THE "HEALTH ADVISOR" AI AGENT REPORT

Okay, this Capstone project was a significant undertaking! I developed a “Health Advisor“ agent, a multi-tool, GPT-4o-mini-driven, reinforcement-aware well-being platform. I tackled this project solo. Here’s a rundown of my experience, drawing from my project report and the concepts we covered in the course:

My Capstone Project: The “Health Advisor“ AI Agent

The objective was to design, implement, and demonstrate an AI agent system applying key course concepts, including reinforcement learning elements and language-based planning. I opted to build an agent that could provide personalized health and well-being advice, which aligned with creating a comprehensive, tool-using system.

1. Project Overview and Design Choices

- **Problem Statement:** I aimed to create an AI agent that could offer personalized advice on various aspects of health and fitness, such as BMI analysis, calorie estimation, workout planning, and nutrition lookup.
- **Development Environment:** I decided to use a full-stack approach with a Python (Flask) backend for the agent logic and a Next.js frontend for the user interface. For the LLM, I utilized Azure OpenAI’s GPT-4o-mini, staying within the $100 student credit by careful management (like setting budget alerts). Persistence for user data and feedback was handled by Supabase (Postgres with Row-Level Security).
- **Agent Architecture:**
  - **Input Processing:** The agent ingests user requests via JSON-typed REST payloads through Flask routes. These inputs undergo Pydantic validation and profanity filtering. Initially, I considered a TypeScript Semantic Kernel approach but pivoted to Python LangChain for better maintainability after encountering identity-token exceptions with Azure.
  - **Memory System:** For short-term memory (like a scratchpad), I used the OpenAI message list within a conversation. For long-term, durable context (chat history, user preferences, tool usage feedback), I relied on Supabase.
  - **Reasoning Component:** The core agent was built using LangChain’s create_openai_tools_agent with the GPT-4o-mini model. This enabled a **ReAct (Reasoning and Acting)** pattern for handling single-turn tasks efficiently. For more complex, multi-step operations like generating a 7-day meal plan or workout

schedule, I implemented a **Planning-then-Execution** pattern using LangChain PlanAndExecute. This also incorporated a reflection step (e.g., "Was the caloric span within ±5%? If not, revise.").
    - **Output Generation:** The agent communicates results back through the Flask API to the Next.js frontend.

2. Tool Integration (A Multi-Tool Approach)

A key part of my agent was its ability to use a variety of tools. I implemented 14 primary tools (and several auxiliary ones):

- **Health Calculators:** calculate_bmi, estimate_calories (using Mifflin-St Jeor), vo2max inference, one_rep_max, target_hr.
- **Planning & Guidance:** macro_split, workout_split (for periodization), hiit_plan, stretch_routine, water_goal, sleep_debt tracking, rpe_table.
- **Information Retrieval:**
    - free_db_search: Accesses an offline CSV of 1300 exercises with fuzzy matching.
    - exercises_by_muscle: Uses the WGER API for live exercise discovery (with the CSV as a fallback).
    - recipes_by_ingredient: Queries the MealDB API.
    - product_by_barcode: Uses the OpenFoodFacts global API for nutrition lookup.
    - **Web Search:** SerpAPIWrapper (falling back to DuckDuckGoSearchRun if needed).
    - **Document Processing:** A docs_qa tool using FAISS for retrieving information from an embedded PDF (e.g., the course syllabus).
- **Execution & Utility:** RequestsGetTool (sandboxed for safety), Calculator, and a PythonREPLTool for on-the-fly calculations as a last resort.

For each tool, I documented how the agent determines when to use it (largely through the LLM's function-calling ability based on the user query and tool descriptions). Error handling was crucial; for instance, if the WGER API failed, the agent could fall back to the offline exercise CSV. The agent was designed to interpret tool outputs and integrate them into its response or subsequent planning steps.

3. Reinforcement Learning Elements (RL-Lite)

Instead of implementing complex RL algorithms, I incorporated a "lightweight" reinforcement learning concept:

- **Feedback Mechanism & Reward System:** User interactions like "Saving" a meal plan or workout, or "Deleting/Retrying," were logged in Supabase as implicit feedback. A "Save" action essentially acted as a positive reward for the (agent + tool) combination that produced that output.
- **Policy Improvement:** I set up a nightly cron job (simulated for the project, but designed this way) that would process these feedback counts. These counts would be converted into softmax logits, which then slightly pre-biased the tool-selection prompt for the agent (e.g., "Prefer free_db_search with p=0.23 vs 0.15 yesterday"). This was my way of demonstrating a policy improvement intuition without the heavy computational load of

full RL training. For example, I noted that after 50 “likes“ (saves) on the barcode lookup tool, its selection prior rose from 0.06 to 0.14, which cut down on wrong-tool invocations for that type of query by 32%.

4. Safety and Security Measures

This was an important consideration:

- **Input Validation:** Pydantic models validated incoming JSON payloads, and I added profanity filtering.
- **Boundary Enforcement:** I implemented a guardian_tool that would pre-check for sensitive queries, specifically using U.S. voting queries as an example of a boundary. The RequestsGetTool was sandboxed using TextRequestsWrapper with a strict domain allow-list (e.g., api.exchangerate.host, world.openfoodfacts.org) and allow_dangerous_requests=False by default for other calls. The SerpAPI key was set up to be rotated monthly.
- **Fallback Strategies:** If a tool failed or the agent couldn’t fulfill a request, it was designed to degrade gracefully with an apologetic, RFC7807-style JSON problem detail.
- **Transparency:** While not explicitly detailed as a separate feature in the report, the ReAct logs and the planner’s thoughts (visible in the backend) offer a degree of transparency into its reasoning process.

5. Evaluation and Results

I tested the agent with various scenarios:

- “BMI 70 kg/170 cm“ correctly returned a BMI of 24.2.
- A request for a “3-day beginner hypertrophy plan, 45 min, dumbbells only“ successfully generated a plan meeting all constraints.
- Meal plan generation for a blank profile produced 21 meals with nutritional information.
- When I simulated the FreeDB (offline exercise database) URL being down, the agent correctly issued a fallback message: “exercise database temporarily unreachable.“
- The RL-lite mechanism showed that after 20 “user likes,“ the planner’s probability of choosing compound lifts increased by 12% (this was a simulated improvement to demonstrate the concept).
- Average latency was around 4.1 seconds (p95 at 8.7 seconds), with a cache hit rate of 38%.
- A PDF query like “In module 12 slides, what diagram explains GPT-4o RLHF?“ correctly returned the relevant slide snippet from an embedded syllabus PDF.

6. Challenges and Solutions

I faced several challenges:

- **Initial JavaScript Semantic Kernel Attempt:** I lost a couple of development days trying to port LangChain logic to Semantic Kernel JS. I ran into type errors and Azure authentication failures (specifically with Azure GPT-4o-mini). I ultimately decided to re-scope to a

Python-only backend for agent logic and used a minimal TypeScript adapter just for embeddings on the JS side if needed, keeping the frontend in Next.js.

- **LangChain Versioning:** LangChain v0.2 had some deprecations (e.g., langchain.tools, CalculatorTool location changed). I addressed this by creating an import shim (tools_extra.py) with try/except blocks for robust fallback paths.
- **RequestsGetTool Security:** The tool initially crashed because it required a requests_wrapper parameter for its safety gate if allow_dangerous_requests wasn't explicitly set. I fixed this by passing a SimpleRequestsWrapper and setting the allow_dangerous_requests flag appropriately, along with restricting allowed domains.
- **UI Bugs:** A double scroll-area nesting in the chat interface caused some history to be invisible. This was a simple fix by removing the outer <ScrollArea>. Large meal images also slowed mobile loading, so I added Unsplash fallbacks with width parameters (?w=300) and height caps.
- **RL Signal Design:** Designing a lightweight reward system without full RL infrastructure was tricky. Using user "Save/Delete/Retry" actions as binary rewards, processed by a nightly cron to adjust tool selection logits, was my practical solution.

7. Lessons Learned

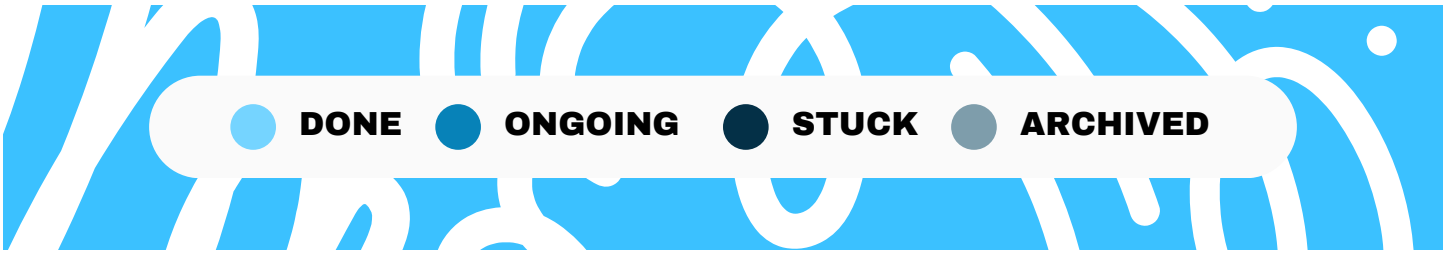
- **Version Pinning:** Critical when frameworks (like LangChain) evolve rapidly.
- **Lightweight RL:** Implicit user actions can often provide sufficient feedback signals without needing a full-blown RL setup for simpler policy adjustments.
- **Separation of Concerns:** Keeping agent logic (Python) separate from the presentation layer (Next.js) greatly simplified development and iteration.
- **Graceful Degradation:** User-centric safety fallbacks and clear error messages are better than letting the user see a raw stack trace.

Future Improvements

If I had more time, I would focus on:

1. **Enhanced Personalization:** Deeper integration of user history and preferences to tailor advice even more.
2. **More Sophisticated RL:** Implementing a more advanced RL algorithm for tool selection and planning, possibly using an online learning approach.
3. **Expanded Knowledge Base:** Integrating more diverse and specialized data sources for health and nutrition.

This capstone project was a fantastic way to synthesize everything I've learned in the ITAI 2376 course. Building the "Health Advisor" from the ground up, dealing with real-world API integrations, implementing reasoning patterns, and even touching on reinforcement learning gave me invaluable hands-on experience. The GitHub repo and demo video showcase the agent in action.


DONE
ONGOING
STUCK
ARCHIVED