Healthy Recipe Chatbot using Flask, Spoonacular API, and a Local Dataset

For this project, I created a chatbot that helps users discover healthy recipes. The chatbot responds to natural language queries like “high protein lunch” or “low calorie vegan dinner” by searching both a local dataset and a live API. The goal was to build a lightweight, accessible tool that combined real-time flexibility with fast local lookups.

To achieve this, I selected two complementary data sources. The live data came from the Spoonacular API, a robust and well-documented service that provides detailed nutritional information and recipes. This allowed the chatbot to return up-to-date results when local matches weren’t found. The local dataset came from a Kaggle source titled “Food Nutrition Dataset,” which included calories, fat, carbohydrates, protein, and various micronutrients for common foods. I wrote an ETL script to clean and standardize the dataset, including renaming columns to be more user-friendly (for example, “Caloric Value” became “calories_kcal”). This structured data served as a fast and reliable first layer of response.

One of the first challenges I encountered was the structure of the downloaded Kaggle dataset. It was stored inside a non-standard folder structure without a typical file extension, which caused file-not-found errors in my initial ETL script. I resolved this by writing a recursive file search that would locate any CSV file within the downloaded folder. Another challenge came during deployment to Google Cloud Platform. Because I was using a university-managed Google account, I encountered IAM permission errors when the default service account attempted to access Cloud Storage. The error messages were not immediately clear, but after some research I discovered that the project lacked storage permissions required for App Engine deployment. Ultimately, I switched to a personal Google account, created a new project, and redeployed successfully.

Using GitHub throughout the project was essential, but it came with its own learning curve. At one point, I encountered a rejection error when trying to push to the remote repository. The error was caused by a

mismatch between the local and remote branches, which I resolved by using a rebase followed by a force push. This taught me the importance of regularly pulling from the remote and understanding how Git handles history and conflicts.

Through this project, I gained a deeper understanding of full-stack application design. I learned how to build a Flask web server with a clearly defined /chat POST route, how to structure a project using templates and modular components, and how to make the application interactive via a lightweight HTML interface. I also became more comfortable working with external APIs and managing deployment using GCP.

If I had more time, there are several features I would have added. First, I would improve the natural language matching in the local dataset by allowing more flexible keyword matching rather than relying on exact substring searches. I would also incorporate filtering logic based on user intent, such as recognizing phrases like “under 400 calories” or “gluten-free” and applying appropriate filters on the DataFrame. Another enhancement would be to tag each recipe with dietary labels and enable users to filter results by diet. Lastly, I would polish the frontend to make it more interactive, with a proper chat interface and user history.

In summary, this project was an opportunity to bring together data cleaning, API integration, web development, and deployment into a cohesive and practical tool. Despite some initial roadblocks, I finished with a working chatbot that leverages both a local dataset and a live API to provide meaningful recipe suggestions. I now feel more confident using Git, deploying to cloud services, and designing modular Flask applications.