MDPI

Article

IronAI: Design and Implementation of an Intelligent Agent for Fitness and Health Assistance

Jhony Penaherrera

Yachay Tech University, School of Mathematical and Computer Science

Abstract

This report details the development cycle of *IronAI*, a conversational agent designed to act as a hybrid personal trainer. The system not only offers generative exercise routines but also integrates mathematical calculation capabilities (BMI) and a motivational support module. Using graph orchestration with *LangGraph* and local models through *Ollama*, the project demonstrates how to overcome the limitations of traditional chatbots through modular architecture, state management, and advanced prompt engineering techniques.

Keywords: Intelligent Agents; LangGraph; Local LLMs; Fitness Assistant; Prompt Engineering

Received:
Revised:
Accepted:
Published:

Citation: Jhony Penaherrera, C. E. IronAI: Intelligent Agent for Fitness and Health Assistance. *Journal Not Specified* **2025**, *1*, 0. https://doi.org/

1. Introduction to the Problem

In the current landscape of wellness and physical health, there exists an invisible but significant barrier: analysis paralysis. An individual wishing to improve their physical condition faces an overwhelming amount of contradictory information on the internet, mathematical formulas to calculate their health status that they often ignore, and perhaps most importantly, a chronic lack of constant motivation.

Traditional fitness applications tend to be rigid; they function as static databases that don't "understand" the user's emotional context nor can they instantly adapt a routine based on natural conversation. On the other hand, generic LLMs (like ChatGPT in its web version) often fail at performing precise mathematical calculations or lose track of the trainer's "personality" throughout a session.

The objective of this project was to build **IronAI**, an agent that solves these problems by acting on three simultaneous fronts: it functions as a strict calculator for health data, a creative designer for gym routines, and an empathetic "coach" for moments of low motivation. The main challenge was not just getting the model to respond, but ensuring it knew *when* to behave like a calculator and *when* to act like a motivational friend.

2. Model and Library Selection

For the construction of this agent, the premise was to maintain efficiency and privacy, opting for 100% local execution.

2.1. Inference Engine: Ollama and Gemma

We decided to use **Ollama** as the execution environment due to its ease of managing quantized models locally, eliminating network latency and API costs.

The selected model was **Gemma 3 (1b version)**. The choice of a 1-billion parameter model was deliberate and strategic. Although larger models exist (7b, 70b), the 1b model

allowed us to iterate quickly during development and demonstrates that with a good agent architecture, you don't need a giant "brain," but rather a well-directed one. Gemma showed a surprising balance between grammatical coherence in Spanish and the ability to follow logical instructions.

2.2. Orchestration: LangChain and LangGraph

While **LangChain** provided us with the necessary primitives (such as prompt templates and output parsers), we quickly realized that a simple sequential chain wouldn't be sufficient. The flow of a real conversation is not linear; the user can jump from requesting a diet to asking about their ideal weight.

This is where **LangGraph** became the centerpiece of the architecture. It allowed us to model the agent not as a straight line, but as a state graph (*StateGraph*). This gave us the flexibility to create cycles, conditional decisions, and maintain persistent memory between the different nodes of the system.

3. Prompt Engineering Strategies

One of the most interesting findings during development was that a single "all-purpose" prompt reduced response quality. Therefore, we implemented a strategy of specialized prompts based on detected intent.

3.1. The Classifier (Router Prompt)

The first challenge was getting the agent to understand intent unambiguously. We designed a strict prompt with *Temperature 0.1*, instructing the model to act solely as a taxonomic classifier. It was prohibited from generating conversational text at this stage, forcing it to return only predefined labels (`routine`, `bmi`, `chat`). This drastically reduced routing errors.

3.2. Few-Shot Prompting for "Personality"

For the motivational chat module (*IronCoach*), we wanted to avoid the robotic and complacent tone typical of AI assistants. We implemented the **Few-Shot Prompting** technique.

By injecting concrete examples into the system prompt—showing how to respond to a complaint with energy and stoicism rather than pity—we managed to "fine-tune" the model's behavior without needing to retrain it. For example, to the input "I'm tired," the model learned not to respond "Do you want to rest?" but rather "Rest is for after victory!"

3.3. Implicit Chain-of-Thought

In routine generation, we used a sequential structure that simulates a chain of thought. Instead of asking "give me a routine and advice," we divided the problem: first, the model generates the technical structure of the exercise, and then, using that output as context, a second prompt generates the supplementation recommendation. This notably improved coherence between the suggested exercise type and the given nutritional advice.

4. LangChain Architecture

The operational foundation of the agent was built on LangChain, using it not as a monolithic orchestrator, but as a library of modular components. The implementation focused on three main types of chains:

4.1. Sequential Chains

For training routine generation, we implemented manual sequential logic within a node. We observed that if we asked the LLM to generate the routine and supplementation in a single step, it would often hallucinate generic supplements.

By dividing the process into two sequential calls—where the output of the `routine_chain` becomes the input to the `supplement_chain`—we achieved nutritionally relevant recommendations (e.g., recommending fast-absorbing proteins after a hypertrophy routine).

4.2. Structured Output Parsers

A critical component was the use of `JsonOutputParser`. For the BMI calculation tool, natural language is useless; we needed precise numerical data. We configured an extraction chain that forces the model to ignore superfluous text and return exclusively a JSON object with the keys `"weight"` and `"height"`. This transformed unstructured data ("I weigh about 80 kilos and I'm 1.80m tall") into structured, computable data.

5. LangGraph Architecture

If LangChain provided the bricks, LangGraph provided the architectural blueprint. The agent's design moved away from linear pipelines (DAGs) to adopt a cyclical state-based architecture.

5.1. State Definition

The heart of the system is the `AgentState` object, implemented as a `TypedDict`. This object acts as the agent's "short-term memory," traveling through nodes and accumulating information. Unlike passing simple text strings, the state transports:

- `input_text`: The original query.
- `intent`: The routing decision.
- `user_data`: Extracted data (weight/height) for persistence during the session.
- `final_output`: The response constructed by specialist nodes.

5.2. The Router Node and Conditional Edges

We implemented a decision node (`router_node`) at the beginning of the graph. This node doesn't generate content for the user, but rather analyzes the state and decides the next step in the flow. Using *Conditional Edges*, the graph branches dynamically:

1. **Analytical Branch:** If the user provides metric data, the flow diverts to the `bmi_agent` node.
2. **Generative Branch:** If exercises are requested, the `routine_agent` is activated.
3. **Conversational Branch:** For any other interaction, the `chat_agent` takes control.

This architecture allows the agent to be "multimodal" in its behavior: it can be a strict calculator in one turn and a motivational poet in the next, without contaminating one context with the other.

6. Memory and Tools Integration

One of the key requirements was to endow the agent with capabilities beyond text generation.

6.1. Tool Use

LLMs are notoriously bad at basic arithmetic. To solve this, we integrated a deterministic tool written in pure Python: `calculate_bmi_tool`.

The innovation here is not the function itself (which is a simple formula), but how the agent decides to use it. The `bmi_agent` node acts as an intelligent intermediary: first, it uses

the LLM to clean and extract data from the user's natural language, and then passes that clean data to the Python function. The calculation result is reinjected into the final prompt so the LLM can explain the result with empathy, combining machine precision with natural language warmth.

6.2. Data Persistence

Although LangGraph allows complex memories with *checkpoints*, for this prototype we opted for runtime state memory. The state dictionary retains user data during graph execution, allowing that if the user states their weight at the beginning, the agent could theoretically remember it in subsequent nodes without having to ask again, simulating coherent conversational memory.

7. Parameter Experiments

A fundamental part of fine-tuning the agent was experimenting with inference parameters, specifically temperature (`temperature`). We conducted comparative tests in two critical scenarios: routing and creative generation.

7.1. Configuration A: High Temperature (0.8 - 0.9)

When configuring the *Router* node with a temperature of 0.8, we observed a phenomenon of "over-creativity." For the input "Hi, I want to train," the model sometimes responded with an enthusiastic greeting instead of returning the classification label `routine`. This broke the graph flow, as the system expected an exact key.

7.2. Configuration B: Low Temperature (0.1)

By reducing the temperature to 0.1 for the *Router* and *JSON Extractor*, system stability increased dramatically. The model stopped "embellishing" its responses and limited itself to following strict classification instructions.

However, using this temperature for *Motivational Chat* resulted in repetitive and dry responses. Therefore, we adopted a hybrid strategy:

- **Logical Nodes (Router/Tool):** Temperature 0.1 (Maximum precision).
- **Creative Nodes (Routine/Chat):** Temperature 0.7 (Controlled variability).

8. Agent Performance Evaluation

To evaluate the effectiveness of *IronAI*, we defined three qualitative metrics and ran a set of 20 interaction tests.

Table 1. Component Evaluation Summary

Component	Success Rate	Observations
Router	95%	Failed only on very ambiguous phrases like "I feel heavy" (BMI or Chat?).
JSON Extraction	85%	Very effective with standard formats ("80kg 1.80m"), less effective with long narrative.
Routine Quality	Subjective: High	Routines were anatomically coherent.

The system proved robust in 90% of typical use cases. The average latency in local execution was 3 to 5 seconds per response, which is acceptable for a personal assistant that doesn't require strict real-time interaction.

9. Error Analysis and Discussion

Despite overall success, we identified areas of friction:

9.1. Router Ambiguity

The most common error occurred in semantic "gray areas." Phrases like "I want to lose weight" could be interpreted as a routine request (`routine`) or a motivational chat (`chat`). Currently, the router tends to send it to chat. A future improvement would be to implement a "clarification" node that asks the user before deciding.

9.2. Small Model Limitations

Using Gemma-1b, we noticed that the ability to follow complex instructions in a single prompt is limited. If the prompt exceeded 10 lines of instructions, the model tended to forget the first ones. This validated our decision to use LangGraph: instead of one giant prompt, we used many small, focused prompts, which is ideal for compact models.

10. Conclusions

The development of *IronAI* demonstrates that a massive cloud model is not necessary to create useful and sophisticated intelligent agents. The true "intelligence" of the system doesn't reside solely in the neural network weights, but in the graph architecture that surrounds it.

By combining **LangChain** for text manipulation, **LangGraph** for logical flow management, and deterministic tools for calculation, we achieved an assistant that is greater than the sum of its parts. The project successfully meets the objective of integrating structured reasoning, adaptable personality, and technical precision in a single conversational interface.