Introduction: In this guide, we will walk through how to build a Retrieval-Augmented Generation (RAG) chatbot from scratch. A RAG chatbot is a question-answering system that **retrieves information from a knowledge base and uses a language model to generate answers** file. This approach helps ground the chatbot in real data, reducing the chance of "hallucinations" (made-up answers) and improving factual accuracy file. We'll cover everything from collecting data (web scraping) and processing it (chunking and embeddings) to storing it in a vector database and hooking up a language model to generate responses. Along the way, we'll explain each concept in simple terms and provide Python code snippets. By the end, you'll have a basic QA chatbot and know how to evaluate it and deploy it.

What is Retrieval-Augmented Generation (RAG)?

Retrieval-Augmented Generation (RAG) is a technique that **improves a language model's responses by injecting external context into its prompt at runtime**. Instead of relying only on what the model was pre-trained on, the chatbot actively **retrieves relevant information** (such as documents or webpages) from a knowledge source and uses that to produce a more accurate, up-to-date answer. In other words, RAG is like an "open-book exam" for AI: the language model (LM) can **consult a reference** (the retrieval database) before answering, rather than answering purely from memory. This greatly **reduces nonsense answers** and **improves factual consistency**file-cx8tjw31rllphv64rncyec.

Why RAG? Large language models are very powerful but can sometimes generate incorrect or made-up information (we call these *hallucinations*). By combining retrieval + generation, we **ground the model's output in real data**. For example, if you ask a RAG chatbot *"When was our university founded?"*, it will search the university's documents for the founding date and then answer based on that data, instead of guessing. This leads to more reliable answersfile-cx8tjw31rllphv64rncyec.

How RAG works (overview): The chatbot will follow a pipeline like *"Question → Retrieve → Answer"*. In implementation terms, we usually break this down into a series of stepsfile.

- **Web Scraping & Data Collection:** Gather the reference text (e.g. university website pages) to build a knowledge base.
- **Chunking & Preprocessing:** Split the text into smaller chunks for easier handling and context preservation.
- **Embeddings:** Convert each text chunk into a numerical vector representation that captures its meaning.
- **Vector Store (Database):** Store all these vectors in a database that can quickly find which chunks are relevant to a new question.
- **Retrieval:** When a user asks something, convert the question to a vector and **query the vector database** for similar vectors (i.e. relevant chunks).
- **Generation (LLM):** Feed the retrieved text chunks + the user's question into a language model, which then generates a final answer.
- **Iteration & Improvement:** (Optional) Log the results, evaluate the answers, and refine the system (e.g., add more data, adjust parameters).

Each step involves different tools and concepts, which we'll explain one by one in a beginner-friendly way. We'll also include **fun analogies** and **challenges** to deepen your understanding.

Analogy: Think of the RAG chatbot like **a smart librarian**. When you ask a question, the librarian first searches books in the library (retrieval) and then uses the relevant pages to compose a helpful answer (generation). The books are the external knowledge, and the librarian's skill in weaving an answer is the language model.
Fun Fact: The RAG approach was popularized to help large models stay factual. Even big chatbots like Bing Chat and ChatGPT's plugins use a form of retrieval to provide up-to-date info!
Optional Challenge: Before building anything, try to write down in your own words how *you* would answer questions using a stack of articles. This will give your insight into the retrieval + generation process.

Step 1: Web Scraping – Collecting the Knowledge Base

The first step in building our chatbot is to **collect a set of documents or pages** that contain the information our chatbot might need. Often, this means scraping a website for relevant content. **Web scraping is the process of extracting data from websites** using an automated program. In our case, we might scrape a university's website (e.g. course pages, FAQ, about page) to build a knowledge base for a university Q&A chatbot.

How web scraping works: Imagine copying and pasting text from a webpage but done automatically with code. We'll use Python tools to download the HTML of pages and pull out the text we care about. Two common tools are **`requests`** (to fetch the page content) and **BeautifulSoup** (to parse the HTML). **Beautiful Soup is a Python library for pulling data out of HTML and XML files**. It helps navigate the HTML tree (which is full of tags like `<p>` for paragraphs, `<a>` for links, etc.) and extract the text.

Plan for scraping:

1. Identify the pages to scrape. Often, websites have a sitemap (e.g. `sitemap.xml`) listing all pages. In our project, we scraped a university sitemap to get all page URLsfile-cx8tjw31rllphv64rncyec. For simplicity, you might start with just a few specific pages.
2. Use `requests.get(page_url)` to download the HTML content.
3. Use `BeautifulSoup` to parse the HTML and extract visible text. We should remove navigation menus, scripts, and other boilerplate so we only keep main content (this is the "cleaning" part).
4. Save the cleaned text for each page, perhaps as a plain text file or in memory.

Let's see a simple example of scraping and cleaning one page:

```
python
CopyEdit
import requests
from bs4 import BeautifulSoup
```

```
url = "https://example-university.edu/about"  # example page to scrape
response = requests.get(url)
html = response.text

soup = BeautifulSoup(html, "html.parser")

# Remove unwanted tags like scripts, style, navigation, footer, etc.
for tag in soup(["script", "style", "nav", "footer", "header"]):
    tag.decompose()  # remove these tags from the soup

text = soup.get_text(separator=" ")  # get visible text
text = text.strip()
print(text[:500])  # print first 500 chars of cleaned text
```

In this snippet, we fetched the page and removed script/style and typical navigation/footer sections to clean the content. We then get the text and strip extra whitespace. In a real scenario, you'd loop over many URLs (perhaps from a sitemap list) and aggregate all the text data.

Storing the data: You can store each page's text in a separate file (e.g., `about.txt`, `admissions.txt` etc.) or in a list of strings in code. This raw text will feed into the next steps. Make sure to note which page each text came from (keeping an ID or filename), so later we can reference the source if needed.

Analogy: Web scraping is like **using a vacuum cleaner on a website** – it sucks up all the dirt and dust (raw HTML) and then you filter out the shiny bits you want (the text content).
Fun Fact: Web scraping needs to be done responsibly. Many sites have a `robots.txt` file that tells scrapers which pages they can or cannot crawl. Always check a site's policy to avoid breaking rules!
Optional Challenge: Try modifying the above code to scrape a simple site of your choice (maybe Wikipedia's page on your school or a favorite topic). Extract the text and see if you can save it to a file. This will give you practice with `requests` and `BeautifulSoup`.

Step 2: Chunking – Splitting Data into Manageable Pieces

Once we have lots of text data (potentially thousands of words from various pages), we face a new problem: **language models have limits on how much text they can handle at once**. We can't feed an entire website's text into the model in one go. The solution is **chunking**: breaking the content into smaller pieces (chunks) that are easier to work with.

What is chunking? In the context of building LLM applications, **chunking is the process of breaking down large pieces of text into smaller segments**. Each chunk should be of a size that makes sense on its own. A good rule of thumb is that a chunk should be like a standalone paragraph or section that still has meaning without too much outside context. If chunks are too large, they may overflow the model's input limits; if too small, the model might not get enough context to be useful.

In our project, we **chunked text with overlap to preserve context** file. Overlap means the end of one chunk is repeated at the start of the next, so that if an important sentence falls on a boundary, it appears in both chunks. This helps maintain continuity of context between chunks.

How to chunk text: We can do this manually or use libraries. One handy library is **LangChain**, which provides utilities for text splitting. (LangChain is a framework for building LLM apps that helps chain together components like splitting, prompting, etc. **LangChain simplifies every stage of the LLM application lifecycle**, but you can also write your own splitter.)

Here's an example using LangChain's text splitter:

```
python
CopyEdit
!pip install langchain tiktoken  # make sure to install langchain
from langchain.text_splitter import RecursiveCharacterTextSplitter

# Suppose 'text' is a very long string containing our scraped content.
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size = 800,    # target 800 characters (approximately tokens) per
chunk
    chunk_overlap = 200  # overlap of 200 characters between chunks
)
chunks = text_splitter.split_text(text)
print(f"Created {len(chunks)} chunks.")
print(chunks[0][:200])  # preview first 200 chars of the first chunk
```

In this code, we aimed for ~800 tokens per chunk with an overlap of 200 tokens (these were the values used in our university project). The `RecursiveCharacterTextSplitter` tries to split at natural breakpoints (like paragraph ends or sentence ends) but ensures no chunk exceeds 800 characters, overlapping 200 chars with the next chunk. If you don't want to use LangChain, you could split by sentences or paragraphs using Python string methods or regex, and then trim chunks to a max length.

After chunking, instead of one huge text blob per page, you might have dozens of smaller chunks for that page. Each chunk can be treated as a separate piece of knowledge for the chatbot.

Organizing chunks: It's helpful to keep track of which page or source each chunk came from. You could store chunks in a list of dicts like `{"text": chunk_text, "source": "about.html"}` or maintain parallel lists. This way, if the chatbot uses a chunk to answer a question, you know where that info originally came from.

Analogy: Chunking is like **slicing a big pizza into small slices** – each slice (chunk) is easier to handle and eat, but some toppings (context) might span multiple slices, so you intentionally overlap them a bit to not miss anything.
Optional Challenge: Take a long article (maybe 5-10 paragraphs) and write your own code to split it into chunks of 100 words each, with an overlap of 20 words. Print the chunks and verify that overlapping words appear in consecutive chunks. This will demonstrate how overlapping context works.

Step 3: Embeddings – Representing Text as Vectors

Now that we have our text split into many chunks, the next step is crucial: we need a way to **compare the user's question with our chunks to find which chunks are relevant**. Computers can't directly "understand" text, but they can work with numbers. This is where **embeddings** come in.

What is an embedding? An **embedding** is a numerical representation of a piece of data (like text) that captures its meaning. In NLP, a **text embedding is a piece of text projected into a high-dimensional space; the position of the text in this space is a vector (a long list of numbers)**. Think of it like coordinates that represent the gist of the text. If two texts have similar content, their vectors will be closer in that space.

For example, the embedding for "student enrollment deadline" will be closer to the embedding for "registration cutoff date" than to "campus dining menu", because the first two are about a similar topic. The magic of embeddings is that they capture *semantic meaning*: even if words are different, if the idea is similar, the vectors will reflect that.

To get embeddings, we use a pre-trained model (often a smaller **transformer** model) that converts text to vectors. A popular choice is **SentenceTransformers** library which provides many embedding models. In our project, we used a model called **all-MiniLM-L6-v2**, which produces a 384-dimensional vector for each text chunk file. (384 dimensions means each text is turned into a list of 384 numbers.)

Generating embeddings with SentenceTransformers:

```
python
CopyEdit
!pip install sentence-transformers   # ensure the library is installed
from sentence_transformers import SentenceTransformer

# Load a pre-trained embedding model
model = SentenceTransformer('all-MiniLM-L6-v2')  # small, fast embedding
model

# Suppose we have a list of text chunks from the previous step
embeddings = model.encode(chunks)  # this returns a list of vectors
print(f"Embedding of first chunk is a vector of length
{len(embeddings[0])}.")
print(embeddings[0][:10])  # print first 10 dimensions of the first chunk's
vector
```

When you run this, the model will download if not already present. Then `embeddings` will be a list of numpy arrays (or Python lists) of numbers. Each corresponds to one chunk in `chunks`, in the same order. You'll see each embedding is indeed length 384 (for MiniLM). These numbers might look meaningless to us, but in this 384-dimensional space, chunks about similar topics will cluster together.

Now we have transformed our text data into a form that's ready for efficient searching. Instead of searching by keywords, we can search by vector similarity – which often finds relevant info even if wording differs.

Step 4: Building a Vector Store – Storing and Searching the Chunks

We have our chunk embeddings; now we need a system to **store these vectors and query them efficiently**. This is where a **vector database** (vector store) comes into play. A regular database is good at exact matches (like look up a record by ID), but here we need "find the *nearest* vectors to this query vector". A vector store is optimized for **similarity search** among high-dimensional vectors.

What is a vector database? *A vector database indexes and stores vector embeddings for fast retrieval and similarity search*. It can handle operations like "give me the top-5 most similar vectors to this query vector" very quickly, even if you have millions of vectors. In our case, we'll use **ChromaDB**, which is an open-source vector database designed for LLM applications.

Think of the vector store as a **knowledge base** for the chatbot: each entry is a chunk of text (with its embedding). When a question comes in (also converted to an embedding), the database will return the IDs of the chunks that are most relevant. We can then retrieve those chunks' text to help answer the question.

Using ChromaDB: First, install and initialize the Chroma client:

```
python
CopyEdit
!pip install chromadb
import chromadb

# Initialize Chroma (in-memory for now; it can also persist to disk)
client = chromadb.Client()
collection = client.create_collection("univ_knowledge")  # create a new
collection of vectors
```

Now we can add our data. We need to provide the chunks, their embeddings, and an id for each:

```
python
CopyEdit
# Assume chunks (list of texts) and embeddings (list of vectors) are ready
from previous steps
ids = [f"chunk{i}" for i in range(len(chunks))]
# Add to the Chroma collection
collection.add(documents=chunks, embeddings=embeddings, ids=ids)
print(f"Added {collection.count()} chunks to the vector store.")
```

We now have all chunk vectors indexed in the collection. The collection can be queried by vector or by text (Chroma will embed the text for you if you give it raw text and specify a model). Since we already have an embedding model, we'll use it directly:

```
python
CopyEdit
question = "When was the university founded?"  # user query example
# Embed the question using the same model as before
q_embedding = model.encode([question])[0]

# Query the vector store for the most similar chunks
results = collection.query(query_embeddings=[q_embedding], n_results=5)
# 'results' will contain the IDs, distances, and documents of the top 5
closest chunks
for doc, dist in zip(results['documents'][0], results['distances'][0]):
    print(f"Similarity: {1-dist:.2f}, Chunk: {doc[:60]}...")
```

We request the top 5 nearest chunks (`n_results=5`). Many vector DBs return a distance or similarity score – Chroma by default returns a distance (lower is more similar). Here we print a similarity (1-distance for simplicity) along with a snippet of each chunk. These chunks should be those whose content closely relates to the question. For example, if the question is about founding date, likely a chunk from the "History" or "About" page containing the founding year will appear with high similarity.

Step 5: Integrating the Language Model – Generating Answers

We've reached the most exciting part: using a **language model (LLM)** to turn the retrieved information into a coherent answer for the user. Up until now, our pipeline can fetch relevant text for a question, but it can't formulate a nice answer sentence – that's the LLM's job.

Recap so far: User question → embed question → find top relevant chunks (texts). Now: **feed those chunks + question into an LLM** → get answer.

Choosing a language model: A **Large Language Model (LLM)** is a type of AI model that can recognize and generate human-like text. Examples include GPT-3, GPT-4, or open-source ones like LLaMA 2, GPT-Neo, etc. For a high school project, you have a few options:

- Use an **online API** like OpenAI's GPT-3.5 (ChatGPT) if you have an API key. This is easy and the model is very strong, but it may cost money and requires internet access.
- Use a **local open-source model** if you have the compute resources. For instance, smaller models like **Mistral 7B or LLaMA-2 7B** can run on a decent GPU or even on CPU (slowly). In our university project, we integrated with a local LLM server called *Ollama* to run models like Mistral, LLaMA2, etc. on our own machinefile.
- Use a smaller pre-built QA model. HuggingFace's `transformers` library has some Q&A pipelines (like for extractive QA), but for generation with context, a full LLM is usually needed.

For learning purposes, we can illustrate using OpenAI's API (assuming you have an API key set in an environment variable for example). We will provide the retrieved context to the model in the prompt. A simple prompt format could be:

```
css
CopyEdit
Use the following context to answer the question.

Context:
{top_chunks_text}

Question: {user_question}
Answer:
```

The model will hopefully use the context to produce the answer after "Answer:".

Example with OpenAI API (pseudo-code):

```
python
CopyEdit
import openai
openai.api_key = "YOUR_API_KEY"  # ensure your API key is set

# Combine top chunks into one context string
context = "\n\n".join(results['documents'][0])  # join the 5 chunks with some
spacing

prompt = f"Use the following context to answer the
question.\n\nContext:\n{context}\n\nQuestion: {question}\nAnswer:"
response = openai.Completion.create(
    engine="text-davinci-003",  # or use 'gpt-3.5-turbo' with
openai.ChatCompletion
    prompt=prompt,
    max_tokens=200,
    temperature=0  # 0 for more factual answers
)
answer = response['choices'][0]['text'].strip()
print("Answer:", answer)
```

If using the ChatGPT API (`gpt-3.5-turbo` or `gpt-4`), you'd format it as a system message and user message instead, but the idea is the same: feed context, ask question. The `temperature=0` makes the model deterministic and focused (good for factual answers).

If you prefer not to use an API, you could try a local model using `transformers` library:

```
python
CopyEdit
from transformers import AutoModelForCausalLM, AutoTokenizer

model_name = "google/flan-t5-small"  # a smaller model for demo (though flan-
t5 is an encoder-decoder)
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)
```

```
input_text = f"{context}\n\nQuestion: {question}\nAnswer:"
inputs = tokenizer(input_text, return_tensors="pt")
outputs = model.generate(**inputs, max_new_tokens=100)
answer = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(answer)
```

Keep in mind small models like `flan-t5-small` might not produce great answers or might not handle long context well. Larger models would work better but are more resource-intensive. If resources are limited, using an API might be the practical choice.

However you do it, the LLM will produce an answer string. This answer is final output of your chatbot for the user's query.

To improve the answer quality, you can do a few extra things:

- **Formatting context:** Maybe add bullet points or section titles in the context to make it clear. Or limit to the most relevant sentences.
- **Prompting technique:** Instruct the model to only use the given context and not make up anything else. You can say: *"If the answer is not in the context, say you don't know."* to make it safer.
- **Post-processing:** You might want to cite sources. This is advanced, but you could have the model include the chunk ID or reference in the answer, or you manually append which documents were used. For now, focus on getting a correct answer.

At this stage, you essentially have a functional RAG chatbot: ask a question, it retrieves info and answers using that info. You can wrap this logic in a loop to handle multiple queries, or better, make a web service or chat interface (next step).

Step 6: Building a Chatbot Interface (FastAPI)

Now that the core logic works in a script or notebook, you might want to turn this into an **interactive chatbot** that others can use. One way is to create a simple web API for it. **FastAPI** is a great Python framework for this because it's easy and automatically creates a web UI for testing your endpoints. *FastAPI is a modern, high-performance web framework for building APIs with Python*.

We'll create a basic API with a single endpoint (say, `/ask`) that accepts a question and returns the answer. This way, you could build a frontend (a simple webpage) that calls this API or use command-line to query it.

FastAPI setup:

```
python
CopyEdit
!pip install fastapi uvicorn[standard]
from fastapi import FastAPI
from pydantic import BaseModel
```

```
app = FastAPI()

# Define a request schema for question input
class Question(BaseModel):
    query: str

# Define the /ask endpoint
@app.post("/ask")
def ask_question(q: Question):
    question = q.query
    # 1. Embed the question
    q_embedding = model.encode([question])[0]
    # 2. Retrieve relevant chunks
    results = collection.query(query_embeddings=[q_embedding], n_results=5)
    context = "\n".join(results['documents'][0])
    # 3. Generate answer using LLM (here using a placeholder for brevity)
    answer = generate_answer_with_model(question, context)  # you would
implement this
    return {"question": question, "answer": answer}
```

In the above code, we define a FastAPI app with a POST endpoint `/ask`. We create a Pydantic model `Question` to parse the JSON input which should have a field `query`. When someone calls this endpoint with a question, the function will embed the question, retrieve top chunks from `collection` (which we assume is globally accessible or loaded when app starts), then generate an answer (you would use the OpenAI API or local model as implemented earlier inside `generate_answer_with_model`). Finally, it returns a JSON with the question and answer.

To run this API, you would use Uvicorn (an ASGI server):

```
bash
CopyEdit
uvicorn myapp:app --reload --port 8000
```

Where `myapp.py` contains the code above. The `--reload` is useful during development to auto-restart on code changes.

Once running, you can go to `http://localhost:8000/docs` in a browser. FastAPI provides an interactive docs interface where you can test the `/ask` endpoint by inputting a question and executing, seeing the JSON response. This is super helpful for quick testing! You can also send requests with `curl` or from a simple frontend.

Alternate interfaces: If you don't want to set up a web server, you could also create a simple command-line interface that prompts the user for input in a loop and prints answers. Or use a library like Gradio or Streamlit to make a quick web UI. For example, Gradio can create a chat interface in a few lines, calling your `ask_question` logic under the hood.

The key is that our backend logic is neatly contained, so hooking up any interface is straightforward.

Analogy: Think of FastAPI as **creating a phone line** to your chatbot. Instead of you calling functions in code directly, now anyone can call your chatbot by dialing the API endpoint with a question, and the chatbot answers on the line (returns JSON).

Step 7: Evaluation – How to Measure Your Chatbot's Performance

After building the chatbot, it's important to **evaluate** how well it's answering questions. Evaluation helps identify if your improvements actually make the bot better and if it's reliable for real use. We can evaluate chatbot responses in two main ways: **qualitative** (manually judging correctness) and **quantitative** (using metrics).

For quantitative evaluation of QA or generated text, common metrics borrowed from machine translation and summarization tasks are **BLEU** and **ROUGE**. Our project specifically used BLEU and ROUGE-L to gauge answer qualityfile-cx8tjw31rllphv64rncyec.

- **BLEU (Bilingual Evaluation Understudy):** Despite the long name, BLEU basically measures the overlap of n-grams (contiguous word sequences) between the generated answer and a reference answerfile-cx8tjw31rllphv64rncyec. It was originally designed to evaluate machine-translated text by comparing it to a human translation. **BLEU is an algorithm for evaluating the quality of text that has been machine-translated from one language to another**, but it's often used for any generated text. A BLEU score ranges from 0 to 1 (or 0 to 100 if scaled), where higher means closer to the reference. For example, if the reference answer is "The university was founded in 1910" and your bot says "It was founded in 1910," they share many 1-gram, 2-gram overlaps ("founded in 1910"), yielding a high BLEU.
- **ROUGE (Recall-Oriented Understudy for Gisting Evaluation):** ROUGE is a set of metrics (ROUGE-1, ROUGE-2, ROUGE-L, etc.) commonly used to evaluate summaries. It checks how much of the *reference* text appears in the generated text (recall-focused) **The ROUGE metric measures how well generated summaries or translations compare to reference outputs**. For QA, ROUGE-L (which looks at longest common subsequence overlap) is often used as a recall metric. In our project, ROUGE-L scores indicated how much of the answer's content matched the ideal answerfile-cx8tjw31rllphv64rncyec.

In simpler terms, **BLEU focuses on precision (did the model say correct phrases?) and ROUGE focuses on recall (did the model cover all the important parts of the answer?)**file. We even combined them by averaging to get a balanced score file.

Using these metrics: To compute these, you need a set of test questions and **reference answers** (the ideal correct answers, perhaps from a FAQ or your own created answer key). Then for each question, have the bot generate an answer and compare to the reference.

There are libraries to compute BLEU/ROUGE:

- The `nltk` library has `nltk.translate.bleu_score`.
- The `rouge_score` library or `datasets` from HuggingFace can compute ROUGE easily.
- `evaluate` (by HuggingFace) can load BLEU and ROUGE metrics as well.

Example with `nltk` BLEU (assuming one reference per question):

```
python
CopyEdit
import nltk
from nltk.translate.bleu_score import sentence_bleu, SmoothingFunction

references = ["The university was founded in 1910."]
candidate = "The university was founded in 1910"
# BLEU expects references as list of list of tokens, candidate as list of
tokens
ref_tokens = [references[0].split()]
cand_tokens = candidate.split()
score = sentence_bleu(ref_tokens, cand_tokens,
smoothing_function=SmoothingFunction().method1)
print(f"BLEU score = {score:.2f}")
```

We apply a smoothing function because short sentences can otherwise get BLEU 0 or 1 unfairly. BLEU score here might come out as 1.0 (or 100) if it matches exactly.

Example with ROUGE (using `rouge_score` library):

```
python
CopyEdit
!pip install rouge-score
from rouge_score import rouge_scorer

scorer = rouge_scorer.RougeScorer(['rougeL'], use_stemmer=True)
scores = scorer.score(references[0], candidate)
print(f"ROUGE-L precision = {scores['rougeL'].precision:.2f}, recall =
{scores['rougeL'].recall:.2f}, F1 = {scores['rougeL'].fmeasure:.2f}")
```

This will give ROUGE-L scores. In our case of identical text, precision and recall would be 1.0.

You would average these scores over a set of Q&A pairs to get an overall sense. In our project, we observed BLEU ~0.70 and ROUGE-L ~0.72 for well-answered questions file These are pretty high, indicating the answers had a lot of overlap with expected answers (good factual accuracy).

Limitations of automatic metrics: BLEU and ROUGE don't fully capture the quality of an answer. Sometimes an answer can be correct but phrased differently than the reference (low overlap), or it might overlap a lot but still be missing something subtle. Use these metrics as a guide, but also **manually review answers**. For a high school project, you could prepare, say, 10 questions about the content you scraped, write down the correct answers, and then test your bot. You'd likely report both the quantitative metrics (if computed) and some examples of correct/incorrect behavior.

Qualitative evaluation: You might categorize errors (e.g., “fails on date questions”, “sometimes mixes two similar concepts”). This helps in thinking about future improvements.

Step 8: Monitoring and Logging – Keeping an Eye on the Bot

When your chatbot is up and running, especially if deployed for others to use, it’s important to include **logging, error handling, and monitoring** to ensure it runs smoothly and you can debug issues.

Logging interactions: It’s useful to log each question asked and the answer given (and maybe the retrieved chunks) to a file or database. This way, you can review the logs to see if the bot is making mistakes or if certain queries are failing. In our project, we logged Q&A pairs into a SQLite database for analysis file. You could do something similar or even just append to a CSV file.

Using Python’s built-in `logging` module is a good practice. For example:

```
python
CopyEdit
import logging
logging.basicConfig(filename="chatbot.log", level=logging.INFO)

# After generating answer:
logging.info(f"Q: {question}\nA: {answer}\n")
```

This will append each Q&A to `chatbot.log`. (Include timestamps, etc., as needed.) If using FastAPI, you could integrate logging in the endpoint logic.

Error handling: Things can go wrong – maybe the website structure changes and scraping fails, maybe the embedding model download fails, or the LLM API doesn’t respond. You should wrap critical parts in try/except blocks to handle exceptions gracefully. For instance:

```
python
CopyEdit
try:
    results = collection.query(query_embeddings=[q_embedding], n_results=5)
except Exception as e:
    logging.error(f"Vector query failed: {e}")
    return {"error": "Knowledge retrieval failed, please try again later."}
```

This way, the server doesn’t just crash on an error; it logs the error and returns a friendly message.

For the LLM generation, if using an API, you might retry once if it times out. If using a local model, be mindful of memory.

Monitoring performance: If you deploy on a server, keep an eye on resource usage (CPU, RAM, etc.). Large models can hog memory. If using an API, monitor the usage so you don't rack up huge bills or hit rate limits.

Automated monitoring: You can incorporate simple health checks. For example, have a background job that periodically calls the `/ask` with a test question to ensure the system responds. Or integrate with monitoring services if this was a bigger project.

MLOps considerations: In a production scenario, you'd also consider:

- Versioning your models and data.
- Setting up alerts if the bot starts returning too many errors.
- Scaling the system if many users are asking questions at once (this could mean running multiple instances or using a more powerful machine).

For our project scope, a big MLOps component was automation with **Airflow** for data refreshing. **Apache Airflow is a platform to programmatically author, schedule, and monitor workflows.** We suggested using Airflow to schedule the web scraping and embedding process periodically (say, once a week) to update the knowledge base with any new content file. In a simple setup, you could achieve something similar with a crone job or a simple loop that runs at intervals.

While setting up Airflow might be overkill for a small project, it's a great tool if your chatbot needs continuous data updates or has multiple steps that you want to run on a schedule with dependencies (for example: Step1 scrape → Step2 process → Step3 update DB, all as a pipeline).

Example scenario: Suppose the university site updates with new courses each semester. You could schedule a scrape of the site every month, re-chunk and re-embed the data, and update the vector store. During this update, you might want to temporarily swap out the old vector index with the new one, or just update it incrementally. Monitoring these workflows ensures your bot stays up-to-date.

Step 9: Deployment – Running the Chatbot Locally or in the Cloud

After all this development and testing, you'll want to **deploy** your chatbot so that others (or at least you on another device) can use it. There are two main deployment scenarios: running it locally on your machine or deploying to a cloud service/server for remote access.

Local deployment: This might simply mean packaging your code and running it on your own computer or a school computer. If using FastAPI, you can run Uvicorn and access `localhost`. For a personal assistant type bot, this might be enough. Ensure you have all the necessary dependencies installed and perhaps freeze requirements (`pip freeze > requirements.txt`) so you know what needs to be installed on the target machine.

If you want a more user-friendly local app, you could create a desktop GUI with Python libraries (like Tkinter or PyQt) or use Electron, but that's optional and more involved. Many projects stick with a local web server (as we did with FastAPI) or a CLI tool.

Cloud deployment: To make your bot accessible from anywhere or to share with classmates, deploying to the cloud is ideal. A few ways to do this:

- **Cloud VM (Virtual Machine):** Services like AWS EC2, Google Compute Engine, Azure, etc. let you rent a server. You'd set up a machine, install Python and your app, and run it similar to how you did locally. For example, launch an EC2 instance with enough RAM (embedding models and small LLMs might need a few GB), deploy your code, and run Uvicorn on 0.0.0.0 so it's externally accessible (you'd also handle security groups to allow the port). Our future plan was to deploy on AWS EC2 with a GPU for the LLMfile-cx8tjw31rllphv64rncyec.
- **PaaS (Platform as a Service):** Platforms like Heroku, Railway, or Render can directly run FastAPI apps. You typically give them your code (often via Git) and they handle the server details. For example, a FastAPI app can be deployed on Heroku with a few configuration steps (like a `Procfile` that says how to start the server).
- **Containers:** Dockerizing your application is another route. You create a Docker image with all dependencies and your code, then run it on any cloud provider or container service (like AWS ECS, Google Cloud Run, or Kubernetes). This encapsulates the environment so it works anywhere. For a small project, this might be extra work, but it's a good skill to learn.
- **Serverless / Functions:** Unlikely for this use-case since our chatbot maintains state (the vector store) and possibly uses heavy models, but theoretically you could host just the /ask logic as a serverless function and use a managed vector DB (like Pinecone) and managed model API (OpenAI) so that each request is stateless. But that's an advanced pattern.

Deployment considerations:

- Make sure to **secure the app**. If it's open on the internet, you might want to add an API key or at least ensure it's not easily spammed (maybe add rate limiting).
- **Scaling:** If you expect many users at once, one instance might not suffice. You could run multiple instances behind a load balancer. But for a small project demo, one instance on a decent machine is fine.
- **Environment variables:** Don't hardcode keys (like OpenAI API key). Use environment variables or config files, especially when uploading code to cloud platforms.
- **Data storage:** If your vector DB is in-memory, note that restarting the server loses data. You might want to persist the ChromaDB data to disk. Chroma can use an SQLite or other file under the hood if configured. Alternatively, you could rebuild the index on start (by re-embedding source texts) but that adds startup time. Persisting is better for deployment. Chroma's default is an in-memory ephemeral DB unless you specify a persist directory.

For example, to persist ChromaDB:

```
python
CopyEdit
client =
chromadb.Client(chromadb.config.Settings(persist_directory="./chroma_data"))
collection = client.create_collection("univ_knowledge")
# ... after adding documents
client.persist()  # saves to disk
```

Next time, if you run the app and call `client = chromadb.Client(Settings(persist_directory="./chroma_data"))` and `get_collection("univ_knowledge")`, your data is loaded without re-processing.

Testing after deployment: Always test your deployed app as a user. Use the `/docs` or send some queries to ensure it works in the new environment. Check logs (if on a VM, you might tail the log file; if on Heroku, use their logs; etc.).

Analogy: Deploying your app is like **launching a rocket** – you move from the controlled environment (your computer, the launchpad) to the open world (cloud/space). You need to make sure all systems are go (dependencies, config) and be ready to handle things when you're not directly there to press buttons (hence good error handling and logging).
Fun Fact: There are free tiers on platforms like Render or Hugging Face Spaces where you could host a demo of your app. Hugging Face Spaces even has a nice UI for Gradio/Streamlit apps, which some use to showcase RAG bots on public datasets. Just be mindful of their usage limits.
Optional Challenge: If you have access, try deploying your chatbot to a free cloud service. For example, you could use Deta (a free service for FastAPI deployment) or Replit. Even if you can't keep it running long-term, going through deployment once will teach you a lot about making your project reproducible elsewhere.

Conclusion

In this guide, we built a Retrieval-Augmented Generation chatbot step by step. We started from scratch: scraping data, chunking it, creating embeddings, and storing them in a vector database. Then we integrated a large language model to generate answers using retrieved context, and we wrapped everything in an API for easy interaction. We also discussed evaluating the chatbot and maintaining it through logging and potential automation, and we brainstormed how to take it to the next level.

By breaking the problem into clear steps and using accessible tools, even a high school student with basic Python knowledge can implement this sophisticated system. Along the way, we touched on web development, machine learning, and data engineering concepts – showcasing how interdisciplinary AI projects can be. We hope you found this journey educational and empowering.

Now it's your turn: tweak the code, try it on new data, and have fun with it. Who knows – maybe this little RAG chatbot will be the foundation of your next big project! Good luck, and happy coding!

Key Takeaways: A RAG chatbot is essentially *"search engine + chatbot"* combined. By giving a language model access to external information (like giving an open-book test), you dramatically improve its usefulness. The skills you practiced – scraping, data processing, vector search, and working with LLMs – are highly valuable in modern AI development. Great job on making it through the entire build.