

Recipe Suggester Tool

System Specification Document

Baratto Luca, Moro Tommaso, Savorgnan Enrico

Contents

1 Problem Definition	2
1.1 Business Problem	2
1.2 Definitions, acronyms, and abbreviations	2
2 Product Description	3
2.1 User-Product Pipeline	3
2.2 General constraints	3
2.3 Assumptions and dependencies	5
3 Requirements	6
3.1 Functional Requirements	6
3.2 Non-functional Requirements	7
4 Project Architecture	9
4.1 High-level Architecture	9
4.2 ML Training & Validation	9
4.3 Deployment & Monitoring	11
5 Risk Analysis	12
5.1 Technical & Infrastructure Risks	12
5.2 Machine Learning & MLOps Risks	12
5.3 Privacy & Compliance Risks (GDPR)	13

1 Problem Definition

The following document is meant to convey the System Specifications for the “Recipe Suggester Tool”, an innovative product designed for helping people being healthier, thus happier, only by eating better the food they already have at home.

1.1 Business Problem

The product “Recipe Suggester Tool” provides real-time, step-by-step, and easy-to-cook recipes to the user. The recipes are based on the food recognized in pictures sent by the user himself. This will allow the user to make less friction in deciding what to cook, reducing food waste and increasing user satisfaction and health through balanced diets.

The product will be available as a web-app, accessible from any device with internet connection. The product will be based on Machine Learning and Large Language Models to provide state-of-the-art solutions to the user, and it will be GDPR-compliant to ensure user data privacy.

1.2 Definitions, acronyms, and abbreviations

1.2.1 Definitions

- *Platform, Webapp, Product, System:* indicate the product “Recipe Suggester Tool”.
- *User:* a person in need of advice for deciding what to cook and how. The user should have an internet connection and should be registered to the webapp for using the product.

1.2.2 Abbreviations

- *RST:* Recipe Suggester Tool, the name of the product
- *ML:* Machine Learning
- *LLM:* Large Language Model
- *GDPR:* General Data Protection Regulation

2 Product Description

2.1 User-Product Pipeline

A general overview on the pipeline of the product and on the interactions between user and product is presented.

1. A user accesses the webapp via a browser and logs in (or registers) using his/her credentials.
2. The user sends to the webapp a picture of food, e.g., sends a photo of a fridge or storeroom.
3. The picture is sent to a trained ML model running in the backend.
4. The different types of food in the picture are recognized by the ML model and stored into an aggregated file.
5. The file is sent to the webapp for a user approval. In case something is missing, or something is wrong, the user can modify the file.
6. The corrected file is sent to a properly initialized LLM, that will provide a recipe to the user.
7. The recipe is sent to the frontend and shown to the user.

A much more detailed version of the User-Product pipeline is described below in the UML Sequence Diagram.

2.2 General constraints

Some constraints are defined to ensure a sufficient quality of the service provided.

- The whole workflow will require at most 5 seconds + 5 seconds of LLM latency.
- The product should be robust to manage multiple (at least 5) concurrent accesses through the platform.
- The product should be robust to failures, e.g., the connection to the backend is not working. In such cases, a proper error message should be shown to the user.
- The percentage of accesses causing failures should be below 1%.
- Some data is possibly prone to corruption. The percentage should be below 1%.
- The webapp should be unavailable for any maintenance operations less than 2 hours per month.
- The size of the input image should be less than 5MB.

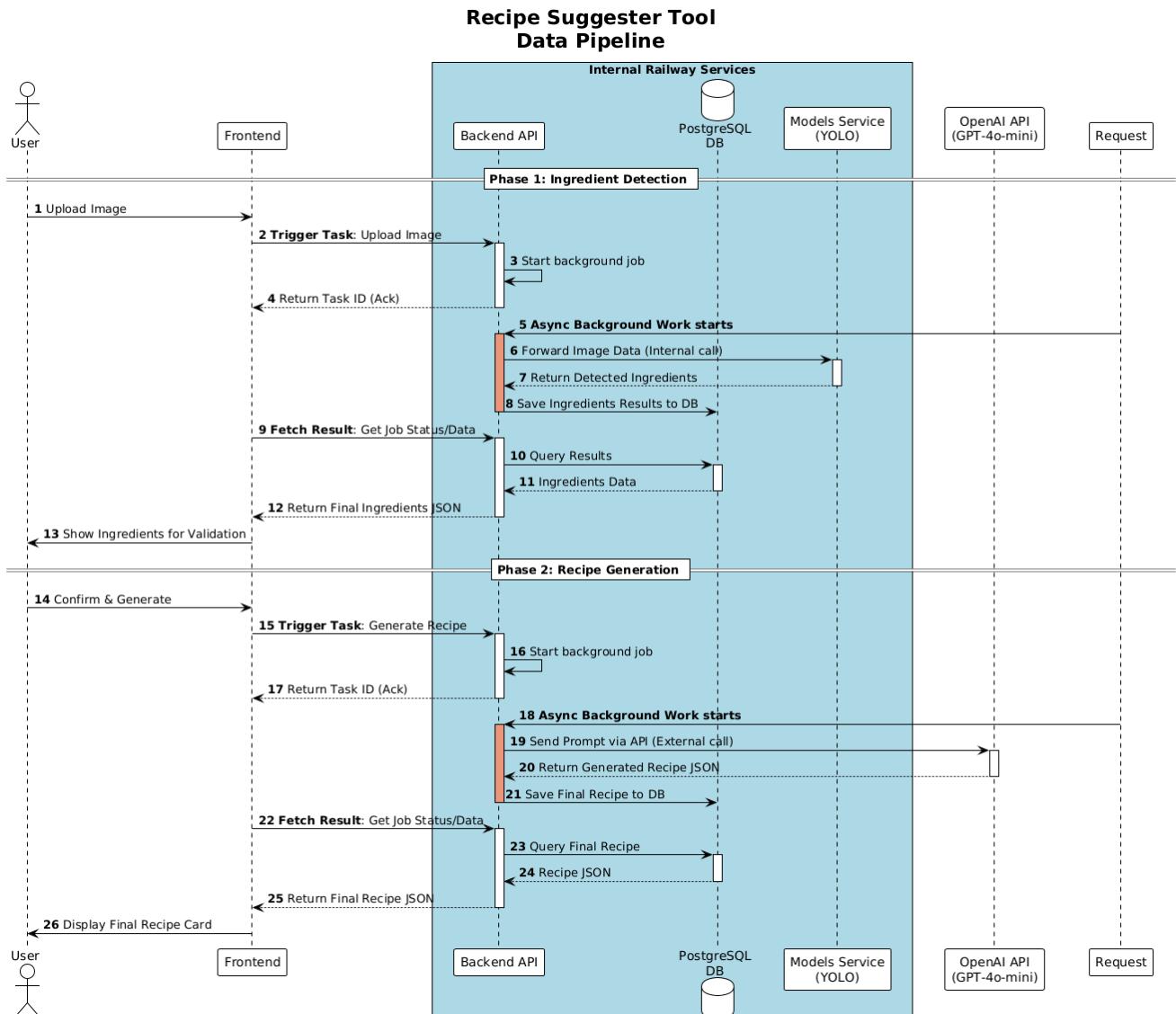


Figure 1: UML Sequence Diagram of the User-Product pipeline.

2.3 Assumptions and dependencies

For a proper functioning of the product, some assumptions and dependencies are defined as follows:

- The user has a device with internet connection and a browser.
- The user is registered to the webapp.
- The user is willing to share some data with the webapp, e.g., images of food.
- The backend is able to run the ML model and the LLM.
- The ML model is properly trained and able to recognize food in images.
- The LLM is properly initialized and able to provide recipes based on the food recognized.
- The webapp frontend is able to communicate with the backend and viceversa.

3 Requirements

The following section will distinguish between Functional Requirements (code F0x,) and Non-functional requirements (code N0x).

3.1 Functional Requirements

3.1.1 User account & Authentication

- F01.01: The system shall allow users to register and log in using an email and password via a secure form.
- F01.02: The system shall support Social Authentication via **Google OAuth2**, allowing users to access the platform without creating a specific password.
- F01.03: The system shall maintain a User Profile that includes preferences and dietary restrictions (e.g., allergies, intolerances) to customize the generated recipes.

3.1.2 Image Upload & Pre-processing

- F02.01: The system shall allow users to upload images of their fridge or pantry items via the frontend interface.
- F02.02: Supported image formats shall include .JPEG, .JPG, .PNG, and .BMP, consistent with the backend validation logic.
- F02.03: The system shall enforce a file size limit (e.g., 5MB) to prevent server overload.
- F02.04: The system shall store the uploaded image within a session-scoped job context to allow the Models Service to access and process the file.

3.1.3 Food Recognition & Segmentation

- F03.01: The system shall utilize the **YOLO11n** (Nano) object detection model to identify ingredients within the uploaded image.
- F03.02: The Models Service shall accept the image file via a HTTP request and return a list of detected ingredients with confidence scores.
- F03.03: The backend shall filter detections based on a confidence threshold (default > 0.25) to ensure relevance.
- F03.04: The system shall present the detected ingredients to the user for validation, allowing them to add missing items or remove incorrect detections before generating recipes.

3.1.4 Recipe Generation

- F04.01: The system shall generate a structured prompt containing: (1) the validated list of ingredients, (2) user dietary preferences (allergies), and (3) system constraints (language, output format).
- F04.02: The system shall utilize the **OpenAI API** (specifically the `gpt-4o-mini` model) in **asynchronous mode** to generate recipes without blocking the server.
- F04.03: The LLM output shall be strictly parsed into a structured JSON format containing: Title, Preparation Time, Cooking Time, Difficulty, Ingredients List, and Step-by-Step Procedure.

3.1.5 Job Orchestration & Monitoring

- F05.01: The system shall manage long-running tasks (recognition and generation) using an asynchronous Job queue pattern, providing status updates (`running`, `completed`, `failed`) to the frontend.
- F05.02: The system shall log the *metadata* of the request (`timestamp`, `number of ingredients detected`, `processing time`) for analytics, without storing the user's image for GDPR compliance.

3.2 Non-functional Requirements

3.2.1 Usability

- N01.01: The web application should be *mobile-first*, ensuring auto-layout from smartphones to desktop monitors.
- N01.02: Error messages displayed to the user shall be non-technical and actionable, e.g., a message “No food recognized. Please try again.” and a button “Try again” instead of only a message “404: Bad Request”.

3.2.2 Performance & Scalability

- N01.01: The Food Recognition inference shall complete within 5 seconds on a standard CPU environment (thanks to YOLO11n optimization).
- N01.02: The LLM recipe suggestion shall complete within 5 seconds on a standard network connection and under nominal load.
- N01.03: The Recipe Generation phase shall utilize Python `asyncio` to handle concurrent user requests without blocking the main event loop.

- N01.04: The system architecture shall be split into microservices (Frontend, Backend, Models) to allow independent scaling of the resource-heavy inference container.

3.2.3 Reliability & Availability

- N02.01: The system shall implement automatic retries and error handling for external API calls (e.g., OpenAI connectivity issues).
- N02.02: Service health shall be monitored via dedicated `/health` endpoints, allowing the orchestration platform (Railway) to restart unresponsive containers automatically.

3.2.4 Security & Privacy

- N03.01: All external traffic shall be encrypted via HTTPS.
- N03.02: Cross-Origin Resource Sharing (CORS) shall be strictly configured to allow requests only from the authorized Frontend domain in production.
- N03.03: API Keys (e.g., OpenAI, Google OAuth) shall never be hardcoded but injected via Environment Variables at runtime.
- N03.04: User passwords shall be hashed using a strong algorithm (e.g., bcrypt) before storage in the PostgreSQL database.

3.2.5 Data Requirements

- N06.01: The system shall maintain a persistent record for each registered user containing relevant attributes, included, but not limited at, `userId`, `name`, `surname`, `mail`, `password`.
- N06.02: The system shall never store passwords in plain text, but they must be encrypted by hashing before storage.

3.2.6 Maintainability & Model Lifecycle

- N07.01: The codebase shall be fully containerized using Docker, with multi-stage builds to minimize image size (using `python:3.11-slim` and `node:20-alpine`).
- N07.02: The backend code shall adhere to PEP 8 standards and be validated via CI pipelines using tools like Ruff.
- N07.03: Frontend code shall be type-safe, utilizing TypeScript for all components and API interactions.

4 Project Architecture

This section is meant to describe the overall architecture of the project, from training to deployment and monitoring; a special focus will be given on the interactions between the components and the ML lifecycle management.

4.1 High-level Architecture

The system follows a **Microservices Architecture**, composed of three distinct containerized services and a persistence layer. This decoupling allows for independent scaling of the ML inference engine (CPU/GPU intensive) and the core API (I/O intensive).

The four main components are:

1. **Frontend Service (Client Tier)**: A Single Page Application (SPA) built with **React** and **TypeScript**, served via **Nginx**. It handles user interactions, secure image uploads, and renders the recipe UI. It communicates exclusively with the Backend API[cite: 6].
2. **Backend Service (Orchestrator Tier)**: A **FastAPI** (Python) application acting as the API Gateway. It is responsible for:
 - Authentication (OAuth2/JWT) and User Session management.
 - Orchestrating the workflow: receiving images, sending them to the Models Service, and forwarding ingredients to the LLM.
 - Managing business logic and database transactions.
3. **Models Service (Inference Tier)**: A dedicated **FastAPI** microservice encapsulating the **YOLO11n** object detection model. It exposes an internal HTTP endpoint to accept image files and return detected ingredient labels with confidence scores. This service is isolated to ensure heavy inference loads do not block the main application thread.
4. **Data Tier**: A managed **PostgreSQL** database used to store persistent user data, session tokens, and recipe history.

4.2 ML Training & Validation

Regarding ML models, a fine-tuned version of Ultralytics YOLO11n is implemented.

Data for the fine-tuning of YOLO models has been collected from a variety of datasets available online and shared through the most famous dataset platforms Roboflow.

It has been chosen because it provided ready-to-use models, natively exportable in a YOLO-like format.

- Fridge Detector Dataset – (21042 train, 2150 validation, 923 test)

- Fridgify Dataset – (17691, 1427, 1104)
- Computer Vision Food Dataset – (8754, 796, 574)
- Food Item Detection Dataset – (10558, 3003, 1486)

The datasets have been merged and properly preprocessed in order to have a unique dataset with 99 different food classes.

Datasets shown a high unbalance in the number of the classes, resulting in some of them almost never represented. To fix this issue, we operated data augmentation by adding food from unrepresented classes to several other food images (“copy-past augmentation”).

The hyperparameter `data_augmentation` of `YOLO` class by Ultralytics was set to `True` as well to helping solve the issue.

The resulting datasets have been stored via Microsoft Azure cloud and versioned using Data Version Control protocol.

Fine-tuning has been performed using CUDA-enabled machines on Univeristy of Trieste cluster “Demetra”.

The model has been fine-tuned for 100 epochs.

Key metrics tracked include:

- **Box-wise Loss:** how well food boxes are identified in the space.
- **Classification Loss:** how well food types are classified.
- **mAP@0.5:** mean Average Precision at IoU threshold of 0.5, measuring overall detection accuracy.
- **mAP@0.5:0.95:** mean Average Precision averaged over multiple IoU thresholds from 0.5 to 0.95, providing a comprehensive accuracy measure.
- **Precision and recall.**

All the metrics and the model weights have been logged using pre-defined Ultralytics YOLO11n functionalities.

The final model weights have been saved in a `.pt` and in a `.onnx` format and versioned using `Git LFS`.

After fine-tuning, the model has been validated on a separate validation set to ensure generalization capabilities, and finally tested on a test set to evaluate final performance.

The final results are the following:

- **Box-wise Loss:** 0.57972
- **mAP@0.5:** 0.95037
- **mAP@0.5:0.95:** 0.80301
- **Precision:** 0.95825

- Recall: 0.93070

Before final deployment, the model had to pass a sequence of tests, implemented to ensure high-quality results. A better explanation of models tests is provided in Operational Governance & Versioning Document at Section 2.1.

The LLM for the recipe creation is instead called via API, without any further process. The model chosen is **GPT-4o-mini** from OpenAI, as it represents the most balanced choice between reliability, security, performances and expense. No further fine-tuning seemed to be required, based on the performances and the quality of the recipes provided.

4.3 Deployment & Monitoring

Regarding ML component, deployment is pretty easy, since only a `.pt` file is saved and versioned.

Considering the web-app frontend-backend components, each of them when deployed builds a Docker Image package.

This choice ensures scalability: for any further architecture changing, only a part of the webapp needs to be modified. A number of tests is implemented for ensures compatibility and full functionality of the product after each change of the code source. A detailed explanation on the tests is provided in the Project Proposal & Development Plan document, at Section 2.3.

Each component (Frontend, Backend, Models) is dockerized using proper ‘Dockerfile’s; the webapp is then pushed to GitHub Container Registry (GHCR) for versioning and easy deployment, so that it is easy to pull the latest version directly in a local machine.

The online deployment is directly operated thanks to **Railway** cloud provider.

Railway dashboard shows real-time information and logs about the WebApp, including CPU and Disk usages, number of access to the platform, requests latency and other.

This allows for a unified strategy of monitoring and performances’ checking, in order to ensure clear reliability and to effectively intervene in case of system crashes.

For a more detailed explanation on CI/CD, Monitoring and Versioning, please refer to Operational Governance & Versioning document, at Sections 2, 3 and 4.

5 Risk Analysis

This section identifies potential risks associated with the development and operation of the Recipe Suggester Tool. Risks are categorized by their nature (Technical, Machine Learning, or Privacy) and assessed based on their **Likelihood (L)** and **Impact (I)**.

Legend:

- **L (Likelihood):** 1 (Rare) to 5 (Almost Certain)
- **I (Impact):** 1 (Negligible) to 5 (Catastrophic)
- **Risk Level (R):** $L \times I$ (Low: 1–9, Medium: 10–19, High: 20–25)

5.1 Technical & Infrastructure Risks

ID	Risk Description	L	I	R
R01	External API Unavailability The LLM provider (e.g., OpenAI) or the Auth provider (Google/Meta) goes offline or times out.	2	5	10 (Med)
R02	Latency Constraint Violation The total response time exceeds the 15s threshold due to high traffic or LLM generation slowness.	4	3	12 (Med)
R03	Database Data Loss User profiles are lost due to storage corruption.	1	5	5 (Low)

5.2 Machine Learning & MLOps Risks

ID	Risk Description	L	I	R
R04	Model Hallucination (LLM) The LLM generates a recipe using ingredients <i>not</i> present in the user's fridge or creates unsafe cooking instructions.	3	5	15 (Med)
R05	Data Drift (YOLO) Real-world user photos (dark, blurry, weird angles) differ significantly from the high-quality training datasets, causing accuracy to drop over time.	4	4	16 (High)
R06	Bias & Sensitivity The model suggests meat recipes to a vegan user or fails to respect allergies.	3	5	15 (Med)
R07	Adversarial Attacks Users upload malicious images (noise patterns) designed to crash the inference server or consume excessive resources.	2	3	6 (Low)

5.3 Privacy & Compliance Risks (GDPR)

ID	Risk Description	L	I	R
R08	Sensitive Data Leak (Images) Personal photos (e.g., selfies in the fridge reflection) are accidentally stored or leaked.	1	4	4 (Med)
R09	API Key Exposure The system's or user's API keys are exposed in logs or GitHub commits.	1	5	5 (Med)