Yann GUEGUEN and Sean SAMUEL

2024 - 2025

Food Tracking SaaS with AI assistant project

Contents

1 Introduction

Here's a link showing how the food search works: https://youtu.be/QN8HwCjERxo

Link for the scrapping project from openfoodfact : https://github.com/YGueguen16u/database-creator

1.1 Project Context

In recent years, personal health monitoring has become a key concern for individuals striving to optimize their lifestyle, improve physical performance, or manage chronic health conditions. While food-tracking applications such as *MyFitnessPal* or *Yazio* offer useful interfaces, they often lack intelligence and flexibility, relying heavily on manual entry. The rise of **large language models** (LLMs) and **cloud-native infrastructures** provides new opportunities to create more dynamic, personalized, and intelligent tools in this domain.

This project was developed during our **data science and engineering program** to demonstrate our skills in **full-stack development** and **natural language processing**. The objective was to build a complete **SaaS platform** for personalized food and activity tracking, integrating a fine-tuned **AI assistant**, modern cloud infrastructure, and real-time user interaction.

1.2 General Objective

The main goal of the project is to provide a seamless food and activity tracking experience that minimizes manual effort and maximizes user engagement. Users can interact with an AI assistant through a *chat-based interface* to automatically log meals and physical activities. The assistant can understand free-form inputs and generate structured entries based on product data extracted from **OpenFoodFacts**.

The system supports:

- Real-time entry and visualization of meals and activities;
- Search of food products stored on **AWS S3**;
- Secure storage of user data in a **PostgreSQL database** (AWS RDS);
- LLM-driven inference to convert natural language into structured logs.

However, due to time and GPU limitations, the AI assistant was not fully integrated into the deployed infrastructure. While the model and the web platform both function independently, their communication pipeline remains incomplete at the time of submission.

1.3 Target Audience

The application is designed for:

- Health-conscious individuals seeking personalized tracking tools;

- Developers interested in applied NLP and SaaS integration;
- Students exploring the full lifecycle of an AI-powered product (from scraping to deployment).

1.4 Overview of Technologies Used

The system combines several modern technologies to ensure modularity, scalability, and performance:

- **FastAPI** backend exposing REST endpoints;
- **React + Vite** frontend for dynamic and responsive UI;
- **AWS S3** for storing transformed nutritional product data;
- **AWS RDS** (**PostgreSQL**) for user log persistence;
- **Mixtral-8x7B-Instruct** fine-tuned using **LoRA** and **Axolotl**;
- **Docker** for containerization and **GitHub Actions** for CI/CD;
- **Render** for backend deployment and **Vercel** for frontend hosting.

While the AI assistant is fully fine-tuned and capable of generating structured output, only a single successful log generation was performed due to time and GPU constraints. Full integration between the model and the web application was initiated but remains *incomplete at the time of submission*.

2 State of the Art: Recent Advances in LLMs and Their Integration in SaaS Products

Over the past few years, large language models (LLMs) have undergone rapid evolution, driven by breakthroughs in transformer architectures, scaling laws, and fine-tuning techniques. Models such as **GPT-4**, **Claude**, **Mixtral**, and open-weight alternatives like **Mistral** and **LLaMA** have demonstrated the capacity to perform complex reasoning, follow instructions, and generate structured outputs from free-form prompts.

2.1 LLMs in Consumer Applications

LLMs have begun to power a wide range of consumer-facing applications, from virtual assistants and customer support bots to productivity tools and personal planners. The rise of AI copilots—like GitHub Copilot for coding or Notion AI for note-taking—illustrates a broader trend: the shift from passive data entry interfaces to *interactive, intelligent systems* capable of understanding user intent and acting accordingly.

These systems typically leverage:

- Instruction-tuned models trained on human-like dialogues;

- Fine-tuning or prompt engineering to specialize the LLM on a domain (e.g., fitness, nutrition, law);
- Hybrid infrastructures combining retrieval-augmented generation (RAG), structured APIs, and AI agents.

2.2 From LLMs to AI-Powered SaaS

The integration of LLMs into Software-as-a-Service (SaaS) platforms marks a paradigm shift. Rather than being static form-based interfaces, modern SaaS tools increasingly offer natural language interaction and personalized automation. For instance:

- **Jasper** and **Copy.ai** use LLMs to generate marketing copy;
- **ChatGPT Enterprise** and **Notion AI** enhance productivity with contextual understanding;
- **Synthesia** uses LLMs and generative models to create video content from text instructions.

2.3 Challenges and Opportunities

While the opportunities are vast, integrating LLMs into SaaS applications introduces new technical and ethical challenges:

- **Latency and cost**: LLM inference can be expensive and slow without quantization or model optimization;
- **Security and privacy**: Handling sensitive user data requires secure architectures, especially in health-related contexts;
- **Reliability**: LLMs must generate deterministic and valid outputs, especially when driving backend actions like database updates.

Our project explores these challenges in the context of food and fitness tracking, using a fine-tuned open-weight model (Mixtral-8x7B-Instruct) to structure user inputs. Although full integration with the live interface was not finalized.

3 Functional Specifications

3.1 Usage Scenarios (Meal Logging, AI Interaction, Product Search...)

The application supports a variety of real-world usage scenarios tailored to users aiming to track their nutrition and physical activity. The main scenarios include:

- **Manual meal logging:** The user manually adds one or more meals (e.g., breakfast, lunch, snacks), including foods, quantities, and timestamps.

- **Physical activity tracking:** The user registers activities such as walking, running, weight training, or swimming, with duration.
- **AI-driven input:** Instead of using forms, the user sends a free-form message to the assistant (e.g., *"I had two bananas and ran 30 minutes"*), and the assistant extracts and fills structured logs.
- **Product search:** Users can search for food items from a curated OpenFoodFacts database, hosted in AWS S3, and receive structured nutritional information.
- **Daily overview:** The user can visualize their daily log (meals, calories, exercise, chatbot interaction) from the frontend.

These scenarios cover both manual and AI-assisted workflows, enabling flexibility and personalized user experience.

3.2 Use case diagram

The use case diagram in Figure 18 summarizes the core user interactions with the system. It distinguishes between direct user actions and delegated actions handled by the AI assistant.

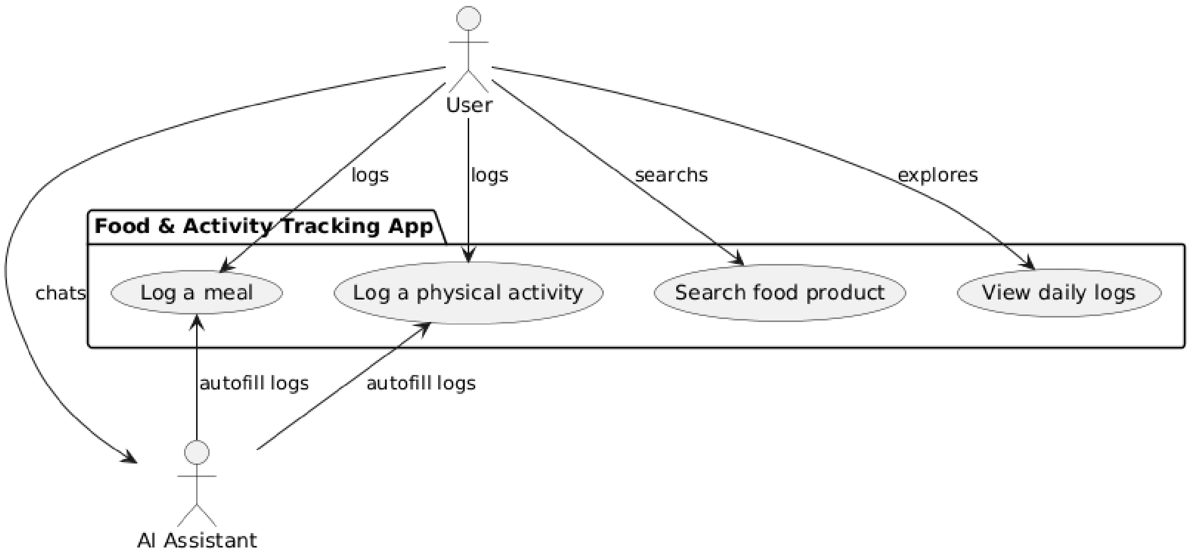


Figure 1: *Use Case Diagram – Food and Activity Tracking Application*

3.3 Full user journey

A typical user journey proceeds as follows:

1. The user registers or logs into the application (future integration with AWS Cognito).
2. The user sets dietary preferences or goals (optional).
3. The user searches for food products or adds meals manually via the UI.

4. Alternatively, the user interacts with the chatbot and describes their daily intake or activity in natural language.
5. The assistant interprets the message, generates structured JSON (with commands), and sends back a suggestion to auto-fill the log.
6. The user reviews and validates the AI-suggested log (UI integration pending).
7. The complete log is stored in the PostgreSQL database under the user's account.
8. The user can navigate through different days and view historical entries.

This flow emphasizes flexibility: users can choose between traditional interfaces and AI-enhanced interaction, depending on their preferences and context.

4 Software Architecture

4.1 Global Architecture Diagram

The system is built as a modular full-stack web application with a modern cloud-native architecture. It follows a clear separation of concerns between frontend, backend, database, AI inference, and external storage layers.

The core components include:

- **Frontend** hosted on **Vercel**, built with React + Vite;
- **Backend API** hosted on **Render**, implemented with FastAPI and containerized via Docker;
- **Relational Database** hosted on **AWS RDS (PostgreSQL)** for user logs and metadata;
- **Static product data** stored in **AWS S3** as JSON files (food database);
- **LLM inference** running locally or via script (not yet integrated into deployed backend);
- CI/CD pipelines using **GitHub Actions** for deployment and testing automation.

A complete deployment diagram is included in the appendix.

4.2 Description of Each Component

Frontend (React + Vite): The frontend is a responsive single-page application offering a clean interface for:

- Logging meals and physical activities;
- Searching food products via a user-friendly search box;
- Chatting with the AI assistant;
- Managing dietary preferences and goals.

Backend (FastAPI): The backend handles all API logic and database interactions. Key features:

- RESTful API endpoints for meals, activities, and user profiles;
- File querying to S3 for OpenFoodFacts product search;
- JSON-based logging system for daily user data;
- Chatbot route for AI-generated output (via inference script).

Database (PostgreSQL - AWS RDS): Used for persistent storage of:

- User profiles;
- Daily logs including meals, foods, activities, and assistant messages;
- Relationships between users, goals, and consumption patterns.

Cloud Storage (AWS S3): OpenFoodFacts product data is preprocessed, cleaned, and stored in S3 buckets as structured JSON files. These are queried during food searches by the user.

LLM Component: Mixtral-8x7B-Instruct, fine-tuned with QLoRA using the Axolotl framework, is capable of generating JSON commands and structured meal logs. Although the model is trained, it is currently decoupled from the live backend and used through a local script during testing.

4.3 Key API Endpoints

- `POST /api/log` – Save daily user logs;
- `GET /api/log` – Retrieve logs for a specific user and date;
- `POST /api/search_product` – Search for food items in S3 by name or brand;
- `POST /api/chat` – Send a user message to the chatbot and receive structured response;
- `GET /api/goals` – Retrieve preset dietary or fitness goals;
- `POST /api/user` – Register or update user profile (with dietary data);
- `POST /api/activities` – Log physical activities manually;
- `GET /api/foods` – Return predefined food data from the backend or cache.

These endpoints are consumed by the frontend and, in future versions, may also be triggered automatically by the assistant once fully integrated.

5 Food Data Scraping and Processing

5.1 Source: OpenFoodFacts

To support food logging and nutrient analysis, the application relies on public datasets provided by **OpenFoodFacts**, a collaborative and open-access database of food products from around the world.

It includes nutritional information, ingredient lists, packaging types, environmental labels, and brand metadata.

5.2 Raw Data Structure (HTML, JSON API)

OpenFoodFacts provides both:

- An HTML-based interface for human navigation and scraping;
- A structured JSON-based API, accessible via endpoints such as: `https://world.openfoodfacts.org/api/v0/prod`

Each product entry may include:

- **Product name**, **brand**, and **categories**;
- **Nutritional information**: calories, proteins, fats, sugars, etc.;
- **Labels and tags**: organic, vegan, eco-score, nutri-score;
- **Image links** and packaging details.

However, the data quality is heterogeneous and often incomplete, necessitating additional filtering and cleaning.

5.3 Scraping Pipeline (HTML Pages)

The scraping process was implemented in Python using libraries such as `requests`, `BeautifulSoup`, and optionally `Selenium` for dynamically loaded content. The pipeline performs the following steps:

1. Retrieve a list of product pages through category listings;
2. For each product:
   - Extract relevant HTML tags (product title, nutrition table, ingredients);
   - Parse structured JSON-LD if available;
3. Store the raw data locally before transformation.

5.4 Data Cleaning, Transformation, and Upload to S3

To standardize and optimize access to product metadata:

- Each product is transformed into a simplified JSON structure;
- Nutrients are normalized into common units (e.g., per 100g);
- Only essential fields are retained (name, brand, nutrients, labels);
- Cleaned JSON files are uploaded to **AWS S3** using a custom `S3Manager` class.

Two separate datasets were created:

- `products_ean13.json`: food products with standard 13-digit barcodes;
- `products_ean8.json`: products with 8-digit barcodes (typically small items).

These files are accessed by the backend for search and autocomplete functionality.

5.5 Applied Filters (Labels, Relevance Criteria...)

To ensure quality and usability:

- Only products with a valid barcode, name, and nutritional info were kept;
- Preference was given to products with **Nutri-Score** and **Eco-Score**;
- Duplicates and incomplete entries were removed.

6 Database and Storage

6.1 Relational model (tables, keys, relationships)

The backend relies on a normalized relational schema using **PostgreSQL** to store all user-specific information. Core tables include:

- `users`: basic user identity and authentication metadata;
- `meals`: high-level meal entries linked to users by foreign key;
- `foods`: list of foods consumed in each meal (includes nutritional breakdown);
- `activities`: user-reported physical activities (duration, type, intensity);
- `user_daily_logs`: JSON-based aggregation of meals, activities, and assistant interactions for each date.

Relationships are enforced via foreign keys to ensure referential integrity. The `user_daily_logs` table supports unstructured yet validated storage of multi-modal data per user and day.

6.2 Use of AWS RDS (PostgreSQL)

The PostgreSQL instance is hosted via **Amazon RDS**, allowing for reliable, scalable, and secure cloud-based database access. Connection credentials are injected via environment variables and managed during Docker container deployment.

Benefits of using AWS RDS include:

- Secure credentials and network configuration (via security groups or VPC);
- Backups and snapshots to avoid data loss;

- Read/Write performance suited to user logging and AI-inferred data insertion;
- Compatibility with `psycopg2` and `SQLAlchemy ORM` in FastAPI.

6.3 JSON structure on S3 (cleaned food products)

The project uses **Amazon S3** to store and serve a cleaned dataset extracted from **OpenFoodFacts**. This dataset includes relevant product metadata:

- Product name and brand;
- Nutritional values (calories, macronutrients, micronutrients);
- Labels and environmental indicators (e.g. organic, Nutri-Score);
- Packaging details, categories, and unique barcode identifiers.

The files are stored as lightweight `.json` objects (split by barcode type: EAN-8 / EAN-13) and accessed via the `boto3` client in the backend. This decouples the food product database from the core RDS schema and enables cheap, scalable data access across all frontend users.

7 AI / NLP Component

This section presents the architecture, training methodology, and output structure of the AI assistant integrated in our food and activity tracking platform. We also discuss qualitative results and known limitations.

7.1 Fine-tuning Setup: Base Model, Dataset, LoRA, Axolotl

To enable intelligent interaction with users, we fine-tuned a large language model on domain-specific data. The base model used is **Mixtral-8x7B-Instruct-v0.1**, chosen for its instruction-following capabilities and efficient multi-expert routing.

The fine-tuning was conducted using the **LoRA** (Low-Rank Adaptation) technique in 4-bit precision via **QLoRA**, which reduces GPU memory usage while retaining performance. The training was orchestrated with **Axolotl**, a flexible open-source framework for supervised LLM fine-tuning.

The dataset consists of **multi-turn conversations** between a user and an assistant, written in French. Each example includes user input in natural language and a structured assistant response containing an `assistant_output` and one or more JSON `commands` representing explicit actions to perform (e.g., filling meals, logging activities).

7.2 Prompt and Output Format (assistant_output, commands)

All prompts are formatted using the instruction-based structure expected by Mixtral:

```
[INST] j'ai mangé 2 pommes et couru 30 minutes [/INST]
```

The assistant is expected to return a JSON-style response, structured as follows:

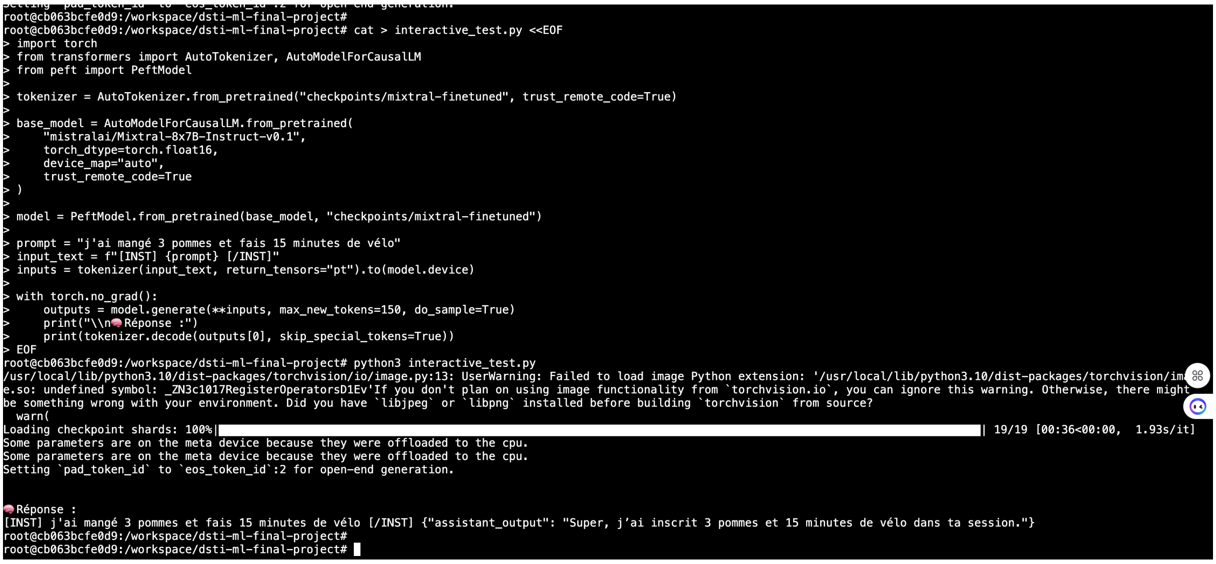
```
{
  "assistant_output": "Bien noté, j'ai ajouté 2 pommes et 30 minutes de course.",
  "commands": [
    {
      "type": "fill_meal",
      "meal_type": "snack",
      "foods": [
        {
          "name": "pomme",
          "quantity": 2,
          "unit": "unité"
        }
      ]
    },
    {
      "type": "fill_activity",
      "activity_type": "course",
      "duration_minutes": 30
    }
  ]
}
```

This structured response can be parsed by the backend to automatically populate user logs in the database without additional user input.

7.3 Qualitative Evaluation (Examples)

A sample generation was tested using the prompt:

```
[INST] j'ai mangé une compote et fait 20 squats [/INST]
```



```
root@cb063bcfe0d9:/workspace/dsti-ml-final-project#
root@cb063bcfe0d9:/workspace/dsti-ml-final-project# cat > interactive_test.py <<EOF
> import torch
> from transformers import AutoTokenizer, AutoModelForCausalLM
> from peft import PeftModel
>
> tokenizer = AutoTokenizer.from_pretrained("checkpoints/mixtral-finetuned", trust_remote_code=True)
>
> base_model = AutoModelForCausalLM.from_pretrained(
>     "mistralai/Mixtral-8x7B-Instruct-v0.1",
>     torch_dtype=torch.float16,
>     device_map="auto",
>     trust_remote_code=True
> )
>
> model = PeftModel.from_pretrained(base_model, "checkpoints/mixtral-finetuned")
>
> prompt = "j'ai mangé 3 pommes et fais 15 minutes de vélo"
> input_text = f"[INST] {prompt} [/INST]"
> inputs = tokenizer(input_text, return_tensors="pt").to(model.device)
>
> with torch.no_grad():
>     outputs = model.generate(**inputs, max_new_tokens=150, do_sample=True)
>     print("\\n🧠Réponse :")
>     print(tokenizer.decode(outputs[0], skip_special_tokens=True))
> EOF
root@cb063bcfe0d9:/workspace/dsti-ml-final-project# python3 interactive_test.py
/usr/local/lib/python3.10/dist-packages/torchvision/io/image.py:13: UserWarning: Failed to load image Python extension: '/usr/local/lib/python3.10/dist-packages/torchvision/im
e.so: undefined symbol: _ZN3c1017RegisterOperatorsD1Ev'If you don't plan on using image functionality from `torchvision.io`, you can ignore this warning. Otherwise, there might
be something wrong with your environment. Did you have `libjpeg` or `libpng` installed before building `torchvision` from source?
  warn(
Loading checkpoint shards: 100%|██████████| 19/19 [00:36<00:00,  1.93s/it]
Some parameters are on the meta device because they were offloaded to the cpu.
Some parameters are on the meta device because they were offloaded to the cpu.
Setting `pad_token_id` to `eos_token_id`:2 for open-end generation.


🧠Réponse :
[INST] j'ai mangé 3 pommes et fais 15 minutes de vélo [/INST] {"assistant_output": "Super, j'ai inscrit 3 pommes et 15 minutes de vélo dans ta session."}
root@cb063bcfe0d9:/workspace/dsti-ml-final-project#
root@cb063bcfe0d9:/workspace/dsti-ml-final-project#
```

Figure 2: *Result*

The assistant successfully returned a coherent output, correctly identifying food and exercise types. However, due to time and GPU constraints, only a single interaction was tested, and the chatbot could not yet be fully integrated with the application UI.

7.4 Current Limitations and Potential Risks

- **Integration gap**: While the assistant is fine-tuned and functional, its real-time integration into the application remains incomplete due to infrastructure constraints.
- **Generalization**: The assistant is trained on a limited number of prompts and may not generalize well to highly ambiguous or poorly formatted inputs.
- **Security**: Model outputs are not yet filtered or validated, which could pose risks if deployed in production without guardrails.
- **Inference cost**: Running the 8x7B Mixtral model, even in quantized form, can be computationally expensive, making real-time deployment challenging without optimized infrastructure (e.g., GGUF, vLLM).

8 Infrastructure & DevOps

This project follows modern DevOps practices to ensure maintainability, reproducibility, and scalability across environments. The infrastructure is containerized and leverages cloud services for both deployment and storage, while also enabling automated workflows for continuous integration and delivery.

8.1 Backend Unit Tests

Although the unit test suite is still in progress due to time constraints, the project includes a dedicated `tests/` directory and follows the **pytest** framework structure. Unit testing is a key process that I aim to consistently integrate in all future projects, especially for API endpoints and data validation logic.

8.2 Manual Validation Strategies (UI, AI Behavior)

In the absence of complete unit tests, validation was performed manually on:

- The frontend interface (meal logging, product search, chat UI);
- Backend endpoints via tools like Postman and direct API calls;
- The AI assistant outputs during local inference, with qualitative analysis of structured JSON generation from user inputs.

Although only one successful inference-to-log scenario was achieved due to GPU and integration constraints, the foundation is functional and can be extended.

8.3 Dockerization (Dockerfile, docker-compose.yml)

The backend is containerized with:

- A custom `Dockerfile` for the FastAPI application, including the fine-tuned LLM and all dependencies;
- A `docker-compose.yml` file that orchestrates services (backend API, database, and optionally model checkpoint directories) for local or production deployment.

This ensures consistency between environments and makes onboarding and deployment easier.

8.4 CI/CD with GitHub Actions

A continuous integration pipeline is defined under `.github/workflows/`, enabling:

- Linting and formatting checks on push;
- Automated Docker builds;
- (Planned) unit test runs and deployment hooks for Render and Vercel.

This setup improves code quality and accelerates iteration cycles.

8.5 Deployment Stack: Render, Vercel, AWS (S3, RDS)

- **Render**: Hosts the containerized FastAPI backend. The image is built from the Dockerfile and linked to the main Git branch.

- **Vercel**: Hosts the React frontend, auto-deployed on commit.
- **AWS S3**: Stores cleaned OpenFoodFacts data as JSON blobs.
- **AWS RDS**: Serves as the persistent PostgreSQL database for storing all user logs and configurations.

These tools were selected for their scalability and free-tier compatibility during development.

8.6 Basic Security Handling (.env Files, Cloud Access...)

Environment variables are stored in a `.env` file, loaded via `python-dotenv` for local runs and injected in production via platform-specific settings (Render and Vercel dashboards). Sensitive credentials such as:

- AWS keys;
- PostgreSQL credentials;
- S3 bucket names;

are never hardcoded and are kept outside the codebase. Access policies on S3 and RDS were minimally secured via IAM roles and security groups, with future improvements planned around token-based authentication (e.g., AWS Cognito).

9 Future Improvements

This project, while functional and showcasing core capabilities, leaves room for several valuable enhancements that would further improve its scalability, usability, and intelligence.

9.1 Planned Integration of AWS Cognito

A full integration with **AWS Cognito** is planned. This will allow for:

- Secure and scalable user authentication;
- Token-based access to backend routes;
- Easy connection with other AWS services via IAM roles;
- Fine-grained session management.

9.2 Mobile UI or PWA

While the React interface is responsive, a dedicated **mobile experience** or **Progressive Web App (PWA)** would greatly increase user engagement and retention. This could include:

- Offline support for logging meals and workouts;

- Push notifications for meal reminders or AI suggestions;
- Integration with mobile health data APIs (e.g., Apple Health, Google Fit).

9.3 Personalized Recommendations

The next logical step is to make the assistant proactive and contextual. Planned features include:

- Nutritional advice based on meal history;
- Activity suggestions based on user goals and habits;
- Warnings for dietary imbalance (e.g., low protein, excess sugar).

This would require incorporating a **recommendation engine** based on historical logs and user profiles.

9.4 LLM Inference Optimization (Quantization, Caching...)

Deploying a large LLM such as Mixtral in production is currently expensive and slow. Several optimizations are being considered:

- **Quantization** using 4-bit or 8-bit models (e.g., GGUF, vLLM);
- **Session caching** for repeated or similar queries;
- Model distillation to smaller, cheaper inference backends;
- Switching to inference frameworks like **vLLM** or **TGI** for performance.

These improvements would enable real-time assistant interaction at a sustainable cost and latency.

10 Conclusion

10.1 Summary of Achievements

This project aimed to design and implement a cloud-native SaaS platform for personalized food and physical activity tracking, enhanced by a fine-tuned AI assistant. Over the course of development, the following key objectives were met:

- Design and deployment of a full-stack architecture using **FastAPI**, **React**, and **PostgreSQL**;
- Integration of cloud services such as **AWS RDS** and **AWS S3** for secure and scalable data storage;
- Implementation of a chatbot interface powered by a fine-tuned **Mixtral-8x7B-Instruct** model using **LoRA** and **Axolotl**;
- Construction of a structured, queryable food database from **OpenFoodFacts** via scraping and transformation;
- CI/CD setup with GitHub Actions and containerization via Docker for reproducible deployment;

- Unit testing of the backend and modular organization of services (S3, NLP, ORM).

The system provides an interactive, extensible interface where users can log meals, track activity, and receive structured assistance from the AI.

10.2 Key Challenges

Several challenges were encountered throughout the project:

- **Infrastructure and Deployment**: Running inference with large LLMs locally or on low-resource machines (e.g., Mac) proved unfeasible due to GPU and memory requirements. Only limited testing of inference was possible on RunPod due to cost constraints;
- **AI Integration**: Although the assistant was successfully fine-tuned and generated structured outputs, its integration into the deployed backend was not completed in time. As a result, model inference and app usage remain decoupled;
- **Data Preparation**: Cleaning and transforming food product data required extensive filtering, restructuring, and verification to ensure it was usable for real-time queries;
- **Time Constraints**: The full pipeline—from scraping, modeling, and fine-tuning to deployment and user interaction—was ambitious within the timeframe of an academic project.

10.3 Outlook for Future Development

This application offers strong potential for expansion and industrial relevance. Planned improvements include:

- Full integration of the fine-tuned LLM into the production backend with optimized inference (e.g., via vLLM or quantized models);
- Secure and scalable user authentication via **AWS Cognito**;
- Mobile or PWA version of the application for better accessibility;
- Visual dashboards to monitor nutritional and fitness progress;
- Personalization features based on user goals, history, and activity patterns.

With further development, the platform could serve as a foundation for real-world health or fitness applications, offering AI-powered assistance for food and activity tracking on a scalable architecture.

11 Appendices

11.1 App sections

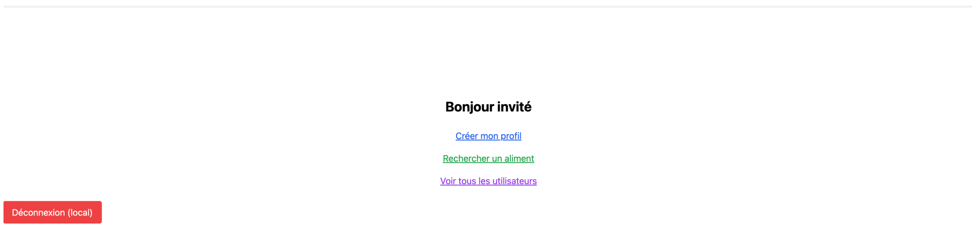


Figure 3: *App section 1 – Home page*

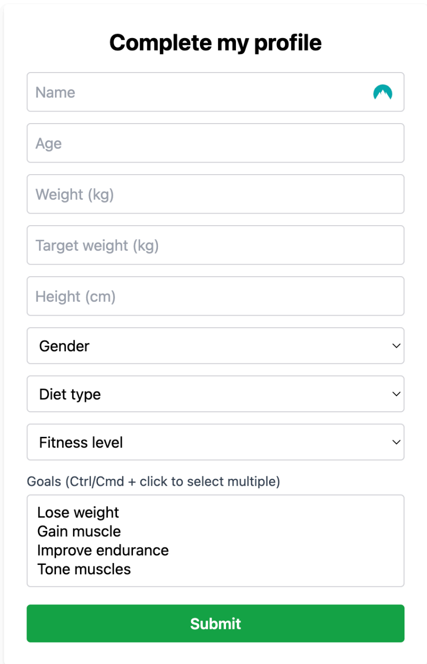


Figure 4: *App section 1 – User creation 1*

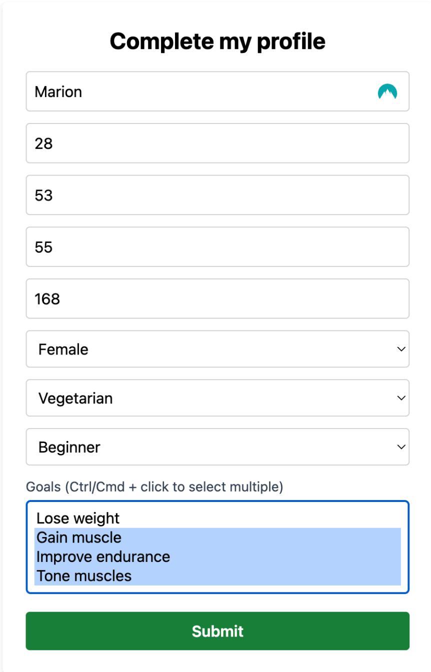


Figure 5: *App section 3 – User creation 2*

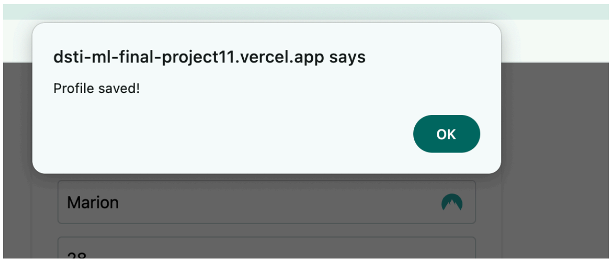


Figure 6: *App section 4 – Profile saved*

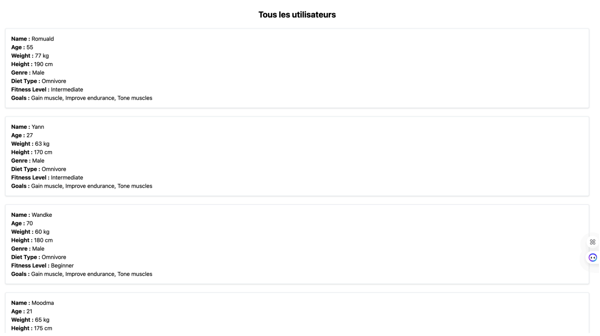


Figure 7: *App section 5 – Displaying all users 1*

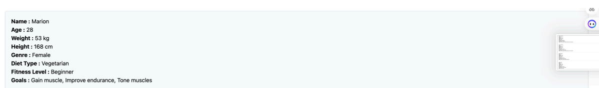


Figure 8: *App section 6 – Displaying all users 2*

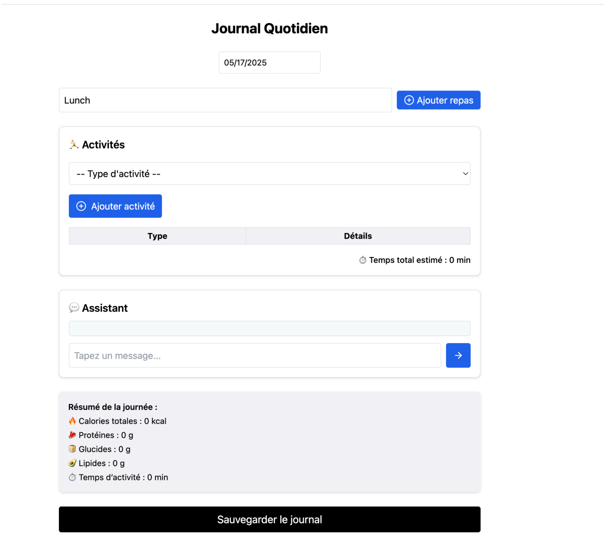


Figure 9: *App section 8 – Logging habits*

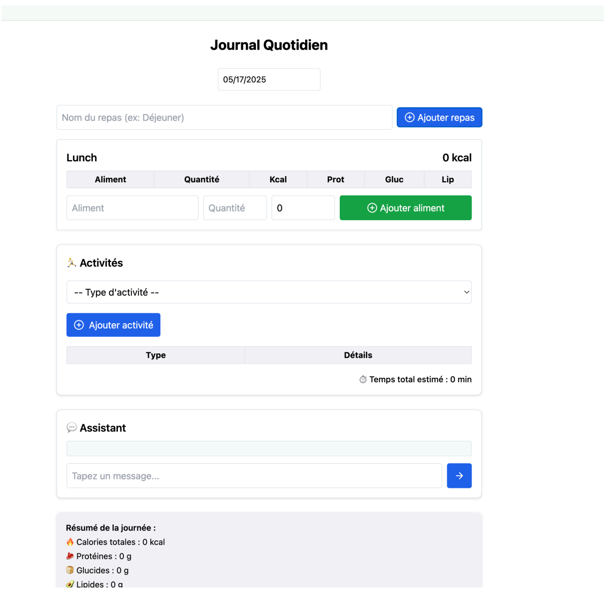


Figure 10: *App section 9 – Adding a meal*

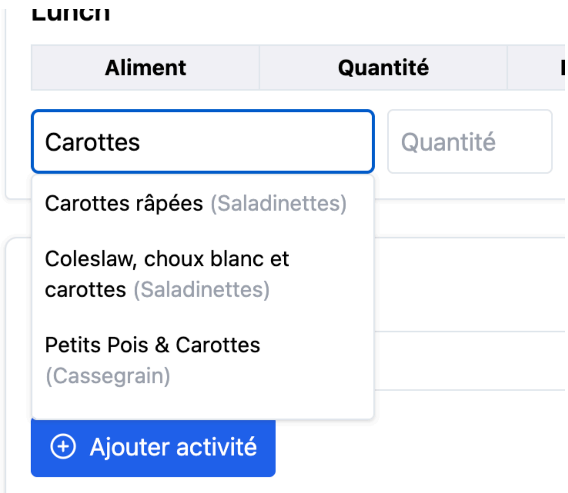


Figure 11: *App section 10 – Adding a food*

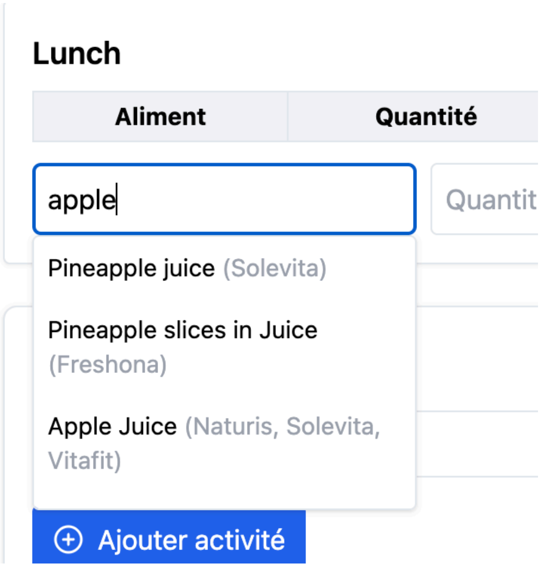


Figure 12: *App section 11 – Adding another food*

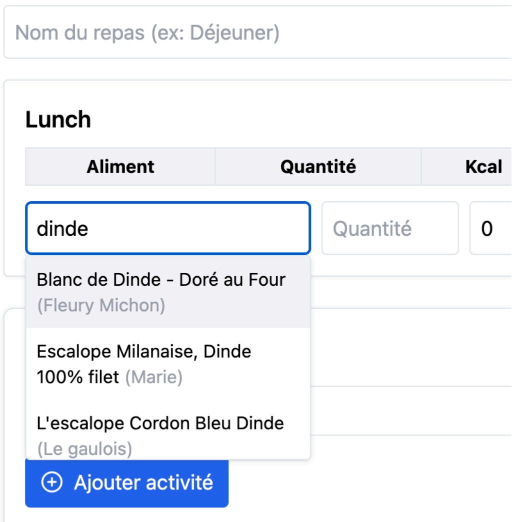


Figure 13: *App section 12 – Adding yet another food*

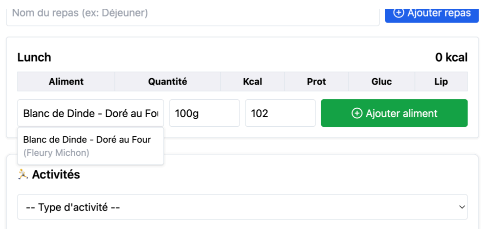


Figure 14: *App section 13 – Portion size*

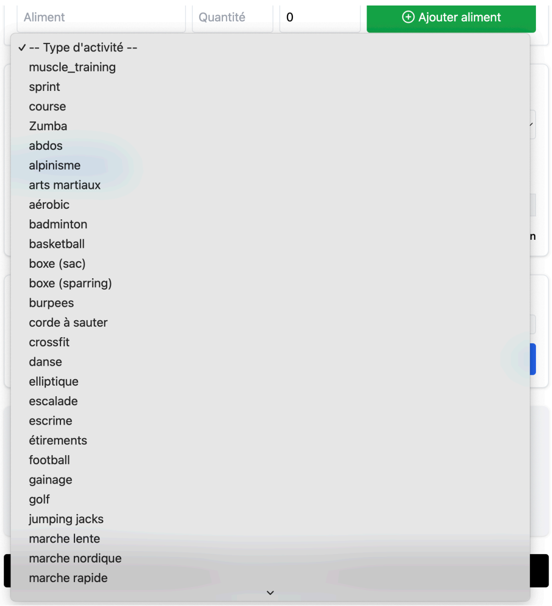


Figure 15: *App section 14 – Selecting activities*

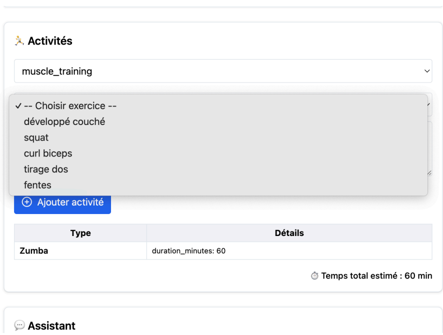


Figure 16: *App section 15 – Selecting a work out activity*

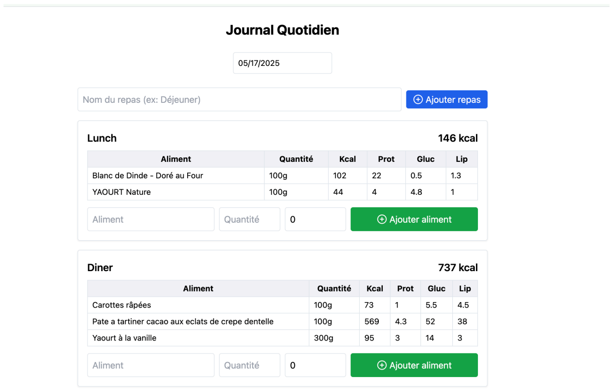


Figure 17: *App section 16 – Tables of meals*

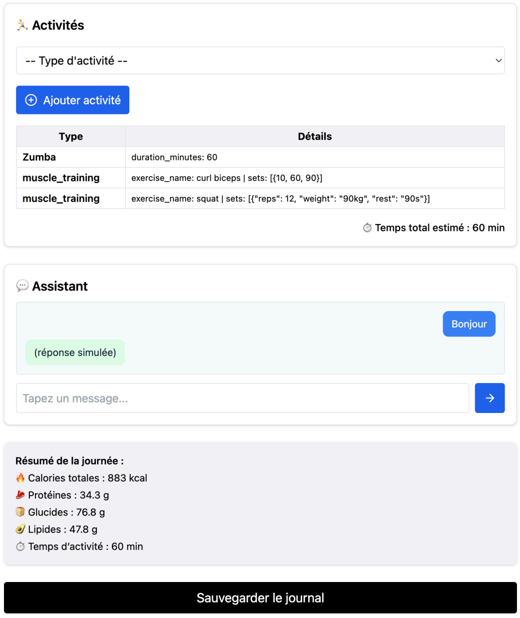


Figure 18: *App section 17 – Table of activities*

11.2 Exploration de la base PostgreSQL sur RDS

```
from sqlalchemy import create_engine, text
from dotenv import load_dotenv
import os

# Chemin vers .env.rds
project_root = os.path.abspath(os.path.join(os.path.dirname(__file__), "../../"))
env_path = os.path.join(project_root, "env_folder/.env.rds")
load_dotenv(dotenv_path=env_path)

# Variables 'denvironnement
DB_HOST = os.getenv("POSTGRES_HOST")
DB_PORT = os.getenv("POSTGRES_PORT")
```

```python
DB_USER = os.getenv("POSTGRES_USER")
DB_PASSWORD = os.getenv("POSTGRES_PASSWORD")
DB_NAME = os.getenv("POSTGRES_DB")

# Debug
print(" DEBUG des variables d'environnement :")
print("POSTGRES_HOST:", DB_HOST)
print("POSTGRES_PORT:", repr(DB_PORT))
print("POSTGRES_USER:", DB_USER)
print("POSTGRES_PASSWORD:", "***" if DB_PASSWORD else None)
print("POSTGRES_DB:", DB_NAME)

# Vérification
if not all([DB_HOST, DB_PORT, DB_USER, DB_PASSWORD, DB_NAME]):
    print(" Une ou plusieurs variables RDS sont manquantes.")
    exit(1)

# Engine
url = f"postgresql+psycopg2://{DB_USER}:{DB_PASSWORD}@{DB_HOST}:{DB_PORT}/{DB_NAME}"
engine = create_engine(url)

def list_databases():
    try:
        with engine.connect() as conn:
            result = conn.execute(text("SELECT datname FROM pg_database WHERE
                datistemplate = false;"))
            print(" Bases de données disponibles sur RDS :")
            for row in result:
                print(" -", row[0])
    except Exception as e:
        print(f" Erreur de connexion à RDS : {e}")

def list_tables():
    try:
        with engine.connect() as conn:
            result = conn.execute(text("SELECT table_name FROM information_schema.
                tables WHERE table_schema = 'public';"))
            print(f" Tables dans la base '{DB_NAME}' :")
            for row in result:
                print(" -", row[0])
    except Exception as e:
        print(f" Erreur lors de la récupération des tables : {e}")
```

```python
def show_users():
    try:
        with engine.connect() as conn:
            result = conn.execute(text("SELECT * FROM users;"))
            rows = result.fetchall()
            print(f"Contenu de la table 'users' ({len(rows)} lignes) :")

            # Récupère les noms de colonnes
            columns = result.keys()

            for row in rows:
                user_dict = dict(zip(columns, row))
                print(" -", user_dict)

    except Exception as e:
        print(f"Erreur lors de l'affichage des utilisateurs : {e}")

def show_goals():
    try:
        with engine.connect() as conn:
            result = conn.execute(text("SELECT * FROM goals;"))
            rows = result.fetchall()
            print(f" Contenu de la table 'goals' ({len(rows)} lignes) :")

            # Récupère les noms de colonnes
            columns = result.keys()

            for row in rows:
                user_dict = dict(zip(columns, row))
                print(" -", user_dict)

    except Exception as e:
        print(f"Erreur lors de l'affichage des objectifs : {e}")

def show_user_daily_logs():
    try:
        with engine.connect() as conn:
            result = conn.execute(text("SELECT * FROM user_daily_logs;"))
            rows = result.fetchall()
            print(f" Contenu de la table 'user_daily_logs' ({len(rows)} lignes) :")
```

```python
            # Récupère les noms de colonnes
            columns = result.keys()

            for row in rows:
                user_dict = dict(zip(columns, row))
                print(" -", user_dict)

    except Exception as e:
        print(f"Erreur lors de l'affichage des user daily goals : {e}")

def show_user_daily_logs_columns():
    try:
        with engine.connect() as conn:
            result = conn.execute(text("""
                SELECT column_name, data_type
                FROM information_schema.columns
                WHERE table_name = 'user_daily_logs';
            """))
            print(f" Colonnes de la table 'user_daily_logs' :")
            for row in result:
                print(f" - {row[0]} ({row[1]})")
    except Exception as e:
        print(f"Erreur lors de l'affichage des colonnes : {e}")

if __name__ == "__main__":
    list_databases()
    list_tables()
    show_users()
    show_goals()
    show_user_daily_logs()
    show_user_daily_logs_columns()
```

Listing 1: *Connection to PostgreSQL with SQLAlchemy*

Available DataBase

```
- postgres
- fitness_db
- rdsadmin
```

Tables in base 'fitness$_d b'$

- genders
- users
- diet_types
- fitness_levels
- user_goals
- goals
- activities
- meals
- muscle_trainings
- sprints
- meal_foods
- muscle_training_sets
- user_daily_logs

Content of the table `users`


```
{'user_id': 'ce7c6428-8c5c-4b41-9f70-074f8cf2d0f4', 'name': 'Romuald', 'age': 55, '
    height': 190.0, 'weight': 77.0, 'target_weight': 88.0, 'gender_id': 4, '
    diet_type_id': 5, 'fitness_level_id': 4}
{'user_id': '427d3f13-6319-4c2d-815a-86be56701e76', 'name': 'Yann', 'age': 27, '
    height': 170.0, 'weight': 63.0, 'target_weight': 71.0, 'gender_id': 4, '
    diet_type_id': 5, 'fitness_level_id': 4}
{'user_id': '835780b6-2fba-446e-8e34-345e6122e723', 'name': 'Wandke', 'age': 70, '
    height': 180.0, 'weight': 60.0, 'target_weight': 69.0, 'gender_id': 4, '
    diet_type_id': 5, 'fitness_level_id': 5}
{'user_id': '20b4fe4b-7126-4332-8df1-4a01eee4c4e3', 'name': 'Moodma', 'age': 21, '
    height': 175.0, 'weight': 65.0, 'target_weight': 80.0, 'gender_id': 4, '
    diet_type_id': 5, 'fitness_level_id': 4}
{'user_id': '11eb5c03-62e4-4eae-bcc5-429f942bcbfc', 'name': 'Martin', 'age': 26, '
    height': 180.0, 'weight': 70.0, 'target_weight': 75.0, 'gender_id': 4, '
    diet_type_id': 5, 'fitness_level_id': 4}
{'user_id': '9f11d508-5916-4afb-94a9-cb677c43291e', 'name': 'paul', 'age': 18, '
    height': 175.0, 'weight': 66.0, 'target_weight': 77.0, 'gender_id': 4, '
    diet_type_id': 5, 'fitness_level_id': 4}
{'user_id': 'df18e20a-af60-489f-b5dd-153b8e77ab4b', 'name': 'Marion', 'age': 28, '
    height': 168.0, 'weight': 53.0, 'target_weight': 55.0, 'gender_id': 5, '
    diet_type_id': 6, 'fitness_level_id': 5}
```

Content of the table `goals`

{'goal_id': 1, 'label': 'Lose weight'}
{'goal_id': 2, 'label': 'Gain muscle'}
{'goal_id': 3, 'label': 'Improve endurance'}
{'goal_id': 4, 'label': 'Tone muscles'}

Content of the table `user_daily_logs`

{'id': 1, 'user_id': '9f11d508-5916-4afb-94a9-cb677c43291e', 'log_date': datetime.date(2025, 5, 5), 'user_snapshot': {'name': 'paul', 'age': 18, 'gender': 'Male', 'weight': 66, 'height': 175, 'target_weight': 77, 'fitness_level': 'Intermediate', 'diet_type': 'Omnivore', 'goals': ['Gain muscle']}, 'log_data': {'meals': [], 'activities': [{'type': 'muscle_training', 'exercise_name': 'développé couché', 'sets': '3'}, {'type': 'sprint', 'distance': '33', 'time': '33', 'rest': '33'}], 'chat_history': []}, 'created_at': datetime.datetime(2025, 5, 6, 0, 3, 8, 727141), 'updated_at': datetime.datetime(2025, 5, 6, 0, 4, 31, 238364), 'weight': None, 'height': None}

{'id': 2, 'user_id': '9f11d508-5916-4afb-94a9-cb677c43291e', 'log_date': datetime.date(2025, 5, 6), 'user_snapshot': {'name': 'paul', 'age': 18, 'gender': 'Male', 'weight': 66, 'height': 175, 'target_weight': 77, 'fitness_level': 'Intermediate', 'diet_type': 'Omnivore', 'goals': ['Gain muscle']}, 'log_data': {'meals': [{'name': 'Lunch', 'items': [{'name': "Pom' potes Pomme Bio", 'quantity': '100g', 'calories': 0, 'protein': 0, 'fat': 0, 'carbs': 0}], 'total_nutrients': {'calories': 0, 'protein': 0, 'fat': 0, 'carbs': 0}}], 'activities': [{'type': 'muscle_training', 'exercise_name': '33', 'sets': '33'}], 'chat_history': [{'time': '02:45 AM', 'user_input': 'zefze', 'input_type': 'message', 'assistant_output': '(réponse simulée)'}, {'time': '02:45 AM', 'user_input': 'zef', 'input_type': 'message', 'assistant_output': '(réponse simulée)'}]}, 'created_at': datetime.datetime(2025, 5, 6, 0, 45, 54, 182085), 'updated_at': datetime.datetime(2025, 5, 6, 18, 0, 51, 186998), 'weight': None, 'height': None}

{'id': 3, 'user_id': 'df18e20a-af60-489f-b5dd-153b8e77ab4b', 'log_date': datetime.date(2025, 5, 17), 'user_snapshot': {'name': 'Marion', 'age': 28, 'gender': 'Female', 'weight': 53, 'height': 168, 'target_weight': 55, 'fitness_level': 'Beginner', 'diet_type': 'Vegetarian', 'goals': ['Gain muscle', 'Improve endurance', 'Tone muscles']}, 'log_data': {'meals': [{'name': 'Lunch', 'items': [{'name': 'Blanc de Dinde - Doré au Four', 'quantity': '100g', 'calories': 102, 'protein': 22, 'fat': 1.3, 'carbs': 0.5}, {'name': 'YAOURT Nature', 'quantity': '100g', 'calories': 44, 'protein': 4, 'fat': 1, 'carbs': 4.8}], 'total_nutrients

```
': {'calories': 146, 'protein': 26, 'fat': 2.3, 'carbs': 5.3}}, {'name': 'Diner
', 'items': [{'name': 'Carottes râpées', 'quantity': '100g', 'calories': 73, '
protein': 1, 'fat': 4.5, 'carbs': 5.5}, {'name': 'Pate a tartiner cacao aux
eclats de crepe dentelle', 'quantity': '100g', 'calories': 569, 'protein': 4.3,
'fat': 38, 'carbs': 52}, {'name': 'Yaourt à la vanille', 'quantity': '300g', '
calories': 95, 'protein': 3, 'fat': 3, 'carbs': 14}], 'total_nutrients': {'
calories': 737, 'protein': 8.3, 'fat': 45.5, 'carbs': 71.5}}], 'activities': [{'
type': 'Zumba', 'duration_minutes': '60'}, {'type': 'muscle_training', '
exercise_name': 'curl biceps', 'sets': '[{10, 60, 90}]'}, {'type': '
muscle_training', 'exercise_name': 'squat', 'sets': '[{"reps": 12, "weight": "90
kg", "rest": "90s"}]'}], 'chat_history': [{'time': '08:15 PM', 'user_input': '
Bonjour', 'input_type': 'message', 'assistant_output': '(réponse simulée)'}]}, '
created_at': datetime.datetime(2025, 5, 17, 18, 16, 17, 961911), 'updated_at':
datetime.datetime(2025, 5, 17, 18, 16, 17, 961915), 'weight': None, 'height':
None}
```


Colonnes de la table 'user$_{d}aily_{l}ogs'$

- height (double precision)
- id (integer)
- log_date (date)
- user_snapshot (json)
- log_data (json)
- created_at (timestamp)
- updated_at (timestamp)
- weight (double precision)
- user_id (varchar)

11.3 Fine-tuning dataset samples

Example 1 :

```
{
  "messages": [
    {
      "role": "user",
      "content": "J'ai mangé un Skyr Mangue passion Danone."
    },
    {
      "role": "assistant",
```

```
      "content": {
        "assistant_output": "Très bien, Skyr Mangue passion ajouté au journal.",
        "commands": [
          {
            "action": "query_s3",
            "target": "Skyr Mangue passion"
          },
          {
            "action": "fill_meal",
            "meal": "Breakfast",
            "item": {
              "name": "Skyr Mangue passion",
              "quantity": "1 pot (140g)",
              "calories": 100,
              "protein": 11,
              "fat": 0.3,
              "carbs": 10
            }
          }
        ]
      }
    }
  ]
}
```

Example 2 :

```
{
  "messages": [
    {
      "role": "user",
      "content": "escalade ce matin avec un ami"
    },
    {
      "role": "assistant",
      "content": {
        "assistant_output": "Génial ! Tu peux me préciser combien de temps vous avez grimpé ?",
        "commands": [
          {
            "action": "ask_user",
```

```
          "message": "Combien de minutes pour l'escalade ?"
        }
      ]
    }
  },
  {
    "role": "user",
    "content": "45min"
  },
  {
    "role": "assistant",
    "content": {
      "assistant_output": "Parfait, 45 minutes d'escalade ajoutées à ton journal.",
      "commands": [
        {
          "action": "fill_activity",
          "activity": {
            "type": "escalade",
            "duration_minutes": 45,
            "MET": 9.5
          }
        }
      ]
    }
  }
 ]
}
```

Example 3 :

```
{
 "messages": [
  {
    "role": "user",
    "content": "Goûter assez léger aujourd'hui, juste une boisson sucrée et un biscuit."
  },
  {
    "role": "assistant",
    "content": {
      "assistant_output": "Tu peux me dire quelle boisson et quel biscuit ? Marque ou type si possi
```

```
    "commands": [
      {
        "action": "ask_user",
        "message": "Quel type de boisson sucrée ? (jus, sirop, soda...)"
      },
      {
        "action": "ask_user",
        "message": "Quel type de biscuit ?"
      }
    ]
  }
},
{
  "role": "user",
  "content": "Un verre de sirop de grenadine Carrefour et un petit beurre U."
},
{
  "role": "assistant",
  "content": {
    "assistant_output": "C’est bien noté pour le snack.",
    "commands": [
      {
        "action": "fill_meal",
        "meal": "Snack",
        "item": {
          "name": "Sirop de grenadine dilué - Carrefour",
          "quantity": "1 verre (250ml)",
          "calories": 80,
          "protein": 0,
          "fat": 0,
          "carbs": 20
        }
      },
      {
        "action": "fill_meal",
        "meal": "Snack",
        "item": {
          "name": "Petit beurre - U",
          "quantity": "1 biscuit (12g)",
```

```
          "calories": 55,
          "protein": 0.6,
          "fat": 2.2,
          "carbs": 8.5
        }
      }
    ]
  }
  }
 ]
}
```

But stored like that, more than 700 examples in the same json file :

Here, just one example.

```
{"messages": [{"role": "user", "content": "Ce midi j’ai mangé un Yaos Le Yaourt à la Grecque Stracci
```

11.4 Key code snippets

11.5 Project Structure

```
.github/
  workflows/
    deploy.yml (1.1 Ko)
back_end/
  api/
    dependencies/
      __init__.py (0 o)
      auth.py (3.0 Ko)
    routes/
      __init__.py (0 o)
      activity_schema.json (4.9 Ko)
      daily_log.py (2.6 Ko)
      user.py (3.2 Ko)
    __init__.py (0 o)
  config/
    mixtral-finetune.yaml (1.1 Ko)
    settings.py (1.2 Ko)
  data_pipeline/
    dags/, scripts/, utils/...
```

```
    database/
        models/, queries/, repository/, connect.py...
    models/
        nlp/, regression/, vision/, model_training.py...
    services/
        product_searcher.py, user_service.py
    main.py, show_tables.py
checkpoints/
    mixtral-finetuned/
        checkpoint-603/ (LoRA model + tokenizer)
data_train/
    fine_tune_dataset_clean.jsonl, clean.py...
docker/
    Dockerfile.backend, Dockerfile.frontend
env_folder/
    .env.* files (Cognito, PostgreSQL, S3)
front_end/
    react_app/
        react_app_vite/
            src/, pages/, components/, vite.config.ts
            public/, dist/
infrastructure/
    aws/
        s3/, s3_manager.py, config.py
notebooks/
    DL_fitness_coach.ipynb
tests/
    back_end/, front_end/, Infrastructures/
.gitignore, docker-compose.yml, README.md, Makefile
```