NutriBudget

Optimisation of nutrient budget in agriculture

D5.3 Architecture blueprints and interface wireframes

Funded by the European Union

Cover Delivery Report

Project Information	
Acronym	NutriBudget
Title	Optimisation of nutrient budget in agriculture
Project no.	101060455
Type of Action	RIA
Website	https://nutribudget.eu
Deliverable Information	
Title	Architecture blueprints and interface wireframes
WP number and title	WP5 – Co-create the NutriPlatform
Lead Beneficiary	PwC
Authors	Susan Vrona Bejina (PwC), Florent Bejina (PwC), Vincent Coulette (PwC), Achille Delobbel (PwC), Maher Megdamini (PwC)
Reviewers	Hongzhen Luo (UGENT)
Description	A detailed architecture blueprint for the NutriPlatform prototype
Type	R – Document, report
Dissemination Level	PU - Public
Status	Final version
Submission due date	31st August 2025
History of Changes	
Version 0.1	Draft created by PwC (28.08.2025)
Version 1.0	Deliverable approved by UGENT (29.08.2025)
Version 1.1	Fix hyperlink errors and overall revision and update of the deliverable (30.09.2025)
Version 2.0	Final version submitted by UGent (30.09.2025)

Executive Summary

This summary provides an organized overview of the main components and processes covered in this documentation. Each subsection is designed to guide the reader through the system's context, architectural foundations, components, and supporting functionalities.

Section "System context" outlines the overall environment and background relevant to the deliverable. It begins by establishing the European agricultural context, highlighting the specific challenges and needs present in this sector. The section then identifies the main challenges to address and the proposed solution through a targeted and structured approach.

Section "Architecture Overview" presents the structure of the NutriPlatform. It details the major system components, their interactions, and how these elements collectively support the solution's objectives. The overview highlights the modular design, specifying the roles of core modules such as data acquisition, processing, and user interface layers. Additionally, it explains how the architecture ensures scalability, security, and interoperability, facilitating efficient integration with external systems and adaptability to changing requirements.

Section "Frontend Component" describes the user-facing layer of the NutriBudget system. It details the technologies and frameworks used for interface development, the main functionalities provided to end users, and the strategies ensuring responsiveness and accessibility across devices. The section also highlights the integration with backend services, allowing seamless data interaction and real-time updates.

Section "Backend Component - Hasura" focuses on the Hasura GraphQL engine as the central data access and management component. It explains how Hasura provides a secure, scalable API layer over the system's databases, enabling efficient data queries and mutations. The section also covers authorization mechanisms, real-time data synchronization capabilities, and the role of Hasura in connecting the frontend with core backend services.

Section "Backend Component - NutriPlatform Services" outlines the suite of backend services supporting NutriBudget functionalities. It describes the primary services responsible for data processing, business logic, and integration with external agricultural data sources.

Section "Data" explains the data model underpinning the platform, along with the rationale behind the selection of related technologies and how these choices support the system's performance and integration capabilities.

Section "Security" addresses the comprehensive measures implemented to protect the NutriBudget platform. It covers authentication and authorization protocols, data encryption methods, and routine security audits. Additionally, the section highlights compliance with relevant cybersecurity standards and the ongoing monitoring processes to detect and mitigate potential threats.

Section "Deployment" is still being defined and will be updated as more information becomes available.

Section "Wireframes" provides visual representations of the main user interface layouts. It showcases the initial design concepts for core screens, illustrating navigation flow and user interactions. The section demonstrates how user experience considerations inform the structure and placement of key features within the platform.

Table of Contents

List of Figures

List of Tables

Introduction

This deliverable arises from the urgent need to steer European agriculture toward more sustainable practices, in response to rising environmental challenges and the ongoing demand for robust food production. Over the past decades, agriculture in Europe has witnessed unprecedented intensification, resulting not only in increased yields but also in mounting pressure on natural resources, soil health, and the broader environment. The persistent reliance on nitrogen and phosphorus inputs, combined with a significant decline in soil organic carbon, underscores the necessity for innovative management tools that address both productivity and environmental integrity.

Against this backdrop, the main purpose of this deliverable is to present the architecture blueprint of the NutriPlatform—a cutting-edge nutrient management system. Designed for a diverse audience that includes farmers, advisory services, policymakers, and regional authorities, the NutriPlatform aims to bridge the existing gap between scientific research and on-farm practice. By providing a comprehensive and integrated decision-support environment, the platform seeks to optimize nutrient use at the farm level, improve sustainability outcomes, and ensure the viability of European agriculture for future generations.

The following sections will outline the European agricultural context, highlight the key challenges related to nutrient management, and detail the vision, functionality, and innovative features that define the NutriBudget approach and its central tool, the NutriPlatform.

1 System context

1.1 Context and Problem Statement

Since the 1960s, European agriculture has undergone massive intensification, resulting in a 68% increase in food production. However, this growth has had significant environmental impacts, affecting biodiversity, climate, as well as water and air quality. This period has been marked by increased use of nitrogen (N) and phosphorus (P) inputs, and a notable decline in soil organic carbon (SOC), exacerbated by climate change.

The major challenge is to reconcile the need for sustainable intensification to meet growing food demand with the necessity of optimizing yields without compromising environmental integrity. Currently, there is a clear lack of integrated tools for effective nutrient management, creating a significant gap between scientific research and on-farm agricultural practices.

1.2 Proposed Solution

To this end, the vision of NutriBudget is to provide an **integrated nutrient management platform**, named **NutriPlatform.** as a **decision-support tool** intended for a wide range of stakeholders: farmers, farm advisors, European policy makers, and regional authorities.

1.3 Stakeholders

Farmers are at the heart of the system. Their needs focus on optimizing practices, regulatory compliance, and the economic viability of their operations. They aim to maximize yields while minimizing environmental impacts, but face constraints related to resources, technical complexity, and sometimes resistance to change.

Farm advisors act as essential intermediaries. They require robust decision-support tools and reliable scientific data to effectively support farmers. Their goal is to improve practices in the field, while managing constraints related to their own technical training and the wide diversity of farms they oversee.

At the European level, **policy makers** seek insights and impact data to guide Union policies. Their goals are to drive the transition to sustainable agriculture and achieve the set environmental objectives, while navigating regulatory complexity and harmonization challenges among Member States.

1.4 Business Goals

The NutriPlatform will be supported with key project outcomes including: (i) co-creation of a Mitigation Measures Catalogue containing more than 50 effective agronomic practices, (ii) development of two integrated NutriModels for regional and farm level nutrient budgets and flows, and design of 10 roadmaps for improved nutrient management, (iii) co-creation of data standards and a NutriKPI framework, (iv) generation of data from research in five pilot regions to validate more than 15 innovative agronomic measures.

The objectives include implementing the platform in at least four regional farming networks, developing policy recommendations in three key areas, and establishing a Nutri-actor network bringing together more than 250 organizations. The goal is to achieve adoption by more than 40,000 target users and validation in the five pilot regions (Belgium, Spain, Italy, Finland, Switzerland). From an environmental perspective, the project must demonstrate a quantifiable reduction in nutrient losses and a significant contribution to the European Union's environmental objectives.

1.5 Constraints and Requirements

The NutriBudget project is developed within a demanding European regulatory framework, which shapes its objectives, technical choices, and deliverables. Compliance with these requirements is not only an administrative necessity: it is a prerequisite for access to European funding, institutional recognition, and the adoption and impact of project results by agricultural stakeholders.

Why are these requirements essential?

- **Common Agricultural Policy (CAP):** The CAP is the cornerstone of European agricultural policy and conditions the granting of subsidies on the adoption of sustainable agricultural practices, reduction of environmental impacts, and traceability of inputs. The new CAP (2023–2027) sets higher environmental and climate ambitions, with new requirements for farmers and more funding for eco-schemes. Digital tools are explicitly encouraged to help farmers comply with CAP requirements, including nutrient management and reporting (https://agriculture.ec.europa.eu/common-agricultural-policy_en).
- **Farm to Fork Strategy:** As a pillar of the European Green Deal, the Farm to Fork Strategy aims to accelerate the transition to a sustainable food system with a neutral or positive environmental impact. It sets concrete targets, such as reducing nutrient losses by at least 50% by 2030 (European Commission – Farm to Fork Strategy). Projects funded under Horizon Europe are expected to align with and contribute to these targets.
- **Zero Pollution Action Plan:** This EU action plan sets key 2030 targets to reduce air, water, and soil pollution to levels no longer considered harmful to health and natural ecosystems, including reducing nutrient losses by 50% (European Commission – Zero Pollution Action Plan). The NutriBudget project is funded under the Horizon Europe call “HORIZON-CL6-2021-ZEROPOLLUTION-01”, which explicitly requires alignment with this action plan.

How are these requirements addressed in NutriBudget?

To meet these obligations, the NutriBudget platform includes:

- Monitoring and measurement tools enabling users to demonstrate compliance with the CAP, Farm to Fork Strategy, and Zero Pollution Action Plan requirements.
- Features that facilitate traceability of inputs, sustainable resource management, and reduction of nutrient losses.
- Indicators and reports aligned with European objectives, allowing farmers, advisors, and policymakers to manage their actions in accordance with regulations.
- A technical architecture ensuring interoperability with regulatory reporting systems and compliance with data protection requirements (GDPR).

This approach ensures that technical developments, scientific models, monitoring tools, and policy recommendations are all aligned with European priorities. It facilitates integration of results, data sharing, and the valorization of deliverables with institutional stakeholders.

Building on these regulatory foundations, the technical architecture of NutriBudget is designed to meet several critical requirements. The platform must ensure a high level of interoperability, enabling seamless information exchange both among its internal components and with potential external systems. Scalability is also a key consideration: the system must reliably support dozens of simultaneous users and provide multi-platform accessibility, including both web and mobile devices. Finally, the architecture anticipates the integration of Artificial Intelligence (AI) models, which will be used to calculate and assess the environmental impacts of various agricultural practices.

The NutriBudget project must operate within a strict European regulatory framework. It must ensure compliance with the reforms of the **Common Agricultural Policy (CAP)** and actively contribute to the objectives of the **Farm to Fork** strategy, which aims for a **50% reduction in nutrient losses** by 2030. In addition, the platform must align with the **Zero Pollution Action Plan** and guarantee rigorous protection of personal and agricultural data, in accordance with the **General Data Protection Regulation (GDPR)**.

On the technical side, several key requirements are defined. The platform must ensure high interoperability to enable information exchange between its own components and with potential external systems. The architecture must guarantee scalability to support dozens of simultaneous users and offer multi-platform support (web, mobile) accessible on all modern devices.
Finally, the system must provide for the integration of an Artificial Intelligence (AI) model to calculate the environmental impacts of different agricultural practices.

2 Architecture Overview

2.1 Five Pillars Architecture

NutriBudget is built on an interdisciplinary and holistic concept structured around five key pillars to support Europe's transition toward balanced nutrient management and environmental targets:

- **NutriDesign**: Defines the critical performance indicator framework to measure progress.
- **NutriData**: Identifies and collects existing and new mitigation measures data.
- **NutriModels**: Models the integrated impact of nutrients across farming systems and spatial scales.
- **NutriPlatform**: Translates knowledge into a decision-support tool (DST) for farmers and regional authorities.
- **Nutri-Actor Approach**: Ensures co-creation with users and stakeholders through a multi-actor methodology.

These five pillars are illustrated in Figure 1 and detailed in the following sections.

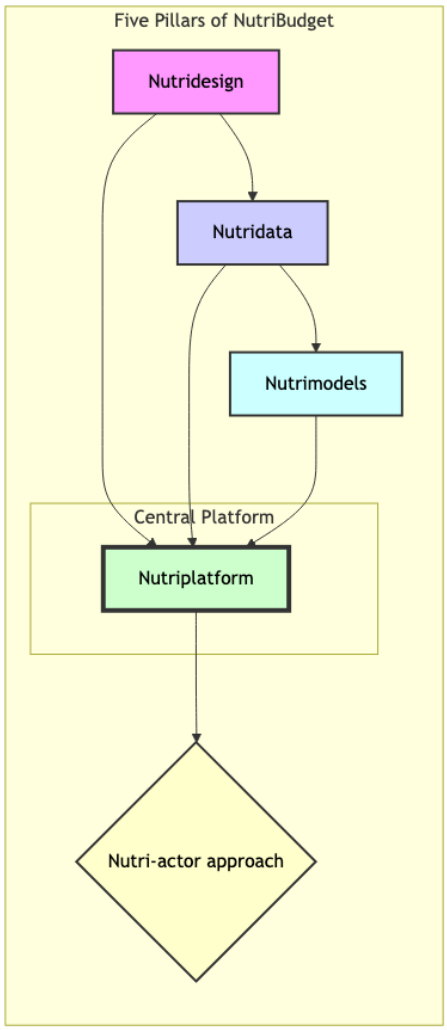

Figure 1. Diagram illustrating the conceptual relationships between the five foundational pillars of the NutriBudget project.

2.1.1 NutriDesign

The **NutriDesign** pillar is primarily aimed at designing opportunity maps and establishing a monitoring framework. To achieve this, it relies on several key components, including meta-analytical models, multidisciplinary co-creation workshops, spatial opportunity maps, agri-environmental indicators.

2.1.2 NutriData

The **NutriData** pillar focuses on the collection and generation of data related to mitigation measures. It aggregates information from scientific literature and existing databases, complemented by field data from the project's five pilot regions. The entire process is based on a common standardized methodology (as stated in D4.2 Description of experimental set-up and methods used in each pilot, submitted in January 2024), covering a wide range of agricultural systems (conventional, organic, and agro-ecological).

2.1.3 NutriModels

The **NutriModels** pillar aims to develop integrated models to simulate nutrient flows and budgets. It combines a top-down approach with the MITERRA-Europe model and a bottom-up approach with MITERRA-Farm. These models are enhanced by integrating the INTEGRATOR tool and machine learning algorithms to refine predictions.

2.1.4 NutriPlatform

The **NutriPlatform** is the technical core of the project, turning research into a user-friendly decision-support tool for end users. Its main components include a modern web user interface, deep integration with databases and scientific models, a cloud-native microservices architecture, and interoperability designed to communicate with external systems.

2.1.5 Nutri-actor Approach

Finally, the **Nutri-actor Approach** pillar ensures co-creation and stakeholder involvement throughout the project. It is structured around a network of more than 250 organizations, over 23 co-creation and validation sessions, and contributes to the development of regional food strategies and agri-food hubs.

2.2 Design principles

2.2.1 Distributed Infrastructure

The architecture of the NutriPlatform is based on a distributed infrastructure, deployed on a cluster of **virtual machines (VMs)**. To ensure portability and environment consistency, all services are managed via containerized deployment with **Docker**. In production, orchestration is handled by **Docker Swarm**, a solution that enables simple and efficient horizontal scalability by adding new nodes to the cluster.

2.2.2 Microservices Approach

The system adopts a microservices architecture to promote a clear separation of responsibilities between different components. This choice ensures deployment independence, improves resilience and fault tolerance, and greatly facilitates the maintenance and evolution of each service independently.

2.2.3 European Standardization

A fundamental principle of the project is standardization at the European scale. The platform aims to normalize practices and harmonize strategies for nutrient management. The objective is to define common standards for all EU Member States, promoting a unified approach for more sustainable agriculture, while also including partners such as FiBL (Switzerland) in the consortium.

2.2.4 Interoperability

Interoperability is at the heart of the technical design. It is ensured by the use of **open data standards** and **APIs exclusively based on GraphQL**, which offer a flexible and powerful interface. The architecture is also designed to allow easy integration with existing systems and relies on standardized data exchange formats to facilitate communication between services.

2.3 High-Level Diagrams

2.3.1 System Context

The **NutriPlatform** is the central system of the NutriBudget project. It is designed as a **Decision Support Tool (DST)** aimed at optimizing nutrient management at multiple scales. Its boundaries encompass all interactive features and data processing services required to provide agronomic analyses and recommendations.

The system interacts with several types of external actors. **Farmers** and **agricultural advisors** are the main users, consuming recommendations for farm-level management. **Regional authorities** and **EU policy makers** use the platform to develop public policies based on aggregated data and modeling. All these actors implicitly provide contextual data (farm parameters, areas of interest) and consume the information processed by the platform.

To operate, the NutriPlatform relies on three major external systems. The **NutriBudget API** exposes scientific models to perform nutrient balance calculations. The **MapTiler** service provides basemaps and geocoding services essential for geospatial visualizations. These interactions highlight the system's key constraints: **interoperability** with scientific and mapping services, **compliance** with EU directives, and the ability to operate at **multiple scales**, from farm plot to region.

2.3.2 Container Diagram

Figure 2 presents a high-level container diagram showing the main components and external integrations of the NutriPlatform. The internal architecture of the NutriPlatform is organized into distinct software containers, each with a clear responsibility, to ensure modularity and separation of concerns.
The user interaction layer is provided by the **Frontend**, a single-page application (SPA) developed with **Vue.js and Quasar.** This is the single-entry point for end users. All data requests and interface actions are centralized to an API gateway.
This gateway, the **API Gateway Layer**, is implemented with **Hasura GraphQL Engine**. Hasura serves as the API hub, exposing a secure GraphQL interface to the Frontend. It manages data access and delegates complex business logic to specialized services via *Actions* and *webhooks*.
The **Services Layer** groups business logic and authentication containers. **NutriPlatform Services**, a **Fastify service in TypeScript**, implements complex logic that cannot be handled by Hasura, such as orchestrating calls to the external NutriBudget API. Alongside, **Keycloak** operates as a dedicated authentication and authorization service, managing the entire user lifecycle and issuing JWT tokens according to the OIDC protocol.
Data persistence is managed in the **Data Layer** according to a database-per-service principle. A **PostgreSQL instance with the PostGIS extension** is dedicated to Hasura for storing application and geospatial data. A second **PostgreSQL instance** is used exclusively by Keycloak for its own user and configuration management needs.

Finally, the system interacts with essential **external services**: the **MapTiler API** for mapping, called directly by the Frontend, and the **NutriBudget API** for scientific modeling, called by the NutriPlatform Services. Containers for **monitoring** and **logging** are planned in the architecture for observability, but their technical implementations remain to be defined.

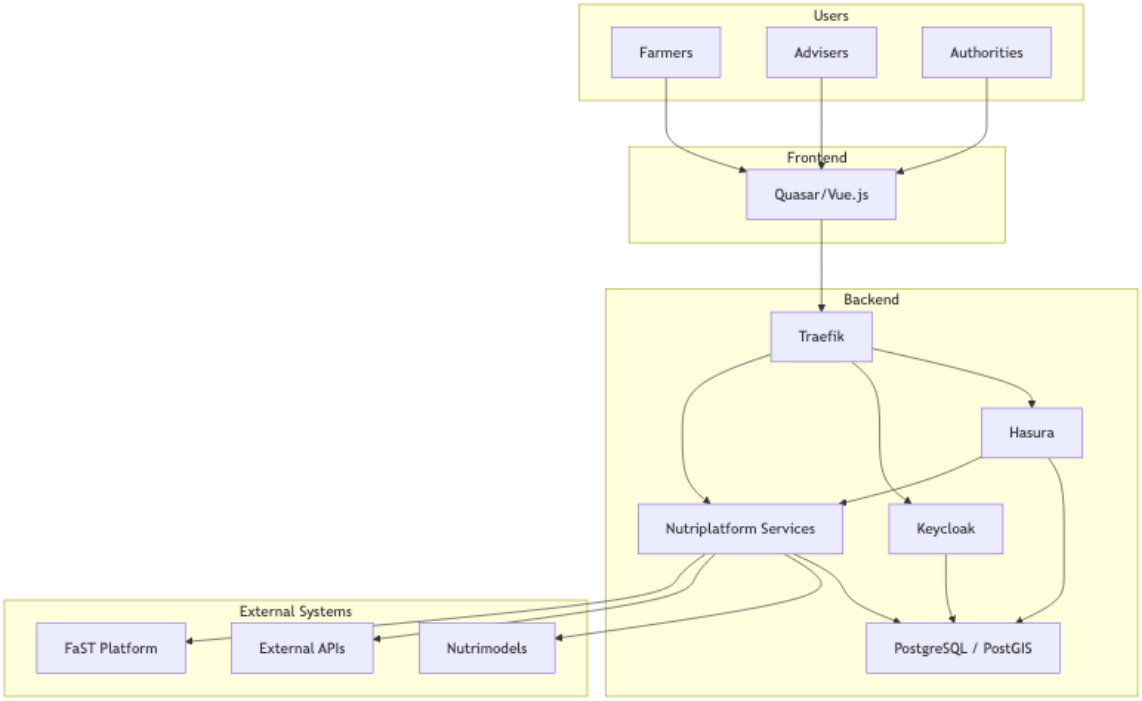

Figure 2. High-level container diagram showing the main components and external integrations of the NutriPlatform.

2.4 Global Technology Stack

2.4.1 Frontend

The frontend technology stack is built around **Vue.js 3** and the **Quasar Framework**. For translation management, **vue-gettext** was chosen, evolving from the solutions used in the FaST project. Communication with the backend is handled via the **URQL** GraphQL client, a lighter alternative to Apollo Client. The mapping component is managed by **MapLibre GL JS**, and the entire application is compiled and optimized by the **Vite** build tool.

Lessons from the FaST Project

The technology choices for NutriBudget build on the experience of the FaST (Farm Sustainability Tool) project. The SPA (Single Page Application) architecture was retained as it proved robust and was validated in nine European regions. Similarly, the Vue.js + Quasar combination is maintained. Major changes include the switch from Apollo to URQL for better performance and the migration to TypeScript to improve type safety and code maintainability.

2.4.2 Backend

The backend is structured around the **Hasura GraphQL Engine**, which serves as the main API. Complementary services are developed in **TypeScript** with the **Fastify** framework. Data persistence is provided by a **PostgreSQL** database enhanced with the **PostGIS** spatial extension. Authentication is delegated to **Keycloak**, and TypeScript data access is facilitated by the **Prisma** ORM.

Backend Evolution Compared to FaST

On the backend side, the **Hasura + PostgreSQL** architecture, whose success was confirmed on FaST, is fully retained. However, several simplifications have been made: the **Django framework was removed** as it was deemed unnecessary. The **microservices architecture has been drastically simplified** to avoid excessive complexity. Finally, a language change was made, moving from Python to **Node.js with TypeScript**, to share skills with the frontend team.

2.4.3 Infrastructure

The infrastructure is entirely based on **containerization** with **Docker**, and orchestration is managed by **Docker Swarm** for pre-production and production environments. **Traefik** is used as a reverse proxy for load balancing and SSL certificate management, while **Nginx** serves the frontend static files. **Monitoring and logging** solutions are yet to be defined.

Infrastructure Simplification Compared to FaST

Major simplification efforts have been made compared to FaST. The **Kubernetes infrastructure was replaced by Docker Swarm**, considered less complex and better suited to the project's needs. The move from a multi-repository organization to a **monorepo** aims to accelerate development. These choices lead to **lighter orchestration**, reducing operational complexity while maintaining the necessary scalability.

2.4.4 Integrations

The NutriPlatform integrates with several key external services: the **MapTiler API** for mapping services and the **NutriBudget API** exposing scientific models.

2.5 Infrastructure and Machine Architecture

2.5.1 Production Environment

The production environment for the **NutriPlatform** is engineered for **high availability** and **optimal performance**, based on a unified DNS architecture centered around the **nutriplatform.eu** domain. All name resolution, application services, and user access are now exclusively provided via the root domain nutriplatform.eu and specialized subdomains as required (e.g.: app.nutriplatform.eu, api.nutriplatform.eu, auth.nutriplatform.eu). The use of any legacy or placeholder domain is strictly deprecated.

The entire production platform is deployed on a cluster of 9 VMs, totalling 26 vCPU, 56 GB RAM, and 470 GB SSD storage. Specific role distribution is shown in Table 1.

Table 1. Virtual Machines specifications and roles (production).

Node	Role	Performance	# Nodes
Manager	Orchestration cluster management	2 vCPU, 4 GB RAM, 40 GB SSD	3
Worker	App services runtime	8 vCPU, 16 GB RAM, 50 GB SSD	1
Database	Persistent storage, spatial queries	8 vCPU, 16 GB RAM, 100 GB SSD	2
Keycloak	Identity & access management	2 vCPU, 4 GB RAM, 40 GB SSD	1
Observability	Metrics & logs collection	2 vCPU, 8 GB RAM, 100 GB SSD	1

Backup / Ops	Scheduled jobs maintenance	2 vCPU, 4 GB RAM, 40 GB SSD	1

SSL/TLS Certificate Management and TLS Termination

SSL/TLS certificates are managed by the hosting infrastructure. **All incoming traffic to NutriPlatform's VMs arrives over HTTP**; the platform does not manage any aspect of TLS or SSL certificates, nor does it support or expose HTTPS internally under any circumstance.
There is absolutely no application-level processing, scripting, or internal process involved in any stage of certificate lifecycle (provisioning, renewal via ACME/Let's Encrypt, or manual management). This responsibility is entirely delegated to the entry infrastructure (edge).
This approach ensures:

1. A **strict separation of responsibilities** (infrastructure vs application)
2. A **reduced application attack surface**
3. Centralized and coherent management of traffic security

Trust Assumption for Internal Traffic

All incoming user traffic (HTTP or HTTPS) is terminated at the edge reverse proxy, which is the **exclusive trusted security boundary** of the platform. After this point, any traffic routed to internal VMs or Docker services (APIs, web frontends, app logic, databases) is plain HTTP, considered secure *because* it runs only on fully isolated, dedicated overlay networks. The NutriPlatform enforces a **total trust model for intra-cluster traffic**: no revalidation, no encryption, no support for TLS/SSL at any inner layer.
This architectural assumption is fundamental: for any newly exposed service, only the *edge* is authoritative for X.509 validation, and all downstream communication is unencrypted HTTP.

2.5.2 Staging/Preproduction Environment

The staging environment faithfully mirrors the production doctrine: every internal service and VM listens in HTTP only, without any form of TLS/SSL support, regardless of environment. All ingress traffic (from the Internet or test users) is routed via subdomains (such as app.staging.nutriplatform.eu), and TLS/SSL is always terminated by the edge Traefik proxy or load balancer. No staging machine or container can or should ever expose HTTPS/TLS endpoints to the cluster network. Security policy parity with production is enforced.

As previously described in section 2.5.1, the staging environment mirrors the production setup in terms of VM roles and service distribution. Table 2 below provides the specific VM specifications and allocation for the staging environment.

Table 2. Virtual Machines specifications and roles (staging).

Node	Role	Performance	# Nodes
Node	**Role**	**Performance**	**# Nodes**
Manager	Orchestration cluster management	2 vCPU, 4 GB RAM, 40 GB SSD	1
Worker	App services runtime	2 vCPU, 8 GB RAM, 50 GB SSD	1
Database	Persistent storage, spatial queries	2 vCPU, 4 GB RAM, 50 GB SSD	2
Keycloak	Identity & access management	1 vCPU, 2 GB RAM, 40 GB SSD	1

2.5.3 Docker Swarm Placement Constraints

To ensure resilience and proper distribution of services across the cluster, **placement constraints** are defined at the Docker Swarm level. These rules, based on node roles and labels, ensure that critical services (such as the database or authentication) are deployed on dedicated and appropriate machines. Label naming convention (Docker Swarm): technical label and constraint literals remain in English and are formatted as monospace. The canonical key is node.labels.tier and the allowed values are: app, database, keycloak, observability, backup. In English text, we write "database", but never translate label values.

2.6 Main Data Flows

2.6.1 General Data Flow Diagram

The diagram below provides an overview of data flows within the NutriPlatform, illustrating interactions between external actors and internal components.

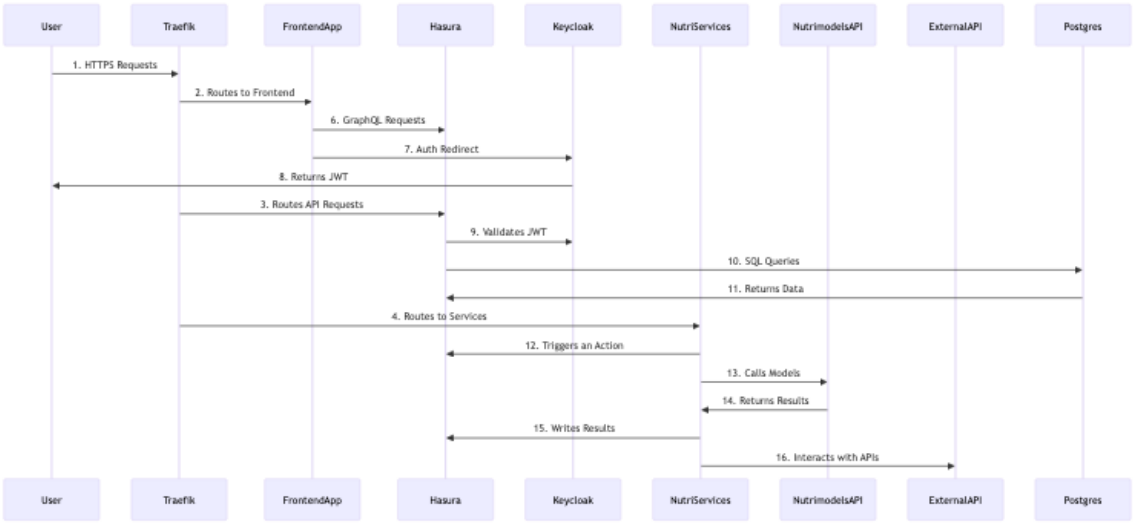

Figure 3. Sequence diagram representing the main data flows and interactions between users, platform components, and external services.

2.6.2 Standard User Flow

The standard data flow describes the fundamental interaction between the user and the platform for information retrieval. The process begins when the **user** performs an action on the **Frontend** interface, such as navigating to a page displaying farm data. This action triggers a **GraphQL request** sent by the Frontend to the **Hasura** server. Hasura, acting as a data access layer, translates this GraphQL request into an optimized **SQL query**. This is executed on the **PostgreSQL** database, which returns the requested raw data. Hasura receives the database response, restructures it in GraphQL format, and transmits it to the Frontend. Finally, the **Frontend** updates its interface to **display the data** to the user, thus completing the cycle.

2.6.3 Authentication Flow

The authentication process is initiated when the **user** attempts to access a protected resource via the **Frontend**. The Frontend, detecting the absence of an active session, **redirects** the user to the **Keycloak** identity server for authentication. The user authenticates with Keycloak (credential entry, SSO, etc.). If successful, Keycloak generates a **JSON Web Token (JWT)** and returns it to the Frontend. For each subsequent request to a protected resource, the **Frontend** includes this **JWT** in the authorization header of its GraphQL request to **Hasura**. Hasura, in turn, **decodes the JWT** to extract

identity information (such as role and user ID) to apply the appropriate permission rules. Note that in this flow, Hasura trusts the JWT and **does not revalidate its signature** with Keycloak on each call, relying on the secure configuration of the relationship between the two services. Once authorization is confirmed, Hasura processes the request and returns the **authorized response** to the Frontend.

2.6.4 NutriBudget Flow

The NutriBudget flow is triggered when a user initiates a calculation operation, such as a NutriBudget balance, from the **Frontend**. The interface sends an **action** (a specific GraphQL mutation) to **Hasura**. Hasura forwards this action to the **NutriPlatform Services**, the backend service responsible for complex business logic. The **NutriPlatform Services** then orchestrate the call by formulating a **request** to the external **NutriBudget API**, providing it with the necessary input data for the calculation. The NutriBudget API executes the scientific model and, once the calculation is complete, returns the **results** to the NutriPlatform Services. The NutriPlatform Services service then **saves these results** in the database via a new request to Hasura. Finally, Hasura sends a **completion notification** to the Frontend, often via a GraphQL subscription, informing the user that the results are available for review.

2.6.5 Map Flow

The map flow manages the display and interactivity of maps. The process begins when the **Frontend initializes** a map component using the **MapLibre** library. Once initialized, the MapLibre library sends **map tile requests** to the **MapTiler** API, which provides the background map images needed for display. At the same time, if the user wants to perform an address search, the **Frontend** can directly query the **geocoding** service of the MapTiler API to convert a textual address into **geographic coordinates**. In return, the **MapLibre** library manages user interactions with the map (zoom, pan, clicks) and emits **events** that the Frontend can listen to in order to trigger application logic, such as displaying information about a point of interest.

2.7 Architectural Patterns

2.7.1 API Gateway

The architecture uses the **API Gateway** pattern, with **Hasura** acting as the **main entry point** for all frontend requests. Hasura manages **routing** requests to the appropriate backend services (e.g., NutriPlatform Services) and centralizes the first layer of **authentication** and authorization.

2.7.2 Database-per-Service

In line with the microservices approach, the **Database-per-Service** pattern is applied. Each service has its own database to ensure **complete data isolation**. Specifically, a main **PostgreSQL + PostGIS** database is dedicated to Hasura and business data, while a **separate PostgreSQL database** is used by the **Keycloak** authentication service.

2.7.3 Event-Driven Architecture (EDA)

The system incorporates elements of an **event-driven architecture**. **Hasura events** (triggers on data changes) are used to **trigger asynchronous actions**, such as calling an external service after a new data insertion. **Webhooks** are also used for integrations with third-party systems.

2.8 Scalability Considerations

2.8.1 Horizontal Scalability

The architecture is designed for **horizontal scalability**, allowing increased capacity by adding resources rather than upgrading individual machines. This is achieved through **database replication**, **load balancing** managed by Traefik (and not Nginx as previously mentioned, a correction to note), and the ability to **scale service containers** via **Docker Swarm**.

2.8.2 Performance Optimizations

Several strategies are in place to ensure optimal performance. These include **continuous optimization of GraphQL queries** to minimize load, the use of **Gzip compression** to reduce data transfer size, and the implementation of **relevant database indexes** to accelerate query response times.

2.8.3 Monitoring and Observability

To ensure system stability and performance, an observability strategy is planned. It will rely on the collection of **application metrics**, **centralized logging** from all services, and the implementation of **health checks** to continuously monitor the health status of each architecture component. Specific tools (such as Prometheus, Grafana, Loki) are planned, but their detailed implementation remains to be defined.

2.9 Security

2.9.1 Authentication and Authorization

Identity and access management is centralized via **Keycloak**, which implements the standard **OAuth 2.0** and **OpenID Connect** protocols. Authentication between the frontend and backend relies on the exchange of **JSON Web Tokens (JWT)**. Authorizations are finely managed through a **role-based access control (RBAC)** model, allowing specific permissions to be defined for each user type (farmer, advisor, etc.).

2.9.2 Data Protection

Data protection is a top priority. All communications between services and with users are secured by **encryption in transit (TLS)**. Sensitive data stored in databases is also protected by **encryption at rest**. The entire platform is designed to be **GDPR compliant**, ensuring the protection of personal and agricultural data.

2.9.3 API Security

API endpoint security is ensured by several mechanisms. **Systematic input validation** is performed to prevent injection attacks. In addition, a **CORS (Cross-Origin Resource Sharing) protection policy** is configured to restrict the domains authorized to query the API, thus protecting against malicious requests from unauthorized sources.

3 Frontend Component

3.1 Overview

The NutriPlatform is a web frontend application for the NutriBudget project, providing a modern and intuitive interface for agricultural nutrient management.

3.1.1 Main Technologies

- **Framework**: Vue.js 3 (Composition API)
- **UI Framework**: Quasar Framework
- **GraphQL Client**: URQL
- **Mapping**: MapLibre GL JS with MapTiler
- **Build Tool**: Vite via Quasar CLI
- **Authentication**: Keycloak JS

3.1.2 General Architecture

The NutriPlatform frontend application follows a **hybrid architecture** combining organization by technical type and by business domain, optimized for scalability and maintainability.

Organizational Principles

- **First level by technical type**: Facilitates navigation and respects Quasar conventions
- **Sub-levels by business domain**: Organizes business logic (farm, field, home, etc.)
- **Clear separation of responsibilities**: Each folder has a specific and well-defined rôle

This structure offers several significant advantages. First, it ensures **Quasar compliance** by respecting the framework's conventions, especially for the boot/, layouts/, and pages/ folders. Second, it guarantees **scalability** by making it easy to add new business domains. Third, **maintainability** is improved thanks to a predictable structure for developers. Finally, it contributes to **performance** by optimizing tree-shaking and route-based code splitting.

Project Structure

The organization follows a clear and modular approach at the main folder level:

```
src/
├── boot/              # Quasar boot plugins
├── components/        # Reusable cross-cutting components
├── css/               # Global styles and variables
├── data/              # Reference data (interfaces, types, possibly mocks)
├── layouts/           # Quasar page layouts
├── lib/               # Utilities and configurations
├── pages/             # Pages organized by business domain
├── router/            # Vue Router configuration
├── stores/            # Pinia stores for global state
├── types/             # TypeScript definitions
└── utils/             # Generic utility functions
```

This structure ensures **controlled scalability** while respecting **Quasar conventions** and **Vue.js 3 best practices**.

3.1.3 Main Architectural Flow

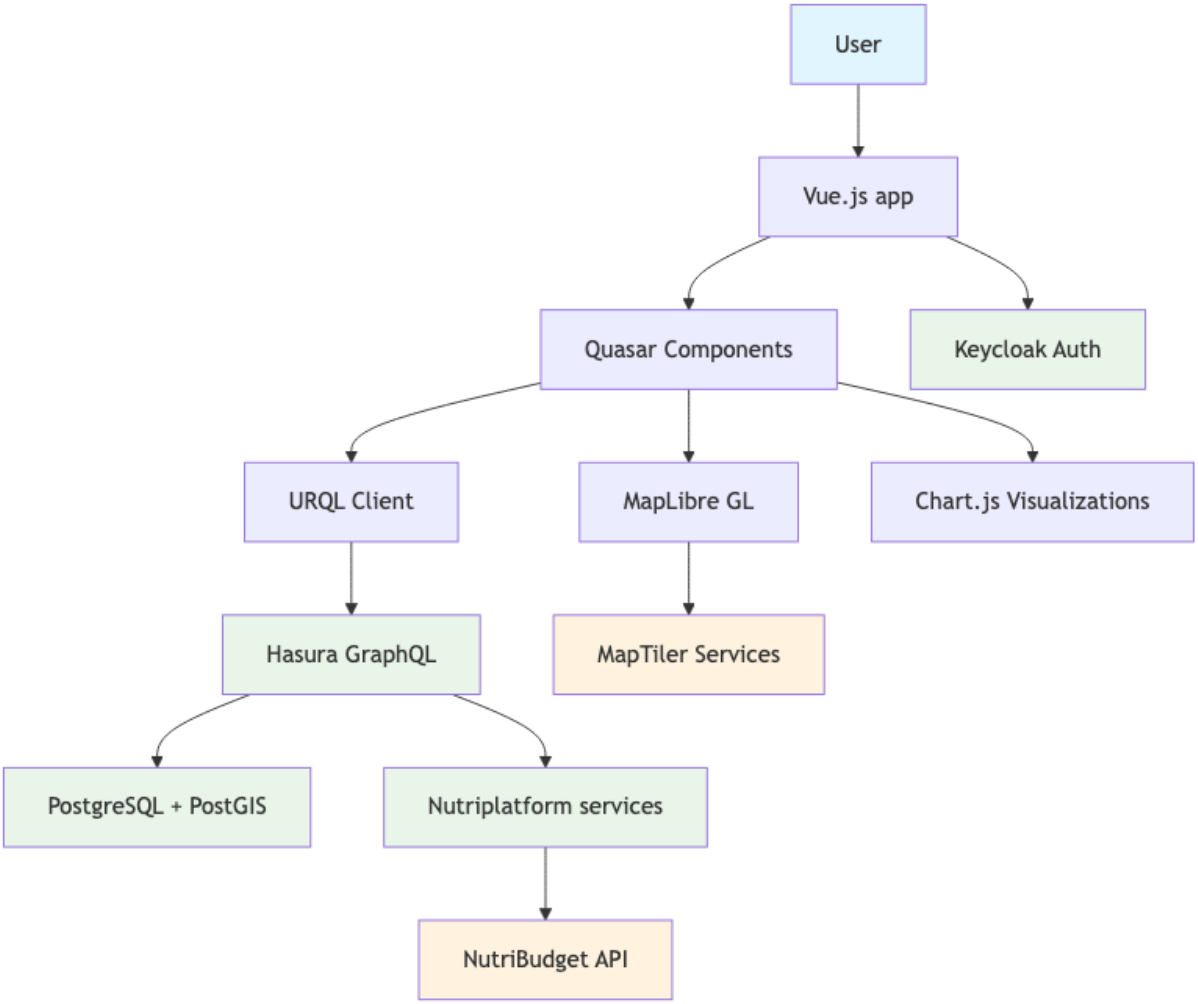

Figure 4. Main architectural flow of the NutriPlatform frontend application.

Main data flow:

- **Authentication**: The user authenticates via Keycloak
- **GraphQL Requests**: URQL manages communications with Hasura
- **Access to agricultural data and services**: Hasura interacts with the data layer stored in PostgreSQL and/or queries the Nutriplatform services layer
- **Nutrient budget calculations**: NutriBudget API processes nutrient models
- **Map visualization**: MapLibre displays geospatial data via MapTiler
- **Graphical visualization**: Chart.js presents results and KPIs

3.2 Core Technologies

3.2.1 Overview

The NutriPlatform frontend application is built on a set of modern technologies selected to meet the specific needs of the European agricultural context.

3.2.2 Rationale for Technology Choices

SPA Architecture (Single Page Application)

The choice of a SPA architecture is preferred over SSR (Server-Side Rendering) alternatives for several reasons specific to NutriPlatform:
First, it is a **proven solution with FaST**, as the SPA architecture was validated on the FaST platform deployed in 9 European regions. It offers **architectural simplicity** by reducing complexity compared to SSR/hybrid solutions. This results in **development savings**, as there is no duplication of logic between client and server. Moreover, the application does not have **specific needs** requiring SSR advantages, such as SEO or static generation. Finally, **deployment is simplified** thanks to a clear separation between frontend and backend, which facilitates maintenance.

Lessons from the FaST Project

FaST (Farm Sustainability Tool) is the previous project for farmers under the CAP, whose lessons guided the architectural choices of NutriPlatform.
Decisions retained from FaST: The **SPA architecture** was validated and maintained for its simplicity and efficiency, and the **Vue.js + Quasar frontend** is a proven technical stack that has been largely retained.
Evolutions compared to FaST: Several major changes have been implemented. The project moved from **JavaScript to TypeScript** to improve code quality by catching errors at compile time, while facilitating collaboration and maintenance thanks to static typing and smooth integration with modern tools. **Apollo was replaced by URQL** to evaluate a more lightweight solution (-40% bundle). **Leaflet was replaced by MapLibre**, which offers superior performance thanks to WebGL rendering and native support for vector tiles. **Django integration was removed**, as the Django Admin part was not needed given the small number of admin screens and backoffice features. The project moved from a **multi-repo to a monorepo** to accelerate development and facilitate future handover. Finally, the infrastructure was drastically simplified, moving from **Kubernetes to Docker Compose/Swarm**, with the frontend now a simple Docker container.

Vue.js 3 + Composition API

Vue.js 3 with the Composition API was chosen for several reasons. It ensures **interface reactivity**, allowing real-time updates of farm data during NutriModel calculations. **Performance** is optimized thanks to a lightweight bundle, suitable for limited connections in agricultural settings. The **mature Vue.js ecosystem** guarantees long-term support for this European project. Its popularity facilitates **international collaboration** among partners. Finally, it ensures **continuity with FaST**, allowing the expertise gained on Vue.js in the previous project to be maintained.

GraphQL and URQL Client

URQL was chosen as the GraphQL client to replace Apollo Client, offering a lighter solution tailored to the project's needs.

MapLibre Mapping

MapLibre GL JS is the open-source mapping solution chosen for visualizing agricultural plots and geospatial data.

Vite Build and Optimizations

Vite serves as a modern build tool to optimize performance and development of the frontend application.

3.2.3 Quasar Framework

Architectural Choice

Quasar Framework was chosen for its advantages in the European agricultural context. It enables a **multi-device approach**, offering an interface suitable for tablets used in the field by farmers, as well as desktops for advisors and authorities. Its **UI components** accelerate development, which is crucial for meeting EU timelines. Quasar also ensures a **professional design**, providing a credible interface

for regional authorities and policy makers. Finally, it contributes to **reduced maintenance** thanks to less custom code, which is an asset for a geographically distributed team.

Essential Configuration

Application Mode: SPA (Single Page Application) with router in History mode
Enabled Plugins: Enabled plugins include the **Loading Plugin** for global loading indicators, the **Dialog Plugin** for system modal dialogs, and the **Notify Plugin** for user toast notifications.

CSS/SASS Customization

Style Architecture: NutriPlatform uses a hybrid approach combining Quasar UI components with custom SASS styles to meet the specific needs of the agricultural domain and implement the NutriBudget graphic charter. Quasar styles are customized via SASS variables defined in quasar.variables.scss. On the other hand, application styles for shared business components in the frontend are described in app.scss. This approach leverages the robustness of Quasar components while maintaining a visual identity specific to the nutrient management domain.

3.2.4 Environment Variables

Centralized Configuration

The configuration envFolder: "../../../" enables sharing variables between all services in the monorepo.

Main variables: Main variables include VITE_GRAPHQL_API_URL for the Hasura API URL, VITE_KEYCLOAK_* for authentication configuration, and VITE_MAP_TILER_API_KEY for the mapping services API key.

3.3 State Management

3.3.1 State Architecture

The NutriPlatform application uses a **modern and decentralized** approach to state management, favouring the Composition API and URQL for server data.

Adopted Strategy

The adopted strategy for state management is based on several pillars. **Local state** is managed by the Composition API (ref, reactive) for the user interface and interactions. **Server state** is handled by URQL, which integrates a cache for GraphQL data. **Minimal global state** is provided by Pinia, which is configured but used only when strictly necessary. Finally, **authentication** is managed by Keycloak, which provides a global context.

3.3.2 Rationale for Choices

URQL vs Apollo Client

The choice of URQL is explained by the specific constraints of the European agricultural context and the need for performance in rural areas. URQL offers a **lightweight bundle**, reducing size by 40%, which is crucial for rural areas with low bandwidth. Its **smart cache** is optimized for frequently accessed agricultural data, such as information on farms and fields. The **simplicity of URQL configuration** reduces overall system complexity. In addition, its **native TypeScript support** ensures type safety for critical data, such as nutrients and NutriModel calculations.

Hybrid Architecture (local + server + minimal global)

This architectural approach meets the needs of the NutriBudget project and optimizes the separation of responsibilities. It ensures a **clear separation of responsibilities**, with UI state managed locally, business data via URQL, and global configuration via Pinia. In terms of **performance**, it avoids unnecessary re-renders, especially during NutriModel calculations. **Network resilience** is improved thanks to the URQL cache, which keeps the application functional even in case of temporary disconnection. Finally, this architecture is designed for **scalability**, being ready to integrate future features such as offline mode or synchronization.

3.3.3 Server Data Management with URQL

Advantages of this Approach

This approach offers several advantages. It provides **automatic caching**, with URQL managing the cache of GraphQL queries. It ensures **native reactivity** thanks to its direct integration with the Vue 3 Composition API. **Error management** is simplified, with loading and error states automatically handled. Finally, it integrates **optimizations** such as request deduplication and smart caching.

3.3.4 Local State and Composition API

Local state is used to complement server data state (via URQL). It has the following advantages:

1. **Separation of responsibilities**: Local state manages UI interactions (navigation, dialogs, selections) while URQL manages business data
2. **Performance**: Avoids unnecessary re-renders by keeping UI state separate from server data
3. **Maintainability**: Consistent pattern throughout the application for user interactions
4. **Native reactivity**: Optimal use of Vue 3 Composition API without overhead

This **hybrid approach** (local state + server cache) is a deliberate architectural choice to optimize the user experience in the agricultural context.

3.3.5 Global Authentication

Keycloak Integration
This Vue 3 module provides native integration with Keycloak, offering:

- **Composition API** with the useKeycloak() hook to access authentication context
- **Automatic reactivity**, allowing the UI to update when authentication state changes
- **Flexible configuration**, supporting different Keycloak initialization modes
- **Native TypeScript support**, providing complete types for safe integration

Official documentation: @dsb-norge/vue-keycloak-js

The advantages of this integration are multiple. It provides a **global context** accessible in all components. It ensures **reactivity** with automatic UI updates. Finally, it guarantees **security** by automatically injecting the token into URQL requests.

3.3.6 Use of Pinia

Official documentation: Pinia - The Vue Store
Role of Pinia in NutriPlatform:
Pinia is the **official state manager** for Vue 3
In the NutriPlatform project, its potential use cases include user preferences, global configuration, and state shared between different business entities.

3.4 GraphQL Integration

3.4.1 URQL GraphQL Client

The application uses **URQL** as the GraphQL client for modern integration with Vue 3.

Configured Exchanges (URQL plugins)

The **configured exchanges** (URQL plugins) include **cacheExchange** for automatic query caching, **authExchange** for automatic Keycloak authentication management, and **fetchExchange** for executing HTTP requests.

3.4.2 Query Patterns

GraphQL queries in NutriPlatform follow standardized patterns. There are **basic queries** for retrieving lists of entities (farms, nutriplans), **queries with variables** for retrieving specific entities by ID, and **reactive usage** thanks to integration with the Vue 3 Composition API.

3.4.3 URQL-Keycloak Integration

The integration between URQL and Keycloak enables **automatic authentication**, where each GraphQL request automatically includes the JWT token. It facilitates **Hasura role management**, with the X-Hasura-User-Id header allowing for permission enforcement. It ensures **efficient error handling** with automatic detection of JWT errors and token refresh. Finally, it integrates a **smart cache** that optimizes performance by avoiding redundant requests. This integration uses URQL's authExchange to automatically handle authentication token injection and transparent refresh of expired sessions.

3.4.4 Rationale for Choices

URQL vs Apollo Client

The choice of URQL is justified in the context of European agricultural data and rural connectivity constraints. URQL offers **increased performance** with a bundle 60% lighter, which is crucial for rural areas with low bandwidth. Its **smart cache** is optimized for complex relational data (farm → fields → crops). Finally, its **native TypeScript support** allows for automatic type generation, ensuring the safety of nutritional calculations.

GraphQL vs REST

The architectural motivations behind the choice of GraphQL are multiple. It allows for **precise queries**, avoiding over-fetching for heavy geospatial data (plots, maps). It facilitates the management of **complex relationships**, enabling retrieval of a farm's data, its fields, crops, and nutritional analyses in a single request. It improves **mobile performance** by reducing the number of network calls in the field, where connections can be unstable. Finally, it offers great **scalability** thanks to a flexible schema for adding new NutriKPIs and agricultural indicators.

3.4.5 GraphQL Fragments

A **fragment** is a reusable piece of a GraphQL query that defines a set of fields for a given type. In a Vue.js component architecture, fragments offer several advantages. In a Vue.js component architecture, fragments offer several **architectural advantages**. They enable **component-data coupling** where each component defines its own data needs. They promote **reusability**, as a fragment can be used in multiple queries. They simplify **maintenance** through centralized modification of the fields required by a component. Finally, they guarantee **type safety** with automatic validation of requested fields.

NutriPlatform usage example: Each component (such as FarmList) defines its fragment corresponding to its display needs.

3.4.6 GraphQL Mutations

A **mutation** is a GraphQL operation that modifies data on the server (create, update, delete).
Mutations use **typed variables** for automatic validation of input parameters. They can include a **data return**, allowing retrieval of modified data. Finally, they guarantee **atomicity**, with the operation being complete or failing entirely.
Mutations in NutriPlatform follow standardized patterns for:

- **Entity creation** (adding new fields, farms, crops)
- **Update** (modifying existing data)
- **Deletion** of items with relationship management
- **Error handling** with user feedback via notifications

3.4.7 Error Handling

Error handling is structured for different types of problems. **Query errors** are managed by conditional display with error banners. **Mutation errors** are handled via try-catch blocks with toast notifications. Finally, **authentication errors** are automatically managed via URQL's authExchange.

3.4.8 Cache and Performance

The **usage of caching** ensures **reactivity** with instant display of already retrieved data. It enables **network savings** by reducing redundant API calls. Finally, it offers a **smooth experience** with latency-free navigation between pages.

Available URQL strategies include cache-first (default), which uses the cache if available and makes a network request otherwise. cache-only uses only the cache and returns an error if data is missing. network-only forces a new network request for manual data refresh. Finally, cache-and-network displays the cache then updates with network data.
NutriPlatform usage example: Cache-first for fast navigation, network-only for manual refresh of farm data.

3.5 Mapping Components

3.5.1 Technology Choices

MapLibre GL JS - Chosen Solution

The **main motivations** for choosing MapLibre GL JS are multiple. Its **open-source license** makes it compatible with the EU budget and allows free redistribution. It offers **optimal performance** thanks to its WebGL vector rendering, ideal for large amounts of data points. It represents an **evolution from FaST**, marking a migration from Leaflet to MapLibre for superior performance and native vector tile management. Finally, its **modern integration** fits perfectly with the Vue.js + TypeScript ecosystem.

Comparison with Leaflet and OpenLayers, other open-source solutions :

Compared to Leaflet and OpenLayers, other open-source solutions, MapLibre stands out for its **advanced graphic performance**. Thanks to WebGL, it offers smooth and efficient rendering of vector tiles, ideal for dynamic and interactive maps, unlike Leaflet which relies on simpler raster tiles. In terms of **customization and styling**, MapLibre supports Mapbox styles and allows fine customization of map layers, whereas OpenLayers requires more configuration for an equivalent result. Regarding **complexity and learning curve**, MapLibre is more demanding to integrate and configure, especially

for developers unfamiliar with WebGL, while Leaflet remains the most accessible solution and OpenLayers the most complete for advanced GIS cases.

3.5.1 Map services provider

MapTiler Services

The **services used** from MapTiler Cloud include **satellite tiles** for the high-resolution imagery needed for plot analysis, **geocoding** for precise farm localization via API, and **map styles** for backgrounds adapted to the agricultural context.
Dependency management for MapTiler Cloud involves **secure configuration** via VITE_MAP_TILER_API_KEY, **quota management**, and usage-based billing. **Alternatives** such as OpenStreetMap or self-hosted services are also considered.
Documentation: MapTiler

3.5.2 Geospatial Data

GeoJSON Format

Exchange standard for web geographic data, used for:
The GeoJSON format is the **exchange standard** for web geographic data, used for **storing farm and field coordinates**, **exchanging with the PostGIS database**, **displaying NUTS 2 regions**, and its **native compatibility** with MapLibre GL JS.

PostGIS Integration

PostGIS integration enables **spatial queries** via GraphQL/Hasura, **geometric transformations** at the database level, and **optimizations** for increased performance on large agricultural areas.

3.6 Authentication

3.6.1 Keycloak Integration

NutriPlatform uses the official Keycloak.js client in the frontend. Login, account creation, or forgotten password features are delegated to Keycloak: the screens are served by the Keycloak server and not by the NutriPlatform frontend.

The useKeycloak composable provides the main interface for accessing authentication information such as authenticated status, subject, userName, roles, and the JWT token.
Official documentation: https://www.keycloak.org/

Authentication Flows

Login Sequence:

The **Login Sequence** starts with **initialization** where initKeycloak() checks the SSO state. Next, a **check** in login-required mode redirects to the authentication form served by Keycloak. If the user is authenticated, the **JWT token** is retrieved. **Headers** are automatically injected into GraphQL requests via URQL. Finally, Hasura **validates** the token with Keycloak.

Refresh Sequence:

The **Refresh Sequence** begins with the **detection** of an invalid-jwt error in a GraphQL request. This is followed by a **refresh** of the token via updateToken() to Keycloak. A new attempt of the request is made with the fresh token (**retry**). If the refresh fails, a **fallback** logs the user out.

Environment Variables

Keycloak environment variables configure the connection.
VITE_KEYCLOAK_REALM specifies the name of the realm containing users and configurations.
VITE_KEYCLOAK_URL indicates the base URL of the Keycloak server. Finally,
VITE_KEYCLOAK_CLIENT_ID is the unique identifier of the frontend client.

3.6.2 Role and Permission Management

Role Definition

NutriPlatform uses a very simple model of roles and permissions that are managed and made available by Keycloak.

The roles of **Farmer** and **Farm advisor** provide access to the main module for managing and optimizing a farm's nutrients. In this first version of NutriPlatform, they are treated the same way because there is no concept of farm ownership sharing; each user can only interact with their own farms and cannot give visibility to a third party.

The **Policy maker** role provides access to nutrient visualization and roadmaps at the European level.

Roles are not exclusive; they can be combined to give access to the entire application.

Route Protection with Vue Router

Vue Router is the official router for Vue.js, enabling the creation of single-page applications (SPA) with client-side navigation. It manages the mapping between URLs and Vue components, allowing smooth navigation without page reloads.
Official documentation: Vue Router
The router manages access to different modules of the application according to the user's roles.

To ensure security, all sensitive routes require authentication via Keycloak. The system systematically checks the JWT token, automatically redirecting unauthenticated users to the login page.

The architecture clearly distinguishes public from private routes using metadata, ensuring smooth integration with Keycloak and Vue. The user benefits from navigation without reloads, smart redirections, and retention of the target URL after login.

3.7 User Experience

3.7.1 Overview

The user experience of NutriPlatform is designed to meet the specific needs of the various stakeholders in the European agricultural sector, with a focus on simplicity and accessibility. Detailed UI mockups document and illustrate the user experience of the project, providing a visual reference for implementation.

3.7.2 Responsive Design and Graphic Charter

The application employs responsive design principles to ensure optimal performance on different devices. It integrates state-of-the-art user experience (UX) methodologies, with Figma screens designed by professionals, using the NutriBudget graphic charter, to guarantee a user-friendly and accessible interface. The application is compatible with desktops, tablets, and smartphones via up-to-date standard browsers for a smooth user experience.

Multi-Device Approach:

The application's **Multi-Device Approach** is available in several formats. For **Desktop**, it offers a complete interface with advanced navigation, intended for advisors and decision-makers. On **tablets**, the display is optimized for field use by farmers. Finally, **smartphones** provide mobile access to essential features.

NutriBudget Graphic Charter:

The **NutriBudget Graphic Charter** ensures **visual consistency** through strict application of the European project's visual identity. It guarantees **professionalism** with Figma design by UX/UI experts. Finally, it aims for **accessibility** by complying with WCAG standards for all users.

NutriPlatform Visual Identity:

The **NutriPlatform Visual Identity** is defined by the **agricultural green** (#13622C) as the main color, the **accent yellow** (#FAB71B) for accent elements, and the main font **Lato** for optimized readability.

3.7.3 Data Visualization

Chart.js was selected as the visualization solution for NutriPlatform due to its simplicity and stability, perfectly suited to the relatively simple needs of the project.

NutriPlatform's specific needs for visualization include **bar and column charts** for comparing budgets by nutrient, and **radar charts** for multi-dimensional visualization of NutriKPIs.
The **motivations for choosing** Chart.js are multiple. Its **ease of integration** is due to an intuitive API and minimal configuration. Its **proven stability** makes it a mature solution with active maintenance. Its **optimized performance** is suitable for rural connectivity constraints. Finally, the **limited needs** of the project do not require complex or advanced interactive visualizations.

3.7.4 Accessibility

WCAG AA Standards are met through several aspects. **Text element contrasts** have sufficient ratios. **Keyboard navigation** offers full accessibility. **Multilingual support** includes automatic detection and dynamic language switching. Finally, **specific adaptations** are planned for colour blindness, reduced mobility, and low vision.

3.7.5 Performance and UX Optimizations

Loading optimizations include **Lazy Loading**, where pages and components are loaded on demand. **Progress indicators** (spinners and bars) are used for long operations. Finally, an **optimized bundle** reduces the application size for areas with limited connectivity.

3.7.6 Map Interactions

Intuitive navigation is facilitated by **standard controls** (zoom, pan, rotation) familiar to users. **Geocoding** enables integrated address search. **Geolocation** ensures automatic centering on the user's position.
Agricultural interactions include **farm visualization**, offering a spatial representation of the farm
Finally, **specific visualizations for Policy Makers** present the nutrient balance at the European / NUTS 2 level.

3.7.7 Notifications and Error Management

The application implements robust error management at several levels with appropriate user feedback.

Strategies by error type are defined for each scenario.

- **GraphQL errors** are automatically managed via URQL with conditional display.
- **Mutation errors** follow a try-catch pattern with toast notifications.
- **External API errors** (notably for the NutriModel API) benefit from specific management with retry.
- **Authentication errors** are automatically managed via URQL's authExchange.

User Feedback

Quasar notifications provide clear visual feedback: red notifications with error icons for **errors**, green notifications with validation icons for **successes**, and orange notifications for **warnings** indicating situations to watch.

Status Banners:

Status banners are used for various situations. For **loading**, spinners with informative messages are displayed. In case of **errors**, persistent banners detail the problem. Finally, for **empty states**, messages guide the user to appropriate actions.

3.8 Internationalization

3.8.1 Overview

Internationalization of the **NutriPlatform** is managed by the **vue3-gettext** library, a solution that, despite its name, has been adapted to offer a simple and centralized workflow while retaining the semantic advantages of gettext.

3.8.2 Technological Choice: vue3-gettext

Context and Justification

The choice of **vue3-gettext** is based on a comparative analysis and experience gained from previous projects such as **FaST,** as listed in Table 3.

Table 3. Localization management: gettext vs. I18n approaches

	vue3-gettext (Nutriplatform implementation)	vue-i18n (classic approach)
Translation keys	English messages as keys	Abstract keys (e.g. user.name)
Code readability	$gettext('Save farm data')	$t('farm.save.data')
File format	Single, centralized JSON file	Message files per language
Ecosystem	Standard and easy to handle	Specific to the Vue.js ecosystem
Maintenance	Centralization simplifies adding keys	Management of multiple files
FaST experience	Continuity of the gettext philosophy	Not used

The main advantage of the current implementation is its **simplicity**. By centralizing all translations in a single JSON file, it eliminates the complexity of managing multiple .po files and their compilation (.mo). This approach is particularly effective for agile development teams.

Improved code Readability:

Using English text as the translation key makes the source code immediately understandable without needing to consult mapping files. The semantics are clear: {{ $gettext('My farms') }} is more explicit than {{ $t('home.farms.title') }}.

3.8.3 i18n Architecture

Language Scalability

The translation architecture is designed to be **highly scalable**. Adding a new language is a simple process that does not require any changes to the application's source code.
Thanks to the **centralization of all translations** in the single translations.json file, integrating a new language boils down to two main steps:

1. **Add a new language entry** in the translations.json file (e.g., "de" for German).
2. **Provide translations** for the existing keys.

This process can be significantly accelerated by using **automatic translation services (AI)** to generate a first version of the translations, which can then be reviewed and refined by native speakers. This flexible approach allows **NutriPlatform** to quickly adapt to the needs of new markets and regions without imposing additional technical burden.

Vue.js Integration

The **vue3-gettext** plugin is configured in a Quasar boot file.
This file initializes the library with translations, detects the user's language, and configures a fallback language.

3.8.4 Translation Process

Automated Translation Workflow

The translation workflow is designed to be both simple for the developer and robust through automation. It relies on standard tools from the gettext ecosystem.

1. **Marking in the code**: The developer marks strings to be translated directly in the Vue.js source code using functions provided by vue3-gettext, such as $gettext() or <translate>. For example: {{ $gettext('Welcome to the platform') }}.
2. **Automatic extraction**: The developer runs the command npm run gettext:extract. This script scans the entire source code (.vue, .js, .ts), detects all strings marked for translation, and automatically updates the src/language/translations.json file. New strings are added, ready to be translated.
3. **Translation**: Translators (or developers) complete the missing values for each language directly in the translations.json file.
4. **Compilation (Validation)**: The command npm run gettext:compile can be used to validate the JSON file syntax and prepare it for production, although in the current configuration, the JSON file is consumed directly.
5. **Validation and Versioning**: The updated translations.json file is then committed to Git, ensuring perfect synchronization between the code and available translations.

This process ensures that **all strings displayed in the interface are systematically extracted** and ready to be translated, eliminating the risk of omissions and the need for tedious manual maintenance of the translation file.

3.8.5 UX Considerations

Cultural Adaptation

Cultural adaptation considerations remain relevant:

- **Regional Formats**: Date, number, and unit formats must be managed, potentially via libraries like date-fns that support localization.
- **Text Length**: The interface must be designed to accommodate variations in text length between languages.

Multilingual Accessibility

- **Technical Support**: **UTF-8** encoding is used to ensure compatibility with all special characters.
- **Navigation**: A clear and accessible language selector is provided, and the user's choice is retained between sessions.

3.9 Build and Deployment

3.9.1 Build Process

Build Tool: Vite

Vite is a modern and fast build tool for web applications, developed by the creator of Vue.js. It uses native ES modules in the browser for development and Rollup for production. Vite offers several significant advantages: it enables **instant startup** of the development server and **ultra-fast Hot Module Replacement (HMR)**. The **build is optimized for production**, and it provides **native support** for TypeScript and Vue.js. Finally, it benefits from a **rich plugin ecosystem**.

Official documentation: https://vitejs.dev/

Build Configuration

The **build configuration** defines the compilation and optimization parameters for the NutriPlatform application. It ensures compatibility with the target users' browsers, support for TypeScript and .vue files, and performance optimization.

Performance Optimizations

Performance optimizations are crucial for rural areas with limited connectivity, enabling a smoother user experience under degraded network conditions. These optimizations include:

- **Tree Shaking**, which is the automatic elimination of unused code to reduce bundle size.
- **Code Splitting** allows on-demand loading of pages to improve initial load times. Finally,
- **Assets Optimization** includes compression of images, CSS, and JavaScript to reduce asset size.

3.9.2 Environment Variable Management

Centralized Configuration

The configuration envFolder: "../../../" points to the root of the monorepo, allowing environment variables to be shared between all services. This offers several advantages: **centralized configuration**, **consistency between environments**, **simplified secret management**, and **ease of Docker deployment**.

Main Variables

The **main environment variables** include the Hasura API URL for **GraphQL**, authentication configuration (realm, URL, client ID) for **Keycloak**, the API key for **MapTiler** mapping services, and the development port for the **Frontend**. These variables allow configuration according to the environment without modifying the source code.

3.9.3 Deployment with Docker

Docker is a containerization platform that allows packaging an application and its dependencies into a lightweight, portable container. Using Docker offers several **advantages for NutriPlatform**: it ensures

consistent deployment across all environments, **service isolation** (frontend, backend, database), facilitates **scaling**, and **simplifies dependency management**.

Official documentation: https://docs.docker.com/

Production Dockerfile

The Dockerfile uses a multi-stage build. **Stage 1 - Build** uses Node.js 22 Alpine to compile the application with environment variable injection. **Stage 2 - Production** uses Nginx Alpine to serve static files. This approach results in a lightweight final image, enhanced security, and optimized performance.

Nginx Configuration

Nginx is a high-performance web server, reverse proxy, and load balancer, known for its stability and low resource consumption. **Nginx usage in NutriPlatform** is multifaceted. It acts as a **static file server**, serving the HTML, CSS, and JS files of the Vue.js application. It manages **SPA routing** by redirecting all routes to index.html for client-side routing. It integrates **optimizations** such as gzip compression, asset caching, and security headers.
Finally, it ensures **high performance** through efficient management of concurrent connections, which is crucial for agricultural users.

Official documentation: https://nginx.org/en/docs/

4 Backend Component – Hasura

4.1 Overview

The **Hasura GraphQL Engine** occupies a central place in the NutriPlatform architecture, acting as a **single entry point** for all data operations between the frontend (Vue.js) and the **PostgreSQL** database enhanced by the **PostGIS** extension. This open-source component instantly transforms the relational database schema into a **complete, performant, and secure GraphQL API**, tailored to the requirements of integrated nutrient management at the European scale.

4.1.1 Introduction and Fundamental Principles

Hasura is designed to automate the creation of real-time GraphQL APIs from any PostgreSQL database. It is based on a **cloud-native architecture** and naturally integrates into containerized environments via **Docker**. In NutriBudget, Hasura is deployed in version 2 (Community Edition), ensuring full control over the infrastructure, no dependency on a third-party cloud service, and optimal compatibility with the project's data sovereignty requirements.

One of Hasura's major strengths lies in its ability to automatically generate GraphQL schemas from the database structure, while offering advanced **security** and **permission management** mechanisms. Access rights are defined granularly, at the level of roles, rows, and columns, allowing precise adaptation of data visibility and modification according to user profiles (farmers, advisors, authorities, etc.).

4.1.2 Role in the NutriPlatform Architecture

In NutriPlatform, Hasura acts as an API Gateway: it centralizes all data requests, orchestrates authentication and authorization via validation of JWT tokens issued by Keycloak, and dynamically applies the security rules associated with each user. This native integration with Keycloak ensures compliance with European standards for identity management and data protection.

Hasura is not limited to data exposure: it also allows **business logic extension** through the concept of **Custom Actions**. These actions serve as a gateway to external services (for example, the Nutriplatform-services developed in TypeScript with Fastify) or third-party APIs, while maintaining a unified access point for the frontend. **Events** and **webhooks** facilitate the implementation of asynchronous workflows and integration with other systems (e.g., triggering calculations, notifications, data synchronization).

The **PostGIS** extension integrated into PostgreSQL enables Hasura to natively support **geospatial data**. This capability is essential for mapping agricultural holdings, spatial analysis of nutrient flows, and producing environmental indicators at different scales (plot, region, EU).

4.1.3 Advantages for NutriPlatform

The adoption of Hasura in NutriPlatform meets several strategic objectives:

- **Development acceleration**: automatic API generation significantly reduces implementation time and facilitates the evolution of the data model.
- **Security and compliance**: fine-grained permission management, JWT/Keycloak integration, and native role support ensure robust access control.
- **Scalability**: the cloud-native architecture and compatibility with Docker Swarm allow the platform to scale from local pilots to the European level.
- **Interoperability**: schema federation (Remote Schemas) and integration of custom actions open the platform to external services (weather, agronomic models, etc.).
- **Geospatial support**: using PostGIS enables full exploitation of spatial data, a key issue for rational nutrient management.

4.1.4 Version Choice and Positioning

The NutriPlatform project has deliberately chosen to rely on **Hasura v2 Community Edition** rather than version v3/DDN (Data Delivery Network). This choice is motivated by the desire to maintain an **open-source**, self-hosted, mature solution with no license cost, while avoiding dependency on proprietary cloud infrastructure. Version v2 offers all the features required by the project: API generation, advanced permission management, custom actions, PostGIS support, and smooth Docker integration.

4.1.5 Technical Architecture and Integration

Technically, Hasura functions as an **intelligent proxy** between the frontend and the database. It validates JWT tokens provided by Keycloak, extracts roles and user identifiers, then dynamically applies the access rules defined in the configuration. Custom actions allow integration of project-specific business logic, while events facilitate task automation and interoperability with other platform components.
The whole is orchestrated in a **Dockerized environment**, with fine-tuned environment variable configuration to ensure portability, security, and reproducibility of deployments.

4.2 Architecture

The architecture of the **Hasura GraphQL Engine** within NutriPlatform is based on a strategy of **systematic containerization**: the development environment uses **Docker Compose** for simplicity and speed, while production relies on **Docker Swarm** for orchestration, scaling, and high availability. This choice, validated by the experience of the FaST project, ensures **maximum portability** between environments, **reproducibility** of deployments, **enhanced security** through strict service isolation, and **easier maintenance**. Containerization also enables automated dependency management, simplifies updates, and ensures infrastructure robustness as the project evolves.

The architecture of the **Hasura GraphQL Engine** within NutriPlatform is based on a **cloud-native** and modular approach, orchestrated using **Docker Compose** to guarantee portability, reproducibility, and infrastructure security. This setup allows deployment of all critical platform components in isolated containers, while ensuring their smooth interconnection and scalability.

At the heart of this architecture, the **Hasura GraphQL Engine** acts as the main access point to data, exposing a unified and secure GraphQL API. It is directly connected to a **PostgreSQL** database enhanced by the **PostGIS** extension, enabling efficient management of the relational and geospatial data required for integrated nutrient management.

The **NutriPlatform frontend**, served by a **Nginx** container, interacts with Hasura for all data operations and with **Keycloak** for user authentication. Keycloak, deployed in its own container, manages all identities, roles, and permissions, and issues the **JWT tokens** used by Hasura to dynamically apply access rules.

Specific business logic that cannot be handled directly by Hasura is implemented in the **NutriPlatform-services**. These services, developed in **TypeScript** with **Fastify**, are also containerized and interact with Hasura via the **Actions** and **Webhooks** mechanisms. This separation of responsibilities ensures a flexible, scalable, and easy-to-maintain architecture.

This entire containerized infrastructure offers **great flexibility**, **strict service isolation**, and facilitates both deployment, scaling, and maintenance of the platform. This architectural choice is fully aligned with the NutriBudget strategy: to offer a robust, secure solution adaptable to the varied needs of sustainable nutrient management stakeholders in Europe.

These paragraphs describe the architecture of the **Hasura GraphQL Engine** in NutriPlatform, detailing service orchestration, security management, infrastructure modularity, integration with other

components (frontend, Keycloak, NutriPlatform-services), and the expected benefits of this approach. The whole faithfully reflects the structure and technical choices implemented in the project.

4.3 PostGIS Configuration

The integration of **PostGIS** within the **PostgreSQL** database is a fundamental pillar of the NutriPlatform architecture. This extension, renowned for its robustness and functional richness, gives the platform the ability to store, index, and query complex **geospatial data**, thus meeting the specific needs of rational nutrient management at different spatial scales.

The PostGIS configuration is fully automated in the NutriBudget Docker environment. When the PostgreSQL container starts, an initialization script runs to create the necessary extensions: postgis, postgis_topology, and postgis_tiger_geocoder. This automation ensures deployment reproducibility and environment consistency, while facilitating platform maintenance and evolution.

PostGIS enables the management of geographic objects such as **farms**. Thanks to native integration with **Hasura**, it becomes possible to expose this data via the GraphQL API, perform **advanced spatial queries** (area calculation, proximity search, topography analysis), and cross spatial information with other dimensions of the data model.

Note: At this stage of the project, the effective use of PostGIS in NutriPlatform is mainly focused on storing the geographic locations of agricultural entities. Concretely, **farms** are represented by points (GPS coordinates). The advanced features of spatial queries and topographic analyses are made possible by the configuration, but their use is currently limited to managing these basic geometries. However, this architecture leaves the door open to more complex uses as the platform evolves.

4.4 Permissions and Security

The management of **permissions** and **security** in NutriPlatform relies on a combination of advanced mechanisms, ensuring both confidentiality, integrity, and strict data isolation for each user. The architecture is based on Hasura's role model, JWT authentication via Keycloak, and fine-grained access rights, tailored to the requirements of a multi-actor system.

The main role currently implemented is **farmer**. This role is configured so that each user can only access their own data: farms, fields, model runs, and all associated entities. Access rules are defined at the row and column level, ensuring that data visibility and modification are strictly limited to the authenticated user.

Authentication is provided by **Keycloak**, which manages all identities and roles. When a user logs in, Keycloak issues a signed **JSON Web Token (JWT)** containing role information and the user's unique identifier. This token is sent to Hasura with each GraphQL request: Hasura validates the signature, extracts the relevant claims (x-hasura-allowed-roles, x-hasura-user-id), and dynamically applies the permission rules defined in the configuration.

This setup enables **strict data isolation**: each farmer can only view, modify, or delete the farms, nutriplans, and model runs explicitly assigned to them. Relationships between entities (farm ↔ nutriplan, nutriplan ↔ NutriBudget runs) automatically inherit the owner's permissions, ensuring logical and secure data separation even within a shared database.

The JWT approach offers several advantages:

- **Enhanced security** through systematic token signing and validation.
- **Decoupling** between authentication (managed by Keycloak) and authorization (managed by Hasura), simplifying system maintenance and evolution.
- **Scalability**: tokens are self-contained, avoiding repeated requests to Keycloak.
- **Flexibility**: centralized user and role management makes it easy to adapt the security policy as project needs evolve.

In summary, NutriPlatform's security strategy is based on **maximum permission granularity**, **strict per-user data isolation**, and **native integration** between Keycloak and Hasura, ensuring compliance, trust, and robustness for all project stakeholders.

4.5 Custom Actions

Custom Actions are a central mechanism for extending the capabilities of the **Hasura GraphQL Engine** within NutriPlatform. They allow the integration of **advanced business logic** and interaction with **external services** while maintaining a unified GraphQL API for the frontend.

In the NutriBudget architecture, Actions are used to delegate complex operations—calculations, validations, workflow orchestration—to specialized services, such as the **NutriPlatform-services** developed in TypeScript with Fastify. When an Action is defined in Hasura, it is associated with an HTTP endpoint (webhook): Hasura forwards the request to this external service, which executes the business logic and returns the result, which is then exposed as a standard GraphQL mutation or query.

This setup promotes **modularity** and **maintainability** of the platform: business logic evolves independently of the database schema, and the integration of new features or third-party services (APIs, microservices, serverless functions) is achieved without disruption for the frontend.

4.6 Performance Optimizations

This section presents the choices and performance practices actually in place for the **Hasura GraphQL Engine** of the **NutriPlatform**, in line with the real state of the code and database. The objective is to ensure stable response times, controlled load, and a progressive scalability trajectory without introducing excessive operational complexity.

4.6.1 Decision: No need for additional caching for Hasura

NutriPlatform does not use additional caching such as **Redis** for caching or queuing at the **Hasura** component level. This decision covers GraphQL read/write traffic, **Hasura Actions** (configured in synchronous mode), and any occasional webhooks. This choice favors **operational simplicity**, **reduced infrastructure complexity**, and **alignment with current needs**: the expected load and query profiles do not justify adding a distributed cache. Optimizations focus on the native capabilities of **PostgreSQL/PostGIS** and **lean GraphQL query modeling**, with a **client-side cache**.

4.6.2 Context and Performance Objectives

NutriPlatform relies on **Hasura** as the GraphQL API layer and **PostgreSQL** with **PostGIS** for persistence and geospatial capabilities. Current use cases involve managing farms, plots, and NutriBudget model runs, with volumes and query profiles compatible with a "database-first" optimization (rigorous schema, targeted queries, reduced client-side over-calling). The performance strategy explicitly targets simplicity, robustness, and gradual scalability.

4.6.3 Currently Applied Optimizations

The state of the code and migrations confirms structural optimizations on the database side and lean practices on the API/client side.

Effective PostgreSQL/PostGIS Optimizations

- Constrained relational schema with **primary keys** on all tables and **systematic foreign keys**, which limits unintended multiplicities and simplifies join plans.

- **Geometry columns** (SRID 4326 for farm points enabling spatial queries on robust foundations (PostGIS). PostGIS activation is ensured at the image and extension level, and types are present in the schema.
- **JSONB columns** for model run inputs/outputs (nutribudget_run table), reducing the need for intermediate tables or additional joins for these non-relational loads.
- Simple application trigger for updating the "updated_at" field on the farm table, ensuring implicit traceability of changes.

GraphQL/Hasura Patterns and Overcalling Reduction

- The nature of **GraphQL** allows for **strict selection of fields** actually consumed by the frontend, avoiding overfetching.
- Queries are designed to remain user-targeted (logical isolation described in the security/permissions page), limiting the scope of result sets.
- **Hasura Actions** are configured in synchronous mode (see CLI configuration), eliminating the need for a specific internal queue for most current interactions.
- Systematic implementation of **server-side filters** (where conditions and permission rules) to limit returned sets and avoid unnecessary full scans.
- **Explicit pagination**: offset/limit by default; cursor pagination (**keyset**) when the sort order and a strictly increasing identifier allow, to avoid costly offsets.
- **Structuring of relationships** and **Hasura permissions** to avoid costly non-indexed joins; any multi-table query relies on indexed columns in PostgreSQL.

Client Side

- The frontend uses **URQL**, which provides a **lightweight and efficient client cache**, limiting redundant calls when required data has not changed during the page lifecycle.
- **Frontend revalidation policies** (stale-while-revalidate or targeted reloads) to synchronize data without multiplying network calls.

4.6.4 Future Considerations (Scalability and Scaling Up)

Several levers are identified to support the increase in data and users, without challenging the decision not to use Redis on the Hasura side:

- Relational indexing: creation of **B-Tree indexes** on filter keys, addition of **composite indexes** for multi-column queries; **GIN indexes** on **JSONB columns** if structured filters are applied.
- Spatial indexing: creation of **GiST** or **SP-GiST** indexes on geometry columns to speed up spatial filters; all index additions are guided by execution plan analysis (**EXPLAIN/ANALYZE**) on realistic datasets.
- Read/write: depending on load, adoption of **PostgreSQL read replicas** to offload reads, or vertical/horizontal scaling of instances.
- Hasura query governance: implementation of depth limits, allow-list and/or query collections if necessary to better control access patterns in production.
- **PostgreSQL tuning**: calibration of **autovacuum** (VACUUM/ANALYZE), adjustment of **work_mem** and **effective_cache_size** according to query profiles and available memory, without setting generic values.
- Asynchronous processing: when longer processes are required, prefer webhooks or dedicated jobs, tracked and idempotent.
- Observability: progressive instrumentation (metrics, traces) to guide optimization on measured points and avoid over-engineering.

These evolutions remain compatible with the scalability trajectory defined in the scalability page and with the security/permissions policy.

4.7 Migrations and Evolution

The management of **migrations** ensure controlled and traceable evolution of the database schema and Hasura metadata. This mechanism synchronizes the state of the database and the GraphQL API with the source code, while facilitating collaborative development and deployment across multiple environments.

A migration groups together SQL scripts (for the PostgreSQL schema) and YAML/JSON files (for Hasura metadata), precisely describing the changes to be applied. The **Hasura CLI** tool provides commands to create, apply, and manage these migrations, thus ensuring rigorous versioning.

The adoption of migrations offers many advantages:

- **Schema versioning** allows tracking the history of changes, auditing modifications, and ensuring consistency between environments (development, staging, production).
- **Collaborative development** is facilitated: each team member can apply the latest migrations to align their local database and merge their own changes in a structured way.
- **Deployments are reproducible**: sequential application of migrations ensures that each environment reaches the same state, eliminating configuration discrepancies.
- In case of problems, a quick **rollback** to a previous state is possible, minimizing risks and service interruptions.
- Integration into **CI/CD pipelines** allows automating database and metadata updates with each deployment, ensuring compatibility between the application and the data structure.

Schema versioning is thus at the heart of the stability, maintainability, and scalability of NutriPlatform. It enables managing evolutions, resolving conflicts, testing for regressions, and ensuring that all environments conform to a reference state. This practice is essential in an agile, distributed, and multi-actor development context such as NutriBudget.

4.8 External Service Integration

The integration of **external services** enriches the GraphQL API exposed by Hasura with data and features from multiple sources: microservices, third-party APIs, specialized business services, etc. Two main mechanisms are implemented: **Remote Schemas** and **Actions**.

Remote Schemas offer the possibility to federate several GraphQL APIs into a single unified interface. Hasura thus combines the schema generated from the PostgreSQL database with those of external services, allowing clients to query heterogeneous data via a single access point. This federation ensures **service decoupling**, **flexibility** in integrating new sources, and **consistency** of the developer experience.

Actions allow extending Hasura's GraphQL API by integrating custom HTTP endpoints, often implemented in Nutriplatform-services or third-party services. In NutriBudget, this mechanism is used to interact with:

- the NutriBudget API for modeling and computation,
- MapTiler for geocoding,

This approach makes Hasura a **unified gateway** for all data interactions, whether from the PostgreSQL database or external services, while maintaining the platform's security, traceability, and scalability.

The integration of external services via Remote Schemas and Actions is fully in line with the NutriBudget vision: to offer an open, interoperable, and scalable platform, capable of adapting to user needs and the rapidly evolving digital agricultural ecosystem.

5 Backend Component – NutriPlatform services

5.1 Overview

The **complementary backend services** of NutriPlatform are a pillar of the NutriBudget project's technical architecture. Their role is to extend the capabilities of the main backend, orchestrated by Hasura, by handling all business logic and specialized processing that cannot be directly expressed via the GraphQL engine.

Within NutriPlatform, these services are responsible for:

- Implementing **complex business logic**, especially processing that requires multiple steps, advanced validations, or interactions with external systems.
- **Integration with external APIs**, such as the NutriBudget API for running agronomic models
- Managing **specialized data processing**, including aggregation, transformation, or analysis of data from the PostgreSQL/PostGIS database.
- Handling **asynchronous events** and webhooks, enabling reactions to state changes or external notifications.

The architecture of these services is based on robust and proven technological choices:

- The **Fastify framework** (TypeScript) is used for its performance, modularity, and compatibility with modern Node.js ecosystem standards.
- The **Node.js 22 runtime** ensures compatibility with the latest language and library developments.
- Database access is via the **hasura-client**, a typed wrapper around Prisma, ensuring strict consistency with the PostgreSQL/PostGIS schema and operation security.
- HTTP calls to external APIs are made with Axios, while structured logging relies on Pino, natively integrated with Fastify.

5.2 Fastify Architecture

This section presents what **Fastify** is in the context of **NutriPlatform** and describes its configuration as actually implemented in the Nutriplatform-services backend service. The objective is to provide a clear understanding of Fastify's role, the architectural principles employed, the enabled plugins and hooks, route structuring, centralized error management, as well as performance, robustness, and security considerations.

5.2.1 Role and Positioning in the Backend Architecture

In NutriPlatform, **Fastify** serves as the foundation for the "nutriplatform-services" backend component. This component complements **Hasura** when needs exceed the purely declarative scope of permissions and GraphQL queries. Specifically, nutriplatform-services exposes **REST endpoints** that can be called either directly by internal clients or via **Hasura Actions** configured in synchronous mode. The two currently declared actions target processing associated with the NutriBudget scientific API and are visible in the reference Hasura configuration.

The architectural positioning is deliberately minimalist and focused on **orchestration**: the service receives the request, validates the payload, applies generic security controls, calls external systems (e.g., the NutriBudget API), structures the response, and logs accesses and errors. Heavy business logic and agronomic calculations remain externalized, preserving a lightweight and maintainable backend.

5.2.2 Key Fastify Principles

The implementation relies on the following idiomatic Fastify properties:

- **Plugins** encapsulate cross-cutting capabilities (CORS policy, security headers, rate limiting, optional metrics) and are registered centrally to ensure a clear separation of responsibilities.
- **Hooks** provide lightweight extension points, for example for authentication (pre-processing), response time measurement, and structured logging (post-processing).
- **Schemas and validation** of requests strengthen endpoint robustness: payloads are validated by JSON schemas attached to routes, limiting execution errors and contributing to security.
- **Encapsulation** and **modularization** of routes allow organizing the domain into coherent subsets (e.g., "nutribudget" routes), each module being loaded via Fastify's internal plugin API.

5.2.3 Actual Configuration in NutriPlatform-services

All configuration is centralized in the application build file and started by the server entry point. The following elements are in place in the code.

Plugins and Cross-Cutting Capabilities

- **CORS**: a dedicated plugin registers the CORS policy with a permissive configuration suitable for development and easily restricted for production. This configuration allows usual headers including **Authorization**.
- **HTTP Security (Helmet)**: **security headers** are applied globally via the official Helmet plugin, strengthening the service's default posture.
- **Rate Limiting**: a global **rate limiter** protects the service against abuse and brute-force attacks. The current configuration limits each IP to 100 requests per minute.
- **Prometheus Metrics (optional)**: exposure of a /metrics endpoint compatible with Prometheus is available and strictly controlled by an environment variable. By default, this feature is disabled. When enabled, it adds standard counters and histograms without impacting other routes.

Authentication and Context Propagation

A **JWT authentication middleware** based on **Keycloak** is applied globally in pre-processing (unless explicitly disabled in initialization options for test uses). It checks for a Bearer token, validates the signature and issuer via the **realm JWKS**, controls the expected audience, then **attaches a minimal user context** to the request to feed logging and possible downstream controls. This context includes, in particular, the identifier, username, and roles, when present in the JWT.

- The application of the authentication hook is configured via a global preHandler.
- **Structured logging** at the end of processing retrieves, if available, the user identifier to link access to an authenticated

For calls made via **Hasura Actions**, Fastify endpoints remain identical. When Authorization header propagation is configured in the calling infrastructure, the authentication middleware operates the same way; otherwise, exposure of these endpoints must be restricted to internal networks and/or protected by upstream controls as recommended in the security section.

Routing Structure and Domain Organization

Route loading is modular. The route aggregation module registers, among others, the subset dedicated to NutriBudget under the /nutribudget prefix as well as some technical support endpoints. The specific routes for NutriBudget API orchestration are implemented in a separate module, loaded under the mentioned prefix.

- "Health" and "ping" endpoints (/health, /ping) exist for observability and simple tests.
- A farm access endpoint (/farms) illustrates the use of a database client for simple reads and ensures rejection of any improper parameterized request, contributing to injection prevention.

The "nutribudget" routes rely on strict JSON validation schemas for their request bodies and orchestrate calls to the external NutriBudget API; they perform the necessary chaining (creation, update, launch/wait for results) and standardize responses on the NutriPlatform.

Centralized Error Management

The service defines a standard application error class, **AppError**, to produce structured responses and control the exposure of sensitive information depending on the environment. A **global error handler** in Fastify translates these exceptions into harmonized JSON responses, masks technical details in production, logs complete structured logs, and also handles schema validation errors.
This approach ensures **observability invariants**: all errors are logged with their context and a stable response format is returned to the client.

Environment Variables and Default Security

Operational configuration relies on **environment variables**, loaded at server startup:

- Server parameters: NUTRIPLATFORM_SERVICES_PORT, NUTRIPLATFORM_SERVICES_HOST, NUTRIPLATFORM_SERVICES_LOGGER, and optional metrics activation via NUTRIPLATFORM_METRICS_ENABLED.
- Keycloak authentication parameters: KEYCLOAK_URL, KEYCLOAK_REALM, KEYCLOAK_CLIENT_ID used by the JWT middleware.
- NutriBudget integration parameters: NUTRIBUDGET_API_URL, NUTRIBUDGET_USERNAME, NUTRIBUDGET_PASSWORD required by the NutriBudget orchestration routes

Default security is expressed by:

- Mandatory authentication on all routes (unless explicitly configured otherwise for test scenarios).
- Application of security headers and a CORS policy, to be restricted in production to authorized origins and methods.
- Global rate limiting and systematic payload validation.
- Standardized errors and structured logging.

5.2.4 Performance and Robustness

Fastify offers a **high-performance HTTP server** with minimal abstraction cost. The adopted configuration leverages its strengths:

- **Server-side schema validation** reduces execution errors and protects against malformed payloads, avoiding unnecessary processing.
- The selected **plugins** (CORS, Helmet, rate-limit, optional metrics) provide guarantees without significant overhead when configured sparingly.
- **Structured logging** and latency measurement hooks allow precise tracking of response times and guide optimizations if necessary.

This strategy on the Fastify service side is consistent with the choice to **avoid Redis on the Hasura side** in the current state of the platform and to focus on native and lean design optimizations. The nutriplatform-services service also does not introduce a Redis dependency and favors a simple and robust approach.

5.2.5 Security Invariant Tests and Validation

Two targeted security tests validate essential invariants:

- A **brute-force protection test** checks the activation and effectiveness of global rate limiting.
- An **injection resilience test** checks the service's response when unauthorized parameters or suspicious payloads are provided, and ensures that no sensitive information leaks appear in error messages.

The architectural intent is to guarantee **measurable and automated security invariants**: reasonable rate per IP, strict input validation, no propagation of technical information in production, and a stable error format for all consumers.

5.2.6 Summary

The **Fastify** architecture adopted in nutriplatform-services emphasizes **lean design**, **default security**, and **observability**. **Plugins** and **hooks** are used in a targeted way to provide CORS, security headers, rate limiting, latency measurement, and structured logging, while **validation schemas** ensure input robustness. The modular structuring of **domain-based routes**, **centralized error management** via **AppError**, and **smooth integration** with Hasura Actions enable orchestration of NutriBudget scientific API calls while keeping the backend lightweight, performant, and maintainable.

5.3 Hasura Client

The **Hasura client** occupies a central place in the technical architecture of NutriPlatform. It is a software component whose exclusive mission is to provide reliable, typed, and secure access to the project's central database, based on PostgreSQL/PostGIS and orchestrated by Hasura.

5.3.1 Role and Design of the Client

The Hasura client, implemented in the shared module libs/hasura-client, is a **minimalist wrapper around Prisma**. It exposes a single function: create(databaseUrl), which returns a PrismaClient instance connected to the database URL. This architectural choice guarantees:

- **Automatic adaptation to the schema**: Prisma dynamically generates TypeScript types from the database schema, ensuring that any change in the data model is immediately reflected in the code.
- **Increased security and robustness**: The combined use of TypeScript and Prisma allows detection at compile time of any inconsistency between business code and database structure, drastically reducing the risk of runtime errors.

It is essential to emphasize that this client implements **no business logic**: it serves exclusively as a **technical gateway** between backend services and the database. All CRUD operations and complex queries are performed via this typed interface, which promotes backend maintainability and consistency.

5.3.2 Integration into Backend Services

In NutriPlatform, the Hasura client is used by complementary backend services (Fastify/TypeScript) to:

- Execute read, write, and update operations on business entities (e.g., fields, farms, NutriBudget runs, etc.)
- Ensure that each database access respects the schema and constraints defined in PostgreSQL/PostGIS
- Benefit from strict typing and automatic documentation of data models

The absence of business logic in this client allows centralizing database access management while leaving services responsible for orchestrating application processing.

5.3.3 Value of TypeScript/Prisma Typing

The use of Prisma and TypeScript for the Hasura client offers several decisive advantages:

- **Strict consistency**: Any change in the database schema is immediately taken into account in the generated types, avoiding divergences between data structure and code.
- **Security**: Type or access errors are detected at compile time, before code execution.
- **Scalability**: The client's clear and typed interface makes it easy to add new models or operations, without risk of regression.

5.3.4 Conclusion

The Hasura client of NutriPlatform embodies a pragmatic architectural choice: it offers a data access interface that is simple, robust, and perfectly aligned with the security, maintainability, and scalability requirements of the NutriBudget project. Its integration into backend services guarantees the quality and sustainability of the entire technical platform.

5.4 Business Logic

5.4.1 Role of "Business Logic" in nutriplatform-services

In the current state of the code, the **business logic** layer of the NutriPlatform backend service primarily acts as an **orchestration layer** between the frontend and the project's external scientific API, **NutriBudget**. It performs **strict input validation**, **normalization of certain parameters** via default values, **centralized error management** using **AppError** objects, and **traceability** through **structured logs**. It does not contain an agronomic rule engine or recommendation algorithms specific to the service.

Additionally, the service exposes a minimal data access point, which queries the database via **hasura-client/Prisma**. This direct read is deliberately limited and carries no specific business rules beyond input checks.

5.4.2 Security and Validation

Application security relies on several invariants. **Authentication** is provided by **Keycloak** and validated by **JWKS** (RS256 signature, issuer and audience verified). Useful **claims** (roles, scopes) are extracted and available for processing, allowing for additional **authorization** checks where needed. **Input validation** is performed by strict schemas attached to Fastify routes, forbidding additional properties and guaranteeing expected types.

Secrets for accessing the NutriBudget API (URL, username, password) are provided by the environment and never transit to the client side. Errors are encapsulated in **AppError** to produce uniform responses and avoid leaking sensitive information in production.

5.4.3 Observability, Errors, Robustness

The service implements **structured logs** for each request (method, URL, status, response time, user if applicable) and logs errors in detail, always via the centralized handler. **Global rate limiting** is enabled to limit abuse and reduce exposure to brute-force attacks. **Security headers** are applied using Helmet, and **CORS** is explicitly configured.

A **Prometheus** endpoint can be enabled via an environment variable to provide standard traffic metrics. On the functional robustness side, the "/compute" orchestration uses **exponential backoff retry** to wait for the availability of results from the external API without blocking indefinitely. At this stage, no **custom HTTP timeout** is defined for outgoing calls; this is a known limitation that does not prevent the observed stability in the current scenario.

Automated security tests serve as architectural safeguards: a **brute-force protection test** checks for HTTP 429 triggering when the quota is exceeded, and an **injection test** checks the expected behavior of the "/farms" route.

5.4.4 Current State and Limitations

The "Business Logic" layer is **deliberately minimal** and focused on **orchestration**. **Agronomic calculations** and **interpretation rules** are externalized in the project's scientific API, **NutriBudget**.

This approach reduces service complexity, facilitates maintenance, and allows iteration on the model side without redeploying the backend.

Known limitations concern the absence of **dedicated HTTP timeouts** on the external client side and the restricted scope of internal data access, deliberately limited to simple reads via **Prisma**. No cross-cutting business logic (e.g., rule engine) is implemented at this stage.

5.4.5 Planned Evolutions

If business logic needs arise (e.g., more advanced I/O normalization, mappings to internal structures, pre/post-processing of results), they will be integrated as **dedicated modules** within this service, respecting existing **security invariants** (JWT, scopes, strict validation) and maintaining **observability** and **centralized error management**. Interactions with **Hasura Actions** may be considered to expose these processes within the GraphQL framework, without duplicating controls already present on the Fastify side.

6 Data

6.1 Data Model

6.1.1 Vision and Design Principles

The data model of the NutriPlatform forms the foundation for the project's analytical and simulation capabilities. It has been designed not as a simple storage schema, but as a structured representation of agricultural reality, tailored to meet the needs of farmers, advisors, and policy makers. Its design is guided by four fundamental principles:

- **Representation of the Farm:** The model aims to capture the key entities and processes of a farm (crops, livestock, inputs) with sufficient granularity to enable precise simulations and site-specific recommendations, in line with the objectives of NutriBudget.
- **Modularity and Flexibility:** Aware that agricultural practices and simulation models evolve, the schema is designed to be modular. The separation of concepts (e.g., crop_area and its inputs fertilizer/manure) and the use of flexible data types (JSON, arrays) allow new parameters or models to be integrated without major changes to the database.
- **Simulation-centric Architecture:** The data architecture is organized around the concept of **nutriplan**, which represents a complete nutrient management scenario. Each simulation (nutribudget_run) is a "snapshot" of the state of a nutriplan at a given time, enabling clear traceability and comparison of results.
- **Interoperability and Integration:** The model natively integrates identifiers (user_id) and structures (location) that facilitate integration with third-party systems such as Keycloak for authentication and GIS tools via PostGIS for spatial analysis.

6.1.2 Conceptual Structure and Key Entities

The model is structured around the farm (farm) and its plans (nutriplan). The latter aggregates all the data required for NutriModels simulations, as illustrated in Figure 5.

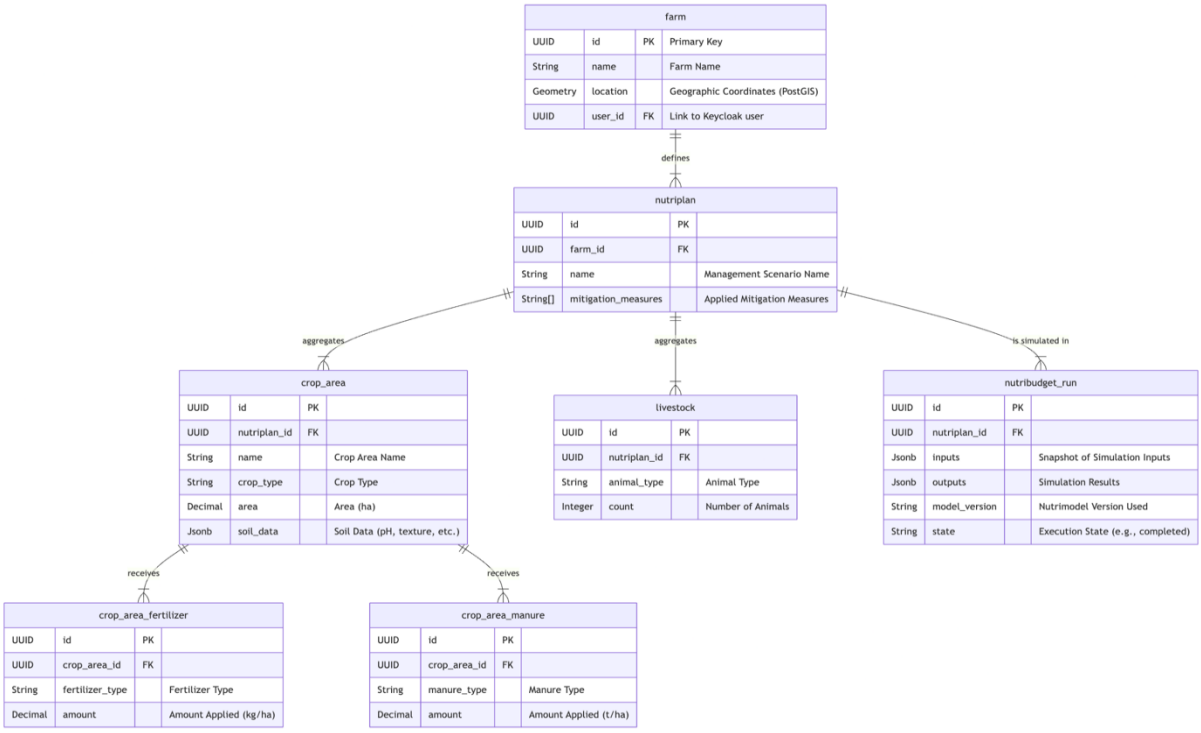

Figure 5. Conceptual data model diagram of the NutriPlatform.

- **farm**: Represents the farm, the root entity. It is linked to a user (user_id) and has a geographic location (location), which is essential for spatial analyses and contextual recommendations.
- **nutriplan**: This is the core of the model. It represents a **scenario** for a farm. It aggregates information on crop areas (crop_area), livestock (livestock), and the mitigation measures (mitigation_measures) considered.
- **crop_area**: Describes a homogeneous crop area. The richness of its attributes, such as the soil properties. It provides the detailed input data required by the Nutrimodels for precise simulations.
- **livestock**: Models information related to livestock, another major contributor to the nutrient balance.
- **crop_area_fertilizer & crop_area_manure**: These tables detail the inputs applied to each crop area. Their separation from crop_area allows for flexible modeling where a single area can receive several types of fertilizers or manures.
- **nutribudget_run**: This table is the simulation log. It records each execution of a Nutrimodel for a given nutriplan. It captures the simulation inputs (inputs) and outputs (outputs), its state (state), and metadata about the model version used.

6.1.3 Justification of Modelling Choices

Some design choices deserve special attention as they reflect a balance between flexibility, performance, and simplicity.

- **Storage of simulation inputs/outputs in JSON (nutribudget_run)**:
  - **Justification**: Nutrimodels are complex and their input/output parameters are likely to evolve. Using **JSON** fields offers **maximum flexibility**. It is possible to add, modify, or remove parameters in the models without requiring a database schema migration, which is a considerable advantage during research and development phases.
  - **Trade-off**: This choice makes complex analytical queries on simulation parameters (e.g., "find all simulations where parameter X was greater than Y") difficult or even impossible in pure SQL. These analyses will have to be performed at the application level, after extracting the JSON data.
- **Definitions of farm crop and livestock in nutriplan**:
  - **Justification**: The farm data such as fields and livestock lives in a **nutriplan** table: it enables to run simulations where all fields data are parametrizable by the user (such as fertilization, soil data...).
  - **Implication**: For the user convenience, it is necessary to provide a feature to create a Nutriplan from an existing one to avoid providing again all the necessary data.
- **Granularity of crop_area**:
  - **Justification**: Grouping fields in crop areas allow the user to quickly declare its data at crop-level instead of field-level , but it also allows for field-level precision if desired by the user as there can be multiple crop areas with the same crop.

6.2 Database Architecture

6.2.1 Introduction

The **PostgreSQL** database is the backbone of the **NutriPlatform** data layer. It serves as the single source of truth for all persistent data, designed to meet the requirements of the NutriBudget project. Its role is to store, structure, and provide access the NutriPlatform data.

6.2.2 Architectural Philosophy

The database design is guided by principles to ensure robustness, scalability, and maintainability.

- **API-Oriented Design**: The schema is built with **Hasura** as the main consumer. Naming conventions, relationship definitions (via foreign keys), and data types are chosen to maximize

Hasura's ability to generate a rich and intuitive GraphQL API. This approach accelerates backend development and ensures consistency between the data and API layers.

- **Single Source of Truth**: All core business data resides in the PostgreSQL database. This centralization avoids data fragmentation and ensures that all components, whether backend services or the frontend application, operate on consistent and up-to-date information.
- **Modularity and Extensibility**: The schema is organized into loosely coupled logical domains (e.g., farm structure, simulation runs). This modularity is essential for the long-term evolution of the platform, allowing new features, data sources, or simulation models to be integrated with minimal impact on existing structures.

6.2.3 Key Technological Choices

The selection of technologies and patterns is crucial to meet the platform's functional and non-functional requirements.

- **PostgreSQL**: Chosen for its proven reliability, performance, and rich feature set. Its open-source nature and strong community support make it a sustainable choice for a long-term research and innovation project.
- **PostGIS Extension**: **PostGIS** Since NutriBudget deals with farms and fields, the ability to store, query, and manipulate geospatial data is a requirement. It is used to store farm location (geometry) and enable spatial queries.

6.2.4 Relational Structure and Logical Organization

The database schema is organized into two main logical domains, reflecting the core concerns of the NutriPlatform.

- **Central Agricultural Domain**: This domain models the physical and logical structure of a farm and its nutriplans. It includes tables such as farm, nutriplan, crop_area, and livestock. These entities are tightly interconnected to represent a view of the farm, forming the basis on which simulations are performed.
- **Simulation Domain**: Centered around the nutribudget_run table, this domain captures the execution of nutrient budget models. Each run is linked to a specific farm nutriplan and stores its input parameters and resulting KPIs in JSONB columns. This design effectively decouples stable, slowly changing agricultural data from dynamic, fast-growing simulation data.
- **Reference Data Management**: In line with the decoupling strategy, the database contains **no reference tables** (e.g., crop types, units, etc.). These data are managed externally and consumed by the client. This approach ensures that the database only stores operational and transactional data, maximizing flexibility.

6.2.5 Scalability and Maintainability

The architecture is designed not only for the present but also for the future evolution of the NutriBudget project.

- **Schema Migrations Management**: Schema changes are managed via the **Hasura migration system**. All modifications are captured as versioned SQL and metadata files (apps/backend/hasura/migrations). This ensures that schema deployments are automated, reproducible, and auditable, which is essential for maintaining consistency across different environments (development, staging, production).
- **Schema Extensibility**: The modular design allows for frictionless extension. For example, integrating a new data source, such as on-farm sensors, would likely involve adding new tables linked to farm or crop_area entities, without disrupting the existing core schema. Similarly, a new type of analytical model can be supported by adding new logic that creates nutribudget_run entries with a different set of parameters.

6.3 Spatial Data Management

Spatial data management provides the technical foundation for geographically contextualizing agricultural information, enabling relevant analyses, and delivering intuitive map-based visualizations to various users, including farmers, advisors, and regional authorities.

6.3.1 Role and Rationale for PostGIS

The choice of **PostgreSQL** as the database management system was deliberately coupled with the activation of its **PostGIS** extension.
This strategic decision equips the NutriPlatform with a robust object-relational database, capable of natively and efficiently storing, querying, and manipulating geographic data.

6.3.2 Integration into the Data Model

The NutriPlatform data model integrates spatial data at strategic levels to meet business needs.

The farm table contains a location field of type **Point**. This field is essential for:

- **Geographically locating each farm** on a map.
- **Contextualizing the farm** with respect to external environmental data (climate data, regional soil types, etc.).
- **Aggregating data** at a regional scale or for the project's pilot zones.

6.4 Reference Data Management

6.4.1 Architectural Principle

Reference data management within the NutriPlatform is based on a fundamental principle of **complete decoupling** between reference data and the operational database. Unlike a traditional approach where reference data (such as crop types, units, model parameters, etc.) are stored in dedicated database tables, the NutriPlatform adopts a **client-driven (frontend) strategy**.
This approach aims to maximize flexibility, simplify reference data maintenance, and allow their evolution independently of the application deployment lifecycle.

6.4.2 Source of Truth: Dedicated Git Repository

The single source of truth for all NutriPlatform reference data is an external, dedicated Git repository: https://gitlab.com/nutribudget/nutriplatform-definitions.

This repository contains a set of structured files (mainly in **JSON** format) that define the lists of values, parameters, and configurations required for the proper functioning of the application. Managing this data via Git offers significant advantages:

- **Native versioning:** Every change to the reference data is tracked, versioned, and can be audited.
- **Collaboration:** Non-developer domain experts can be involved in data maintenance through review processes (Merge Requests).
- **Asynchronous deployment:** Updates to reference data do not require redeployment of the backend application or database migration.

6.4.3 Reference Data Consumption Architecture

The reference data flow is managed exclusively by the frontend application.

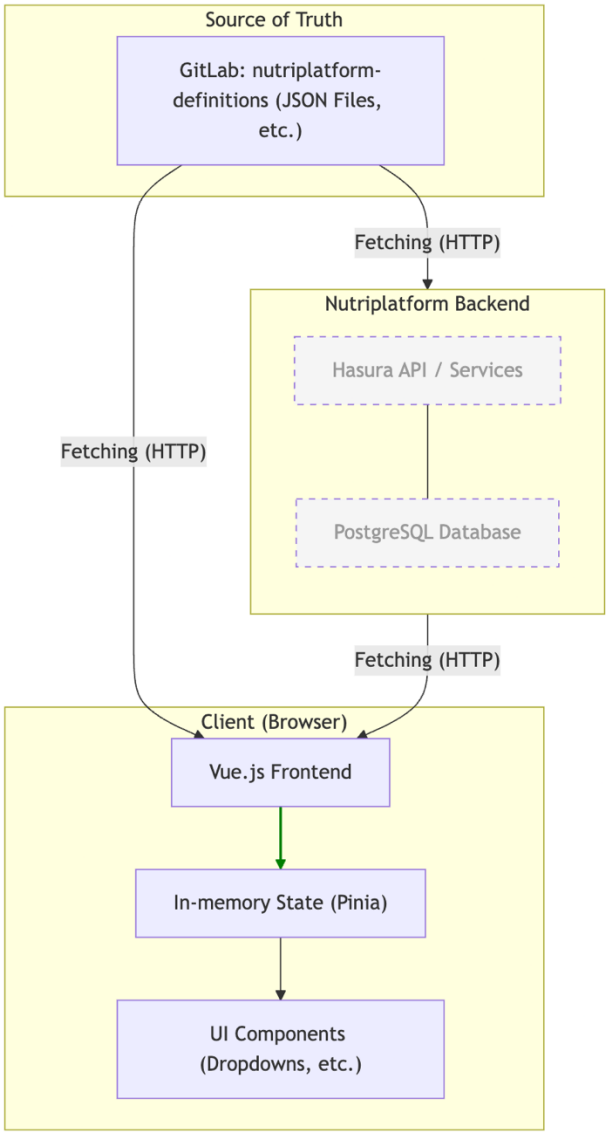

Figure 6. Reference data management architecture diagram.

Flow Description:

1. **Retrieval**: Reference data from the Git repository are synchronized and copied into the frontend project by a dedicated script. This script ensures that the latest version of all required reference files (such as crop-types.json) is available within the application. On startup or when necessary, the Vue.js frontend loads these files locally, keeping the reference data up to date and ready for immediate use.
2. **In-memory Management:** The retrieved data is parsed and stored in the application's global state (managed by **Pinia**). It is never persisted in the database.
3. **Usage:** User interface components (dropdown lists, validation logic, etc.) consume this data directly from the in-memory state.

7 Security

7.1 Overview

The adopted approach is **defense in depth**, where multiple layers of security are implemented to protect both data and services.

7.1.1 Security Principles

- **Least privilege**: Users and services have access only to the resources strictly necessary for their functions.
- **Defense in depth**: Security controls are applied at every layer of the architecture (infrastructure, network, application, data).
- **Security by design**: Security is integrated from the design phase and throughout the entire development lifecycle.
- **Centralized identity management**: Identity and access management is centralized through a single identity provider (Keycloak).

7.1.2 Threat Model (Simplified)

Table 4. Summary of security risks and their mitigations in the NutriPlatform.

Threat	Attack vector	Mitigation measure
Unauthorized data access	Credential theft, privilege escalation	Strong authentication (MFA under consideration), strict RBAC, data isolation.
Malicious data injection	Public APIs, forms	Systematic input validation (client and server side), parameterized queries.
Data interception	Man-in-the-middle attacks	Encryption in transit (TLS) for all communications.
Denial of service (DDoS) attack	Request flooding	Ingress-level protection (Traefik), rate limiting.
Exposure of sensitive data	Data leaks, misconfiguration	Encryption at rest, secure secret management, regular audits.

7.2 Authentication

Authentication within the NutriPlatform is fully delegated to **Keycloak**, which acts as a centralized Identity Provider (IdP). This approach decouples user management from application logic.

7.2.1 Keycloak Architecture

- **Realm**: A dedicated realm named NutriPlatform is configured to completely isolate users, roles, and clients for the project.
- **OpenID Connect Client**: A public client, nutriplatform-app, is defined for the frontend application (SPA). It is configured to use the standard Authorization Code Flow.
- **Custom Theme**: The NutriPlatform theme is applied to provide a login and registration experience consistent with the project's visual identity.
- **Internationalization**: The realm supports multiple languages (en, fr, es, it, de, fi, nl) to accommodate the various pilot regions.

7.2.2 Authentication Flow (OIDC)

The authentication flow follows the OpenID Connect standard, ensuring secure and standardized integration.

The process begins when an **unauthenticated user** (U) attempts to access a protected resource on the **Frontend** (F). The application detects the absence of a valid session and initiates the OIDC flow by **redirecting** the user's browser to the login page hosted by **Keycloak** (K).

The user enters their **credentials** directly on Keycloak's secure interface. After successful authentication, Keycloak redirects the user back to the Frontend with a single-use **Authorization Code** in the URL.

The Frontend receives this code and immediately uses it to make a direct (back-channel) request to Keycloak's token endpoint. The purpose of this call is to **exchange the authorization code** for a set of tokens: an **Access Token**, an **ID Token**, and a **Refresh Token**.

Once the **JWT tokens** are received, the Frontend stores them securely. For each subsequent API call to the **Backend** (B), the Frontend attaches the **Access Token** in the Authorization header of the GraphQL request.

Upon receipt, the Backend performs a critical validation step: it **validates the JWT signature** to ensure its authenticity and that it has not been tampered with. It then **decodes the claims** (token content) to identify the user and their permissions. If the token is valid and the user is authorized to access the resource, the Backend processes the request and returns the **authorized data** to the Frontend.

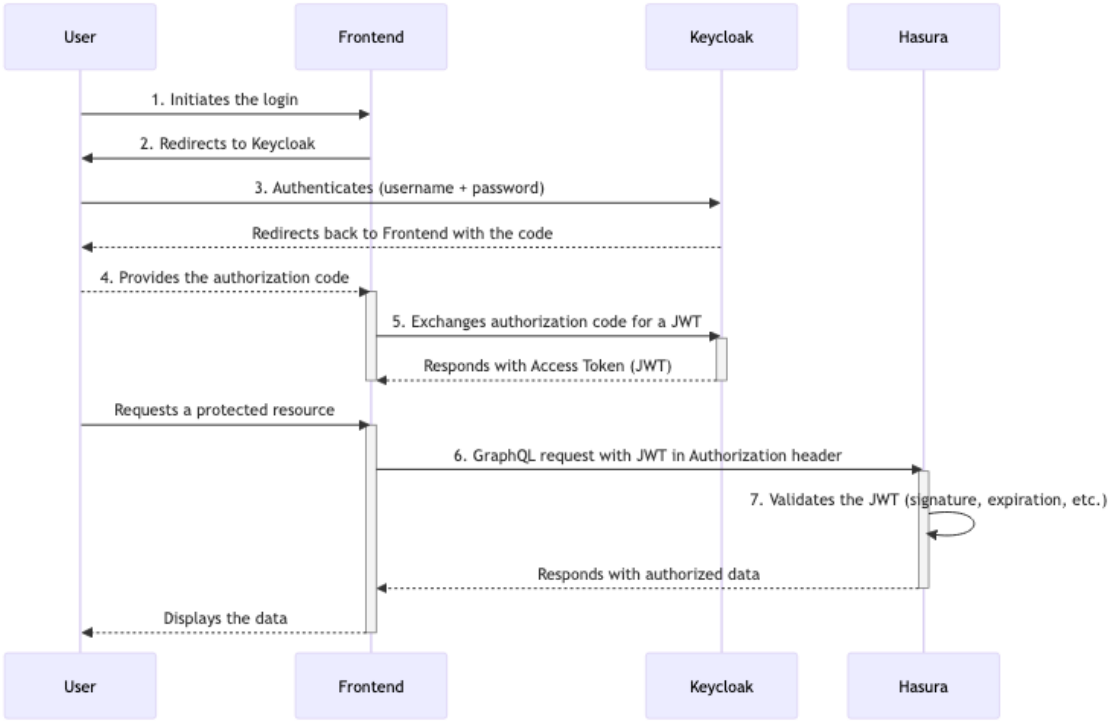

Figure 7. Sequence diagram of the authentication and authorization flow (OIDC/Keycloak/Hasura).

7.2.3 User Management

The realm configuration enables key features for user lifecycle management:

- **Registration allowed** (registrationAllowed: true): New users can create an account.
- **Email as username** (registrationEmailAsUsername: true): Simplifies identification.

- **Email verification** (verifyEmail: true): Ensures that email addresses are valid.
- **Password reset allowed** (resetPasswordAllowed: true): Allows users to recover their account.

7.3 Authorization

Authorization is managed through a **Role-Based Access Control (RBAC)** model. Keycloak is responsible for role assignment, while Hasura enforces permissions based on these roles.

7.3.1 Role Model

Two main business roles are defined in the Keycloak realm NutriPlatform:

- **farmer**: Represents a farmer or an agricultural advisor. This role grants access to farm-level management features.
- **policy_maker**: Represents a policy maker. This role grants access to dashboards and aggregated data visualizations at the regional or European level.

A user can be assigned multiple roles.

7.3.2 Authorization Integration with Hasura

The transfer of authorization information from Keycloak to Hasura is performed transparently and securely via **custom claims** injected into the JSON Web Token (JWT).

Protocol Mappers

The Keycloak client nutriplatform-app is configured with specific protocol mappers to create a JWT that Hasura can interpret:

1. **x-hasura-user-id**: This claim contains the user's unique identifier (id) from Keycloak. Hasura uses this value to filter data and ensure that a user can only access their own information.
2. **x-hasura-allowed-roles**: This claim contains an array of the user's roles (e.g., ["farmer", "policy_maker"]). Hasura uses this list to determine which set of permissions to apply.
3. **x-hasura-default-role**: This claim defines the default role to use for the request. In the current configuration, it is set to farmer.

Authorization Flow

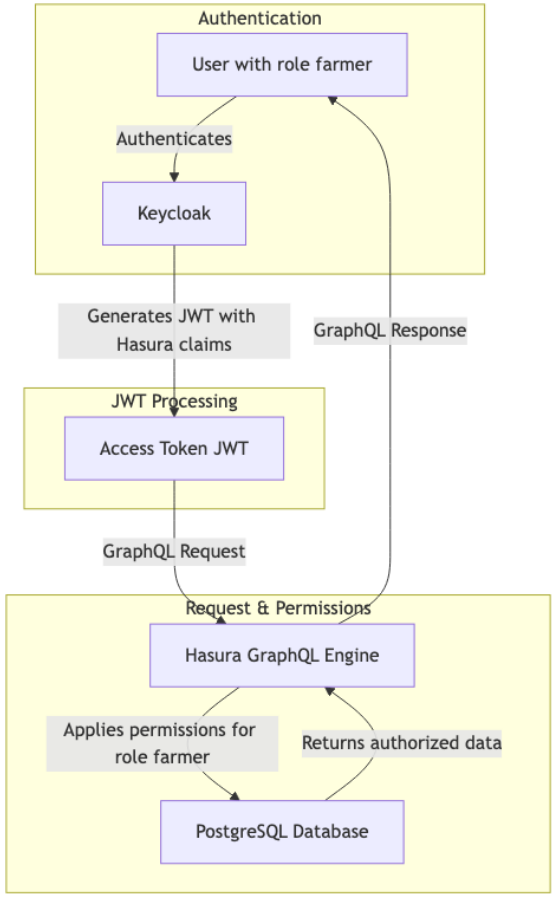

Figure 8. This mechanism Hasura to apply row - and column - level security rules declaratively, without requiring custom authorization code in backend services.

This diagram presents the complete authorization flow that establishes secure communication between the authentication infrastructure and the GraphQL engine. The architecture reveals six primary entities: the user bearing the **farmer** role, the **Keycloak** server responsible for authentication, the **JWT access token** containing custom claims, the **Hasura GraphQL Engine** that enforces permissions, and the **PostgreSQL database** that stores business data.

The process begins when the user authenticates with Keycloak, which generates a JWT enriched with Hasura-specific claims (x-hasura-user-id, x-hasura-allowed-roles, x-hasura-default-role). This token then accompanies every GraphQL request sent to Hasura. The GraphQL engine extracts and validates the authorization information from the JWT, then applies permission rules corresponding to the **farmer** role before executing the query on PostgreSQL. The authorized data flows back through the same path to the user.

The diagram structures this logic into three conceptual subgraphs: the **Authentication** zone that encapsulates the user and Keycloak, the **JWT Processing** zone that isolates the access token, and the **Request & Permissions** zone that groups Hasura and PostgreSQL. This segmentation illustrates the separation of responsibilities between identity authentication, secure transport of authorizations, and effective application of permissions at the data level.

This mechanism allows **Hasura** to apply row- and column-level security rules declaratively, without requiring custom authorization code in backend services.

7.3.3 Authorization Model Diagram

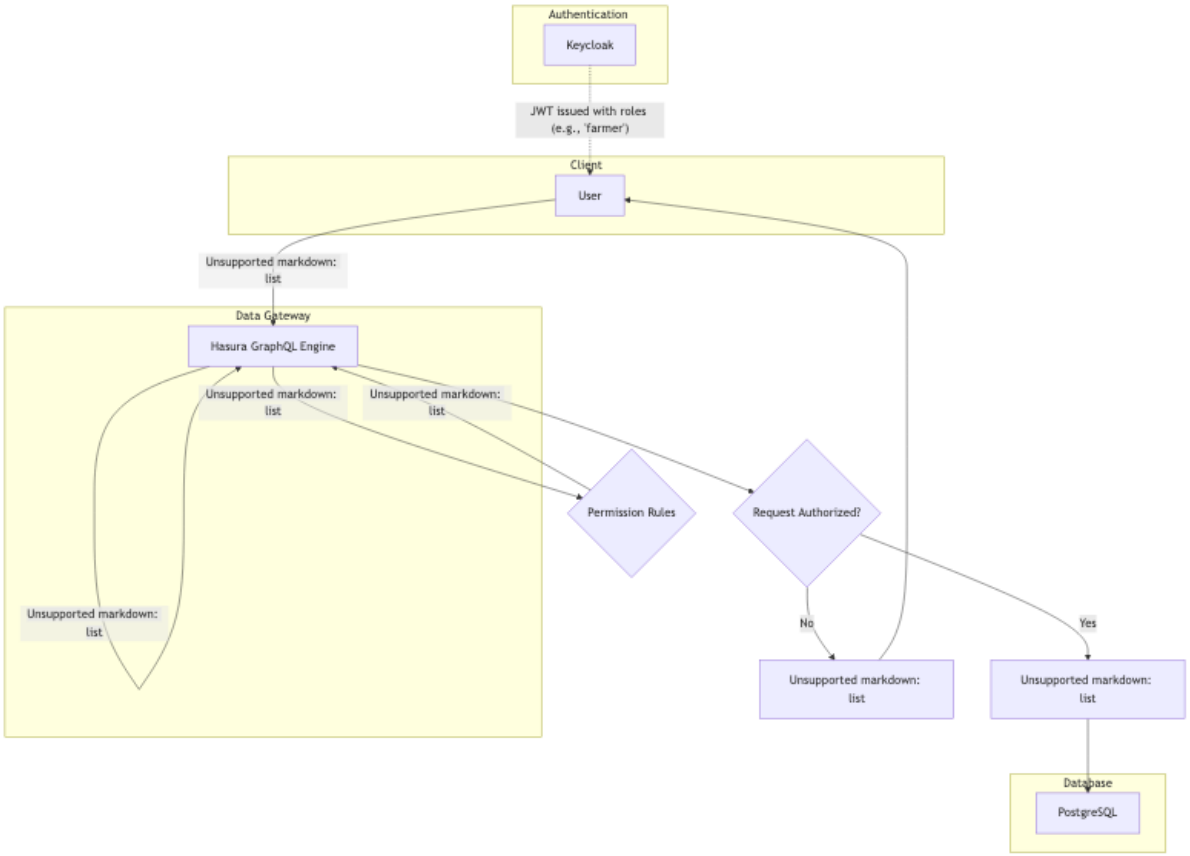

Figure 9. Authorization model diagram detailing the evaluation of permission rules and query execution in Hasura.

This flowchart diagram illustrates the complete authorization model architecture by revealing the step-by-step information flow from the user to the database or to an error response. The architecture is structured around four distinct functional subgraphs: the **Client** representing the end user, the **Authentication Infrastructure** encapsulating Keycloak, the **Data Gateway** containing the Hasura GraphQL Engine, and the **Database** hosting PostgreSQL.

The authorization process is articulated through six numbered steps and one critical decision point. The user initiates the flow by sending a GraphQL request accompanied by the JWT to Hasura (step 1). In parallel, Keycloak issues the JWT enriched with user roles (e.g., 'farmer') to the user, materialized by a dotted arrow that symbolizes the prior token issuance. Hasura then validates the JWT and extracts the role via the X-Hasura-Role header (step 2), then consults its internal permission rules (step 3). These rules define the authorized access in terms of operations and columns for the concerned role (step 4).

The system's core relies on the **Request Authorized?** decision that determines the request authorization. In case of positive authorization, Hasura executes the query on the database (step 5) and PostgreSQL processes the operation. In case of refusal, the system immediately returns an authorization error to the user (step 6), short-circuiting any data access. This architecture ensures that every request is systematically evaluated according to the defined permission rules, thus guaranteeing data security at the granular level of rows and columns.

7.4 Data Protection

Data protection is a central concern for the security of the NutriPlatform. This section details the current practices and recommendations adapted for a small team.

7.4.1 Data Encryption at Rest

As of now, **encryption at rest** is not enabled on the PostgreSQL instance used by the NutriPlatform. No native encryption parameters (TDE, disk encryption) are configured in the deployment or configuration files. This situation is common in Docker environments, where encryption is often managed at the underlying infrastructure layer (disk, volume, cloud provider).

Recommendation: To strengthen security, it is advised to consider encrypting the Docker volumes hosting PostgreSQL data, either via disk encryption solutions (LUKS, VeraCrypt) or via cloud encryption options if the infrastructure allows. This measure is particularly relevant if sensitive or personal data is stored.

7.4.2 Secret Management

Secrets (passwords, API keys, tokens) are stored in a **private repository** and encrypted using **git-crypt** technology. This approach ensures that only authorized team members can access sensitive information, while facilitating version control and secret rotation.

Advantages:
- Simple to implement
- Fine-grained access control via the private repo
- Transparent encryption compatible with Git

Limitations:
- Depends on the security of the repository and user access management
- Manual secret rotation (should be scheduled regularly)

7.4.3 Access and Operation Logging

Logging of access and operations on the PostgreSQL database is enabled by default via the DBMS's native parameters: logging_collector, log_statement, log_connections, etc.
- PostgreSQL logs are enabled in the configuration file or via Docker variables.
- Logs are stored in a persistent Docker volume, allowing for consultation and archiving.
- Log rotation and archiving are automated via a simple tool (Logrotate or equivalent).

Advantages:
- Continuous traceability of access and operations
- Automated archiving, reducing the risk of oversight
- Easy auditing in case of incident

Points of attention:
- Regularly monitor the disk space allocated to logs
- Document the procedure for log consultation and archiving

7.4.4 Data Backup and Restoration

The **backup and restoration** strategy has been formalized to ensure data resilience in on-premise environments (Docker Swarm). The adopted solution relies on the combined use of **NFS (Network File System)** for storage and **GPG (GNU Privacy Guard)** for encryption.

The backup policy is defined by strict objectives:
- **RPO (Recovery Point Objective):** 6 hours, meaning the maximum acceptable data loss is six hours of work.
- **RTO (Recovery Time Objective):** 4 hours, defining the maximum time to restore service after an incident.
- **Retention policy:** 14 daily backups and 8 weekly backups are kept.

The process is fully automated via the script “backup-postgres.sh”, which uses pg_dump to export the data. Backups are then encrypted using a GPG public key before being stored on a dedicated NFS share, mounted on a backup node. No unencrypted artifacts are kept. This centralized and secure approach replaces previous methods that relied on local volumes or less formalized shares.

7.4.5 Validated Protection Strategy

The NutriPlatform's data protection strategy is now formalized and is based on the following decisions:

- **Access logging:** Native PostgreSQL logging is enabled and logs are stored on persistent volumes to ensure traceability.
- **Secret management:** Development and configuration secrets are managed via git-crypt in a private repository.
- **Secure backups:** The backup policy is rigorously defined with **RPO/RTO** objectives, clear retention, and a technical implementation based on **NFS + GPG** for storage and encryption.

7.5 Infrastructure Security

SSL/TLS certificates are managed strictly by the hosting infrastructure (load balancer, external proxy, or security appliance). All network traffic reaching NutriPlatform VMs is plain HTTP; there is no management of certificates, HTTPS ports, or TLS logic at the application or cluster level.

The security of the **NutriPlatform** infrastructure is designed to be robust, auditable, and strictly aligned with project governance and DevSecOps best practices. **All encryption and certificate management (TLS/SSL)—including provisioning, storage, renewal, and termination—are handled exclusively at the infrastructure edge, via Traefik (or equivalent) load balancer/reverse proxy**. There is **no handling of TLS/SSL certificates, private keys, or secrets at the cluster, application stack, or container level**: all application traffic delivered to NutriPlatform VMs is plain HTTP, and authentication and validation occurs exclusively at the edge. This strategy of **total externalization** guarantees operational simplicity, architectural clarity, and uniform security, as validated by the NutriBudget project doctrine.

7.5.1 Orchestration and Isolation

The infrastructure is orchestrated using **Docker Swarm**, which provides a native clustering environment and simplifies the management of distributed services. This approach replaces the use of docker-compose for production and pre-production environments, reserving the latter for local development.

Network segmentation is a cornerstone of our security strategy. Several **overlay networks** are defined to isolate services according to their role:

- traefik-public: An external network connecting the **Traefik** reverse proxy to publicly exposed services.
- app-net: An internal network for communication between application services (frontend, backend).
- db-net: A highly restricted network for communication between services and their respective databases.

This isolation ensures that services can only communicate with each other if they share a common network, thus limiting the attack surface in case a container is compromised.

7.5.2 Node and Access Security

Access to Swarm cluster nodes is strictly controlled. Administration is performed exclusively via **SSH**, with authentication based on **individual public keys**. No password-based SSH access is allowed. Access management (creation, revocation) is centralized and documented.

Service placement on nodes is managed by **Docker Swarm placement constraints**. Nodes are labeled according to their role (e.g., tier=manager, tier=worker, tier=database), ensuring that sensitive containers, such as databases, run only on specifically secured and isolated nodes.

7.5.3 TLS Edge-Only Doctrine and Complete Externalization of Certificate Management

In strict accordance with project documentation and operational policy, **TLS/SSL certificate management is fully externalized**: the hosting infrastructure (edge/load balancer via Traefik or equivalent) is solely responsible for secure TLS provisioning, renewal, storage, and termination of all HTTPS connections.

There is no processing, storing, or handling of TLS/SSL keys, certificates, or secrets within the Swarm cluster, application stack, or containers. This means:

- No TLS certificates are imported, stored, or referenced in Docker Swarm/NutriPlatform stacks.
- No “certificate” secret files are created or transmitted to any application component.
- All TLS/OIDC compliance validation takes place **immediately at the edge reverse proxy**. The cluster only receives traffic considered trusted and already validated.

This “edge-only” doctrine is non-negotiable. Any evolution that would (re)introduce certificate handling at the cluster or application level must be explicitly reviewed, documented, and traceable.

For any security-related information regarding TLS/SSL in the platform, users, auditors, and developers must **refer to the “Infrastructure”, “Networking”, and blueprint-progress documentation for full audit trail and decisions** (see links below).

Important: ***SSL/TLS certificates are managed strictly by the hosting infrastructure (load balancer, external proxy, or security appliance). All network traffic reaching NutriPlatform VMs is plain HTTP; there is no management of certificates, HTTPS ports, or TLS logic at the application or cluster level.*** *No documentary or technical exceptions are permitted.*

7.5.4 Secret Management

Secret management in production is handled by **Docker Swarm secrets**. This feature allows sensitive information (passwords, API keys, TLS certificates) to be stored and transmitted securely to containers that require them.

The advantages of this approach are numerous:

- Secrets are **encrypted at rest** and transmitted to containers via a secure channel.
- They are only mounted in the container's memory (/run/secrets/), never written to the container's disk.
- Access to a secret is limited to only those services explicitly authorized in the stack definition.

This method is significantly more secure than using plain environment variables in .env files, which is still used for non-sensitive configuration or in development environments.

7.5.5 Container Security

Container security is reinforced through several practices:

- **Minimal images**: We use minimalist base images (such as Alpine) whenever possible to reduce the attack surface.
- **Non-root users**: Services in containers are run as non-privileged users whenever the application allows.
- **Read-only file system**: The root file system of containers is configured as read-only (read_only: true) whenever possible, with specific volumes mounted for paths requiring write access.

- **No excessive privileges**: No container runs in --privileged mode.

7.5.6 Monitoring and Observability

Infrastructure security monitoring is being structured. Currently, it relies on:

- **Centralized logging** from Docker Swarm, which collects logs from all services.
- The **Traefik dashboard**, protected by authentication (htpasswd), providing visibility into incoming traffic and service status.
- **Standardized healthchecks** for each service, allowing Swarm to monitor their health and automatically restart failing containers.
-

The integration of a more comprehensive observability stack (based on Prometheus, Grafana, and Loki/Promtail) is planned to provide deeper monitoring and proactive alerts.

7.6 API Security

The security of the **NutriPlatform** APIs is a critical component of the overall architecture. It is primarily ensured at the API gateway level, a role fulfilled by the **Hasura GraphQL Engine**. This approach centralizes security controls and guarantees consistent protection for all exposed data and services.

7.6.1 Single Entry Point: Hasura API Gateway

All client requests (mainly from the frontend application) are directed to a single entry point: the **Hasura** GraphQL API. Backend services, such as nutriplatform-services, are not directly exposed to the Internet. They are only accessible by Hasura through secure internal mechanisms (Actions, Remote Schemas).

This **API Gateway** topology is fundamental for security, as it allows protection efforts to be focused on a single component specifically designed for this task.

7.6.2 Authentication and Authorization (RBAC)

API access security relies on a robust authentication and authorization model based on roles (RBAC).

1. **Authentication via JWT**: Authentication is delegated to **Keycloak**. When a user logs in, Keycloak issues a **JSON Web Token (JWT)** containing the user's identity and, crucially, their roles (e.g., user, admin, regional_authority). This token is then sent in the Authorization header of each GraphQL request to Hasura.
2. **JWT Validation**: Hasura is configured to validate the signature of each incoming JWT using Keycloak's public key. If the token is invalid or expired, the request is immediately rejected with a 401 Unauthorized error.
3. **Role-Based Permissions (RBAC)**: This is the core of API security. For each table, view, and operation (select, insert, update, delete) in the database, **permission rules** are defined directly in Hasura's metadata. These rules are associated with a role. When a request is received, Hasura extracts the role from the validated JWT (e.g., X-Hasura-Role: user) and dynamically applies the corresponding permissions. This ensures that a user can only view or modify the data their role entitles them to. For example, a user with the user role can only see data from their own farm.

7.6.3 Protection Against Abusive Requests

Hasura integrates several mechanisms to protect the API against GraphQL requests that could overload the server:

- **Limited query depth**: It is possible to set a maximum depth for queries to prevent infinitely nested requests, whether malicious or accidental.

- **Node count limitation**: Hasura can limit the total number of nodes (records) returned in a single request.
- **Rate limiting**: Although not native to Hasura, rate limiting is applied at the ingress controller level (**Traefik**), which can limit the number of requests per IP over a given period.
- Allow Lists

For maximum security in production, the **Allow List** feature is enabled. When this feature is active, Hasura will only process GraphQL requests that exactly match a query previously registered and approved in a collection.

Any other request, even if syntactically correct and the user has the necessary permissions, will be rejected. This measure is extremely effective in preventing introspection attacks and ensuring that only the intended attack surface is exposed.

7.6.4 Internal Communication Security

Communication between Hasura and internal services (such as nutriplatform-services for Actions) is also secured:

- **Header secrets**: When calling a webhook for an Action, Hasura includes a shared secret in an HTTP header (e.g., X-Hasura-Action-Secret). The nutriplatform-services service checks for the presence and validity of this secret before processing the request. This ensures that only requests from Hasura are accepted.
- **Internal network**: These communications transit through internal Docker networks and are not exposed externally.

7.7 GDPR Compliance of the NutriPlatform

7.7.1 Introduction

The NutriPlatform, the technical component of the NutriBudget project, integrates concrete measures from its design phase to ensure compliance with the General Data Protection Regulation (GDPR). This approach is part of a continuous improvement process, in line with European requirements and industry best practices. This section describes the current state of implemented mechanisms, identified points of attention, and the improvement dynamics adopted by the project team.

7.7.2 Effective Measures in Place

The NutriPlatform implements several technical and organizational mechanisms to ensure the protection of personal data:

Access Security and Permission Control Access to all backend services of the NutriPlatform is strictly protected by strong authentication based on JWT tokens issued by Keycloak (OAuth2/OIDC protocol). A dedicated middleware systematically checks the validity of the token, the client association, and the presence of required roles or scopes. Any unauthenticated or insufficiently authorized access attempt is explicitly denied, with event logging.

Data Minimization and Traceability The platform applies the principle of data minimization: only information strictly necessary for operation and security is processed. API access logs are structured in JSON and include only the user's technical identifier (if available), without collecting unnecessary personal data. These logs are suitable for audit, anomaly detection, and traceability, while respecting confidentiality.

Input Validation and Error Management All user inputs are subject to strict validation, limiting the risks of injection or abusive exploitation. Application errors are returned in a structured format, with technical details masked in production to prevent any sensitive information leakage. A rate limiting system protects the API against abuse and denial-of-service attacks.

7.7.3 Points of Attention

GDPR compliance is the subject of ongoing attention, with several focus areas:

- Progressive implementation of data subject rights (access, rectification, erasure, portability) and processing records.
- Adaptation of consent management procedures according to the evolution of features and usage.
- Maintenance of up-to-date, accessible documentation in line with regulatory requirements.
- Regular review of logs, metrics, and alerts to ensure operational robustness and early detection of potential anomalies.

7.7.4 Continuous Improvement Approach

The project team adopts a proactive approach to continuous improvement in data protection:

- Existing mechanisms are regularly evaluated and strengthened with each major platform evolution.
- Security tests are extended and adapted according to new features or feedback.
- Technical and operational documentation is kept up to date to reflect the actual state of the system and facilitate auditability.
- GDPR compliance is monitored as an evolving process, with progressive adaptation to regulatory requirements and industry best practices.

This approach ensures that the NutriPlatform remains aligned with the security and data protection standards expected in a high-stakes European project.

8 Deployment

Deployment activities for the NutriBudget project have not started yet. This chapter is temporary and does not reflect any final technical decision. At this stage, **Docker Swarm** is being considered as the likely orchestration solution.

Once deployment begins, this section will cover:

- The infrastructure strategy selected (containerization, orchestration, networking, security, CI/CD, monitoring, backups)
- The tools, methods, and workflows that will be adopted (Docker, Swarm/Kubernetes/alternative, monitoring, backup, ...)
- All technical details, diagrams, and related procedures

A complete update will be carried out as soon as the project reaches the deployment phase and technical choices are validated. Until then, no advanced information is available.

9 Wireframes

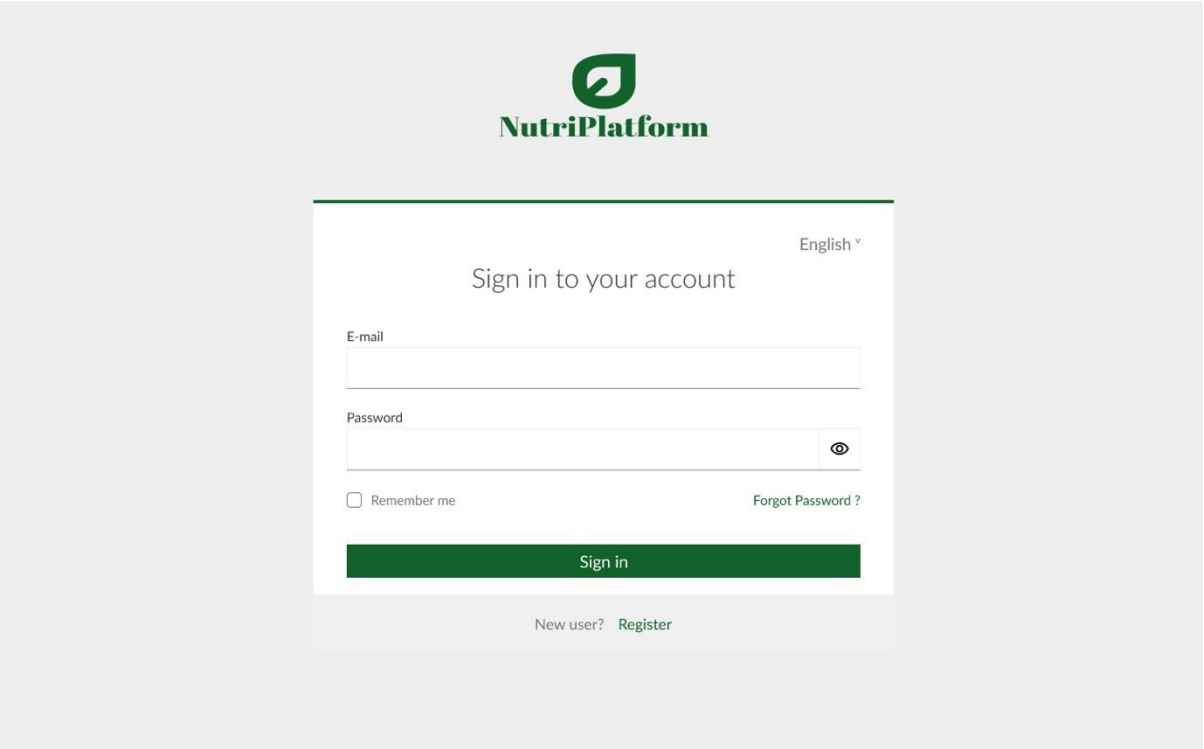

Figure 10. Login page

Login page with an email and password pair, and the necessary features to reset a password in case it is forgotten. Language selection is also available from this screen.

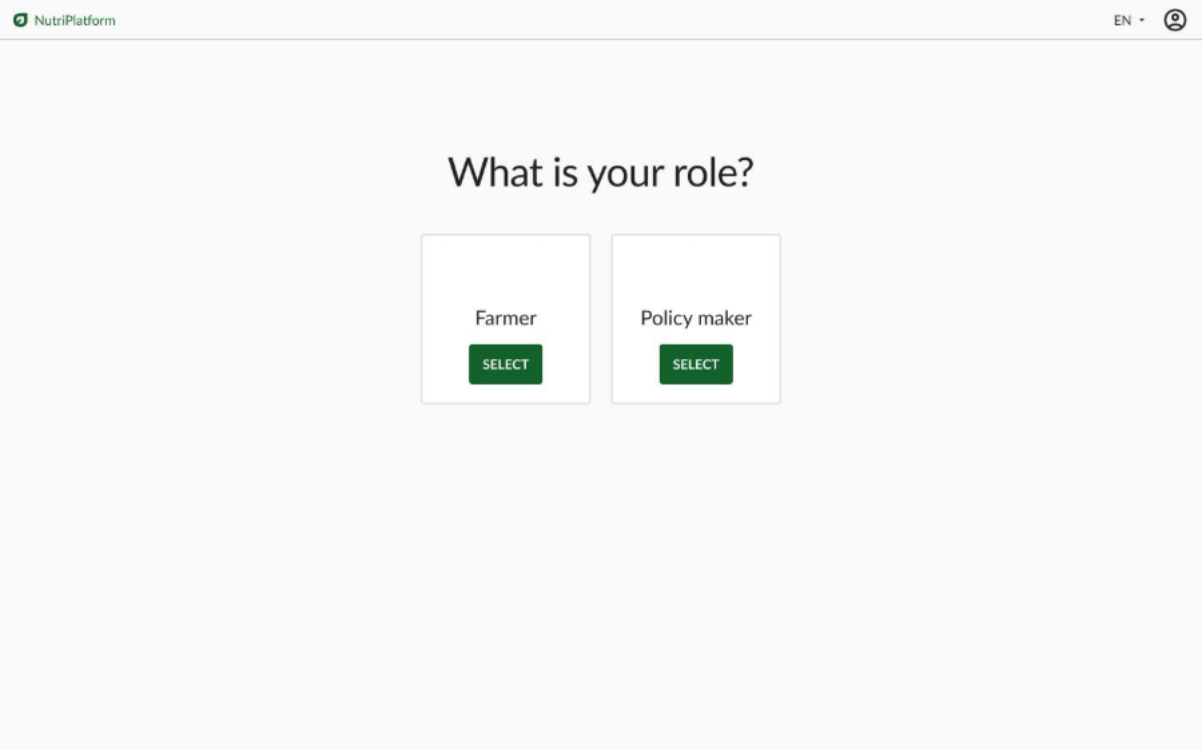

Figure 11. Role selector page

During the first login, we ask the user to select their default persona. They can then change their persona at any time via the topbar.

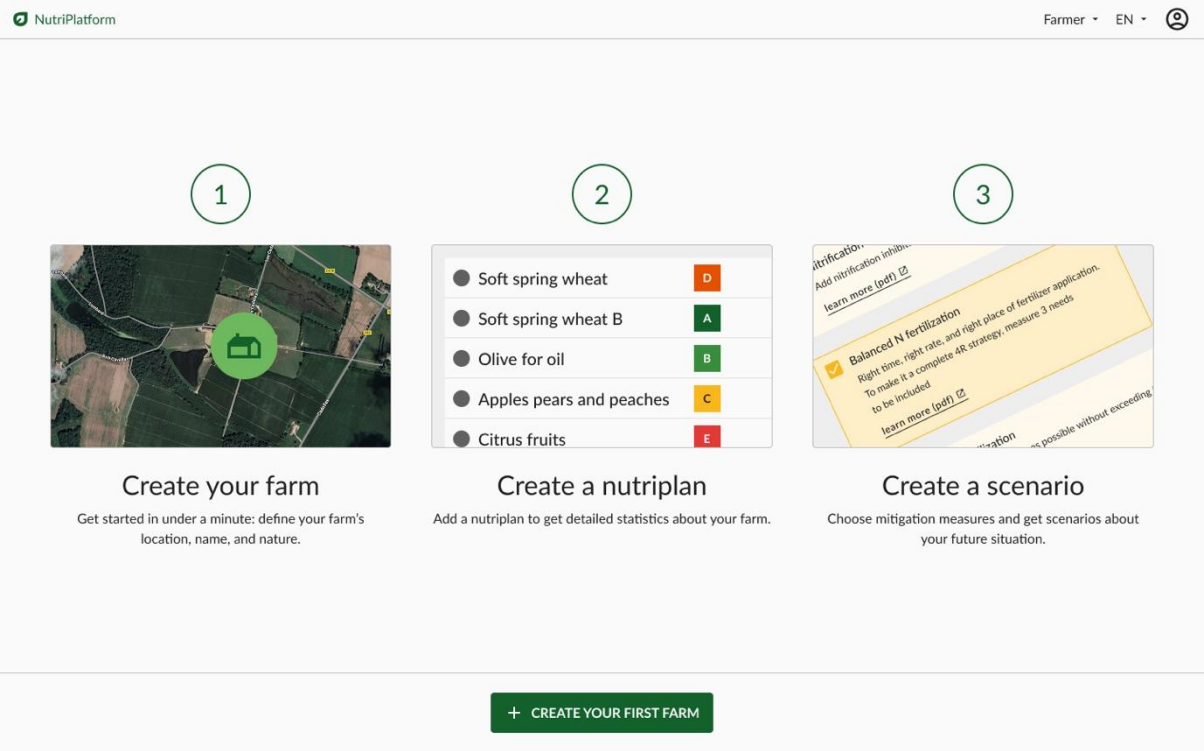

Figure 12. First connection

At first login, if the user is a farmer, he is guided through the main concepts of the NutriBudget platform: Farm, Nutriplan, and Scenario, to help him understand the structure and logic of the application. A call-to-action button encourages the users to create their farm.

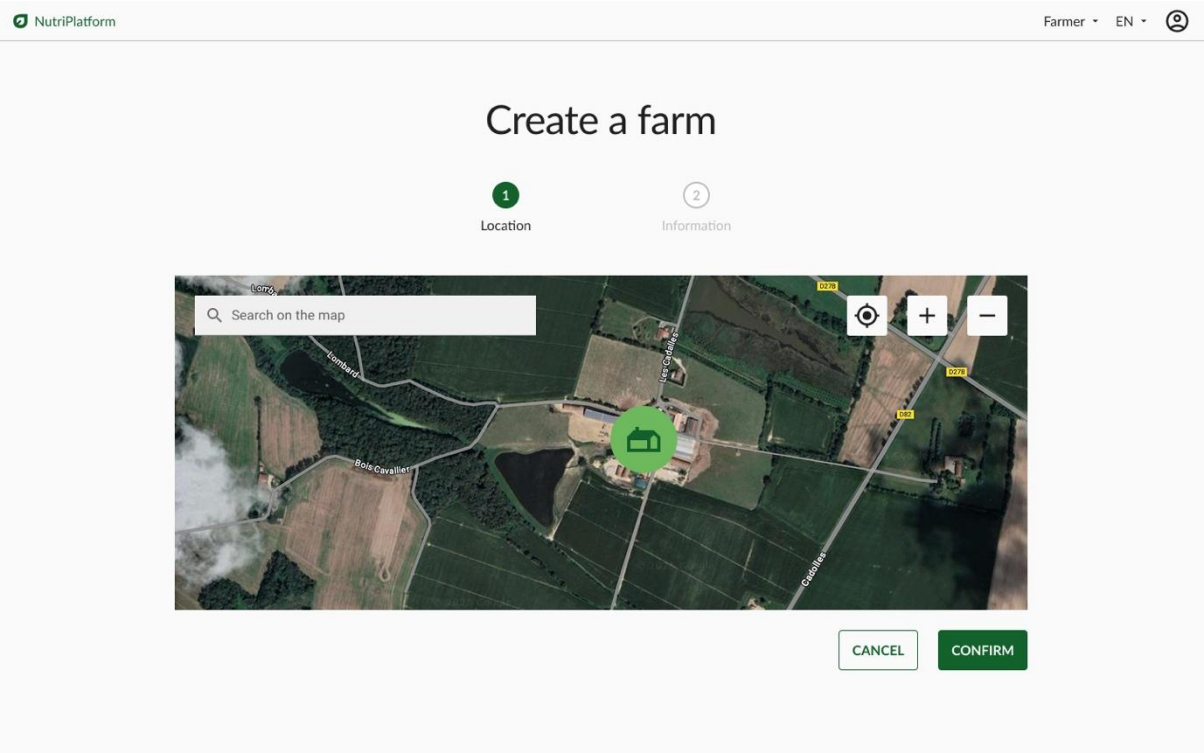

Figure 13. Farm creation, step 1

Farm creation is a two-step process. The first step involves using a map to locate the farm. Tools such as keyword search, geolocation, and zoom assist the user in this task.

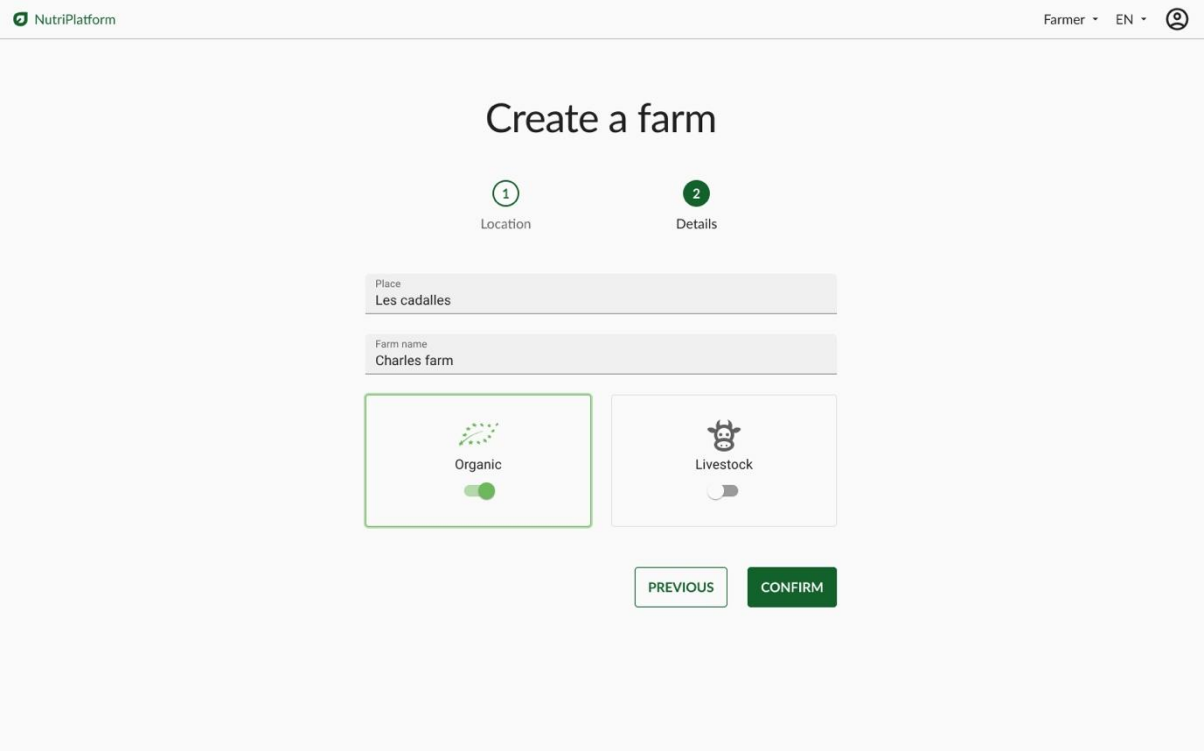

Figure 14. Farm creation, step 2

In the second step, we ask for some additional information: the name of the farm, whether or not there are animals, and whether or not the farm is organic.

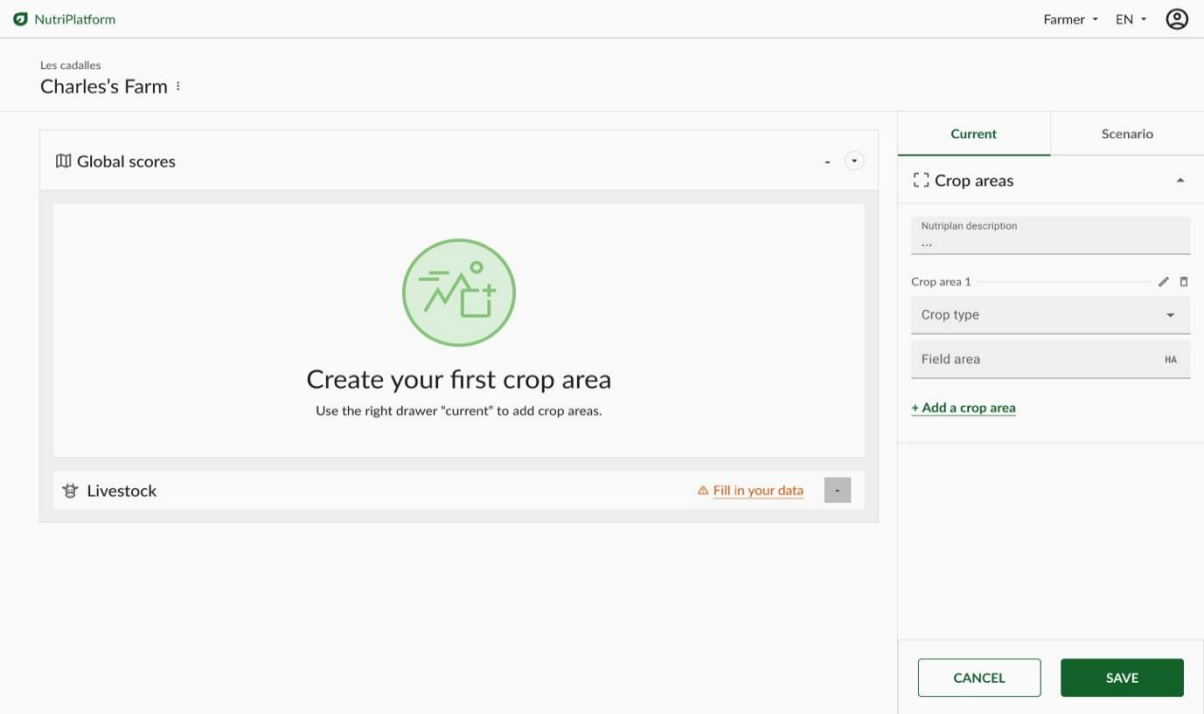

Figure 15. Nutriplan (no crop area)

Once a farm is created, the user is automatically redirected to an empty nutriplan linked to that farm. They are prompted to add cropland or livestock in order to obtain statistics about their operation.

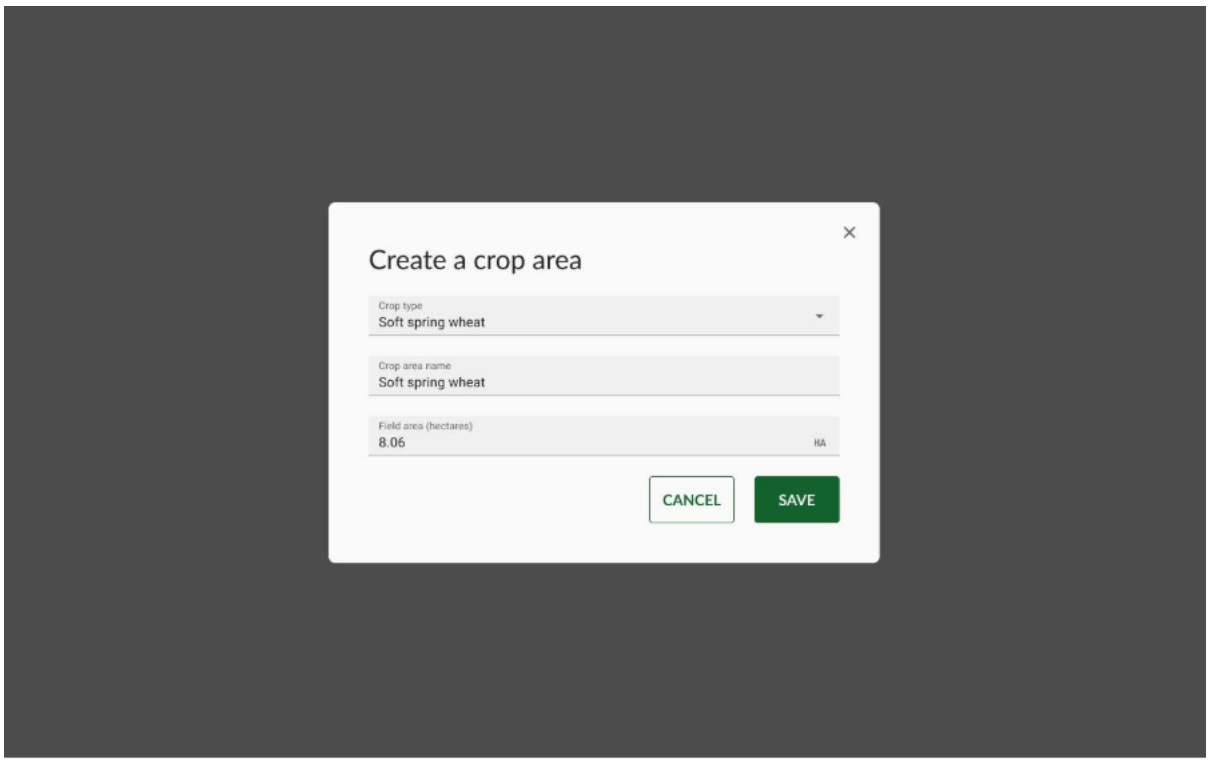

Figure 16. Crop area creation popup

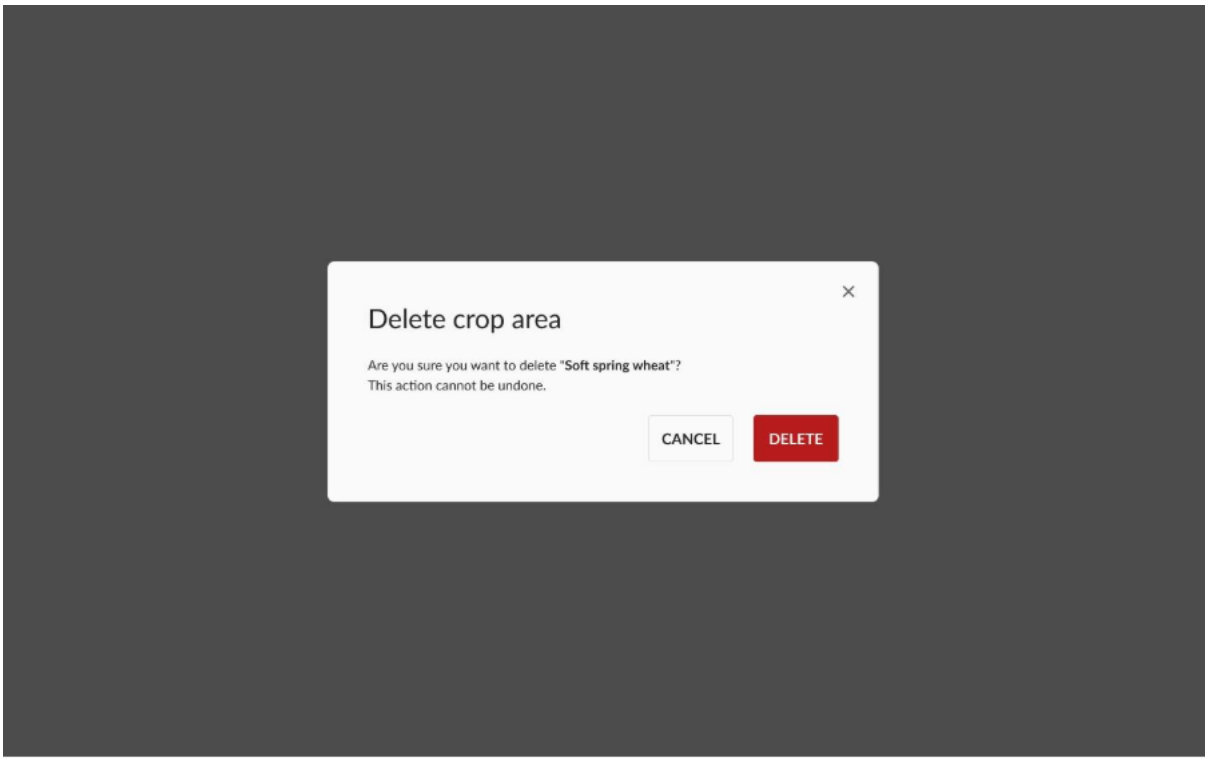

Figure 17. Crop area deletion popup

A pop-in window is triggered when the user wants to add, edit, or delete a crop area, providing a focused and streamlined interaction.

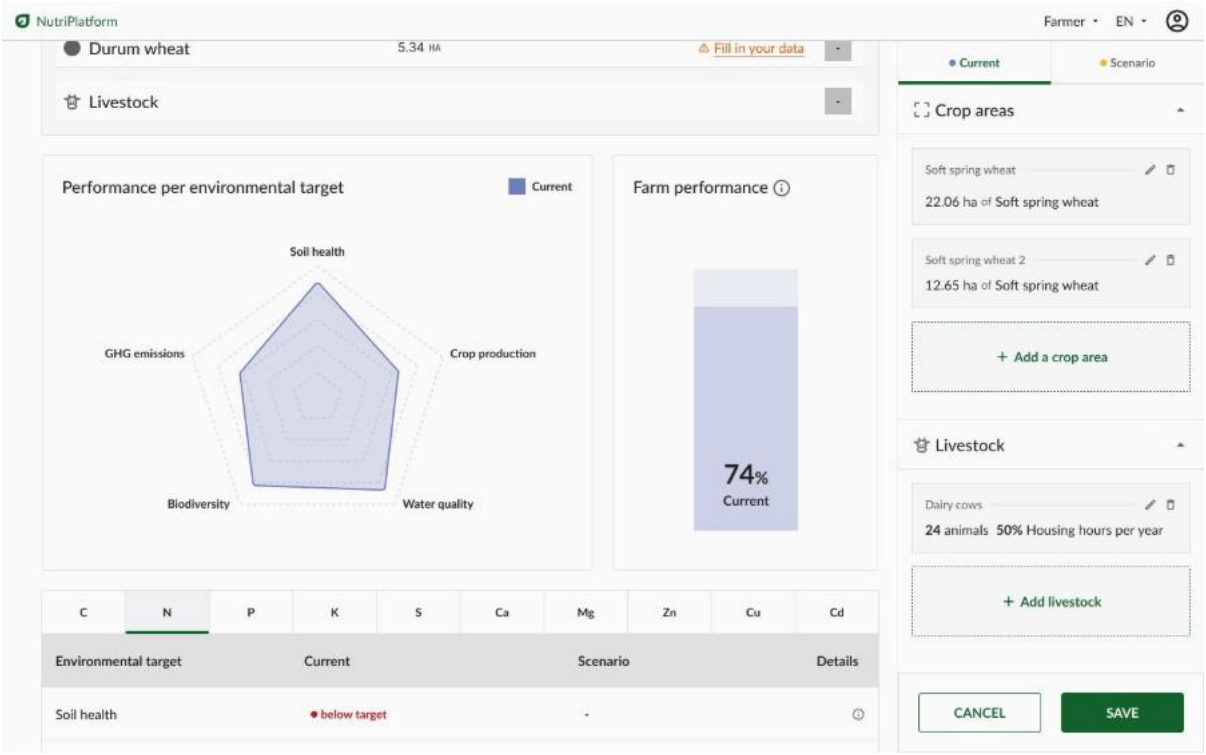

Figure 18. Nutriplan

Statistics for the farm are displayed based on the crop areas and livestock provided by the user.

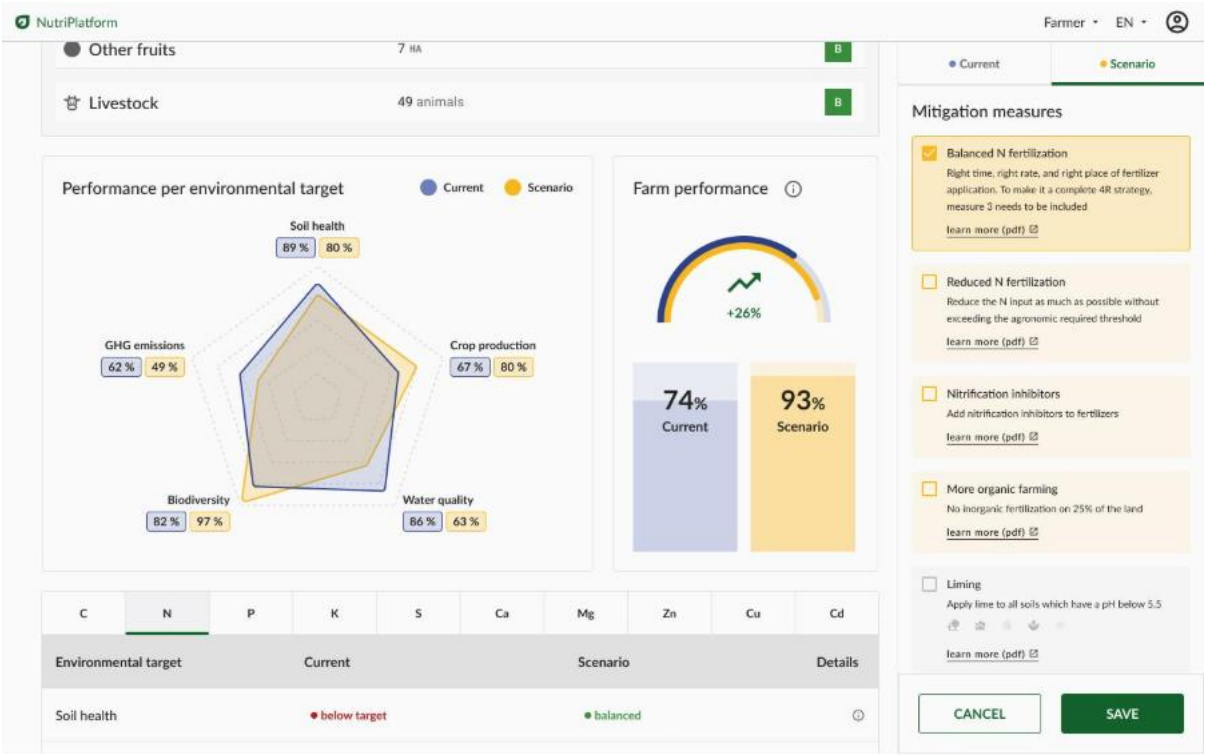

Figure 19. Nutriplan with senario (on scenario tab)

In the Scenario tab, the user has access to mitigation measures, which allow them to view additional statistics (displayed in yellow). Blue represents the current data, while yellow shows the projected data if the mitigation measures are applied.

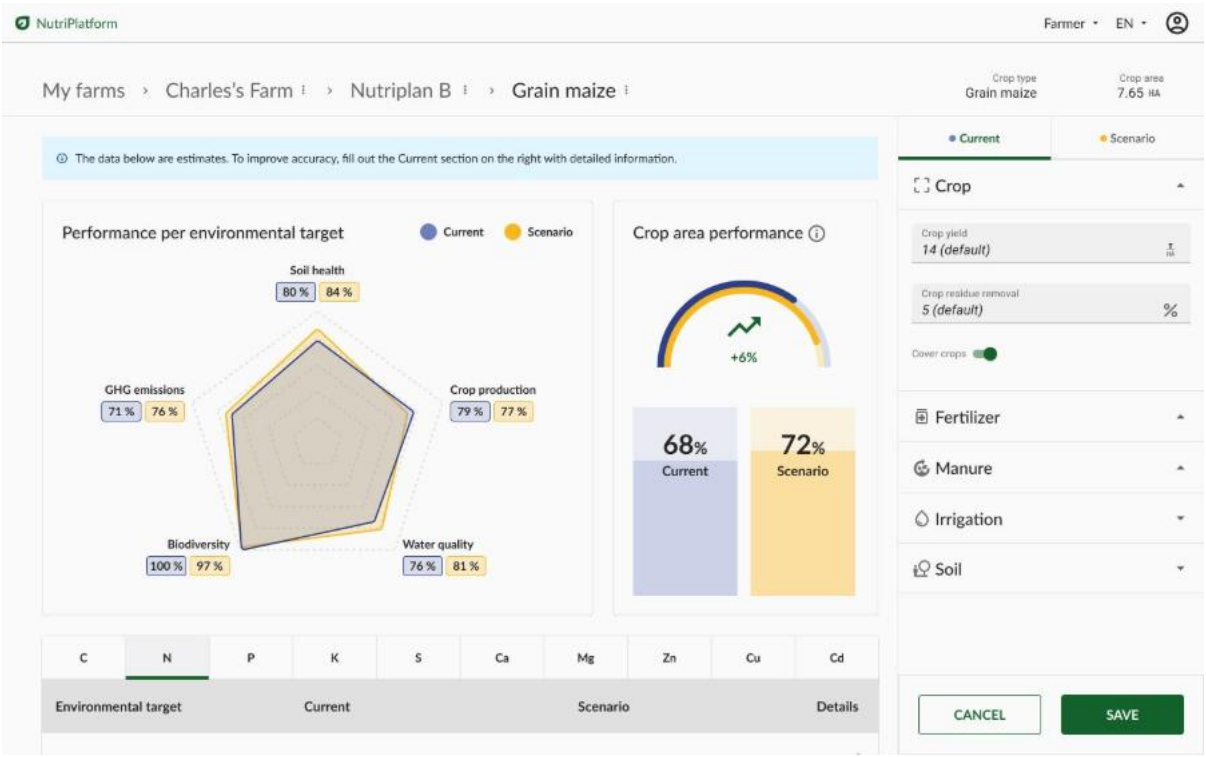

Figure 20. Nutriplan with senario (on current tab)

Beyond the overall farm statistics, users can also view "current" and "scenario" data for each individual crop area, allowing for more detailed analysis and comparison.

Users can use the breadcrumb navigation to easily move through the platform's structure: Home and farm list > Farm and nutriplan list > Nutriplan and crop area list > Crop area.

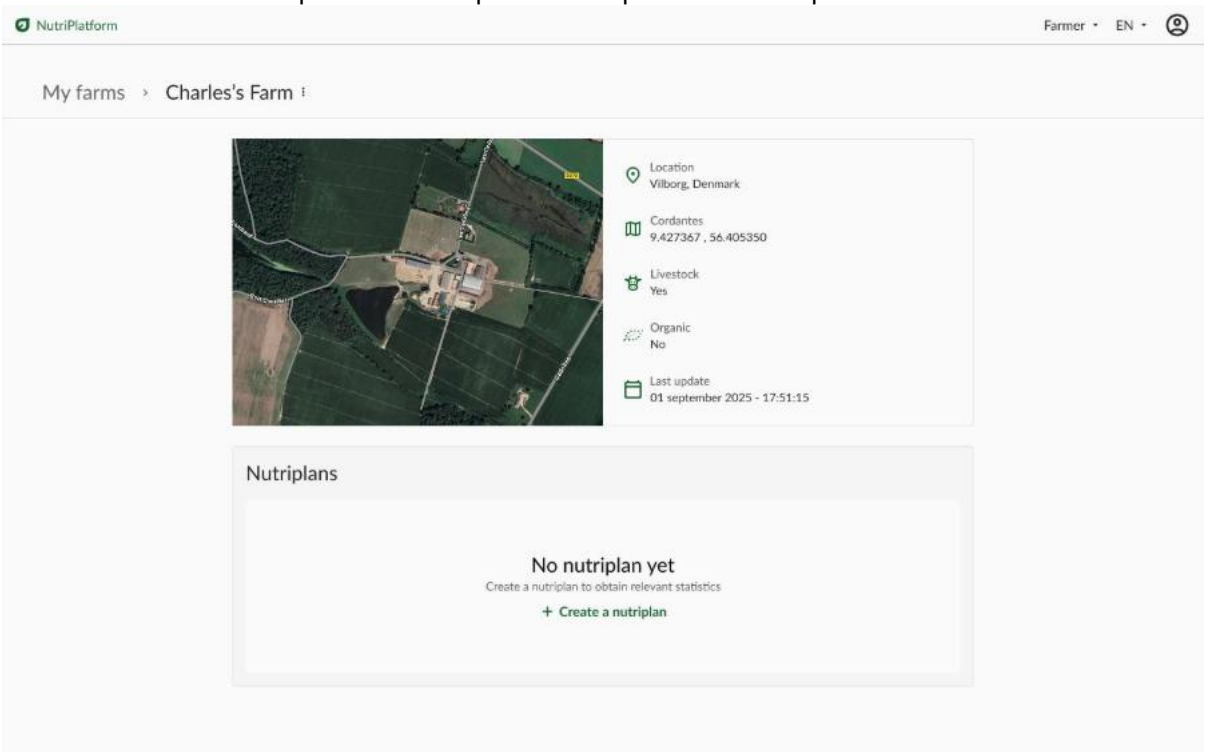

Figure 21. Empty **f**arm page (without nutriplans)

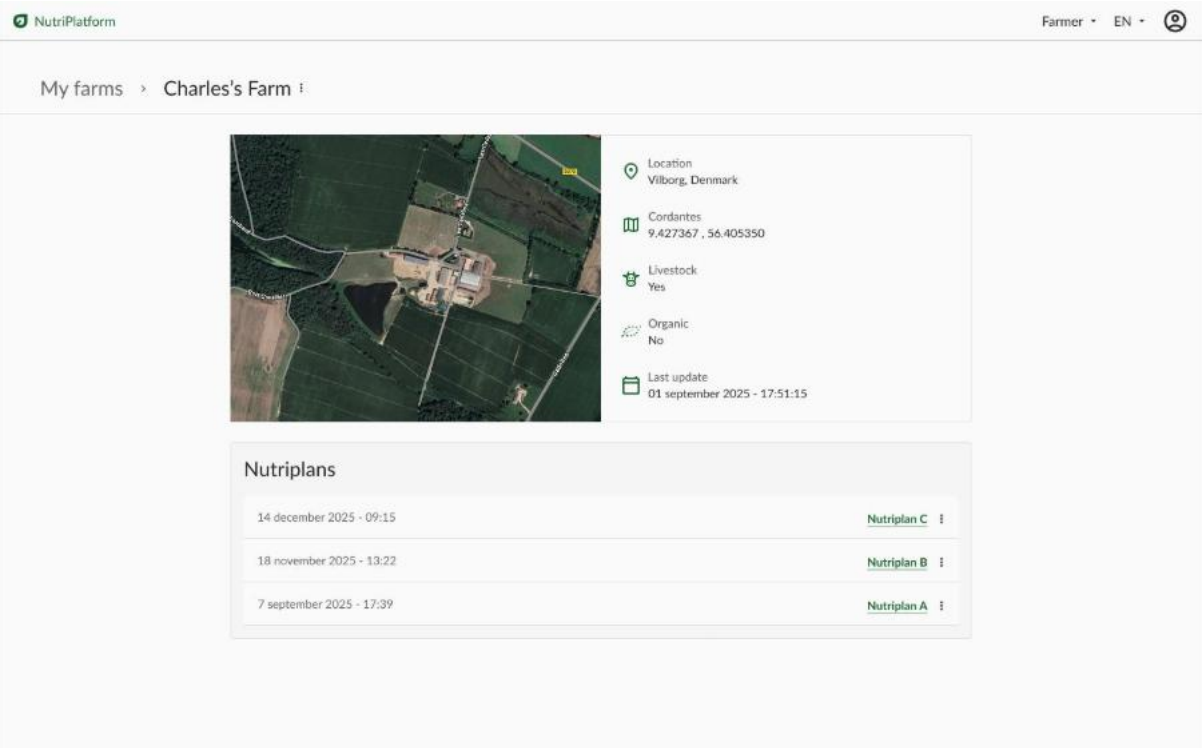

Figure 22. Farm page (with nutriplans)

The farm page displays key information about the farm as well as the list of associated nutriplans.

As a general rule, we intend to provide an empty state for every page where users need to create items, to guide and encourage them when no data is available yet.

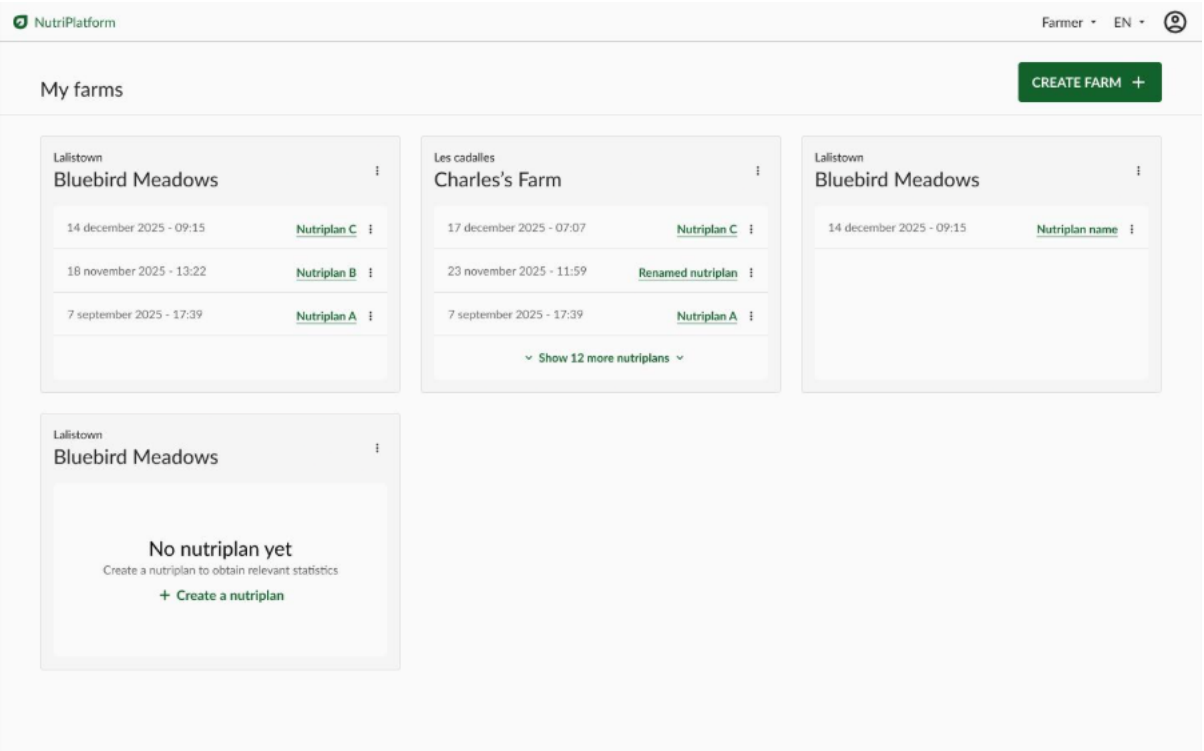

Figure 23. My farms

The home page for farmers also serves as the list of farms. There is a shortcut to quickly access the nutriplans for each farm.

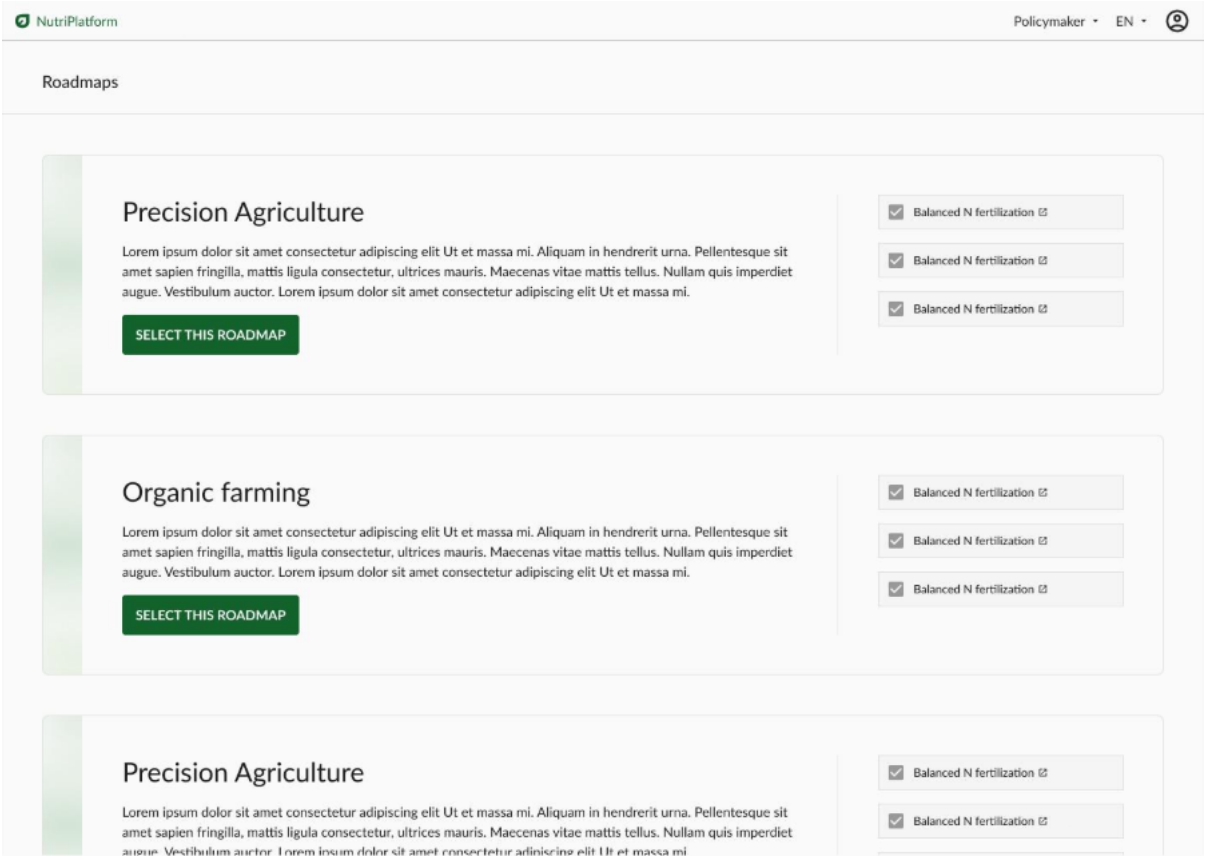

Figure 24. Roadmap selector for policy maker persona

This is the home page for the policy maker persona. Users can switch persona at any time using the topbar.

For policy makers, the home page displays the list of roadmaps. Each roadmap consists of a set of mitigation measures.

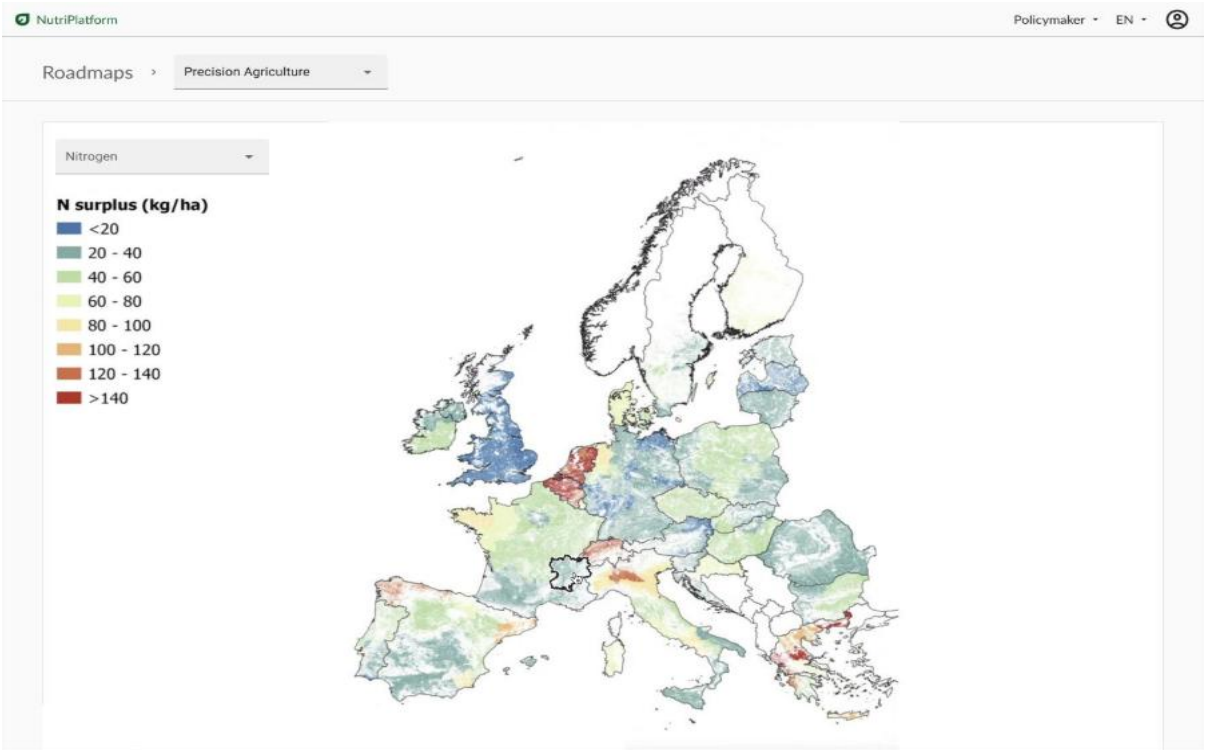

Figure 25. NUTS2 region selector for policy maker persona

When a roadmap is selected, a map appears, enabling the user to select a NUTS2 region.

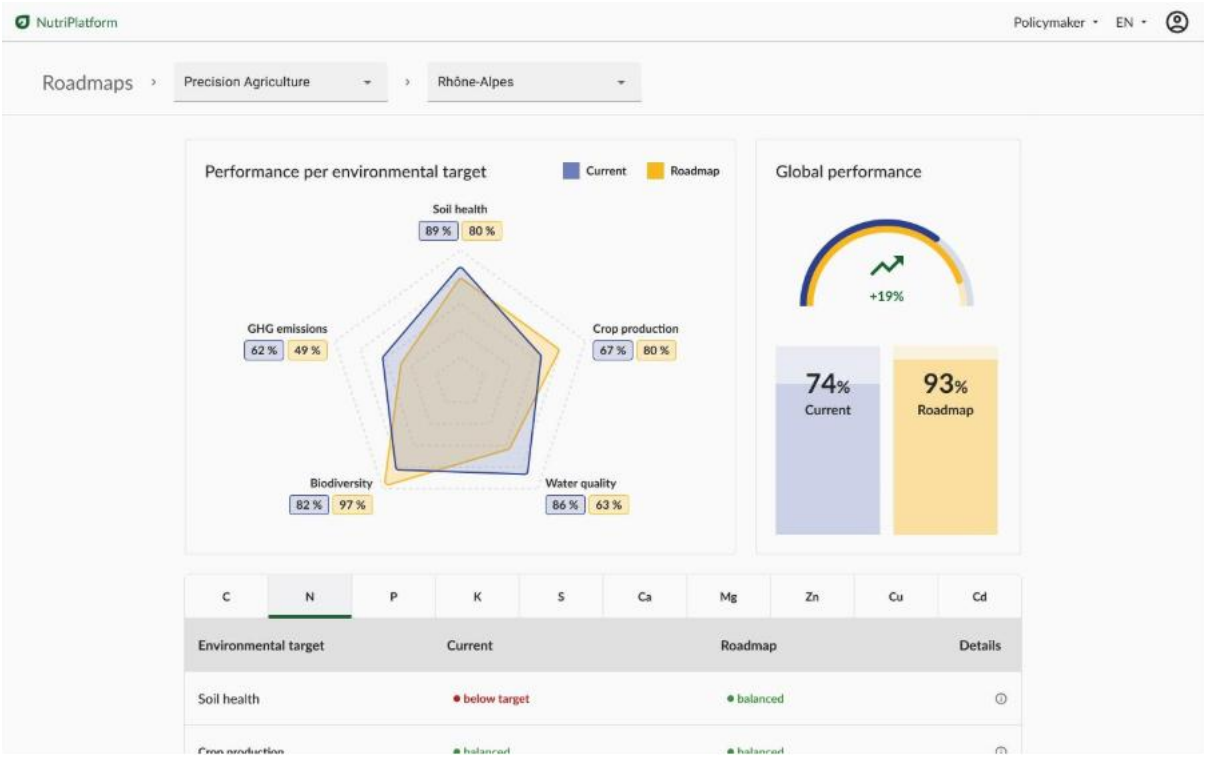

Figure 26. Statistics page for policy maker persona

After selecting a NUTS2 region, users are taken to a statistics page where they can compare current data with projections based on the selected roadmap.

Optimisation of nutrient budget in agriculture

Project Coordinators:

Prof. Dr. ir. Erik Meers, Erik.Meers@UGent.be

Dr. Ivona Sigurnjak, Ivona.Sigurnjak@UGent.be

Ghent University, Sint Pietersnieuwstraat 25, Ghent 9000, Belgium.

The Consortium:

Funded by the European Union