

École polytechnique de Louvain

EFMS: Experimental Farming Management System

An open-source web platform for the management of recursive sub-parcels and multi-year operation history at the UCLouvain experimental farm

Authors: **Alexandre WERY**, **Arnaud WERY**

Supervisor: **Hélène VERHAEGHE**

Readers: **Benoît DUHOUX**, **Jean VANDERDONCKT**

Academic year 2025–2026

Master [120] in Computer Science

Abstract

Farm management software can generally be classified into two categories. The first (Geofolia, Smag, ISAGRI) is designed to manage regulatory compliance and is therefore built around the official parcel defined in the Common Agricultural Policy declaration. The second (Agrivi, Farmbrite, Agworld) is focused on yield optimisation and financial management. Both share a common assumption: that the field is the fundamental unit and remains fixed throughout the season. This assumption does not hold on an experimental farm. At the Louvain-la-Neuve experimental farm, operated by UCLouvain, a single official parcel can be subdivided into smaller, often short-lived, zones, each carrying its own treatments and a multi-year history. No existing tool supports this workflow adequately.

This thesis presents EFMS (*Experimental Farming Management System*), an open-source web application designed to address this gap. EFMS supports recursive parcels at any depth, attaches operations to every level of the hierarchy, allows the user to query the history by attribute and by area, and shares specific parcels with external researchers without exposing the remainder of the farm. The backend is a Spring Boot service on top of PostgreSQL with the PostGIS extension. It performs shapefile import via GeoTools and uses JWT-based authentication. The frontend is a server-rendered React application (React Router 7) using Leaflet. Its interactive overlap-fix combines a bounding-box pre-filter, a layered overlap test and a polygon-difference computation based on the Martinez–Rueda algorithm.

The application was developed in collaboration with the farm staff over one academic year. It was validated through regular meetings with end-users and through a load-test campaign conducted on the class of low-power, single-board hardware on which it is likely to be deployed, at a request rate well above its expected usage. The result is a functional and readily deployable system. The remainder of this report presents the design, the algorithms, the validation rounds, the test results and the current limitations of the application.

Keywords: farm management information system, precision agriculture, experimental farm, geographic information system, PostGIS, Spring Boot, React, Leaflet, polygon clipping, recursive parcels, open source.

Résumé

Les logiciels de gestion agricole peuvent être classés en deux catégories. Les premiers (Geofolia, Smag, ISAGRI) sont destinés à la gestion de la conformité réglementaire et reposent donc sur la parcelle officielle, telle que déclarée à la Politique Agricole Commune. Les seconds (Agrivi, Farmbrite, Agworld) sont orientés vers l'optimisation du rendement et la gestion financière. Tous deux reposent sur une hypothèse commune : la parcelle est l'unité de base et demeure fixe durant toute la saison. Cette hypothèse ne tient plus sur une ferme expérimentale. À la ferme expérimentale de Louvain-la-Neuve, gérée par l'UCLouvain, une parcelle officielle doit pouvoir être subdivisée en zones plus petites, dont certaines éphémères, chacune portant ses propres traitements et un historique annuel. Aucun outil existant ne répond correctement à ce besoin.

Ce mémoire présente EFMS (*Experimental Farming Management System*), une application web open source conçue pour répondre à ce besoin. EFMS permet de gérer des parcelles imbriquées à n'importe quelle profondeur, lie les opérations à n'importe quel niveau de la hiérarchie, permet de récupérer l'historique par attribut et par zone géographique, et permet le partage sécurisé de parcelles ciblées avec des chercheurs externes. Le backend est un service Spring Boot reposant sur PostgreSQL avec l'extension PostGIS, intégrant l'import de shapefiles via GeoTools et la sécurisation par authentification JWT. Le frontend est une application React utilisant Leaflet, avec une correction interactive des chevauchements des parcelles en combinant un pré-filtre par boîte englobante, un test de chevauchement à plusieurs étapes et un calcul de différence de polygones basé sur l'algorithme de Martinez-Rueda.

L'application a été développée en collaboration avec le personnel de la ferme tout au long de l'année académique. Elle a été validée par des réunions régulières avec les utilisateurs finaux ainsi que par une campagne de tests de charge réalisée sur le type de matériel monocarte à faible consommation sur lequel elle est susceptible d'être déployée, à un débit de requêtes nettement supérieur à son usage attendu. Le système est aujourd'hui fonctionnel et facilement déployable. Le reste du rapport présente la conception, les algorithmes, les cycles de validation, les résultats des tests et les limites actuelles de l'application.

Mots-clés : système de gestion agricole, agriculture de précision, ferme expérimentale, système d'information géographique, PostGIS, Spring Boot, React, Leaflet, découpe polygonale, parcelles récursives, logiciel libre.

Acknowledgments

We would like to thank our supervisor, Professor H  l  ne VERHAEGHE, for her availability throughout the year, her patient guidance through the design decisions and her regular feedback, which kept the project well directed. The meetings with her shaped the report at least as much as they shaped the application.

We are grateful to our readers, Beno  t DUHOUX and Jean VANDERDONCKT, for accepting the task of reading and evaluating this manuscript.

A particular thank you to the staff of the Louvain-la-Neuve experimental farm and especially to Simon Vanderveken and Hugues Falys. They devoted many hours of their time, explained the realities of running an experimental site and patiently participated in the demonstration sessions. EFMS would not exist in its current form without them.

We would also like to thank them for agreeing to test the application during the validation rounds. Their feedback ensured the design remained aligned with the operational requirements.

Finally, we thank our family for their constant support throughout the year and for the times we could discuss the application and its features with them and benefit from their advice.

Use of generative AI. In accordance with the EPL master thesis regulation, the following details the use of generative AI in this project. Generative AI tools were used for two purposes. The first was assistance with the writing of this manuscript: rephrasing, English-language polishing and occasional suggestions regarding the structure of a section. The second was repetitive, low-complexity development tasks, such as refactoring suggestions, syntax queries and boilerplate generation. The tools were not used to design the architecture or to produce results.

List of Acronyms

API	Application Programming Interface
BBCH	Biologische Bundesanstalt, Bundessortenamt und CHemische Industrie (plant growth stage codes)
BCrypt	Blowfish-derived password hashing function
CAP / PAC	Common Agricultural Policy (Politique Agricole Commune)
CI	Continuous Integration
CORS	Cross-Origin Resource Sharing
CRS	Coordinate Reference System
CRUD	Create, Read, Update, Delete
CSV	Comma-Separated Values
CTE	Common Table Expression (SQL)
EFMS	Experimental Farming Management System
EPL	École polytechnique de Louvain
EPPO / OEPP	European and Mediterranean Plant Protection Organization
EPSG	European Petroleum Survey Group (geodetic registry)
ER	Entity-Relationship (diagram)
FMIS	Farm Management Information System
GDPR	General Data Protection Regulation
GIS	Geographic Information System
GiST	Generalized Search Tree (PostgreSQL index)
HMAC	Hash-based Message Authentication Code
HTTP / HTTPS	HyperText Transfer Protocol (Secure)
JPA	Jakarta Persistence API
JSON	JavaScript Object Notation
JTS	JTS Topology Suite
JWT	JSON Web Token
LLN	Louvain-la-Neuve
LTS	Long-Term Support
OGC	Open Geospatial Consortium
ORM	Object-Relational Mapping
PostGIS	Spatial extension for PostgreSQL
REST	Representational State Transfer
SQL	Structured Query Language

SRID	Spatial Reference System Identifier
SSR	Server-Side Rendering
SUS	System Usability Scale
UCL	Université catholique de Louvain (UCLouvain)
UI / UX	User Interface / User Experience
UML	Unified Modeling Language
WGS84	World Geodetic System 1984
WKT	Well-Known Text

Contents

Abstract	i
Résumé	ii
Acknowledgments	iii
List of Acronyms	iv
List of Figures	x
List of Tables	xi
1 Introduction	1
1.1 Context	1
1.2 Problem Statement	2
1.3 Motivation	2
1.4 Project Objectives	3
1.5 Our Contribution	4
1.6 Roadmap	5
2 State of the Art	7
2.1 Belgian and French regulatory tools	7
2.2 International commercial FMIS	8
2.3 Niche and academic projects	8
2.4 Comparative analysis	8
2.5 Gap analysis	9
2.6 Algorithms used in existing systems	10
3 Problem Statement and Requirements Elicitation	11
3.1 Methodology	11
3.2 Results of the elicitation	12
3.3 Non-functional requirements	13
3.4 Conclusion	14
4 User Stories	15
4.1 The Farm Manager	15

4.2	The Researcher	16
4.3	The Administrator	17
5	Features Overview	18
5.1	The interactive map	18
5.2	Parcels and sub-parcels	20
5.3	Anti-overlap detection and automatic fix	20
5.4	Crop periods and the agronomic record	21
5.5	Operation logging	21
5.6	The official phyto-product catalogue	22
5.7	Filtering and history retrieval	24
5.8	PAC import	26
5.9	Sharing and access control	27
5.10	Reference data, public farms and administration	29
5.11	User profile, tutorial, translations and documentation	31
6	Design	32
6.1	Design overview	32
6.2	Database design	32
6.2.1	Spatial entities	33
6.2.2	Operations and products	35
6.2.3	Users, farms and shares	36
6.2.4	Why not a graph database, or a document store?	37
6.3	Algorithmic choices	37
6.3.1	Overlap detection	38
6.3.2	Overlap fix through polygon difference	41
6.3.3	Why Martinez–Rueda and not Vatti or Sutherland–Hodgman	42
6.3.4	Spatial filtering on the server	42
6.4	Concluding remarks on the design	42
7	Implementation	43
7.1	Technology stack	43
7.2	System architecture	44
7.2.1	Frontend structure	45
7.2.2	Backend structure	45
7.3	Security model	46
7.4	Deployment	46
7.5	Code conventions	47

8	Validation and User Feedback	48
8.1	Iterative cadence	48
8.2	Validation rounds and what they changed	49
8.3	Lessons learned	50
8.3.1	Common mistakes to avoid	50
8.3.2	Recommended practices	51
8.4	System Usability Scale evaluation	51
8.4.1	Methodology	51
8.4.2	Results	53
8.4.3	Qualitative open-ended feedback	54
8.5	Current state of the validation	55
9	Testing and Metrics	56
9.1	Unit tests	56
9.2	Coverage	57
9.3	Deployment tests	58
9.4	Performance and load testing	59
9.4.1	Setup and method	59
9.4.2	Results	59
9.4.3	Interpretation	61
9.5	Behavioural and concurrency tests	62
9.6	Limitations and known gaps	62
10	Ethics, Legal and Sustainability Considerations	64
10.1	Personal data and GDPR	64
10.2	Data ownership and confidentiality	65
10.3	Intellectual property	65
10.4	Phyto products and health-relevant data	66
10.5	Sustainability	66
10.6	Out of scope	67
11	Conclusion and Perspectives	68
11.1	Summary of contributions	68
11.2	Objectives met and not met	69
11.3	Critical analysis	70
11.4	Limitations	71
11.5	Lessons learned	72
11.6	Perspectives	73

11.7 Final remarks	74
Bibliography	75
A Application screenshots	78

List of Figures

5.1	The interactive map and its controls.	19
5.2	Browsing the official phyto-product catalogue.	23
5.3	The operation-history view and its filters.	25
5.4	A generated PDF export of the operation history.	26
5.5	The PAC import review screen.	27
5.6	The farm shares screen.	28
5.7	Creating a research-zone share.	29
5.8	The Belgian PAC/SIGEC culture codes.	30
6.1	High-level architecture of EFMS.	32
6.2	Parcel hierarchy, farm scope and lifecycle status.	33
6.3	The <code>parcel_periods</code> association entity.	34
6.4	Provenance of imported rows.	35
6.5	Operations chain: parcels, operations, products and tools.	36
6.6	Users, farm membership and the sharing mechanisms.	37
6.7	Activity diagram of overlap detection and auto-correct.	40
8.1	Distribution of SUS responses per statement ($n = 4$).	54
9.1	Mean read latency of representative endpoints on the Raspberry Pi 4.	61
A.1	The parcels list.	78
A.2	Reference data: units, product types and operation types.	79
A.3	Farm products.	80
A.4	Farm tools.	80
A.5	Managing parcel periods.	81
A.6	The imports list.	81
A.7	Per-farm settings.	82
A.8	The user profile and preferences.	83

List of Tables

2.1	Fit of the surveyed farm management tools to the requirements of the LLN experimental farm. The criteria are the farm’s own needs; a cross denotes a mismatch with this use case, not a general deficiency of the tool.	9
7.1	Main technology choices in EFMS.	44
8.1	The ten statements of the System Usability Scale [30] used in this evaluation.	52
8.2	Aggregated SUS scores per participant.	53
9.1	Indicative coverage of the backend test suite, by area, on the main branch at the time of writing.	58
9.2	End-to-end latency of representative API calls on the Raspberry Pi 4.	60

Chapter 1

Introduction

This first chapter establishes the context for the remainder of the thesis. It presents the context of the work, the problem addressed, the motivation for the project, the objectives, the technical contribution and, finally, the structure of the document.

1.1 Context

Agriculture has changed substantially over the past three decades. Where the experience and judgement of the farmer once sufficed, software, sensors and extensive administrative reporting now account for a significant share of daily activity. The literature refers to this shift as “precision agriculture”. The European Union has actively promoted it through successive reforms of the Common Agricultural Policy [1]. For a Belgian farm, the practical consequences are well known to anyone responsible for the annual declaration: every parcel must be delineated, every crop linked to its parcel, and every fertiliser and phytosanitary product recorded. A substantial part of farm management has, in this respect, become an administrative activity.

For most commercial farms this poses no difficulty. The fields are large, change little from one year to the next and correspond to what was declared to the regional administration. A small number of established tools (Geofolia, Smag, ISAGRI) handle the regulatory aspects. A few supplementary spreadsheets are generally sufficient to cover what the dedicated tools do not.

The Louvain-la-Neuve experimental farm, operated by UCLouvain, differs fundamentally. It is, before being a farm, a research site: its objective is not to maximise yield but to produce reliable data on tillage, varieties, fertilisation, intercropping and related topics. What an external observer would regard as a single field is, in practice, composed of many smaller experimental zones. A PAC parcel of three or four hectares may contain ten, twenty or more sub-plots, each with its own protocol, treatments and history. These zones are not static: they change from one season to the next, and sometimes within a single season, according to the research being conducted.

Recording what was done, where and when, at this level of detail is the central

difficulty. A statement such as “Parcel 12 was sown with winter wheat in 2024” is insufficient, because it is far too coarse. The eastern half of parcel 12 may have received a low-nitrogen treatment in one experiment while the western half served as the control of another. If this information is lost, any comparison performed three or four years later becomes unreliable: a residual effect from a forgotten trial is precisely the kind of confound that invalidates a study. High-resolution spatial tracking is therefore not a refinement but a strict operational requirement.

1.2 Problem Statement

Existing farm management tools can generally be classified into two categories. The first addresses regulatory obligations: the PAC declaration, phytosanitary records and the traceability required by the administration. The second addresses scale: stock management, finance, planning and, in some cases, integration with farm machinery. Despite their differences, both rest on the same assumption: a parcel is the legal field shown on the official map, and remains that field for the entire season.

This assumption fails as soon as a sub-parcel is drawn within a parent boundary. Some tools permit such a subdivision, but their support is limited: nesting is restricted to a single level, history is not propagated, and the link to the PAC declaration is lost once the geometry ceases to match. The farm staff described the consequence in the first interview: a fragmented workflow, with one tool for the regulatory side, paper notebooks for the trials and a set of spreadsheets to consolidate everything at the end of the year, supplemented by considerable recollection to reconstruct what occurred on a given area of ground.

The cost is concrete. Errors are easy to introduce and difficult to detect, and the longer a parcel remains in use, the less its records can be relied upon as the scientific resource they are intended to be.

1.3 Motivation

The motivation for this thesis is both practical and methodological. The project concerns an actual research site with a clear, measurable and recurring operational need that made it a relevant subject for an applied engineering thesis. We approached it from a software-engineering background, having limited prior knowledge of how an experimental farm operates; the year-long collaboration with the farm staff therefore also constituted a substantial learning process, covering agronomy,

regulation and the interaction between the farm’s research agenda and its legal obligations.

The second motivation is the open-source one. Most of the existing farm management software is commercial: its subscription models are unsuited to a public research site, and its closed code cannot be adapted by the university to its specific needs. The aim was a system that the farm could continue to use after our departure, that a future maintainer could assume responsibility for and that other research farms could adopt should they encounter the same problem. That choice shaped the technical decisions throughout the project: standard open-source dependencies, a documented database schema, a containerised deployment and a public Git repository.

1.4 Project Objectives

EFMS, the *Experimental Farming Management System*, is the proposed system. It is an open-source web application, designed from the outset around the workflow observed at LLN rather than adapted from a commercial product. It was developed in collaboration with the farm staff throughout the year. Most of the design choices described in this report originate, directly or indirectly, from these meetings.

Concretely, the application is designed to provide the following:

- **Parcels with sub-parcels at any depth.** A parcel can have a parent. The link is recursive in the database, so we do not impose a fixed limit on the depth. The model can hold the official PAC parcel, the experimental block inside it and the small treated zone inside the block, all at the same time.
- **An accurate operation history.** Every operation (sowing, treatment, harvest, observation, etc.) is saved with its date, type, products, units and tools and attached to the parcel(s) it was done on. The history can then be searched by parcel, by period, by product, by tool, or by an area drawn on the map.
- **Spatial checks.** Whenever a polygon is drawn or reshaped, the application checks it against the parcels at the same level. When an overlap is found, the browser computes a correction from the polygon difference and displays it immediately. A toggle allows the user to disable this automatic correction and retain the geometry as drawn.
- **Sharing that keeps farm data and research data apart.** A farm holds administrative data that a researcher has no reason to see. The adopted

solution is a per-parcel share: an external user is granted a viewer or editor role on the parcels that concern them, and nothing else of the farm is exposed.

- **Long-term usability.** The application is intended to outlast the thesis. The frontend is a server-rendered React application in English and French, designed so that a new language can be added through configuration alone. The backend is a Spring Boot service backed by a documented, versioned database schema. Both are delivered with Dockerfiles and Docker Compose configurations, so that deployment remains straightforward.

1.5 Our Contribution

We developed the entire application over the course of the year. The backend is implemented in Java 17 using the Spring Boot 3.5 framework, on a PostgreSQL 15 database with the PostGIS 3.3 extension. The geometry is handled in JTS through Hibernate Spatial, so the spatial columns reside in the same model as the remainder; native SQL can be used directly when a spatial filter requires it. Authentication relies on JWT (HS256, with BCrypt-hashed passwords). PAC declarations are imported as zipped shapefiles [2] and read with GeoTools [3]; reprojection to WGS84 [4] is performed dynamically, the source usually being Lambert 72 [5], the Belgian projection.

The frontend is a React 19 application built on React Router 7 and Vite 6. The map is Leaflet, wired in through `react-leaflet` and `leaflet-draw`. The principal algorithmic work on the client is carried out by `martinez-polygon-clipping`: it subtracts the existing polygons from a freshly drawn one, and that subtraction is what enables the live auto-fix. Translations are handled through `i18next`. Each project includes a Dockerfile and a `docker-compose.yml`; initialising a new deployment requires only the database credentials and a JWT secret, which can be generated at run time.

The result is more than a prototype. The application is used during the validation meetings with the farm, the schema has been stable for several months, and the test suite covers the components where a regression would be most damaging, namely authentication, JWT handling and permission checks. The open-source codebase is maintained in two repositories on a public Git forge, separating the frontend¹ from the backend²; the deployment configuration resides in a third repos-

¹<https://github.com/W-EFMS/EFMS-frontend>

²<https://github.com/W-EFMS/EFMS-backend>

itory.³

1.6 Roadmap

The remainder of this thesis is organised as follows.

Chapter 2

surveys the existing farm management software, the regulatory tools used in Belgium and France, the international commercial platforms and the academic prototypes. It identifies the functional and non-functional gap that motivates EFMS.

Chapter 3

presents the methodology used to convert the farm's informal request into a concrete requirement list: the iterative meeting cadence, the artefacts retained from each round and the consolidated outcome.

Chapter 4

formalises those requirements as user stories, grouped by role (farm manager, user, administrator), identifying the requests deliberately left out of scope.

Chapter 5

walks through the features of the application as they are today, from the user's point of view, with a short note on the design choice behind each one.

Chapter 6

is the engineering chapter. It covers the database design, the algorithmic choices on the spatial side and the reasoning behind them.

Chapter 7

documents the technical implementation: the system architecture, the security model, the deployment and the technology stack.

Chapter 8

describes the validation strategy: the iterative meetings with the farm managers and a System Usability Scale evaluation conducted at the end of the development.

³<https://github.com/W-EFMS/EFMS-deploy>

Chapter 9

reports the automated tests, the coverage and the performance campaign run on single-board hardware.

Chapter 10

addresses the legal, ethical and sustainability considerations: GDPR, data ownership, intellectual property and the boundaries of the work.

Chapter 11

sums up the contributions, lists the objectives that were met and those that were not and discusses the directions in which the work could be extended.

Chapter 2

State of the Art

An analysis of existing farm management information systems (FMIS) was conducted during the early stage of the project. One conclusion emerged clearly from the first meeting with the farm: although many tools exist, none was used to the staff's satisfaction. This chapter therefore reviews the available systems, their strengths and their limitations in the specific context of an experimental farm, before the proposed design is presented.

2.1 Belgian and French regulatory tools

The first family of tools is the local one. **Geofolia** (from the Isagri group), **Smag Farmer** and **ISAGRI** are widely used in Wallonie and in northern France. They are designed in close relation with the public administration: they import the PAC declaration [6, 7], they organise it by “îlots” and parcels and they record the operations the farmer is required to declare, in particular phyto applications.

For a conventional farm these tools can be sufficient. The interface is designed around the official boundaries and cannot be easily adapted to the experimental farm's needs.

This is precisely where these tools prove inadequate for an experimental farm. The parcel they manipulate is the official one drawn on the regional map, and there is no genuine notion of a sub-parcel nested within it. In Geofolia, the nearest operation is to split a parcel or an îlot into separate pieces [8]: the original is not given a child but is replaced by several independent parcels, so neither the parent relationship nor the link to the PAC declaration survives the cut. To our knowledge, and consistent with the farm's own use of these tools, none of them maintains a parcel hierarchy in which a sub-parcel keeps its own history while remaining tied to its parent. The LLN farm therefore continues to use Geofolia for the legal declaration and maintains all research-related data separately, on paper or in spreadsheets. EFMS aims to eliminate this parallel workflow.

2.2 International commercial FMIS

The second family is broader and more recent. Tools such as **Agrivi**, **Farmbrite** and **Agworld** [9, 10, 11] are cloud platforms aimed at medium and large farms worldwide. They provide a map, parcel management, operation logs, financial dashboards and, in some cases, stock or machinery management. They are visually polished, integrate with weather services and are oriented towards yield optimisation rather than legal compliance.

For the present use case, these tools present two problems. The first is regulatory: the Belgian declaration formats are not natively supported and no automated PAC import is provided, so producing an acceptable declaration for the regional administration would require significant manual effort. The second is more structural. As with the regional tools, the field is the smallest unit in the data model. The requirement that a researcher subdivide a field into smaller, sometimes short-lived zones and then retrieve several years of history for a single one of them is, to our knowledge, absent from these platforms. In addition, they are subscription-based [9, 10, 11], which is incompatible with the requirements of a public research institution seeking to retain its own data locally and over the long term.

2.3 Niche and academic projects

A third, more diffuse group includes domain-specific add-ons (digital field notebooks, phyto stock managers such as the official tools at fytoweb.be and corder.be) and academic prototypes built for one specific research project. Surveys of the FMIS landscape describe it as fragmented, with most systems addressing a single function or production type rather than the full management cycle [12, 13]. These niche tools tend to be highly precise on a narrow problem (computing a treatment dose, recording a trial) but rarely cover the complete cycle of parcel, operation, sharing, history and declaration. They also tend to become unmaintained once the project that produced them ends, which is precisely the outcome the design of EFMS sought to avoid.

2.4 Comparative analysis

The table below summarises how the surveyed tools fit the requirements identified during the meetings with the farm. It measures fit to this specific use case rather than ranking the tools in general: the criteria are the LLN farm's own requirements,

and several of them (per-micro-plot history, polygon-based search, automated overlap correction) are particular to an experimental-farm workflow rather than recognised shortcomings of a commercial FMIS. A criterion was retained only if it was raised in the discussions; a financial dashboard, for instance, exists in Agrivi but was never requested by the farm and was therefore not included. A cross (×) thus marks a mismatch with this farm’s needs, not a defect in the tool. The assessment draws on the vendors’ public product documentation [6, 7, 9, 11, 10], consulted in 2026, together with the farm’s first-hand use of Geofolia; the commercial products were not trialled feature by feature, so the matrix reflects documented capabilities rather than exhaustive hands-on testing.

Table 2.1: Fit of the surveyed farm management tools to the requirements of the LLN experimental farm. The criteria are the farm’s own needs; a cross denotes a mismatch with this use case, not a general deficiency of the tool.

Functionality	Smag Farmer	Geofolia	Agrivi	ISAGRI	EFMS (this work)
Regulatory compliance (BE/FR)	✓	✓	×	✓	partial
PAC import (shapefile)	✓	✓	×	✓	✓
Hierarchical sub-parcels	×	split only	×	×	✓ (recursive)
Operation history per micro-plot	×	×	×	×	✓
Filtering history by period/product/tool	limited	limited	✓	limited	✓
Polygon-based history search	×	×	×	×	✓
Per-parcel sharing with external users	×	×	×	×	✓
Automated overlap detection / fix	×	×	×	×	✓
Self-hosted, open-source	×	×	×	×	✓

2.5 Gap analysis

As the table shows, the element missing for the LLN farm is not a single isolated feature. PAC import exists in Geofolia, polygon drawing in Agrivi and history filtering in most tools, and for the conventional farms these products target they are, on the farm’s own account, adequate. What is missing, for the experimental-farm case, is the combination: a tool that supports a recursive hierarchy of parcels, attaches an operation history to each level of that hierarchy, allows that history to be queried both by attribute and by area, and can be shared in a controlled manner with researchers without exposing the full administrative dataset. None of the systems surveyed provides all of these capabilities simultaneously; the academic

projects identified cover at most one or two of them.

The remainder of the gap is non-functional. The thesis imposed its own constraints: the result had to be open, self-hosted, free of any commercial subscription and simple enough for a maintainer to operate once the project ended. That single requirement excludes most of the cloud platforms, regardless of their feature sets.

2.6 Algorithms used in existing systems

From a user’s perspective, the geospatial functionality in these tools appears limited. Map filtering rarely extends beyond a bounding box, and overlap handling is, in effect, delegated to the administration: the official map is free of overlaps by construction, so the tool is not required to verify it. Containment of a sub-shape within its parent, where it is handled at all, reduces to a point-in-polygon test against the parent, and is in some cases omitted entirely. The internal storage engines of the commercial tools are not publicly documented.

EFMS has to be more careful on this front, because the user is constantly drawing, splitting and resizing parcels and because the parent of a sub-parcel is itself an editable polygon. EFMS uses two engines in parallel: PostGIS [14] on the server for the spatial history queries (`ST_Intersects`, `ST_MakeEnvelope` and `ST_GeomFromText` are used directly in the native queries) and the `martinez-polygon-clipping` JavaScript library on the client for the interactive overlap-fix. The library implements the Martinez–Rueda algorithm for boolean operations on polygons [15], which provides a fast, well-established method for computing the difference between a freshly drawn polygon and the existing ones; an earlier alternative is Vatti’s algorithm [16], and the broader family of geometric intersection problems is summarised in [17]. The full description is given in Section 6.3; the relevant observation is that existing FMIS do not require such logic. That absence is one of the reasons they cannot serve the present use case.

For a wider view of the FMIS landscape, recent surveys [12, 13] discuss the typical reference architectures and the historical adoption patterns. Cloud-based designs [18] have been proposed for general farms but, as discussed above, do not address the recursive sub-parcel use case that is at the core of an experimental farm. Adoption studies [19] also confirm that most existing tools target large commercial farms; a tool aimed at a research site has to be evaluated against a different set of criteria.

Chapter 3

Problem Statement and Requirements Elicitation

The previous chapter established that the existing tools are not suited to the LLN farm’s workflow. This chapter addresses a distinct concern: the process by which that observation was converted into a concrete set of functional requirements. It is treated separately from the user stories of the following chapter because the methodology of problem identification is itself a contribution of this work. Accurate problem definition is critical to the success of a project of this kind, and the effort it required would not be apparent from the final requirements alone.

3.1 Methodology

The definition of requirements followed a hybrid approach: an initial, in-depth interview with the farm managers and our supervisor early in the year, followed by a sequence of brief demonstration-driven meetings with our supervisor, alongside extended demonstration-driven interviews with the farm staff during development. This is the same pattern used in user-centred design and, more generally, in agile software engineering: the early interview fixes the direction and the demonstrations refine the details once the user has a concrete artefact to evaluate.

Stakeholder identification. Three groups were identified:

- The **farm staff**, represented by the manager of the LLN experimental farm and by Benjamin, the agronomist who also tests the application. They are the primary users of the application.
- The **researchers**, mostly PhD students and post-docs from the UCLouvain agronomy departments, who need access to the operation history of the plots that host their experiments. They are the secondary users.
- The **maintainers**, who will host the application after the thesis ends. They are not direct users of the features but they shape the non-functional requirements (deployment, backups, monitoring).

Data collection. We used two complementary methods.

The first was a series of **open-ended interviews** with the farm staff and with Benjamin, held during the autumn of 2025. Each interview was structured around a small number of seed questions (“how do you currently track an operation?”, “what would you want to find back about a plot in three years?”). The interviews were not recorded, but extensive notes were taken, which are kept in the **notes/** folder of the project repository.

The second was a series of **demo-driven validation meetings**, held every two weeks during the development phase. Each meeting began with a short demonstration of the latest state of the application, followed by an open discussion. The list of these meetings and the resulting action items is given in Chapter 8.

From data to requirements. After every meeting, the notes were re-read and the action items were extracted. Each action item was either converted into a concrete user story (recorded in the project board) or, when it raised a deeper question, recorded as a follow-up topic for the next meeting. The user stories were grouped by user profile, prioritised and used to plan the next sprint. The consolidated outcome of this process is the user-story list presented in Chapter 4.

3.2 Results of the elicitation

The output of the requirements definition phase, before any code was written, was a list of user stories. We summarise the four themes here; the formal user stories are in the next chapter.

Hierarchical parcels. The single most consistent request, raised in the very first interview and confirmed in every subsequent meeting, was the ability to define parcels inside other parcels, at arbitrary depth. The farm uses, in practice, two or three nesting levels, but after discussion, the conclusion was clear: no fixed limit should be coded into the application, since a new research protocol may, one day, need a fourth level.

Operation history. The second theme was the operation log. The farm wanted to record what was done on each parcel, when, with which product, in which dose, with which tool and by whom. The product and tool lists had to be editable by the farm itself (no completely fixed catalogue) and the history had to be queryable

in three ways: by parcel, by attribute (period, product, tool) and by geographical area (a polygon drawn on the map).

Spatial integrity. The third theme arose from a practical issue: occasionally, a plot is drawn on top of an existing one by mistake. The application was required to detect this and propose a correction. After the second meeting, it was agreed that this rule must not be a hard block: agronomists sometimes draw overlapping geometries deliberately, and the application must accept that case.

Controlled sharing. The fourth theme was access control. The requirement of the farm staff was straightforward to state: provide a researcher with the plots belonging to their experiment, and with no other part of the farm. The solution evolved over the year. The initial proposal was a read-only link requiring no authentication; this approach was rejected during stakeholder validation, because such a link can be neither revoked nor attributed to a specific user. The named-account, per-parcel model used by the application today is the outcome of that discussion.

3.3 Non-functional requirements

Alongside the functional requirements, three non-functional ones recurred consistently and shaped the architecture from the outset.

Self-hosted, open-source. Our supervisor insisted on a solution that the university could host on its own infrastructure, with no commercial subscription and no third-party SaaS in the loop. This rules out most of the cloud-based FMIS surveyed in Chapter 2.

Maintainable after the thesis. The application must remain operable by its future maintainers without the involvement of the original authors. In practice this requires a documented database schema, a containerised deployment, a public Git repository under a permissive licence and code that follows standard framework conventions. The objective is for a future student, or any subsequent maintainer, to become productive quickly rather than spend an extended period understanding the codebase.

Bilingual. The application requires bilingual support since the farm staff work in French while the academic documentation is in English. To achieve this, the

frontend separates the codebase from the localisation assets, allowing new languages to be added purely through configuration updates without modifying the source code.

3.4 Conclusion

The elicitation work confirmed that the LLN farm has a genuine, well-defined need that no existing tool covers and produced a list of requirements detailed enough to start designing the application. It also produced two secondary findings worth noting. First, the discrepancy between the stated requirements and the actual operational requirements became apparent only after the first demonstration, which confirmed the value of demonstration-driven meetings following the initial interviews. Second, several requirements that appear to be simple feature requests carry, in fact, deep design implications: the choice between blocking and auto-correction on an overlap, the choice between anonymous and named sharing, the choice of how deep the parcel hierarchy may extend; each of these is small in code but large in consequences, and each was decided in consultation with our supervisor and the farm rather than by the authors in isolation. The remainder of the report carries these decisions through to the implementation.

Chapter 4

User Stories

The user stories presented below are derived directly from the requests expressed repeatedly during the meetings with the farm staff. The standard “As a [role], I want [...] so that [...]” format was retained because it requires the rationale for a feature to be stated, not only its function. The stories are grouped by user profile.

4.1 The Farm Manager

The farm manager is the primary user of EFMS, responsible for maintaining the parcel layout, logging operations, preparing the annual PAC declaration and granting access to researchers. The stories below are those expressed most frequently.

- **As a Farm Manager**, I want to draw and edit my parcel boundaries on a map so that what is in the application matches what is in the field, including the small zones we use for trials.
- **As a Farm Manager**, I want to import the official PAC declaration as a shapefile so that I do not have to redraw boundaries the administration has already approved and so that my parcels stay linked to the declaration they came from.
- **As a Farm Manager**, I want to split a parcel into sub-parcels and split those again, so that experimental zones can be linked to an official PAC parcel without breaking the link to the declaration.
- **As a Farm Manager**, I want the application to propose a fix when I draw a parcel that overlaps an existing one, so that I do not silently create a conflict, while retaining the option not to apply this automatic fix.
- **As a Farm Manager**, I want to log an operation (sowing, fertilisation, treatment, harvest, observation or others) on one or several parcels at once, with the products, units and tools that were used, so that the history is complete enough for both the legal declaration and the research follow-up.

- **As a Farm Manager**, I want to select phyto products from the official Belgian catalogue, with their authorisation number and validity dates, so that what I record matches the legally registered products instead of free-typed names.
- **As a Farm Manager**, I want to see the operation history of a given parcel, filtered by period, product, tool, or by an area I draw on the map, so that I can answer questions such as “what has happened on this corner of field 12 over the last three years?”.
- **As a Farm Manager**, I want to export the operation history I am viewing to an Excel sheet or a PDF report so that I can keep an offline record and provide a printable document to the administration.
- **As a Farm Manager**, I want to invite individuals (other staff, researchers) to my farm with a clear role (admin, editor, viewer) so that I retain control over who can modify which data.
- **As a Farm Manager**, I want to share specific parcels (and only those) with a researcher so that he or she can directly access the data without seeing the remainder of the farm.

4.2 The Researcher

The researcher is the secondary user of the application: the data is produced by the farm but consulted by individuals outside it. The stories below are derived from the farm staff, who reported the typical requirements of their researchers, and from the meeting dedicated to external access.

- **As a Researcher**, I want to draw experimental sub-parcels inside the field that hosts my trial so that farm managers and I can track experimental zones on a stable spatial unit, even if the official boundary changes.
- **As a Researcher**, I want to obtain the full historical timeline of my plots, filtered by year and by treatment type, so that I can readily gather the data I need without depending on hand-kept records.
- **As a Researcher**, I want to export the filtered history of my plots to a spreadsheet so that I can analyse it in my own tools without re-typing anything.

- **As a Researcher**, I want to be given access to my plots without having to navigate the farm's full administration interface, so that I see only the data that concerns me and so that I do not accidentally modify anything else.
- **As a Researcher**, I want the labels of the products I see (cultures, treatments, units) to follow standard codes (EPPO [20] for cultures, BBCH [21] for stages) so that the data is comparable with what other research groups produce.

4.3 The Administrator

The administrator is the technical role, responsible for maintaining the deployment. The stories are few, because the application is intended to require little maintenance once installed, but those that exist had a concrete influence on the design.

- **As an Administrator**, I want to manage user accounts and enable a validation mode for new farms so that the deployment cannot be overrun by uncontrolled registrations.
- **As an Administrator**, I want to refresh the official phyto-product catalogue from the federal source so that the regulated reference data the farm relies on remains up to date.
- **As an Administrator**, I want the deployment to fit into a docker-compose file with a small set of environment variables so that a maintainer can host it on a standard server without having to reverse-engineer the project.
- **As an Administrator**, I want the application to expose a backup mechanism (a clear-text export of the database content as well as a binary dump) so that the data survives a server replacement.

The features that address these stories are presented, with screenshots, in Chapter 5.

Chapter 5

Features Overview

This chapter presents the features of EFMS in their current state, from the farm manager’s perspective. Each feature is described together with the user story it addresses and, where relevant, the design choices that shaped it. The description focuses on what the user observes in the application rather than on the underlying code; the architectural details are deferred to Chapter 6 and Chapter 7.

5.1 The interactive map

The map is the central feature of the application. On opening it, the user first selects a farm, after which an interactive Leaflet view loads the parcels of that farm. The view is initially centred on the LLN farm coordinates, a minor but useful choice for the primary user: the map opens on a familiar area, so a researcher who has been granted access does not have to locate their plot manually.

The page is divided into three zones. The main zone is an interactive map on which the parcels are displayed; the second is a list of the parcels of the selected farm, on the left of the map; the third is a menu of three buttons in the top-right corner of the map, the first for the filters, the second for showing or hiding sub-parcels and the third for adding a new parcel.

Right-clicking opens one of two menus, depending on the click’s location. On the empty map it is minimal, a single “Add a new parcel” entry; on a parcel it is more extensive.

It lists: “Edit” (reshape the polygon), “Manage parcel” (open the management panel, where renaming, recolouring, crop periods and sharing of the parcel are handled), “Parcel operations”, “Add sub-parcel”, “Select parent” and “Delete”. In the import-review mode it also offers “Approve parcel”.

- The “Edit” enters the reshaping mode on the polygon.
- The “Manage parcel” opens a modal containing a list of options for the selected parcel.
- The “Parcel operations” opens the list of operations belonging to that parcel

- The “Add sub-parcel” enters a drawing mode similar to the one for adding a new parcel, but with the additional constraint that the new geometry must be contained within the parent parcel.
- The “Select parent” changes the selection to the parent parcel.

Figure 5.1 shows the map in a typical state, with the parcel list open on the left and the map controls in the top-right corner.

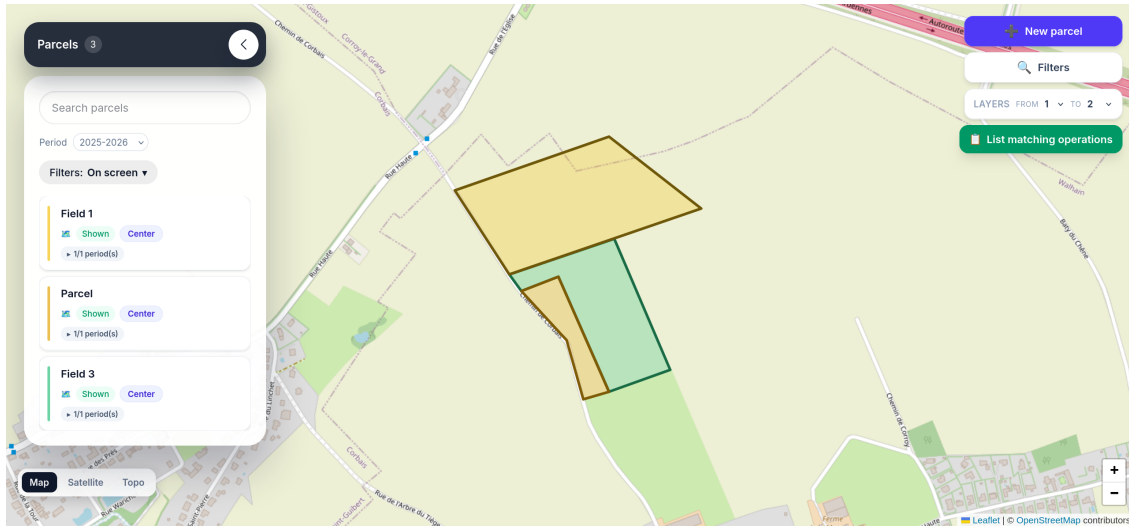


Figure 5.1: The interactive map: the parcels list on the left (with its search, period and filter), the controls in the top-right corner (new parcel, filters, layers, operation list) and the base-layer switcher at the bottom left. The parcels are drawn over the OpenStreetMap layer, here 3 distinct parcels.

Adding a parcel relies on a customised drawing layer based on the standard `leaflet-draw` polygon tool with custom hooks and helper markers. The hooks add magnetic snapping to existing edges, live overlap detection/autocorrection (see Section 6.3) against the other parcels at the same hierarchical level and, for sub-parcels, a check that the new geometry sits inside its declared parent. Once the polygon is closed, the application opens a small modal that asks for the name and the period of the parcel; the default colour of the parcel is blue and will follow the default colour of the crop cultivated on it but can be overwritten later through the right-click menu.

Editing follows the same principle but is invoked from the right-click menu on an existing parcel. The default `leaflet-draw` edit mode was modified for the same reasons as the drawing mode, in order to enforce certain rules during editing, such as shrinking sub-parcels when the reshaped parent no longer contains them, all of which must occur in real time.

5.2 Parcels and sub-parcels

A parcel can have a parent parcel. The link is stored as a self-referencing foreign key in the database, so the depth is not capped in advance. The parcel list on the left of the map reflects this hierarchy by indenting children under their parent, so the position of each parcel in the tree is immediately apparent from the list.

When a sub-parcel is created, the application checks whether it is contained within its declared parent. If the new geometry is not entirely inside the parent, the autocorrection reshapes the sub-parcel to fit within it. This is not enforced as a hard rule: a button allows the user to disable the autocorrection and draw sub-parcels that cross the borders of other parcels.

5.3 Anti-overlap detection and automatic fix

Overlap detection required the greatest engineering effort, as it is also the feature in which an incorrect implementation would most degrade usability.

While a polygon is being drawn or reshaped in the browser, the application compares it against the other parcels on the map. Running a full intersection against every parcel would be inefficient, so the check is organised as four tests of increasing cost:

- Bounding-box overlap: if the bounding boxes of the two polygons do not intersect, the parcels cannot overlap.
- Ray-casting test for vertex inclusion: determines whether a vertex of either polygon lies inside the other.
- Edge-crossing test: checks the edges for a proper crossing.
- Exact Martinez intersection: computes an exact intersection for the thin slivers the earlier tests do not detect.

As soon as one test reports an overlap, the remaining tests are skipped. This keeps the common case inexpensive while preserving exactness where it is required.

When an overlap is detected, it is highlighted on the map by default. The proposed correction is computed by subtracting the existing polygons from the newly drawn one using the Martinez–Rueda boolean operation provided by `martinez-polygon-clipping`. The result may comprise one or several polygons; the largest by signed area is retained, which in almost all of our tests corresponded to the intended geometry. A preview allows the user to toggle between the original and the

corrected polygon before accepting it. If the difference returns no usable result (for example, when the new polygon lies entirely within an existing one), the polygon is kept exactly as drawn, leaving the overlap visible on the map; this was judged preferable to deforming the geometry into a shape the user did not intend.

On the server side, the backend accepts the geometry as received and persists it immediately. PostGIS is used by the application for history retrieval (`ST_Intersects` on the polygon-based search and `ST_MakeEnvelope` on the viewport reload, both implemented within `ParcelRepository`) but not for save-time overlap blocking. Validation is handled in the browser simply as a user aid: if a user intentionally wishes to bypass the frontend constraints to submit an overlapping or malformed geometry, the server has no reason to prevent it, which aligns with the soft-boundaries model discussed.

5.4 Crop periods and the agronomic record

A parcel on its own only says *where*; what is grown there changes from one season to the next and the application captures that through the notion of a *period*. A period is a named time window defined at the farm level (typically a campaign or a season) and a `ParcelPeriod` attaches a parcel to one of those windows together with the crop that occupies it. This is where the agronomic data of the plot lives: the crop (kept both as a free label and as an official PAC/SIGEC culture code), the variety, the declared and measured areas in hectares, the sowing density, the row spacing, the sowing date, the harvest date, the target yield and the realised yield. The parcels list (Appendix A) exposes these agronomic columns directly.

Since a parcel has one `ParcelPeriod` per period, the ordered list of its periods constitutes the crop history of the plot. An operation, in turn, is attached to its parcel (or to several at once) through the many-to-many link described in Chapter 6. It also records the crop period to which it belongs, set explicitly by the user or resolved from the operation’s date. A treatment therefore remains associated with the crop actually present on the plot at the time. Together, these per-period records already capture most of the information that a paper *carnet de culture* would contain.

5.5 Operation logging

An “operation” is the central historical event in EFMS. It has a date, a duration, a type drawn from a configurable list (typically sowing, fertilisation, treatment,

harvest or observation, though each farm can extend the list) and a list of products that were applied. Each product line carries a quantity, a unit and the tool that was used. Tools and products are scoped to the farm, so two farms can have a product called “urea” without sharing the same row.

The logging modal can be accessed either via the contextual right-click menu of a specific parcel or through the dedicated `/operations` route. The system supports batch operations, allowing a single entry to be attached to multiple parcels simultaneously, reflecting field conditions where a single treatment is applied across multiple parcels. Furthermore, the interface allows for “retro-operations”, i.e., operations that are recorded after the fact, enabling farm staff to record operational data asynchronously after the work has been completed.

A significant point raised during the discussions concerns the link between a tool and a product. The default tool of an operation type can be configured, but the user can override it per product line. This directly addresses the request for a “definition of the tool depending on the operation” collected during the discussions: a fertilisation operation typically involves a spreader, for instance, but when logging a manual fertilisation the user can set the tool to “hand” or an equivalent.

An attachment can be associated with an operation, a product or a tool: a photograph of the field after a treatment, a scanned delivery note, a product data sheet. Each retains its original filename, MIME type and size, and is served by the backend through a farm-scoped endpoint. These files are consulted in the product and tool *sheets*, the modals that display a single item alongside its picture and description.

5.6 The official phyto-product catalogue

Phyto products are regulated and the legal status of a product (its authorisation number, the decision that authorises it, the dates between which it may be used, the user group it is reserved to) is public data published by the Belgian federal administration. EFMS mirrors that catalogue. An administrator triggers a synchronisation that connects over SFTP to the federal open-data server (`ftp.health.belgium.be`), downloads the catalogue files and upserts one *official* product per entry, keeping the regulatory metadata: the authorisation number, the decision code, the first-authorisation and validity dates, the user-group and formulation-type codes and the product-type classification. Each run is written to a sync log (files processed, rows created, updated, skipped), so the operation is auditable and can be repeated whenever the official data changes. The run is

incremental, processing only the files it has not already imported. When an authorisation is superseded, the previous entry is kept as a past version rather than overwritten, so the catalogue carries its own history.

Official products reside in the same `products` table as the farm's own products; they simply carry an "official" flag and belong to no farm. From the interface, a user can browse and filter the catalogue by decision code, user group or product type, and when managing the farm's products or logging an operation can select an official entry, including its authorisation number and validity dates, rather than re-entering the data manually. No constraint is enforced at entry time: the catalogue serves as a reference, not as a gate. Nevertheless, maintaining a current, searchable copy of the federal data is essential, as it underpins the phytosanitary traceability to which the farm is subject. Figure 5.2 shows the catalogue being browsed.

ASSETS

Products & Tools

Manage the consumables and equipment available on the selected farm.

[Refresh farms](#)

Products

Tools

Periods

Cultures

Reference data

[Official catalog](#)

OFFICIAL CATALOG

Official PHYTO products

Read-only list of products imported from the PHYTO registry.

Search by name or auth number

All decisions All user groups All product types

Official: 1,4SIGHT (10576P/B)
Version: 2026_05_01_FULL_4 · Decision: AKK · Validity: 11-11-2025 - 30-11-2028 · User group: PRO · Formulation: HN/KN · Product types: PESTICIDE_GR
[Official](#)

Official: 1,4SIGHT (1358P/P)
Version: 2026_05_01_FULL_2 · Decision: INT · Validity: 22-04-2024 - 31-10-2025 · User group: PRO · Formulation: HN/KN · Product types: PESTICIDE_GR
[Official](#)

Official: 1,4SIGHT (1359P/P)
Version: 2026_05_01_FULL_2 · Decision: INT · Validity: 22-04-2024 - 31-10-2025 · User group: PRO · Formulation: HN/KN · Product types: PESTICIDE_GR
[Official](#)

Official: 1,4SMART (1410P/P)
Version: 2026_05_01_FULL_4 · Decision: AKK · Validity: 15-12-2025 - 30-11-2028 · User group: PRO · Formulation: HN/KN · Product types: PESTICIDE_GR
[Official](#)

Official: AAKO CHLORTOLURON 500 SC (9549P/B)
Version: 2026_05_01_FULL_2 · Decision: AKK · Validity: 08-03-2024 - 15-08-2027 · User group: PRO · Formulation: SC · Product types: PESTICIDE_HE
[Official](#)

Previous Page 1 of 200 Next

Figure 5.2: The official phyto-product catalogue. Products can be filtered by decision code, user group or product type; each carries its authorisation number and validity dates and can be picked directly when managing products or logging an operation.

5.7 Filtering and history retrieval

Once the operations are stored in the database, they must be retrievable. EFMS provides three ways to query the history.

The first one is a per-parcel timeline: right-clicking on a parcel then “Parcel operations” opens a popup that lists, in reverse chronological order, every operation linked to that parcel, with its date, type and products. The second is a global filter at the farm level: the filter button in the top right of the map allows the user to combine a period (an interval defined by the farm), a tool, a product and a date range. The third is a polygon-based search: the user draws an arbitrary area on the map and the backend returns every parcel whose geometry intersects that area, optionally restricted by the same attribute filters. The polygon-based search is implemented as a native PostGIS query that combines `ST_GeomFromText` on the WKT received from the client and `ST_Intersects` against the parcel geometries.

The history view is also paged by the current map extent: as the user pans or zooms, the application fetches only the parcels within the current bounding box. This approach was adopted after the initial render was found to be slow when the entire farm was loaded at once, particularly in zones dense with sub-parcels. Each reload is a debounced `ST_MakeEnvelope` request. Figure 5.3 shows the operation-history view with its filters.

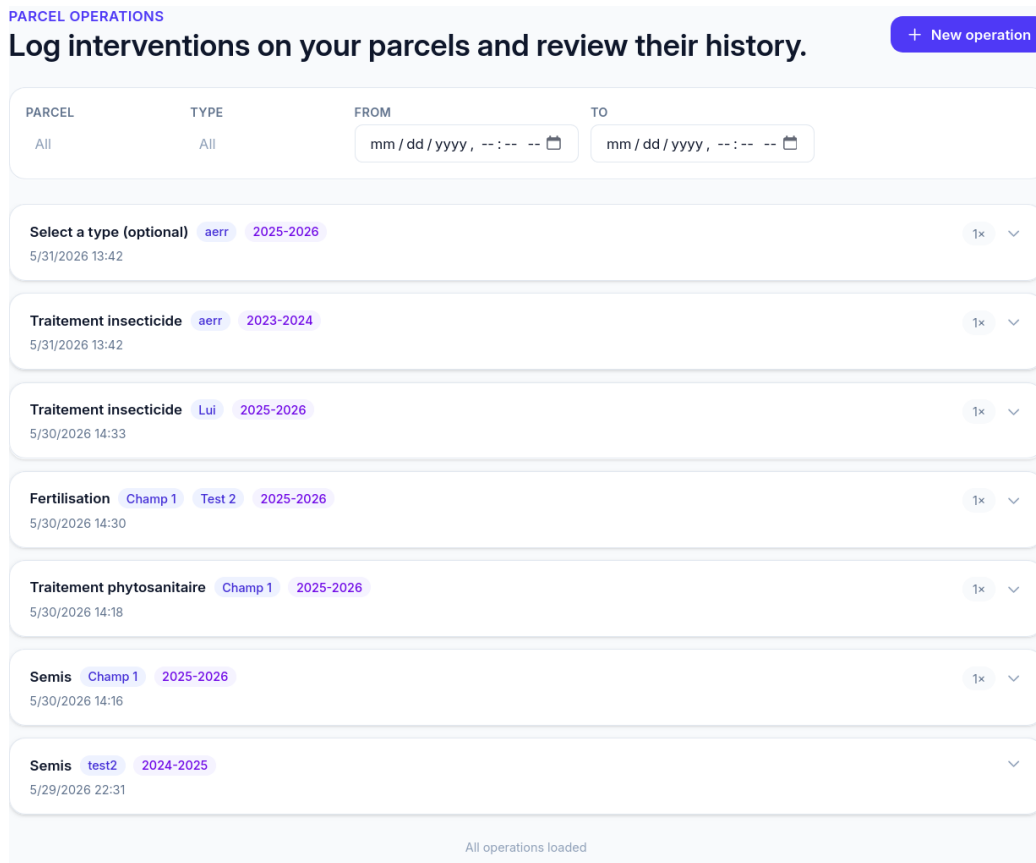
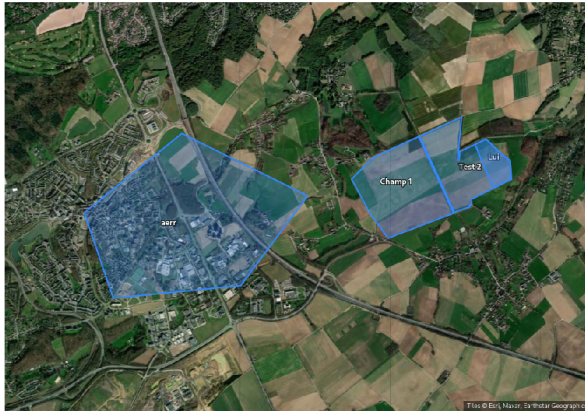


Figure 5.3: The operation-history view: every logged operation in reverse chronological order, each with its type, the parcels it touched and its crop period, filtered here by parcel, type and a date range.

A filtered history can also be exported from the application. The filter panel provides an *Export operations* control with two options, Excel and PDF, both of which act on the operations selected by the current filters. The Excel file, produced with `exceljs`, is a formatted spreadsheet with one row per operation: the date, the operation type, the parcels involved, the crop period, the duration and the products. The PDF, produced with `jspdf` and `jspdf-autotable`, presents the same rows as a landscape table, preceded by the active filter summary, the generation date and a satellite snapshot of the on-screen parcels, rendered by the browser onto a canvas from the map tiles. This PDF approximates the printed *carnet de culture* maintained by the staff and constitutes a first step towards it. Figure 5.4 shows a generated report.

Rapport d'opérations

Généré le 31/05/2026 13:43
Périodes : 2



Date	Type	Parcelles	Période	Durée	Produits
31/05/2026 13:42		aerr	2025-2026		Patates — 50 graines/m²
31/05/2026 13:42	Traitement insecticide	aerr	2023-2024		Insecticide 3000 — 30 L (Tracteur)
30/05/2026 14:33	Traitement insecticide	Lui	2025-2026		Insecticide 3000 — 150 L
30/05/2026 14:30	Fertilisation	Champ 1, Test 2	2025-2026		T — 30 L
30/05/2026 14:18	Traitement phytosanitaire	Champ 1	2025-2026		T — 50 L (Tracteur)
30/05/2026 14:16	Semis	Champ 1	2025-2026		Patates — 5 graines/m²

Figure 5.4: A generated PDF export of the filtered operation history: the filter summary and generation date at the top, a satellite snapshot of the parcels in view, and the operations table (date, type, parcels, period, duration, products) below.

5.8 PAC import

The PAC declaration is the official starting point of the agricultural year. EFMS accepts it as a zipped bundle and supports the two shapes in which the farm hands it over: a plain shapefile [2] (`.shp`, `.shx`, `.dbf`, `.prj`), which is the format the regional administration produces, and a Geofolia export, where multiple fields are exported as one `Field.Json` (and an optional `Action.Json`) produced by the farm-management software the staff already use. The upload route extracts the archive into a temporary directory and, for each field it finds, picks the matching reader: GeoTools' `FileDataStoreFinder` for a shapefile, a JSON parser for a Geofolia export. In both cases the geometry is reprojected to WGS84. A shapefile carries its own coordinate system in the `.prj` file, while a Geofolia export is always in Lambert 72, the Belgian projection.

The importer extracts more than the geometry. From the file's attributes it also reads a stable parcel identifier (`GUID_PARC`), the culture name and its code, and the pointer linking a sub-plot to its parent. When the Geofolia export includes an `Action.Json`, the past operations it describes are imported as well.

The imported rows do not become live parcels immediately. Rather than using

a separate table, an imported parcel is an ordinary `parcels` row carrying a `STAGED` status; the user reviews the import and either promotes a row to a live parcel (or assigns it to an existing one) or rejects it, through the imports management screen (Figure 5.5). This too was a request from the farm: the PAC declaration sometimes contains artefacts (parcels that have changed name, parcels that no longer exist), so the staff required the ability to review the list before committing it to the working data. The stable identifier is what allows a re-import to recognise a previously seen parcel rather than duplicating it.

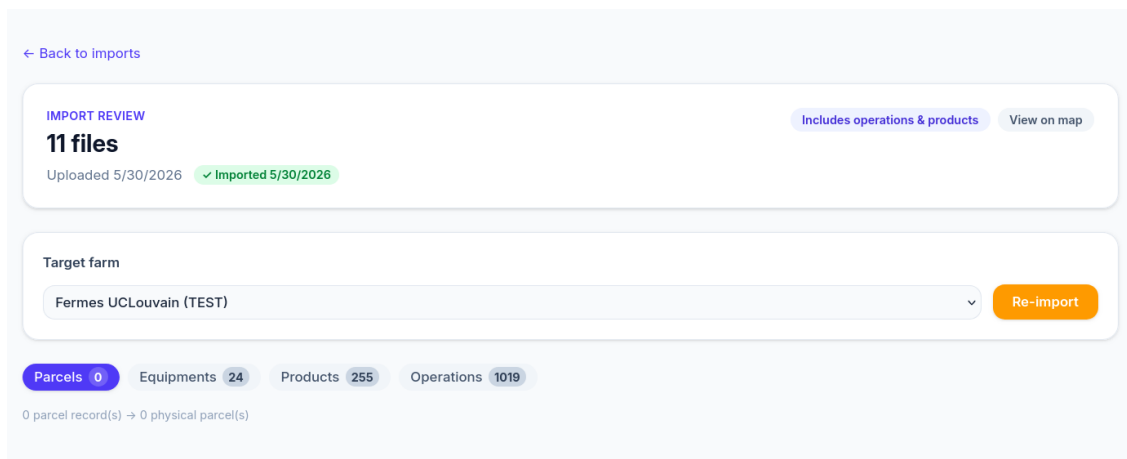


Figure 5.5: The import review screen. An uploaded bundle is broken down into parcels, equipment, products and operations; each staged parcel can be approved, either promoted to a new live parcel or merged into an existing one, or rejected, before any of it enters the working data.

5.9 Sharing and access control

Access control in EFMS works on three levels. The first is the farm itself, through the `FarmUser` table: every farm has one `Owner` and, alongside them, any number of `Admin`, `Editor` or `Viewer` members. An owner has full rights. An administrator has all rights except deletion of the farm, an editor may modify parcels and operations but nothing further, and a viewer has read-only access. At every level, the authorisation decision is made in a single location, the `PermissionService`.

The second one, at the parcel level, is the `ParcelShare` table. A farm manager can share a specific parcel with a specific user, with either a `VIEWER` or an `EDITOR` role and a toggle (`includeChildren`, on by default) that extends the share to every sub-parcel below the chosen one. The recipient sees that parcel (and, if the toggle is enabled, its sub-parcels) on their map, despite not being a member of the farm. The motivation for this feature is the researcher use case: a doctoral student requires

only their three plots rather than the entire farm. A per-parcel share provides exactly that. Figure 5.6 shows the share dialog as seen by the farm manager.

Farm shares
Manage user access and share links for the selected farm.

MY ENROLLED SHARES
Shares you can currently use.
You are not enrolled in any research share for this farm.

PARCEL SHARES

Share Name	User	Includes sub-parcels	Role	Action
test2	Admin	☒	Viewer	Delete

RESEARCH SHARE LINKS

Link-based share [Edit] [Copy link] [Delete]

Token: f0e52e63948c4569...

Users with access: Yonyc

Periods: 2025-2026

Operation types: Any

Tools: Any

Products: Any

Filter window: Any - Any

Users: unlimited

Window: 5/30/2026 02:19 → no end

Created: 5/30/2026 14:19

Link-based share [Edit] [Copy link] [Delete]

Token: 31a68105fbde4f64...

Users with access: Yonyc

Periods: Any

Operation types: Any

Tools: Any

Products: Any

Filter window: Any - Any

Users: unlimited

Window: 5/30/2026 17:13 → no end

Created: 5/30/2026 17:13

Figure 5.6: The farm shares screen. Parcel shares grant a single user a **VIEWER** or **EDITOR** role on one parcel (optionally its sub-parcels); research-share links are listed below with their token, scope and validity window. Each can be edited or revoked at any time.

The third one, added later, is the **ResearchZoneShare** table. Instead of picking a parcel from a list, the farm manager draws an arbitrary polygon zone on the map, attaches the usual filters (period, tool, product, operation type and a date range) and the application generates a unique 64-character share token. The link can be sent to a researcher; opening it requires authentication, after which the recipient claims the share with their named account (a row in **ResearchZoneShareClaim** records who claimed it and when). This is the successor to the original “public URL” concept: the link form simplifies distribution, while the claim step preserves the audit trail and the ability to revoke access per user. This feature was discussed at length: the initial proposal was an unauthenticated URL, but it was concluded that anonymous access could be neither revoked nor audited; the token-claim model is the resulting compromise. Figure 5.7 shows the creation of such a share.

Figure 5.7: Creating a research-zone share. The manager chooses between sharing directly with named users or generating a limited link, caps the number of claimants, attaches the filters that scope the access (period, operation type, tool, product and a date window) and sets the validity window before the tokenised link is created.

5.10 Reference data, public farms and administration

Several catalogues underlie the operational data. Crops are identified by the official Belgian PAC/SIGEC culture codes, each a code with a French and a Dutch label (Figure 5.8). A separate culture-type classification, matched to a crop by its code, adds a category and a colour, the colour being what the map uses to fill a parcel by default. Culture types, units, product types and operation types all follow a shared scope pattern: a row is either *global*, visible to every farm, or scoped to a single farm; an administrator can *promote* a useful farm-local entry into the global set so that other deployments benefit from it (tools are themselves farm-scoped, while their categories are a small global list). This allows each farm to name its own inventory while common entries still converge. The other management screens behind these catalogues (products, tools, periods, units and the others) are collected in Appendix A.

ASSETS

Products & Tools

Manage the consumables and equipment available on the selected farm.

Refresh farms

Products

Tools

Periods

Cultures

Reference data

Official catalog

BELGIAN PAC / SIGEC CROP CODES

Culture Types

Reference crop codes used in PAC declarations. Global codes follow the Belgian SIGEC reference list; farm-specific codes extend it for your farm.

Global reference (89) Farm-specific (1) + Add culture type

All Cereals Ecology Forestry Fruits Grasslands Industrial Legumes Other Potatoes Vegetables

CODE	NAME (FR)	NAME (NL)	CATEGORY	COLOUR
CEREALS				
100	Froment d'hiver	Wintertarwe	Cereals	—
101	Froment de printemps	Lentetarwe	Cereals	—
102	Épeautre	Spelt	Cereals	—
103	Froment dur (blé durum)	Durumtarwe	Cereals	—
110	Seigle d'hiver	Winterrogge	Cereals	—
111	Seigle de printemps	Lenteogge	Cereals	—
112	Triticale	Triticale	Cereals	—
120	Orge d'hiver	Wintergerst	Cereals	—
121	Orge de printemps	Lentegerst	Cereals	—
122	Escourgeon	Wintergerst (6-rijen)	Cereals	—
130	Avoine d'été	Zomerhaver	Cereals	—
131	Avoine d'hiver	Winterhaver	Cereals	—
140	Maïs grain	Korrelmaïs	Cereals	—
141	Maïs ensilage	Silomais	Cereals	—
142	Maïs doux	Suikermâis	Cereals	—

Previous Page 1 of 6 Next

Figure 5.8: The culture-types catalogue: the official Belgian PAC/SIGEC crop codes, each with its French and Dutch label, a category and the colour the map uses to fill a parcel by default.

A farm may also make itself visible to non-members. Enabling its *public* flag adds it to a listing that anyone can browse. Separate switches control how much is revealed: its name, description and location, in any combination. The parcels and operations remain private regardless. The intended use case is a research site that seeks to be discoverable without exposing its data.

Finally, a system administrator (a role above any single farm) has a dedicated console for user management, system settings, the editable templates of the transactional emails sent by the application, and per-farm actions to approve or delete a farm. A registration-time email verification can be enabled from these settings; a farm-approval flag exists alongside it, although the gate that would keep an unapproved farm inactive is not yet implemented. These tools are what allow the

deployment to be operated independently after the thesis, rather than remaining dependent on the original authors. The per-farm settings, including the public-listing toggles and the email-alert switches, are shown in Appendix A.

5.11 User profile, tutorial, translations and documentation

The user-facing infrastructure comprises three additional components. The profile page allows the user to upload an avatar (using a built-in cropping tool) and to edit a set of preferences: preferred interface language, time format (12- or 24-hour), date format, default farm opened on login, opt-in email notifications and the position of the operations popup on the map. The preferences are stored on the user's row in the database and loaded when the application starts.

When an administrator enables the corresponding option in the system settings, registration requires an additional step. The backend sends a one-time link to the address provided at registration. The new account cannot log in until that link is followed. The message is routed through a self-configured SMTP relay (`spring-boot-starter-mail`, configured to use `mail.werya.be` in our deployment), so that no third-party email provider is involved.

An interactive tutorial runs the first time a user opens the map. The tutorial state (`NOT_STARTED`, `IN_PROGRESS`, `COMPLETED`) is stored on the server, so that the tour is not displayed on every session, which had been a minor but recurring issue during the early validation rounds. A dedicated `/wiki` page complements the tour with a written reference: keyboard shortcuts, a step-by-step walkthrough of the main workflows (map, parcel operations, sharing, settings) and a button that resets the tutorial state to allow the tour to be replayed. The wiki content is also localised through dedicated translation files (`src/locales/<lang>/wiki.json`). The entire interface uses `i18next`; French and English are supported by default. The user profile and its preferences are shown in Appendix A.

Chapter 6

Design

This chapter presents the design of the system. It describes how the data is organised, the spatial algorithms on which the application relies and the reasoning behind each choice. It is deliberately separated from Chapter 7, which describes *how* the design is realised technically: the objective here is to identify the choices that an alternative implementation, in a different language or on a different stack, would still have to address.

6.1 Design overview

At a high level, the architecture of EFMS is a standard three-tier web application: a stateless backend serving a REST API, a relational database with a spatial extension, and a browser application communicating with the API. The data model is centred on three primary entities (the parcel, the operation and the user), complemented by secondary entities (farm, product, tool, period, share). Figure 6.1 provides an overview; the remainder of the chapter expands on each component.

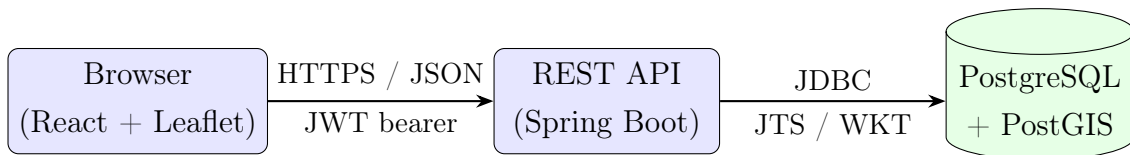


Figure 6.1: High-level architecture of EFMS. The browser holds the interactive map, the Spring Boot backend exposes a REST API and the PostgreSQL database with PostGIS stores both attribute and spatial data.

6.2 Database design

The full schema is in `EFMS.sql` in the backend repository. It runs to around twenty-five JPA entities: most feed the seed data (parcels, operations, products and their supporting tables), while a few (the share tables, import records, email templates) are bootstrapped on first start-up by Hibernate’s schema validation step. Rather than presenting every table in a single diagram, the schema is shown as a set of

focused relationship figures, each isolating one group of tables, which are discussed in the subsections below. In each figure, an arrow points from the table holding the foreign key to the table it references, labelled with the key column; the 1 and N marks indicate the cardinality at each end.

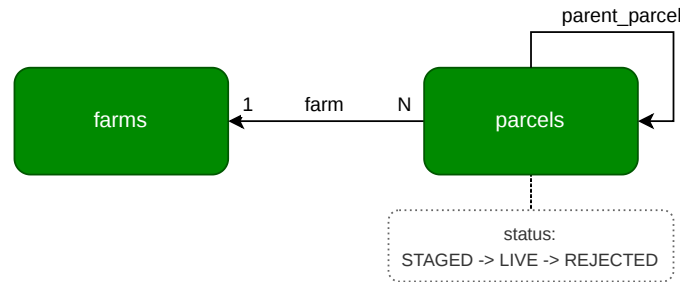


Figure 6.2: Each `parcels` row is scoped to one `farm` and points to its parent through the self-referencing `parent_parcel` foreign key (the loop): a `NULL` parent marks a top-level parcel, any other value a sub-parcel. A `status` column drives the import lifecycle `STAGED` $\rightarrow$ `LIVE` $\rightarrow$ `REJECTED`, replacing the former separate staging table.

6.2.1 Spatial entities

The `parcels` table is the main spatial component of the schema. Its `geodata` column is a PostGIS `geometry` stored in SRID 4326 (WGS84). The choice of WGS84 over a projected CRS was deliberate: every map tile used is in WGS84, every imported shapefile is reprojected to WGS84, and the user-facing coordinates exchanged with the browser are expressed as latitude/longitude. Storing the geometry in a projected CRS would have benefited area-based computations but would have required a reprojection on every read, which is not an acceptable trade-off for a real-time interactive application.

The hierarchy of parcels is encoded in the `parent_parcel` column, a self-referencing foreign key to `parcels(id)`. A `NULL` parent means a top-level parcel; any other value means a child. Traversal of the hierarchy, where required, is performed in the application code, fetching the children one level at a time through the `parent_parcel` link rather than with a single recursive SQL query. The more elaborate *materialised path* and *nested set* encodings [22] were considered but not adopted: the trees are shallow (typically 2 to 3 levels), the workload is predominantly read-oriented with infrequent moves, and the simplicity of an adjacency list offers clear advantages in code clarity. For the few cases in which the descendants of a parent parcel are required, the level-by-level traversal is sufficient.

What a parcel grows is not stored on the parcel itself but on the `parcel_periods`

association entity, which resolves the many-to-many link between `parcels` and `periods`: a parcel exists across several periods and a period groups many parcels. Each `parcel_periods` row is the agronomic snapshot of one parcel for one period (Figure 6.3).

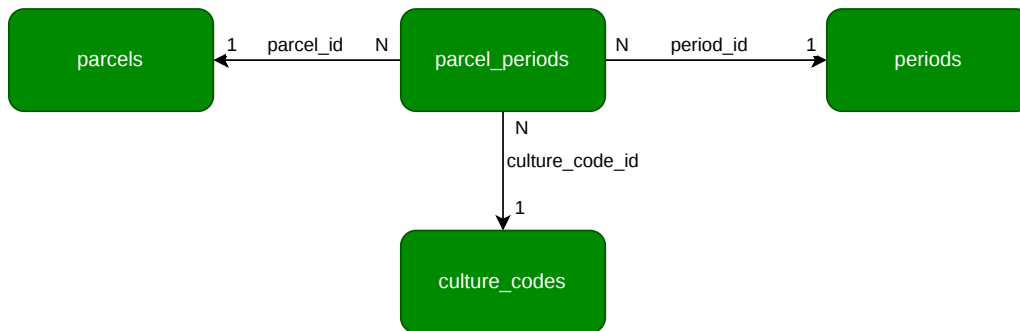


Figure 6.3: The many-to-many link between `parcels` and `periods` is resolved by the `parcel_periods` association entity: a parcel exists across several periods and a period groups many parcels, so neither side can hold a single foreign key. Each `parcel_periods` row is the agronomic snapshot of one parcel for one period (culture, variety, declared and measured area, yields, sowing and harvest dates) and names the declared crop through `culture_code_id`.

There is no separate staging table for imports (Figure 6.4). An upload becomes a single `import` record, which expands into one or more `import_source_files`; the rows produced by the pipeline are written directly into the working tables: `parcels`, `tools`, `products`, `parcel_operations`. Each such row carries its `import_id` and `source_file_id`, so that its origin is preserved, and is created with the `STAGED` status. From there, the same `status` column described earlier governs the transition, moving the parcel to `LIVE` on approval or to `REJECTED` otherwise. Keeping all rows in the working tables in this way preserves the full provenance of each row and avoids a parallel set of staging tables that would otherwise duplicate the schema.

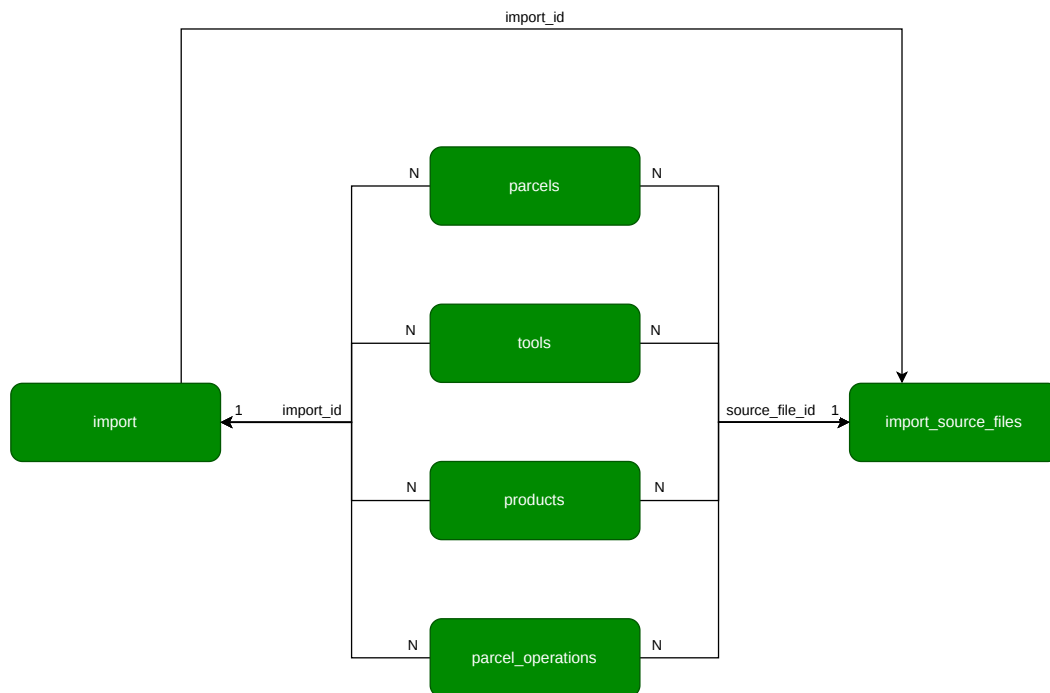


Figure 6.4: Provenance of imported rows. An upload is a single `import` record that fans out into one or more `import_source_files`. Every row the pipeline writes into a working table (`parcels`, `tools`, `products`, `parcel_operations`) keeps an `import_id` pointing back to the upload and a `source_file_id` pointing back to the file it came from, so a staged row can always be traced to its origin without a parallel staging schema.

6.2.2 Operations and products

`parcel_operations` stores the events. Each row carries the `date` on which the operation took place (a full timestamp, which allows the farm to log a precise time where necessary and a date-only granularity otherwise), an optional `duration_seconds` for the time spent on the operation, a type and the usual `created_at`, `modified_at` and `deleted_at` audit columns (the last enables soft deletion, so an operation that is removed from the UI can still be recovered from the database). `operation_types` stores the categories (sowing, harvest, treatment, etc.) and a default tool for each category. Because an operation can target several parcels at once, the link between operation and parcel is a many-to-many table, `parcels_parcel_operations`. Each operation can also reference several products, through `operation_products`; a product line carries a product, a quantity, a unit and a tool. Products and tools are scoped to a farm, which avoids cross-pollination between deployments and matches the way the farm staff named their inventory in the meetings. Figure 6.5

shows how these tables connect.

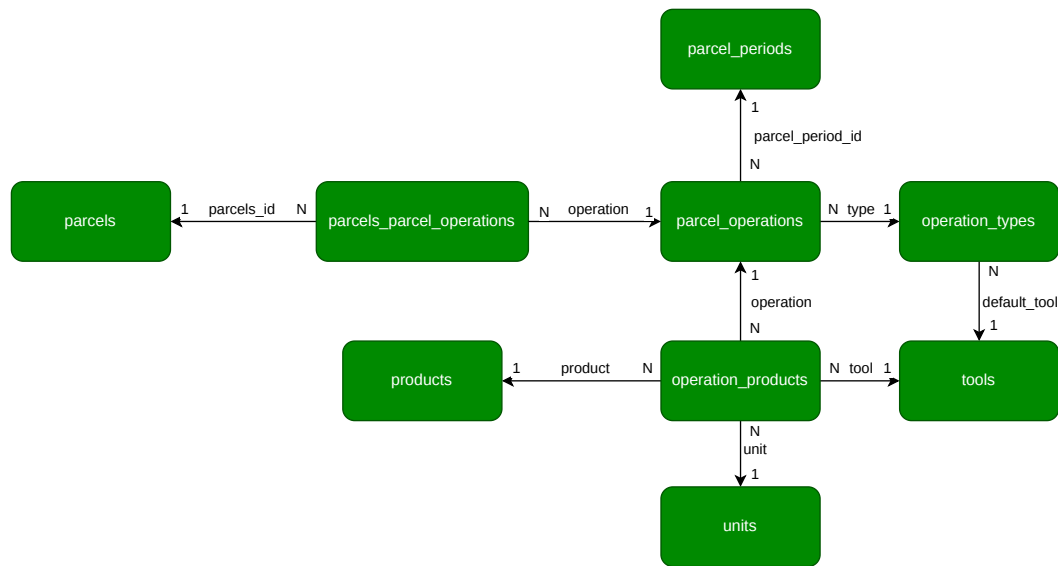


Figure 6.5: An operation can target several parcels at once, so `parcels` and `parcel_operations` are joined by the many-to-many table `parcels_parcel_operations`. Each operation has a `type` (`operation_types`, which carries a `default_tool`) and may be attributed to a `parcel_periods` row. The products applied during an operation are recorded as `operation_products` lines, each naming a `product`, a `unit` and a `tool`.

6.2.3 Users, farms and shares

A `User` owns zero or more farms (one-to-many through `farms.owner_id`) and is a member of zero or more farms through `farm_users`. The `farm_users` row carries a `Role` value: `ADMIN`, `EDITOR`, or `VIEWER`. The owner has all rights by definition and is not stored in `farm_users`; this keeps the role enum smaller and avoids a special case in the SQL. Per-parcel sharing is handled by `parcel_shares`, a separate table that gives a `ParcelShareRole` (`VIEWER` or `EDITOR`) on a single parcel to a single user, with an `include_children` boolean (on by default) that propagates the share down the sub-parcel tree; it is independent of the farm membership. A third sharing mechanism, `research_zone_shares`, complements the per-parcel one for the researcher use case: instead of pointing at an existing parcel, it stores an arbitrary polygon zone (WKT) together with the attribute filters that scope the access (period, tool, product, operation type and a date range) and a unique 64-character token; the recipient claims the share through a separate `research_zone_share_claims` row, which preserves the audit trail. Figure 6.6 brings the identity and access tables together.

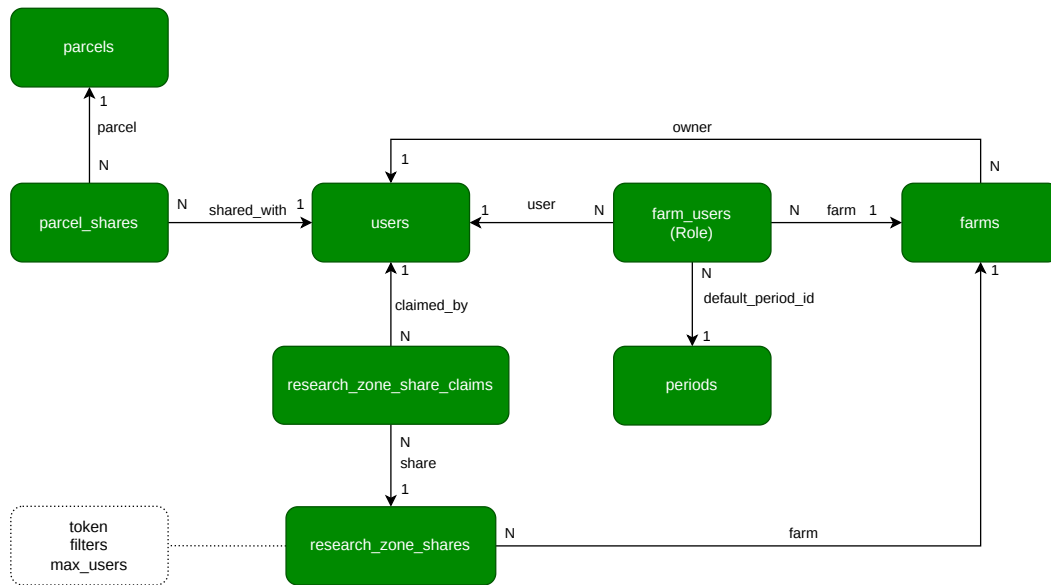


Figure 6.6: A `user` owns farms directly (the `owner` link) and joins others through `farm_users`, the association entity that carries the `Role` and the per-farm `default_period_id`. Beyond membership, `parcel_shares` grants a single user a role on one parcel (optionally its sub-parcels), while `research_zone_shares` grant a tokenised, filter-scoped zone that an external recipient records through a `research_zone_share_claims` audit row.

6.2.4 Why not a graph database, or a document store?

Both options were considered but not retained, for the following reasons. A graph store (e.g. Neo4j) would have made the parent-child relationship between parcels more natural to traverse, but the remainder of the schema is heavily relational (operations, products, units, accounts); PostgreSQL with PostGIS is the de facto standard for spatial data. A document store would have provided greater flexibility in operation logging, but at the cost of the constraints on which the application relies (foreign keys, transactional integrity). The combination of PostgreSQL and PostGIS is therefore the most appropriate choice for what is essentially a spatial relational application.

6.3 Algorithmic choices

The algorithmic content of EFMS is concentrated in the spatial-integrity layer. This section describes the two main components: overlap detection and the overlap-fix computation through a polygon difference.

6.3.1 Overlap detection

Let $\mathcal{P} = \{P_1, \dots, P_n\}$ denote the parcels at the same level as the polygon Q being drawn or edited. The direct approach is to test $P_i \cap Q \neq \emptyset$ with a full polygon-polygon intersection for every P_i ; this is, however, too slow to be performed on every cursor movement, particularly on a field already subdivided into many pieces. The single expensive test is therefore replaced by a cascade of four, ordered from least to most precise, so that the early tests eliminate most candidates. The first test to detect an overlap ends the cascade.

1. **Bounding-box pre-filter.** The axis-aligned boxes $\text{BBox}(P_i)$ and $\text{BBox}(Q)$ are intersected. If they are disjoint, $P_i \cap Q = \emptyset$ is guaranteed, and P_i is discarded immediately.
2. **Point-in-polygon test.** For the parcels that pass the box test, a classical ray-casting check (`isPointInPolygon`) determines whether any vertex of Q lies inside P_i , and conversely. A single vertex lying inside the other polygon is sufficient to declare an overlap.
3. **Edge-crossing test.** Two polygons can still cross without any vertex of either lying inside the other. To detect this case, the edges of Q are tested against the edges of P_i for a proper segment intersection.
4. **Boolean intersection.** The final test computes the exact Martinez–Rueda intersection of Q and P_i . It is counted as an overlap only when the result has a strictly positive area, above a small tolerance that absorbs floating-point noise, so that two parcels merely touching along an edge or a corner are not flagged. This test detects the thin slivers that the earlier tests do not.

The first two tests are fast but not sufficient on their own; the last two ensure exactness. Because of this ordering, the common case remains inexpensive: a disjoint parcel is eliminated at the bounding-box stage, an overlapping one is detected by the vertex test, and only the ambiguous cases reach the costlier edge and intersection tests. The result of the cascade feeds the auto-correction step. When auto-correct is enabled, which is the default, a detected overlap triggers the polygon difference described below; the user is then presented with the corrected geometry. When it is disabled, the correction is skipped and the polygon is stored exactly as drawn, including overlaps, on the principle that the user may have information the application does not. The server does not repeat this verification at save time: it

accepts the geometry as submitted, a deliberately permissive approach that leaves the user free to perform unusual operations when required.

The activity diagram in Figure 6.7 summarises the full overlap-detection / auto-correct flow.

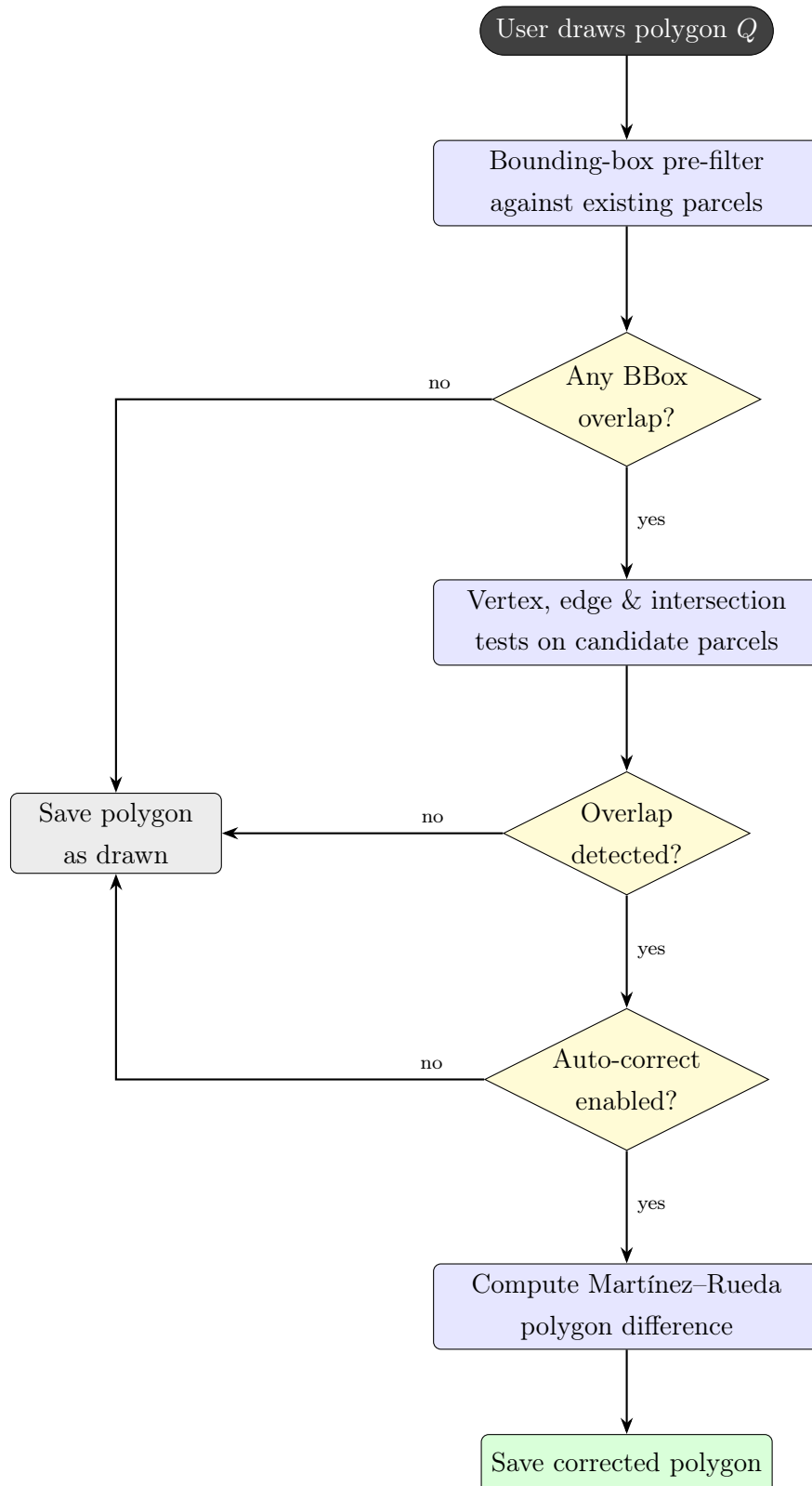


Figure 6.7: Activity diagram of the overlap-detection and auto-correct flow on the client side. When auto-correct is enabled (the default), a detected overlap triggers the polygon-difference computation and saves the corrected geometry. When disabled, the polygon is saved as drawn, preserving the user's intent. The server accepts geometries as submitted.

6.3.2 Overlap fix through polygon difference

When an overlap is detected and the auto-correct option is left enabled, the geometry is repaired as follows. Let Q be the new polygon and $\mathcal{C} = \{C_1, \dots, C_m\}$ the set of conflicting parcels. The fix computes

$$Q' = Q \setminus \bigcup_{i=1}^m C_i.$$

This is performed in two steps. First, the conflicting parcels $C_1, \dots, C_m$ are unioned into a single clip geometry; second, that union is subtracted from the working polygon Q in a single difference operation. Both operations rely on the Martinez–Rueda algorithm for boolean operations on polygons [15], provided by the `martinez-polygon-clipping` JavaScript library. Performing a single union-then-difference rather than m successive differences avoids the keyhole-bridge artefacts that iterative subtraction can produce when two intervening parcels share an edge. On degenerate input, the union step may itself fail. The algorithm then falls back to subtracting the obstacles individually. Each pass takes the working polygons accumulated so far, together with the next obstacle C_i , and replaces every working polygon by its difference with C_i . A single difference may yield several disconnected pieces (Martinez may return a multi-polygon), so the resulting pieces are flattened back into the list and processing continues.

At the end of the difference, several pieces may remain. The application discards the degenerate ones (fewer than three vertices, vanishing area, or no vertex shared with the original Q) and retains the piece of largest area, on the assumption that it corresponds to the part of Q intended by the user. During validation, this assumption held in every observed case; the edge case in which the user intends the smaller piece can be addressed by drawing the polygon differently.

When the difference returns no usable result (the polygon lay entirely within an existing parcel), the application retains the polygon as drawn. The overlap then remains visible on the map, on the principle that preserving the user’s original polygon is preferable to algorithmically distorting the input into a shape that was not drawn.

6.3.3 Why Martinez–Rueda and not Vatti or Sutherland–Hodgman

The choice of the boolean-operation algorithm warrants a brief justification. Three classical families exist: the early Sutherland–Hodgman algorithm, which clips a subject polygon against a convex clipper; Vatti’s algorithm [16], which generalises Sutherland–Hodgman to arbitrary polygons and remains widely used; and the more recent Martinez–Rueda algorithm [15]. Sutherland–Hodgman is the simplest but does not handle concave clippers, which immediately excludes it for the present use case, since an actual PAC parcel can be highly concave. Vatti’s algorithm handles arbitrary polygons but is more difficult to implement correctly in the presence of self-intersecting edges. The Martinez–Rueda algorithm, also based on the plane-sweep approach summarised in [17], benefits from a well-tested and well-maintained JavaScript implementation (`martinez-polygon-clipping`), which is the reason it was selected. An alternative, `polygon-clipping` (a Greiner–Hormann implementation in JavaScript), was also briefly considered, but the Martinez library produced cleaner output on the degenerate cases encountered.

6.3.4 Spatial filtering on the server

The history retrieval queries are written as native SQL in `ParcelRepository.searchParcels`. The filter combines, when present, a period, a tool, a product, a date range, an arbitrary polygon (passed as WKT [23]) and a viewport bounding box. The polygon and the bounding box use `ST_Intersects` and `ST_MakeEnvelope` respectively, both anchored to SRID 4326. Native SQL was chosen over JPQL because the JPA criteria API does not expose PostGIS functions cleanly, and because the query is read-intensive, deterministic and clearer in plain SQL.

6.4 Concluding remarks on the design

The design presented above is deliberately conservative. It introduces no new spatial algorithm, no new method for encoding hierarchies in SQL, and no new permission model. Instead, for each problem, a well-established textbook solution suited to the constraints of an experimental farm was selected; these solutions were then combined into a coherent whole. The contribution of the work, insofar as there is one, lies in this combination and in the fit between the design and the user’s actual workflow, a point developed further in Chapter 11.

Chapter 7

Implementation

This chapter documents how the design of the previous chapter is realised in code. It covers the technology stack, the system architecture, the security model, the deployment pipeline and the conventions adopted throughout the codebase. The objective is to provide a future maintainer with sufficient information to navigate the project without reading every file.

7.1 Technology stack

EFMS is split into two code repositories. The backend, `EFMS-backend`¹, is a Spring Boot [24] 3.5.7 application written in Java 17. The frontend, `EFMS-frontend`², is a server-rendered React application built with React 19.1, React Router 7.7, Vite 6.3 and Tailwind 4. Both parts are containerised; a `docker-compose.yml` at the root of the backend starts a `postgis/postgis:15-3.3` container alongside the application image. A third repository, `EFMS-deploy`³, packages the deployment for an operator: the container-orchestration files and an installer script that deploys the entire stack on a fresh server.

The dependencies worth mentioning are the following. On the backend: `spring-boot-starter-data-jpa` for the persistence layer, `spring-boot-starter-security` [25] together with `jjwt` 0.11.5 for authentication (the JWT specification is given in [26]), `hibernate-spatial` and `jts-core` [27] 1.19 for the geometry types (the column type is JTS Geometry, mapped through Hibernate Spatial), `geotools/gt-shapefile` 29.2 with `gt-epsg-hsql` for the PAC import, `spring-boot-starter-mail` for the transactional emails (account verification and notifications) and `spring-boot-starter-actuator` for the health endpoints used during deployment tests. On the frontend: `react-leaflet` [28] 5.0 together with `leaflet-draw` 1.0 and `react-leaflet-draw` 0.20 for the map and the drawing toolbar, `martinez-polygon-clipping` [15] 0.7 for the polygon difference used in the overlap-fix, `react-easy-crop` for the avatar cropping in

¹<https://github.com/W-EFMS/EFMS-backend>

²<https://github.com/W-EFMS/EFMS-frontend>

³<https://github.com/W-EFMS/EFMS-deploy>

the profile page, `exceljs` 4.4 and `jspdf` 4.2 with `jspdf-autotable` 5.0 for the Excel and PDF export of the operation history, and `i18next` 25 together with `react-i18next` for the translations. Table 7.1 summarises the main components.

Table 7.1: Main technology choices in EFMS.

Layer	Technology	Rationale
Backend language	Java 17 (LTS)	Familiar, mature, long-term support
Backend framework	Spring Boot 3.5.7	REST API, dependency injection, conventions
Persistence	JPA + Hibernate Spatial	Maps JTS geometry to PostGIS geometry
Database	PostgreSQL 15 + PostGIS 3.3	Standard SQL + spatial extension
Authentication	JWT (HS256) + BCrypt [29]	Stateless, easy to scale
Shapefile import	GeoTools [3] 29.2	PAC declaration is a zipped shapefile
Frontend framework	React 19.1 + React Router 7.7	Mature React stack, file-based routing
Build tool	Vite 6.3	Fast HMR, simple config
Styling	Tailwind 4	Utility-first, predictable output
Map library	Leaflet [28] + react-leaflet	Open source, no API key needed
Polygon ops	martinez-polygon-clipping [15]	Boolean operations on the client
i18n	i18next	Standard, JSON-based locale files
Containerisation	Docker + docker-compose	Reproducible deployment

7.2 System architecture

The application follows a decoupled architecture with a clear boundary between the frontend and the backend. The backend exposes a REST API under a few stable prefixes (`/auth`, `/users`, `/farm`, `/parcels`, `/imports`, `/units`, `/admin`), with farm-scoped resources such as products, tools and operations located under `/farm/{farmId}`. Every endpoint accepts and returns JSON. The geometry is encoded as WKT (Well-Known Text) [23] in both requests and responses, which the backend parses with the `JTS WKTRewriter`. Authentication relies on JWT (HMAC-SHA-256, 24-hour validity by default), the frontend attaching the token as an `Authorization: Bearer ...` header.

Beyond the standard arguments, this separation proved advantageous during development. Throughout the year, the backend was run on a fixed port and the frontend on the Vite development server, so that the React side reloaded on every change without restarting the JVM. The CORS configuration currently accepts any origin (`allowedOriginPatterns` set to `*`), which suited the split-port development setup in which the frontend and backend run on different ports; restricting it to the deployment's own origin is a straightforward hardening step that has not yet been applied. The production profile (`application-prod.properties`) points the

application at the production database URL.

7.2.1 Frontend structure

Routing on the frontend is file-based, following the React Router convention: a route is a file under `app/routes/`, while a thin wrapper handles the locale prefix in the path. The map component (`MapWithPolygons.client.tsx`) is loaded dynamically because `react-leaflet` is not server-renderable; the remainder of the application is server-rendered through React Router's framework mode. The state shared between routes (the authenticated user, the selected farm) is held in two React contexts (`AuthContext`, `FarmContext`); all other state is local to its component.

Translations are stored in flat key-value files under `locales/`, one per language and per namespace (a general `translation.json` and a separate `wiki.json`). The keys are grouped by feature area, `map.draw.toolbar` or `operations.modal.title` for instance, so that a new feature introduces its own keys without affecting the existing strings. Adding a language requires no code change: a JSON file containing the same keys, translated, is added and registered in the `i18next` configuration.

7.2.2 Backend structure

The backend follows a standard Spring Boot n-tier architecture, separating responsibilities across REST controllers, business services, JPA repositories, and persistence entities. The main exception to this standard pattern is the `PermissionService`, a single component invoked by every mutating service method before it modifies data. It is not an interceptor: the call is explicit and is made at each call site (a little over two hundred across the services). The benefit is therefore not automatic enforcement but centralisation: the rules governing who may view, edit, manage or share are defined once, in a single place, so that a new endpoint reuses the same checks rather than re-implementing them; a change to a rule is then made in a single file.

`ParcelRepository` contains two native spatial queries: `searchParcels` for the filtered polygon search, and a second for the viewport reload. Both are written in plain SQL, because the JPA criteria API does not expose the PostGIS functions cleanly. Each spatial predicate anchors the geometry to SRID 4326 directly in the SQL, using `ST_SetSRID` on the stored column. The import process is handled by `ImportService`: it extracts the uploaded archive into a temporary directory and, for each bundle, either reads a shapefile with `GeoTools` (`FileDataStoreFinder`, reprojecting every feature to WGS84) or parses a Geo-

folia `Field.Json/Action.Json` pair. A field, whether obtained from a shapefile or a `Field.Json`, becomes a `parcels` row marked `STAGED` rather than a separate entity; an accompanying `Action.Json` stages the past operations in the same way. The user then decides, row by row, which to promote to live records and which to reject.

7.3 Security model

The security model is built in three layers. The first one, the JWT layer, authenticates the request: the `JwtAuthenticationFilter` reads the bearer token, checks the signature with `JwtUtil` and fills the Spring `SecurityContext` with the username. The second one, the permission layer, authorises the request: `PermissionService` centralises the rules (who can view, edit, manage, share) and is called from every service method that mutates state. The third, the database layer, enforces structural integrity: foreign keys ensure that an operation cannot reference a non-existent parcel, and the `@Transactional` annotations ensure that an aborted save does not leave partially written rows.

JWT was chosen partly for familiarity, but it offers a concrete benefit: the session state is carried in the token, so the backend retains none of its own. Scaling and load-balancing are consequently straightforward. Passwords are hashed with BCrypt [29] at the default cost factor of 10. The signing secret is read from an environment variable in production and falls back to a placeholder in development; a secret shorter than the 32 bytes required by HS256 is first stretched to the required length using SHA-256.

7.4 Deployment

The deployment artefact is a docker-compose stack: one container for PostgreSQL+PostGIS, one for the backend. The frontend is built into static assets and served, in production, by the React Router server (`react-router-serve`). Using nginx as a reverse proxy was considered but proved impractical with the SSR mode; the Spring Boot host has sufficient capacity to serve the assets in any case. The database is exposed on 5432 for inspection during development and is placed behind the docker-compose network in production; the backend runs on 8080.

The data directory is a named docker volume (`pgdata`). Backups are produced by running `pg_dump` on the volume from the host (the standard Postgres procedure; the command is documented in the deployment notes), with the output written to

a host directory; the request for a more readable export (a ZIP of CSVs) and for a backup button in the UI are both on the roadmap. The deployment procedure for a fresh server is: clone the two repositories, fill in the `.env` file with the database credentials, the JWT secret and the SMTP credentials and run `docker compose up -d`. The boot sequence takes about a minute on the target hardware.

7.5 Code conventions

A few small conventions, set at the start of the year, are worth mentioning because they shape the readability of the code.

One reason per commit. Each commit on the main branch addresses one concern, with a message that explains the reason (“why”) rather than the contents (“what”). This made the validation meetings easier, as it was always possible to present what changed between the previous demo and the current one.

No dead code, no commented-out code. When a feature is removed, its code is removed with it rather than left behind a comment. The Git history retains the previous state, so the working tree remains concise. Although this may appear self-evident, over a year and with two contributors it is the practice that prevents the accumulation of deprecated code in the repository.

Chapter 8

Validation and User Feedback

The validation strategy of EFMS combines two complementary approaches: a series of iterative, qualitative meetings with the farm staff throughout the year and a structured usability evaluation at the end of the development based on the System Usability Scale (SUS). The qualitative side shaped the application as it was being built; the SUS evaluation gives a quantitative read on how it stands today. This chapter covers both, in that order.

8.1 Iterative cadence

Development was organised into sprints of roughly two to four weeks, each concluded by a meeting presenting the latest state of the application. The meetings were short demonstrations of the new changes with the supervisor followed by a discussion about the next sprint goals. This open format proved the most productive. The comments rarely followed the anticipated agenda; the requests raised spontaneously were almost always the most valuable.

The full timeline of the project, with what was completed in each sprint, is documented in the `agile_roadmap.md` file in the project's notes. Sprint 0 (September-October 2025) established the project structure, the first map with polygons and the database design. Sprint 1 (mid-November 2025) introduced authentication, registration and the basic API. Sprint 2 (January 2026) added the farm entity, user profiles, tools, operations endpoints and the PAC shapefile import. Sprint 3 (February 2026) covered sub-parcels, history filtering and the parcel display selectors. Sprint 4, in March-April 2026, addressed the permission model and the polygon editing layer through a full code refactor. Sprint 5, in April-May 2026, delivered the more advanced features: multi-parcel operations, research-zone sharing with token-based access, filters on the operation history, email notifications and account verification and the administration interface.

8.2 Validation rounds and what they changed

Several meetings shaped the application in non-trivial ways; the most influential are listed below, together with the change each triggered.

15 December 2025. This round produced the first usability feedback. Consultations with the farm staff yielded several insights regarding user requirements: the necessity to remove obsolete tools and products from the interface without removing them from the database (they are still used for historical purposes), the question of colour-blind-accessible default palettes and the implementation of an explicit map zoom affordance. The first requirement led to the introduction of an **active** boolean flag on tools and products; the second led to a redesign of the colour-picker within the parcel modal; the third was resolved by restoring the native Leaflet zoom controls, which had been hidden in a previous design iteration.

2 February 2026. This meeting led to the the addition of user tests and scaling tests along with two product changes: filters on the operation history (treatment, culture, time) with the default view restricted to the current year and a polygon-based search on the map. Both were implemented in the following sprint and are now available in the search panel.

16 March 2026. This meeting concerned minor refinements: keyboard shortcuts for parcels creation, clear visual differencier for parcel levels, default colour per culture and a refresh button for the phyto database in place of an automatic re-import.

13 April 2026. This was a long session, responsible for most of the recent UI work, but not only: logging an operation across several parcels at once (associated with the period), default colours for parcel levels and culture types. Two further requests were taken on: combo-boxes within the filter inputs and an expiry date on the share tokens. The first has been implemented right away; the second was implemented later.

27 April 2026. This was a late meeting, concerned primarily with the report structure and clarifications. On the technical side, the discussion led to the request for an admin interface covering farm validation and the account-creation rules, potentially with email alerts.

8.3 Lessons learned

Three patterns recurred frequently during the validation rounds and are worth stating as general lessons, separately from the meeting-by-meeting log. They are presented, following the structure adopted in comparable applied theses, as common mistakes to avoid and recommended practices.

8.3.1 Common mistakes to avoid

Encoding constraints as hard blocks rather than warnings. The first version of the anti-overlap logic was strict: any overlap prevented the save entirely. The farm objected firmly at the next demonstration: drawing complex adjacent plots is an iterative process, so a hard block forces the user to abandon the drawing in progress. Constraints encoded as warnings, with a one-click correction, are considerably more usable than constraints encoded as blocks. The same lesson applied to the sub-parcel containment rule: a child polygon was initially forbidden from extending beyond its parent, but since agronomists sometimes deliberately draw such overflows (a buffer strip, a control zone), the strict rule had to be relaxed.

Designing access control without the operator. The first design of the share mechanism was an unauthenticated, view-only URL that the farm could send to a researcher. It would have been quick to implement and easy to use, but it was clearly rejected once presented: anonymous access cannot be revoked or audited and constitutes a long-lived security vulnerability. Rewriting the share into named accounts with per-parcel roles required a substantial amount of work that could have been avoided had the question been raised earlier. Security and access-control decisions must be made together with the operator of the system, not on their behalf.

Triggering interactive overlays on every session. The first version of the in-app tutorial started on every login, which the staff found intrusive. The tutorial state was therefore stored on the user (`NOT_STARTED`, `IN_PROGRESS`, `COMPLETED`); the tour was triggered only when the value was not `COMPLETED`. The change required a few hours of work but resolved the complaint immediately. Any pop-up or guided tour should be one-time by default and configurable in the user profile.

8.3.2 Recommended practices

Demonstrate rather than ask. The most useful feedback consistently came from a demonstration rather than from a question. After the second meeting, the question “would you like feature X?” was replaced by “here is feature X, what would prevent you from using it in practice?”. The answers were more precise, more concrete and more often unexpected.

Trace every feature to a request. Every feature in the application can be traced to a meeting note. This simplified design discussions, since no feature was a personal preference to be defended. It also facilitated writing the report, because the justification for each feature was already recorded in the notes.

8.4 System Usability Scale evaluation

Alongside the qualitative feedback loops, a structured usability evaluation was carried out at the end of the development using the System Usability Scale (SUS). SUS is a standardised, ten-question Likert questionnaire originally designed by Brooke [30] that has become one of the most used instruments in human-computer interaction; the meaning of a given score has been studied empirically [31, 32] and is now a well-understood reference.

8.4.1 Methodology

Participants. The questionnaire was completed by a panel of four at the end of the development phase: Simon, the member of the farm staff who tested the application; Benjamin, the agronomist who follows the operations; and two students with an agricultural background (Matthieu and Kyara). Two of the four (Simon and Benjamin) are potential daily users of the applications, the other two providing an outside view from the same domain. The panel remains small, so the resulting score is treated as an indicative figure rather than a firm quantitative claim.

Tasks. Before the questionnaire, each participant first completed a short set of tasks covering the most frequently used features:

1. log in and pick the LLN farm;
2. create a new sub-parcel inside an existing parent and accept the overlap-fix proposal;

3. log an operation on that sub-parcel;
4. retrieve the operation history of that sub-parcel using the per-parcel timeline;
5. retrieve the same operation using the polygon-based search over an area drawn on the map;
6. share the sub-parcel with a colleague's account in viewer mode;
7. find and toggle the language between French and English.

Evaluation instrument. The standard System Usability Scale (SUS) was used without modification for accurate results. Each participant rates ten statements on a five-point Likert scale, from “Strongly Disagree” (1) to “Strongly Agree” (5). The wording alternates in polarity, the odd-numbered statements are positive and the even-numbered ones are negative to eliminate uniform responses. The score is computed by the standard procedure: for an odd-numbered item, 1 is subtracted from the response for an even-numbered item, the response is subtracted from 5; the ten adjusted values are then summed and multiplied by 2.5, yielding a single value between 0 and 100.

Table 8.1: The ten statements of the System Usability Scale [30] used in this evaluation.

#	Statement
1	I think that I would like to use this system frequently.
2	I found the system unnecessarily complex.
3	I thought the system was easy to use.
4	I think that I would need the support of a technical person to be able to use this system.
5	I found the various functions in this system were well integrated.
6	I thought there was too much inconsistency in this system.
7	I would imagine that most people would learn to use this system very quickly.
8	I found the system very cumbersome to use.
9	I felt very confident using the system.
10	I needed to learn a lot of things before I could get going with this system.

8.4.2 Results

The aggregated SUS score over the four participants is reported in Table 8.2; the per-statement breakdown is shown in Figure 8.1. The mean SUS score is **75.0**, which sits above the commonly cited average of 68 [31] and corresponds to a “*good*” rating on the adjective scale. The individual scores span 65.0 to 90.0, the highest coming from the member of the farm staff who tested the application and the lowest from the agronomy student. The two participants from the farm (Simon and Benjamin) bracket the mean, at 90.0 and 67.5 respectively.

Table 8.2: Aggregated SUS scores per participant.

Participant	Role	SUS score
Simon Vanderveken	Farm staff	90.0
Benjamin	Agronomist	67.5
Matthieu	Student (farm background)	77.5
Kyara	Agronomy student	65.0
Mean		75.0

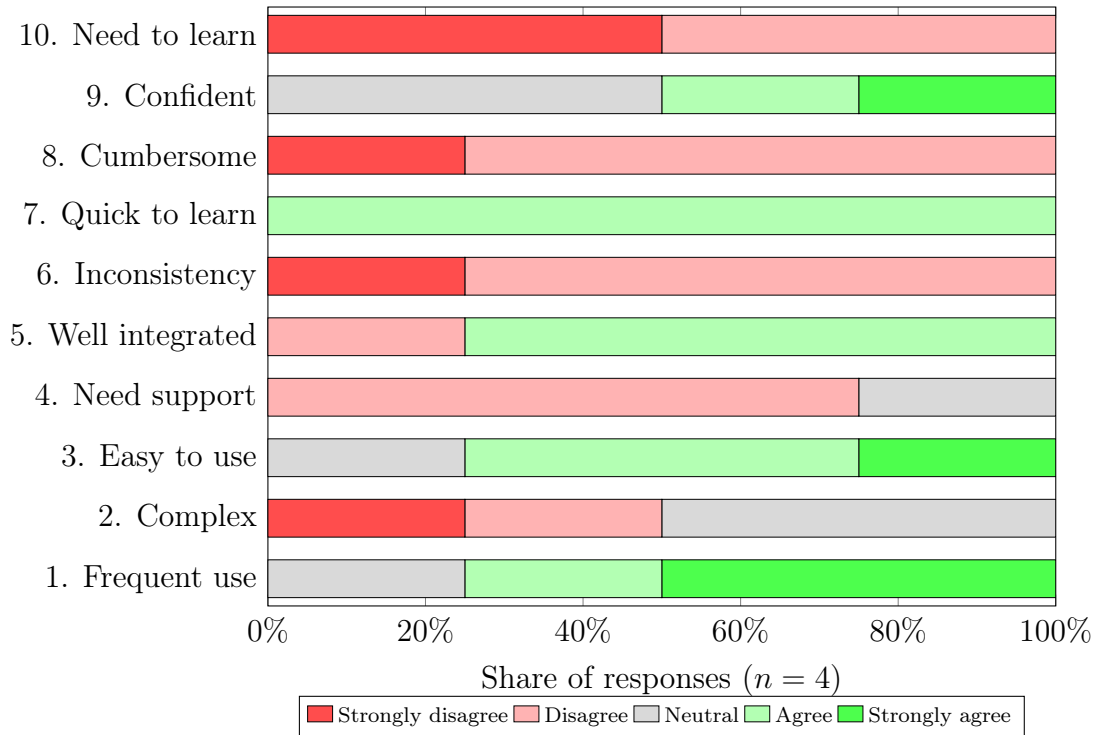


Figure 8.1: Distribution of the four participants’ raw responses to each of the ten SUS statements of Table 8.1, as a percentage of the panel. The statements alternate in polarity: for the positively worded, odd-numbered ones, agreement (green) is favourable, whereas for the negatively worded, even-numbered ones, it is disagreement (red) that is favourable. The bars therefore show raw agreement, not the polarity-adjusted SUS contribution.

8.4.3 Qualitative open-ended feedback

In addition to the standard SUS statements, each participant answered three open-ended questions: *which three aspects of the system are the most effective or successful?*, *which three need the most improvement or caused the most frustration?* and *what should be improved or added first?*. The recurring points are summarised below.

What worked.

- The speed of the application and the absence of any installation: it runs in the browser with nothing to set up beforehand.
- Drawing parcels and, in particular, dividing a parcel into sub-parcels.
- Managing the productions and following the operations of a parcel over the season.

- A simple menu with quick access, and the parcel-sharing feature.

What to change.

- Improve the mobile view, which several remarks found awkward to use.
- Reduce the number of steps required to add data to a parcel (for instance the fertiliser used), without having to go through the profile page.
- Add phytosanitary stock management and link the phyto reference catalogue to the crop.
- Provide an export of the operations per parcel and per year (since partly addressed through the Excel and PDF exports).

8.5 Current state of the validation

At the time of writing, the application is not yet in operational use on the LLN farm. A demonstration instance has been set up with representative data and the PAC declaration for the current year has been imported. Several experimental sub-parcels have been drawn and a small number of operations logged, but the farm does not yet rely on it for its day-to-day record-keeping.

The validation therefore rests on the demonstration-driven meetings documented above and on the SUS evaluation, complemented by the scaling and behavioural load tests reported in Chapter 9. An in-situ trial on the farm with multiple concurrent users remains the natural continuation of the work once the system is put into service.

Chapter 9

Testing and Metrics

This chapter addresses the technical aspects of the validation: the automated tests, the performance and scaling experiments, and the behavioural tests conducted on the application. The user-acceptance aspects are covered in the previous chapter; the objective here is to document the behaviour of the code under automated execution.

9.1 Unit tests

The backend includes a JUnit 5 test suite, which has grown considerably beyond the security core from which it began. Authentication, JWT handling and the permission checks remain the most heavily covered, but the suite now extends to the controllers and to the operation, attachment and catalogue services. On the controller side it comprises `AuthControllerTest`, `ParcelControllerTest` and `ParcelOperationControllerTest`; on the security side `JwtUtilTest`, `JwtAuthenticationFilterTest` and `CustomUserDetailsServiceTest`; and on the service side `PermissionServiceTest`, `FarmServiceTest`, `FarmAssetServiceTest`, `ParcelOperationServiceTest`, `AttachmentServiceTest` and `OfficialProductServiceTest`, alongside the default `EfmsApplicationTests`, which boots the Spring context. Across these twelve classes, the suite contains close to one hundred and fifty test methods. `PermissionServiceTest` in particular was extended several times during the year as new share modes were introduced (parcel-level shares with `includeChildren`, research-zone shares with token claiming).

The most extensively tested component is the permission service, with approximately thirty-five methods exercising the role hierarchy. They verify that owners and administrators can both manage and edit, that an editor can edit but not manage, that a viewer has read-only access, and that an unauthorised user is denied. The per-parcel shares are tested separately: a `VIEWER` share should permit viewing but not modification, and an `EDITOR` share should permit editing. The tests mock the repositories with Mockito and inject the authentication through Spring Security's `TestingAuthenticationToken`. Unit tests were chosen over integration tests

here because an authorisation decision is a pure function of its inputs (the role, the share and the ownership), so that a running Spring context would contribute nothing but start-up time. Consequently, each test runs in a few milliseconds, which maintains a short feedback loop.

The JWT tests cover the round-trip: a token is generated for a username, the username is read back from the token and the token is rejected after the expected expiry. The authentication controller is tested end-to-end through `MockMvc`: a successful login returns a token, an incorrect password returns 401 and a malformed request returns 400. Two further security classes complete these: `JwtAuthenticationFilterTest` confirming that only valid bearer tokens populate the security context and `CustomUserDetailsServiceTest` for the username lookup.

The more recent classes apply the same approach to the data layer. `ParcelOperationServiceTest` is the largest: it exercises operation creation, the many-to-many link to several parcels at once and the resolution of the crop period from the operation's date. `AttachmentServiceTest` covers file handling and `OfficialProductServiceTest` the merging of the federal catalogue into the farm's products, while `ParcelControllerTest` and `ParcelOperationControllerTest` exercise the corresponding endpoints through `MockMvc` to verify their status codes and payload contracts.

9.2 Coverage

The test effort is deliberately uneven, concentrated where the tests are most valuable rather than spread evenly across the codebase. Table 9.1 provides an indicative picture per area: the security layer and the permission service are the best covered, the farm, asset, operation, attachment and catalogue services are covered to a moderate degree, and the controllers, previously checked only manually, are now exercised through `MockMvc`. The figures below are indicative orders of magnitude, derived from the distribution of the test classes rather than from a coverage tool integrated into the build.

Table 9.1: Indicative coverage of the backend test suite, by area, on the main branch at the time of writing.

Area	Coverage	Tested concerns
security	high	JWT issue/verify & expiry, auth filter, user lookup
service (permissions, sharing)	high	Role hierarchy, per-parcel shares
service (farm, assets)	moderate	Membership, asset CRUD paths
service (operations, attachments, catalogue)	moderate	Operation create, attachments, official-product merge
controller (auth)	high	Login flow, response codes
controller (parcels, operations)	moderate	Endpoint contracts, status codes

A high overall coverage figure was not a goal. Coverage measures *what was executed during the tests*, not *what was meaningfully verified*; padding the suite with redundant tests on getters and setters would have inflated the figure without improving safety. The tests were written, in order of priority, for the code that failed in production during validation and for the security layer, where a subtle defect would have wide-ranging consequences.

9.3 Deployment tests

Every push to the main branch undergoes a manual deployment test in a clean environment: a blank database, a fresh build of the backend image, and the frontend served from a clean Vite build. The local script for this is `launch.sh` at the root of the backend; it loads the `.env` file and starts the application with the active Spring profile. The full container stack is described by the `docker-compose.yml`, where the backend waits on the PostgreSQL health check (`depends_on: service_healthy`) before it starts; the deploy repository’s installer goes one step further and polls the backend and the frontend until both answer before declaring the deployment up. The backend health itself is exposed by Spring Actuator at `/actuator/health`.

The deployment test was kept in scope, even at this stage of the project, because the application is intended to be operated by someone other than the authors after the thesis ends. A deployment that requires undocumented knowledge to deploy will not survive a server replacement. As it stands, the entire procedure comprises three steps, cloning the repository, completing the `.env` file and running `docker compose up`, all of which are documented in the README, so that the next operator can follow it without contacting the authors.

9.4 Performance and load testing

Rather than benchmarking on a development laptop, the performance campaign was run on the class of hardware on which the farm is likely to deploy the application: a single-board Raspberry Pi 4. This serves two purposes. It acts as a deployment test, since it demonstrates that the entire `arm64` container stack (an Eclipse Temurin `arm64` backend image and the `aarch64` PostGIS image) builds and runs on a low-power board; and it acts as a load test, since it characterises latency, throughput and correctness under concurrent, mixed read/write traffic.

9.4.1 Setup and method

The board is a Raspberry Pi 4 Model B ($4 \times$ Cortex-A72, 3.7 GiB RAM) running Debian 13 with Docker Compose; the backend, the frontend and a PostgreSQL 15 / PostGIS 3.3 container all run co-located on the same board, with the PostgreSQL data directory on the on-board microSD card. The load generator is a Python harness running on a separate LAN-connected x86 host, so that the cost of generating traffic does not compete with the backend for the Pi's CPU; the sub-millisecond LAN round-trip (≈ 0.13 ms) means the reported latencies reflect server-side processing.

Before each run the harness seeds a fresh farm with 60 parcels (random WGS-84 polygons of 6–24 vertices), ≈ 480 operations (8 per parcel), 50 products, 15 tools and the associated reference data. It then runs 54 scenarios covering authentication, paginated and unbounded reads, the three spatial search modes (viewport, polygon, multi-filter), reference-data reads and 12 write scenarios. Each scenario is driven by 8 concurrent virtual users $\times$ 10 measured iterations (after 2 discarded warm-up iterations per user), for 4 320 measured requests per run. The campaign was run twice, once as an administrator and once as a standard, non-admin user. Ahead of the load itself, nine assertions verify that the data is not merely returned but correct, among them a geometry round-trip, the per-parcel operation history, the `parcelId/typeId` filtering and the membership of the spatial query results.

9.4.2 Results

Both runs passed all 9 correctness checks and completed with no infrastructure errors. The administrator run recorded 4 320 requests with 0 failures; the standard-user run recorded 80 failures out of 4 320 (1.9%), every one of which is the `farm_list` endpoint correctly returning HTTP 403 (listing all farms is an

administrator-only operation), so they are expected, not defects. No connection-pool exhaustion occurred: with a HikariCP pool of 30 and only 8 concurrent users, no request ever waited for a database connection.

Table 9.2 reports the latency of representative endpoints on the warm (standard-user) run; Figure 9.1 shows the read latencies visually. The figures are an upper bound rather than expected operating latency: the test drives 8 requests concurrently with no think time, which is roughly 30 to 80 times the request rate a realistic ≤ 5 -user single-farm deployment would generate.

Table 9.2: End-to-end latency of representative API calls on the Raspberry Pi 4, warm run, standard user (8 concurrent users, 60 parcels, ~ 480 operations). P95 is the latency below which 95 % of that scenario’s requests fall; RPS is the sustained throughput.

Endpoint	Avg (ms)	P95 (ms)	RPS
Single parcel by id (<code>parcel_get</code>)	14	17	42.6
Parcels, paginated	24	31	40.5
Parcels, full list (unbounded)	70	121	36.3
Viewport spatial query (around farm centre)	42	57	37.6
Polygon search (free-form WKT)	43	68	37.8
Multi-filter search (period + type + bbox + date)	31	63	38.0
Single parcel operation history	78	99	32.1
Farm-wide operations, paginated	350	517	14.8
Products merged with official catalogue	88	146	30.0
Create parcel (PostGIS write)	23	30	40.3
Create operation (with products)	37	46	38.3
Update product	21	27	41.4

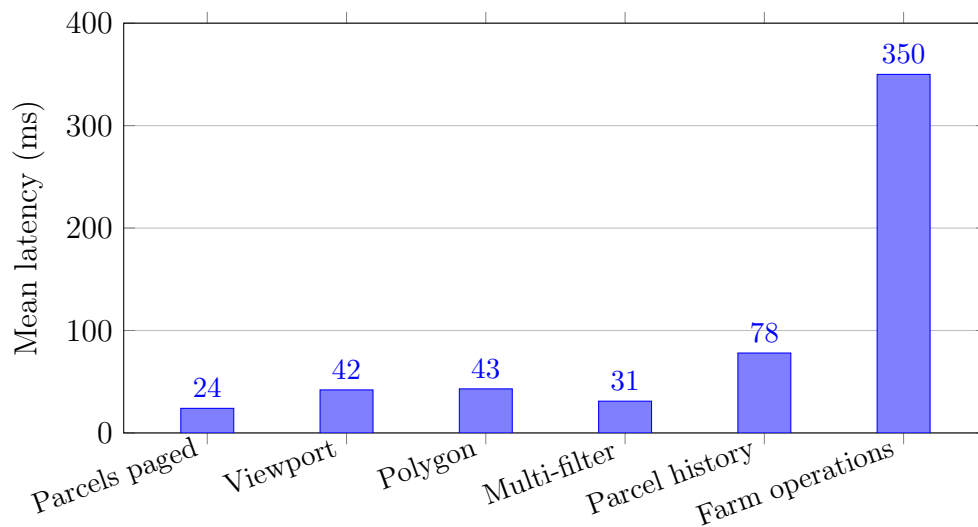


Figure 9.1: Mean read latency of representative endpoints on the Raspberry Pi 4 (warm run, 8 concurrent users). The paginated and spatial reads stay well under 100 ms; the single outlier is the farm-wide operations listing, whose cost is the pagination `COUNT(*)` plus the joins over the operations table.

9.4.3 Interpretation

Several observations follow from these figures. The writes are uniform and fast, every `create_*` and `update_*` falling between approximately 14 and 37 ms, which indicates that the create/update path is bound by the Pi’s CPU rather than by locks or a saturated connection pool. Where latency increases, it increases with the amount the database must read and join: the paginated and filtered endpoints remain inexpensive, whereas the unbounded whole-collection reads (`farm_parcels`, `farm_products`), and especially the farm-wide operations listing, are the slowest. This points to result-set size and to the `COUNT(*)` and joins, rather than object serialisation, as the dominant cost. Underlying all of this is the principal hardware constraint, the microSD card on which the database resides; the most effective single change would be to relocate the data directory onto a USB-3 SSD.

The spatial reads deserve a specific note, since they exercise the part of the system this thesis is built around: the viewport query, the free-form polygon search and the GiST index on `parcels.geodata` all behave correctly and stay under 70 ms at P95 on `arm64`. The practical conclusion is that a Raspberry Pi 4 has roughly one order of magnitude of headroom over the intended workload on ordinary endpoints, and still about 30× on the worst-case endpoint, so the latencies reported here are a stress ceiling that users under the assumed usage would essentially never observe.

The campaign also has its limits. The two runs are not a clean measurement

of authorisation cost: the administrator run was the first traffic after start-up and so paid a JVM warm-up and cold-cache penalty (constant-time endpoints are $\approx 2\text{--}3\times$ slower on it), so the per-request permission overhead would need both roles exercised on an already-warm instance to be isolated. The dataset is also moderate (60 parcels), the services are co-located on one board, and the standard user owns its farm, so the more expensive shared-parcel permission path is not exercised. A large, multi-year farm of several thousand parcels would widen the gap between the paginated and the unbounded endpoints, which is the scaling campaign noted as future work in Chapter 11.

9.5 Behavioural and concurrency tests

The backend is stateless, but the database is not. The principal concurrency scenarios were tested manually: two users editing the same parcel simultaneously, two users creating a sub-parcel of the same parent simultaneously, and a user editing a parcel that another user has just deleted. The first two scenarios currently resolve as last-write-wins: there is no version column on `parcels`, so the second save overwrites the first without raising a conflict; the third returns a 404 when the second user saves, which is the expected behaviour. The application implements no concurrency control here (neither optimistic versioning nor a row-level lock on `parcels` during edits), as the user-facing cost was judged to be lower than the complexity of a full locking layer at the scale of the LLN farm.

The frontend has a smaller set of behavioural tests, all manual at this stage: drawing a polygon and cancelling partway through must remove the temporary layer, editing an existing polygon and pressing escape must restore the original, and the “unsaved changes” guard on the operation modal must trigger when the user navigates away. None of these has yet been automated with a tool such as Playwright; this is planned as future work.

9.6 Limitations and known gaps

Several gaps remain. The frontend tests are not automated. The polygon-based history search is backed by a GiST index on `parcels.geodata`; it performed well in the Raspberry Pi load test above, but on a moderate dataset, so its behaviour on a large, multi-year farm of several thousand parcels has not yet been benchmarked. The agronomic data required by a carnet de culture is captured per crop period. The filtered operation history can now be exported (Chapter 5) to an Excel sheet

and to a PDF report that already includes a map snapshot and the operations table; what remains is a sheet formatted exactly like the document maintained by the farm staff. The CSV/ZIP backup, also requested by the farm, exists in script form (a `pg_dump` wrapper) but is not yet exposed in the user interface.

These gaps are documented as follow-up work in the conclusion of the report. In its current state, the application is, in our assessment, usable by the LLN farm for the current agricultural year, although further work is required before it can be considered complete.

Chapter 10

Ethics, Legal and Sustainability Considerations

This chapter considers the regulations and obligations that apply to the application as a whole. EFMS holds phytosanitary records regulated by Belgian law, operation data of significant value to the farm, and a limited amount of personal data concerning its users. The following sections describe, in turn, how each is handled, the topics deliberately excluded from scope, and the position of the project on intellectual property and sustainability.

10.1 Personal data and GDPR

EFMS stores a small amount of personal data per user: a username, an email address, a hashed password, an avatar image and a small number of preferences (display settings, default farm, preferred language, opt-in for email notifications). This data is processed under the legitimate interest of operating the application, in accordance with the GDPR [33]; a user may request an administrator to remove their account, in which case the row in the `users` table is deleted through the admin interface. Parcels are not owned by users but by a farm, so they remain in place after a deletion; only the audit pointers (`created_by`, `updated_by`, `deleted_by`) on past records reference a row that no longer exists. Operations and parcels are not personal data within the meaning of the GDPR, but the link between an operation and the user who logged it is, indirectly, personal: an audit log stating “user X applied product Y on date Z” identifies a person. This link is deliberately not exposed in the user interface; it remains at the database level for traceability, the application not allowing one user to query another user’s actions.

Passwords are hashed with BCrypt (cost factor 10), never stored in clear, never sent in the JSON response of any endpoint and never logged. Sessions are stateless on the server (JWT), with a default validity of 24 hours and a server-side secret read from an environment variable in production. The application does not set tracking cookies and does not load third-party analytics. The only outbound calls from the frontend are to the EFMS backend and to the tile servers behind

the map view, which is unavoidable for a Leaflet-based map; the default is the community-run, non-commercial OpenStreetMap network, with optional satellite (Esri World Imagery) and topographic (OpenTopoMap) layers the user can switch to. The backend has one outbound channel: it contacts an SMTP relay (configurable per deployment through the `MAIL_HOST` environment variable; our test setup uses `mail.werya.be`, a relay operated by the authors, while a production install would point at the operator's own) to send the transactional emails that the application produces, namely the verification email at registration time and the opt-in notifications. No mailing-list provider, analytics endpoint or third-party tracker is contacted.

10.2 Data ownership and confidentiality

The data produced by the LLN farm belongs to UCLouvain. The application is designed to be self-hosted, with no third-party SaaS involved; the demonstration instance used during the thesis runs on a server operated by the authors, while the long-term hosting can be readily provisioned by the party that deploys the production instance for the farm. This is a deliberate consequence of the farm's request for a self-hosted, open-source solution: the operator retains full control over physical access to the storage; no commercial entity can subsequently increase the cost of access to data the farm has already produced.

The two share mechanisms described in Chapter 5, the per-parcel `ParcelShare` and the per-zone `ResearchZoneShare`, are the only paths through which farm data leaves the strict farm-membership boundary. By design, every share is bound to a named user (either selected from a list in the per-parcel form, or via a token that the recipient claims with an authenticated account in the per-zone form); it can be revoked at any time, the actions performed on the shared parcel remaining traceable. An unauthenticated, view-only URL was initially considered but rejected on the grounds that anonymous access cannot be revoked or audited; the token-claim model used by the research-zone share is the compromise that preserves the convenience of a link without sacrificing the audit trail. This discussion is documented in Chapter 6 and Chapter 8.

10.3 Intellectual property

The codebase is released under the MIT licence; each repository includes its own `LICENSE` file at its root. The repositories are hosted on a public forge, with commit

history attributable to the two authors of this thesis. The dependencies are all standard open-source libraries with permissive licences (Apache 2.0, BSD, MIT, EPL); no proprietary library is bundled.

The data produced and ingested by the application (the PAC declarations, the operation history, the phytosanitary records, among others) is not covered by the open-source release; it belongs to the farm. The only data distributed with the code is seed data, which is entirely synthetic.

10.4 Phyto products and health-relevant data

Phytosanitary records have legal and health-related implications. EFMS performs no safety check on the products and doses entered by the user; it stores them as entered, with the units chosen by the user. The official Belgian database at fytowebe.be [34] remains the authoritative source for the legal status of a product. EFMS maintains a synchronised copy of the federal phyto catalogue (the admin-triggered import described in Chapter 5), which an administrator can refresh whenever the official data changes, so that the reference does not become outdated without notice. The application does not, however, validate the product and dose recorded by a user against that catalogue at entry time; the catalogue is provided as a reference, not enforced as a constraint. This report makes no claim of regulatory compliance for the phytosanitary data stored by EFMS.

10.5 Sustainability

The application is small and runs on modest hardware: the typical resident-set size of the backend after start-up is well below 512 MB, the database container fits on a single shared volume, and the static frontend assets amount to a few hundred kilobytes after gzip. Deployments follow the standard container model and can be shut down when the farm does not require them; there is no always-on cloud dependency. The application’s own components, the backend and the database, run on the operator’s own machines, with no commercial data centre involved. The only component that communicates externally is the map: a Leaflet map retrieves its tiles from a tile server rather than rendering them itself. By default, those tiles are obtained from OpenStreetMap, which is community-run and non-commercial and represents the neutral choice in the open-source geospatial domain; the user may remain on this layer exclusively. Two additional base layers communicate with other providers if selected: a satellite view from Esri World Imagery, a commercial

provider, and a topographic view from OpenTopoMap. The tile source is, however, simply a configurable URL, so none of these represents a form of lock-in.

10.6 Out of scope

A few topics are deliberately excluded from this section, as they were outside the scope of the work. EFMS does not provide an explicit consent flow at registration time; there is currently no terms-of-use checkbox and no privacy-notice acknowledgement on the registration form. A proper consent flow would be required if the application were ever opened to a wider audience outside UCLouvain. Nor does it encrypt individual sensitive fields at the application level. In production, traffic between the frontend and backend is protected by HTTPS, but data at rest in the database relies solely on the protection provided by the host's filesystem. For the current pilot, this is a reasonable boundary; for a wider deployment, these are precisely the features that would have to be implemented.

Chapter 11

Conclusion and Perspectives

This thesis began with a concrete operational problem. The Louvain-la-Neuve experimental farm had no software matching its actual mode of operation. The available tools all assumed a parcel to be a fixed legal field; none could track the small, often short-lived experimental zones that constitute the core of the farm’s research. The year was therefore spent building EFMS, an open-source web application, and evaluating it against that need. This final chapter reviews what was accomplished, where the work remained incomplete, what the year demonstrated and what merits further effort.

11.1 Summary of contributions

The contributions of this thesis fall into three categories.

Technical contributions.

- Design and implementation of an open-source farm management web application supporting recursive sub-parcels at arbitrary depth.
- A spatial layer operating in two locations: the browser detects overlapping parcels at their creation or reshaping through a cascade of checks and offers an immediate fix using the Martinez–Rueda algorithm; while the server uses PostGIS only to store and retrieve history.
- Two farm access levels: the farm level, a farm has an owner but allows for admins, editors or viewers roles, and parcel level that allows for sharing with view or edit access to specific plots.
- A reproducible Docker-based deployment with documented backup procedures, ready to be operated by the person who deploys it after the thesis.

Methodological contributions.

- A documented requirements-elicitation methodology combining open-ended interviews, on-site observation and demonstration-driven validation meetings, carried out over a year in collaboration with an experimental farm.
- Three explicit lessons learned (Section 8) about the cost of hard constraints, the design of access control and the use of interactive overlays.

Experimental contributions.

- A performance and load-test campaign run on the target class of hardware (a single-board Raspberry Pi 4 with the backend, frontend and PostGIS database co-located), exercising 54 read/write scenarios at 8 concurrent users for 4 320 requests per run with zero functional failures, with the measured latencies reported in Chapter 9.
- A structured usability evaluation based on the System Usability Scale with a small four-participant panel drawn from the people who tried the application, yielding a mean score of 75.0 (described in Section 8.4).

11.2 Objectives met and not met

Out of the five objectives stated in Chapter 1, four are met today.

Hierarchical parcels that allow unlimited sub-parcels. Implemented and tested at the depths used by the farm, typically 2 to 3. The implemented model supports greater depth without code changes but was not stress-tested beyond depth ten as no realistic use case requires it.

An accurate operation history. Implemented, with the per-parcel timeline, the multi-criteria search and the polygon-based search all available in the interface. The history remained correct under the concurrent read/write load of the Raspberry Pi campaign in Chapter 9.

Spatial integrity. Implemented in two places (browser and server). The interactive overlap-fix became one of the features explicitly identified as useful by the farm staff during the validation rounds.

Sharing that respects the boundary between farm and research. Implemented through two complementary mechanisms: the per-parcel `ParcelShare` entity (with an `includeChildren` toggle to propagate the share down the sub-parcel tree) and the per-zone `ResearchZoneShare` entity (which allows the farm manager to define an arbitrary polygon and issue a tokenised link, which the recipient claims with a named account). The original idea (anonymous, view-only URLs) was rejected during validation and replaced with named-account models; both forms in the application today preserve revocation and auditability.

Long-term usability. This is the one objective that is only partially met. The application is documented at the level of the codebase (READMEs, schema file, and comments where useful). The deployment is reproducible through Docker. The internationalisation supports French and English. However, the operational documentation intended for future maintainers (runbooks, backup/restore procedures, monitoring) is incomplete, and there was insufficient time to establish an automated CI pipeline of the desired quality. This is the principal item of follow-up work in the technical track.

Two of the user requests that were still pending at the time of the last validation round have since been implemented: the transactional email layer (account verification at registration time and opt-in notifications, both routed through a configurable SMTP relay) and the admin console (a dedicated `AdminController` on the backend, paired with a set of `/admin/*` routes on the frontend that cover user management, system settings, email templates and farm administration, including a per-farm approval flag). A first step towards the carnets de culture has also been completed: the filtered operation history can now be exported to an Excel sheet and to a PDF report (complete with a map snapshot), even if a sheet matching the farm's exact hand-kept layout remains to be produced. The CSV/ZIP backup as a UI feature remains on the roadmap; both items are documented as open work in Chapter 9 and in the perspectives section below.

11.3 Critical analysis

Measured against the academic literature, EFMS is an engineering contribution rather than a fundamental research. The two algorithmic components on which it relies (Martinez–Rueda for the polygon difference, ray-casting for point-in-polygon) are textbook techniques, and the architecture (Spring Boot REST, server-rendered React, PostgreSQL/PostGIS) is conventional. The contribution lies in their combi-

nation and in the fit between the application and a genuine, specific user need that the existing market does not address. No new spatial algorithm was invented and none is claimed to be. For the farm, however, this fit is precisely the value of the work.

The validity of the results is established in two ways. Functional validity derives from the validation rounds with the farm staff, it is documented chronologically in Chapter 8: the application performs as requested, with the changes negotiated at successive meetings. Non-functional validity derives from the load tests in Chapter 9 which confirmed that the application deploys correctly and maintains acceptable response times on low-power single-board hardware under a request load one order of magnitude above that of the intended deployment.

11.4 Limitations

The main limitations of the work as it stands are collected here. Each is also addressed in the perspectives section.

Usability study sample size. Four participants constitute a small panel, of whom two belong to the farm (the staff member who tested the system and the agronomist) and two are students from the agricultural domain. A score derived from so few participants must therefore be interpreted with caution. Establishing a firm quantitative claim would require a broader study, with at least ten participants from the agricultural domain.

Spatial scaling not evaluated at volume. The polygon-based search is backed by a GiST index on `parcels.geodata`, created at start-up, so that the planner can use a spatial index rather than scan the entire `parcels` table. It performed well in the Raspberry Pi load test (Chapter 9), but that campaign used a moderate dataset (60 parcels, ~ 480 operations per farm). The index gain on a large, multi-year farm of several thousand parcels has not been measured; conducting that scaling campaign and confirming the query plan on production-sized data is the immediate next step.

Lack of automated frontend tests. The interface is currently tested manually. A Playwright suite implementation would be the next step to automate the behavioural checks, which has not yet been developed.

Single-tenant deployment. Although the database structure supports multiple farms, the system is currently limited to a single-tenant deployment. Operational necessities for a true multi-tenant system such as data quotas, resource monitoring and dedicated isolation paths have not yet been worked out.

Map tile sourcing. The map retrieves its tiles from third-party tile servers under fair-use policies: the default option, OpenStreetMap network, and the optional OpenTopoMap layer are community-run while the satellite layer is provided by Esri World Imagery, a commercial provider whose keyless endpoint is not licensed for production use. A single, low-traffic farm like the UCL experimental farms falls within these policies, but a heavier or multi-farm deployment would exceed them. With heavier load, local tile caching or a self-hosted tile server would become necessary. The satellite layer would require a proper provider key. The default layer remains OpenStreetMap precisely because it is the most neutral and least constrained of the three.

Share-token threat model. The research-zone links sharing rely on long random tokens that requires an account to get access. The creator can set a start and end date, a number of claimants and can revoke access at any time. However, audit logging of failed claim attempts has not been implemented.

No robustness testing of inputs. None of the application's entry points, the shapefile import (where the most fragile parsing occurs), the upload endpoints, the search forms or the share-claim endpoint, is currently fuzz-tested. For instance, a malformed payload could bring issues that the current tests would not detect. Systematic robustness testing of these inputs is left as future work.

Concurrency between users. If two users edit the same parcel simultaneously, the last write will persist, no merge is performed and no version check is made. In practice, two users rarely modify the same record at the same time so this is unlikely to cause problems. A wider site would require proper concurrency control.

11.5 Lessons learned

The most important lesson, in retrospect, is the value of soft constraints over hard ones. The first design encoded several rules as hard blocks (overlap forbidden, sub-parcel required to be inside its parent, shared link required to be anonymous and

time-limited). Each of these blocks was either rejected by the farm at the next demonstration or had to be relaxed into a warning with an automatic fix. The pattern was consistent: the user possesses information the application does not, so a hard block forces the user to abandon the task in progress. A warning with a one-click fix allows the user to proceed and, in our experience, yields a more correct outcome on average. This principle would be applied from the outset in any future iteration.

The second lesson concerns the cost of an incorrect share model. The unauthenticated, view-only URL initially designed would have been quick to implement and easy to use, but it was judged unacceptable (no revocation, no audit, no clear ownership of the access). Rewriting the share into named accounts required a substantial amount of work that could have been avoided through earlier consideration of the problem.

The third lesson is more technical and more positive: PostGIS is well suited to this class of application. The combination of spatial indexes, native intersection functions and seamless integration with the remainder of the relational schema avoided the need for a custom spatial layer; the same stack would be chosen again.

11.6 Perspectives

A few directions follow naturally from the current state of the work.

Technical track. The Raspberry Pi load test (Chapter 9) is a starting point rather than a conclusion. The obvious next step is to repeat it on a large, multi-year dataset, with the database on a USB-3 SSD rather than the microSD card, examining the query plan on data of that size. The campaign already identified one hotspot worth addressing: the farm-wide operations listing incurs a pagination `COUNT(*)` and the joins over the operations table on every request, and an index together with a means of avoiding that count would be a small, well-contained improvement. Beyond this, an end-to-end suite (Playwright on the frontend, Testcontainers on the backend) would complete the test pyramid, and a CI pipeline, whether GitHub Actions or a self-hosted GitLab runner, would be a clear improvement.

Deployment track. The application is currently configured for a single farm. The data model is designed to handle multiple independent farms, but infrastructure features such as data quotas, system monitoring and user support are still missing.

Making the software available to other farms is a clear future objective, consistent with the open-source licence of the project.

11.7 Final remarks

We began this project as software engineers with limited prior operational experience in agriculture. We conclude it aware of the distance between a polished academic application and a tool that must function reliably under typical field conditions, when a sub-parcel has to be redrawn before the next operation can be logged. EFMS is our attempt to close that distance. The work is not complete, and in a sense it never will be: software of this kind remains useful only for as long as it is used. It is, however, functional and ready for the LLN farm to adopt, and a subsequent student or maintainer can continue to develop it well after this thesis has been defended.

Bibliography

- [1] European Commission, Directorate-General for Agriculture and Rural Development. *The Common Agricultural Policy: 2023–27*. Tech. rep. Publications Office of the European Union, 2023.
- [2] Environmental Systems Research Institute. *ESRI Shapefile Technical Description*. White Paper. ESRI, July 1998.
- [3] *GeoTools Open Source GIS Toolkit*. <https://geotools.org/>.
- [4] National Geospatial-Intelligence Agency. *World Geodetic System 1984 (WGS 84)*. Standardization Document. National Geospatial-Intelligence Agency, 2014.
- [5] EPSG Geodetic Parameter Dataset. *EPSG:31370, Belgian Lambert 72*. <https://epsg.io/31370>.
- [6] ISAGRI. *Geofolia: Logiciel de gestion de parcelles*. 2026. URL: <https://www.isagri.fr/geofolia/logiciel-de-gestion-parcellaire> (visited on 06/01/2026).
- [7] SMAG. *Smag Farmer: Logiciel de gestion de parcelles et déclaration PAC*. 2026. URL: <https://smag.tech/nos-logiciels/declaration-pac/> (visited on 06/01/2026).
- [8] ISAGRI. *Geofolia: Comment découper mes parcelles ou îlots ?* 2026. URL: <https://client.isagri.fr/centre-aide/geofolia/comment-decouper-mes-parcelles-ou-ilots> (visited on 06/01/2026).
- [9] AGRIVI. *AGRIVI Farm Management Software: Products*. 2026. URL: <https://www.agrivi.com/products/> (visited on 06/01/2026).
- [10] Farmbrite. *Farmbrite: Plans and Pricing*. 2026. URL: <https://www.farmbrite.com/farm-mapping> (visited on 06/01/2026).
- [11] Agworld. *Mapping farms and fields, and importing boundary files*. 2026. URL: <https://help.agworld.com/en/collections/3065758-mapping> (visited on 06/01/2026).
- [12] S. Fountas et al. “Farm management information systems: Current situation and future perspectives”. In: *Computers and Electronics in Agriculture* 115 (2015), pp. 40–50.

- [13] J. Tummers, A. Kassahun, and B. Tekinerdogan. “Reference architecture design for farm management information systems: a multi-case study approach”. In: *Precision Agriculture* 22 (2021), pp. 22–50.
- [14] PostGIS Project Steering Committee. *PostGIS Spatial and Geographic Objects for PostgreSQL*. <https://postgis.net/documentation/>.
- [15] F. Martínez, A. Rueda, and F. R. Feito. “A new algorithm for computing Boolean operations on polygons”. In: *Computers & Geosciences* 35.6 (2009), pp. 1177–1185.
- [16] B. R. Vatti. “A generic solution to polygon clipping”. In: *Communications of the ACM* 35.7 (1992), pp. 56–63.
- [17] M. I. Shamos and D. Hoey. “Geometric intersection problems”. In: *17th Annual Symposium on Foundations of Computer Science (FOCS)*. 1976.
- [18] A. Kaloxylos et al. “A cloud-based Farm Management System: architecture and implementation”. In: *Computers and Electronics in Agriculture* 100 (2014), pp. 168–179.
- [19] M. Paustian and L. Theuvsen. “Adoption of precision agriculture technologies by German crop farmers”. In: *Precision Agriculture* 18 (2017), pp. 701–716.
- [20] European and Mediterranean Plant Protection Organization. *EPPO Global Database (EPPO codes)*. <https://gd.eppo.int/>.
- [21] U. Meier, ed. *Growth Stages of Mono- and Dicotyledonous Plants: BBCH Monograph*. 2nd ed. Federal Biological Research Centre for Agriculture and Forestry, 2001.
- [22] J. Celko. *Trees and Hierarchies in SQL for Smarties*. Morgan Kaufmann, 2004.
- [23] Open Geospatial Consortium. *OpenGIS Implementation Standard for Geographic Information: Simple feature access (Well-Known Text)*. OGC document 06-103r4. Open Geospatial Consortium, 2011.
- [24] Pivotal Software. *Spring Boot Reference Documentation*. <https://docs.spring.io/spring-boot/>. Version 3.5.x. Accessed May 2026.
- [25] VMware. *Spring Security Reference*. <https://docs.spring.io/spring-security/reference/>.
- [26] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Token (JWT)*. RFC 7519. Internet Engineering Task Force (IETF), May 2015.

- [27] M. Davis. *JTS Topology Suite*. <https://locationtech.github.io/jts/>. LocationTech Foundation. Accessed May 2026.
- [28] V. Agafonkin. *Leaflet: a JavaScript library for interactive maps*. <https://leafletjs.com/>.
- [29] N. Provos and D. Mazières. “A Future-Adaptable Password Scheme”. In: *USENIX Annual Technical Conference*. BCrypt password hashing. 1999.
- [30] J. Brooke. “SUS: A “quick and dirty” usability scale”. In: *Usability Evaluation in Industry*. Ed. by P. W. Jordan et al. London: Taylor & Francis, 1996, pp. 189–194.
- [31] A. Bangor, P. Kortum, and J. Miller. “Determining what individual SUS scores mean: adding an adjective rating scale”. In: *Journal of Usability Studies* 4.3 (2009), pp. 114–123.
- [32] B. Klug. “An overview of the System Usability Scale in library website and system usability testing”. In: *Weave: Journal of Library User Experience* 1.6 (2017).
- [33] European Parliament and Council. *Regulation (EU) 2016/679 (General Data Protection Regulation)*. Official Journal of the European Union. 2016.
- [34] SPF Santé publique, Sécurité de la Chaîne alimentaire et Environnement. *Phytoweb: Produits phytopharmaceutiques autorisés en Belgique*. <https://fytoweb.be/>.

Appendix A

Application screenshots

This appendix collects the secondary management and reference screens of EFMS, referred to from Chapter 5. They are gathered here, rather than inlined, to keep the body of the report focused on the features themselves.

PARCELS

test

Search by name, code, culture...

NAME	CODE	BLOCK	PERIOD	CULTURE	DECL. AREA (HA)	MEAS. AREA (HA)	SOWING	HARVEST	MUNICIPALITY	STATUS	ACTIVE	ACTIONS
prés pirois	1	—	2025-2026	—	—	—	—	—	—	—	Active	Manage periods
Champ 1	—	—		Orge d'hiver	—	—	—	—	—	—	Active	Manage periods
Test 2	—	—	2025-2026	—	—	—	—	—	—	—	Active	Manage periods
Test 3	—	—	2023-2024	—	—	—	—	—	—	—	Active	Manage periods
Test 4	—	—	2024-2025	510	—	—	—	—	—	—	Active	Manage periods
Lui	—	—	2025-2026	—	—	—	—	—	—	—	Active	Manage periods
aerr	—	—		Orge d'hiver	—	—	—	—	—	—	Active	Manage periods
aaa	—	—	2025-2026	—	—	—	—	—	—	—	Active	Manage periods
atat	—	—	2025-2026	—	—	—	—	—	—	—	Active	Manage periods
aeazeazeazeaze	—	—	2025-2026	—	—	—	—	—	—	—	Active	Manage periods
ae	—	—	2025-2026	—	—	—	—	—	—	—	Active	Manage periods

11 of 11 parcel(s)

Figure A.1: The parcels list, with the agronomic columns shown directly: code, block, period, culture, declared and measured area, sowing and harvest dates, municipality and status.

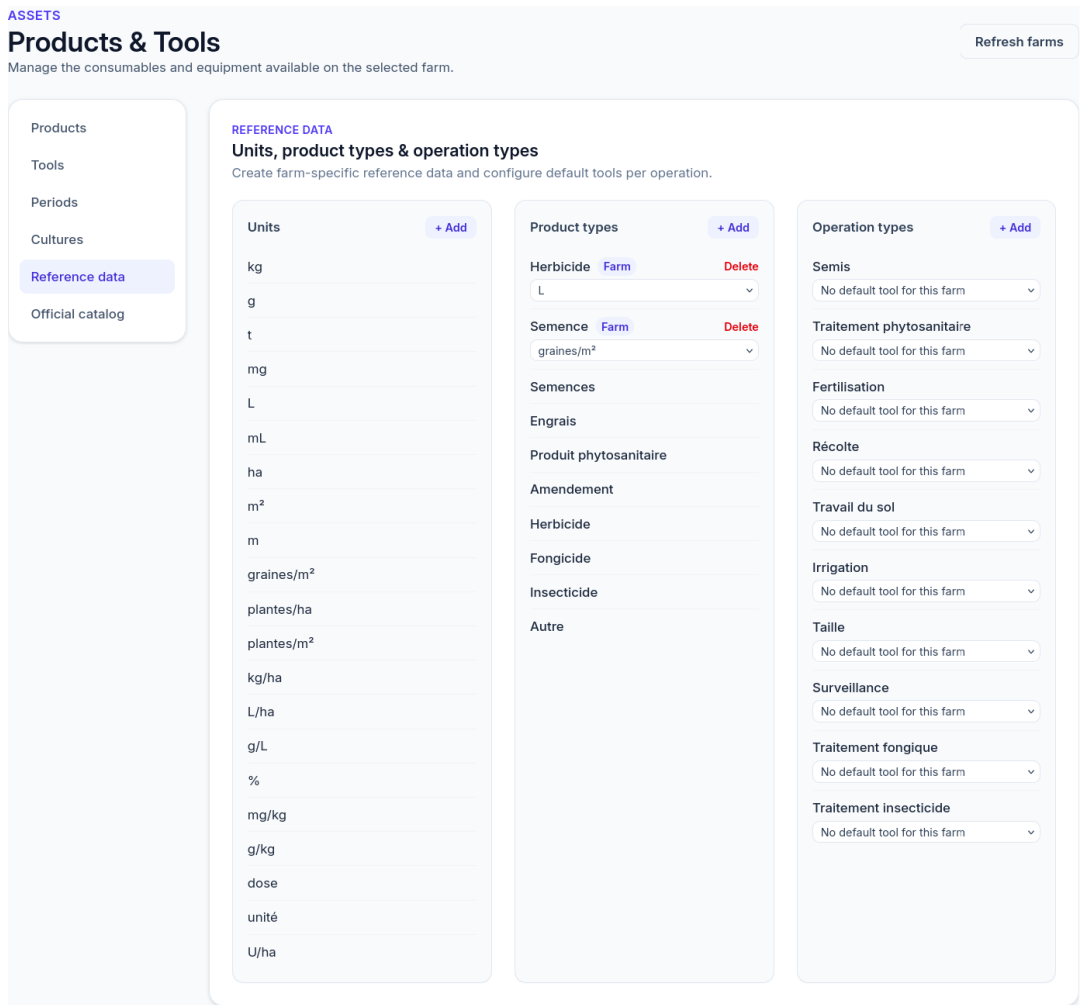


Figure A.2: The reference-data screen: the measurement units, the product types and the operation types (each operation type carrying its default tool), all editable per farm or promotable to the global set.

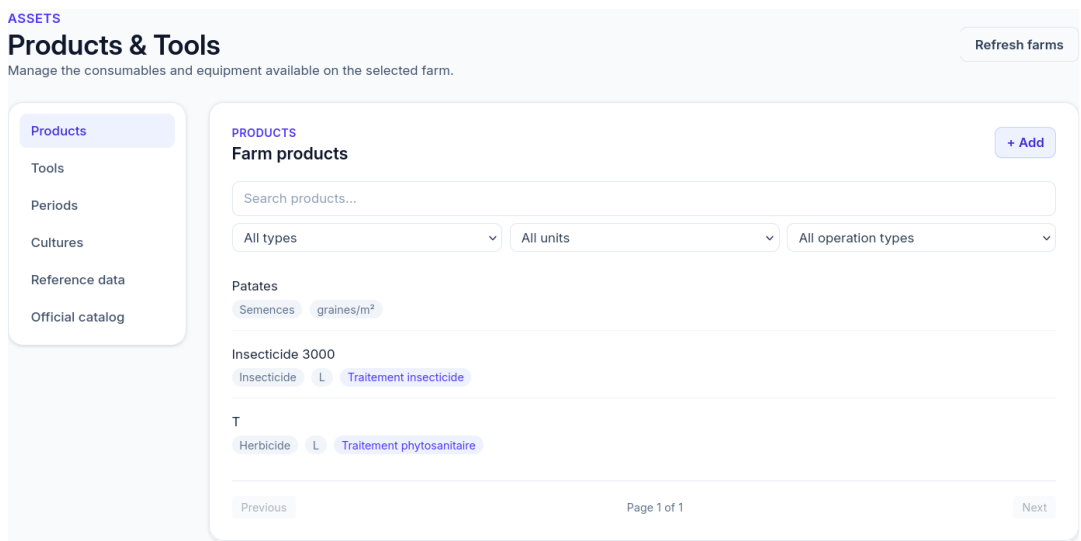


Figure A.3: The farm products screen: each product with its type, unit and the operation type it is used for, searchable and filterable, with the option to add a new one.



Figure A.4: The farm tools screen: the equipment held by the selected farm, searchable and filterable by category.

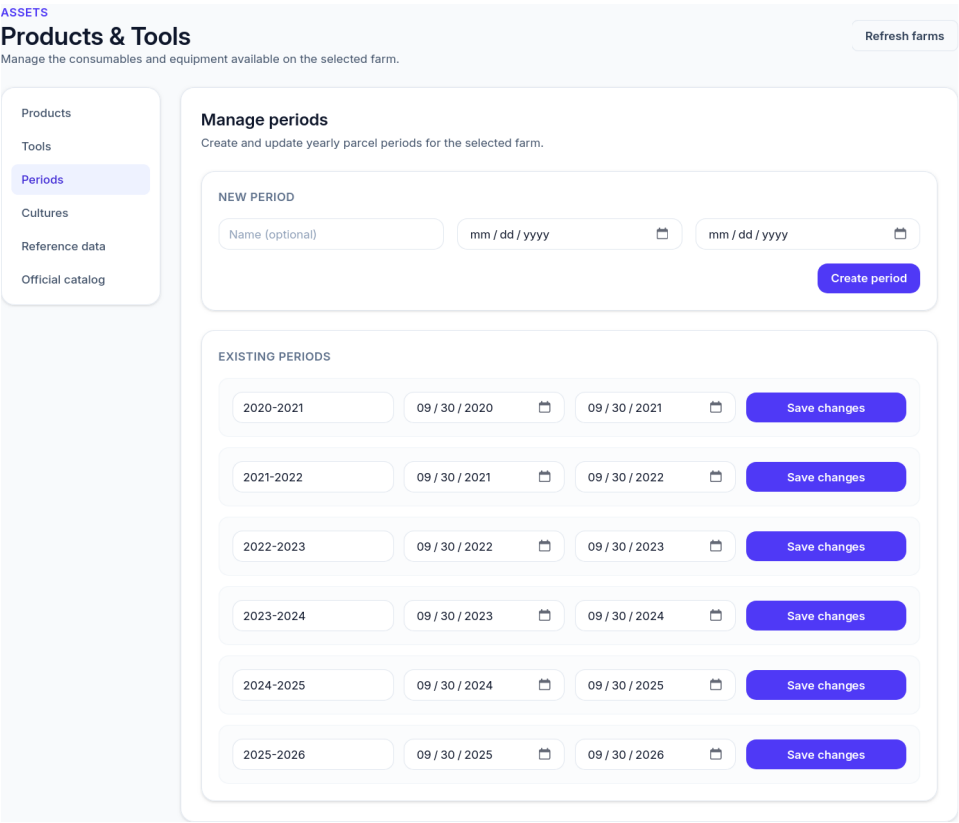


Figure A.5: The periods screen: the yearly campaigns defined at the farm level, each a named window with a start and an end date, that a parcel attaches to through a `ParcelPeriod`.

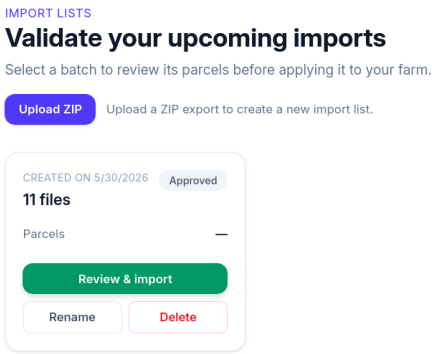


Figure A.6: The imports list: each uploaded bundle is a batch that can be reviewed before being applied to the farm, then renamed or deleted.

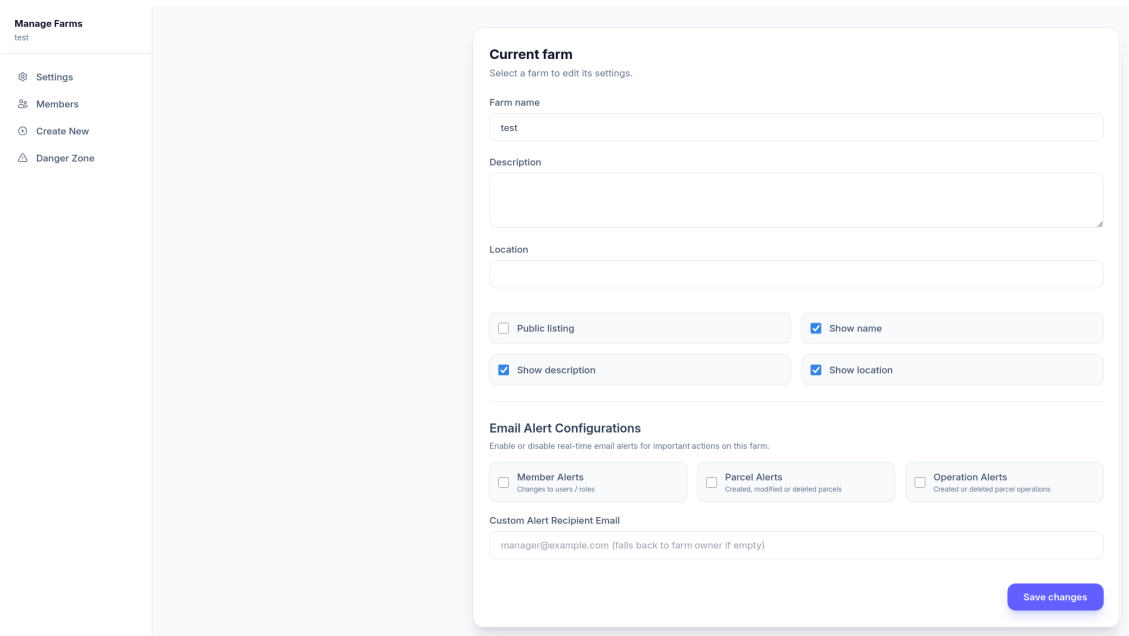


Figure A.7: The farm settings screen: the name, description and location, the *public* flag with its separate toggles for the name, the description and the location, and the per-action email-alert switches.

PROFILE

Manage your account

[Back to map](#)

**ACCOUNT**
farmer1
ID: 2
[Change avatar](#)

[Reset](#) [Reload](#)

Username
farmer1

Email
you@example.com

PREFERENCES

☒ **Open operations popup at top-right**
Also used on the map operations popup.

Time format
How clock times are shown across the app.
[24-hour \(13:45\)](#) [12-hour \(1:45 PM\)](#)

Date format
Auto matches your language; pick a fixed format to override.
[Auto \(from language\)](#)

Language
Overrides your browser language.
[English](#)

Default farm
Auto-selected after fresh login. A refresh keeps whatever farm you last used.
[No default \(show picker\)](#)

Default period per farm
The map opens with this period filtered by default. Leave blank to default to the most recent period.

Fermes UCLouvain (TEST)	2025-2026
test	2025-2026

☒ **Email notifications**
Master switch for shares, imports and parcel alerts.

[Save changes](#)

Figure A.8: The profile screen: the avatar, and the preferences stored on the user row, the interface language, the time and date formats, the default farm and per-farm default period, the operations-popup position and the opt-in for email notifications.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl