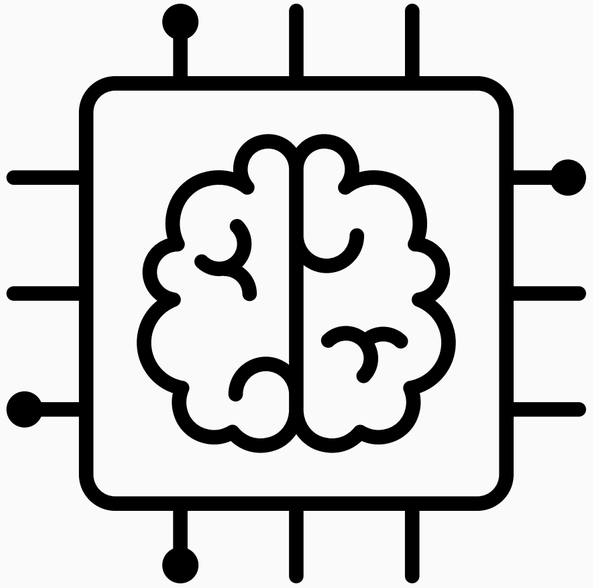
PartSelect LLM

Geneva Ng



How does it work?

User queries are sent to a **Flask** app, then used to retrieve the information that pertains to them from a database of product information.

React Frontend

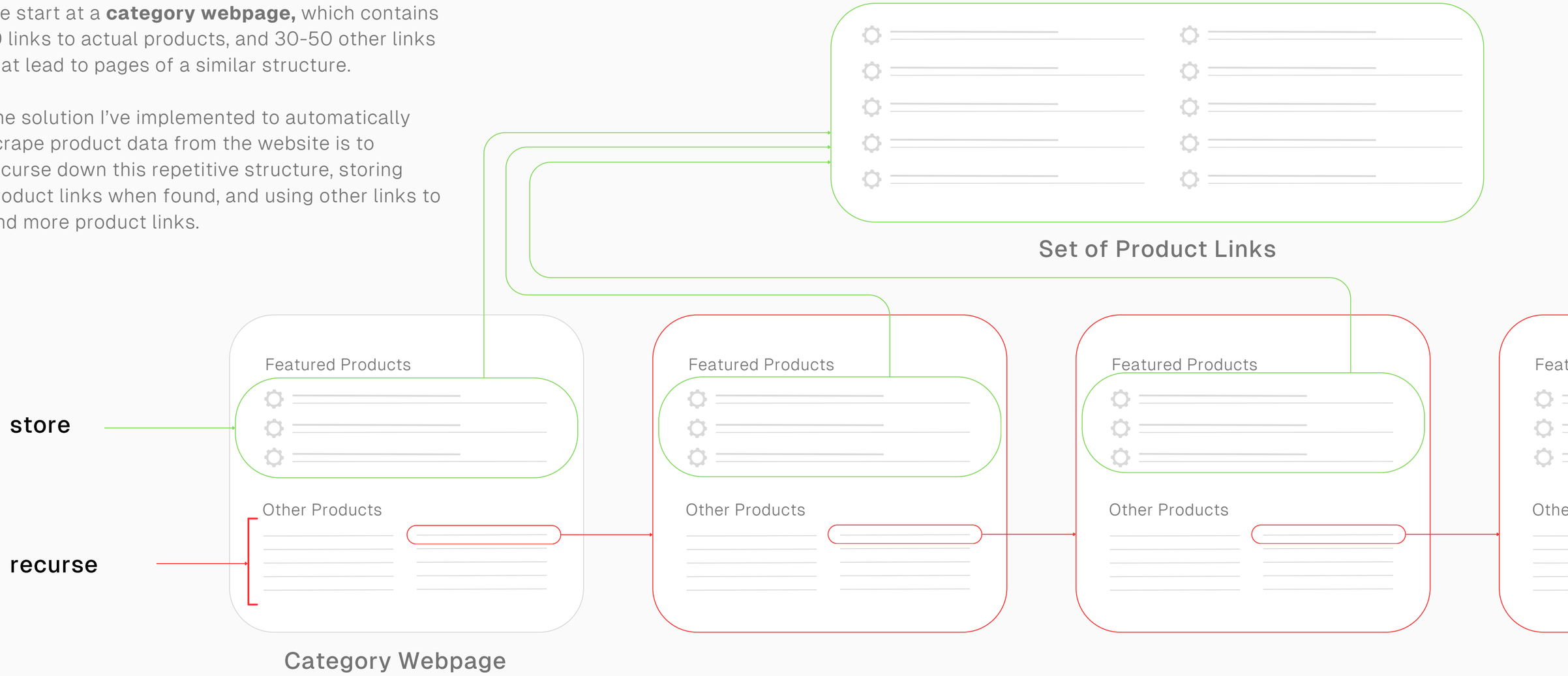
Flask Backend

Product Information

How’d we get this product information?

We start at a **category webpage,** which contains 10 links to actual products, and 30-50 other links that lead to pages of a similar structure.

The solution I’ve implemented to automatically scrape product data from the website is to recurse down this repetitive structure, storing product links when found, and using other links to find more product links.



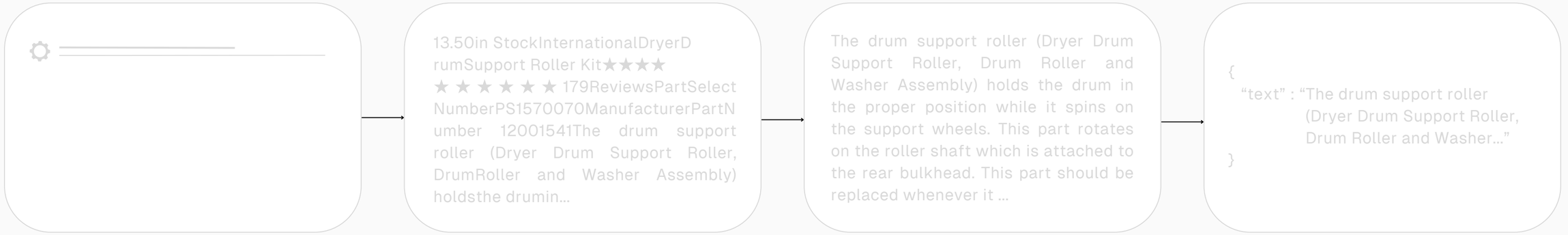
How’d we get this product information?

On every product link, we can see all the information anyone would want to know about a product, from installation guides to lists of compatible parts.

This information is scraped from the page using BeautifulSoup, then processed using gpt-3.5-turbo to be converted to natural langauge, and finally stored in a json-like format.

This process is repeated for every product link that was found during the recursive webscraping process.

After every found product link has been processed, we’re left with a json-formatted python dictionary filled with the product information of hundreds of products.



Product Webpage

Raw Site Data

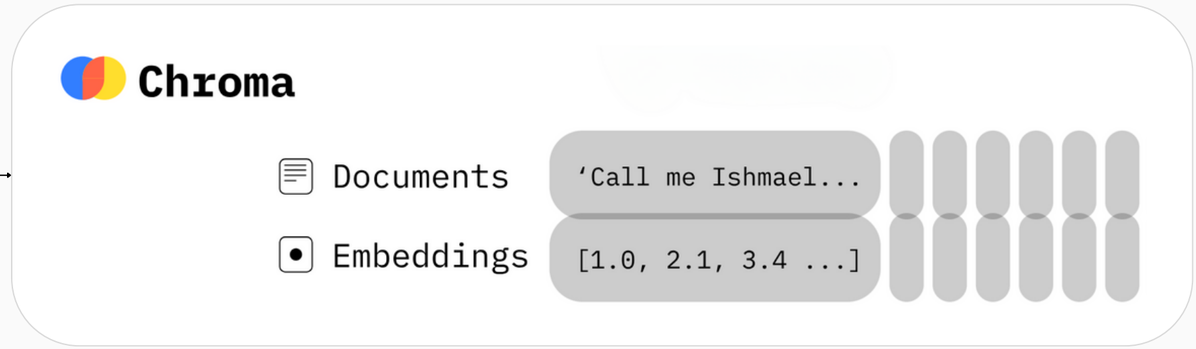
Natural Language

Json format

How does the agent access this information?

And how does it access **only** what it needs?

Database of Product Information



Embeddings are generated for each entry in our product database using the **multi-qa-MiniLM-L6-cos-v1** model from Sentence Transformers, selected due to it’s optimality in semantic search applications.

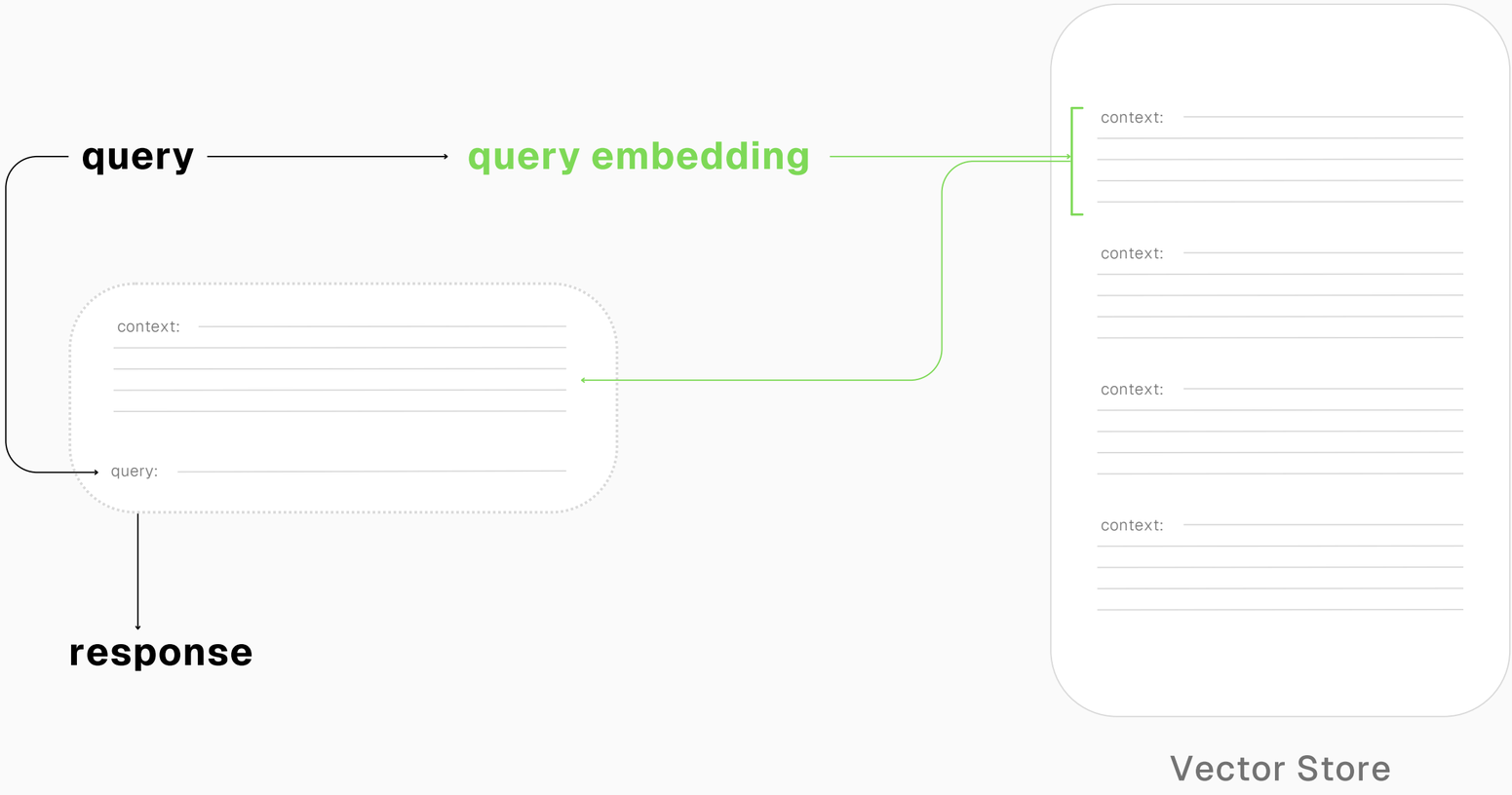
These embedding-document pairs are then stored in **ChromaDB**, an open-source vector database. This vector store will serve as our agent’s knowledge base.

How does the agent access this information?

And how does it access **only** what it needs?

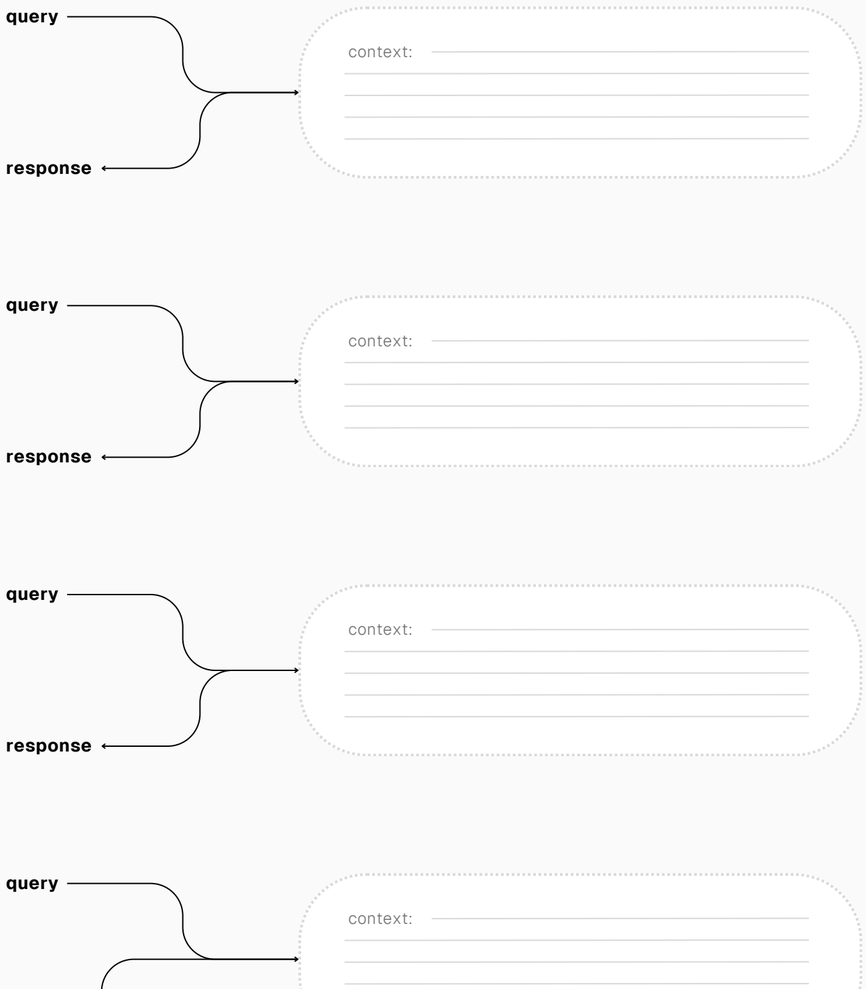
An embedding is generated from the user's query, then that embedding is used to search the vector store for relevant context.

When context is found, it's packaged up along with the original query into a composite prompt that's fed into gpt-3.5 turbo. This method allows the agent to have domain-specific information delivered along with the queries that require it.



Vector Store

Conversational Memory

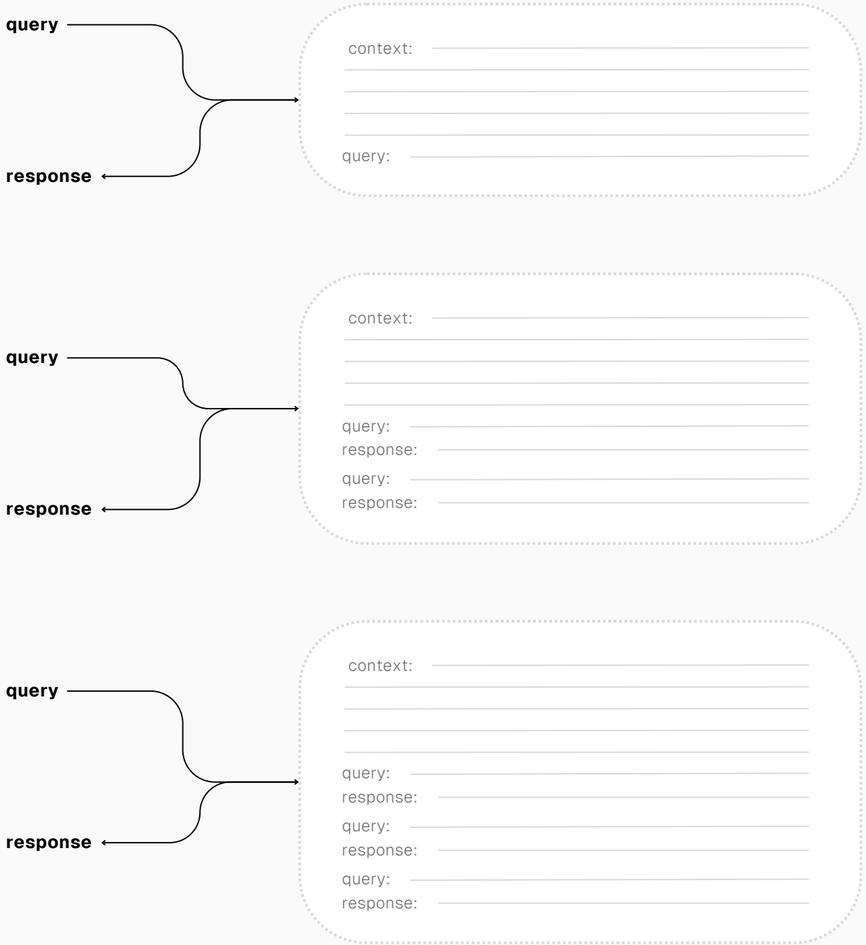


This was my original implementation, where each query would be used to fetch a new piece of context, which would be used to inform the agent's response.

However, this meant that the user wouldn't be able to ask follow-up questions due to the natural lack of context in a typical follow-up question.

Each query would have to explicitly outline what product or issue the user needed assistance with, or the context returned would not be accurate or applicable.

Conversational Memory



I decided to implement a “memory” mechanism in the agent so users could ask follow-up questions, which are unavoidable in circumstances like these where a user might be troubleshooting their dishwasher repair.

The first query is used to fetch context from the vector store. That context is then referred back to with every subsequent query.

Additionally, every query and agent response is stored to “memory”, a temporary dictionary, and this comprehensive conversation transcript is sent through gpt-3.5-turbo on every query to give the machine a complete view of what’s been going on in the conversation.

If the user wants to talk about something new, they can press “new conversation” to wipe this memory and prime the system to fetch a new block of context to form responses with.