SUBMISSION PACKAGE

- ☐ updated github repository with back and front (give them the link)
  - ☐ make sure .env is empty
    - ☐ make .env.template file
  - ☐ can this be pulled off github and run?
- ☐ zipped folder of source code
  - ☐ front and backend
  - ☐ empty .env file
- ☐ loom video of usage
  - ☐ practice fully
  - ☐ then do the real thing

assume they're just going to watch the loom video, then run the code to test it.

INCLUDE IN LOOM VIDEO ***VERBALLY***

1. **WHY DATABASE CREATION TAKES SO LONG:** The lengthy runtime of this process is primarily due to its reliance on the OpenAI API for converting the raw text data from each product page into natural language. This conversion is essential for the embedding function to encode the data accurately, which enhances the accuracy of queries on the vector store. This solution was implemented to prioritize response quality over response range, but if I had more time, this process is one of the first I'd optimize.
2. (...)

SCRIPT

- This is my submission for the Instalily Case study
- I'll walk you through my presentation, then we'll jump into the code and a quick demo of the chat being used

- User queries are sent to a Flask app which runs the backend.
- I chose Flask because it's what I'm currently the most familiar with, so I went with this framework for the sake of development speed
- A query is used to fetch context, which is product information in this case, and a response is made based on a combination of the context *and* the query.

- On PartSelect, there's two types of pages we're primarily dealing with here.
- We have Category pages, which give an overview of dishwasher stuff, fridge stuff, all that
- then product pages. These have all the information for a given product.
- On the category pages, the structure is always the same:
  - 10 product links,
  - then a bunch of other links that lead to other pages that look just like this, where it's 10 products, other links, 10 products, other links.
- What my web-scraping script does is it takes *all* the links on a page, stores the product ones, then *enters* into the others. There, it does the same thing, storing the product links, then entering the rest to find more product links.
- This is done with recursion, so you can edit how many iterations of this your system can handle,
- A depth of 0 gives you a database of 10 products from that first page, and a depth of 1 bumps it up to 455.
- The base case is if we've already seen a link before, we don't go down it.

- So theoretically, the process would cease when all products on the site have been seen.

[go to the code for create_database]

- so in the code right here, you select your baseURL, run it, and it'll add product data from that product category to the ChromaDB.
- You can run this as many times as you need for different categories and just keep adding to your knowledge base.

- Once we've gathered all our product links into a set,
- we iterate through that and scrape all the information off.
- We start with the product webpage, then use beautiful soup to get all the text out,
- then use gpt-3.5-turbo *here too* to parse it into natural language.
- The reason why I found this had to be done was that text ripped straight off the site was too far gone in terms of character formatting that any standard character-based parsing would not have been successful, so I had to use another language model to come in and fix that up.
- the reason why this information really does have to be as close to natural language as possible is because the embeddings won't be as accurate if an entire sentence is compressed into a single word
- I decided to prioritize the backbone of the agent -- which is the efficacy of matching context to a query via embeddings -- over the speed of loading up a database.
- The latter can be fixed with more time, but the largest database in the world doesn't mean much if it's not that searchable,
- So those factors are what motivated this design decision.

- We take that database of natural language product data and generate embeddings for each product,
- then store all of that in ChromaDB
- which is an open-source vector database

- and then we reference this database for each of our queries.
- So the agent takes a query,
- generates an embedding for it,
- searches the vector store with that query embedding to find the most relevant chunk of context
- returns that context and formats it into a context-query bundle
- which is then sent through gpt-3.5 to get a response out.
- which is what we're doing here

- This was my original implementation
- where every query would be used to fetch new context,
- and that new context would be used to inform the response.
- I found that this didn't allow the user to make follow-up questions, which are the backbone of troubleshooting a dishwasher repair, or something like that,
- because follow-up questions rarely have any context of them.
- and without specific information about the question, like in a follow-up question, the embedding of the query matches *everything* and whatever's returned by that query isn't particularly relevant to the subject matter.

- what I decided to implement is a temporary “memory” in the agent,
- which is just a python dictionary
- so on the first query, we fetch context,
- log that to memory,
- log the query to memory, and the response to memory,
- then pass the entire block of memory through gpt-3.5 with every query
- it’s like a chat transcript that keeps growing as the conversation goes on that’s being fed to the gpt model every time the user asks a new question
- The agent has a complete view of everything that was said in the conversation so far.
- this works so well for follow-up questions, I personally love this implementation, it feels so much more natural,
- and to overcome the issue of refreshing the context when the topic needs to change,
- I added a button for the user to do that manually.

- At first, I tried to experiment with a gpt-based topic detection mechanism, but I wasn’t able to get it to pick up on all the nuances of a topic change as well as I would have liked, so I scrapped that idea and settled on this one for the sake of the user’s experience.
- The tradeoff with this decision is
    - you can’t automatically sense a topic change, and you aren’t pulling new context as often
    - but you *do* get follow-up question functionality and naturally-flowing conversation.

- So that’s my implementation of the PartSelect chat bot!
- I had a lot of fun making this, I definitely learned a lot and it
- really solidified my interest in getting into projects like these
- I love figuring out how to play with AI models to augment their capabilities and getting familiar with the tools of the trade.
- Thank you for the opportunity to show this to you guys!

Fridge Homepage
Dishwasher Homepage

1. Fridge 1
2. Fridge 2
3. Dishwasher 1
4. Dishwasher 2

CODE TESTING:

How can I install part number PS10065979?

Is this part compatible with my W10712395 model?

How do I install it?

Will it work with my KitchenAid?

The ice maker on my Whirlpool fridge is not working. How can I fix it?

What's part WP8565925?

Rollers broke off top rack assembly of my dishwasher. What part should I buy to fix it?

How many come it a set of those?

How long will the repair take?

What should I have for lunch today?