



Milestone 5:

Model Productionization

CSC 5382 - SPRING 2025

By:

Khadija Salih Alj

Supervised by: Dr. Asmaa Mourhir

Introduction

This milestone focuses on the productionization of the Laddaty Ingredient Substitution System. The primary objective was to transition from a proof-of-concept into a robust, scalable, and maintainable machine learning system that can be deployed in a real-world mobile application. This involved implementing a complete backend capable of responding to user queries via API, integrating intelligent inference via Large Language Models (LLMs), managing data access with low-latency caching, ensuring reliable CI/CD deployment, and aligning all of this within a modular architecture based on the Model-View-Controller (MVC) pattern. The result is a system that delivers AI-driven substitutions to mobile users on demand, while remaining extensible and cloud-deployable.

I. ML System Architecture

The architecture of the ingredient substitution system is grounded in the traditional MVC paradigm but enhanced to incorporate intelligent machine learning components. It consists of four primary layers: the frontend (View), the backend API controller (Controller), the data layer and model logic (Model), and an external services layer which handles LLM inference and deployment (Figure 1).

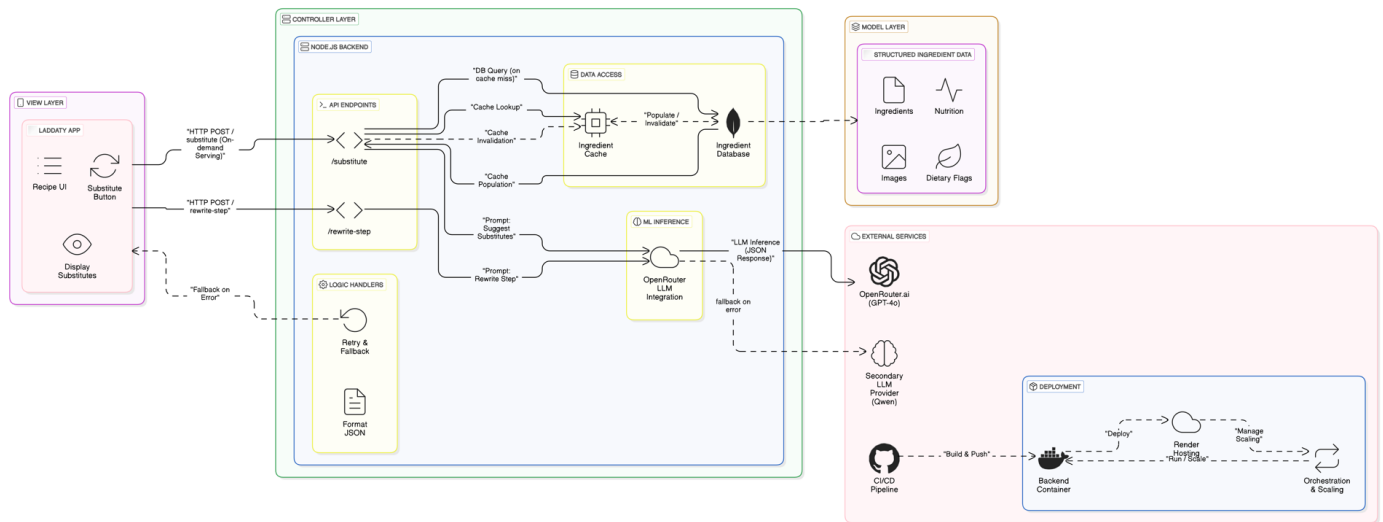


Figure 1: Ingredient substitution MVC-enhanced ML system architecture.

The **frontend layer** is implemented using React Native, serving as the user interface through which individuals can interact with the system. When the user wants to substitute an ingredient, a “Substitute” button is available next to each one in all recipes. This simple click triggers and whole logic of substitution, where -based on the context of the recipe- 10 substitutes are generated and once the user picks the preferred one from the returned list, another request is sent to rewrite the steps of the recipe. All interactions from the frontend are facilitated via HTTP POST requests, and responses are rendered with visual feedback, nutritional metadata, and enhanced interaction support.

At the core of the **controller layer** is a Node.js backend; it exposes two primary endpoints: `/substitute` and `/rewrite-step`. These endpoints are responsible for coordinating the end-to-end logic of the substitution pipeline. They receive requests from the frontend, look whether that request exists in the cache, and if not dynamically query a large language model through OpenRouter’s API. The controller layer also manages response handling, fallback mechanisms, and transformation of LLM outputs into valid structured responses.

The **model layer** consists of both structured storage and intelligent data preparation. MongoDB serves as the main database, where structured ingredient data -including names, nutritional information, and associated images- is persisted. To improve latency and reduce repetitive reads, an in-memory cache is implemented and preloaded on server startup. This cache includes a list of normalized ingredient names as well as enriched ingredient metadata, allowing fast fuzzy matches and substitutions without hitting the database on every request.

The **machine learning component** is integrated via OpenRouter’s hosted GPT-4o model. This LLM is queried in two contexts: (1) to generate up to 10 candidate substitutes for a given ingredient in the context of a recipe, each with associated ratio values, and (2) to rewrite a specific recipe step so that it matches the substituted ingredient’s action requirements (e.g., replacing “melt butter” with “whisk yogurt”). These LLM queries are designed with robust prompt engineering, strict JSON formatting enforcement, and retry mechanisms to recover from non-parsable responses.

II. Application Development

1) *Serving Mode Implementation*

The role-aware ingredient substitution system is designed to operate in a **real-time, on-demand serving mode**, particularly geared toward machine-to-machine interaction. Every ingredient substitution request is initiated from the React Native frontend as a direct API call to the backend. Unlike batch-serving pipelines or offline recommendations, this system is reactive and inference-based. When the user selects an ingredient to replace, or requests a rewritten cooking instruction, the backend performs all processing synchronously -including fuzzy matching, database lookups, and LLM inference- before responding with actionable outputs. This choice of serving mode supports the mobile-first, user-interactive nature of the application, and allows the AI model to generate context-aware suggestions tailored to the specific recipe in real time.

The architecture also supports fallback behavior to ensure robust user experience even in case of partial system failures. For example, if the LLM response is malformed or exceeds parsing limits, the system will either retry the inference request (3 times in case of malformed JSON), try another

LLM, or gracefully fallback to the original instruction without breaking the application flow. This fault tolerance is essential for production deployment where internet latency, LLM service availability, or token budget exhaustion may cause errors.

2) Model Service Development

The core logic of the ingredient substitution and step adaptation pipeline is encapsulated within the backend's service layer, demonstrating well-structured model service development. One of the main features implemented is the ingredient substitution flow, which queries the GPT-4o LLM using a custom-designed prompt. This prompt asks for JSON-formatted substitutions, each including the name of the substitute and a substitution ratio expressed as a float (e.g., "0.75") (Figure 2).

```
async function generateSubstitute(ingredient, recipe) {
  const prompt =
    `Suggest the best 10 substitutes for "${ingredient}" in this recipe "${recipe}".
    For each substitute, provide:
    1. Ensure that the ratio provided can logically be applied **in both directions**.
    Provide the response in strict JSON format with:
    {
      "substitutes": [
        {"name": "Substitute Name", "ratio": "Substitution ratio as a decimal number"}
      ]
    };`;
  const maxRetries = 3;
  let retryCount = 0;

  while (retryCount < maxRetries) {
    try {
      const response = await axios.post(
        "https://openrouter.ai/api/v1/chat/completions",
        {
          model: "openai/chatgpt-4o-latest",
          messages: [{ role: "user", content: prompt }],
          temperature: 0.1
        },
      );
    } catch {
      retryCount++;
    }
  }
}
```

Figure 2: Snippet of code showing the used prompt, model, and temperature.

Once the response is received, the service performs two key operations: it parses and cleans the raw JSON (with a secondary manual parsing attempt if automatic parsing fails), and it performs a round of fuzzy matching to validate that each proposed substitute corresponds to an ingredient already known in the system. Nutrition and image metadata is then re-attached to each valid substitute, making the output user-friendly and informative.

In the second core service -step rewriting- the model is prompted with the original recipe step, the name of the original ingredient, and the new substitute. The prompt instructs the model to rewrite the step in natural language, avoiding mention of the original ingredient and aligning the action verb with the substituted item (e.g., switching “melt” to “stir” for a non-solid substitute). The system enforces JSON output with fields title and description, ensuring the rewritten step integrates seamlessly into the app’s step-by-step cooking flow.

```

async function updateStepForSubstitution(originalStep, originalIng, substituteIng) {
  const prompt = `
    You are rewriting a cooking instruction step for a recipe.

    Original step:
    "${originalStep}"

    The ingredient "${originalIng}" has been replaced with "${substituteIng}".

    Your task:
    - Rewrite the step so it makes sense with the new ingredient.
    - Make sure the action (like melt, chop, blend...) suits the substitute.
    - Keep it short and natural.
    - Do NOT mention "${originalIng}" at all.

    Return ONLY a JSON object like this:
    {
      "title": "Short step title (action-based)",
      "description": "Rewritten step using the substitute ingredient"
    }
    Make sure it's valid JSON. No explanation, no formatting, no notes.
  `;

  try {
    const response = await axios.post(
      "https://openrouter.ai/api/v1/chat/completions",
      {
        model: "openai/chatgpt-4o-latest",
        messages: [{ role: "user", content: prompt }],
      }
    );
  }
}

```

Figure 3: Prompt to rewrite the steps.

Both services include retry loops, timeout handling, and logging hooks for debugging and observability. The architecture decouples the AI logic from routing logic, adhering to best practices in service modularity and clean separation of concerns.

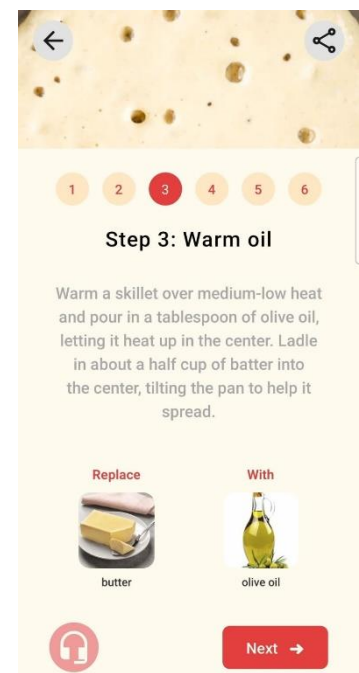
3) Frontend Client Development

On the frontend, the React Native application has been extended to integrate seamlessly with the newly developed substitution backend. When a user navigates to a recipe and selects an ingredient, an HTTP POST request is made to the /substitute endpoint, along with the full recipe context to guide the model's reasoning. Once a response is received, the application renders a list of up to 10 substitutes, each having the name, image, nutrition information, and the computed substitution ratio.



Additionally, when an ingredient is replaced, the app triggers a second API request to /rewrite-step, which returns an updated recipe step. This step is then dynamically injected into the recipe's step list, ensuring the instructions shown to the user reflect the substitutions they have made. These interactions are wrapped in user feedback mechanisms, including loading spinners, error messages, and fallbacks.

The frontend logic is written in a modular and extensible manner, allowing new APIs or response formats to be



integrated with minimal changes. By decoupling API handlers from UI rendering components, the app remains maintainable and testable as the project evolves 😊.

III. Integration and Deployment

1) Packaging and Containerization

```
# Use Node.js image
FROM node:18-alpine

# Create app directory
WORKDIR /app

# Install dependencies
COPY package*.json ./
RUN npm ci --omit=dev

# Copy the rest of code
COPY . .

# Expose the port
EXPOSE 5000

# Start your server
CMD ["node", "app.js"]
```

The backend microservice powering the ingredient substitution system is packaged using Docker to ensure **portability**, **reproducibility**, and **seamless deployment** across development, staging, and production environments. The Dockerfile is custom-written and optimized for the Node.js environment, based on the official lightweight node:18-alpine image (Figure 4).

The containerization workflow includes the following key steps:

- It establishes the working directory using WORKDIR /app.
- It leverages Docker's layered caching strategy by copying package.json and package-lock.json before the rest of the source code, allowing dependency installation to be cached unless the manifest changes.
- Dependencies are installed via npm ci to ensure deterministic builds and to skip devDependencies in production.
- All source files are copied into the container, and the server is started using node app.js.
- Port 5000 is exposed to match the Express API service expected by the frontend.

```
F:\Laddaty\Code\backend_dock>docker build -t laddaty-backend .
[+] Building 120.8s (10/10) FINISHED                                docker:desktop-linux
=> [internal] load build definition from Dockerfile                  0.3s
=> => transferring dockerfile: 313B                                  0.0s
=> [internal] load metadata for docker.io/library/node:18-alpine    5.7s
=> [internal] load .dockerignore                                     0.2s
=> => transferring context: 2B                                         0.0s
=> [1/5] FROM docker.io/library/node:18-alpine@sha256:8d6421d663b4c28fd3ebc498332f249011d118945588d0a35cb9bc4b8 90.7s
=> => resolve docker.io/library/node:18-alpine@sha256:8d6421d663b4c28fd3ebc498332f249011d118945588d0a35cb9bc4b8c 0.2s
=> => sha256:8d6421d663b4c28fd3ebc498332f249011d118945588d0a35cb9bc4b8ca09d9e 7.67kB / 7.67kB 0.0s
=> => sha256:929b04d7c782f04f615cf785488fed452b6569f87c73ff666ad53a7554f0006 1.72kB / 1.72kB 0.0s
=> => sha256:ee77c6cd7c1886ecc802ad6cedef3a8ec1ea27d1fb96162bf03dd3710839b8da 6.18kB / 6.18kB 0.0s
=> => sha256:f18232174bc91741fdf3da96d85011092101a032a93a388b79e99e69c2d5c870 3.64MB / 3.64MB 5.6s
=> => sha256:dd71dde834b5c203d162902e6b8994cb2309ae049a0eabc4feea161b2b5a3d0e 40.01MB / 40.01MB 85.2s
=> => sha256:1e5a4c89cee5c0826c540ab06d4b6b491c96eda01837f430bd47f0d26702d6e3 1.26MB / 1.26MB 8.0s
=> => extracting sha256:f18232174bc91741fdf3da96d85011092101a032a93a388b79e99e69c2d5c870 0.5s
=> => sha256:25ff2da83641908f65c3a74d80409d6b1b62ccfaab220b9ea70b80df5a2e0549 446B / 446B 8.2s
=> => extracting sha256:dd71dde834b5c203d162902e6b8994cb2309ae049a0eabc4feea161b2b5a3d0e 4.3s
=> => extracting sha256:1e5a4c89cee5c0826c540ab06d4b6b491c96eda01837f430bd47f0d26702d6e3 0.2s
=> => extracting sha256:25ff2da83641908f65c3a74d80409d6b1b62ccfaab220b9ea70b80df5a2e0549 0.0s
=> [internal] load build context                                     3.8s
=> => transferring context: 19.64MB                                    3.7s
=> [2/5] WORKDIR /app                                              0.3s
=> [3/5] COPY package*.json ./                                     0.1s
=> [4/5] RUN npm ci --omit=dev                                     20.4s
=> [5/5] COPY . .                                                  1.6s
=> exporting to image                                              1.3s
=> => exporting layers                                              1.2s
=> => writing image sha256:29470a409002b006b32bf7527bab3c0ab2d8e45ac225b4be930d67eb26296337 0.0s
=> => naming to docker.io/library/laddaty-backend                  0.0s

F:\Laddaty\Code\backend_dock>docker run -p 5000:5000 laddaty-backend
Server running on http://localhost:5000
Database Connected
```

Figure 5: Docker build and execution output for the ingredient substitution service.

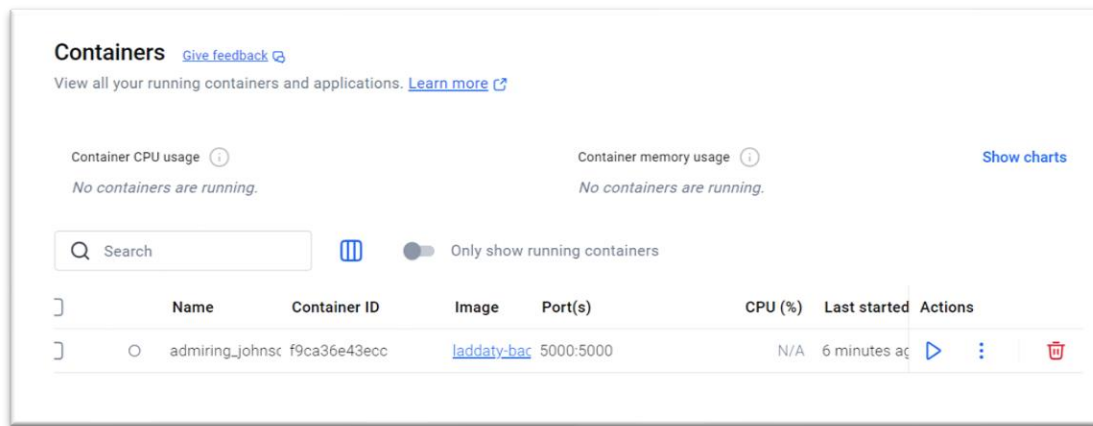


Figure 6: Docker Desktop UI showing the backend container.

The application supports runtime configuration through environment variables such as `API_KEY` and `MONGODB_URI`, which are passed securely by the deployment platform (Render).

2) CI/CD Pipeline Integration

To automate deployment and maintain reproducibility, a CI/CD pipeline was implemented using **GitHub Actions**. This workflow is triggered automatically upon each push to the main branch. The pipeline performs the following steps:

1. **Checkout** the latest version of the source code from the repository.
2. **Install dependencies.**
3. **Deploy** to Render via a webhook.

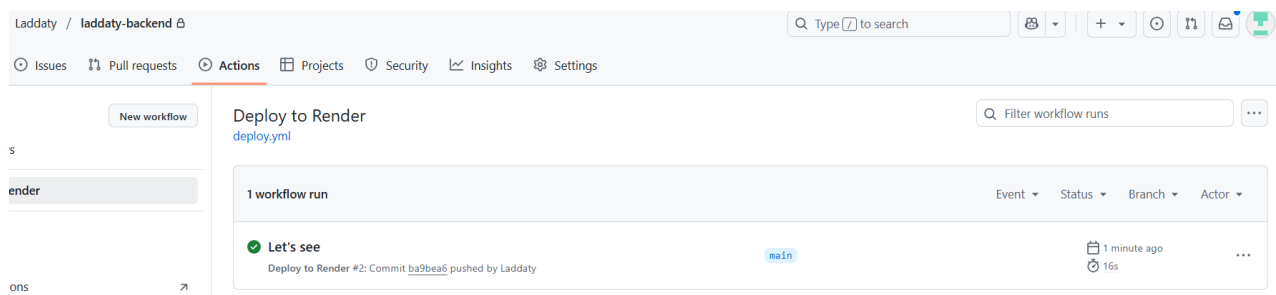


Figure 7: CI/ CD working perfectly.

This setup ensures that every change pushed to the repository can be automatically tested, built, and deployed in a consistent and repeatable manner. The pipeline can be extended in the future to include integration tests.

3) *Hosting the Application*

The application is hosted on **Render**, a modern cloud platform that offers both native Node.js deployment and Docker-based infrastructure. In this project, Render's native Node.js environment is used for deployment, which simplifies configuration by relying on `package.json` and `start` commands. Once deployed, the service runs in a managed environment that automatically exposes the specified port (5000) and serves the Node.js API behind an HTTPS-enabled endpoint.

Custom Domains

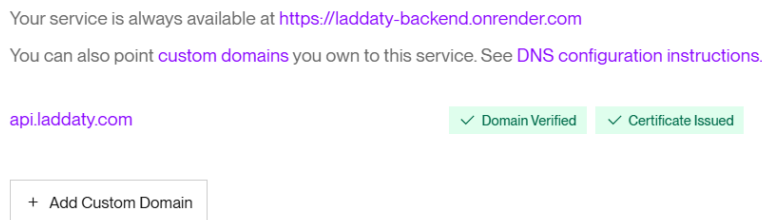


Figure 8: Screenshot showing the custom domain was successfully linked.

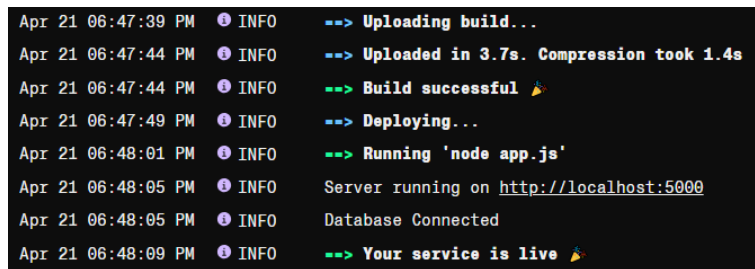


Figure 9: Successful deployment in Render.

The Render service is configured to:

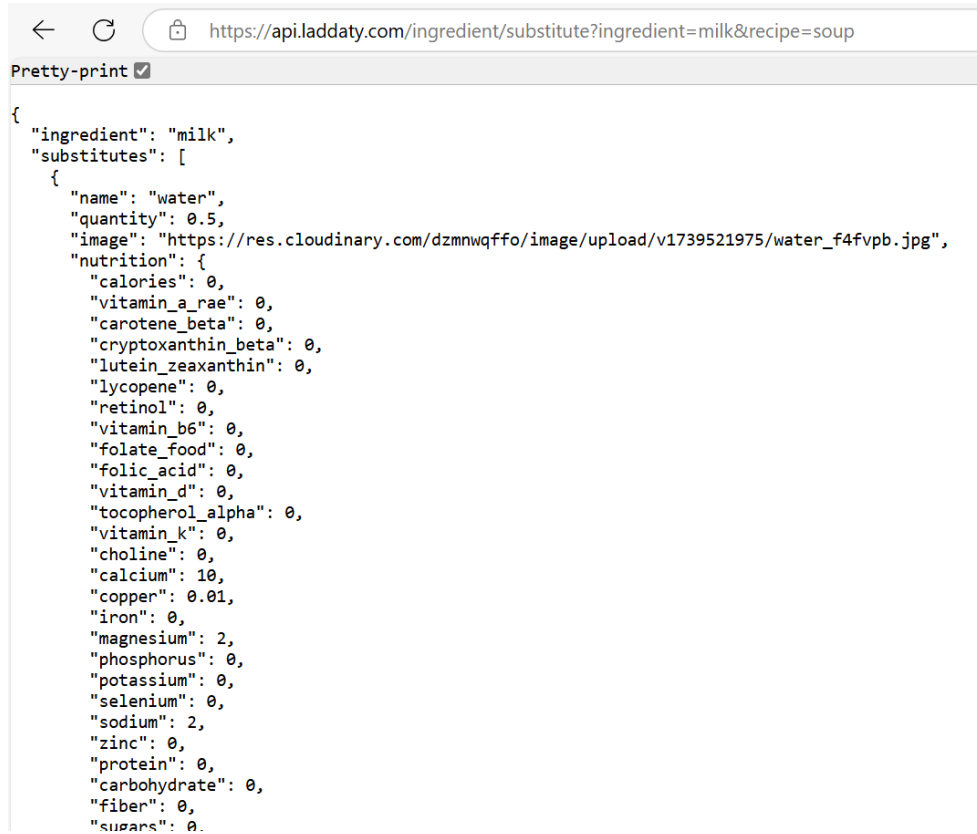
- Use **auto-deploy on push to GitHub** (main branch).
- Inject necessary **environment variables** like API keys and database URIs.
- Restart automatically on crash or when the deployment fails.
- Support public HTTPS access for mobile API communication.

SERVICE NAME 1	STATUS	RUNTIME	REGION	DEPLOYED ↓
🌐 laddaty-backend	✓ Deployed	Node	Frankfurt	<1m ...

Figure 10: Service deployed in Render.

The deployed application has average response times within acceptable bounds for mobile interactions (typically under 2 seconds for full substitute resolution, including LLM inference). The

deployment includes HTTPS termination and secure CORS policies and is easily linked to *laddaty.com* custom domain name for user-facing deployments.



```
← ↻ 🔒 https://api.laddaty.com/ingredient/substitute?ingredient=milk&recipe=soup
Pretty-print ☒
{
  "ingredient": "milk",
  "substitutes": [
    {
      "name": "water",
      "quantity": 0.5,
      "image": "https://res.cloudinary.com/dzmnwqffo/image/upload/v1739521975/water_f4fvpb.jpg",
      "nutrition": {
        "calories": 0,
        "vitamin_a_rae": 0,
        "carotene_beta": 0,
        "cryptoxanthin_beta": 0,
        "lutein_zeaxanthin": 0,
        "lycopene": 0,
        "retinol": 0,
        "vitamin_b6": 0,
        "folate_food": 0,
        "folic_acid": 0,
        "vitamin_d": 0,
        "tocopherol_alpha": 0,
        "vitamin_k": 0,
        "choline": 0,
        "calcium": 10,
        "copper": 0.01,
        "iron": 0,
        "magnesium": 2,
        "phosphorus": 0,
        "potassium": 0,
        "selenium": 0,
        "sodium": 2,
        "zinc": 0,
        "protein": 0,
        "carbohydrate": 0,
        "fiber": 0,
        "sugars": 0
      }
    }
  ]
}
```

Figure 11: The service working from Laddaty custom domain 😊

IV. Model Serving

The model serving runtime for the ingredient substitution system is architected as a **cloud-native, on-demand inference setup using openrouter.ai**, a hosted model serving platform that abstracts infrastructure management while exposing a clean API interface for multiple LLMs. In this project, I use the **GPT-4o** model as the primary reasoning engine.

Rather than deploying and maintaining a self-hosted model server (which would require GPU provisioning, autoscaling, logging, and maintenance), I opted to leverage **OpenRouter's external runtime**. This decision aligns with modern **Model-as-a-Service (MaaS)** paradigms that prioritize scalability, reliability, and focus on application logic rather than infrastructure burden. The hosted model is invoked using HTTPS endpoints, with custom-designed prompts and temperature control (temperature = 0.1) to ensure both consistency and diversity in generated outputs.

This runtime setup is used in two key functionalities, substitution generation and step(s) rewriting. In both cases, the OpenRouter serving runtime acts as a **stateless inference layer**, allowing the backend to remain thin, fast, and decoupled from the complexity of model training or hosting.

Additionally, this architecture supports future model switching (e.g., Llama, Claude, Mistral, or Qwen) with minimal changes to the codebase, making the system more adaptable to shifts in performance benchmarks, costs, or licensing requirements.

Conclusion

This milestone marked a critical transition in the role-aware ingredients substitution project, transforming a functional prototype into a deployable, production-grade machine learning system. The design is anchored in a modular, ML-enhanced **MVC architecture**, wherein the backend controller orchestrates real-time logic for fuzzy matching, model inference, and structured response formatting. The React Native frontend integrates tightly with the backend, providing seamless interactivity to end users and delivering ingredient substitutions and adapted recipe steps in real time.

The deployment stack leverages **Docker** for portable containerization and **Render** for scalable cloud hosting. A **GitHub Actions** pipeline ensures continuous integration and delivery, guaranteeing that updates can be safely deployed without interrupting service. The backend is hosted via containerized infrastructure, with secure environment variables, robust error handling, and health-monitoring readiness.

The choice of **OpenRouter** as the model serving runtime allowed me to offload computational overhead while retaining flexibility in model selection and prompt design. By combining prompt engineering with structured response parsing, I was able to build a resilient system that can reliably serve machine learning predictions to mobile users at scale.

In summary, this milestone showcases the practical implementation of MLOps principles within an AI-driven application. It demonstrates how inference-time reasoning, container-based deployment, and intelligent caching can be harmonized into a performant and maintainable system. Laddaty's backend is now not only production-ready but also extensible -capable of supporting future features such as personalized substitution rules, user feedback loops, and multilingual support...