by pulling in relevant context from multiple documents.

Customer Support
Enhance automated agents by retrieving real-time information from knowledge bases and delivering tailored answers.

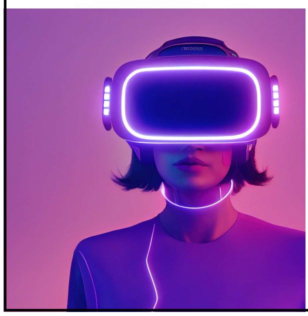
05

Benefits of RAG

01 **Improved Accuracy**
Augmenting responses with retrieved knowledge improves factual accuracy.

02 **Handling Out-of-Domain Queries**
When a model hasn't seen a specific topic during training, RAG allows it to pull in relevant information from external sources.

03 **Contextual Relevance**
RAG ensures that generated content remains highly relevant to the input query.



04

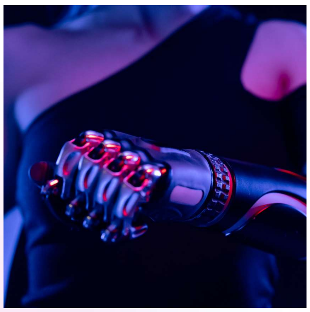
Future Trends

- **Improved Retrieval Mechanisms**

Advancements in search and ranking algorithms to better align with the generation model.

- **End-to-End Learning**

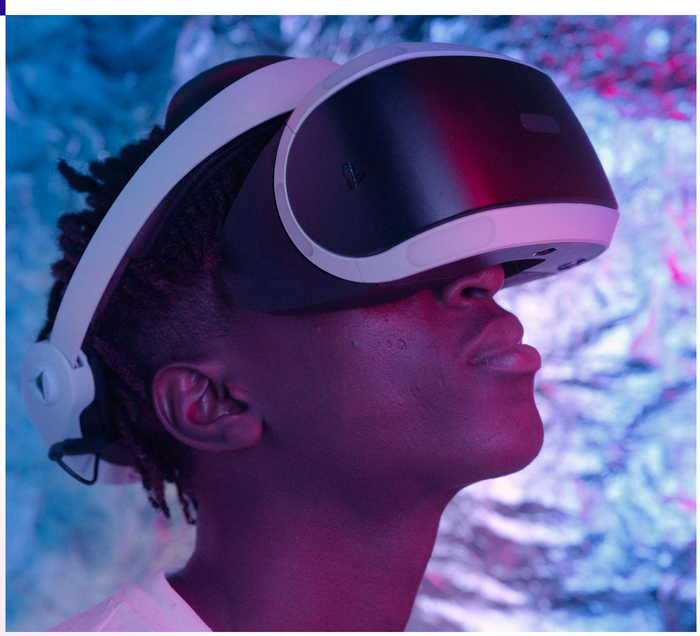
More seamless and efficient end-to-end models that combine retrieval and generation processes.



Challenges

- **Retrieval Quality**: The quality of retrieved information directly impacts the output.
- **Scalability**: Efficient retrieval mechanisms are needed for large-scale knowledge bases.
- **Combining Retrieval and Generation:** Balancing the integration of external knowledge and generative capabilities remains a challenge.

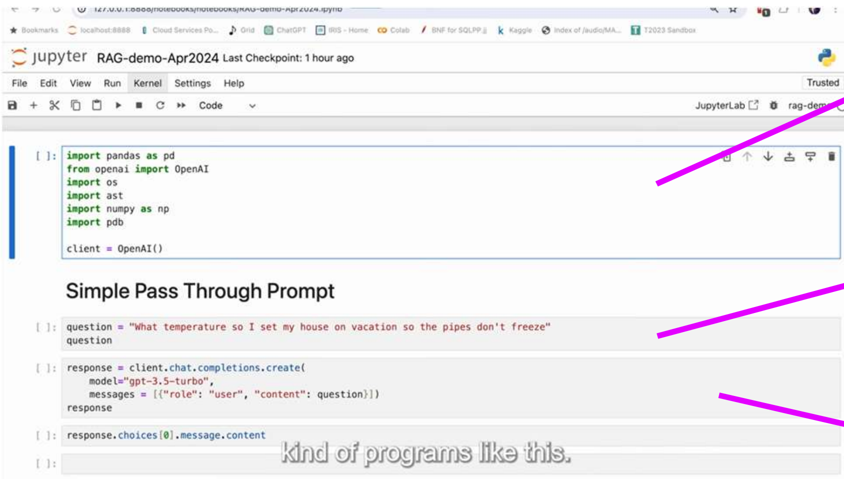
07



RAG IN PRACTICE

10

Using a Simple Pass Through Prompt

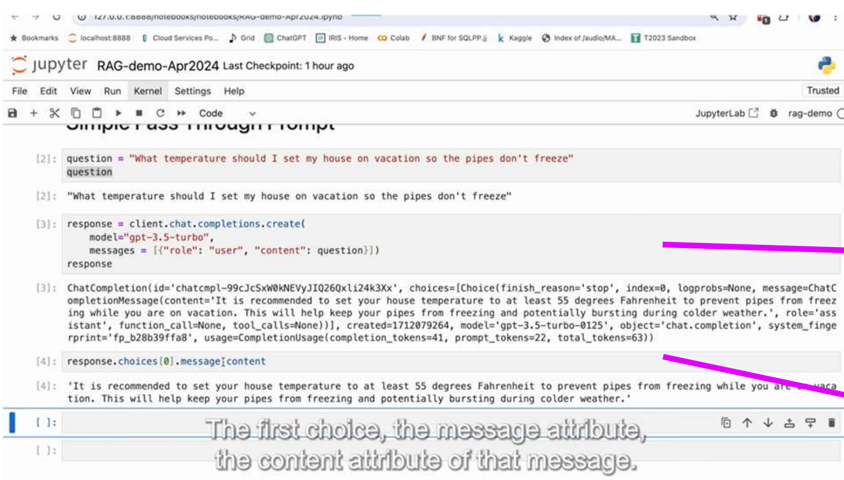


This block imports necessary Python libraries: pandas for data manipulation, openai for interacting with OpenAI models, os for OS interactions, ast for abstract syntax trees, numpy for numerical operations, and pdb for debugging. These provide tools for working with data, accessing LLMs, and general Python development.

question = ...: This line defines the user's question as a string.
question: This line simply displays the value of the question variable in the Jupyter Notebook output.

It sends a request to the GPT-3.5-turbo model for chat completion.
It stores the response containing the generated text, usage details, etc., in the response variable.

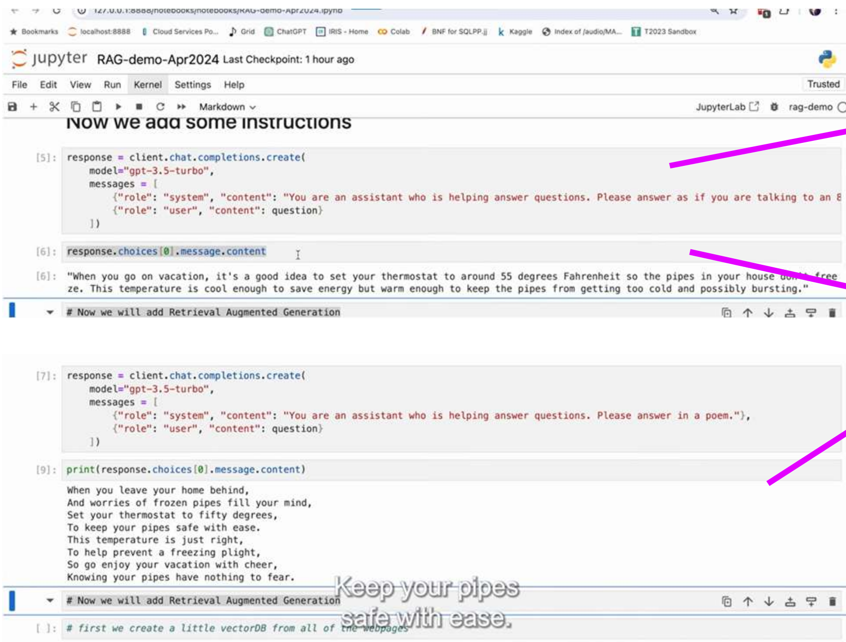
Here we are just trying to get response using GPT model by taking question



This code sends a request to the OpenAI API using the gpt-3.5-turbo model with the user's question.
The returned API response, containing the model's answer and metadata, is stored in the response variable and displayed.

This line accesses the first (and usually only) generated response from the list of choices.
It extracts the actual text content of the model's message from the response object and displays it.

Adding Instructions to your prompt

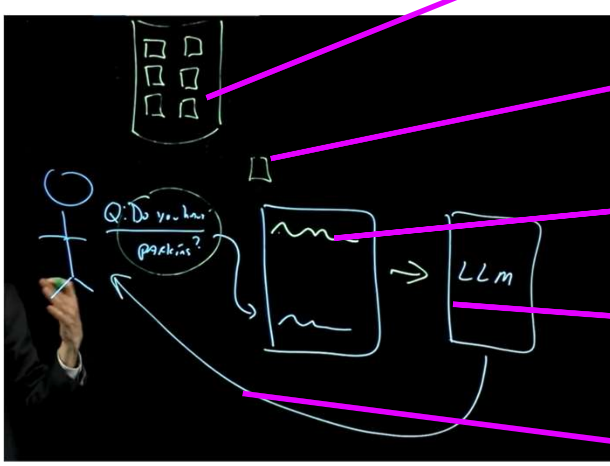


This code now includes a "system" message to instruct the LLM's behavior (act as if talking to an 8-year-old).
It sends the request to the GPT-3.5-turbo model with both the system instruction and the user's question.

It extracts and displays the text content of the model's message, now formatted according to the system instructions.

Setting up RAG

Webpages are stored as a database of text. This is the source of information.



Text is converted to numerical vectors (embeddings). These allow for semantic search.

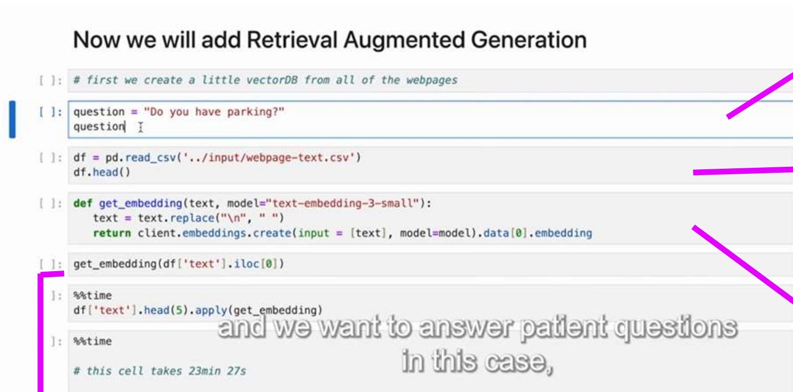
A user queries the chatbot (e.g., "Do you have parking?"). This starts the process.

The query and relevant text are sent to the LLM. It generates a context-aware response.

The query is also embedded, and similar embeddings (relevant text) are found.

A user asks a question; the system uses embeddings to find relevant text from a database; the question and retrieved text are then fed to an LLM to generate an answer.

Setting up Retrieval Augmented Generation



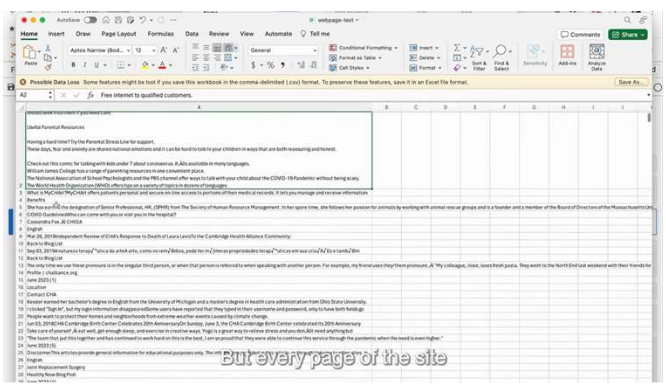
This sets the user's question about parking availability, which the RAG system will answer.
The question variable's value is displayed for confirmation.

This loads textual data from a CSV file (presumably containing web page content) into a pandas DataFrame. This DataFrame will serve as the knowledge base for retrieval.
df.head() displays the first few rows of the DataFrame to show a sample of the loaded data.

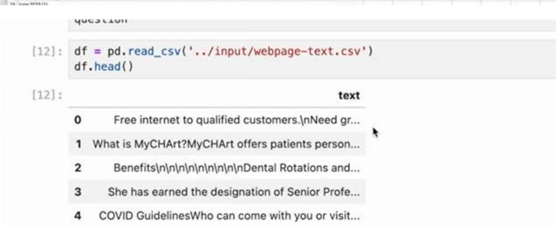
This defines a function get_embedding that converts text into a numerical vector representation (embedding) using OpenAI's embedding model (text-embedding-3-small in this case). The text.replace("\n", " ") part replaces newline characters with spaces, which is a common preprocessing step.

get_embedding(df['text'].iloc[0]) calculates and returns the embedding (vector representation) of the first text entry in the DataFrame df.

The CSV file which contain 7000 web pages and their content



Embeddings of each



Top 5 head files

```
Name: text, dtype: object
```

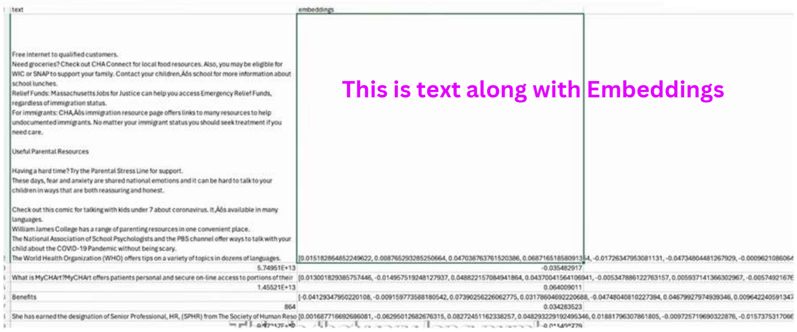
```
[ ]: %%time

     # this cell takes 23min 27s

     df['embeddings'] = df['text'].apply(get_embedding)
     df.to_csv('./website-with-embeddings.csv', index=False)
     df.to_pickle('./website-with-embeddings.pkl')
```

```
[ ]: %%time
     df = pd.read_pickle('./website-with-embeddings.pkl')
     df.head()
```

```
[ ]: question_embedding = get_embedding(question)
     question, question_embedding[0:10], "..."
```



This block calculates embeddings for all text entries in the DataFrame df using the get_embedding function and stores them in a new column called embeddings. The %%time magic command shows this process took 23 minutes and 27 seconds.
It then saves the DataFrame (including the newly generated embeddings) to both a CSV file (website-with-embeddings.csv) and a pickle file (website-with-embeddings.pkl). Saving to a pickle file is usually faster for loading later.

```
[18]: %%time
      df = pd.read_pickle('./website-with-embeddings.pkl')
      df.head()

      CPU times: user 316 ms, sys: 88 ms, total: 404 ms
      Wall time: 411 ms
```

[18]:

	text	embeddings
0	Free internet to qualified customers.\nNeed gr...	[0.015182864852249622, 0.008765293285250664, 0...
1	What is MyCHArt?MyCHArt offers patients person...	[0.013001829385757446, -0.014957519248127937, ...
2	Benefits\n\n\n\n\n\n\n\n\nDental Rotations and...	[-0.04129347950220108, -0.009159773588180542, ...
3	She has earned the designation of Senior Profe...	[0.001687716692686081, -0.06295012682676315, 0...
4	COVID GuidelinesWho can come with you or visit...	[0.009362754411995411, -0.03360440582036972, 0...

```
[19]: question_embedding = get_embedding(question)
      question, question_embedding[0:10], "..."

[19]: ("What temperature should I set my house on vacation so the pipes don't freeze",
       [0.026638785377144814,
        0.016546063125133514,
        -0.007697733584791422,
        -0.017371488735079765,
        -0.05767984688282013,
        0.006953598465770483,
```

So we want to get the embedding for the question.

This code loads a pre-computed set of embeddings from a pickle file named website-with-embeddings.pkl into a pandas DataFrame df. This is a crucial optimization because generating embeddings can be computationally expensive.
df.head() displays the first few rows of the DataFrame, showing the 'text' and corresponding 'embeddings' columns. The %%time magic command measures the time it takes to load the data

This line calculates the embedding for the user's question using the get_embedding function (which uses the OpenAI embedding API). This embedding represents the semantic meaning of the question in vector form. It then displays the original question along with the first 10 elements of the calculated question_embedding vector (and "..." to indicate it's truncated).

```
...,
[22]: def fn(page_embedding):
          return np.dot(page_embedding, question_embedding)
      df['distance'] = df['embeddings'].apply(fn)
      df.head()
```

[22]:

	text	embeddings	distance
0	Free internet to qualified customers.\nNeed gr...	[0.015182864852249622, 0.008765293285250664, 0...	0.167978
1	What is MyCHArt?MyCHArt offers patients person...	[0.013001829385757446, -0.014957519248127937, ...	0.154547
2	Benefits\n\n\n\n\n\n\n\n\nDental Rotations and...	[-0.04129347950220108, -0.009159773588180542, ...	0.173591
3	She has earned the designation of Senior Profe...	[0.001687716692686081, -0.06295012682676315, 0...	0.109746
4	COVID GuidelinesWho can come with you or visit...	[0.009362754411995411, -0.03360440582036972, 0...	0.239370

```
[ ]: df.sort_values('distance', ascending=False, inplace=True)
     df

[ ]: context = df['text'].iloc[0] + "\n" + df['text'].iloc[1] + "\n" + df['text'].iloc[2] + "\n" + df['text'].iloc[4] + "\n"
     print(context)
```

- This defines a function fn that calculates the dot product between a page embedding (page_embedding) and the previously calculated question_embedding. The dot product is used as a similarity metric; a higher dot product indicates greater similarity.
- It applies this fn to each embedding in the embeddings column of the DataFrame, storing the resulting dot products (similarities/distances) in a new column called distance. df.head() then displays the first few rows, showing the calculated distances.

- This sorts the DataFrame df in descending order based on the distance column (the dot product/similarity scores). inplace=True modifies the DataFrame directly.
- The sorted DataFrame is then displayed. The rows at the top now represent the text passages most similar to the user's question.

This section performs the core retrieval part of RAG:

- Calculates similarity: It uses the dot product to measure the similarity between the question embedding and each document embedding.
- Sorts by similarity: It sorts the documents by their similarity scores to find the most relevant ones.
- Creates context: It combines the text from the top most similar documents into a single context string.

```
    model="gpt-3.5-turbo",
    messages = [
        {"role": "system", "content": "You are an assistant who is helping the Cambridge Health Alliance (CHA) health system respond to
        {"role": "user", "content": question},
        {"role": "assistant", "content": f"Use this information from CHA's website as context to answer the user question: {context}. Pl
    ])

6]: response.choices[0].message.content

6]: 'Yes, we have parking available at our facilities. Our main entrance is on Revolution Drive in Somerville, across from Home Depot, and
    we have a free parking lot next to the building, though spaces are limited at peak hours. Additionally, we have another parking lot on
    Crown Street behind the hospital, which is recommended for certain services like Urgent Care, the OB/GYN Center, Surgery, or Medical Sp
    ecialties. The cost for parking ranges from $3 to $8, depending on the length of stay, and both parking lots are cash-only. We also off
    er shuttle services between our Cambridge Hospital, Somerville Campus, and other CHA locations.'
```

This code makes the final call to the OpenAI API using the gpt-3.5-turbo model. The key here is the structure of the messages:
System Message: Sets the overall persona of the LLM (an assistant for CHA).
Assistant Message (Context): This is where the retrieved context (from the previous steps) is injected into the prompt. The f-string formatting makes it easy to embed the context variable. This tells the LLM to use the provided information when answering the question.
User Message: Contains the original user question.
The API response is stored in the response variable.

his final section completes the RAG workflow:
Contextualized Prompt: It constructs a prompt that includes both the user's question and the retrieved relevant information.
LLM Generation: It uses the LLM to generate a response based on this contextualized prompt.
Display Response: It displays the final, contextually relevant answer to the user.

This entire process demonstrates the power of RAG. By combining the strengths of LLMs with the ability to retrieve and use external knowledge, it's possible to create much more accurate, informative, and grounded responses to user queries.