Project Report: ShopAssist 2.0

# Objectives

The main objective of this project was to enhance ShopAssist AI into ShopAssist 2.0. The focus was on using the OpenAI Function Calling API to make the chatbot smarter and more reliable. The idea was to simplify the architecture, improve the conversation flow, and make recommendations more structured and useful for users.

# Design

The design of ShopAssist 2.0 makes use of OpenAI’s function calling feature. A function schema was created for laptop recommendations. This allows the assistant to automatically call the function when required, rather than relying on manual parsing. The system is divided into two main files:

* app.py: Flask app that manages routes, user inputs, and conversation history.
* functions.py: Handles all AI logic, laptop dataset loading, summarization, and function calling integration.
* Index.html: User interaction

The design keeps things modular and easy to maintain.

# Implementation

Implementation steps included:

* Building Flask routes for UI (/, /invite, /end\_conv).
* Connecting with the OpenAI API using function calling.
* Creating FUNCTIONS schema to define the laptop recommendation function.
* Writing get\_laptop\_recommendations to filter, score, and return laptops from the dataset.
* Adding summarization logic so that outputs are short and clear.
* Relaxing strict criteria when no laptops matched (for example, changing 'high' to 'medium').
* I also added a simple 'Please wait...' message in the flow so that the user experience feels smoother.

# Challenges

While working on this project, I faced several challenges:

* Handling function arguments safely without breaking JSON decoding.
* Keeping the conversation history short enough to avoid token limits.
* Ensuring that fallback recommendations still feel natural and relevant.
* Cleaning and preparing the laptop dataset where features were not always consistent.
* Balancing detailed answers with a display limit to keep responses user-friendly.

# Lessons Learned

This project taught me that function calling makes chatbot architecture much simpler. Instead of writing complicated intent parsing code, I could rely on the model to decide when to call a function. Summarizing recommendations also turned out to be very useful—it saves tokens and gives users quick insights. I also learned the importance of fallback logic, error handling, and keeping code modular. Finally, adding small touches like a 'Please wait...' message improves user experience.

# Code Documentation

The code is organized into three files:  
  
1. app.py

Manages Flask routes and conversation state.  
Handles user input and resets chat sessions.  
Calls the chat\_with\_functions method to process queries.  
  
2. functions.py

* Contains helper functions like truncate\_conversation, parse\_budget, load\_laptops, and summarize\_recommendations.
* Implements get\_laptop\_recommendations which filters and scores laptops.
* Defines the FUNCTIONS schema for function calling.
* Includes chat\_with\_functions which manages the API calls, function execution, and final response.

3. Index.html

* Frontend template rendered by Flask.
* Provides a simple user interface with an input box for user queries and a display area for the chatbot’s conversation.
* Shows both user messages and assistant responses, including summarized laptop recommendations.
* Helps create an interactive experience by looping through conversation\_history passed from Flask.

# Conclusion

ShopAssist 2.0 shows how the Function Calling API can be applied in practice. It makes conversations smoother, provides structured recommendations, and simplifies the backend code. The chatbot is now easier to extend for future use cases, and it feels more natural to interact with. The project also gave me good hands-on experience in balancing technical design with user experience improvements.