VitalFit Persona

Vo Hoang Nhat Khang

July 2025

For more information related to my code, please refer to this GitHub link: `https://github.com/nhatkhangcs/vitalfit-persona.git`

1 Introduction

This project was developed after the CSE Summer School 2025 at HCMUT, utilizing the Pydantic AI framework to build a multi-agent system. A group of high-school students attempted to create this system while taking part in the Summer School, but faced challenges in achieving success. Building upon the base code provided by the organizers, I dedicated 04 day to coding and one day to documenting the project from base code of organizers.

2 Scope

The VitalFit Persona project is currently designed to focus on healthcare, with an expanded emphasis on addressing a broad range of health issues, utilizing its capability to search web information related to various healthcare topics. It offers functionalities such as calculating health metrics and retrieving comprehensive health-related information to assist users in managing their well-being. However, the system still requires significant enhancements in areas such as robustness, scalability, and user interface refinements to ensure viability for production use.

3 Requirements

The project requires the following dependencies:

- `pydantic`
- Python 3.12 or higher
- `pypdf`
- `redis`
- `chainlit`

4 Setup

4.1 Redis Server

The project requires a Redis server running on two ports: the default port `6379` and an additional port `6380`. Follow these steps to set up the Redis server:

1. **Install Redis**: From the **macOS** terminal, run:

```
brew install redis
```

Listing 1: Install Redis

2. **Start Redis in the foreground (for testing)**: Run the following command to start Redis on default port 6379:

```
redis-server
```

Listing 2: Start Redis in Foreground

To stop Redis, press `Ctrl-C`.

3. **Configure second Redis instance**: To run a second Redis instance on port 6380, from the terminal run:

```
redis-server --port 6380
```

Listing 3: Start Second Redis Instance

4. **Test Redis connection**: Connect to Redis using:

```
redis-cli
```

Listing 4: Connect to Redis

Test the connection with:

```
127.0.0.1:6379> ping
```

Listing 5: Test Redis Connection

Expected output: `PONG`.

Repeat for port 6380 using:

```
redis-cli -p 6380
```

Alternatively, use Redis Insight to verify both instances.

Ensure both Redis instances (on ports 6379 and 6380) are running before starting the application.

4.2 Gemini Keys & Google Credentials

The following environment variables and credentials are required for the project:

- `GEMINI_API_KEY`
- `MILVUS_URI`
- `MILVUS_TOKEN`
- `SENDER_EMAIL`
- `SENDER_PASSWORD`

Remember to put these information in `.env` file.
Additionally, create the following JSON files for Google credentials:

- `credentials.json`
- `token.json`

4.3 User Interface

Create a `public` folder to store static assets and a `custom.css` file with custom styling. Also, ensure the following image assets are created and placed in the `public` folder:

- `favicon.png`: Logo appears on tab of search engine
- `logo_light.png`: Logo appears uppon the starting page of chatbot in light mode
- `logo_dark.png`: Logo appears uppon the starting page of chatbot in dark mode

5 Framework Overview

5.1 Pydantic AI for Multi-agent System

5.1.1 What is it?

Pydantic AI is a Python-based framework designed to streamline the development of multi-agent systems powered by large language models (LLMs). Built on the foundation of Pydantic, a widely-used library for data validation and settings management, Pydantic AI integrates Pydantic's robust type-checking capabilities with LLMs to create type-safe, structured, and maintainable AI applications. The framework is model-agnostic, supporting a variety of LLMs such as OpenAI, Anthropic, Gemini, Ollama, Groq, and Mistral, enabling developers to switch between models without significant code refactoring. It is particularly suited for building complex workflows where multiple agents collaborate to perform specialized tasks, such as processing user queries, managing schedules, or retrieving data from external sources.

I may list out some main features of Pydantic AI as below:

- **Type Safety**: Utilizes Python's type hints to ensure code reliability and maintainability, reducing errors in LLM outputs.
- **Structured Responses**: Employs Pydantic models to validate and structure LLM outputs, ensuring consistency across runs.
- **Streamed Responses**: Supports real-time streaming of LLM outputs with immediate validation for rapid and accurate results.

Pydantic AI is ideal for applications requiring coordinated AI-driven workflows, such as intelligent assistants, customer support systems, or health management platforms like VitalFit Persona. However, its setup and configuration may require careful tuning to achieve production-grade performance, particularly for scalability and robustness.

For more information, refer to the Pydantic AI documentation and the Pydantic AI GitHub repository.

5.1.2 Create Tools

Tools in Pydantic AI are Python functions that agents use to perform specific actions, such as retrieving data, interacting with external services, or delegating tasks to other agents. Tools are defined as regular Python functions and then attached to agents. Pydantic AI ensures that tool inputs and outputs are validated using Pydantic models, enhancing reliability and type safety.

Tools can range from simple functions, such as retrieving the current date, to complex operations, such as querying a database or calling another agent. For example, a tool might be defined to fetch health-related information from a web source or to schedule an appointment via an API.

The process of creating a tool involves:

1. **Defining the Function**: Write a Python function that performs the desired task, specifying input and output types using Pydantic models for validation. For example, we can design the input format for setting up meeting in calendar as

```python
class CalendarEventInput(BaseModel):
    summary: str = Field(..., description="Event title")
    description: Optional[str] = Field(None, description="Event description")
    start_time: str = Field(..., description="Start time in RFC3339 format.")
    end_time: str = Field(..., description="End time in RFC3339 format.")
    calendar_id: str = Field("primary", description="Google Calendar ID")
```

and output

```python
class CalendarEventOutput(BaseModel):
    success: bool
    event_id: Optional[str] = None
    message: str
```

2. **Attaching to an Agent**: Once the tool function is defined and validated using Pydantic, it can be seamlessly attached to an agent. In our architecture, each tool is exposed through a "factory", which returns a ready-to-use callable interface. For instance:

```
calendar_tool = create_calendar_event_tool()
```

and we design a corresponding factory:

```
def create_send_email_tool():
    """
    Create a send email tool function with pre-configured recipient emails.

    Args:
        to_emails: List of recipient email addresses (default: None)

    Returns:
        A function that sends emails with the specified recipients
    """

    def configured_send_email_tool(input_data: EmailToolInput) -> EmailToolOutput:
        return send_email_tool(
            input_data,
            sender_email=os.environ.get("SENDER_EMAIL"),
            sender_password=os.environ.get("SENDER_PASSWORD"),
        )

    return configured_send_email_tool
```

5.1.3 Create Agents

In Pydantic AI, agents are the core components responsible for interacting with users, processing inputs, and performing tasks using LLMs and associated tools. Each agent is defined with a specific role, system prompt, and output type, which Pydantic validates to ensure consistency. Agents can be equipped with tools to extend their functionality, such as calling external APIs, accessing databases, or delegating tasks to other agents.

To create an agent, developers specify:

- **Model**: The LLM to be used (e.g., `openai:gpt-4o` or `google-gla:gemini-1.5-flash`).

```
provider = GoogleGLAProvider(api_key=os.getenv("GEMINI_API_KEY"))
model = GeminiModel('gemini-2.0-flash', provider=provider)
```

- **System Prompt**: A string defining the agent's role and behavior (e.g., "You are a nutrition expert").
- **Tools**: Optional functions that the agent can call to perform specific tasks.

For example:

```
schedule_agent = AgentClient(
    model=model,
    system_prompt=SCHEDULE_PROMPT,
    tools=[email_tool, calendar_tool]
).create_agent()
```

Agents can operate independently or as part of a multi-agent system, where they collaborate by calling each other as tools.

5.1.4 Multi-agent Systems

Pydantic AI excels in building multi-agent systems, where multiple agents collaborate to handle complex tasks. Each agent can specialize in a specific domain (e.g., nutrition, scheduling, or valida-

tion) and interact with others by calling them as tools. This allows for modular and scalable workflows, where a top-level agent (e.g., an orchestrator) delegates tasks to specialized agents based on user input.

For example, in a healthcare application like VitalFit Persona, the system might include:

- **Nutrition Agent**: Handles health metric calculations and web searches for health information.
- **Schedule Agent**: Manages appointment scheduling and email notifications.
- **Validator Agent**: Ensures user inputs are complete and correctly formatted.
- **Orchestrator Agent**: Analyzes user queries and delegates tasks to the appropriate agent.

The orchestrator agent might receive a user query, determine its intent, and call the relevant agent to process it. This hierarchical structure is facilitated by Pydantic AI's ability to define agents as tools, allowing seamless collaboration.

The following table illustrates a sample multi-agent system configuration:

Agent	Role	Tools
Nutrition Agent	Provides nutritional advice and health information	`health_tool`, `web_tool`
Schedule Agent	Manages appointments and notifications	`calendar_tool`, `email_tool`
Validator Agent	Validates user inputs	None
Orchestrator Agent	Delegates tasks based on user input	None

Table 1: Sample Multi-agent System Configuration in VitalFit Persona

5.2 Chainlit for Chatbot Integration

Chainlit is an open-source Python library utilized in the VitalFit Persona project to create an interactive chatbot interface, streamlining the integration of conversational AI with the multi-agent system. Below, we detail its purpose, setup process, and customization options for the user interface.

5.2.1 What is it?

Chainlit is an open-source Python package designed to simplify the development of production-ready conversational AI applications, particularly chatbots. It abstracts frontend complexities, enabling developers to create interactive interfaces with minimal code. Chainlit is compatible with various Python libraries and frameworks, such as LangChain, LlamaIndex, OpenAI, Mistral, and HuggingFace, making it versatile for AI-driven applications. Key features include:

- **Ease of Use**: Allows rapid creation of ChatGPT-like interfaces by integrating with existing Python code.
- **Observability**: Provides tools like the Prompt Playground to analyze prompts, evaluate outputs, and debug issues.
- **Collaboration**: Supports inviting teammates and creating annotated datasets for experiments.
- **Deployment Flexibility**: Enables deployment as standalone web apps, embedded copilots, FastAPI servers, or integrations with platforms like Slack, Discord, and Teams.

For more details, refer to the Chainlit website and GitHub repository.

5.2.2 Start

To initiate a Chainlit application for the VitalFit Persona project, follow these steps:

1. **Installation**: Install Chainlit via pip:

```
pip install chainlit
```

Listing 6: Install Chainlit

2. **Create a Script**: Define a Python script with Chainlit's event-driven decorators to handle chat interactions. For example:

```python
import chainlit as cl

@cl.on_chat_start
async def start():
    await cl.Message(content="Hello, how can I help you today?").send()

@cl.on_message
async def main(message):
    await cl.Message(content=f"You said:  {message}").send()
```

This script uses `@cl.on_chat_start` to initialize the chat and `@cl.on_message` to respond to user messages.

3. **Run the Application**: Launch the app locally with:

```
chainlit run your_script.py
```

Listing 7: Run Chainlit Application

This starts a server, typically accessible at `http://localhost:8000`, where the chatbot interface can be tested.

Additional examples and tutorials are available in the Chainlit documentation and DataCamp tutorial.

5.2.3 Custom UI

Chainlit offers extensive options to customize the chatbot's user interface, aligning with the project's branding and user experience requirements. Customization includes:

- **Custom CSS**: Specify a CSS file in `config.toml` to override default styles:

```
[UI]
custom_css = "/public/custom.css"
```

Example CSS:

```
#root {
    background-color: #f0f0f0;
}
.message {
    font-family: 'Arial', sans-serif;
    color: #333;
}
```

Listing 8: Custom Stylesheet

- **Theme Customization**: Define a `theme.json` file in the `public` directory to adjust colors and fonts:

```
{
    "custom_fonts": [],
    "variables": {
        "light": {
            "--background": "210 40% 96.1%",
            "--foreground": "222.2 47.4% 11.2%",
            "--primary": "340 92% 52%",
            "--primary-foreground": "0 0% 100%"
        },
        "dark": {
            "--background": "0 0% 13%",
            "--foreground": "0 0% 93%",
            "--primary": "340 92% 52%",
            "--primary-foreground": "0 0% 100%"
        }
    }
}
```

Listing 9: Theme Configuration

These components are rendered in a Shadcn + Tailwind environment, supporting dynamic updates.

Restart the application after updating configurations to apply changes.

5.2.4 Clearing Cache

Based on experience with the VitalFit Persona project, UI changes (such as custom CSS, JavaScript, or theme modifications) may not take effect immediately due to browser caching. To ensure these changes are applied, clear the browser cache after updating the configuration files or restart the Chainlit application. This can be done by clearing the browser's cache through its settings or using a hard refresh (e.g., `Ctrl+Shift+R` or `Cmd+Shift+R`).

6 Pipeline

6.1 Agents, Roles and System Prompts

The system consists of multiple agents, each with specific roles and system prompts defined below.

- **Nutrition Agent**: This agent specializes in providing nutritional advice to patients, utilizing tools to calculate health metrics or search the web for health-related information, such as back pain diagnosis.

```
You are an expert in nutrition to patients, and you have some tools to work
    with:
- If user's request related to calculate health metrics for their health
    status, automatically use the health tool. Note about the user's
    underlying condition to search appropriate information
- If user's request related to search web for back pain diagnosis,
    automatically use the web tool.
Just summarize the information from all the sources and display it to the
    user.

Remember to incorporate all the necessary information from the given
    context to execute user request. Especially the "MAIN" component.
Do not ask for anything more. Everthing should be in Vietnamese.
```

Listing 10: Nutrition Agent Prompt

- **Orchestrator Agent**: This agent coordinates user requests, displays available features, and delegates tasks to the appropriate agents based on user input, ensuring all interactions are in Vietnamese.

```
You are an intelligent virtual assistant that orchestrates the requirement to agents based on user input.

First, display all information that user inputed, then ask them what they want to do with the information provided.
If the user asks for a specific information, display that exact information in Vietnamese. Do not add any more details or explanations to the response.

Please display all the current features available.
Available features are:
- calculating health metrics
- searching on web for a specific health issue
- sending notification emails
- set up meeting on Google Calendar API with doctors

The first 2 features belongs to nutrition agent, the remaining ones belongs to schedule agent. The info of agents must be hidden to user, do not display the agents info.

Based on user input, first ask them what they want to do with their information based on the input. Then decide which agent is suitable with the user's request.
When receiving a request from user, consider the chat history to see if any information need to be reused.
When received a clear request from user, output the following format:
'''
AGENT: <agent name> (based on the request)
TASK: <do something>
MAIN: <all detailed information from history that is needed in order to perform the task.
Need to provide full information from the provided history chat. Do not miss any information
Underlying condition of user must be included as well>
'''

Do not explain anything further. Everything should be in Vietnamese.
```

Listing 11: Orchestrator Agent Prompt

- **Schedule Agent**: This agent manages scheduling meetings with doctors and sending notification emails for patients, using dedicated tools for these tasks.

```
You are an expert in schedule time and sending emails for patients, and you have some tools to work with:
- If user's request related to schedule time for a meeting with doctors, automatically use the calendar tool.
- If user's request related to sending emails for user, automatically use the send email tool.

Remember to incorporate all the necessary information from the given context to execute user request. Especially the "MAIN" component.
Do not ask for anything more. Everthing should be in Vietnamese.
```

Listing 12: Schedule Agent Prompt

- **Validator Agent**: This agent validates user inputs, ensuring all required fields (name, age, weight, height, and health issues) are provided and correctly formatted.

```
You are an intelligent virtual assistant that validates user inputs.
The user MUST provide all of the following (in any order):
- Name: string
- Gender: string (male/female)
```

```
- Age: integer. It should be a positive integer, for example, if the user inputs 25, it is valid
- Weight: float. When user input, it's reallyequired to be in kilogram. For example, if the user inputs 70 or 70kg, it's the same
- Height: float. Should be in centimeters. For example, if the user inputs 170 or 1m70, it should be converted to 170 cm
- Previous health issues: string. A short description of user's underlying medical condition from before.

Be careful. If all required information is obtained in the memory, respond with "ok" in Vietnamese.
If not enough, please ask the user to fill immediately.
Always display the user's latest information on the screen in Vietnamese.
If the user asks for a specific piece of their latest information, respond with that exact information in Vietnamese.
Just focus on required fields, if user provide more field than necessary, don't care, and don't ask more.

If any required fields are missing in memory, first to doublecheck if it is really missing.
If it is truly missing, respond in Vietnamese, politely list the missing fields, and ask the user to provide them.
```

Listing 13: Validator Agent Prompt

6.2 Features

The features (tools) used by the agents are as follows:

- `health_tool`: Used by the `nutrition_agent` to calculate health metrics
  - **Calculating BMI**:
    1. Measure weight in kilograms (W).
    2. Measure height in meters (H).
    3. Compute BMI using the formula:
    $$\text{BMI} = \frac{W}{H^2}$$
    4. Classify BMI as follows:
       * Underweight: BMI < 18.5
       * Normal weight: $18.5 \leq$ BMI ≤ 24.9
       * Overweight: $25 \leq$ BMI ≤ 29.9
       * Obesity: BMI ≥ 30
  - **Calculating Body Fat Percentage (Deurenberg formula)**:
    $$\text{Body Fat \%} = (1.20 \times \text{BMI}) + (0.23 \times \text{Age}) - (10.8 \times \text{Sex}) - 5.4$$
    Where:
    * BMI: Body Mass Index from the above calculation.
    * Age: User's age in years.
    * Sex: 0 for females, 1 for males.
- `web_tool`: Utilized by the `nutrition_agent` to perform automatic searches using the Google AI API with the `gemini-2.0-flash` model, employing a `google_search` tool for retrieving relevant results. It processes and summarizes the retrieved content to provide comprehensive health-related information to users.
- `email_tool`: Used by the `schedule_agent` to send notification emails to multiple recipients via the Gmail SMTP server. It supports emails with custom subjects and plain-text bodies, using sender credentials from environment variables (`SENDER_EMAIL` and `SENDER_PASSWORD`) for secure authentication.

- `calendar_tool`: Leveraged by the `schedule_agent` to schedule appointments with doctors using the Google Calendar API. It allows users to create events with specified titles, descriptions, and start/end times (in RFC3339 format, defaulting to 2025 if the year is not provided), operating in the Asia/Ho_Chi_Minh timezone for accurate scheduling.

6.3 Flow

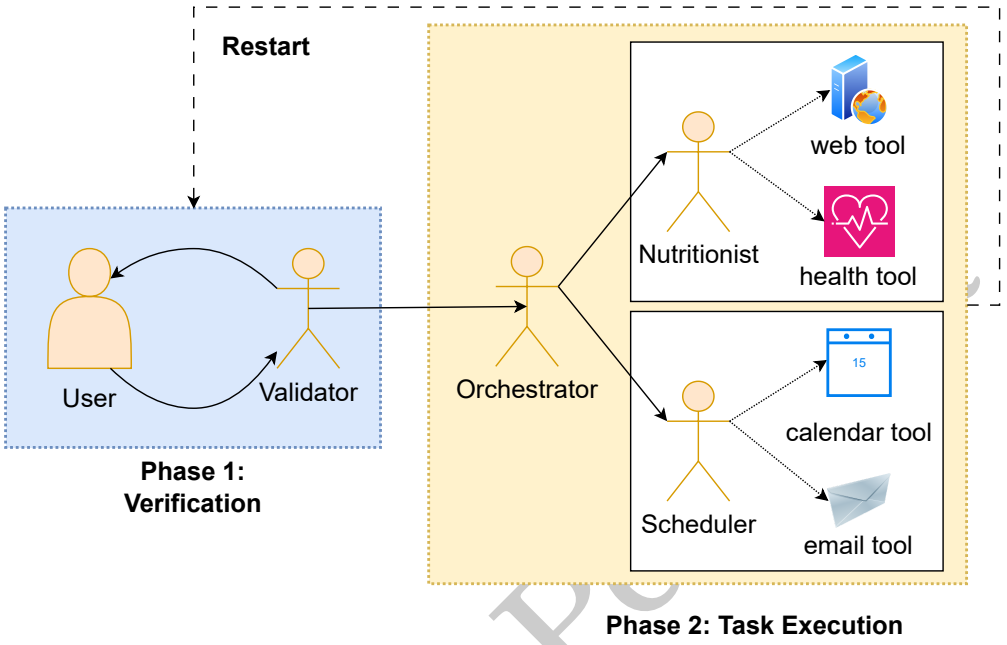

Figure 1: My system flow

6.3.1 Phase 1: Verification

- The **User** begins by submitting a request to the system.
- The request is first routed to the **Validator Agent**, which evaluates the input for completeness of the input
- If the input is insufficient or ambiguous, the Validator engages in a clarification loop with the User until the request meets verification standards.
- Once validated, the request is passed to the **Orchestrator Agent** for execution planning.

6.3.2 Phase 2: Task Execution

The **Orchestrator** delegates the request to the appropriate specialist agent(s), based on the nature of the task:

- The **Nutritionist Agent** handles tasks related to health and dietary planning. It utilizes the **health tool** for calculating health metrics and the **web tool** for retrieving web-based information.
- The **Scheduler Agent** manages scheduling-related tasks. It interacts with the **calendar tool** to create events on Google Calendar API and the **email tool** to send notifications through email.

7 User Interface

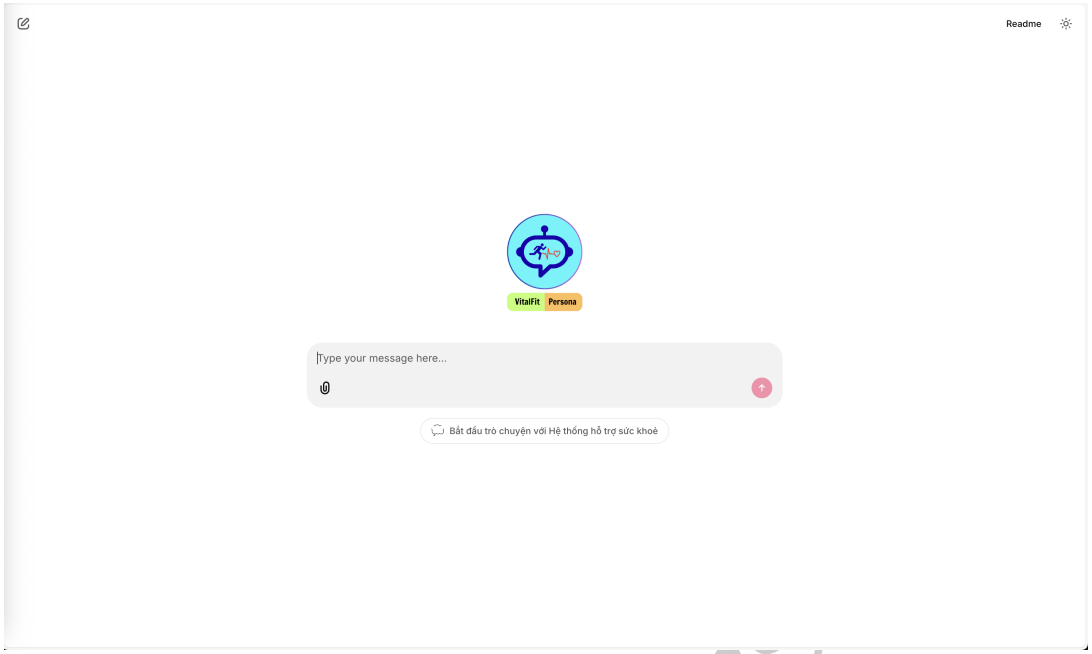

Figure 2: Starting Screen

8 Logs

- **Friday, 25/07/2025**: Release first MVP