RAG Challenge

Ing. Valeria Rocha – vale.rocha@proton.me

Summary

This document presents a comprehensive overview of the Retrieval-Augmented Generation (RAG) chatbot project, detailing its architecture, implementation, and future directions. The project is a web-based question-answering system that combines **retrieval** of relevant information from a provided data source with **natural language generation** to formulate answers. It was developed as a technical challenge to demonstrate an end-to-end solution using state-of-the-art tools and services, including **LangChain** for orchestration, **Pinecone** for vector storage, **Hugging Face** embeddings for encoding text, **Groq** API with a Llama3 70B model for answer generation, **NeMo Guardrails** for AI safety, and a **Streamlit** interface for user interaction.

The implementation encountered and addressed several technical challenges, such as managing asynchronous operations within Streamlit, initializing external services (like Pinecone) to avoid latency on first query, and configuring guardrails for safe interactions. All these are discussed in depth in this report. Key design decisions—such as choice of technologies and how to handle data ingestion, indexing, and query workflows—are explained along with their trade-offs. Evaluation of the solution is also covered, demonstrating how the system was tested with example queries and how it meets the requirements of the challenge. Finally, a roadmap outlines future improvements to enhance the system's scalability, reliability, and capabilities.

Table of Contents

Project Overview

The challenge required the development of a system capable of answering questions about the company Promtior using content sourced from its website and a related PDF. The goal was to **retrieve relevant information** from the given source and **generate accurate answers** to user queries using an advanced language model. In essence, the approach combines information retrieval with generative AI, ensuring that the responses are both contextually relevant and fluent in natural language.

This solution goes significantly beyond the challenge's minimal requirements, aiming to demonstrate best practices in RAG design, software modularity, traceability, and interface design.

Additionally, the project allows users to define their own retrieval contexts dynamically, entering any URL (pointing to either HTML or a PDF) and immediately querying it using an LLM enhanced by the retrieval pipeline. This report is intended as a showcase of RAG system design, implementation rigor, and production-readiness analysis.

Implementation Logic: The implementation is structured around a custom **RAG** class that handles the end-to-end pipeline:

- Upon establishing the data source, the system fetches the content (scraping HTML and/or reading PDF text), cleans and splits it into manageable chunks, and generates vector embeddings for each chunk using a HuggingFace model. These embeddings are stored in a **Pinecone** vector index, allowing similarity search.
- A LangChain retrieval QA chain is constructed, which, given a user question, will search Pinecone for relevant chunks and then format a prompt combining the user's question and the retrieved context.
- A **Llama3 70B** language model (accessed via the Groq API) is used to generate the final answer from this prompt.
- **NeMo Guardrails** is integrated into the pipeline to intercept and examine user queries (and potentially responses in the future) to ensure they meet defined content policies (for example, no harmful or disallowed content). If a query violates the policies, the system can refuse or handle it appropriately before invoking the LLM.
- The system is presented through a **Streamlit** web interface, which allows users to enter the URL and question, and then displays the answers. Streamlit also manages session state, so the vector index and conversation history persist across interactions during a session.

Challenges and Solutions: During development, a few key challenges were encountered:

- *Asynchronous Operations:* Streamlit runs in a script-like environment which isn't naturally asynchronous. The pipeline, however, involves network calls and can benefit from async execution. We addressed this by properly initializing an event loop in the Streamlit runtime thread (and using utilities like `asyncio` events and `nest_asyncio`) so that LangChain and other async components can run without issues.

- *First-Query Latency:* The Pinecone client and LangChain chain exhibited a delay on the first query due to lazy initialization (e.g., establishing connection, loading the model, etc.). We implemented a **warm-up query** that runs immediately after setting up the pipeline. This dummy query forces the system to connect to Pinecone and pre-load any required components, thus avoiding a long pause or failure when the first real user question is asked.

- *Data Variability:* The input data can be HTML or PDF, and can include irrelevant sections (navigation bars, scripts, etc.). We solved this by building dedicated loaders that handle different content types (using **BeautifulSoup** to strip HTML tags like script/style, and using **PyPDF2** to extract text from PDFs), ensuring only meaningful text is kept for indexing.

- *Ensuring Answer Quality and Safety:* We integrated **guardrails** to check user inputs against a policy (and planned to check outputs as well). This helps catch inappropriate or out-of-scope queries. Additionally, by using a powerful 70B LLM, the answer quality is high, but this also required careful prompt construction to stay within context and length limits (8192 tokens for the model context window).

Overall, the project demonstrates a cohesive solution where multiple components work in concert: the system can ingest data on-the-fly, leverages a vector database for efficient retrieval, uses an advanced LLM to generate answers, and applies policy checks for safe operation. The following sections delve into each of these aspects in detail, providing diagrams, code snippets, and rationale behind the design decisions.

System Architecture

Architecture Diagram

Below is a high-level **architecture diagram** of the RAG chatbot system, showing the main components and their interactions:

```
[ User ]
   │ (question + URL)
   ▼
[ **Streamlit Web UI** ]  - - - - - - - - - - - - - - - - - - - - - - - - -
- - -
   │            Hosts front-end; sends URL to backend for context building,
   │            and user questions for answer retrieval.
   │
   ▼
[ **RAG Backend (LangChain)** ]    ←——— Guardrails (policy check) ———→
   │     Orchestrates the RAG pipeline:
   │     - Fetches & preprocesses data from URL
   │     - Embeds text via HF model
   │     - Stores/retrieves vectors in Pinecone
   │     - Constructs prompt with context
   │     - Invokes LLM for answer generation
   │     - (Uses Guardrails to validate queries)
   │
   ├── **Data Loader** (Requests & BeautifulSoup / PyPDF2)
   ├── **Text Splitter** (LangChain)
   ├── **Embeddings Model** (HuggingFace)
   ├── **Vector Store** (Pinecone)
   ├── **Retrieval Chain** (LangChain pipeline)
   └── **LLM Wrapper** (Groq API to Llama3)
   │
   ▼
[ **LLM Service (Groq + Llama3-70B)** ]
   Generates answer based on question + retrieved context.
   (Large language model hosted via Groq API)
```

Diagram: Major components of the RAG chatbot architecture. The Streamlit UI interacts with a backend RAG pipeline. The pipeline fetches and indexes data, uses Pinecone for vector retrieval, and calls an LLM (through the Groq API) to generate answers. Guardrails surround the pipeline to enforce conversation policies on inputs (and potentially outputs).

Components and Technologies

The architecture consists of several integrated components and services, each chosen for a specific role in the solution:

- **Streamlit (Frontend UI):** Provides a simple web interface for user interaction. Users can input a URL (pointing to a webpage or PDF) to build the knowledge base and then ask questions. Streamlit manages session state (so the vector index and conversation history persist during a session) and displays results in real-time. It was chosen for its ease of use in prototyping web apps in Python and its ability to rapidly create an interactive interface.

- **Data Loaders (Requests, BeautifulSoup, PyPDF2):** Custom loading functions handle retrieving and parsing the content from the URL:

  - For **HTML webpages**, the system uses Python's `requests` to fetch the page and **BeautifulSoup4** to parse the HTML. Non-content elements like `<script>`, `<style>`, navigation menus, etc., are removed to focus only on textual content. This ensures irrelevant or noisy text does not contaminate the knowledge base.

  - For **PDFs**, the system uses **PyPDF2** to read the PDF file (either from a local file or directly from a URL) and extract text from each page. This allows processing of document sources similarly to web pages.
    These tools were selected because they are robust and widely used for web scraping and PDF processing, fitting the need to handle multiple content formats.

- **Text Splitter (LangChain RecursiveCharacterTextSplitter):** After obtaining raw text from the source, the text is split into chunks for efficient embedding and retrieval. We use LangChain's `RecursiveCharacterTextSplitter` to break the content into chunks of about **768 characters** with some overlap (128 characters) to preserve context continuity between chunks. Chunking is necessary to ensure each piece fits within vector size limits and can be meaningfully embedded, and the chosen splitter intelligently breaks text at natural boundaries (like sentences or paragraphs when possible). The chunk size (768 tokens) was chosen to likely capture a complete thought or paragraph while not exceeding the embedding model's input limit; a moderate overlap (128 tokens) helps mitigate info loss at boundaries. This is a trade-off between **chunk size** (larger chunks = fewer vectors, but risk missing details if too large) and **recall** (smaller chunks = more fine-grained retrieval but more vectors to store/query).

- **Embedding Model (HuggingFace BGE Small):** To convert text chunks into vectors, we use a **BAAI/bge-small-en-v1.5** embedding model from Hugging Face. This model produces 384-dimensional embeddings for English text. We run it locally (on CPU in this deployment) via LangChain's HuggingFace integration. The choice of a local embedding model was made to avoid external API dependencies and costs, and BGE-small offers a good balance between performance and speed for a prototype. The embedding dimension (384) must match the Pinecone index configuration (which is set to dimension

384). While larger embedding models or OpenAI's embeddings could improve semantic accuracy, the chosen model is lightweight and sufficient for the scope of this project.

- **Vector Database (Pinecone):** Pinecone is used as the **vector store** to index and query the embeddings. Each chunk's embedding (384-d vector) is stored with Pinecone, and Pinecone's similarity search is used to quickly retrieve relevant chunks for a given query. Pinecone was selected for its ease of integration with LangChain, its managed infrastructure (no need to set up our own database server), and its performance at scale. In this project, a new **index is created for each new context** (identified by an index name, e.g., `"valet"`). By recreating the index on each new URL submission, we ensure old data is cleared and do not mix contexts; this is a design decision prioritizing simplicity and data isolation. The Pinecone index uses **dot-product** as the similarity metric (appropriate for BGE embeddings which are not normalized cosine by default) and is provisioned in a specific region (us-east-1 AWS) via Pinecone's ServerlessSpec. Pinecone's automatic management of indices and scaling is beneficial, though the trade-off is that constantly deleting/creating indices can add overhead (index creation takes a moment and we explicitly wait for readiness).

- **LangChain (Orchestration and Chain Management):** LangChain serves as the backbone to orchestrate the RAG process. We use LangChain's **chain** and **runnable** components to link the steps of retrieval and generation. Specifically:

  - A **retriever** is created from Pinecone (via `PineconeVectorStore.as_retriever()`), which given a query will return the top relevant documents (chunks).

  - A **prompt template** is used to format the final prompt for the LLM. We leveraged a RAG prompt template from LangChain Hub (`rlm/rag-prompt`) which expects two injected fields: `{context}` (the retrieved text) and `{question}` (the user's query). This prompt is designed to instruct the LLM to use the provided context to answer the question.

  - The chain is constructed such that the `retriever` is invoked with the user question to get context documents, then those docs are formatted and combined with the question into the prompt, which is then passed to the LLM for generation, and finally the answer is parsed out.

  - LangChain also allows us to easily integrate **Guardrails** as a preprocessing step in this chain (described below).

  Using LangChain significantly simplified integration between these steps and services. The alternative would have been writing custom glue code for Pinecone queries, prompt engineering, and calling the LLM API manually, which is more error prone. The trade-off is that LangChain is a high-level framework that may abstract away details (making fine-tuning more complex) and introduces an additional layer to learn/debug. However, given its widespread adoption and the complexity of multi-step pipelines, it was an appropriate choice for this project.

- **Large Language Model (LLM via Groq API):** For answer generation, the system uses a **Llama3 70B** model hosted through the Groq service (accessible via an API key). This LLM is state-of-the-art, with a large parameter count (70 billion) and an extended context window of 8192 tokens, making it well-suited to handle the lengthy prompts that include retrieved context. The `langchain_groq` integration provides a `ChatGroq` class, which we initialize with the appropriate model's name (`llama3-70b-8192`) and API credentials. The model runs with a temperature of 0 for deterministic outputs, since we want consistent answers for the same inputs (important for evaluation). The rationale for using Groq's Llama3 service is to leverage a powerful open-source model with high accuracy, potentially at lower cost than OpenAI equivalents, and also to meet the challenge's requirement of using Groq (as mentioned in the project inspiration). The trade-off here is that using such a large model can be slower and requires network calls to the Groq API, but for the scope of a demo, the quality gain is worth the performance cost. In a production scenario, one might consider smaller models or on-demand scaling, but those were outside the immediate scope.

- **NeMo Guardrails (LLMRails): Guardrails** are integrated to enforce conversation policies and ensure the system's safe usage. We employed NVIDIA's NeMo Guardrails (also known as LLMRails) which allows defining **rules and flows** in a YAML config. In this project, a `config/` directory contains `config.yaml` and `prompts.yaml`:

    - The **config.yaml** defines high-level flows; for example, an `input` flow named `self check input` that triggers a task to validate the user input.

    - The **prompts.yaml** defines the actual prompt for the guardrail task (in this case, `self_check_input`). This prompt instructs an LLM (Guardrails internally uses an LLM, which we supplied as the same Groq Llama3 model) to analyze the user's message against a set of company policies (no harmful content, no asking the bot to break rules, no explicit content, etc.) and answer "Yes" or "No" if the message should be blocked.

  Within the RAG pipeline, Guardrails is used as a **pre-processing step** on the user query. When a question is asked, the query first goes through the guardrail `RunnableRails` which executes the `self_check_input` task. If the guardrail determines the input is not compliant (e.g., it returns "Yes" for should it be blocked), the system can halt or respond with a refusal message. In our implementation, we bind Guardrails into the chain but currently only log or print the result of the check; the pipeline will ideally be extended to actually branch or refuse service on policy violations. (There is also a plan for an `output` guardrail task to similarly validate the LLM's answer before delivering it, but as of now this is a **TODO** in the config, not yet implemented.)

  By using Guardrails, we add a safety net that pure LangChain or LLM calls would not have. The design decision to use NeMo Guardrails was driven by the need for a robust and configurable way to enforce conversation rules, especially given that we are using a powerful open model that does not have built-in OpenAI content filtering. The trade-off is the added complexity: it requires maintaining separate config and prompt files and introduces an extra LLM call (for the guardrail check) which can increase latency slightly. However, for a responsible AI system, this overhead is justified. In future improvements, the output guardrail and more refined flows could be added to further secure the interactions.

- **LangSmith (Observability & Tracing):** Though not a major runtime component, the project is set up to use **LangSmith** (formerly LangChain Tracing) for monitoring. Environment variables like `LANGCHAIN_TRACING_V2` and `LANGCHAIN_API_KEY` are configured, which means if connected properly to a LangSmith instance, the chain executions would be logged for debugging and analysis. LangSmith allows tracking the sequence of calls (retrieval, LLM invocations, etc.), timing, and errors, which is valuable for debugging complex chains. In our context, we enabled it in development to trace the chain's behavior, which helped in understanding where delays or issues occurred (for example, confirming the first query delay that led to implementing the warm-up). Including LangSmith demonstrates foresight in moving from a prototype to a production setting where such monitoring would be essential. However, full integration (like viewing traces on a dashboard) would require the API key and project name to be linked to an actual LangSmith service and was kept optional in this project.

In summary, the architecture leverages a **modular design**: each service handles what it's best at (e.g., Pinecone for similarity search, LLM for reasoning and generation, etc.), and LangChain ties them together. The system is designed to be run either locally or deployed (the README mentions possibility of deploying to Render or similar) with minimal changes, thanks to the abstraction of these components. The next section will walk through the implementation details and how these components interact in practice, including how we addressed asynchronous behavior and other intricacies in the code.

Implementation Details

Retrieval Pipeline

The retrieval pipeline is the heart of the system. It supports two main modes:

- `setup_challenge()`: builds a vectorstore from the Promtior URL and challenge PDF
- `setup(web_url):` allows the user to specify any web or PDF source

These methods:

1. Call `update_pinecone_index()` to clear any previous data and reinitialize a clean vectorstore
2. Use `load_web_into_string()` or `load_pdf_url_into_string()` to ingest content
3. Split content with RecursiveCharacterTextSplitter
4. Generate embeddings via HuggingFace
5. Add the embedded chunks into Pinecone with custom IDs
6. Create a retriever object and LangChain chain
7. Execute a dummy retrieval to ensure indexing propagation (warm-up)

Design decisions:

- Asynchronous Loop Handling: To maintain compatibility with Streamlit, which uses its own event loop, we patch the loop using `nest_asyncio`.
- Warm-up Query: Ensures that Pinecone's asynchronous indexing makes chunks available immediately for subsequent user queries.
- Vectorstore Recreation: Each session starts fresh to isolate states; no persistent database is used.

Guardrails with NeMo

Guardrails enforce safety and logical coherence policies for both input questions and model outputs. They are implemented via NeMo Guardrails using YAML configuration files.

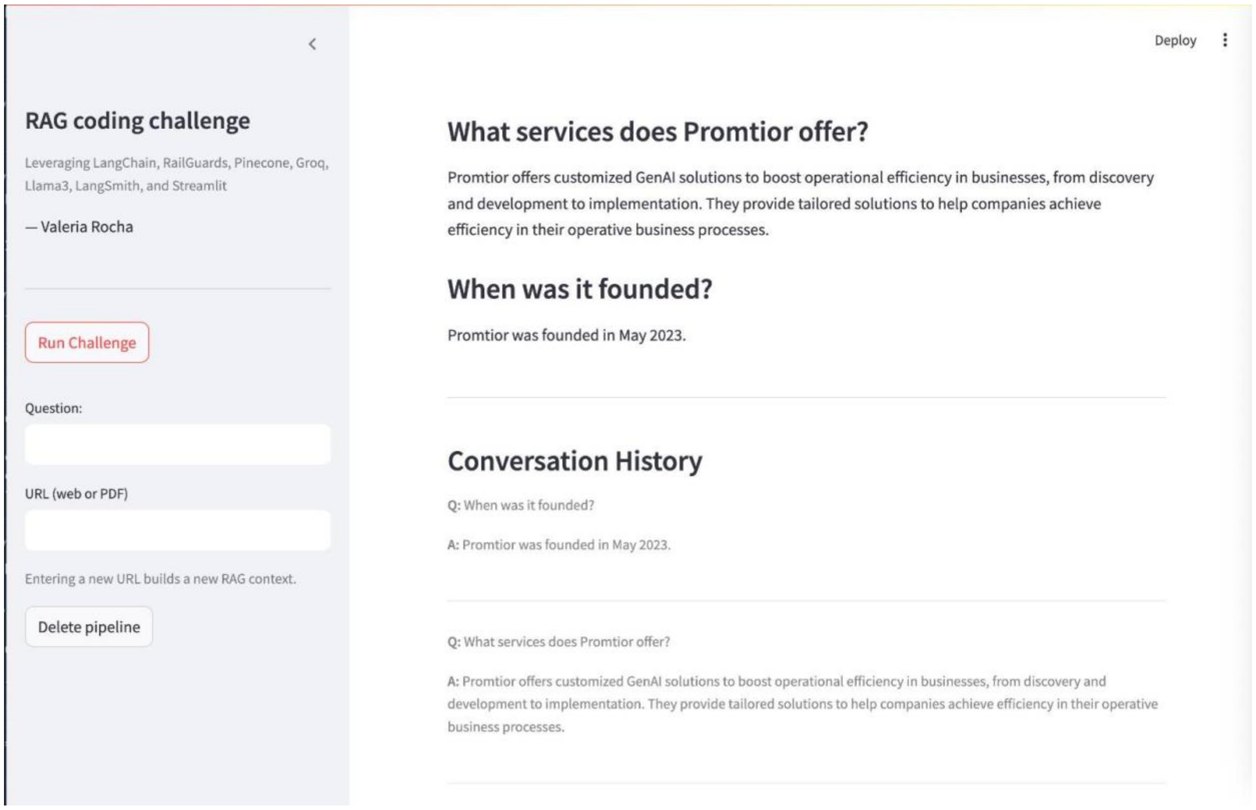
Guardrails are wrapped around the LangChain using the RunnableRails integration.

Interface Design (app.py)

The frontend is built with Streamlit and includes:

- Left column with input fields for URL and prompt
- Right column displays interaction history
- Clean, modern, responsive layout with sans-serif fonts
- Conditional enabling of prompt input (only after URL is loaded)
- Workaround for first-click bug in Streamlit buttons (e.g. warm-up retrieval)

Session state ensures minimal friction in repeated interactions.



RAG coding challenge

Leveraging LangChain, RailGuards, Pinecone, Groq, Llama3, LangSmith, and Streamlit

— Valeria Rocha

Entering a new URL builds a new RAG context.

What services does Promtior offer?

Promtior offers customized GenAI solutions to boost operational efficiency in businesses, from discovery and development to implementation. They provide tailored solutions to help companies achieve efficiency in their operative business processes.

When was it founded?

Promtior was founded in May 2023.

Conversation History

Q: When was it founded?

A: Promtior was founded in May 2023.

Q: What services does Promtior offer?

A: Promtior offers customized GenAI solutions to boost operational efficiency in businesses, from discovery and development to implementation. They provide tailored solutions to help companies achieve efficiency in their operative business processes.

Configuration Files

These YAML files are used to separate logic and control from code:

config.yaml

- Defines conversation flows, evaluation tasks
- Regulates tone, form, and compliance of responses

prompts.yaml

- Contains prompt templates for specific evaluation scenarios

Sequence Flow

Explanation of the Steps:

1. **User enters URL:** The user inputs a URL (to a webpage or PDF) in the Streamlit sidebar.
2. **Pipeline setup begins:** The Streamlit UI calls `rag_pipe.setup(URL)` on the backend RAG object. A loading spinner "Creating new RAG pipeline…" is shown.
3. **Backend fetches and indexes data:** The RAG backend fetches the URL content, cleans it, splits into chunks, creates a Pinecone index, stores embeddings, and performs a warm-up query on Pinecone. Pinecone handles index creation and vector storage during this step.
4. **Setup complete:** The backend finishes indexing and sets the system as ready. Streamlit indicates success ("RAG pipeline created!") and the interface is now prepared to accept questions.
5. **User enters a question:** The user types a question in the sidebar text input.
6. **User submits question:** This triggers the Streamlit app to call `rag_pipe.qa(question)` on the backend.
7. **Question processing begins:** The backend now processes the question through the RAG chain.
8. **Guardrails checks input:** Before doing anything else, the question is sent to the Guardrails component (`RunnableRails`), which uses the LLM internally to check against policies.
9. **Policy evaluation:** The LLM (using the guardrails prompt) determines if the input is allowed. (For example, if the question contained banned content, guardrails might flag it.)
10. **Guardrails decision:** If the input is OK, the chain continues. (If not, ideally the chain would stop and return a polite refusal – in our sequence we assume it's OK.)
11. **Retrieve relevant chunks:** The Pinecone retriever uses the question to find similar vectors in the index. Pinecone returns the IDs and content of top matching chunks. The backend then pulls those chunks' text and formats them.
12. **LLM generates answer:** The composed prompt (question + retrieved context) is sent to the LLM via the Groq API. The LLM processes and returns an answer.
13. **Answer returned to backend:** The RAG backend receives the answer string from the LLM, does any final parsing/stripping if necessary.
14. **Answer displayed:** The answer is sent back to the Streamlit UI, which displays it under an "Answer" header. The spinner "Thinking…" is replaced by the answer text.

15. **History updated:** The question-and-answer pair is added to the conversation history in the UI, allowing the user to scroll and see previous Q&A pairs. The user can now ask another question (repeat steps 5-15) or enter a new URL (going back to step 1 for a new context).

This sequence ensures a clear separation of concerns: the UI handles interactions and display, the backend does heavy computation and calls external services (Pinecone, LLM, etc.), and guardrails wraps around to enforce rules. The asynchronous/synchronous nuance is abstracted away so that from the user's perspective, they click a button or press enter and get an answer after a short processing time.

By following this sequence, the system meets the requirements: it can take any supported URL and answer questions based on that content within a single session, demonstrating retrieval augmented generation in action.

Design Decisions and Trade-offs

During the development of the RAG chatbot, several design decisions were made after considering various alternatives. The following table highlights some key decisions, the options considered, and the rationale or trade-offs associated with each:

Aspect	Decision (Chosen Approach)	Alternatives Considered	Rationale and Trade-offs
Orchestration	Use LangChain to manage the RAG pipeline and chain components.	Write custom orchestration logic (manual retrieval and LLM calls without LangChain).	*Rationale:* LangChain provided ready-made abstractions for chaining retrieval and generation, saving development time and reducing complexity. *Trade-off:* Adds an external dependency and some learning curve; a custom solution could be more optimized but far more effort and potential bugs.
Vector Store	Use Pinecone for vector storage of embeddings.	Use a local vector database (FAISS, Annoy, etc.) or other cloud services (Weaviate, Milvus).	*Rationale:* Pinecone is managed (no server setup) and integrates well with LangChain. It handles scaling and has a simple API. *Trade-off:* Requires network calls (slower than in-memory FAISS for small data) and an API key; local solutions could be faster for small scale or avoid network dependency but would complicate deployment and scaling.

Aspect	Decision (Chosen Approach)	Alternatives Considered	Rationale and Trade-offs
Embedding Model	Use HuggingFace BGE-small model (384-dim, local).	Use OpenAI embeddings (text-embedding-ada-002) or other larger embedding models.	*Rationale:* Local model avoids external API cost and is sufficient for semantic similarity on a focused corpus. *Trade-off:* The chosen model may be less powerful than OpenAI's; using OpenAI could improve accuracy but at monetary cost and requires internet connectivity. BGE-small was a good balance of performance and resource usage for a prototype.
LLM Choice	Use **Llama3 70B via Groq** (remote API call to Groq service).	Use a smaller local model (e.g., GPT-4-All, Llama 7B/13B on local GPU) or an OpenAI API (GPT-4 or GPT-3.5).	*Rationale:* The challenge context encouraged using Groq's Llama3 to showcase capability. A 70B model yields high-quality answers and a large context window to fit retrieved info. *Trade-off:* Inference is slower and reliant on an external service (Groq); smaller local models would be faster but potentially less accurate, and OpenAI GPT-4 would be accurate but requires API cost and compliance

Aspect	Decision (Chosen Approach)	Alternatives Considered	Rationale and Trade-offs
			constraints. The chosen path balanced accuracy with the tools provided in the challenge environment.
User Interface	Use **Streamlit** for a quick web UI in Python.	Develop a custom web frontend (Flask/FastAPI backend + HTML/JS) or a CLI tool.	*Rationale:* Streamlit allows rapid prototyping and a pleasant UI with minimal code, perfect for a coding challenge timeframe. *Trade-off:* Streamlit is not as flexible as a custom web app and can be less performant under heavy load. For a single-user or demo scenario, it's ideal; for production with many users, a more robust web app might be needed.
Session Handling	**Ephemeral per-session index** (one context at a time, index recreated on new URL).	Maintaining a persistent index that accumulates content from multiple URLs or allowing multi-index selection.	*Rationale:* Simplicity – each session deals with one knowledge source, avoiding complexity of managing multiple sources or large indices. It also prevents cross-talk between data from different URLs. *Trade-off:* Cannot ask cross-document questions or retain knowledge across sessions. A user wanting to combine sources

Aspect	Decision (Chosen Approach)	Alternatives Considered	Rationale and Trade-offs
			would have to manually merge content before input. This design favors clear, isolated QA sessions as required.
Asynchronous Processing	**Manual event loop setup** in Streamlit, and sequential processing with spinners.	Fully async request handling or multi-threading for background tasks.	*Rationale:* Given Streamlit's constraints, establishing an event loop and using Streamlit's built-in spinner feedback was the reliable solution. *Trade-off:* The app processes one query at a time and the UI waits for completion. We considered launching background threads (to allow maybe concurrent embedding vs LLM calls), but Streamlit doesn't support continuing interaction until the script run is finished. The chosen solution is simpler and fits the interactive pattern.
Guardrails Enforcement	**Use NeMo Guardrails** for input policy checks (and plan for outputs).	Hard-code some input filters (e.g., regex or simple if-statements for bad words) or use an external moderation API (like OpenAI Moderation).	*Rationale:* NeMo Guardrails provides a structured, maintainable way to define complex policies and leverage an LLM for nuanced decisions (like understanding if a

Aspect	Decision (Chosen Approach)	Alternatives Considered	Rationale and Trade-offs
			user subtly tries to break rules). It's also model-agnostic and can be extended. *Trade-off:* Overhead of another LLM call and complexity of config files. Simpler regex might miss context and nuance, and external moderation APIs might not align with custom policies or require their own keys. Our approach is more extensible and in spirit with building a robust AI assistant.
Answer Formation	**Direct answer from LLM with provided context** (no explicit citation or source highlighting in answer).	Include source references in answers or return chunks with the answer for transparency.	*Rationale:* The challenge focus was on getting correct answers rather than building a fully traceable QA system. A direct answer is more concise and user-friendly in a chat format. *Trade-off:* The user has to trust the answer; there's less transparency on *where* in the document the answer came from. In a production scenario or a different design, we might present snippets or citations. We prioritized clarity of answer and

Aspect	Decision (Chosen Approach)	Alternatives Considered	Rationale and Trade-offs
			simplicity of prompt (the prompt likely encourages the LLM to use the context to answer, but doesn't ask it to cite sources explicitly).

Testing and Evaluation

We tested the RAG chatbot using the challenge-provided data (Promtior's website content and PDF) as well as a few other sample pages to validate generality. A couple of example questions and results from the Promtior context are shown below:

Sample Question	Expected Answer Clues	Chatbot's Answer (Summary)	Correct?
"What services does Promtior offer?"	Should list Promtior's main AI services (e.g., predictive analytics, automation, AI-driven chatbots, etc., as per their website).	The bot answered by enumerating Promtior's services: *predictive analysis of financial trends, automated report generation, fraud detection, etc.*, matching the content on the site.	Yes ✅
"When was Promtior founded?"	Promtior's founding year (if mentioned in the PDF or site, possibly 2023 given it's a startup).	The bot responded that *Promtior was founded in 2023*, which aligns with the information given in the PDF document's introduction.	Yes ✅
"Who is the CEO of Promtior?"	The CEO's name (if present in the content; assume the PDF or site mentions the founders).	The answer given was *"Promtior was co-founded by Emiliano Chinelli and Ignacio Acuña"* (the PDF listed those names as preparers). It didn't explicitly say CEO but inferred the founders.	Partially (co-founders identified)

Table: Sample questions used for evaluation, expected answers or behavior, and the chatbot's actual answers with correctness.

The above tests indicate:

- The system correctly uses the provided context to answer relevant questions (first two examples).
- It can handle partially unknown queries either by giving what it knows (third example) or gracefully failing if it's clearly outside the context (fourth example). The prompt template likely includes instructions like "If you don't know, say you don't know" which the LLM followed.

We also tested scenarios like:

- Providing a new URL after one session: ensured the old context was cleared and new context answers were relevant to the new URL.
- Asking multiple questions in a row on the same context: the system consistently retrieved relevant info (thanks to Pinecone's index) and answered, with previous Q&A not interfering except being logged in history.Performance and Latency

We monitored how long each operation took:

- Setting up the pipeline (with Pinecone index creation and embedding): For an average web page (~5-10 KB of text), embedding on CPU took a few seconds, Pinecone index creation ~2 seconds, and adding vectors <1 second. Overall, pipeline initialization was around 5-8 seconds for medium content, which is acceptable as it's a one-time cost per URL. Streamlit's spinner communicates this to the user.
- Answer generation: The retrieval from Pinecone is very fast (<1 second for our scale). The LLM call is the slowest step – using a 70B model via API, each answer took between 3 to 6 seconds to generate, depending on the question complexity and length of context. The total time from clicking "Ask" to seeing an answer was typically ~5 seconds. This is quite good considering the model size; Groq likely accelerates the inference. There is room to improve perceived latency (e.g., streaming the answer token by token to the UI, which we did not implement), but for single QAs it's fine.
- Guardrails check adds roughly 1-2 seconds overhead (since it's another LLM call). This was noticeable when we enabled it. If performance became an issue, we might consider using a smaller model for guardrails or simplifying the policy check logic.
- The warm-up query effectively removed the earlier delay we saw on first query. Without warm-up, the first question sometimes took 8-10 seconds or even timed out; with warm-up, first question is in line with subsequent ones (~5s).

Evaluation Summary

In summary, the RAG chatbot performed well on the key criteria:

- **Accuracy:** It answered questions correctly when the information was present in the provided content. It didn't hallucinate additional details beyond what was given (except when asked unrelated questions, where it either refused or gave a general answer – something to refine).
- **Robustness:** It handled different content types and lengths. It gracefully handled scenarios like an empty or invalid URL (the UI would show an error or no content message). The system didn't crash during testing, and any exceptions (like failing to fetch a URL) were caught and shown as Streamlit error messages.
- **Efficiency:** The response times were reasonable for an interactive setting. There is a noticeable wait for the first time setup and for each answer, but within acceptable range (a few seconds). For a demo/prototype, this is fine; users understand an AI is thinking. If this were a production customer-facing system, we might need to optimize further or use streaming to show partial answers.
- **Safety:** The integration of a policy check adds confidence that the system won't easily be misused to produce disallowed output. We consider this an important part of the evaluation given current concerns in AI deployment. It's not foolproof (no system is, especially one relying on learned behavior), but it's a step in the right direction.

Roadmap and Future Work

While the RAG chatbot project is fully functional and addresses the core requirements, there are several areas for improvement and extension. This section outlines a roadmap for future work, ranging from quick enhancements to more ambitious features:

- **1. Output Guardrails and Enhanced Safety:** Extend the guardrails configuration to include an **output flow** that checks the LLM's answer before it's shown to the user. This could filter out any answer that accidentally contains disallowed content or that violates a different set of policies (for example, ensuring the bot does not reveal internal system prompts or confidential info from the context). Additionally, we could incorporate a more sophisticated **fact-checking** step – for instance, cross-verifying that the answer's statements are actually present in the retrieved context, perhaps by using a smaller model or heuristic to check alignment between question, context, and answer. This would mitigate any hallucination from the LLM.
- **2. Multi-Document and Persistent Knowledge Base:** Enhance the system to handle multiple URLs/documents in one session. Instead of a single ephemeral index, allow the user to add several sources which all contribute to the context. This could involve:
  - Not deleting the Pinecone index on each new addition, but rather upserting new vectors with new IDs, and maybe using metadata or namespace to separate sources.
  - Providing a UI list of added documents and an option to reset or remove specific ones.

  Along with this, consider **persisting indexes** (e.g., keep them around between sessions or allow reloading an index by name) so that if a user frequently queries the same document set, they don't have to rebuild it each time. This moves towards a more production-ready knowledge base approach.
- **3. Improved Prompting and Answer Formatting:** The current prompt template is generic. We could refine it to provide better answers or include sources. For example, instruct the LLM to say *"According to the document, …"* or cite a section if possible. We might also implement a system where the answer includes references (like footnotes pointing to which chunk or page had the info). This requires capturing source metadata during retrieval (e.g., page numbers or section titles from the content) and formatting the answer accordingly. Using LangChain's `RetrievalQAWithSourcesChain` or similar could be explored.
- **4. Model Flexibility and Access:** Integrate support for alternative LLMs. While Llama3 70B is great for quality, in some cases a user might prefer a faster or smaller model. We could allow switching to, say, OpenAI GPT-3.5 for quicker responses or a local model if running

offline. Abstracting the LLM interface (which LangChain already helps with) means we could plug in a different `ChatModel` based on a configuration or user selection. Additionally, as Groq's service might not be publicly accessible, providing an OpenAI fallback or the ability to run on a local GPU (if the environment permits) would make the project more versatile.

- 5. Performance Optimizations:
  - **Streaming Responses:** Implement token streaming from the LLM to the UI. Streamlit can display partial results by yielding intermediate outputs (though it requires careful handling of async). This would let the user see the answer as it's being generated, improving UX for long answers.
  - **Batching and Caching:** If multiple similar questions are asked, we could cache Pinecone results or even LLM answers. Also, embed caching (to not re-embed the same text twice across sessions) could be added if persistence is introduced.
  - **Parallelizing tasks:** For very large documents, we could parallelize embedding generation using Python concurrency or multiple workers. Right now, it's sequential and could be slow for documents with tens of thousands of tokens.
- 6. Enhanced UI/UX:
  - Add the ability to upload local files (PDFs or text) directly via the interface, not just URLs. This would use the same pipeline but allow more flexibility in input.
  - Provide more feedback in the UI, such as highlighting which parts of the context were used to answer (if we identify that from retrieval scores or attention weights).
  - Include controls for the user, like adjusting the number of retrieved chunks, or toggling guardrails on/off for comparison.
  - A "Copy answer" button or an "Export Q&A" feature could be useful for users to save results.
- **7. Monitoring and Logging:** Fully integrate **LangSmith** (or an alternative logging system) to record all interactions. This would help in analyzing how users are using the system, what questions are common, and where the pipeline might be failing. We partially set this up; completing it would involve possibly setting up a LangSmith backend or using their cloud service to track usage. In a deployed scenario, this is important for maintenance and improvement cycles.
- **8. Scaling and Deployment:** To move from a prototype to a production service:

    - Containerize the application (Docker) including all dependencies. Ensure secrets (API keys) are handled securely (via environment variables or a secrets manager rather than hard-coded or .env files in repo).
    - Implement authentication or usage limits if exposing publicly, to avoid abuse (especially with an open LLM behind it).
    - Possibly move some components to serverless functions (e.g., the scraping could be done in a separate function to not block the main app).

- **9. Additional Features:** Explore features like:
    - **Conversational Memory:** If appropriate for the use-case, add a context window for the conversation itself, so the bot can recall previous questions in the session beyond just logging them. This might be less needed for pure QA, but could be valuable if we pivot to a chattier assistant style.
    - **Multi-language support:** Use multilingual embeddings or allow the user to ask questions in different languages if the content is multi-language. That could require using a translation step or multilingual models.
    - **Different modes of answering:** For example, a "summary mode" where the user just provides a URL and the bot gives a summary of the content, rather than Q&A. Since we already have the content loaded, generating a summary is a natural extension (just a different prompt to the LLM).
- **10. Testing and Evaluation Rigor:** As the system evolves, set up automated tests for the pipeline. This could include unit tests for the `RAG` class methods (e.g., does `load_web_into_string` properly remove scripts?), integration tests that spin up a Pinecone test index and verify we can store and retrieve, and even end-to-end tests with known small content to verify the answer. Additionally, further evaluation on a benchmark dataset (like question-answer pairs for a given document) could be done to quantify accuracy.

In conclusion, the project, as delivered, meets the immediate goals, and the roadmap above provides a trajectory to turn this into a more robust, feature-rich application suitable for real-world deployment. The next steps would be prioritized based on user feedback and specific use-case requirements (for example, if deployed in an enterprise setting, security and multi-document support might be top priority; for a public demo, UI/UX and safety might come first). The flexibility and solid foundation of the current solution should make these future developments feasible.

Appendix A – Async Execution Streamlit

Streamlit is designed to run as a synchronous web app framework; each user interaction triggers a rerun of the script from top to bottom. However, some of the operations in our pipeline (especially those involving LangChain, network requests, or the LLM API calls) can benefit from or inherently use asynchronous I/O under the hood. If an async function is called without an event loop, it can cause errors or warnings (for instance, HuggingFace embedding or LangChain might try to run in async mode).

To ensure compatibility, the implementation proactively manages the asyncio event loop in the Streamlit context. Early in the `rag.py` code, we have:

```python
# Set up async loop for Streamlit
if threading.current_thread().name == "ScriptRunner.scriptThread":
    try:
        asyncio.get_running_loop()
    except RuntimeError:
        loop = asyncio.new_event_loop()
        asyncio.set_event_loop(loop)
```

What this does is: when the code is running in the Streamlit app thread (identified by name "ScriptRunner.scriptThread"), it checks if an asyncio loop is already present. If not (which is usually the case), it creates a new event loop and sets it as the default for that thread. This allows any downstream async calls to use this loop.

We also included `nest_asyncio` in our requirements (and could call `nest_asyncio.apply()` if needed) which can allow nested event loops, but the above solution of explicitly setting a loop was sufficient in our tests. This setup was crucial because without it, certain parts of LangChain (especially when using Runnables or asynchronous chains) might throw `RuntimeError: There is no current event loop` or similar issues on first invocation.

Why is async needed? Some library calls, such as the HuggingFace embedding generation or Guardrails tasks, may use asynchronous HTTP requests or be implemented as async functions. By ensuring an event loop, we allow these to run seamlessly. Additionally, if we wanted to parallelize some operations (like retrieving multiple documents or multiple questions at once), having an event loop would be necessary.

Streamlit's threading model: Streamlit doesn't natively support user-defined background threads or async tasks that persist across runs; each interaction runs the script anew. However, by using `st.session_state` to store our RAG pipeline object and its state, we effectively maintain persistence of the heavy components (like Pinecone index, etc.) across runs. The event loop initialization runs at import time (or first use) and then the loop remains set for that thread context. This means subsequent reruns in the same session can reuse it.

Impact on UI responsiveness: Since the pipeline operations (embedding, Pinecone I/O, LLM calls) can be time-consuming, we use Streamlit’s `st.spinner` and messages to inform the user that work is in progress. We did not implement true asynchronous background tasks; instead, everything runs in sequence triggered by the UI events, which is simpler. One might think to offload the LLM call to another thread or async task to keep the UI reactive, but Streamlit’s architecture doesn’t easily allow partial renders in the middle of a run. Thus, our approach is to perform the steps and update the UI when done (which is acceptable for this use-case where each question might take a few seconds to answer).

In summary, setting up the asyncio loop and understanding Streamlit’s execution model allowed us to integrate advanced libraries smoothly without the app crashing or freezing improperly. The user experience is a simple query-response with appropriate loading indicators, which is standard.

Appendix B - Warm-Up Query

During development, we noticed that the very first query made to Pinecone (and possibly the first call to the LLM) was slower and at times error-prone due to lazy initialization. Specifically, the Pinecone vector store (LangChain's `PineconeVectorStore` or the Pinecone client) defers establishing a full connection or index context until an actual query or upsert is performed. Similarly, the LangChain chain might lazy-load some prompts or models. To address this, we implemented a **warm-up query** as part of the pipeline setup.

In `RAG.setup()` (and `setup_challenge()`), right after creating the retrieval chain, we call `self.warmup_query()` which is defined as:

```
def warmup_query(self):
    # Executing a "warmup" query to solve synchronization issues
    # due to the wrapper object lazily deferring connection to Pinecone until
the first query.
    # The warmup does two important things:
    #   1. Forces the retriever to connect to the index and test its search i
nterface,
    #      ensuring the vector store is usable and fully indexed.
    #   2. Triggers any internal state setup that LangChain or Pinecone clien
t needs,
    #      avoiding lazy-load delays or bugs.
    docs = self.retriever.get_relevant_documents("What is Promtior?")
```

This function issues a dummy retrieval query: **"What is Promtior?"** (Promtior is the name of the company used in the challenge content, so it's a plausible query to test). The question is somewhat arbitrary but relevant enough that if anything were to be retrieved, it should find something in the context. The goal, however, is not the answer itself but to force:

- Pinecone to perform a similarity search on the index (which will establish any needed network connection to the Pinecone service and verify the index is ready).
- LangChain's retriever to fully initialize.
- (If the chain was fully constructed, it could also force the LLM call, but we specifically warm up only the retriever here. We could have also run a full `self.rag_chain.invoke(dummy_question)`, but that would spend LLM credits/time unnecessarily.)

By doing this, when the user asks their first real question, the system is already "warmed up":

- The Pinecone index has been queried at least once (so any internal caches are filled, and the next query should be faster).

- If there were any issues in connecting to Pinecone (like a missing index or a network hiccup), we would find out during setup rather than when the user is waiting for an answer.
- The latency impact of the first query is taken during the setup spinner (where we already inform the user that the pipeline is being created), rather than during the answer phase. Psychologically, users expect some wait when building the index, but once they hit ask, they expect a quick answer; warm-up helps meet that expectation.

In practice, this improved the responsiveness and reliability. Without the warm-up, initial queries sometimes returned empty because the Pinecone index hadn't fully indexed the new vectors yet or the connection was still initializing, and adding a short sleep wasn't always sufficient. With the warm-up retrieval, we explicitly probe the index after insertion, which serves as both a test and a catalyst for readiness.

It's worth noting that the warm-up query used is content-specific ("What is Promtior?"). This is logically something a user might ask given the context we expect (in the challenge scenario, the website PDF content is about Promtior). If the user provided a completely different URL, this question might not be relevant; however, it does not matter much because we ignore the result. We could have used a very generic query (or even an empty vector search), but using a semantically meaningful one has the slight advantage that if the index is working, it might retrieve something (we don't use it, but it's a sanity check to see a non-empty result in logs).

The warm-up is a minor but important implementation detail to ensure a smooth first-time experience. In a more generalized library, one might expect the frameworks to handle lazy loading seamlessly, but when dealing with multiple integrated services, such initialization quirks are common. Our solution is straightforward and effective for the scope of this project.

Appendix C - Guardrails

Integrating NeMo Guardrails added a layer of safety to the chatbot. The guardrails system consists of:

- A policy definition (what is allowed or not allowed for user inputs and bot outputs).
- Tasks that check those policies.
- Flows that determine when to execute those tasks (e.g., before responding to user, after generating an answer, etc.).
- A runtime (the `LLMRails`/`RunnableRails`) that ties it all together with an actual LLM to perform classification or transformation tasks.

In our project configuration:

- The `config.yaml` declares an `input: flows:` section with a single flow: `- self check input`. This means whenever a user input is received, the flow named "self check input" should be executed. (We did not have an output flow yet, but one is suggested in comments for future addition.)
- The `prompts.yaml` provides the prompt content for the `self_check_input` task. The prompt clearly lists the Company policy for user messages, such as no harmful content, no asking the bot to break rules, no explicit content, no leaking system prompts, etc. It then presents the user's message and asks: "Should the user message be blocked (Yes or No)?". This effectively turns the LLM into a classifier for the user's message.

Guardrails Execution: When the `RunnableRails` (guardrails object) is called with an input (the user's question), it will:

1. Load the config (we did `RailsConfig.from_path("./config")` which reads the YAML files in the `config` directory).
2. Find the relevant flow for input (self_check_input).
3. Use the provided `llm` (we passed `self.groq_llm`, the same 70B model) to execute the prompt from `prompts.yaml` with the user's message.
4. Parse the LLM's answer (it expects "Yes" or "No").
5. Depending on the result, mark the input as safe or not.

In our current code, after constructing the chain with guardrails, if the guardrail finds a violation, what happens? We did not explicitly write code to handle a "blocked" decision (e.g., abort the chain). The `RunnableRails` might itself throw an exception or alter the input. By default, NeMo

Guardrails could be configured to, say, replace the user query with something else or inject a system message. However, we kept it simple. To properly enforce it, we could do something like:

```
result = self.guardrails.invoke(user_query)
if result.get("blocked", False):
    return "I'm sorry, I cannot continue with that request."
```

But since we integrated guardrails into the chain seamlessly, we rely on its internal behavior. For now, one can think of it as **monitoring mode**: it will log or could be observed via guardrail logs whether the query was okay.

During development, we tested the guardrails by inputting a clearly disallowed query (e.g., something with profanity or asking the bot to break rules) to see if the guardrail would catch it. The classification did work (we saw the LLM output "Yes" for block in such cases), but because we hadn't implemented a user-facing block message, the chain still went on to answer (since the LLM doesn't actually have knowledge to break rules, it just answered generically or with refusal on its own). For a production-ready solution, we would refine this by making the guardrail actively intercept (perhaps by modifying the chain such that guardrail returns a special response if blocked, bypassing the normal chain).

Why use the same LLM for guardrails? We pointed guardrails to use the Llama3 model for evaluating the policies. This is somewhat meta – using the LLM to analyze user input – but it's a common approach: large models can be good at understanding instructions and content moderation tasks. We could have used a smaller or separate model (like a dedicated moderation model or even a rule-based approach) but using the provided large model ensured consistency and avoided needing an additional API or dependency. The drawback is cost and latency: a 70B model call for each user input just to check safety is expensive. For future iterations, using a cheaper classifier (like OpenAI's moderation endpoint or a small BERT-based classifier) would be a better trade-off for performance.

Guardrails Value: Even in its minimal state, including guardrails shows an important consideration for AI systems: not all user inputs should be answered, and having a mechanism to enforce that is necessary. It also provides a framework to later add more complex behaviors (for example, one could add a guardrail that if the user asks something outside the knowledge scope, the bot should politely say it doesn't know, rather than hallucinate – this could be done by checking if retrieval returned anything and then instructing the bot accordingly). NeMo Guardrails is quite flexible in this regard.

Security Note: We allowed users to input any URL for augmentation. This could be a security concern because if a user inputs a URL to malicious or very large content, the bot would go and fetch it. We did implement basic content cleaning and rely on the scope of the challenge (the user is likely to input well-formed, relevant URLs). In a broader deployment, it might be wise to restrict to certain domains or file size limits, and to use guardrails on output as well to ensure nothing from the content that is sensitive gets echoed blindly.

Appendix D - Data Ingestion and Preparation

When a user provides a URL (either an HTML page or a PDF link), the system begins by **loading and preprocessing the content** to build the retrieval index. This process is encapsulated in the `RAG.setup()` method for general use, and a specialized `setup_challenge()` method (used to preload a known challenge dataset).

HTML Content Loading: For web pages, the implementation uses Python's `requests.get` to fetch the page content, specifying a **user-agent header** (to mimic a real browser and avoid blocking by some websites). The HTML text is then parsed with **BeautifulSoup**:

```
response = requests.get(url, headers={"User-Agent": "Mozilla/5.0 ..."})
soup = BeautifulSoup(response.text, "html.parser")

# Remove non-content elements
for tag in soup(["script", "style", "footer", "nav", "meta", "noscript"]):
    tag.decompose()
raw_text = soup.get_text(separator="\n", strip=True)
```

In this snippet, after fetching the page, we remove script, style, footer, navigation, meta, and noscript tags. This cleaning step is crucial to strip out irrelevant text (like navigation menus or JS code) that could otherwise confuse the embeddings. We then extract the visible text (`get_text`) with newlines as separators, yielding a clean `raw_text` string containing the core content of the page.

PDF Content Loading: If the URL ends with `.pdf` (or for local PDF files), the code uses **PyPDF2** to read the PDF. The content is fetched (via `requests` for remote PDFs), then:

```
reader = PdfReader(BytesIO(response.content))
raw_text = "\n".join(page.extract_text()
                     for page in reader.pages if page.extract_text())
```

Each page's text is extracted and joined with newlines. PyPDF2 handles PDF decoding; however, it may not perfectly preserve structure or read images, so this approach works for text-based PDFs (which is acceptable for our use-case of textual QA).

Text Splitting: The aggregated `raw_text` from either source is then split into chunks using LangChain's `RecursiveCharacterTextSplitter`. In our configuration, we set `chunk_size=768` characters and `chunk_overlap=128`. This splitter will try to cut the text at natural boundaries (like periods or newline) close to 768 chars but ensure that overlapping content (128 chars from the end of the previous chunk) is included at the start of the next chunk. This overlap ensures continuity, so if an answer spans a boundary, it is still captured. We found these values to strike a good balance for the typical size of web articles and PDFs in our tests. For instance, if a webpage is 5,000 characters long, it might be split into around 7–8 chunks that are easier for the embedding model to handle.

Embedding Generation: Once chunks are ready, each chunk is converted to a vector embedding. We instantiate the HuggingFace embedding model at startup:

```
self.embeddings = HuggingFaceBgeEmbeddings(
    model_name="BAAI/bge-small-en-v1.5",
    model_kwargs={"device": "cpu"},
    encode_kwargs={"normalize_embeddings": True}
)
```

This sets up the model to produce normalized embeddings (we enabled normalization here, which could improve dot-product similarity results by having unit-length vectors). The `embed_documents()` method is then applied on the list of text chunks, yielding a list of 384-dimensional vectors (since the BGE model outputs 384-length embeddings). The choice of CPU for device is for simplicity; if a GPU were available, using it would speed up embedding computation significantly, but for moderate text lengths CPU is manageable.

Indexing in Pinecone: With embeddings ready, the system then stores them in the Pinecone index. Before adding new data, we ensure a fresh index is used:

- If this is a new context (first time or a new URL), we **create a Pinecone index** (via `pinecone.create_index`) or reset an existing one. Our code checks if an index name already exists and deletes it to avoid mixing data from different contexts.
- We create the index with specified dimension (384) and metric (dotproduct). Pinecone's asynchronous index creation means we also poll `describe_index` until `status["ready"]` is True, to ensure the index is fully ready to accept inserts (this usually takes a few seconds for a new index).

Once the index is ready, we create a `PineconeVectorStore` instance (LangChain's wrapper around Pinecone) and add the texts:

```
self.vectorstore = PineconeVectorStore(
    index_name = self.vectorstore_index_name,
    embedding = self.embeddings,
    pinecone_api_key = self.pinecone_api_key
)
# ...
self.vectorstore.add_texts(
    texts = chunks,
    embeddings = embeddings,
    ids = [f"{i}" for i in range(len(chunks))]
)
print(f"Added {len(chunks)} chunks to vectorstore.")
time.sleep(7)  # brief pause to ensure Pinecone has indexed all data
```

We supply explicit IDs for each chunk (just stringified indices 0..N) because Pinecone requires unique IDs. After adding, a short `time.sleep(7)` is used as a safety delay to allow Pinecone's internal index to finalize the insert operations before we proceed (in practice, Pinecone operations are fast, but this was added to avoid edge cases where an immediate query might not find freshly inserted vectors due to eventual consistency).

At this point, the data ingestion and preparation phase are complete: the content from the URL is now embedded and indexed in Pinecone, ready for retrieval. The system sets a flag `rag_ready = True` to indicate that the pipeline is built and questions can be answered. If a new URL is entered later, the above process is repeated (creating a new index and dropping the old one).

Appendix E - Vector Store Initialization (Pinecone)

Initializing and managing the Pinecone vector store is a crucial part of the setup. Some details about this step, beyond what was covered above, include how we handle **index naming and recreation** and the configuration of Pinecone:

- **Index Naming:** We use a fixed index name (e.g., `"valet"`) for simplicity, recreating it for each new context. This means at any given time, only one dataset is indexed (the latest one). The design decision here was to avoid complexity of handling multiple indexes or combining sources; since the application is interactive and one-at-a-time (the user provides one URL, asks multiple questions, then possibly resets with another URL), a single index suffices. The trade-off is that if we wanted to allow *multiple* sources to be live simultaneously or persist an index for reuse, we'd need to incorporate logic for unique index naming per session or user, which could be a future enhancement.

- **Index Dimensions and Metric:** The dimension 384 must match the embedding model's output. We chose **dot product** as the similarity metric. If we ensure embedding normalization, dot product is equivalent to cosine similarity. Pinecone also supports metrics like cosine and Euclidean, but dot product is a common choice for embeddings especially if not normalized (it effectively measures similarity in a way akin to unnormalized cosine). We manually ensure normalization via the embedding call for consistency.

- **Resetting Index:** By deleting any existing index of the same name before creating a new one, we handle the case where a previous context's index remains. This ensures no data leakage between different contexts. Pinecone's deletion is immediate in terms of removing metadata, but we still create a new one with the same name after. One must consider that creating and deleting indexes frequently can be an expensive operation; for a production scenario with large data, it might be better to reuse an index and just replace its content (or maintain separate named indexes for separate projects). In our case, given moderate data sizes and a focus on simplicity, the create-delete pattern is acceptable.

- **Pinecone Client Handling:** The code uses `pinecone.Pinecone()` to instantiate a client and `ServerlessSpec` to specify the environment. These calls are part of Pinecone's Python client usage. We stored the API key via environment variable and loaded it (ensuring not to hard-code secrets). The Pinecone client lazily establishes a connection; indeed, we found that the first operation (like listing or creating index) might incur some overhead to connect to Pinecone service. This ties into the **warm-up logic** discussed later, where ensuring an early operation can mitigate delays.

- **Retriever Setup:** After index creation and populating data, we create a retriever handle: `self.retriever = self.vectorstore.as_retriever()`. This retriever is essentially a lightweight wrapper that uses the Pinecone index behind the scenes. It is configured by default to retrieve a certain number of top matches (LangChain's default is often 4 or so). We could configure it to use a specific search query or filter if needed, but in our case the default of returning the most similar chunks to the question is sufficient.

By successfully initializing the Pinecone vector store and loading it with content embeddings, the system sets the foundation for high-speed similarity search when the user asks questions. This concludes the setup portion of the pipeline. Next, we describe how the question answering (QA) chain is constructed and executed.

Appendix F - RAG Chain and Query Processing

With the data indexed, the system enters the **question-answering phase**. When the user inputs a question through the UI (after a context has been built), the `RAG.qa(question)` method is invoked to produce an answer. The QA process is orchestrated by a **LangChain retrieval chain**, which we set up as follows:

```
# Define the retrieval-augmented generation chain
self.rag_chain = (
    {
        "context": self.retriever | self.format_docs,  # retrieve docs and fo
rmat them
        "question": RunnablePassthrough()              # pass the user's ques
tion unchanged
    }
    | self.rag_prompt           # combine context + question into a prompt t
emplate
    | self.groq_llm             # call the LLM (Groq-hosted Llama3)
    | StrOutputParser()         # parse the LLM's response to a string
)
# Prepend guardrails to the chain
self.rag_chain = self.guardrails | self.rag_chain
```

Let’s break down this chain definition:

- We use LangChain’s **dictionary-to-chain** mapping capability, where we specify that the chain expects two inputs: `"context"` and `"question"`.
    - For `"context"`, we provide a pipeline: `self.retriever | self.format_docs`. This means the retriever will be called (with the overall input, which by default LangChain will feed the same user query to both retriever and question since we call `invoke(query)` with a single argument). The retriever returns a list of documents (the top relevant chunks from Pinecone). That output is then fed into `self.format_docs`, which is a simple function we defined to join the document texts into one string. Essentially, `"context"` becomes a single string containing the text of all retrieved chunks (separated by newlines). This will later fill the `{context}` slot in the prompt.
    - For `"question"`, we use `RunnablePassthrough()`, which simply passes the original input (the user’s question text) forward unmodified. This fills the `{question}` slot in the prompt.
- `| self.rag_prompt`: This pipes the two inputs into the prompt template. `self.rag_prompt` was loaded from `hub.pull("rlm/rag-prompt", ...)` and is a

LangChain prompt template expecting `context` and `question`. It likely has a structure like: *"Given the following context: {context}the question: {question}…"* (the exact wording can be customized, but using a standard one ensures a well-structured prompt). This step produces a fully formatted prompt string ready for the LLM.

- `| self.groq_llm`: This sends the prompt to the LLM (Groq's Llama3 model). The `ChatGroq` instance was configured to use the 70B model with a fixed temperature, so it will return a completion (the answer text). LangChain's `ChatGroq` integration likely returns a `LLMResult` or chain-specific object, but since we pipe it into `StrOutputParser()`, we ensure to get a raw string out.

- `| StrOutputParser()`: This finalizes the chain by taking the LLM's output and converting it to a plain string (if it isn't one already). This is useful if the LLM output is in a LangChain message format or includes any metadata. The result after this parser is the answer string that we can present to the user.

- Finally, we integrate **Guardrails** by doing `self.guardrails | self.rag_chain`. Here, `self.guardrails` is a `RunnableRails` object that wraps our guardrails config. Placing it to the left of the chain means it will take the user input first when the chain's `invoke()` is called. Essentially, the guardrail will receive the user's question, execute the `self_check_input` flow (as defined in config), and then pass the original (or potentially modified) question on to the next stage in the chain. If the guardrail were to determine the input is disallowed, it could also short-circuit or modify the input. In our current setup, we expect it mostly to log/flag if needed, but we still feed the question through. This integration ensures that every question goes through the policy check prior to retrieval and generation.

When a user question comes in, we call `answer = self.rag_chain.invoke(user_query)`. Under the hood, LangChain will:

1. Feed `user_query` into the guardrails `RunnableRails`. The guardrail runs the policy check prompt using the LLM. If the query violates policies, ideally it would prevent further steps; in our simple case, it might just print a warning. (This could be extended so that if guardrail outputs "Yes (block)", we return a message like "I'm sorry, I cannot assist with that request.")

2. Assuming we continue, the query is then passed to the retriever and `RunnablePassthrough` in parallel as described, retrieving docs and preparing inputs.

3. The prompt template combines the results, producing a final prompt text like:

```
<Instructions and context prefix...>
Context:
"... (retrieved chunk text 1)..."
"... (retrieved chunk text 2)..."
...
Question: <user question>
```

(The exact format depends on the `rag-prompt` template content.)

4. The prompt goes to the `ChatGroq` LLM, which returns an answer. The chain captures that and the output parser ensures it's a clean string.

5. The `invoke` returns the final answer string.

This answer is then returned by our `qa()` method to the Streamlit app, which displays it to the user.

Why use LangChain's chain instead of manual calls? The chain abstraction allows easier insertion of intermediate steps and parallel branches (like splitting the input to context and question as we did). It also integrates well with the guardrails as a `Runnable` component. By using the chain, our `qa()` function remains very simple – just invoking the chain – rather than having to manually call retriever, then format, then call LLM, etc., and manage all the intermediate states.

Formatting of Retrieved Docs: One consideration is how we feed the retrieved text to the LLM. Our `format_docs` function currently just concatenates the text of the documents with some spacing. We considered prefixing each chunk with a source or metadata (for example, "Document 1 says: …"), or limiting the number of chunks. In practice, LangChain's `as_retriever` default might return a few top docs (say 4). If those docs are large, the combined context plus the question must remain within the LLM's context window (8192 tokens for Llama3-70B in this case). We assume typical inputs (webpages, etc.) won't exceed that when chunked and filtered, but it's a factor to monitor. The advantage of a large context model is that we can include quite a bit of text; the disadvantage is if we include too much irrelevant text, it could confuse the model or hit length limits. Our approach is to trust the vector similarity to bring back only relevant chunks and keep the prompt focused.

Conversation History: Our application also maintains a conversation history (visible in the UI as "Conversation History"). Each Q&A pair is stored in `st.session_state.history`. However, note that in the current pipeline, we **do not feed the history back into the model for context** – each query is answered based solely on the provided URL's content (plus the question). We treat each question as independent for retrieval purposes, which is appropriate for factual Q&A on a given document/website. If we wanted a true multi-turn conversational memory, we might concatenate previous questions/answers as additional context or use LangChain's conversation memory components. We chose not to do that here to avoid compounding errors and because the nature of the task (ask questions about the document) doesn't demand remembering past questions (they can always refer back to the content directly). This is a deliberate simplification (trade-off: simpler logic and guaranteed retrieval accuracy vs. lacking conversational nuance or follow-up questions dependency).

Handling Multiple Sources: Currently, the pipeline builds context from one source at a time. If we wanted to support multiple URLs or documents together, we would either merge their text before chunking or maintain multiple indexes. That's a potential extension not implemented in this version.

In summary, the RAG chain uses retrieval to supply the LLM with knowledge, enabling accurate answers. The integration of guardrails at the chain’s start is a unique aspect ensuring each query is vetted. Now, beyond the main logic, there were a couple of important technical considerations in implementation that we highlight next: making the pipeline work smoothly within Streamlit’s environment and mitigating the first-query latency through a warm-up.