# Project Report: ShopAssist 2.0

**Enhanced AI Shopping Assistant with Function Calling**
By : Nikhil Kalra

### 1. Project Objective

The primary goal of this project was to upgrade the existing "ShopAssist" chatbot from a simple heuristic-based model to a Production-Grade AI Agent (ShopAssist 2.0). The specific objectives were:

1. Replace manual parsing with OpenAI's native Function Calling API for accurate data retrieval.
2. Enhance System Reliability by implementing auto-retry mechanisms for network stability.
3. Ensure Safety & Security through a moderation layer.
4. Optimize Memory Management to handle long conversations without hitting token limits.

### 2. System Architecture & Design

The ShopAssist 2.0 architecture follows a ReAct (Reason + Act) pattern. Unlike a standard chatbot that simply replies, this agent operates in a loop:

1. Input Layer: User text is checked against the OpenAI Moderation API.
2. Memory Layer: Conversation history is retrieved and "trimmed" (Sliding Window) to keep only the last 5 turns.
3. Reasoning Layer (LLM): The model analyzes the user's request.
    - If specifications are missing, Ask clarifying questions.
    - If specifications are clear, Generate a Function Call.
4. Execution Layer: The Python function `get_laptops_from_db` executes with the extracted parameters.
5. Response Layer: The LLM summarizes the tool output into natural language for the user.

### 3. Key Technical Implementations

**A.** OpenAI Function Calling (vs. Manual Parsing)

Instead of using Regex or string splitting to find budgets or brands, I defined a strict JSON Schema.

- Why: This ensures the model treats "50k" as `50000` (integer) automatically.
- Implementation: Defined a `tools` list with strict typing (`type: integer`, `type: string`) passed to the `call_gpt` function.

**B.** Robustness with `Tenacity`

Production APIs often experience transient timeouts. I implemented the `tenacity` library to handle this.

- Implementation: The `@retry(stop=stop_after_attempt(3))` decorator wraps the API call.
- Benefit: The system automatically retries failed requests up to 3 times before raising an error, preventing crashes during network blips.

**C.** Memory Management (Token Optimization)

To prevent the "Context Window Exceeded" error, I implemented a `trim_history()` function.

- Logic: It maintains a "Sliding Window" of the System Prompt + the last 5 messages.

- Benefit: Allows for infinite conversation duration without crashing the memory buffer.

**D.** Safety Guardrails

I integrated the client.moderations.create endpoint at two stages:

1. Input: Checks if the user asks something harmful.
2. Output: Checks if the AI generates something inappropriate.

```
ShopAssist is live, Type "exit" to close
--------------------------------------------------
You:  i will kill you
ShopAssist: I cannot assist with that request.
--------------------------------------------------
You:  why not? will kill you
ShopAssist: I cannot assist with that request.
--------------------------------------------------
You:  exit
ShopAssist: Goodbye!
```

**4. User Experience & Prompt Engineering**

**Intent Confirmation**

To prevent the bot from searching blindly (e.g., searching for "laptops" when the user just says "Hi"), I implemented a **"Gatekeeper Rule"** in the System Prompt:
*"INTENT CHECK: DO NOT call the search tool immediately. First, check if the user has provided at least 2 specific preferences."*
This creates a natural, consultative dialogue similar to a real human sales engineer.

**Handling "No Results"**

A challenge encountered was "Hallucination" where the AI invented products when the database returned empty.

- **Solution:** I added a **"Negative Constraint"** to the System Prompt: *"IF the tool returns 'No laptop found', DO NOT invent or suggest options."*

```
What's your priority right now?
--------------------------------------------------
You:  gaming laptop low budget and travel friendly
   (🤖 AI searching database...)
ShopAssist: Here are the top 3 options I found that balance gaming capability with travel-friendliness and stay within your budget range:
```

**5. Challenges & Solutions**

| Challenge | Root Cause | Solution Implemented |
|---|---|---|
| **API Timeouts** | Unstable internet connection caused the bot to crash. | Added tenacity library with exponential backoff retries. |
| **Hallucinations** | When the DB returned 0 results, the "Helpful" AI invented fake recommendations (e.g., Chromebooks). | Updated System Prompt with **"Strict Data Adherence"** rules. |

| Challenge | Root Cause | Solution Implemented |
| --- | --- | --- |
| **Context Overflow** | Long conversations consumed all available tokens (some limit). | Implemented trim_history to keep only the last 5 exchanges. |
| **Parsing Errors** | Model confused "under 25k" with "greater than 25k" in logic. | Relied on Function Calling schema to force integer extraction rather than text parsing. |

**6. Lessons Learned & Conclusion**

ShopAssist 2.0 is now a fully functional, resilient, and safe AI agent. By leveraging Function Calling, it eliminates the fragility of the previous version. The addition of memory management and retry logic ensures it is ready for real-world deployment scenarios.