Project Introduction

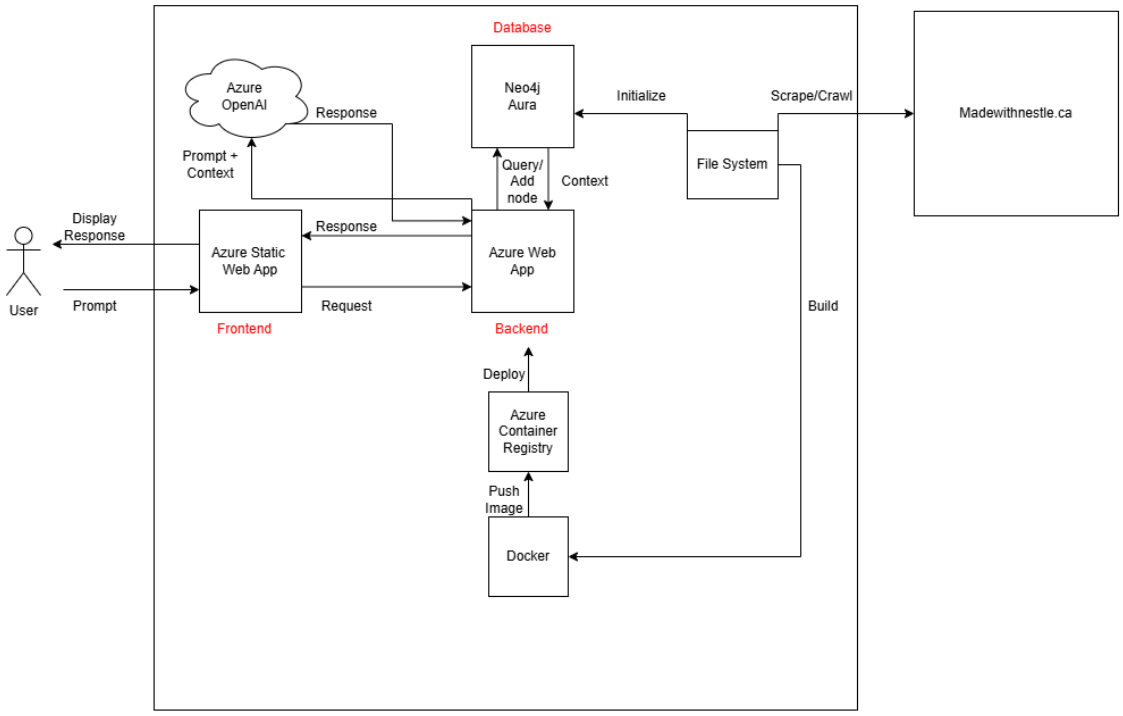
This project is a technical assessment focused on developing a fully functional AI-based chatbot for the website madewithnestle.ca. The objective is to create an intelligent assistant capable of answering user queries by leveraging content scraped from the website, while delivering a seamless, branded, and engaging user experience. This assessment tests a range of capabilities including AI development, web scraping, vector-based search, graph database integration, UI design, cloud deployment, and version control. The final solution is expected to function in real-time, maintain high responsiveness, and enhance the usability of the Nestlé site for visitors.

Project Objectives

The chatbot must be able to:

- Dynamically retrieve and respond to queries using real website data.
- Provide source references where applicable.
- Feature a customizable name and icon, with a responsive pop-out interface.
- Be deployed on a scalable cloud platform (Azure or Google Cloud).
- Utilize a GraphRAG module to understand and respond using structured relationships within the scraped content.

High Level Solution Architecture & Information Flow



Technical Build & Cost Compute Considerations

1. Web Scraping Feasibility

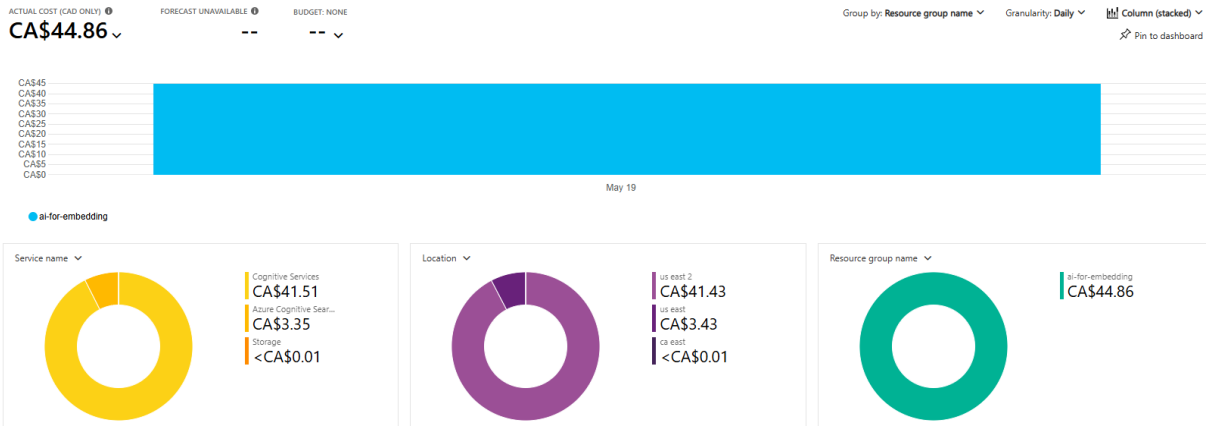
Scraping 82 pages from the madewithnestle.ca website took approximately 15 minutes, largely due to delays in bypassing anti-bot mechanisms. If granted direct access to the raw site data (e.g., via API or internal feed), this process could be reduced to mere seconds. Under such conditions, scraping all 1,820 pages would become significantly more feasible.

However, transforming the scraped content into usable graph documents still introduces a compute overhead. In my experience:

I. Document transformation (after loading and chunking): ~120 minutes

II. The remaining indexing pipeline: ~7 minutes

III. Total Cost: ~CA $44.86 (combining usage of GPT-4o and the text-embedding-3-large model)



Projected Full Scale Cost:
To fully process all 1,820 pages, the projected compute time is **~2,818 minutes (~2 days)**, at a cost of **~CA $995.67**. While not exorbitant, this is still substantial for dynamic real-time updates and there are various opportunities to further optimize this on further iterations.

Suggested Optimization

I. Use a local inference engine such as Ollama, running open-source LLMs.

II. Deploy the system on a virtual machine, reducing both inference time and cost.

III. Schedule updates every 48 hours for near-real-time freshness without the high dynamic cost.

2. Backend Optimization

Given that the backend handles prompt construction, retrieval from both graph/vector stores, LLM communication, and response parsing, it must be efficient and scalable.

To address this:

I. I implemented asynchronous programming and multi-threaded operations.

II. This design enables concurrent request handling, improves response times, and ensures resource efficiency.

III. No additional cost is incurred from this optimization.

3. Frontend Integration

To maintain brand consistency and improve development efficiency:

I. I cloned the original Nestlé website and layered the chatbot widget on top.
II. I studied frontend patterns consistent across Nestlé domains, including:
    - The widget's slide-in animation
    - Matching colour theme
    - Matching text font
    - The transparent overlay that appears when the chatbot is opened

This approach ensures seamless UX integration without rebuilding the UI from scratch.

4. Response to Untrained Queries

When the chatbot encounters a query, it has not been trained on or lacks relevant data for, it will gracefully inform the user that it does not currently have an answer. If the question is later answered by a user or admin and that answer is stored in the system, the chatbot will be able to recall and respond accurately to similar future queries.

Suggested Optimization

1. **User Preference Caching**:
   Store widget preferences (name, icon, etc.) locally or via cookies to provide a personalized experience on return visits.
2. **Voice Recognition**:
   Add voice input support to enhance accessibility and allow users to interact hands-free.
3. **Input Moderation & Security**:
   User input should never be trusted blindly. If input is to be processed directly (e.g., for database queries), it should be:
    - Sanitized thoroughly
    - Passed through a moderation layer, such as OpenAI's free Moderation API

Second Iteration Planning: Timeout and Reliability Enhancements

In the second iteration of development, I will place greater emphasis on timeout handling and overall system responsiveness. Timeout settings will be thoughtfully introduced at every critical layer of the architecture to ensure a smooth and efficient user experience.

The Azure OpenAI API will be configured with timeout thresholds to prevent prolonged response delays. Neo4j graph queries will also include timeout settings to avoid operations from hanging or blocking the system. On the frontend, Axios requests will implement client-

side timeouts, so users receive timely feedback if the backend takes too long to respond. Additionally, the Uvicorn server configuration will be updated to manage keep-alive connections more efficiently, freeing up stalled sessions.

These improvements will help the system remain fast, resilient, and user-friendly even under heavy usage or during unexpected service disruptions.