Promtior Challenge RAG chat assistante

–

Faivel Fedullo Silva
January 2nd

Chatbot Frontend:
http://promtior-alb-198600562.us-east-1.elb.amazonaws.com/ui/

Chat Langserve Playground:
http://promtior-alb-198600562.us-east-1.elb.amazonaws.com/rag/playground/

Github page:
https://github.com/FaivelFedulloSilva/promtior-rag-challenge

Overview

This document describes the approach, design decisions, and implementation of Promtior's technical challenge.

Goals

1. Implement a RAG chatbot capable of answering questions about Promtior based on the content of its public website and the documentation provided as part of the technical challenge.
2. Deploy the application in AWS.

Approach & Design Decisions

After reviewing the challenge requirements, the solution was designed with the following capabilities:

1. Crawl Promtior's public website, extracting text content and following internal links to cover the full website structure.
2. Extract relevant text information from the provided PDF documentation.
3. Clean the raw data to reduce noise and remove non-informative content.
4. Chunk the processed data into manageable segments optimized for semantic retrieval.
5. Generate embeddings for each chunk and store them in a vector database.
6. Implement a RAG pipeline capable of retrieving relevant information from the vector store and injecting it as context into the prompt sent to the LLM.
7. Expose an API endpoint using LangServe to allow external interaction with the RAG pipeline.
8. Include source references in the generated responses to improve transparency and traceability.

Data Collection and Storeage

Once the overall architecture was defined, the solution was implemented incrementally, starting from data ingestion and progressing towards inference and deployment.

The first component developed was the web crawler. It was implemented using Crawl4AI, a library that addresses common web crawling challenges such as link discovery and content extraction.

Unlike traditional crawlers that return raw HTML, Crawl4AI produces a cleaned markdown representation of each page, which significantly simplifies the downstream data ingestion and preprocessing stages.

Once the crawler was validated, the ingestion module was implemented. For local development, ChromaDB was selected as the vector store due to its lightweight nature, ease of integration, and suitability for local workflows.

The ingestion module consumes the output of the crawler, provided as a .jsonl file, and chunks the extracted content using LangChain's RecursiveCharacterTextSplitter. A chunk size of 1800 characters with an overlap of 200 characters was selected as a trade-off between contextual completeness and retrieval precision.

Finally, embeddings were generated for each chunk using the text-embedding-3-small model, and stored in the vector database for later semantic retrieval during inference.

Challenges Encountered

The main challenge encountered during this stage of the project was data cleaning. Removing boilerplate content typical of HTML web pages, such as navigation menus, footer text, and image links, proved difficult to solve using generic tools alone.

Several existing libraries, including Trafilatura, were evaluated in an attempt to automatically extract only the relevant text content. However, the results were comparable to the cleaned markdown already produced by Crawl4AI, and did not fully eliminate non-informative sections.

Given these limitations, a pragmatic approach was adopted. The remaining noisy sections were removed using custom, rule-based cleaning methods specifically designed to target known structural elements of the Promtior website.

This approach provided better control over the final dataset while keeping the overall ingestion pipeline simple and predictable.

Backend

The backend was implemented as a LangServe application that exposes a RAG pipeline built with LangChain. Its main responsibility is to receive a user question, retrieve the most relevant context from the persisted vector store, and generate a grounded answer using an OpenAI chat model.

Core Components

Vector Store (ChromaDB)

The backend loads a persisted ChromaDB collection from a local directory and uses a fixed collection name. This ensures the runtime API does not need to rebuild the index; it only consumes the existing embeddings generated during ingestion.

Embeddings (OpenAIEmbeddings)

The same embedding model used during ingestion is used in the backend (text-embedding-3-small). This is critical, the embedding function used at query time must match the one used to build the index, otherwise retrieval may not find any relevant information.

Retriever (MMR)

Document retrieval is performed via Chroma's retriever interface using **Maximal Marginal Relevance (MMR)**. MMR is used to reduce redundancy in results (e.g., multiple chunks from the same page with similar text) and increase coverage of different relevant sections of the knowledge base.

The retriever is configured with:

- k = 10: number of chunks returned to the LLM as final context
- fetch_k = 40: candidate pool size before diversification
- lambda_mult = 0.5: trade-off between relevance and diversity (balanced setting)

Prompting Strategy

A ChatPromptTemplate was used with a **system message** that enforces two key behaviors:

1. **Grounding constraint**: the assistant must answer *only* using the provided context.
2. **Unknown handling**: if the answer is not explicitly present in context, it must respond:

   "I don't know based on the provided context."

This constraint is intended to minimize hallucinations.

Context Formatting and Source Attribution

Retrieved documents are transformed into a compact context string via a formatting step. Each chunk is prepended with a [SOURCE] tag containing the originating URL found in the document metadata. This formatting serves two purposes:

- It keeps the context readable and structured for the LLM.
- It enables the model to include a **Sources** section in the final answer listing the URLs used.

This design choice ensures traceability and directly supports the challenge requirement of including sources in responses.

Inference Configuration

Answer generation is performed using ChatOpenAI with:

- OPENAI_CHAT_MODEL (default: gpt-4o-mini) as a cost/latency-optimized model selection.
- temperature = 0 to encourage deterministic, factual responses aligned with the retrieved context.

API and LangServe

The backend is exposed as a REST API built on top of FastAPI and LangServe. LangServe is used to wrap the LangChain RAG pipeline and expose it as an HTTP-accessible service with minimal boilerplate, allowing the application to focus on chain design rather than request handling logic. The RAG chain is instantiated once at application startup and reused across requests, ensuring efficient access to the vector store and retriever without unnecessary reinitialization.

The API exposes the RAG pipeline under the `/rag` endpoint and includes a lightweight `/health` endpoint for deployment verification and monitoring. In addition, the same FastAPI application serves a static frontend build when available, enabling a simple chatbot interface for interaction and testing while keeping the API and UI deployment unified.

End-to-End Request Flow

At runtime, the backend processes a request as follows:

1. The API receives a user question.
2. The question is embedded using the configured embeddings model.
3. The retriever performs an MMR-based similarity search against ChromaDB.
4. Retrieved chunks are formatted into a single context string, preserving source URLs.
5. The prompt template injects {context} and {question} into the final prompt.
6. The LLM generates an answer grounded in context and includes a **Sources** section.
7. The response is returned as plain text.

Challenges Encountered

During testing, some questions were answered using suboptimal sources. A representative example was:

"What services does Promtior offer?"

The expected primary source for this answer was Promtior's dedicated Services page. However, the system consistently responds based on the website homepage.

With further inspection it was found that the homepage contains the word "service" (and related variations) multiple times, while the Services page mentions it mainly in the title and relies more heavily on service-specific descriptions and headings.

As a consequence, similarity-based retrieval tended to favor homepage chunks because they contained stronger lexical signals aligned with the user query.

To reduce redundancy and increase coverage of different pages, the retriever was configured to use MMR. The intention was to promote diversity across retrieved chunks and reduce the probability of selecting multiple homepage chunks that were semantically similar.

Although MMR improved the diversity of retrieved chunks in general, the system still frequently selected homepage content over the Services page for this specific query. Within the scope and time constraints of the challenge, the retrieval behavior could not be consistently corrected for this case.

Frontend

For completeness, a minimal chat-like frontend was implemented. It allows users to interact with the backend in a more user-friendly manner by displaying questions and responses in a familiar chat interface. Unlike the default `/rag/playground` endpoint, this frontend preserves the conversation history, enabling users to review previous questions and answers instead of losing them on each interaction.

Deployment

The application was deployed to AWS as a containerized service. Docker was used to package the backend API, the LangServe-based RAG pipeline, and the static frontend into a single, self-contained artifact. This approach ensures consistency between local development and the production environment, and simplifies dependency management and deployment.

All runtime configuration is handled through environment variables, including API keys, model selection, and embedding configuration. This allows the same container image to be reused across different environments without code changes, while keeping sensitive information out of the source code.

The service exposes both the RAG API and the frontend from the same FastAPI application. This unified deployment model reduces infrastructure complexity and makes the solution easier to evaluate, as no additional services are required to interact with the system.

A lightweight health endpoint is provided to validate that the application is running correctly after deployment. This endpoint can be used for basic monitoring and readiness checks in the cloud environment.

Overall, this deployment strategy prioritizes simplicity, reproducibility, and clarity, which are appropriate for the scope and goals of the technical challenge.

Challenges Encountered

As I had no prior experience with AWS services, setting up a working deployment environment proved to be challenging. In addition, my initial knowledge of Docker was limited, which made the dockerization and deployment of the application a significant learning process. Multiple iterations were required to refine the Dockerfile structure, correctly configure environment variables, and align the internal file system layout of the container with the application's expectations.

One of the main issues encountered was related to serving the frontend in the deployed environment. While the application worked correctly when running locally, the deployed version was unable to locate and render the frontend. After several attempts, the root cause was identified as an incorrect directory path: the backend was configured to look for the frontend build in a directory that did not exist inside the Docker container. Once this mismatch was corrected, the frontend was successfully served.

Additionally, the multi-step configuration process required to deploy the application on AWS was another considerable challenge. Identifying where configurations should be set, how they should be defined, and how different AWS components interacted was not straightforward and required careful experimentation and iteration.

Conclusions and future work

This project successfully delivers an end-to-end RAG-based chatbot capable of answering questions about Promtior using grounded information extracted from its public website and provided documentation. The solution covers the complete workflow, from data collection and ingestion to inference, API exposure, and cloud deployment.

Although the main idea behind the choices made for the architecture was to not overengineer the solution, make it as close to what it was ask as possible, there are features and pieces of the software that I would add or change if time were not limited, or if I were to implement a new version of the application.

Some potential improvements and future work include:

1. This application was meant to answer questions based on a single webpage and a single pdf file. For this requirement, ChromaDB was considered a suitable choice, but, if the application were meant to be a production software, served to users, basing its answers in a broader collection of webpages and documents, I would switch to Pinecone database.

2. I would further research how to fully clean the row data. A considerable part of the data stored in the vector store is noise, which degrades the accuracy of the answers. A better data cleaning before the data storage would improve the quality of the answers.
3. Due to time constraints, commonly accepted defaults were used for the embedding model, chunk size and overlap configurations. For a second version of the application, I would research and iterate over a few other embedding models, and perform some data analytics, to better set the chunk size and overlap, finding a better fit for the specific data that the application works with.
4. Currently, the application runs the web-crawl and the data ingestion as local script, outputting the ChromaDB vector store filled, which is embedded into the container, which is then deployed into AWS. For a new version, I would expose a dedicated endpoint to re-run crawling and ingestion processes, enabling the knowledge base to be updated without redeploying the application
5. Although short term memory was not a requirement of the challenge, I consider it to be an addition that may improve the user engagement. Then, if a new version were to be implemented, I would add the short term conversational memory to the application.

Final Words

Although the core implementation of the RAG architecture itself was not particularly challenging, the complete workflow, from data collection to deployment on AWS, represented a significant learning experience.

My prior experience with RAG systems was mainly limited to procedural implementations, developed and executed in notebook-based environments such as Google Colab, where the entire workflow is treated as a single script. This challenge required a different mindset: approaching RAG not as a standalone script, but as one component within a bigger system architecture that includes ingestion pipelines, APIs, deployment, and operational concerns.

Overall, this challenge proved to be a valuable opportunity to measure my ability to quickly learn, adapt, and integrate technologies with which I had little or no prior experience, while delivering a complete, end-to-end solution.

Disclaimer

Given that Promtior is a company focused on making Generative AI more accessible to organizations and users, the use of Generative AI tools was considered not only acceptable but encouraged. Generative AI was used both during the development of the application and in the preparation of this document, as a support tool for ideation, iteration, and refinement.

All architectural decisions, implementation choices, and final content were reviewed and validated to ensure technical correctness and alignment with the challenge requirements.