# Recipe Suggester Tool

Project Proposal & Development Plan

Baratto Luca, Moro Tommaso, Savorgnan Enrico

## Contents

# 1 Project Summary

This document is meant to provide information about the project "Recipe Suggester Tool" and its management. Deliverables, milestones and architectural decisions are described as in their final version.
Some changes have been due to the project advances.
Implementation details are left to other documents related to the same project.

"Recipe Suggester Tool" is a platform meant to provide innovative and healthy recipes to the user who shares a picture of some food in his/her storeroom or fridge.
The case-of-use for the product will be, e.g., situations in which a customer does not have any idea on what to cook or usually eats very badly. Researches by NCD Alliance and Statista organizations shown that at least a 46% of U.S. citizens and 41% of European citizens usually have a poor diet, rich of *ultra-processed* food.
The mission of the project is to improve customers life quality by having better and healthier diets. Our product can not be seen as a proper solution for the whole problem: indeed, it is bounded by the food already bought by the customer; however, it can be intended as an effective helper.
The primary objective is to develop and effective webapp infrastructure that allows users to login, share picture with a Machine Learning model and receive in output 2 recipes.
System effectiveness will be measured through key performance indicators including Answer Latency, Food recognition & segmentation Effectivness, Dashboard Responsiveness.

## 2 Deliverables

The project delivers integrated software components, ML artifacts, and documentation that implement the specified architecture and satisfy the requirements.

| # | Deliverable | Description |
|---|---|---|
| 1 | **Web Application Frontend** | Responsive web UI for login/registration, image upload, ingredient review, and recipe visualization. |
| 2 | **Backend API (FastAPI)** | Central orchestrator exposing REST endpoints, managing authentication/session, business logic, and integration with ML and LLM services. |
| 3 | **Food Recognition & Segmentation Model** | Trained YOLO-based model for multi-class food detection/segmentation, exported weights and evaluation report. |
| 4 | **Recipe Generation Service (LLM)** | Prompting layer and response parser producing structured recipes (JSON schema) and graceful error handling. |
| 5 | **Data Pipeline & Dataset Versioning** | Image preprocessing, labeling/augmentation workflow, dataset manifest (sources, splits, checksums), and reproducible training input snapshots. |
| 6 | **CI/CD & Containerization** | Automated checks (lint/tests/build), Docker images for components, and release tagging to enable reproducible deployments. |
| 7 | **Project Documentation** | SSD, Development Plan, and Versioning/Governance documents aligned with the final implementation decisions. |

### 2.1 Data Pipeline

Regarding user data ingestion, the product implements the pipeline defined in the System Specification Document at Section 2.1. Regarding instead the data used for the finetuning of the YOLO model, an extensive explanation is provided in the System Specification Document at Section 4.2.

### 2.2 ML Model

As a model for Food Recognition, a fine-tuned version of YOLO11n by Ultralytics is used. YOLO (*You Only Look Once*) is a **one-stage object detector**: it predicts **all objects in one forward pass** of a single neural network, directly producing (i) *where* an object is in the image and (ii) *what* the object is. Given an input image (resized to a fixed resolution), the network generates feature maps and, at many spatial locations (often at multiple scales), outputs candidate detections. Each candidate contains: a **bounding box** $(x, y, w, h)$, the **class probabilities** (e.g., *tomato*, *apple*,

*bread*, ...), and an **objectness/confidence** score, i.e, a probability that the box actually contains an object.
The final detections are obtained by discarding low-confidence candidates and applying **Non-Maximum Suppression (NMS)**: if two boxes overlap significantly and refer to the same class, only the most confident one is kept. Modern YOLO implementations, as the one sued by the product, are commonly described with three blocks: a **convolutional neural network**, serving as feature extractor that transforms pixels into multi-resolution representations, a **multi-scale feature fusion model**, combining fine details and high-level semantics to improve detection of both small and large objects, and finally some **prediction layers** that output box coordinates, objectness, and class scores from the fused features. Historically, many YOLO variants used **anchor-based** heads (predicting offsets from predefined box shapes). Several modern implementations (including the ones by Ultralytics family) also use **anchor-free** heads, which predict boxes more directly and reduce manual anchor tuning.

The project is not strictly dependent on the model of YOLO used.
The choice fell on a model combining ease of fine-tuning and capabilities for providing the best experience as possible to the user. YOLO11n is the *nano*, i.e., the smallest, version of the YOLO11 family, provided by the Ultralytics library.
The model, consisting of 2.4 million parameters, provides inference results on a 640 px squared image in about 1.5 milli-seconds if running on a T4 machine, in 56 milli-second if runnning on a CPU machine.
The model needed a proper fine-tuning since complex images (e.g., fridges plenty of different types of food) revelaed a drawn in the performances.

### 2.3 CI/CD & Monitoring

To ensure code stability and rapid iteration, the project implements a full automation pipeline using **GitHub Actions**.

**Continuous Integration (CI)**: Every Pull Request undergoes a strict validation process defined in `ci.yml`. This includes linting (Ruff/ESLint), static type checking (TypeScript/Mypy), and unit testing (Pytest). Additionally, a "Docker Build Test" ensures that changes to the codebase do not break the containerization process.

**Continuous Deployment (CD)**: Upon merging to the `main` branch, the `cd.yml` workflow triggers. It utilizes **Docker Buildx** to build optimized images for the Frontend, Backend, and Models services. These images are pushed to the **GitHub Container Registry (GHCR)**. The production environment on **Railway** is configured to watch this registry and automatically pull and redeploy the new containers immediately upon update, ensuring a seamless delivery loop.

**Monitoring**: Post-deployment monitoring is handled by the Railway platform, which provides aggregated logs for all microservices and real-time metrics for CPU and Memory usage. This allows the team to identify performance degradation (e.g., slow model inference) or application errors immediately.

# 3 Milestones

The following milestones summarize the main project checkpoints at a high level. Each milestone is expected to be achieved through multiple tasks across one or more sprints.

| Code | Milestone |
|---|---|
| **M01** | **Foundations ready**: development environment, repository workflow, and baseline project documentation in place (scope and constraints aligned with the SSD). |
| **M02** | **Core webapp skeleton**: frontend and backend API structure defined, with authentication flow and basic user journey (login, image upload UI, request/response wiring) enabled end-to-end. |
| **M03** | **Food recognition capability**: YOLO-based model fine-tuned and integrated behind the backend, producing a structured ingredient output (e.g., JSON with labels and confidence) that can possibly be reviewed/edited by the user. |
| **M04** | **Recipe generation capability**: LLM prompting/parsing layer integrated so the system can produce structured recipes (respecting user information such as country/allergies) and handle failures with readable messages. |
| **M05** | **Operational readiness**: CI checks active (tests, type checks, linting, Docker build verification), container images produced, and deployment/monitoring approach (Railway) documented in the Versioning/Governance document. |
| **M06** | **End-to-end validation & release candidate**: integrated demo meets the main constraints (latency, robustness, and error handling) and the final implementation are consistent with the documentation set (SSD, Development Plan, Versioning/Governance). |

# 4 Work Breakdown Structure

The work required for the project has been organized following the common Work Breakdown Structure scheme to increase the work quality.
In practice, the WBS has been followed thanks to the use of a public GitHub Projects page, accessible to all the team members, in order to get a general clear overview on what is going on.
For a better clarification on the meaning of the Tasks and Subtasks, please refer to the following table.

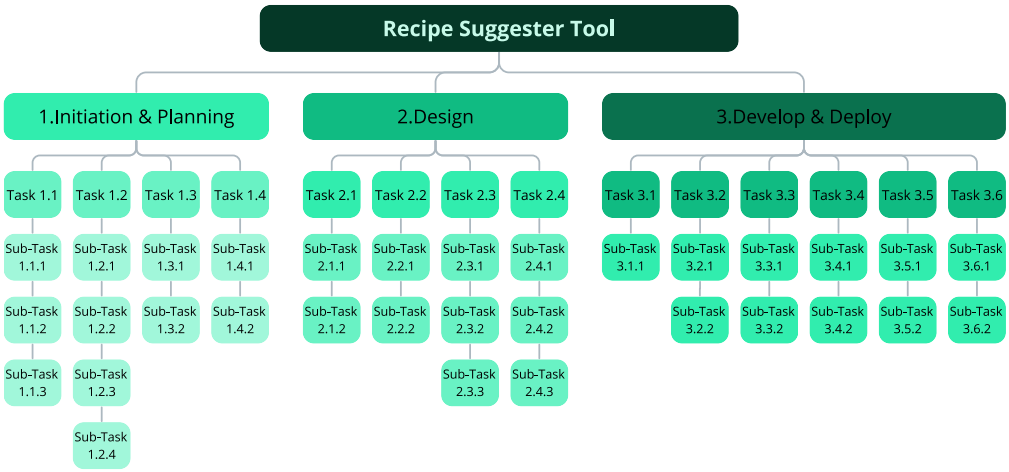


Figure 1: Work Breakdown Structure (WBS) diagram.

| Task ID | Task Description |
|---|---|
| **1** | **Initiation & Planning** |
| 1.1 | *Preliminary research and ideation activities* |
| 1.1.1 | Infrastructure research |
| 1.1.2 | ML Models research |
| 1.1.3 | ML Datasets research |
| 1.2 | *Team organization* |
| 1.2.1 | Roles |
| 1.2.2 | Tentative Schedule |
| 1.2.3 | GitHub Repository |
| 1.2.4 | Code versioning and issue tracking conventions |

# 5 Sprint Plan

The project has been organized in **weekly sprints** to keep a fast feedback loop and to continuously integrate the different components (frontend, backend, ML model, and LLM service).

Each sprint follows a lightweight and practical structure:

- **Sprint planning**: at the beginning of the week, the team selects tasks from the WBS/GitHub Projects board and clarifies scope, priorities, and expected outputs.
- **Implementation**: during the week, tasks are developed in parallel (when possible) and integrated frequently to reduce merge and integration risks.
- **Review & alignment**: at the end of the week, the team reviews what has been completed, collects lessons learned, and updates documentation accordingly (SSD, Development Plan, Versioning/Governance).

The Sprint Plan is summarized below in a compact format, mapping each sprint to the main WBS tasks addressed.

| **Sprint ID** | **Date** | **Week** | **WBS Tasks** |
|---|---|---|---|
| **Sprint 1** | 05/01–11/01 | Week 1 | 1.1, 1.2, 1.3, 1.4 |
| **Sprint 2** | 12/01–18/01 | Week 2 | 2.1, 2.3.1, 2.3.2, 2.4 |
| **Sprint 3** | 19/01–25/01 | Week 3 | 2.2, 2.3.3, 3.5.2 |
| **Sprint 4** | 26/01–01/02 | Week 4 | 3.1, 3.2.1, 3.3.1, |
| **Sprint 5** | 02/02–08/02 | Week 5 | 3.2.2, 3.3.2, 3.4, 3.6.1 |
| **Sprint 6** | 09/02–11/02 | Week 6 | 3.6.2, 3.5.1 |

# 6 Definition of Done & Definition of Ready

This section defines shared criteria used to decide when a work item can be started (**Definition of Ready, DoR**) and when it can be considered completed (**Definition of Done, DoD**).

## 6.1 Definition of Ready (DoR)

| Criterion | Requirements |
|---|---|
| **Goal** | The item has a clear definition, a title and is aligned with project objectives and milestones. |
| **WBS Alignment** | The item is linked to a Work Breakdown Structure task or subtask. The item is assigned to a specific sprint. |
| **Technical Feasibility** | The item requires a reasonable amount of work and effort for a weekly sprint. |
| **Requirements** | The item clearly identifies both functional and non-functional requirements. |
| **Dependencies** | Dependencies and owners are clearly identified. |
| **Acceptance Criteria** | Measurable and clear criteria can be defined for evaluate the completion of the item. |

## 6.2 Definition of Done (DoD)

| Criterion | Requirements |
|---|---|
| **Implementation and integration** | The change is implemented, integrated with the relevant components, and does not break the end-to-end user pipeline. |
| **Quality gates** | CI checks via GitHub Actions are "green", including working/startup checks, type checks, lint checks, and Docker build verification. |
| **Errors handling** | Failures are handled with readable messages and the behavior is consistent with the robustness expectations stated in the SSD. |
| **Security and privacy respect** | GDPR-driven constraints are respected (e.g., do not persist user images; log only request metadata where required); see Requirements at System Specification Document. |
| **Updated Documentation** | If interfaces, behavior, or operational procedures changed, the corresponding documentation is updated (SSD, Development Plan, Versioning/Governance). |
| **General Acceptance** | Deliverables are reviewed and accepted during a weekly sprint review. |

# 7 Resources & Infrastructure

This section summarizes the main resources and infrastructure components required to develop, test, deploy, and operate the Recipe Suggester Tool, consistently with the architecture described in this document and in the SSD (webapp, backend API, YOLO-based food recognition, and external LLM for recipe generation).

## 7.1 Resources

| Category | Resource | Purpose / notes |
|---|---|---|
| Compute | Local Servers | Code Development, Testing and Documentation. |
| Compute | GitHub Actions | Automated checks after pull requests (startup checks, type checks, lint checks, Docker build verification) as described in the Versioning/Governance document. |
| Compute | Railway | Hosting the containerized services (frontend/backend) and providing runtime logs for monitoring. |
| Compute | GPU | Used for YOLO fine-tuning and for faster inference. |
| Storage | GitHub | Code Versioning and collaboration through branches and pull requests (branching model in Versioning/Governance). |
| Storage | GitHub Container Registry | Storage and distribution of Docker images used for deployment. |
| Storage | Microsoft Azure | Versioned datasets used for ML training (sources, splits, checksums). |
| Monitoring | Railway Dashboard | Monitoring of runtime behavior (latency, errors) with access restricted to authorized owners, as stated in the Monitoring section. |

## 7.2 Environments

The project distinguishes the following environments:

- **Local development**: used for implementation and quick checks by developers.
- **CI**: GitHub Actions used to run automated quality gates on pull requests.

- **Staging**: a pre-production deployment (same container images, controlled configuration) used for end-to-end smoke tests before release.
- **Production**: Railway deployment exposed to users, operated with monitoring and restricted access to operational logs.

### 7.3 Human Organizational Resources

- **Project Team**: The project team consisted of three different key roles, each assigned to a member of the team. Baratto Luca participated as a Data Scientist and focused on ML models and training; Moro Tommaso worked as a Full-stack Software Developer for the webapp implementation; Savorgnan Enrico participiated as the Project Manager, guided the whole project workflow and wrote the project documentation.
- **Coordination Tools and Conventions**: Agile methodology has been exploited for project planning, reviews, fast delivery and adaptability. GitHub Projects served as a tool where to look the current state of the project.

# 8 Future Improvements

This section outlines the planned enhancements to address the current system limitations and risks identified during the risk analysis phase.

- **GPU Inference Acceleration**: Currently, the YOLO11n model runs on CPU, with an inference time of $\approx 2$ seconds. Migrating the `models` service to a GPU-enabled instance (e.g., on AWS or a higher Railway tier) would reduce latency to $< 100$ms, mitigating Risk **R02** (Latency Constraint Violation).
- **Database Caching**: Implementing a Redis layer for the Backend API would allow caching of frequent recipe requests, reducing the load on the OpenAI API and lowering costs.
- **Advanced Hallucination Checks**: To mitigate Risks **R04** (Model Hallucination) and **R07** (Adversarial Attacks), secondary "Verifier LLM" agent could be introduced to cross-check the generated recipe against the list of detected ingredients before sending the response to the user.
- **Personalized Dietary Experience**: Expanding the user profile to include important aspects like allergies, dietary goals (e.g., vegan or vegetarian habits, "Low Carb", "High Protein") and other personal information (e.g., food specific for some cultures/nationalities) to provide more targeted and effective suggestions. This will add the project a "recomender-system" complexity layer, will address the Risk **R06** and solve the requirements `F04.01`, currently unsatisfied.
- **Community Features**: Future versions will allow users to share their generated recipes and "fork" recipes created by others, moving from a single-player tool to a social platform.

Obviously, the improvements will require new Milestones, and several new Sprints in order to keep and possibly increase the product quality.
However, the current product structure, developed to ensure rapid scalability, will guarantee relatevely low effort to introduces the new featurees described.