<u>Technical Considerations & Cost/Compute Analysis</u>

1. Web Scraping Feasibility

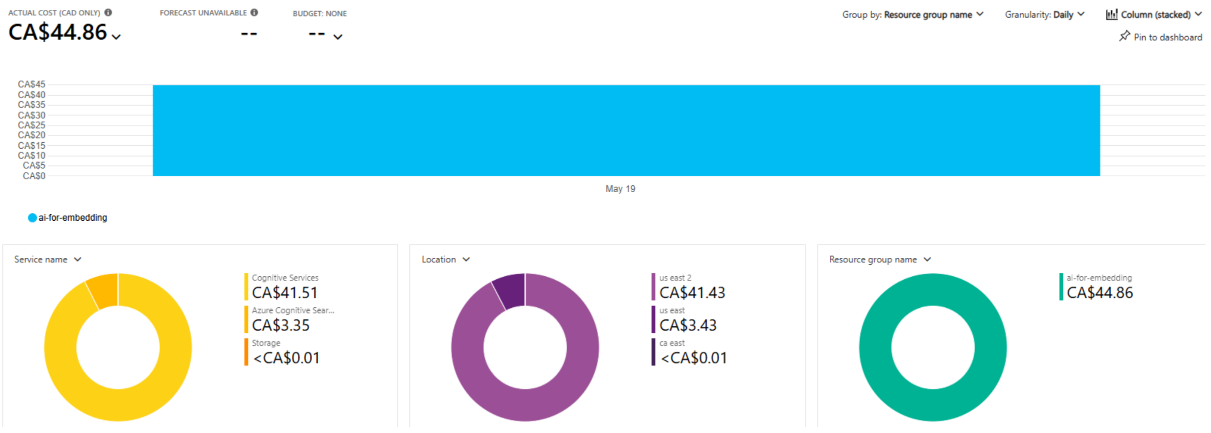
Scraping 82 pages from the madewithnestle.ca website took approximately 15 minutes, largely due to delays in bypassing anti-bot mechanisms. If granted direct access to the raw site data (e.g., via API or internal feed), this process could be reduced to mere seconds. Under such conditions, scraping all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


ACTUAL COST (CAD ONLY): CA$44.86
FORECAST UNAVAILABLE: --
BUDGET: NONE: --
Group by: Resource group name
Granularity: Daily
Column (stacked)

Service name: Cognitive Services CA$41.51; Azure Cognitive Sear... CA$3.35; Storage <CA$0.01
Location: us east 2 CA$41.43; us east CA$3.43; ca east <CA$0.01
Resource group name: ai-for-embedding CA$44.86

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.

Suggested Optimization

Instead of real-time updates:

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.

Additional Suggestions

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