**Technical Design Document: Fitbit Conversational AI**

**Author:** Ilan Aizelman

**Date:** August 2, 2025

**1. High-Level Architecture**

The assistant is designed as a modular, API-driven service to ensure scalability, maintainability, and a clear separation of concerns. The architecture consists of three primary components:

1. **Client Layer**: A user-facing interface (represented by the Jupyter Notebook and debug\_client.py) responsible for sending user queries to the API service and displaying the assistant's responses.
2. **Service Layer (FastAPI)**: A high-performance API service built with FastAPI. This layer handles HTTP requests, manages user sessions, and serves as the entry point to the conversational agent.
3. **Agent Layer (LangChain)**: The core of the system. This layer encapsulates the AI logic, including the Large Language Model (LLM), the tools for data retrieval, and the reasoning engine that orchestrates the conversation.

**2. LLM Orchestration Framework**

I have selected **LangChain** as the primary orchestration framework, specifically utilizing a **ReAct (Reasoning and Acting) agent**. This approach was chosen over a direct LLM API call for several key reasons that are critical for a production-grade system:

* **Tool Usage**: The ReAct framework allows the agent to dynamically decide which "tool" (i.e., a data-fetching function) to use based on the user's query. This is fundamental to grounding the assistant's responses in real, user-specific data and makes the system highly extensible.
* **State Management**: LangChain's **ConversationBufferMemory** is used to maintain the context of a conversation based on session id (In the current implementation). This allows the user to ask follow-up questions without having to repeat information, creating a more natural and fluid dialogue.
* **Observability**: The agent's chain-of-thought process (the "Think" step) is explicitly logged. This is invaluable for debugging, understanding why the agent made a certain decision, and fine-tuning its behavior.
* **Modularity**: The agent, tools, and prompts are defined as separate, composable components. This allows for independent development and testing. For instance, a new data source can be added as a new tool without altering the core agent logic.

**3. Data Storage**

The data storage strategy is designed to evolve from the POC to a scalable production environment.

**POC Data Storage**

For the **Proof of Concept**, I decided that it will be very practical that the data will be stored in a simple, in-memory Python dictionary (**mock\_user\_database in tools.py**). This approach is ideal for rapid development and testing as it has no external dependencies and allows for easy modification of user data to test different scenarios.

**Production Data Storage**

For a production environment, a more robust and scalable solution is required:

1. **User Health Data**: A secure, scalable, and HIPAA-compliant database, such as **PostgreSQL** (for structured relational data) or a NoSQL alternative like **Amazon DynamoDB** (for flexible schema and high scalability), will be used as the primary data store for all user health metrics. The agent's tools will be refactored to query this database instead of the mock dictionary.
2. **Conversation History**: To ensure low-latency responses, conversation memory will be managed by a fast, in-memory cache like **Redis**. This offloads the primary database and provides the quick access needed for real-time conversation. Each user session will have its history stored under a unique key, which will be automatically expired after a period of inactivity.

**4. Prompt Strategy and Agent Behavior**

The agent's behavior is almost entirely defined by the system prompt in agent.py. The strategy is multi-faceted:

* **Persona Definition**: The prompt begins by establishing the assistant's persona: "You are a friendly and encouraging Fitbit AI assistant. Your name is Aura." This ensures a consistent and positive tone in all interactions.
* **Tool-Use Instructions**: The prompt provides explicit, structured instructions on *how* the agent should use tools. It defines the Thought -> Action -> Action Input -> Observation cycle, which is the core of the ReAct framework. This structured guidance significantly improves the reliability of the agent's reasoning process.
* **Grounding and Safety**: A critical instruction is that the agent **must** use its tools to access user data. This prevents the LLM from "hallucinating" or making up health data. By grounding the agent in specific data sources, we ensure the insights it provides are accurate and personalized.
* **Dynamic Context**: The prompt is dynamically populated with the current user\_id and conversation history. This ensures that every reasoning cycle is tailored to the specific user and the immediate context of the dialogue.

**5. Evaluation and Monitoring Strategy for Production**

Moving from a POC to a production service requires a robust strategy for ensuring reliability, performance, and continuous improvement.

* **Logging & Monitoring**:
  + **Structured Logging**: Expand the existing logging to output structured JSON logs for each API request. These logs will include the session\_id, user\_id, the full agent thought process, the final response, and the response latency.
  + **Performance Dashboard**: Use a monitoring service like **Datadog** or **Grafana** to visualize key metrics: API latency (p95, p99), error rates, and average number of tool calls per query. This will allow us to quickly identify performance bottlenecks or an increase in agent failures.
* **Evaluation & Improvement Loop**:
  + **Offline Evaluation**: Before deploying any change to the prompt or the underlying model, it will be tested against a "golden dataset" of representative user queries and their ideal responses. This regression testing ensures that improvements in one area do not degrade performance in others.
  + **A/B Testing**: For significant changes (e.g., a new prompt strategy or a major model upgrade), I will deploy the change to a small subset of users. By comparing engagement metrics (e.g., conversation length, use of suggested features) between the control and test groups, we can make data-driven decisions about which version to roll out.
  + **User Feedback**: Integrate a simple feedback mechanism (e.g., a thumbs-up/thumbs-down icon) into the UI. This qualitative data is invaluable for identifying responses that are technically correct but unhelpful or tonally inappropriate.
  + **Failure Analysis**: Create an automated alert for when the agent's handle\_parsing\_errors mechanism is triggered. A weekly review of these failures will help us identify edge cases and continuously improve the robustness of the prompt and tool-handling logic.