Project Report: ShopAssist 2.0

1. Objectives

The primary objective of this project was to refactor the existing ShopAssist AI chatbot by integrating the **Function Calling** feature of Google's Gemini LLM. The goal was to transform the proof-of-concept into a robust, reliable, and scalable application.

The key secondary objectives were:

- To simplify the system architecture by removing brittle, prompt-based logic for data extraction.
- To increase the reliability of the communication between the AI and the application's business logic.
- To create a more streamlined and efficient conversational flow, where the AI could act on user intent directly.
- To gain practical, hands-on experience with modern, tool-augmented AI agent development.

---

2. Design

The project involved a significant redesign of the chatbot's core architecture.

- **Original Design (ShopAssist 1.0):** The initial design was a multi-stage process that relied heavily on complex prompts to force the AI to generate a dictionary-like string. The Python application then had to parse and validate this string before it could execute the laptop search. This was a "disconnected" architecture where the AI had no true awareness of the application's functions.
- **New Design (ShopAssist 2.0):** The new, integrated design redefines the AI's role from a simple text generator to an intelligent orchestrator. The core of this design is the `find_laptops()` Python function, which is exposed to the AI as a "tool." The AI's job is now to have a natural conversation, understand when the user wants to find a laptop, and then call the appropriate tool with the correct arguments.

---

3. Implementation

The implementation was focused on three main parts:

1. **Tool Definition (`find_laptops()`):** The core business logic for filtering and scoring laptops was encapsulated in a clean Python function. This function serves as the application's "search engine."
2. **Tool Declaration:** A detailed schema was created using `protos.FunctionDeclaration`. This schema acts as a "manual" for the AI, describing the tool's name (`find_laptops`), its purpose, and all its required and optional parameters (e.g., `budget`, `portability`).
3. **Interactive Engine (`run_chatbot()`):** A main application loop was built to manage the conversation. This loop sends user input to the model and then uses a `try/except` block to intelligently handle the two possible responses from the AI:
    - A standard text response.
    - A structured `function_call` object, which triggers the execution of our Python tool.

---

4. Challenges Encountered

Several technical challenges were encountered and successfully resolved during development:

- **Model Availability (`NotFound: 404`):** Initial attempts to call the API failed because the specified model names (`gemini-pro`, `gemini-1.5-pro-latest`) were incomplete. **Solution:** We used the `genai.list_models()` function to programmatically find the exact, full model name (`models/gemini-pro-latest`) available to the API key.
- **API Syntax Errors (`TypeError`):**
    - The first `TypeError` was caused by using string literals (e.g., `"object"`) in the tool declaration instead of the required `protos.Type` enums.

    - A second `TypeError` occurred because the `chat.send_message()` function was called with an incorrect keyword argument (`part=...`). **Solution:** Both issues were resolved by correcting the code to match the library's specific syntax requirements.
- **API Rate Limiting (`ResourceExhausted`):** During rapid testing, we exceeded the free tier's requests-per-minute limit. **Solution:** The issue was resolved by waiting one minute for the limit to reset and restarting the notebook kernel to ensure a clean session.

---

5. Lessons Learned

This project provided several key insights into modern AI development:

- **The Power of Function Calling:** Function calling is a vastly more robust and reliable method for integrating LLMs with external code compared to prompt engineering and string parsing.
- **Importance of Precise Schemas:** The AI's ability to use a tool effectively is directly dependent on a clear, accurate, and well-documented tool declaration.
- **Real-World Error Handling:** Development with APIs involves more than just writing correct logic; it requires handling specific service errors like `NotFound` and `ResourceExhausted`. Using the API's own diagnostic tools (like `list_models()`) is crucial for debugging.