# Zomato Gen AI Internship Assignment

## Restaurant Data Scraper & RAG-based Chatbot

Assignment Report

submitted by

## Bhavya Garg

Undergraduate 3rd year student in Biological Sciences and Bioengineering  
Indian Institute of Technology

**Overview**

This project was my dive into building a Retrieval-Augmented Generation (RAG) pipeline to answer questions about restaurant menus and details, using natural language processing (NLP) and vector-based retrieval. As someone with a bit of machine learning experience, this was my first time tackling web scraping and RAG, which made it both challenging and exciting. The journey unfolded in two phases: a working initial model and an ambitious (but bumpy) attempt to improve it. Below, I’ll walk you through what I did, how I did it, the hurdles I faced, and what I learned along the way. This document reflects the project’s state as of April 21, 2025.

**Personal Learning**

This project was a whirlwind of learning. Web scraping was completely new to me, and I loved figuring out how to extract clean text from messy websites. While I’d worked with machine learning models before, RAG opened up a new world of combining retrieval and generation. Despite some frustrating moments, it was a fun and rewarding experience that pushed me to grow.

**Phase 1: Building the Initial Working Model**

**Workflow Summary**

My goal was to create a system that could fetch relevant restaurant information from the web, store it efficiently, and answer user queries like “What vegetarian options are available at O\_Pedra?” Here’s how I approached it:

1. **Target Selection**  
   I focused on websites with restaurant menus, FAQs, and detailed descriptions. I prioritized pages with structured HTML to make parsing easier, as messy layouts were a headache to deal with.
2. **Tools Used**  
   I leaned on a few key Python libraries:
   * requests: To fetch web pages via HTTP requests.
   * BeautifulSoup: For parsing HTML and pinpointing the tags I needed.
   * trafilatura: A lifesaver for extracting clean, readable text from complex web structures.
3. **Scraping Process**  
   Here’s a breakdown of how I scraped the data:
   * **Fetch HTML Content**: I used requests to pull the raw HTML from a webpage and parsed it with BeautifulSoup. For example:
   * import requests
   * from bs4 import BeautifulSoup
   * url = 'https://example.com/menu'
   * response = requests.get(url)

soup = BeautifulSoup(response.text, 'html.parser')

* + **Extract and Clean Text with Trafilatura**: To get high-quality text, I used trafilatura to strip away clutter like ads or navigation menus:
  + import trafilatura
  + downloaded = trafilatura.fetch\_url(url)

text = trafilatura.extract(downloaded)

* + **Text Post-processing**: I cleaned up the extracted text by removing extra whitespace, boilerplate content (like footers), and irrelevant sections. The cleaned text was saved in plain text format for later chunking and embedding.

1. **Error Handling**  
   I added safeguards to handle common scraping issues:
   * Basic exception handling for 404 errors, timeouts, or malformed URLs.
   * Delays between requests and custom user-agent headers to avoid getting blocked by websites.

**Challenges Faced**

Scraping wasn’t as straightforward as I’d hoped:

* Some websites loaded content dynamically with JavaScript, which requests couldn’t handle.
* Poorly structured HTML forced me to experiment with different BeautifulSoup selectors.
* The quality of content varied wildly, with some pages offering rich menu details and others barely providing anything useful.

**Outcome**

Despite the challenges, Phase 1 was a success:

* I built a diverse dataset of restaurant-related content.
* The RAG system could answer queries from a broader context, like dietary options or pricing.
* The scraped content was chunked and embedded for semantic search, laying the foundation for the pipeline.

**Next Improvements**

Looking ahead, I planned to:

* Use tools like Selenium or Playwright to scrape JavaScript-heavy pages.
* Add automatic filtering to prioritize relevant domains.
* Store metadata (e.g., source URL, access date) for better traceability.

**Environment Setup**

To get the RAG pipeline running, I set up my environment in a Kaggle notebook:

* **Libraries Installed**: I used numpy==1.26.4, torch==2.5.1, transformers==4.44.2, sentence-transformers==2.7.0, langchain==0.2.10, faiss-cpu, and langchain-huggingface.
* **Verification**: I wrote a script to confirm all packages were installed correctly and compatible.
* **Challenges**: I initially tried faiss-gpu but switched to faiss-cpu after installation issues in Kaggle.

**Vector Store Creation**

I created a FAISS vector store to index the scraped content for fast retrieval:

* **Approach**: Using langchain.vectorstores.FAISS and embeddings from HuggingFaceEmbeddings with the BAAI/bge-base-en-v1.5 model.
* **Code**:
* from langchain.vectorstores import FAISS
* from langchain.embeddings import HuggingFaceEmbeddings
* embedding\_model = "BAAI/bge-base-en-v1.5"
* embeddings = HuggingFaceEmbeddings(model\_name=embedding\_model)
* db = FAISS.from\_documents(\_chunked\_docs, embeddings)

retriever = db.as\_retriever(search\_type="similarity", search\_kwargs={"k": 4})

* **Outcome**: The vector store successfully indexed the documents, enabling similarity-based retrieval of relevant chunks.

**RAG Pipeline Setup**

The RAG pipeline was the heart of the project, combining retrieval and generation:

* **Approach**: I used langchain.prompts.PromptTemplate to craft a prompt that guided the model to answer restaurant-related queries.
* **Code**:
* from langchain.prompts import PromptTemplate
* prompt\_template = """
* Answer the question based on the following context about restaurant menus and information. Provide accurate and helpful details about menu items, dietary options, prices, or restaurant details as requested. If the query is unclear or out of scope, politely ask for clarification or state that the information is unavailable.
* """

prompt = PromptTemplate(input\_variables=[], template=prompt\_template)

* **Outcome**: The pipeline worked decently, answering queries like “What vegetarian options are available at O\_Pedra?” by retrieving relevant chunks and generating responses. However, it wasn’t perfect due to dependency issues and retrieval inconsistencies.

**Challenges in Phase 1**

* **Dependency Issues**: I hit errors with numpy, torch, and sentence-transformers, requiring multiple reinstalls and version tweaks.
* **Performance**: Retrieval was functional but sometimes missed the mark, possibly due to embedding model mismatches or FAISS indexing quirks.

**Phase 2: Attempting to Improve the Model**

**Motivation**

I wanted to make the pipeline more accurate, robust, and scalable. My goals were:

* Improve retrieval accuracy.
* Enhance response quality.
* Fix dependency conflicts for a stable environment.

**Environment Updates**

I upgraded the environment to address compatibility issues:

* **Libraries Installed**:
* !pip uninstall -y numpy torch torchvision torchaudio transformers sentence-transformers langchain faiss-cpu pydantic
* !pip install -q --force-reinstall numpy==1.26.4
* !pip install -q torch==2.5.1 torchvision==0.20.1+cu124 torchaudio==2.5.1 --index-url https://download.pytorch.org/whl/cu124
* !pip install -q transformers==4.44.2 sentence-transformers==2.7.0 faiss-cpu
* !pip install -q langchain==0.3.0 langchain-community streamlit pypdf unstructured jq
* !pip install -q pydantic==2.11.3

!pip install -q langchain-huggingface

* **Verification**: I checked that langchain==0.3.0 and pydantic==2.11.3 were correctly installed.
* **Challenges**:
  + **Pydantic Mismatch**: Dependencies like unstructured-client required pydantic>=2, but langchain==0.2.10 expected pydantic v1, causing conflicts.
  + **Dependency Resolution**: I faced ResolutionImpossible errors due to mismatched langchain-community and langchain-core versions.
  + **Incomplete Installation**: Some runs failed to install langchain, leading to ModuleNotFoundError.

**Vector Store and Pipeline Refinement**

I reused the FAISS vector store setup with updated langchain syntax and tested the pipeline in the new environment. Unfortunately, the dependency issues derailed progress.

**Outcome of Phase 2**

Phase 2 didn’t go as planned:

* The upgraded model never stabilized due to pydantic version mismatches, causing errors like PydanticUserError.
* Dependency conflicts made the pipeline unusable.
* With the deadline looming, I reverted to the Phase 1 model for submission, as it was partially functional.

**Final State**

* **Submitted Model**: The Phase 1 model, built with langchain==0.2.10, pydantic==1.10.12, and basic RAG functionality, was submitted. It handles queries with moderate accuracy.
* **Limitations**:
  + Inconsistent retrieval performance.
  + Potential embedding model mismatches.
  + Fragile dependencies.
* **Recommendations for Future Work**:
  + Standardize on langchain>=0.3.0 and pydantic v2 to resolve conflicts.
  + Experiment with FAISS settings (e.g., larger k values) or alternative embedding models like all-MiniLM-L6-v2.
  + Test in a local virtualenv to avoid Kaggle’s environment quirks.

**Conclusion**

This project was a rollercoaster, but I’m proud of what I accomplished. Phase 1 delivered a working RAG pipeline that could answer restaurant queries, proving the concept worked. Phase 2’s dependency nightmares were frustrating, but they taught me a lot about environment management. The submitted model isn’t perfect, but it’s a solid foundation. I’m excited to keep exploring RAG and web scraping in future projects, hopefully with fewer version conflicts!