FieldMind

An Agricultural Management Tool

A Manuscript

Submitted to

the Department of Computer Science

and the Faculty of the

University of Wisconsin–La Crosse

La Crosse, Wisconsin

by

Jacob McAllister

in Partial Fulfillment of the

Requirements for the Degree of

Master of Software Engineering

April, 2025

FieldMind

By Jacob McAllister

We recommend acceptance of this manuscript in partial fulfillment of this candidate's requirements for the degree of Master of Software Engineering in Computer Science. The candidate has completed the oral examination requirement of the capstone project for the degree.

Prof. Jason Sauppe Examination Committee Chairperson	Date
Prof. Kenny Hunt Examination Committee Member	Date
Prof. Dipankar Mitra Examination Committee Member	Date

Abstract

McAllister, Jacob, T., “FieldMind,” Master of Software Engineering, April 2025, (Jason Sauppe, Ph.D.).

FieldMind is a web application designed as an agricultural management tool tailored to the needs of crop farmers in the Midwestern United States. It enables farmers to track a variety of critical metrics across their operations, including fields, crop planting and harvesting, jobs, inventory, and equipment usage. From this data, FieldMind generates a robust suite of data visualizations, providing farmers with meaningful insights into the state of their farm for a given year, as well as trends observed across a range of years.

Acknowledgements

I would like to express my sincere appreciation to my project advisor Dr. Jason Sauppe for his invaluable guidance and untiring support. I would also like to express my thanks to the Department of Computer Science at the University of Wisconsin–La Crosse for providing the education for me to build the skills needed to embark on a project such as this. The education and experience that I have gained at the University of Wisconsin–La Crosse has been truly unique.

My gratitude extends as well to Glenn Christ, my uncle and a crop farmer in Iowa, who generously served as a consultant for this project. Without his insights and thoughtful feedback, FieldMind would not be what it is. His support and the time that he dedicated to this work were invaluable.

Table of Contents

List of Tables

List of Figures

Glossary

DTO

Data Transfer Object. An object used to encapsulate data and send it from one subsystem of an application to another, often used to transfer data between the backend and frontend.

Entity Framework (EF) Core

An open-source object-relational mapper (ORM) for .NET applications. It allows developers to work with a database using .NET objects.

LINQ

Language Integrated Query. A feature in .NET that allows developers to write queries directly within C# or other .NET languages to retrieve and manipulate data from collections, databases, XML, and other data sources using a consistent, readable syntax.

Bushel

A unit of volume used for measuring agricultural commodities. In the context of corn, one bushel equals 56 pounds of dry corn.

Scale Ticket

A document generated when a load of grain is delivered to a facility such as a grain elevator or ethanol plant. It typically includes details such as gross weight, tare weight, net weight, test weight, and moisture content.

Crop Rotation

The practice of growing different types of crops in the same area across different planting seasons to improve soil health, reduce pest and weed pressure, and enhance crop yield.

Yield

The amount of crop produced per unit area, commonly measured in bushels per acre in the Midwest.

Tare Weight

The weight of an empty vehicle or container, used in combination with gross weight to calculate the net weight of the crop delivered.

Gross Weight

The total weight of the vehicle or container when it is full, including the weight of the crop.

Net Weight

The weight of the crop itself, calculated by subtracting the tare weight from the gross weight.

Test Weight

A measurement indicating the density or quality of grain. It represents how many pounds a bushel of grain weighs under standardized conditions.

Harvest

The process of gathering mature crops from the fields. This is a key operation in the farming cycle and is tracked in FieldMind to assess performance and yield over time.

Harvest Summary

A year-specific overview of the harvest performance for a given crop type within a particular field. It typically includes the total net yield (in bushels) gathered from all harvest load trips associated with that crop in the field. Farmers use this information to calculate yield per acre by dividing the total harvested bushels by the number of acres that the crop occupied in that field. Harvest summaries help evaluate crop performance, inform planning for future seasons, and track year-over-year trends.

1. Introduction

1.1. Overview

When considering a farm, one might initially envision a vast cultivated space akin to an expansive garden - an area managed by an individual or a small group tending to crops. However, modern agricultural operations are far more complex, involving extensive land divisions, specialized equipment, and data-driven decision-making to optimize productivity and resource allocation. Running a farm means running a mass production business. A typical farm in the United States has about 463 acres of land (1). In Iowa typically will yield around 200 bushels per acre of corn (2).

In Iowa specifically, the average number of crop acres per farm in 2023 was reported to be 587 acres (3). In 2024, Iowa farms operated 30 million acres, with 12.9 million of those acres dedicated to corn cultivation (4). The typical yield was 211 bushels per acre, with total corn production reaching 2.63 billion bushels (4). While a more detailed explanation of the term "bushel" will be provided later, it is sufficient for now to understand that, for corn, one bushel is equivalent to 56 pounds. Based on that, approximately 147.1 billion pounds of corn were harvested in Iowa alone (4).

Farms must provide resources for a nation, while also generating the financial income to support their own operations, the family of the farmer, and all of those who tend to the farm. The operational costs associated with sustaining these efforts are equally substantial. In 2023, Iowa's agricultural sector spent over $3.28 billion on fertilizers and related inputs, $1.95 billion on agricultural chemicals, over $1.37 billion on farm supplies and repairs, and more than $2.38 billion on seeds and plants (5).

Managing such operations requires the tracking of large amounts of data, at a very granular level, over spans of years for a farm which is going to vary what crops are planted in which fields, while also requiring the need to contrast various statistics among crops and fields. This is where this project, FieldMind, positions itself.

FieldMind aims to use software to put information into the hands of farmers. It is a web application that allows farmers to track their fields, jobs on their fields, crops, and their harvests. A farmer is able to track this data per planting year, and can toggle between years to see the state of their farm for whichever year that they would like to view.

Not only can data be entered and managed through FieldMind, but FieldMind also creates visualizations of the data so that the farmer can garner insights on various metrics of their farm. These insights come in a variety of scopes for three main categories: fields, crops, and jobs.

For each of these categories, such as fields, the farmer can view data and visualizations from the point of view of the **entire farm**, comparing all of the fields on the farm with each other. A **single field** can be viewed and data and visualizations specific to that field can be viewed. For each perspective, whether looking at the farm as a whole or a specific field, the user can toggle between which year they would like to see data for.

Not only can this data for the entire farm be viewed for a **single year**, but data can also be viewed for each perspective over a **range of years**. This perspective - whether comparing fields across the entire farm or analyzing data for a single field - allows trends to emerge over a selected range of years, providing valuable insights.

These same insights from the data can be extracted with the other entities of the crops and jobs as well. By using all of these various perspectives to view data and operations across the farm, FieldMind is able to provide rich insights about these metrics for any given year, and how they have changed over time.

The target user for this application is a Midwestern crop farmer - specifically one farming grain crops such as corn, soybeans, and wheat. Corn and soybeans are the most prevalent crop types farmed in the Midwest. Though this alone would be a perfectly good reason to target this user group for FieldMind, the major reason that this demographic was chosen was because I have personal experience in this kind of farming through my family. My uncle, Glenn Christ, is a crop farmer in Iowa, farming corn and soybeans. Glenn agreed to be a consultant on this project, being both a great source of information and giving valuable feedback during demos of the project.

Consulting with Glenn has been an integral part of creating an application which has utility for farmers. Glenn was able to convey the various ways that farms operate and the metrics which are of concern to farmers that help them assess the performance of their farm and the fields which comprise it. The consultations with Glenn were pivotal not just in determining the functional capabilities of FieldMind, but also in ensuring that the application has labels which are meaningful to farmers. Based on how Glenn described to me the process in which farmers will plant, harvest, and track their crops, I've built a user interface which should coalesce nicely with the farmer's workflow.

As technology has progressed, it has made its way to the agricultural industry as well. This penetration has been of assistance to farmers in some ways, and there are some products out there which are similar to FieldMind, such as farmbrite, AgWorld, Granular Insights, and more. However, from the research that I have done, many of these products are quite expensive. There are some additional issues as well that have come with this growing integration of technology into agriculture.

In the past, this has required significant amounts of paperwork - and for many farmers this is still the case. Though technology has made its way into agriculture through a variety of avenues, there are still some significant obstacles. I would characterize two of the biggest obstacles as trust and cost.

When looking at a typical farm in Iowa many of these farms are family owned and have been passed down generationally. The average age for a farmer in Iowa is 57, and more than half of all farmers are 45 years of age or older (2). People in this age demographic did not grow up using software solutions as liberally as those in a younger age demographic. Instead, farmers in this age demographic have grown accustomed to alternative methodologies for tracking and managing their farm, and have inherited the data and methodologies from those who came before them.

It is easy to understand a hesitation to migrate to a new way of operation when the stakes are so high. Additionally, even when a given farmer is keen on using a software solution, there is the issue of what to do with data which was not managed in a software system previously. For example, if one is wanting to get a comprehensive view of a farm over a twenty year timespan, at first adoption, this data is not going to all be there in the system. So there is the ever-present issue of integrating previous data into a new system. This typically takes a commitment of time and resources, and not always viewed as a priority.

The aforementioned issue is one aspect of trust - trusting that something is worth the risk - but there is another major issue as well. Right now in agriculture as in many other industries, there is a growing trend to use software not to put power into the hands of the users, but instead into the hands of the company issuing the software. This is one of the barriers to entry that FieldMind has when thinking about it as something that would be used by farmers. Products which are similar to FieldMind present data ownership issues as they will often claim ownership over the data which has been generated by a farm. These companies can in turn, analyze, aggregate, and potentially sell this data to third parties. The idea of a farmer's data being made available like this, or having the risk that a given company could bar access to the data is also an unpleasant idea.

For FieldMind to overcome this, I think that always allowing the farmer to have access to their data and, in future iterations of FieldMind, to export their data out of the application is important. These actions should always be treated as the user's right to do. At the end of the day, their data is theirs, and nothing should stand in the way of that. FieldMind's current state allows users to export the charts that have been created from the data, but some additional work will need to be done to export all of the data in a format that is organized and will make sense for the farmer.

FieldMind combats hesitation for the use of a software solution by giving great emphasis on visual representation of data and actions which can be performed. Great effort has gone into making the application's user interface (UI) as intuitive as possible and using recognizable symbols over wordy layouts. The flow of how the application works has also been carefully considered so as to signal to the user how to maneuver about the application.

The key aspect is ease of use and having as low of a learning curve as possible. The last thing that a farmer, or anyone running such a large scale business, wants to concern themselves with when in the midst of operations is how to navigate a new software tool. Software should be there to serve the user and should feel like a collaboration rather than an obstruction.

1.2. Background

There are some key terms which are used in crop farming, and are thus used throughout this paper, which are important to know when looking at FieldMind.

When a Midwestern grain crop farmer plants crops in a field and later harvests those crops, a general process is followed: The harvested crop is placed into a wagon or semi, and then, generally, either taken to a drying bin on the farm, or taken to a grain elevator or ethanol plant. A grain elevator is a facility which stores, handles, and can also process the crop. These facilities will determine how much crop has been **yielded** from a particular load that was brought in, and will purchase this crop from the farmer. The ethanol plant will do the same thing. One of these trips is called a **harvest load trip** or a **load trip**.

Whether the crop is taken to a grain elevator or to an ethanol plant, something called a **scale ticket** will be produced. Among the information on the scale ticket, the following information is most pertinent for our current discussion on yield:

- **Gross Weight**: The weight of the semi or wagon upon arrival at the facility, when it is full of the crop.
- **Tare Weight**: The weight of the semi or wagon when empty.
- **Test Weight**: The weight of the crop required to fill one bushel (a more detailed discussion on this follows shortly).

The net weight from a load is then determined as a calculation of the tare weight subtracted from the gross weight. This net weight is then often given in terms of bushels of corn which was yielded from that given load.

A **bushel** is a volume measurement. Though it is a volume measurement, a certain weight in pounds of a given crop is typically used as the correlation of the quantity of the crop per bushel. For instance, though a bushel is a volume measurement, to fill that volume with kernels of corn, it is typically going to take a certain amount of pounds of corn. Just as you may say that a certain poundage of sand, for instance, is needed to fill a 5 gallon bucket - the same idea is used here with bushels.

With this idea of how a bushel of a crop, like corn, is correlated to the weight of corn that is needed to fill that bushel, the idea that this may vary based on certain states of the corn may cross your mind. And you would be correct. For instance, perhaps a farmer's corn is a little larger or smaller than another farmer's - or maybe has more or less moisture. These factors can cause a given farmer's corn to be more voluminous, or more or less heavy. Therefore we have two important terms - a **test weight** and a **standard weight**. The **test weight** is the amount of pounds of **your** crop it takes to fill a bushel. The **standard weight** is the agreed upon weight that is used as a measure for how many pounds of a given crop is needed to fill a bushel. The **standard weight** is what used is for payment and what is used to assess yield on a field.

The standard weights for crops tracked by FieldMind are as follows:

- Corn: 56 pounds per bushel
- Soybeans: 60 pounds per bushel
- Wheat: 60 pounds per bushel

To assess how well a particular field has done, the net yields (in bushels) from all of the harvest load trips for the field (a harvest load summary), are added together and are then divided by the number of acres that crop covered in the field. This will result in the yield for a field in **bushels per acre**. This is what is used to assess the output produced by a given field. Not all crop yields are tallied in the form of bushels per acre, but the kinds of crops which FieldMind tracks are.

2. Software Development Process

FieldMind was developed using Agile and Scrum methodologies. Agile was the most suitable methodology for this project due to its flexibility, iterative approach, and continuous feedback loop with the project's sponsor.

When beginning this project, I had a clear overall objective: to create a software management tool for Midwestern crop farmers to track various farm metrics and analyze data through visualizations. However, many specific details were uncertain, such as:

- How a farm is structured in terms of fields, sections, and crops.
- The exact workflow of planting, harvesting, and tracking farm data.
- The key insights farmers prioritize when making decisions.

Given these unknowns, a rigid, upfront design approach would not have been effective. Instead, an iterative process was necessary to incorporate findings along the way, refining features and workflows based on evolving understanding.

2.1. Comparison with Alternative Software Development Models

Several other software development methodologies exist, but they were less suitable for this project for reasons outlined below:

1. Waterfall Model

The Waterfall model follows a linear, sequential approach, where requirements are gathered upfront, and development proceeds in distinct phases (e.g., Design → Implementation → Testing).

This model was not appropriate for FieldMind because it assumes that all requirements can be fully defined at the beginning of the project. In reality, FieldMind's development required frequent adjustments as new insights were discovered through consultations with agricultural stakeholders. Additionally, the rigid structure of the Waterfall model makes it difficult and costly to return to previous stages once a phase is completed. Given the complexity and evolving nature of farm management workflows, an iterative approach was far more suitable.

2. V-Model (Verification & Validation Model)

The V-Model is an extension of Waterfall where each development phase has a corresponding testing phase before moving forward.

Despite its emphasis on verification and validation, the V-Model was not a good fit for FieldMind. Like Waterfall, it relies on the assumption that all requirements can be clearly specified at the start of the project, which was not the case here. FieldMind's design evolved as understanding of real-world farming needs grew. The V-Model's rigidity made it difficult to accommodate these necessary mid-development changes, making it too inflexible for this type of project.

3. Spiral Model

The Spiral Model is an iterative approach that combines elements of Waterfall and risk assessment, emphasizing early identification of risks and prototyping.

While the Spiral Model is well-suited for projects with high risk or uncertain feasibility, it was not appropriate for FieldMind. The project did not require the extensive risk analysis and prototyping phases that the Spiral Model demands. FieldMind was designed as a practical, goal-oriented application using well-understood technologies.

4. Kanban

Kanban is a visual workflow management method that focuses on continuous delivery rather than time-boxed iterations.

Although Kanban is effective for managing ongoing work and maintenance tasks, it lacks the structure offered by Scrum, particularly in terms of fixed Sprint cycles and milestone planning. For FieldMind, it was important to have defined intervals of progress and scheduled feedback sessions. These characteristics are essential in a project that evolves based on stakeholder input. The more structured approach of Scrum allowed for clearer planning and timely demonstrations of progress, which made it a better fit than Kanban.

2.2. Why Agile and Scrum Were the Best Fit

Agile's core principles—flexibility, collaboration, iterative development, and incremental delivery—aligned closely with the needs of FieldMind. Within the Agile framework, Scrum was especially effective for managing the project's evolving requirements and ensuring consistent progress. The ability to develop features in small, incremental iterations allowed me to continually refine FieldMind as new insights emerged. Regular consultations with my project advisor, a Midwestern crop farmer, provided valuable real-world feedback that guided these refinements and ensured the application remained practical and relevant to its intended users.

As the project progressed, a deeper understanding of how farmers manage and track data led to several adjustments in features and workflows. Scrum's adaptability made it well-suited for incorporating these evolving requirements. Additionally, to support effective task management, I used Notion, a web-based productivity application that enables flexible organization of notes, task boards, to-do lists, and databases. Notion served as a lightweight and intuitive tool for managing the project backlog, breaking down complex features into actionable tasks, and tracking progress throughout development. This helped ensure the project stayed organized and aligned with development priorities, supporting the iterative and collaborative nature of the Scrum methodology, all while being backed up on Notion's web servers.

2.3. Task Organization and Execution

For tracking development tasks, Notion was used as a central hub for the following:

- **Task Backlog** – A list of all features needed for FieldMind.
- **Sprint Planning** – Prioritizing tasks for each development cycle.
- **Feature Implementation** – Breaking down tasks into frontend, backend, and data model components.
- **Progress Tracking** – Keeping notes on incomplete tasks and areas requiring further refinement.

Whenever I stopped working on a feature, I recorded notes in Notion to minimize the ramp-up time when resuming development later.

3. Requirements

Glenn, my uncle who is an Iowa crop farmer, agreed to act as a consultant for this project, and he provided invaluable insights and direction for many detailed aspects for how a farm of this kind works, and what insights and tracking tools would be of value for a Midwestern crop farmer. Before even embarking on this project, I conferred with Glenn to see if the concept itself would indeed be something of value for farmers. After expressing that it would indeed be of value, Glenn was able to give details about how a crop farm is typically structured - the mechanics of how planting and harvesting work, the various measurement units which are used for each, and the manner in which yield is determined and tracked.

Throughout the project, I relied on Glenn's expertise and domain knowledge in order to inform both the direction of FieldMind and further research of my own. For example, some of the standard weights, soil nutrient information, and other information of the like is made available by the University of Iowa through their agricultural division. They put out formal information regarding these specifications. When doing research on my own, many of the items that I found assumed some prerequisite domain knowledge in order to properly contextualize the information. The information was either too niche or too broad for the specific questions I had related to developing FieldMind. Additionally, there were certainly times when I didn't even know the question to ask. This is where having Glenn as a consultant was an invaluable asset.

The insights that I was able to gather from Glenn and the ability to ask direct questions in the context to how FieldMind should handle certain aspects of its workflow was something that I don't think I could have achieved through Internet research alone. Research would have given me broad strokes of a direction to go and what to implement, but I think the details about FieldMind which make it shine were only able to come through via the direct consultation that I was able to get from Glenn.

From my early meetings with Glenn, I was able to get clear vision of the requirements which FieldMind would need to embody. A crop farmer has a number of essential items to manage on the farm - and it is not just the tracking of these items, but also the assessment of them.

The functional requirements of FieldMind are as follows:

1. The system must provide secure login and account management functionality.
2. Each account must be associated with a single farm.
3. Users must be able to track fields on a farm. For each field, the user must be able to record acreage, crops planted, crops harvested, yield per acre, yield over time, and the jobs associated with the field.
4. Users must be able to track crops. For each crop, the system must allow the user to specify the crop type, the amount planted, the area it covers, the planting date, and the location of the crop within the farm.

5. Users must be able to track jobs. The system must record which jobs are currently active, the status and urgency of each job, the job category, the location of the job, and the inventory used by each job.

6. Users must be able to track inventory. Inventory must be tracked by type, which includes:

   - **Individually tracked items**, such as a planter, grinder, or diesel storage tank. These items are reusable and their use does not reduce inventory stock.
   - **Bulk tracked items**, such as fertilizer, diesel fuel, seed bags, or screws. These items are consumed upon use, and their usage reduces the inventory stock. For each bulk item, the system must allow tracking of:
     - The quantity in stock,
     - The reorder level,
     - The reorder quantity, and
     - The batch number, if available.

   Additionally, the inventory system must allow tracking of cost per unit and the date the item was added.

7. Users must be able to track equipment. For each equipment item, the system must specify its category, any farm resources it requires (e.g., diesel fuel), and the quantity of those resources required for its operation.

8. The system must provide data visualizations for fields, crops, and jobs. These visualizations should reflect various tracked metrics within each entity to support user insight and decision-making.

The data visualizations are a critical part of the application - particularly the visualizations revolving around yields on the fields. These insights allow the farmer to track how this crucial aspect of the farm is performing, and they can track this over time.

The non-functional requirements are as follows:

1. The application must use terminology which is accurate for the industry domain.

2. The application should use intuitive symbols whenever possible and avoid excessive verbiage.

3. The application's design must be intuitive to use and must not require an instruction manual to understand.

4. Data visualizations must provide useful, detailed insights without crowding the layout.

5. The application should encourage interaction with the charts from the user.

My process for gathering requirements began with initial research to familiarize myself with what I could about farming operations in general. After this initial information gathering, I composed a variety of questions and then consulted with Glenn and received clarifications on the information I had gathered as well as the questions that I had. I would also ask Glenn to elaborate on various processes that would be followed for tasks such as the planting, fertilization, and harvesting of crops. Through doing this, I was able to get a far more nuanced view of what the operations at hand entail, both how those would translate into tangible requirements for the construction of the software, as well as how to convey these processes visually on the application to a farmer in a way that is meaningful to them.

After these meetings with Glenn, I would take my hand written or typed notes and diagrams and would organize them with Notion. I used Notion to create a backlog of tasks, broken down by category, and would consult this to create daily tasks and sprints. This was very useful as I was able to both create these lists of tasks as well as write notes for myself about any issues that came up with any of the tasks, or things that I would need to either look into further on my own or would need to meet with Glenn again and receive more clarification on.

While building FieldMind, I periodically checked with Glenn to make sure that the structure and process that I was building into the application was representative of what is done on a farm. Often, while building the project, intricacies would manifest themselves that I'd not thought of initially. When these would come up, I would schedule a meeting with Glenn in order to shed light on the approach that I should take.

As mentioned, the charts are one of the the core features of FieldMind. Therefore, once I'd built enough charts, I was eager to demo this to Glenn in order to get feedback on whether the charts that I'd created were showing and comparing metrics that would make sense for a farmer, and would provide value. I also wanted to be sure that the measurement units that I was using, such as bushels in certain places and pounds in others, made sense given the context for a given chart.

The feedback that I gathered from Glenn allowed me not only to refine the charts but also make sure that the labels that I was giving to certain features of the app would be something that farmers would recognize.

4. Design

I believe that well-designed applications should not require an instruction manual. Instead, the design and layout should lend itself as much as possible to intuitive behavior with recognizable patterns. Achieving this requires thoughtful design in both the backend architecture of the application and the user interface (UI).

FieldMind is designed to support farmers in managing the complex operations of a crop farm. At a high level, the system enables users to track the structure of their land, manage the planting and harvesting of crops, oversee farm-related jobs, and monitor inventory usage. In addition to serving as a data entry and management tool, FieldMind also emphasizes data-driven insights through the use of visual analytics.

The central behavior of the system revolves around managing the crop lifecycle. Users can enter data on planted crops for a particular year, specifying details such as crop type, amount planted, area covered, and relevant dates. Harvest data can also be entered, including yield information such as bushels per acre. These entries are contextualized within the farm's fields, allowing data to be organized and filtered by specific areas of the farm.

A unique aspect of FieldMind is its handling of time. All data entries are linked to a specific planting year, which the user can select and toggle across the application. This allows the farmer to view their farm's state as it was in a given year. In addition, some parts of the application support selecting a range of years, which enables the user to view patterns and trends across time.

To make this data meaningful and accessible, FieldMind generates a variety of interactive visualizations. These charts – such as bar charts, donut charts, and sunburst charts – enable users to compare yields between fields, evaluate job distributions by category or urgency, and assess how inventory is being used. The insights pages in particular allow the user to analyze this data across multiple years, surfacing longer-term patterns that may inform future decisions.

To construct FieldMind, it was necessary to develop an in-depth understanding of farming operations and the structural organization of a typical Midwestern crop farm. These details were not known at the outset, meaning that the complete set of requirements – as well as the design of the frontend layouts – could not be fully determined in advance. Instead, requirements emerged gradually through consultations with Glenn, whose insights helped shape both the functionality and user interface of the system. As development progressed, the implementation of previously gathered requirements often revealed additional complexities that required further clarification. In such cases, I conducted independent research to better understand the issues at hand and then consulted with Glenn to gain practical insight grounded in real-world farming operations. I tailored my solutions based on Glenn's experiences, as he was able to provide a realistic perspective on the workflows and priorities of Midwestern crop farmers – FieldMind's target user group.

Beginning with the backend architecture of FieldMind, I first thought about the data models, or entities. These are the models which will have a direct mapping to a database table. Since all of the functionality of the application revolves around the manipulation of the data models, it makes sense to me to start here when first building a project. The core entities which drive the application are the `LandDivision`, `Crop`, `CropType`, `HarvestedCrop`, `JobTask`, `IndividuallyTrackedItem`, and `BulkTrackedItem` entities. Being that FieldMind is an application for crop farmers, the central entity is one which represents a field. The field is the central entity because the planted and harvested crops, as well as the ultimate yield calculations, all are associated with a field. For this crucial data model, I created an entity called a `LandDivision`.

The `LandDivision` entity is what the crops and harvests are associated with. The `LandDivision` entity and the `Crop` entity have a one-to-many relationship from the `LandDivision` to the `Crop`. Each year, the crop planted in a given field may change – and often does every few years due to crop rotation. The `LandDivision` and the `HarvestedCrop` also have a one-to-many relationship as well. Along with these relationships, the `LandDivision` entity also needs to represent the scenario of a farmer subdividing a field into smaller divisions, each of these divisions having their own set of crops and potentially subdivisions of its own. Therefore the `LandDivision` entity has a `ParentId` property which will either hold the ID of the given `LandDivision`'s parent, or it will hold the value of `null` to represent that the given `LandDivision` has no parents and is thus a "root division".

Each `Crop` entity also has a `CropType` which, as the name suggests, contains the information about what kind of crop it is - such as corn, soybeans, or wheat. The `CropType`, entity also contains information about the yield measurement unit, via a `YieldMeasurementUnit` entity, that the given `CropType` uses. In the case of FieldMind, this is "bushels" for all the crops that it currently allows for, but by having this be contained in its own entity, this provides room for growth of other yield measurement units, as well as the increasing of the properties that the `YieldMeasurementUnit` has.

The `Crop` entity represents the planted crop on a given `LandDivision`, and has properties relevant to this such as the amount planted, the area covered, the planted date, crop type, and the harvest completion date. One of the restrictions that are placed on a `Crop` entity when being associated with a `LandDivision`, is that only one `Crop` entity per `CropType` can be associated with any given `LandDivision`. For example, a `LandDivision` can have multiple `Crop` entities associated with it, but each of those `Crop` entities each have a distinct `CropType`. This is important because when the crops are harvested, a `HarvestedCrop` entity is used to track this, and they are grouped by `CropType` for a `LandDivision` to produce the derived type of a `HarvestedCropSummary`. The `HarvestedCropSummary` is not a database entity due to it being derived from the `HarvestedCrop`s for a given `LandDivision`.

When a crop is harvested, it needs a different set of properties and thus is represented by the `HarvestedCrop` entity. In practice, what the `HarvestedCrop` is a representation of a harvest load trip. As mentioned in the **Background** section of this paper, when a crop is harvested, it is typically loaded up into either a semi or a wagon which is then taken to an elevator, ethanol plant, or some other buyer. One of these trips is a harvest load, and this will produce a scale ticket. It is the information on the scale ticket which the `HarvestedCrop` entity is modeled after. Like the `Crop` entity, there is a one-to-many relationship from the `LandDivision` to the `HarvestedCrop`.

Figure 1 shows a the portion of the Entity-Relationship (ER) Diagram which focuses on the `LandDivision` entity its relationship with these entities.

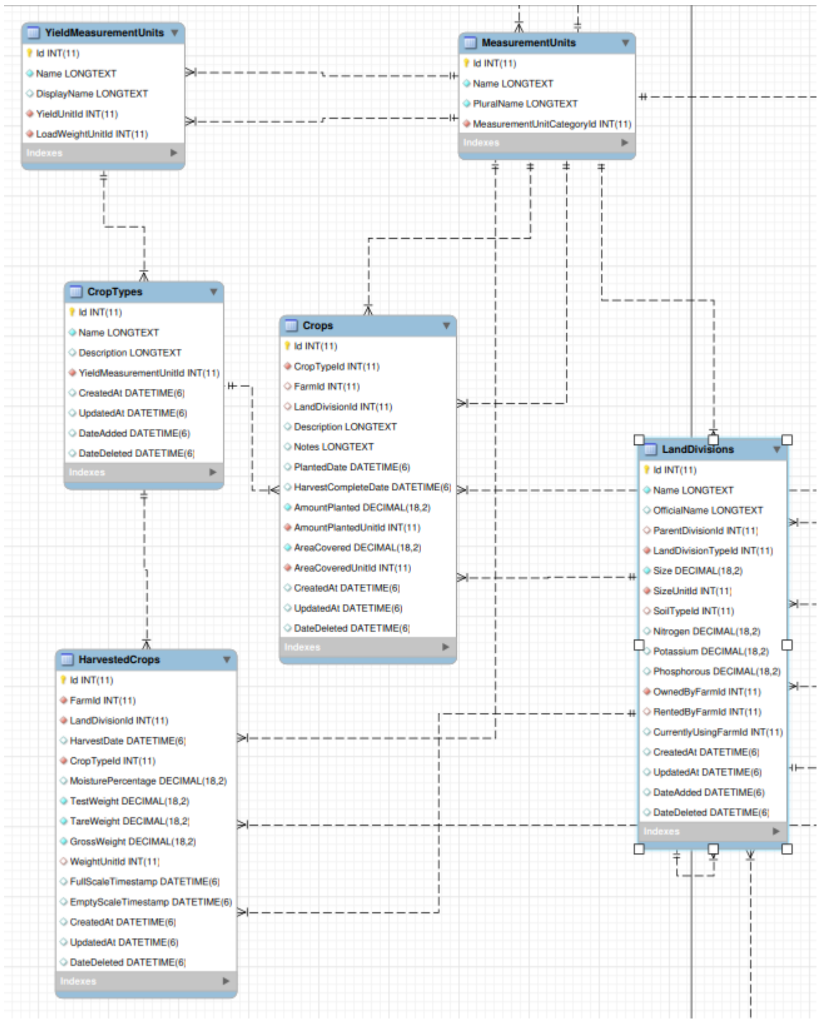


Figure 1. Partial view of ER Diagram focusing on relationships between the `LandDivision`, `Crop`, `CropType`, `HarvestedCrop`, and some measurement unit entities.

Because the `LandDivision` entity has a one-to-many relationship with both `Crop` and `HarvestedCrop`, and because these associated records change from year to year, it becomes necessary to compute certain values dynamically. Some of these values – such as the Harvest Summary – are useful to display as if they were direct properties of a `LandDivision`. However, since they are derived from related data, it would violate database best practices to include them as actual columns on the entity's table. To address this, a data transfer object (DTO) called `LandDivisionDto` was created to hold these computed values alongside the core properties. Through the use of the DTO, the frontend is able to get a representation of the "full" `LandDivision` object as it would expect to see – with all of its crops and harvest data for a particular year – while still being able to separate the database entity into its appropriate parts to follow best practices in this way.

DTOs (Data Transfer Objects) are used for most entities in FieldMind to provide an appropriate representation of the data for the frontend. In many cases, the frontend requires either a subset or a superset of the properties defined on the original entity. A subset may be necessary for security reasons—for example, omitting sensitive fields from a user object. More commonly, however, a superset of information is needed. This is especially true for DTOs like `LandDivisionDto`, which include additional or derived properties to support frontend functionality such as data visualization. These properties may include metrics related to the `LandDivision`'s parent division (e.g., acreage), details about its subdivisions, or a collection of jobs associated with that `LandDivision`. DTOs are also used during the creation or modification of entities, as the frontend typically transmits only partial representations of these objects to the backend.

The `JobTask` entity is used to track jobs which are done on the farm. These have the option of being associated with a `LandDivision`. When they are, there is once again a one-to-many relationship from the `LandDivision` to the `JobTask`. Each `JobTask` has a `JobCategory` that it is associated with, and each job can also have inventory associated with it in the form of either `IndividuallyTrackedItem`s, `BulkTrackedItem`s, or both. A job can also have one or more `Equipment` associated with it.

The key distinction between the two types of inventory lies in how they are consumed. `IndividuallyTrackedItem`s refer to reusable items that are not depleted through use. For example, if a drill set is used for a job, it remains in inventory afterward; its usage does not reduce the total quantity of drill sets. In contrast, `BulkTrackedItem`s represent consumable inventory that **is** reduced upon use. For instance, applying 20 lbs of fertilizer results in a 20 lbs reduction in available inventory, reflecting its consumption.

To facilitate tracking of bulk inventory consumption, a `JobTaskInventoryUsage` entity was introduced. When a job is created and one or more `BulkTrackedItem`s are associated with it, those items are marked as available for use within that job. The actual quantity used is specified after job creation when viewing or editing the job details. It is this recorded usage that draws down inventory levels and is tracked by the `JobTaskInventoryUsage` entity. This action can be seen in figures 32 and 34 later in this section.

To further illustrate the main entities involved in supporting job-related functionality within FieldMind, Figure 2 presents a partial view of the ER Diagram. This excerpt highlights the core entities related to jobs and their relationships with other components in the system, helping to visualize how job data is structured and interconnected.

For the frontend design, thankfully, the Fuse template gives a great starting point for the

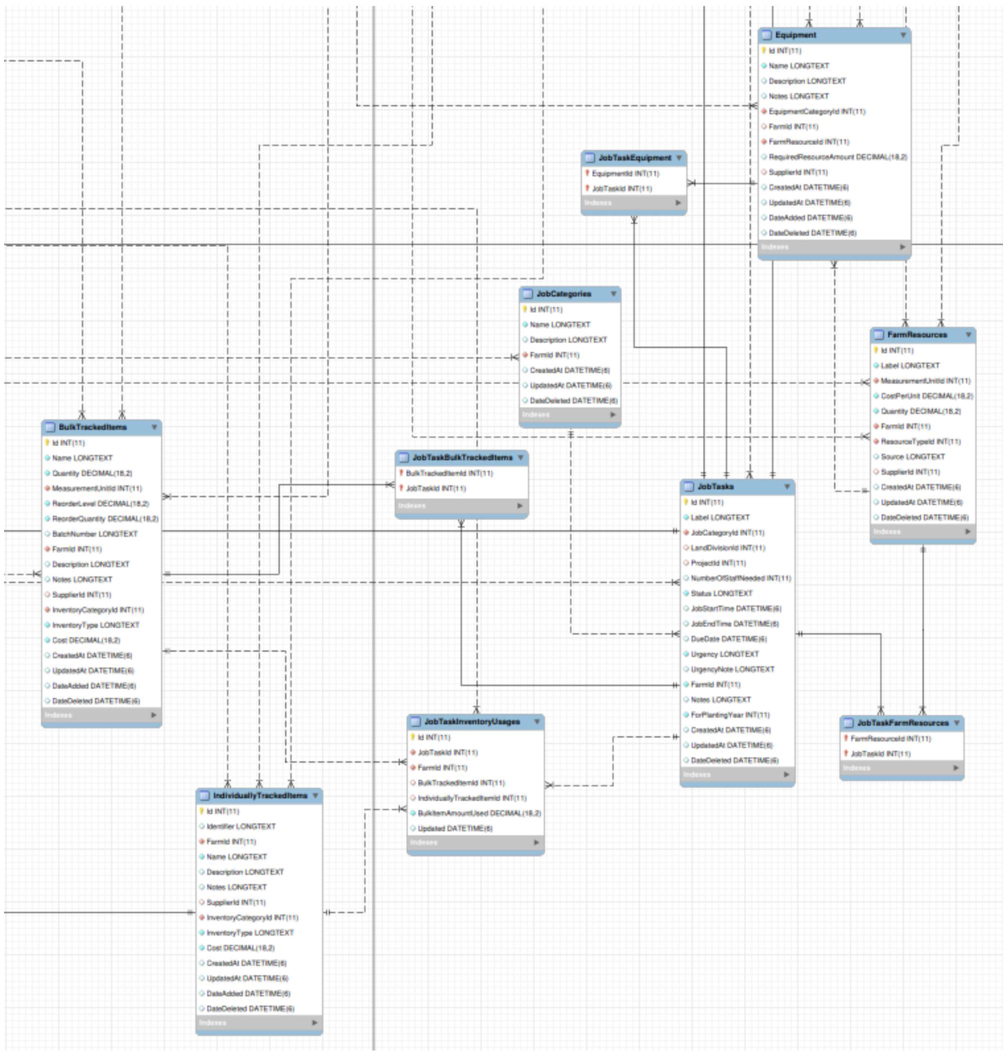


Figure 2. Partial view of ER Diagram focusing on relationships between tables related to the `JobTask` entity.

layout of the application. To decide how to fill out the template with my customizations, I first started with the navigation menu. I filled out each option with what I, as a user, would expect to see an interact with there. This meant a highlight of the main entities of FieldMind of Fields, Crops, Jobs, Inventory, and Equipment. Additionally, there is a dashboard which combines highlighted features and insights from each of the entities; and, an "Insights" option which allows for the the examination of Fields, Crops, and Jobs over a year range rather than just for a single year.

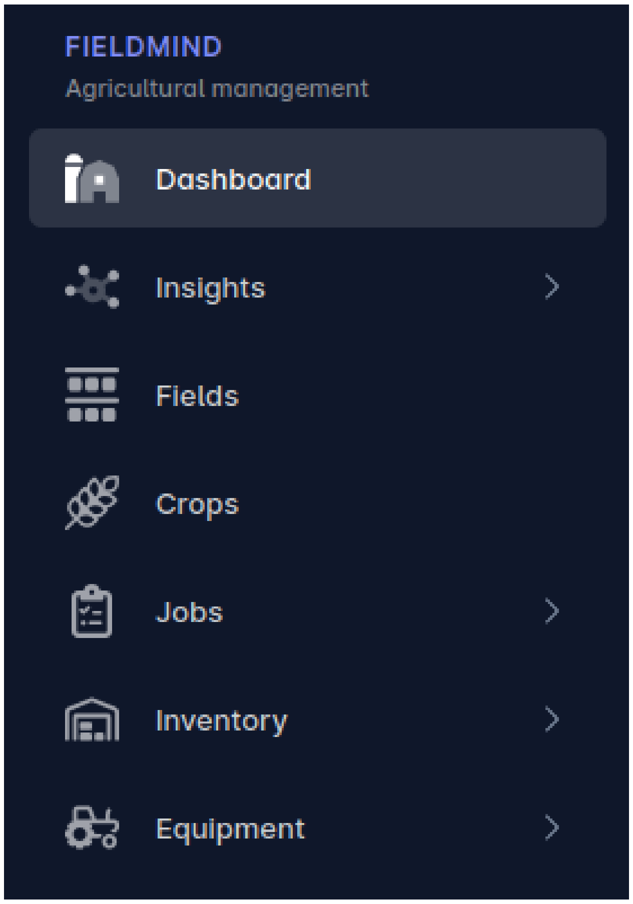


Figure 3. FieldMind navigation menu.

The side navigation bar, shown in Figure 3, shows the high-level structure of the frontend of FieldMind, as well as some core entities. Both the “Dashboard” and “Insights” are related to the “Fields”, “Crops”, and “Jobs” options; therefore, these will be addressed first, and we will return to the ”Dashboard” and ”Insights” shortly. Each one of these is going to allow the user to see statistics and options regarding all fields, crops, or jobs, respectively, across the entire farm. Each one of those pages represents data from the perspective of that data type. Each one of these pages also allows you to toggle the year for which the data is shown as well. The “Dashboard” allows you to see an overview of highlights from the entire farm. This page showcases some of the most notable charts from other pages.

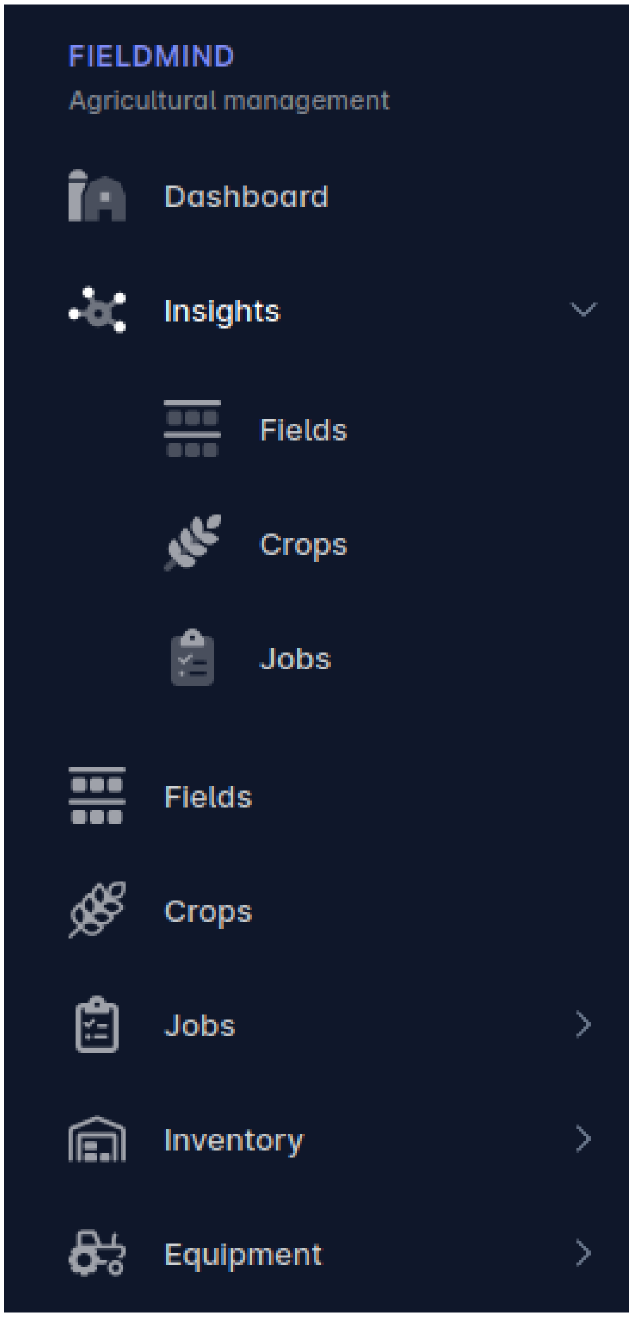


Figure 4. Insights option toggled open in the navigation menu.

The “Insights” option opens up and presents the options of “Fields”, “Crops”, and “Jobs”. Each of these again presents data for each of these entities at the forefront, except the unique perspective this time is that these pages allow the user to look at a **range** of years, rather than just a single year. By allowing the user to toggle between a start and end year, and subsequently displaying data for the respective entities for the selected time range, a different variety of insights can be gleaned and patterns over time can begin to manifest.

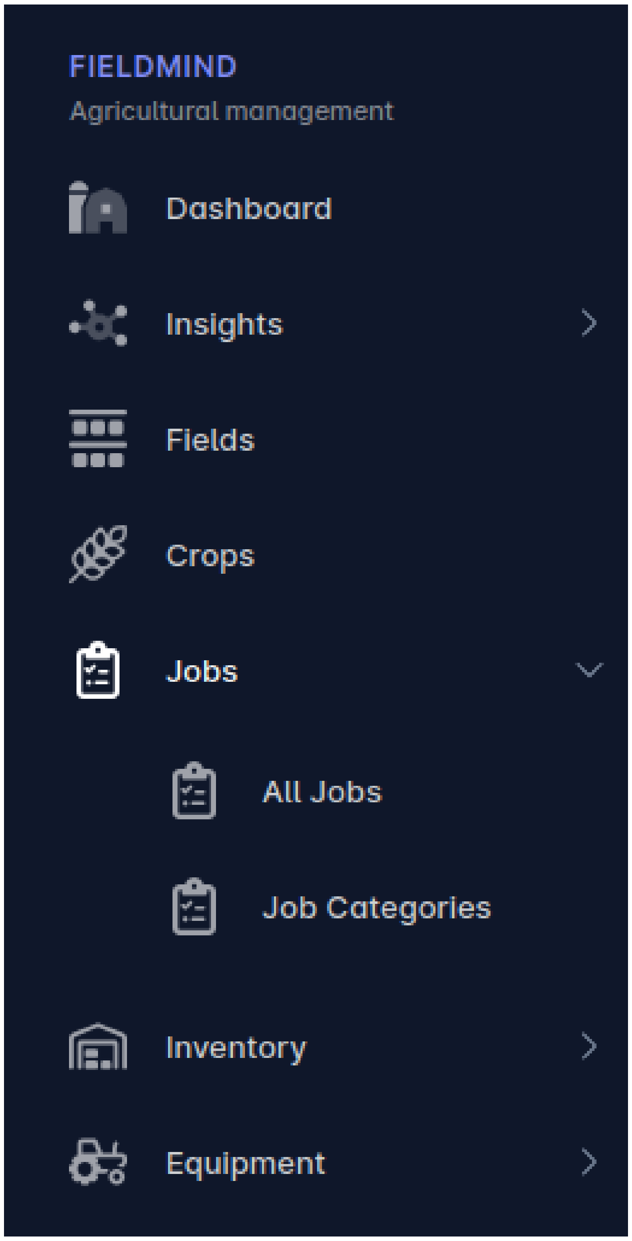


Figure 5. Jobs navigation menu toggled open.

Each of the “Jobs”, “Inventory”, and “Equipment” options in the navigation menu are also expandable and reveal the option to see and modify categories for each of these entities. Figure 5 shows the “Jobs” option toggled open as an example.

Certain implementation decisions have introduced restrictions on specific actions and choices. The measurement units, for example, have been chosen by the system ahead of time and only allow for specific types. For example, only measurements in the imperial system are allowed, and only acres are allowed for area measurements rather than also allowing for hectares, square feet, etc. This limitation exists to avoid the complexity of unit conversions and rounding adjustments. An example of what is meant with rounding of measurement units would be if you had 18 inches as a measurement. This could be turned into 1.5 feet. Likewise, the same could be done in the other direction when considering fractional amounts of a foot.

FieldMind also has predefined crop types which it allows, and it does not yet permit the user to add additional crop types. This is because of the standard weights which are associated with a given crop type for determining the yield on the field. The conversion of the harvested amount to the proper standard weight is something that would be unwise to place in the hand of the user as this will likely lead to clerical errors which would have the affect of causing FieldMind to produce incorrect calculations.

One other area of the design of FieldMind which takes careful consideration is what actions are and are **not** allowed for the user. An application needs to strike a balance between what **it** should guard against and what it must simply allow the user to do regardless of potential consequences. If too many guards are put into place, the application becomes overly restrictive, and thus stands in the way of the user instead of enhancing the user's abilities.

As an example, consider the file explorer program that all computers have. This application provides the user with a graphical way to move about and interact with the file system. Those of us who are programmers can see a program like this and seek to impose some rules on the file structure that we ultimately allow the user to create. One such rule may be that you do not have a directory (folder) with a subdirectory of the same name, creating a path like `apple/apple` for instance. However sensible this rule, and others of this kind, may seem, by having the application impose these rules upon the user, you take the application out of what its domain of concern should be and push it into the domain of concerns that should being to the user.

These kinds of restrictions do not add to the user experience but instead subtract from it, making the user feel that the application hinders rather than supports them.

Too many guards actually makes the application, or any tool, unusable. An application should only guard what it has to and nothing more. An example of this with our file explorer example would be not allowing a user to create two files or directories of the same name within the same level of a directory. For example, if we are in our `apple` directory, the file explorer will not allow us to create files each named `berry.txt`. If one wanted to have two files named `berry.txt`, one would need to house each file in different directories.

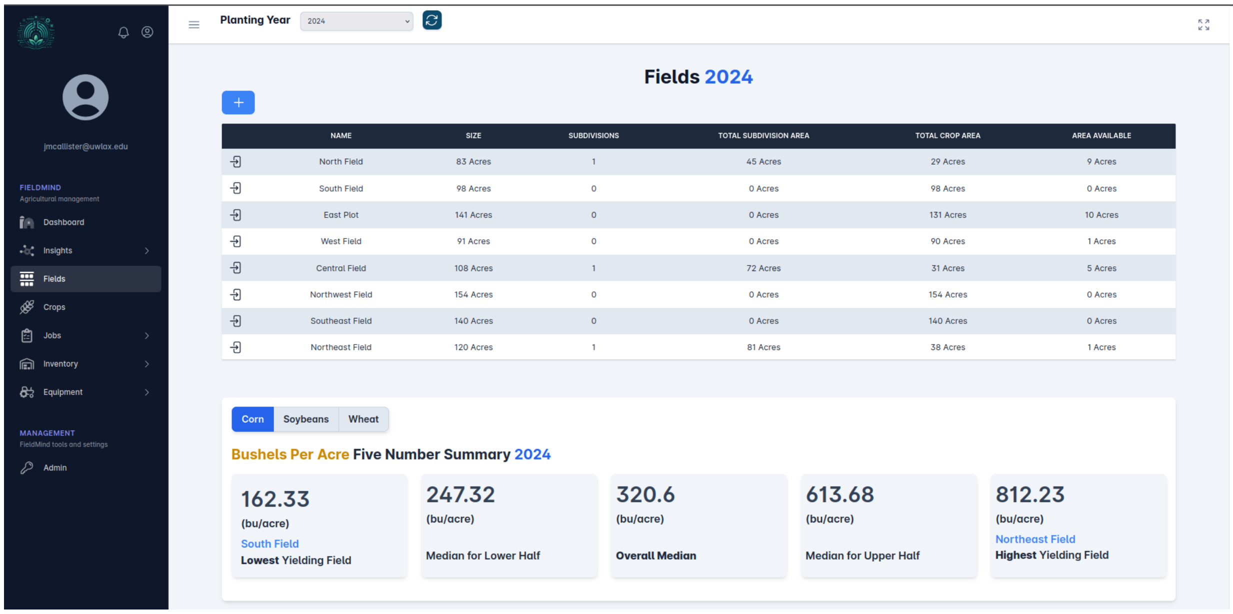
In addition to enforcing necessary system rules for functionality, FieldMind relies on a well-structured database consisting of 43 tables to ensure smooth operation.

4.1. Site Tour

The following section provides an overview of the major components of FieldMind. The discussion begins with the "Fields" section and proceeds in order through the navigation menu, eventually returning to the "Dashboard" and concluding with an examination of each of the “Insights” options.

4.2. Fields

When clicking on the “Fields” option, a table appears with one row for each root field - referred to as a `LandDivision` in the application. A `LandDivision` represents a field entity within the farm. Each row contains high-level information, and clicking on a row reveals its associated child subdivisions. Any `LandDivision` in the table can be selected to view a detailed page with more information.



Fields 2024

NAME	SIZE	SUBDIVISIONS	TOTAL SUBDIVISION AREA	TOTAL CROP AREA	AREA AVAILABLE
North Field	83 Acres	1	45 Acres	29 Acres	9 Acres
South Field	98 Acres	0	0 Acres	98 Acres	0 Acres
East Plot	141 Acres	0	0 Acres	131 Acres	10 Acres
West Field	91 Acres	0	0 Acres	90 Acres	1 Acres
Central Field	108 Acres	1	72 Acres	31 Acres	5 Acres
Northwest Field	154 Acres	0	0 Acres	154 Acres	0 Acres
Southeast Field	140 Acres	0	0 Acres	140 Acres	0 Acres
Northeast Field	120 Acres	1	81 Acres	38 Acres	1 Acres

Corn | Soybeans | Wheat

Bushels Per Acre Five Number Summary 2024

162.33 (bu/acre)	247.32 (bu/acre)	320.6 (bu/acre)	613.68 (bu/acre)	812.23 (bu/acre)
South Field				Northeast Field
Lowest Yielding Field	Median for Lower Half	Overall Median	Median for Upper Half	Highest Yielding Field

Figure 6. The top of the “Fields” landing page.

As one scrolls down this page, various charts compare and contrast metrics across all `LandDivisions` on the farm. The first thing that is encountered is a five-number summary of the bushels per acre have been harvested across all the `LandDivisions` for the selected planting year based on the crop type selected. As one continues to scroll, a tree map chart is encountered. This tree map shows a visualization of the farm’s area usage across all the fields.

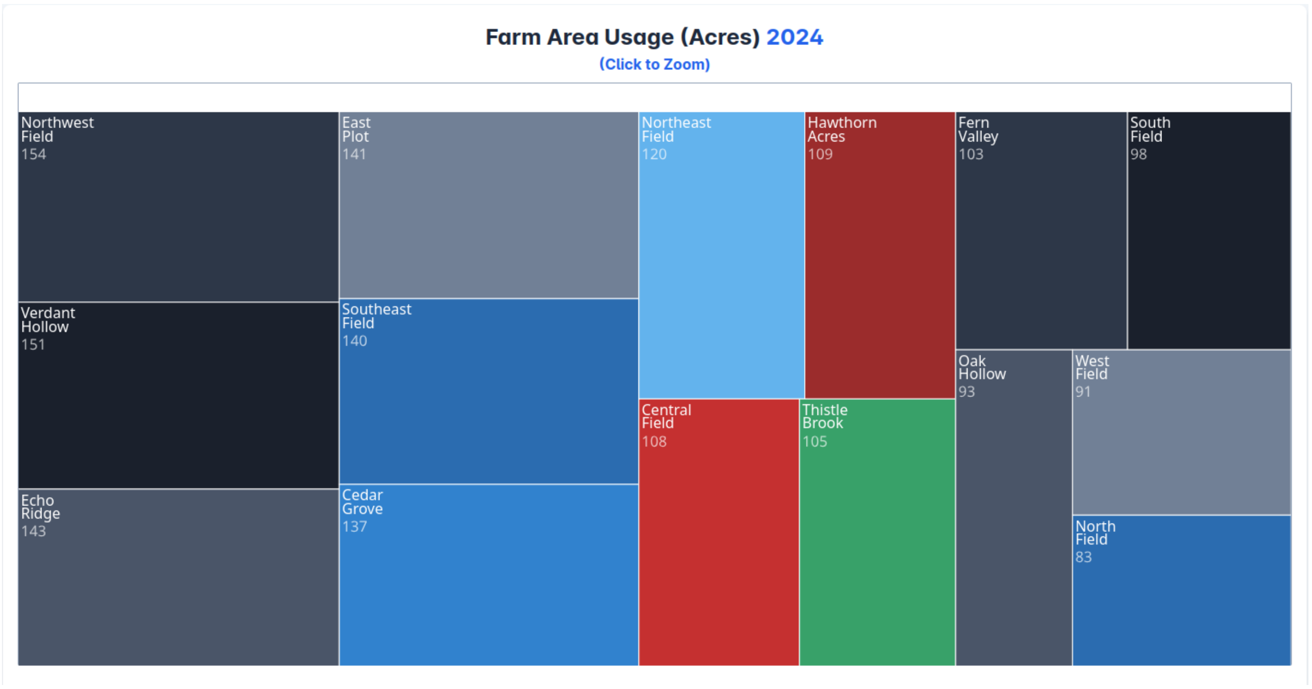


Figure 7. Farm area usage tree map.

We can see that the number of acres that each `LandDivision` is displayed in the top-left corner of each tile. When a tile is clicked on, the chart zooms into that `LandDivision` to see a further breakdown of what is contained within that `LandDivision`.

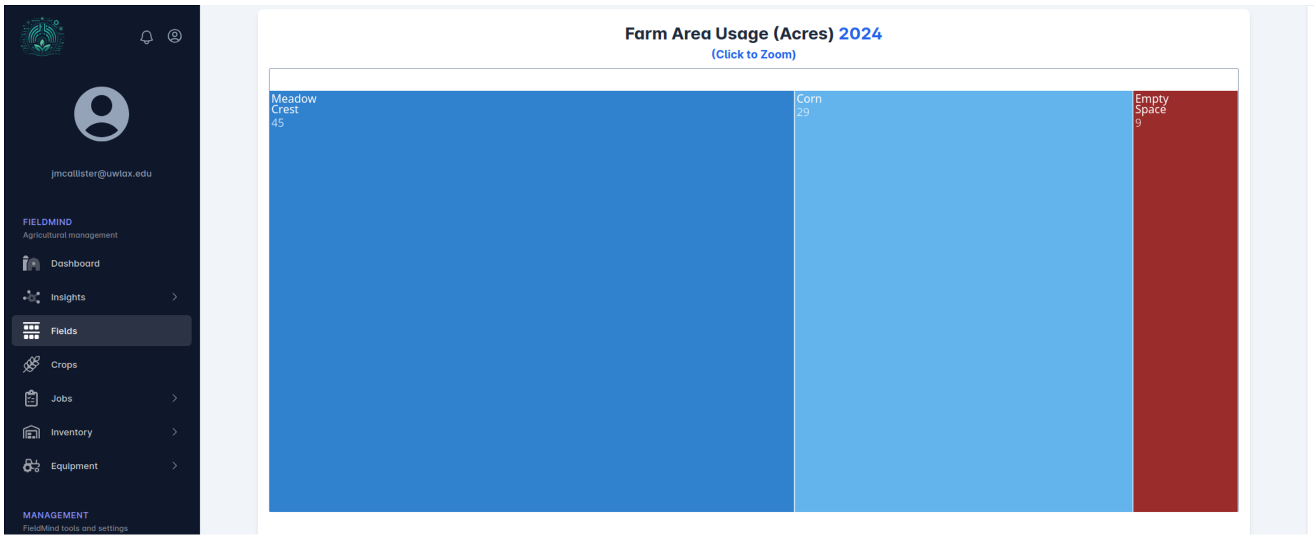


Figure 8. Zoomed in on the "North Field" tile.

In Figure 8, we can see what is contained inside of the North Field. We see that there are 45 acres used for its subdivision of Meadow Crest, 29 acres of corn, and then 9 acres of empty space. Because Meadow Crest is a subdivision, this can be clicked on and further zoomed in to see the contents of that `LandDivision`.

Continuing down on the “Fields” page, we can see charts which further compare `LandDivisions` on important metrics, such as yield per acre, number of harvest trips, soil nutrient levels, and the job category breakdown.

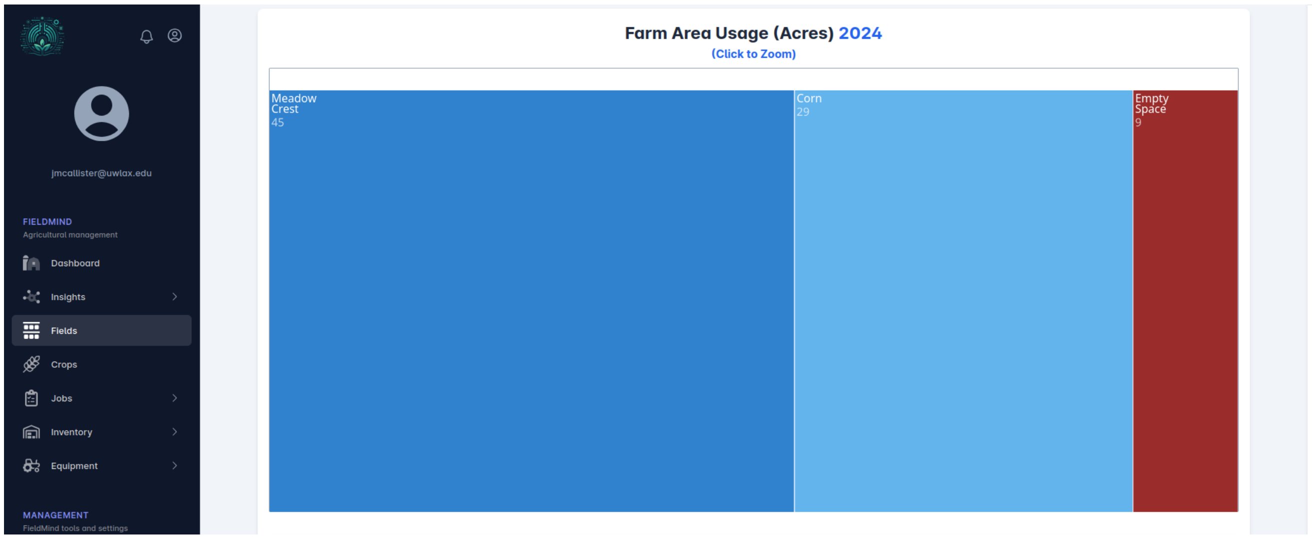


Figure 9. Zoomed in on the “North Field” tile.

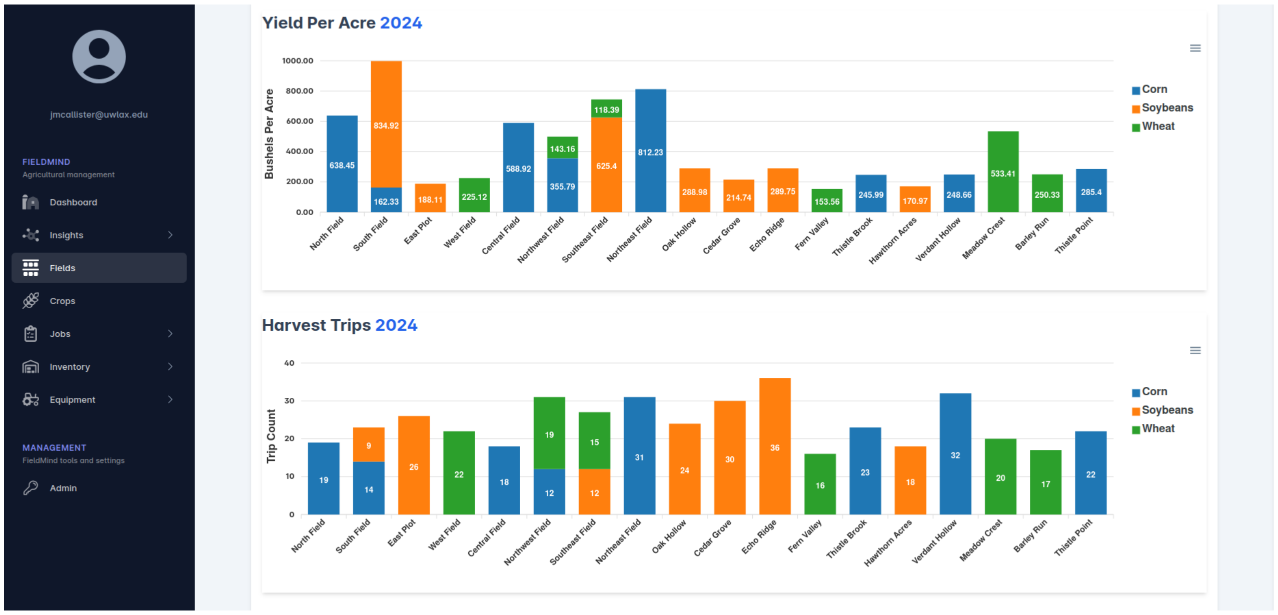


Figure 10. Yield per acre and harvest trip count charts.

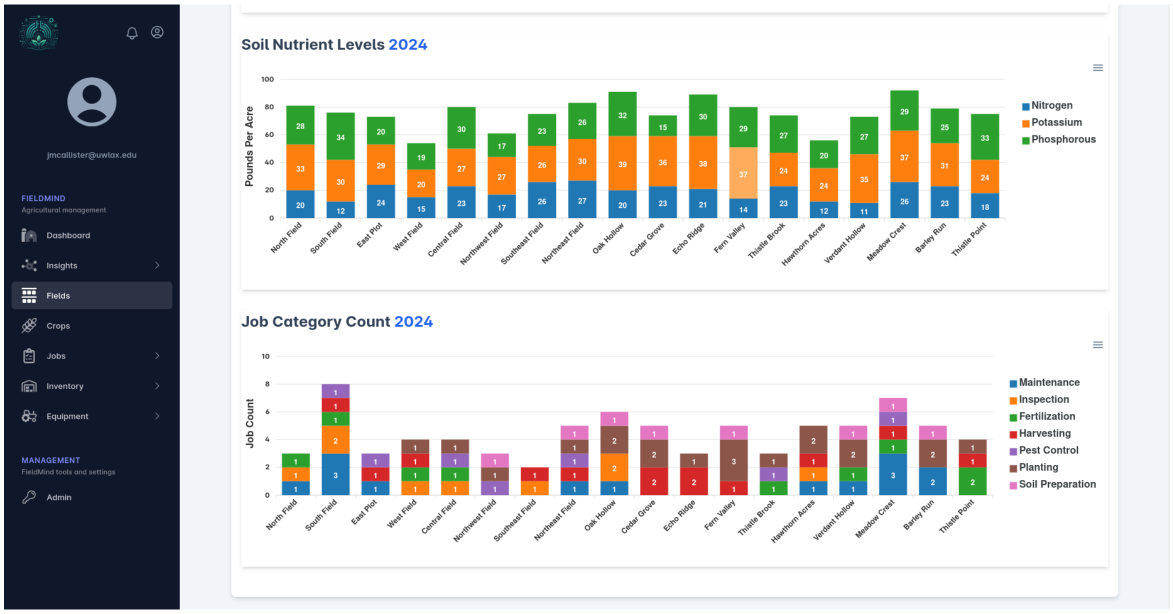
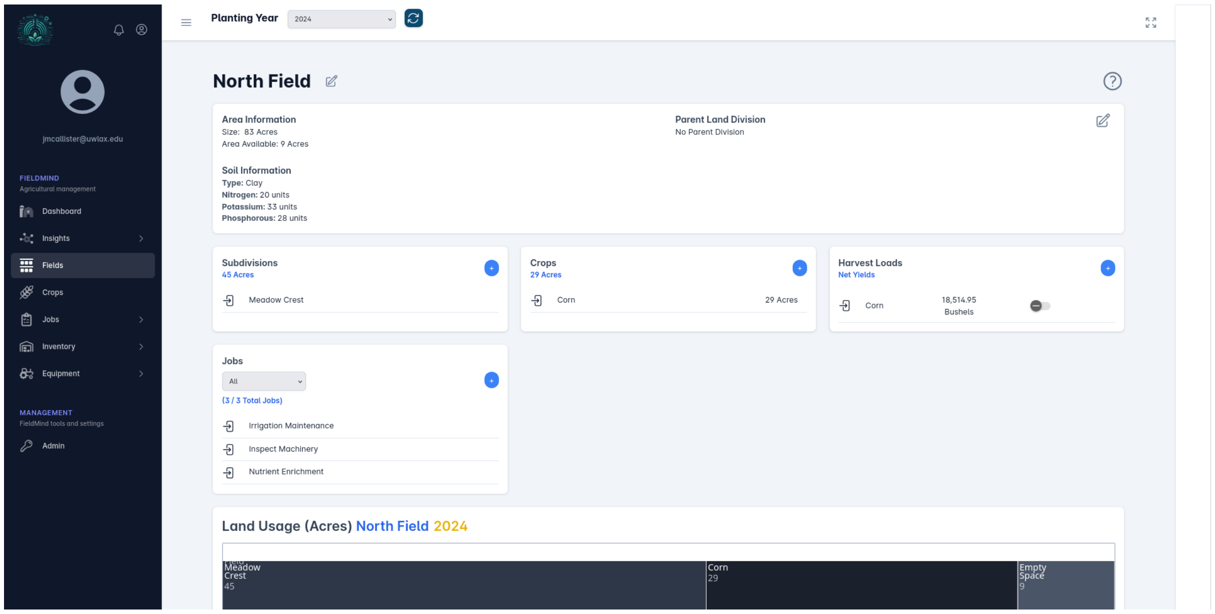


Figure 11. Soil nutrient level and job category count charts.

As mentioned, this page presents data from the perspective of the `LandDivision` entity, making it the focal point of the displayed information.

4.3. Field Details

When a specific `LandDivision` is selected from the table at the top of the page, details on that specific field for the selected planting year are shown.



Planting Year 2024

North Field

Area Information
Size: 83 Acres
Area Available: 9 Acres

Parent Land Division
No Parent Division

Soil Information
Type: Clay
Nitrogen: 20 units
Potassium: 33 units
Phosphorous: 28 units

Subdivisions
45 Acres
Meadow Crest

Crops
29 Acres
Corn 29 Acres

Harvest Loads
Net Yields
Corn 18,514.95 Bushels

Jobs
All
(3 / 3 Total Jobs)
Irrigation Maintenance
Inspect Machinery
Nutrient Enrichment

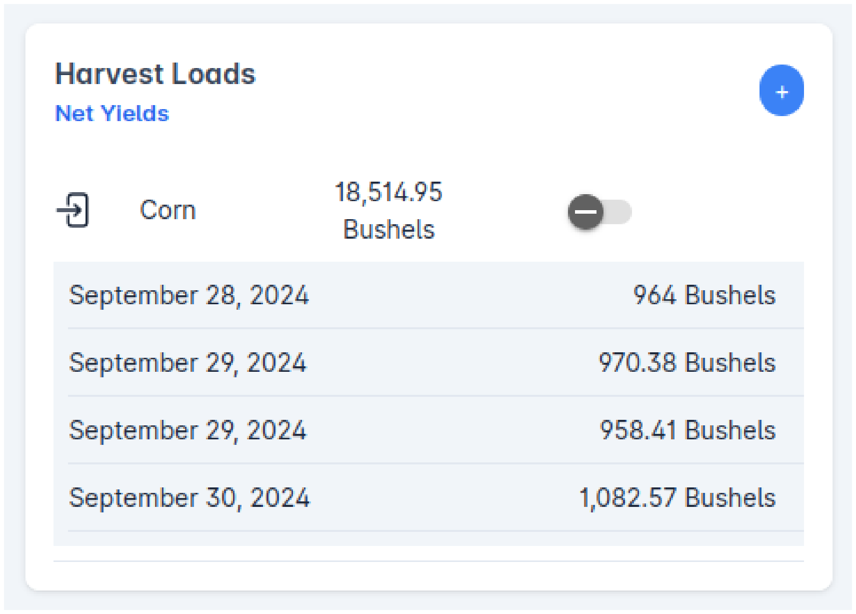
Land Usage (Acres) North Field 2024

Meadow Crest 45
Corn 29
Empty Space 9

Figure 12. Top of the Land Division details page.

From this page, one can manage various aspects of the `LandDivision` such as various metadata about it as well as adding subdivisions, crops, harvest load trips for those crops, and jobs. We can notice from Figure 12 that both the size and the area available are listed in the top section for the `LandDivision`. And for each of the **Subdivisions** and **Crops** tiles, the acreage which each of these take up is also listed. When creating a new subdivision or adding crops to the `LandDivision`, FieldMind will ensure that the area being added for either is within the area available. Additionally, when adding new crops to the `LandDivision`, it is enforced that only crops of a different crop type than what is already on the `LandDivision` is allowed to be added. For example, in Figure X, there is corn on the `LandDivision` already; therefore, only crops which are not of the `CropType` of "Corn" are allowed to be added to the `LandDivision`. These rules are important when it comes to the creation of `HarvestedCrops` since these are grouped by `CropType`.

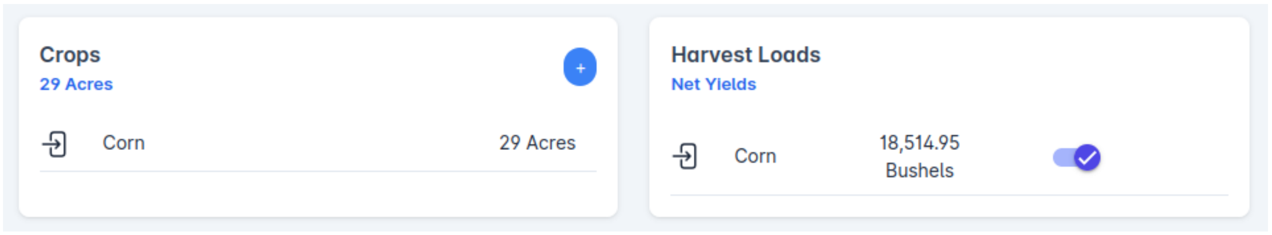
The **Harvest Loads** tile in Figure 12 currently shows the bushel count for the harvest summary for corn. All of the `HarvestedCrop` entities for corn for this `LandDivision` for the given planting year make up the harvest summary. When that row for corn is clicked on, it expands into a scrollable list of all of the trips (`HarvestedCrops`) for that crop. We can see this in Figure 13.


Harvest Loads
Net Yields

Corn	18,514.95 Bushels
September 28, 2024	964 Bushels
September 29, 2024	970.38 Bushels
September 29, 2024	958.41 Bushels
September 30, 2024	1,082.57 Bushels

Figure 13. Harvest load summary expanded.

Figure 13 shows a slider that can be toggled. This toggle is for marking this crop's harvest complete for this `LandDivision`. This will prevent further harvest trips from being created and will also internally timestamp that crop has having its harvest completed for that particular `LandDivision`. In future versions of FieldMind, this data could be used to create charts which visualize the efficiency of harvests.



Crops

29 Acres

Corn	29 Acres

Harvest Loads

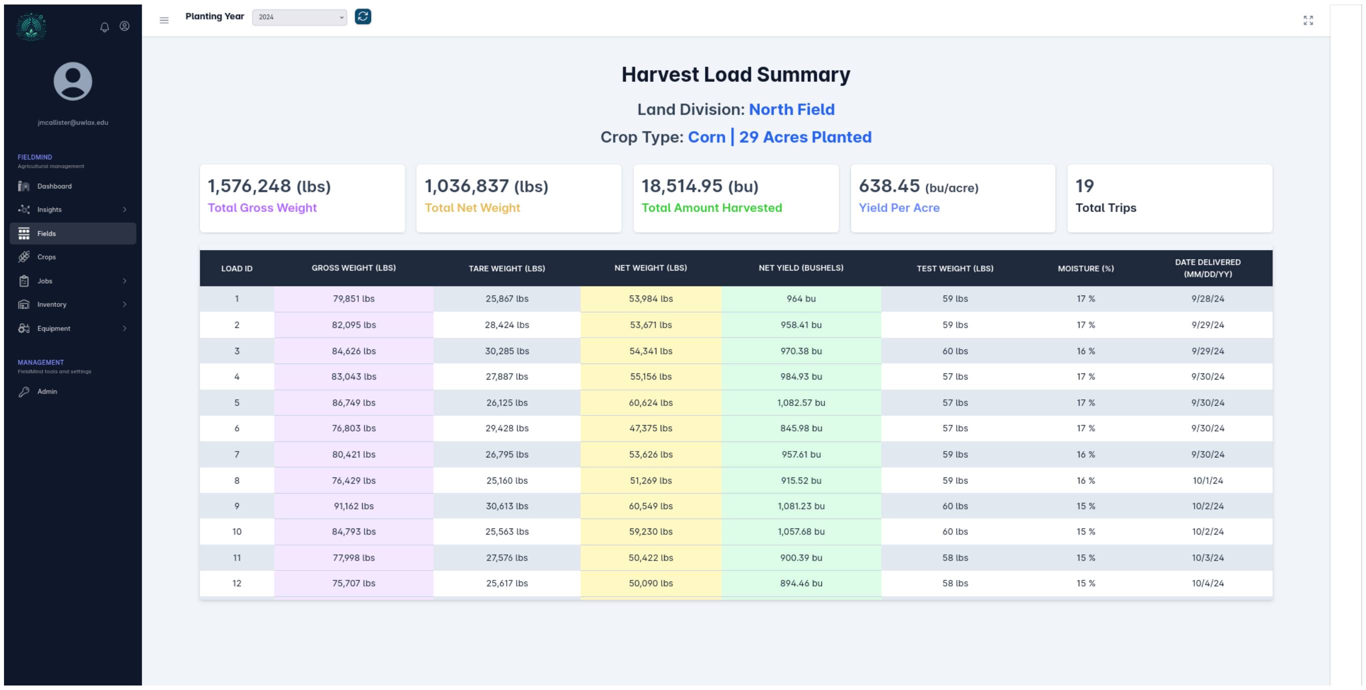
Net Yields

Corn	18,514.95 Bushels

Figure 14. Harvest complete for corn for the given `LandDivision`.

We can see in Figure 14 that the "+" button to add more harvest load trips has disappeared. This is because, for this `LandDivision`, there are no other planted crops other than corn; so, once the harvest has completed for corn, there are no more harvest load trips which can be created. If there were additional crops planted on the `LandDivision`, then the add button would not disappear. When on the form to create a new harvest load trip, the only option for `CropTypes` which can be selected are ones which are both on the `LandDivision` and whose harvest has not been completed. When the slider seen in Figure 14 is hovered over, a message appears which indicates that the toggle will either complete the harvest or mark the harvest as not complete, respectively.

When the arrow in a rectangle icon is clicked for a certain crop in the **Harvest Loads** tile, the user is brought to a page which has a table of all of the harvest trips for that crop.



Harvest Load Summary

Land Division: North Field

Crop Type: Corn | 29 Acres Planted

1,576,248 (lbs) Total Gross Weight

1,036,837 (lbs) Total Net Weight

18,514.95 (bu) Total Amount Harvested

638.45 (bu/acre) Yield Per Acre

19 Total Trips

LOAD ID	GROSS WEIGHT (LBS)	TARE WEIGHT (LBS)	NET WEIGHT (LBS)	NET YIELD (BUSHELS)	TEST WEIGHT (LBS)	MOISTURE (%)	DATE DELIVERED (MM/DD/YY)
1	79,851 lbs	25,867 lbs	53,984 lbs	964 bu	59 lbs	17 %	9/28/24
2	82,095 lbs	28,424 lbs	53,671 lbs	958.41 bu	59 lbs	17 %	9/29/24
3	84,626 lbs	30,285 lbs	54,341 lbs	970.38 bu	60 lbs	16 %	9/29/24
4	83,043 lbs	27,887 lbs	55,156 lbs	984.93 bu	57 lbs	17 %	9/30/24
5	86,749 lbs	26,125 lbs	60,624 lbs	1,082.57 bu	57 lbs	17 %	9/30/24
6	76,803 lbs	29,428 lbs	47,375 lbs	845.98 bu	57 lbs	17 %	9/30/24
7	80,421 lbs	26,795 lbs	53,626 lbs	957.61 bu	59 lbs	16 %	9/30/24
8	76,429 lbs	25,160 lbs	51,269 lbs	915.52 bu	59 lbs	16 %	10/1/24
9	91,162 lbs	30,613 lbs	60,549 lbs	1,081.23 bu	60 lbs	15 %	10/2/24
10	84,793 lbs	25,563 lbs	59,230 lbs	1,057.68 bu	60 lbs	15 %	10/2/24
11	77,998 lbs	27,576 lbs	50,422 lbs	900.39 bu	58 lbs	15 %	10/3/24
12	75,707 lbs	25,617 lbs	50,090 lbs	894.46 bu	58 lbs	15 %	10/4/24

Figure 15. Harvest load summary for corn for the North Field.

At the top of Figure **15**, we can see the information about which `LandDivision` is being viewed, which crop, and the amount of acres of that crop that is planted on the

LandDivision. When one of these rows are selected, the view shown in Figure **15** is presented. The details on that specific trip are shown and are able to be edited.

Returning back to the details page for the North Field shown in Figure **12**, when we continue to scroll down on the page, there are also charts that are related to the performance of the LandDivision for its yield, as well as various metrics about the jobs which are associated with this LandDivision.

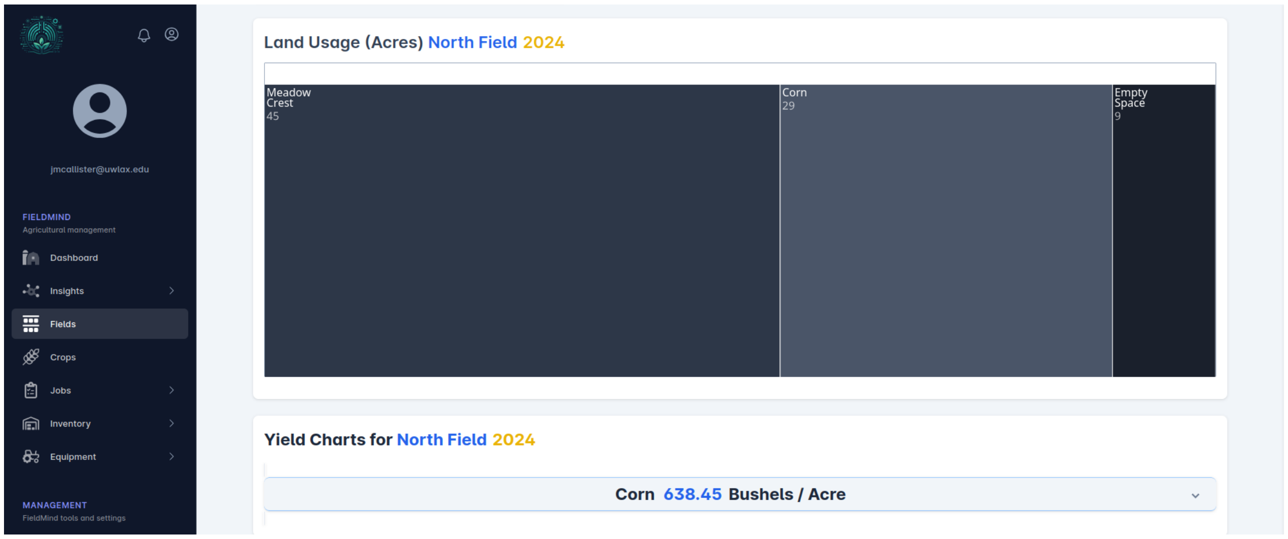


Figure 16. Land usage tree map for the North Field.

First, in Figure **16**, there is a tree map which shows the breakdown of the area usage specifically for the LandDivision which is being viewed. Also in Figure **16**, there is a dropdown menu for corn which will show various charts about the harvest information for that crop. The North Field only has one crop planted in it, but if there were multiple, there would be one dropdown for each crop.

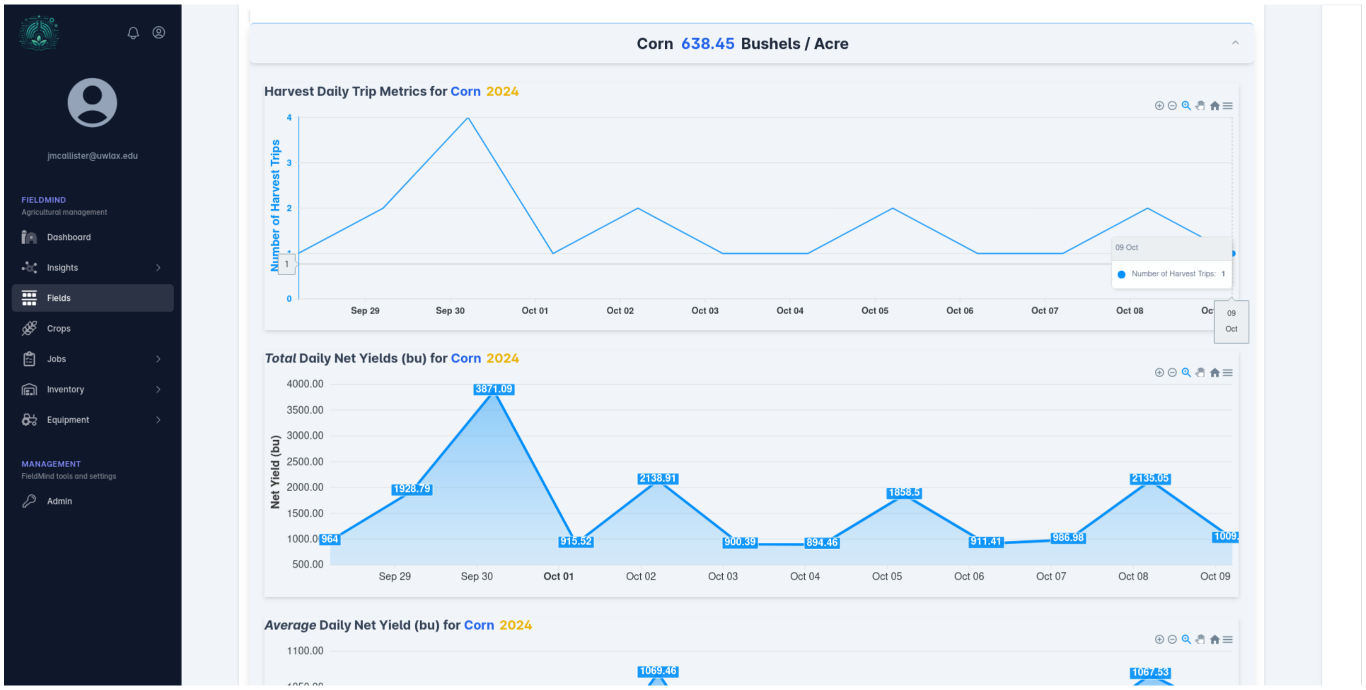

jmcallister@uwlax.edu
FIELDMIND
Agricultural management
Dashboard
Insights
Fields
Crops
Jobs
Inventory
Equipment
MANAGEMENT
FieldMind tools and settings
Admin
Corn 638.45 Bushels / Acre
Harvest Daily Trip Metrics for Corn 2024
Number of Harvest Trips
Total Daily Net Yields (bu) for Corn 2024
Net Yield (bu)
Average Daily Net Yield (bu) for Corn 2024

Figure 17. Yield charts for corn on the North Field.

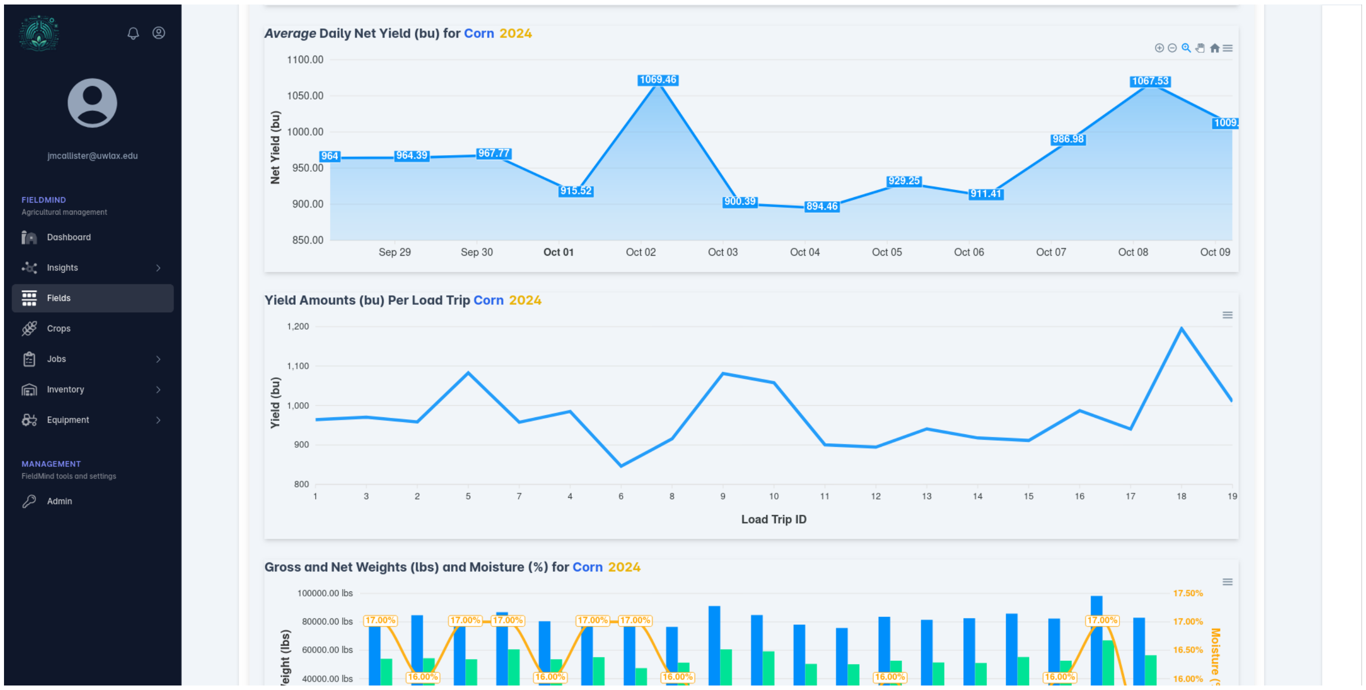

jmcallister@uwlax.edu
FIELDMIND
Agricultural management
Dashboard
Insights
Fields
Crops
Jobs
Inventory
Equipment
MANAGEMENT
FieldMind tools and settings
Admin
Average Daily Net Yield (bu) for Corn 2024
Net Yield (bu)
Yield Amounts (bu) Per Load Trip Corn 2024
Yield (bu)
Load Trip ID
Gross and Net Weights (lbs) and Moisture (%) for Corn 2024

Figure 18. Yield charts for corn on the North Field.

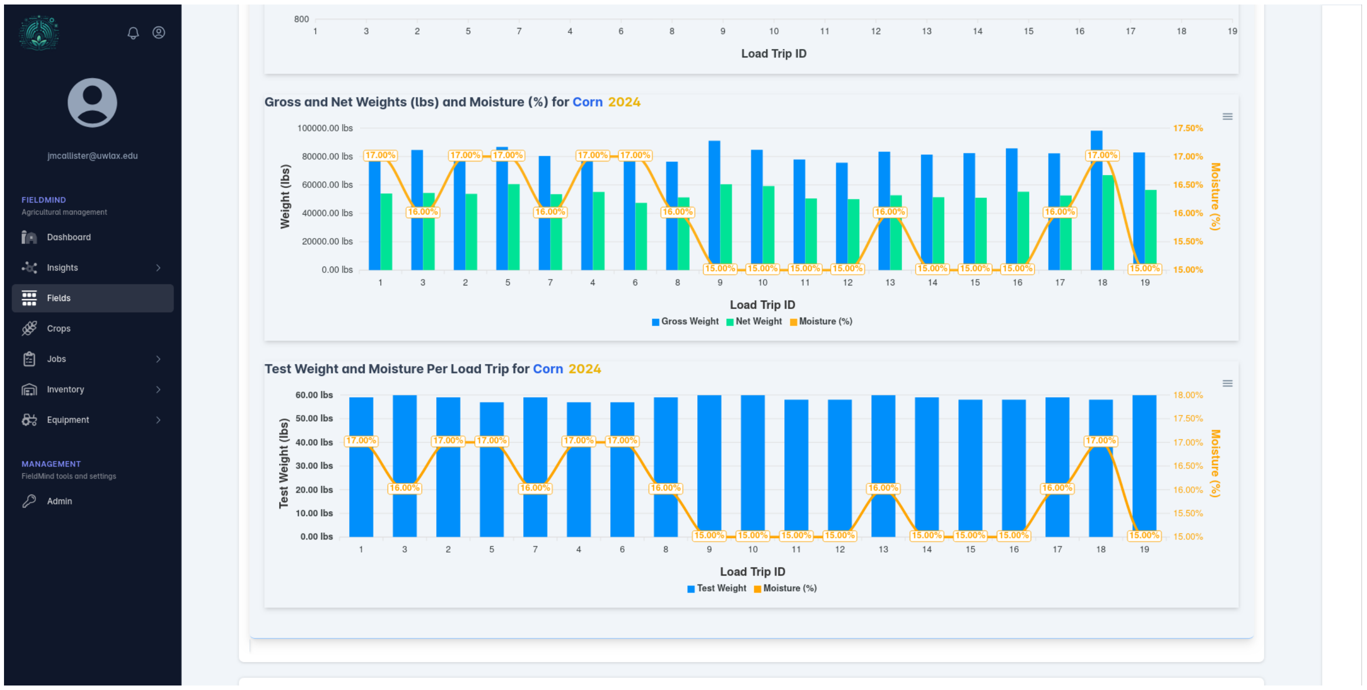


Figure 19. Yield charts for corn on the North Field.

Figures **17**, **18**, and **19** show the yield charts for corn for the North Field.

After these yield charts, there are charts which show information on the jobs and inventory usage on this `LandDivision`.

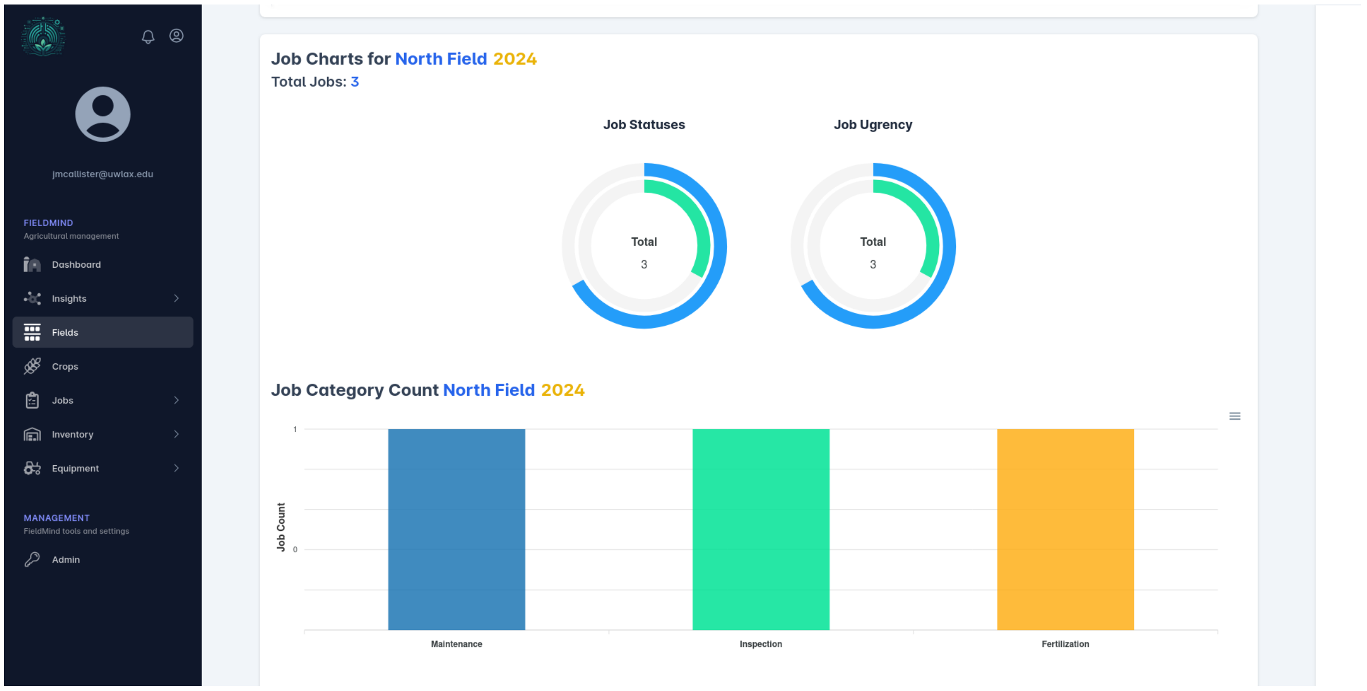


Figure 20. Job charts for the North Field.

By tracking inventory usage at the job level and associating each job with a category, FieldMind enables the visualization of inventory consumption by job category through a dedicated chart. On the individual field page, I have implemented a zoomable sunburst chart that illustrates inventory usage for jobs on that specific field. The chart breaks down usage by job category, then further categorizes it into bulk-tracked or individually tracked inventory, followed by inventory type.

Users can interact with the sunburst chart by clicking on any segment to zoom into a specific category. Hovering over a slice displays the corresponding inventory count, while clicking the center of the sunburst zooms back out. Figures **21** and **22** showcase the sunburst chart at different zoom levels, demonstrating its interactive functionality.

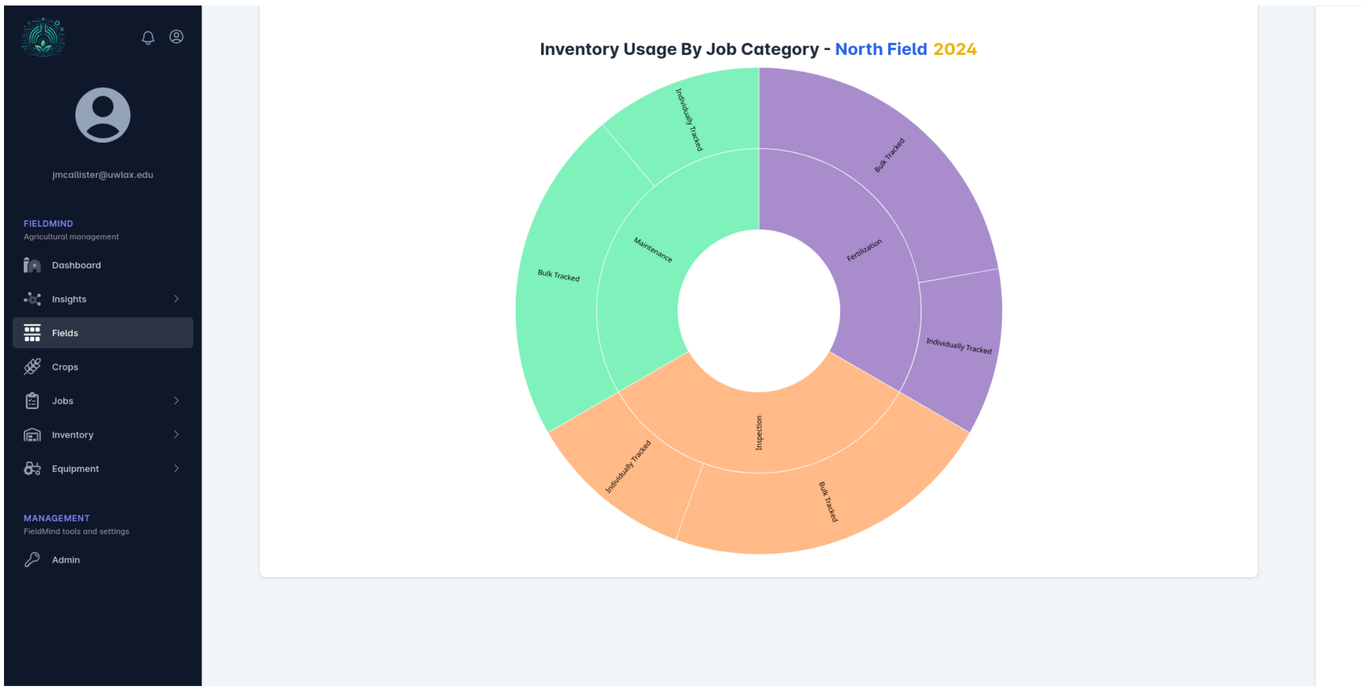


Figure 21. Inventory usage by job category and inventory category for the North Field.

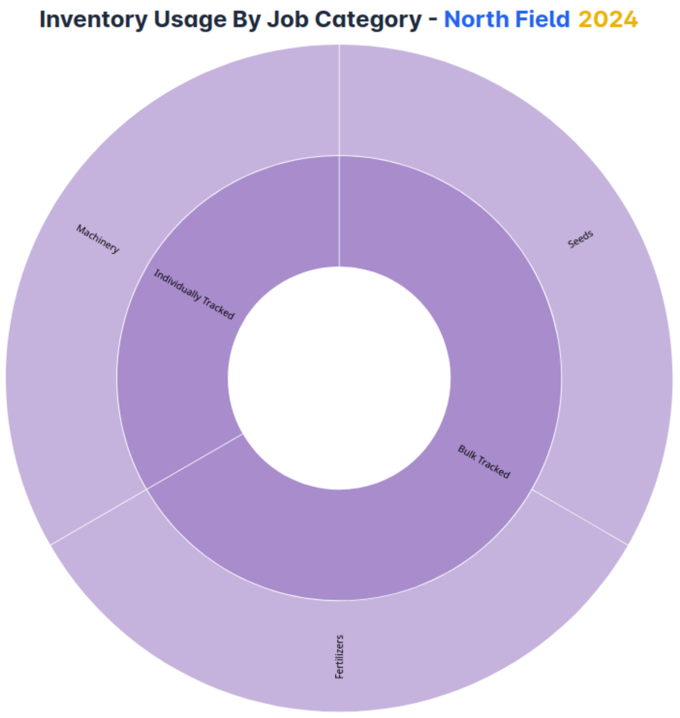
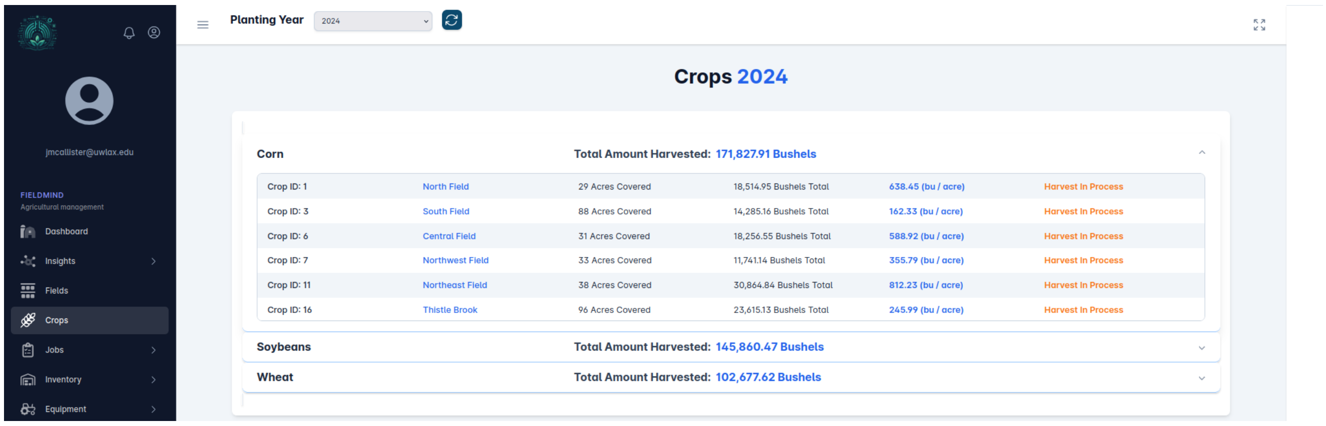


Figure 22. Job inventory usage zoomed in on the "Fertilizers" job category.

4.4. Crops

The crops page shows data from the perspective of the crop across the entire farm for the selected planting year.



Crops 2024

Corn — Total Amount Harvested: 171,827.91 Bushels

Crop ID: 1	North Field	29 Acres Covered	18,514.95 Bushels Total	638.45 (bu / acre)	Harvest In Process
Crop ID: 3	South Field	88 Acres Covered	14,285.16 Bushels Total	162.33 (bu / acre)	Harvest In Process
Crop ID: 6	Central Field	31 Acres Covered	18,256.55 Bushels Total	588.92 (bu / acre)	Harvest In Process
Crop ID: 7	Northwest Field	33 Acres Covered	11,741.14 Bushels Total	355.79 (bu / acre)	Harvest In Process
Crop ID: 11	Northeast Field	38 Acres Covered	30,864.84 Bushels Total	812.23 (bu / acre)	Harvest In Process
Crop ID: 16	Thistle Brook	96 Acres Covered	23,615.13 Bushels Total	245.99 (bu / acre)	Harvest In Process

Soybeans — Total Amount Harvested: 145,860.47 Bushels

Wheat — Total Amount Harvested: 102,677.62 Bushels

Figure 23. Top of the crops landing page.

In Figure **23**, the top of the **Crops** page is shown. In it, we can see that there is one dropdown for each crop type that has been planted on the farm for the selected year. When one of those dropdowns is toggled open, as we can see the "Corn" dropdown is, a scrollable list of all of the corn crops can be seen. For each row, there are various metrics about that crop, such as which `LandDivision` it is planted in, how many acres are planted there, how many total bushels have been harvested and how that comes out to bushels per acre for that `LandDivision`, and whether or not a harvest is in progress. Each row is also clickable, and will take the user to the `LandDivision` where that crop is planted when clicked.

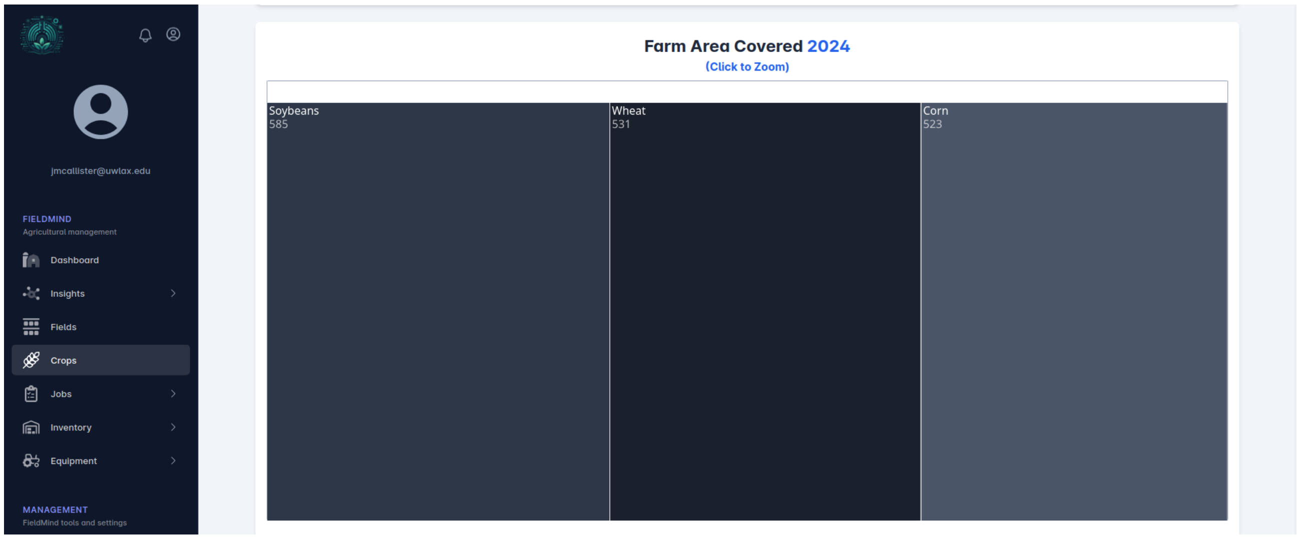


Figure 24. Crops area covered tree map.

Below the aforementioned set of dropdowns, there is a tree map, which we can see in Figure **24**. Rather than show the area of all of the `LandDivisions` as was seen on the **Fields** page, here we see the total area of the farm covered by each of the crops planted on the farm.

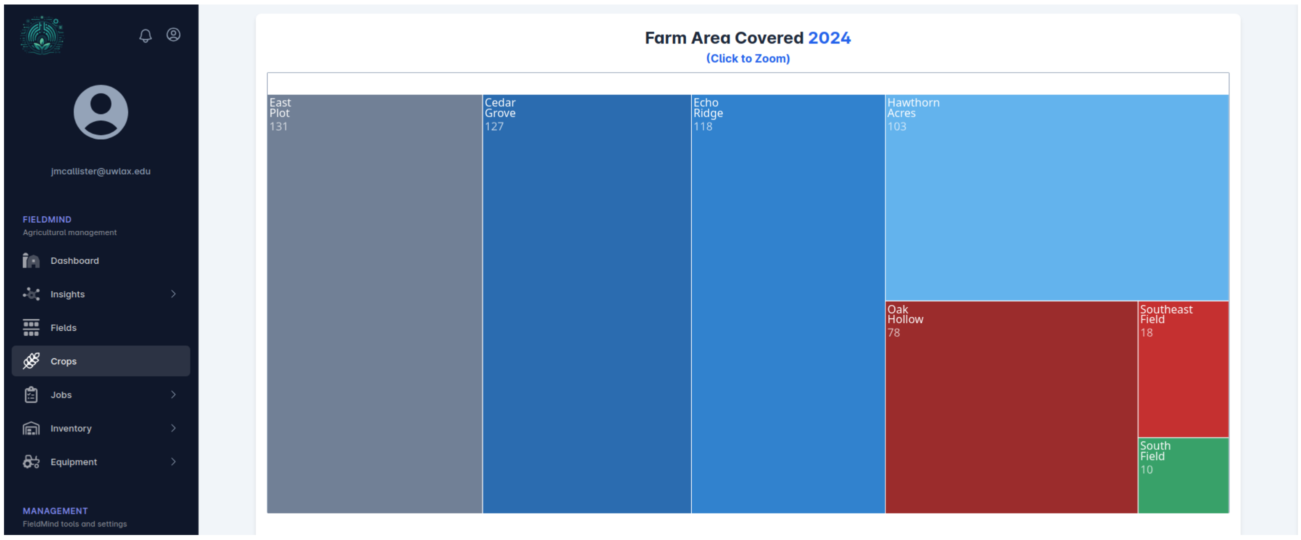


Figure 25. Soybeans selected in the area tree map, showing which `LandDivisions` soybeans are planted in and how many acres are in each `LandDivision`.

When one of the crop tiles is selected, such as the soybeans tile, that part of the tree map zooms in to show all of the `LandDivisions` in which soybeans are planted, and the amount of acres which it takes up on each of those `LandDivisions`.

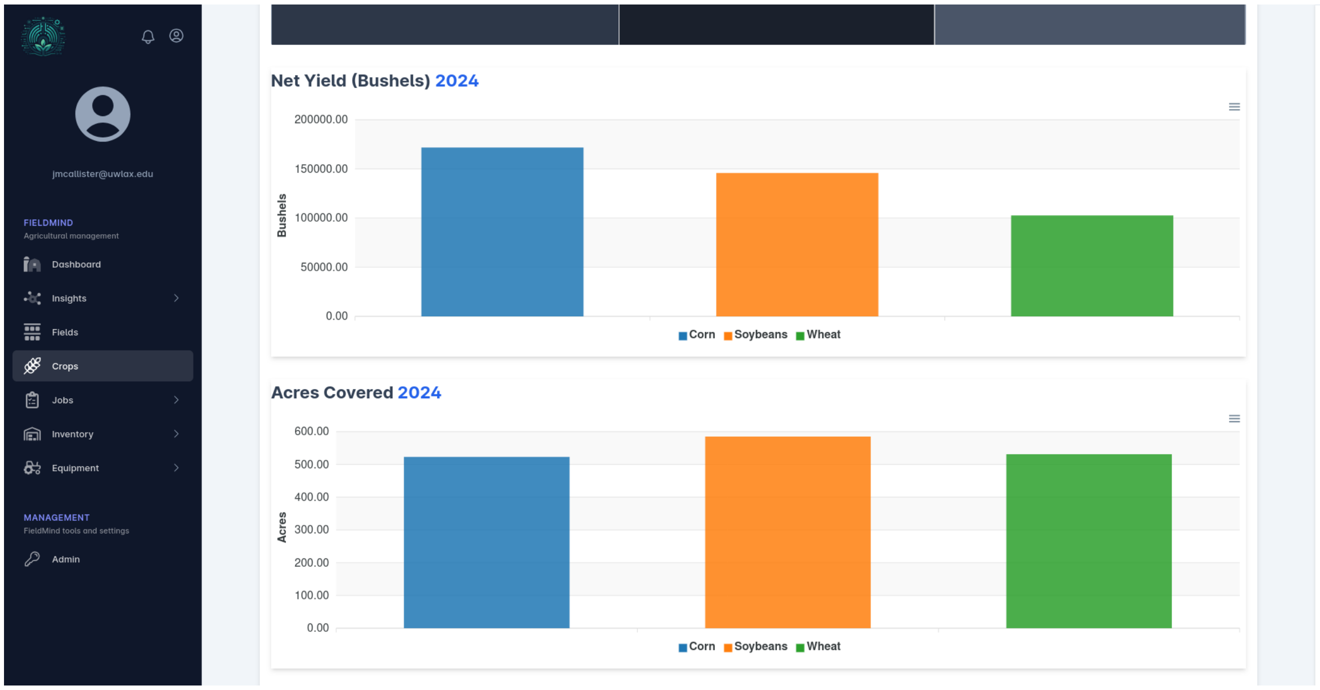
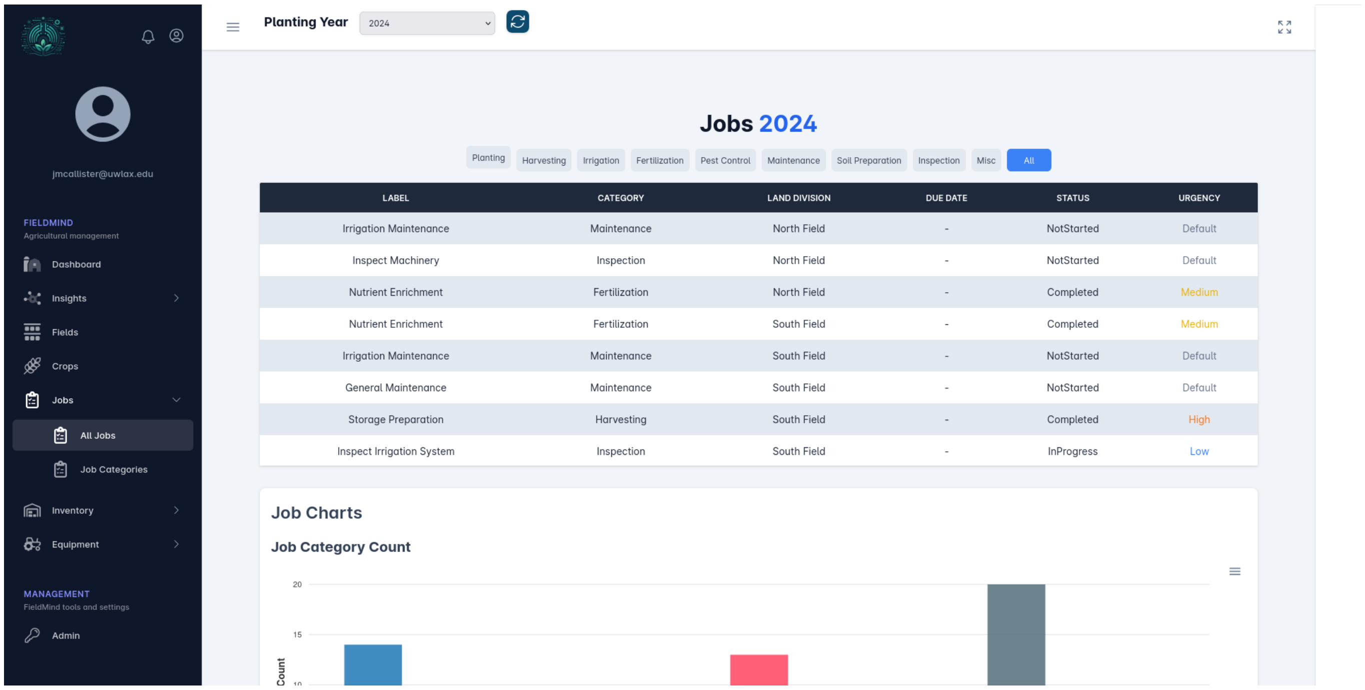


Figure 26. Bar charts showing net yield and acres covered for each of the crop types on the farm.

After the tree map, Figure **26** shows the bar chars which show the net yield and total acres covered, respectively, by each crop type that is planted on the farm.

4.5. Jobs

The **Jobs** page shows charts about various metrics on the jobs throughout the farm for the selected planting year.



LABEL	CATEGORY	LAND DIVISION	DUE DATE	STATUS	URGENCY
Irrigation Maintenance	Maintenance	North Field	-	NotStarted	Default
Inspect Machinery	Inspection	North Field	-	NotStarted	Default
Nutrient Enrichment	Fertilization	North Field	-	Completed	Medium
Nutrient Enrichment	Fertilization	South Field	-	Completed	Medium
Irrigation Maintenance	Maintenance	South Field	-	NotStarted	Default
General Maintenance	Maintenance	South Field	-	NotStarted	Default
Storage Preparation	Harvesting	South Field	-	Completed	High
Inspect Irrigation System	Inspection	South Field	-	InProgress	Low

Figure 27. Top of the Jobs landing page.

Each job is only associated with one planting year. If a job is not completed within a planting year, it does not "roll over" to the next year - instead they are each contained to the one year. Figure **27** shows the table of jobs which is at the top of the **Jobs** landing page. This table lists all of the jobs across the farm for the selected planting year, with each row being clickable to view the details of a given job.

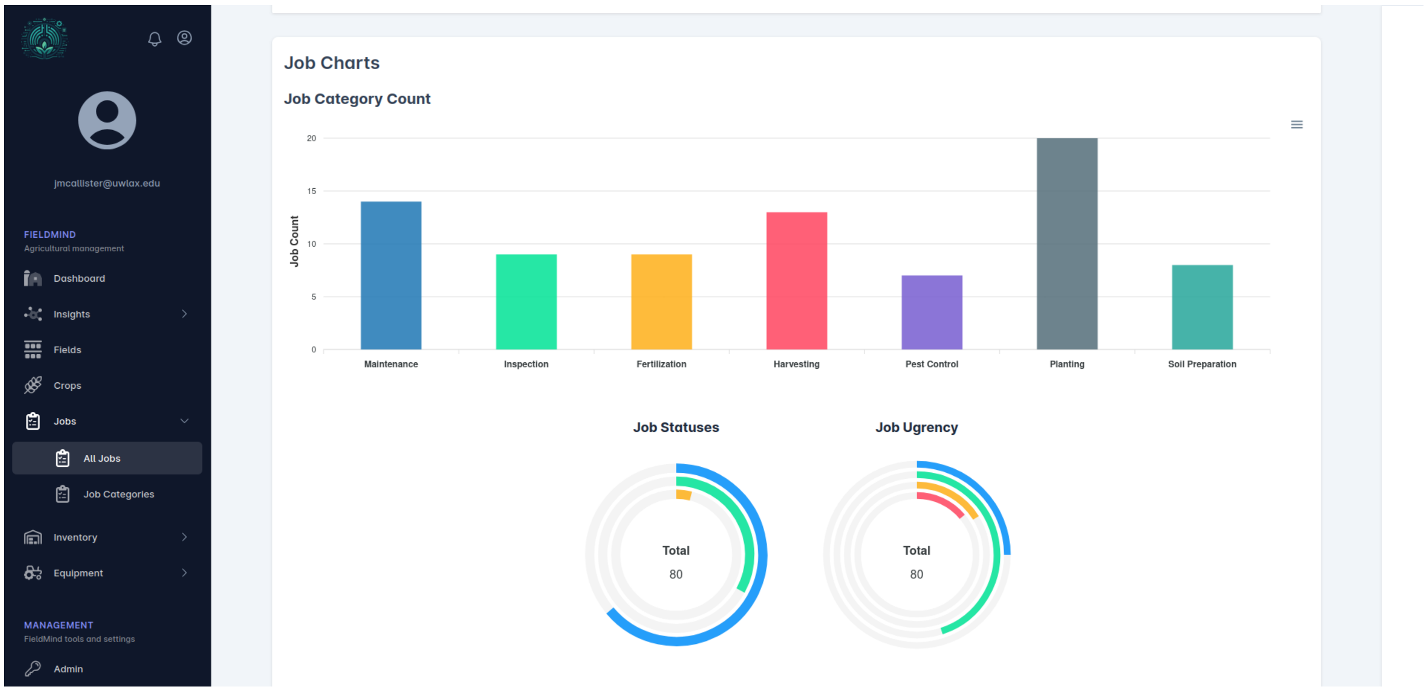


Figure 28. Jobs bar and radial charts.

Scrolling down, there are bar charts which show the count of each category of job across the farm, as well as two radial charts - one representing the count of the various statuses of jobs, and the other the count of the urgency classifications of the jobs.

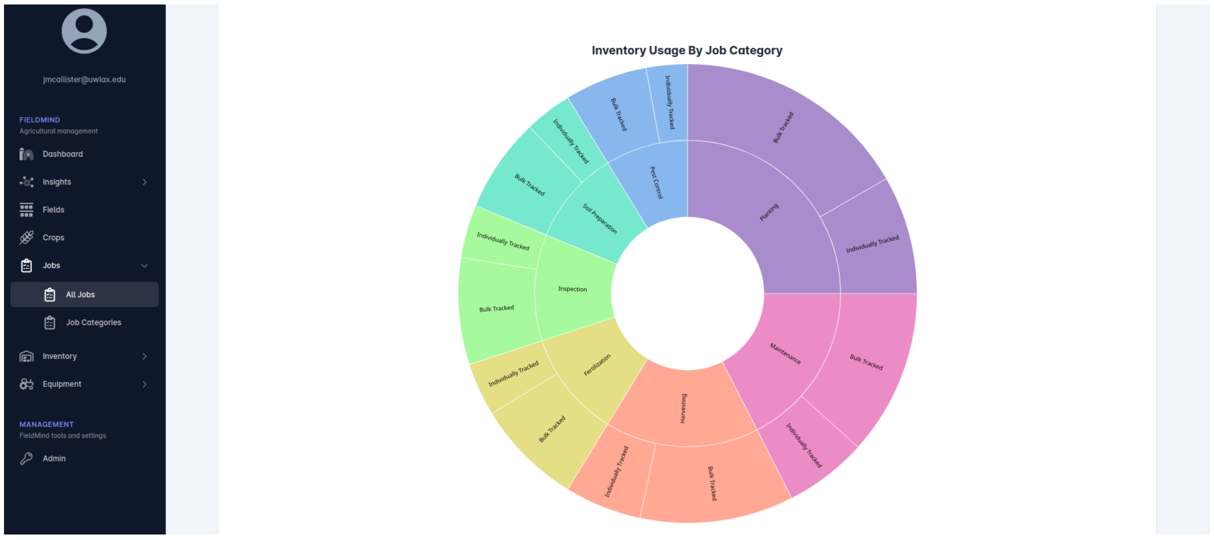


Figure 29. Inventory usage by job category sunburst chart.

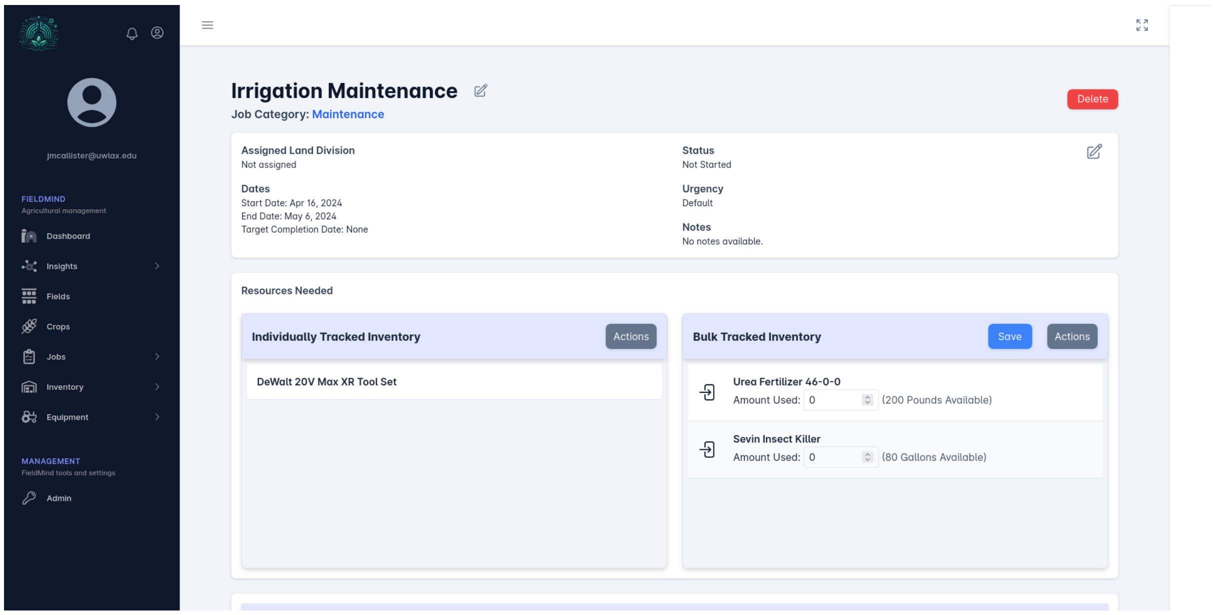
Below these there is a zoomable sunburst chart, shown in Figure 29, which shows inventory usage by job category. When one of the job categories is selected, the chart will zoom into that category to show the proportion of either Bulk Tracked or Individually Tracked inventory there are for that job category.

When either the “Bulk Tracked” or the “Individually Tracked” options are chosen, that slice is zoomed in on and the breakdown of what inventory category of that inventory type is shown.

Below the inventory usage sunburst charts are two other sunburst charts - one allows the user to start by the job category, and then click into a tile to see the status of the jobs in those categories, and then click once more to see that urgency classification of the jobs of a given status. The other sunburst chart shows the inverse once a job category is selected - first the urgency classification and then the status.

4.6. Job Details

When a specific job is selected, whether from the table on the **Jobs** page, or on the job area on the details page for a specific `LandDivision`, we can interact with the details of a specific job.



Irrigation Maintenance
Job Category: Maintenance

Assigned Land Division
Not assigned

Dates
Start Date: Apr 16, 2024
End Date: May 6, 2024
Target Completion Date: None

Status
Not Started

Urgency
Default

Notes
No notes available.

Resources Needed

Individually Tracked Inventory
DeWalt 20V Max XR Tool Set

Bulk Tracked Inventory
Urea Fertilizer 46-0-0
Amount Used: 0 (200 Pounds Available)
Sevin Insect Killer
Amount Used: 0 (80 Gallons Available)

Figure 30. Top of job details page.

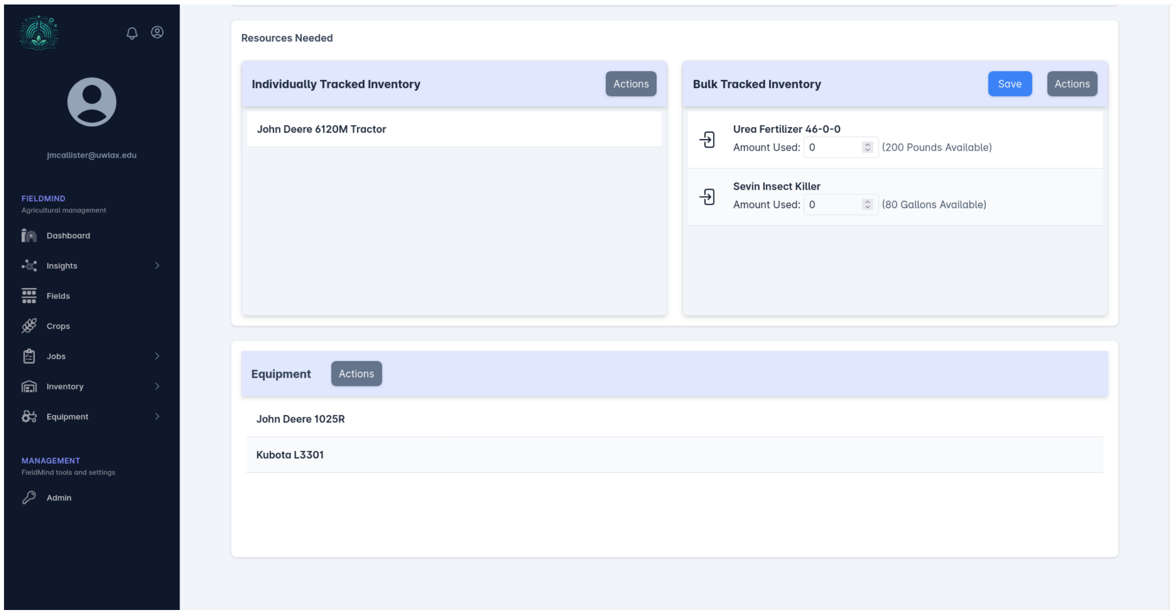
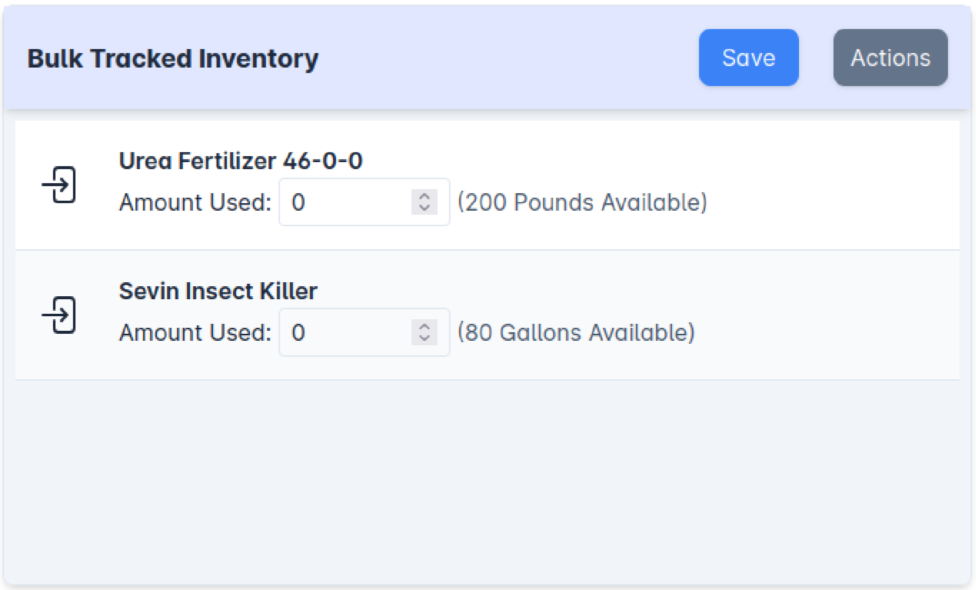


Figure 31. Inventory and equipment sections of the job details page.

Figure **30** shows the top of the job details page for the "Irrigation Maintenance" job. The top section allows the user to see various metadata on the job, and also to edit it if the pencil icon is clicked. We can also see that there is both individually tracked (`IndividuallyTrackedItem` entity) and bulk tracked ('BulkTrackedItem' entity) inventory associated with this job. Figure **31** shows the equipment associated with this job as well.

The job allows the user to select whichever `IndividuallyTrackedItem`(s) the job needs, as well as whichever `BulkTrackedItem`(s) the job needs. Once a `BulkTrackedItem` is added to the job, the user is then allowed to adjust how much of that item was used. We can see this in Figure **32**.

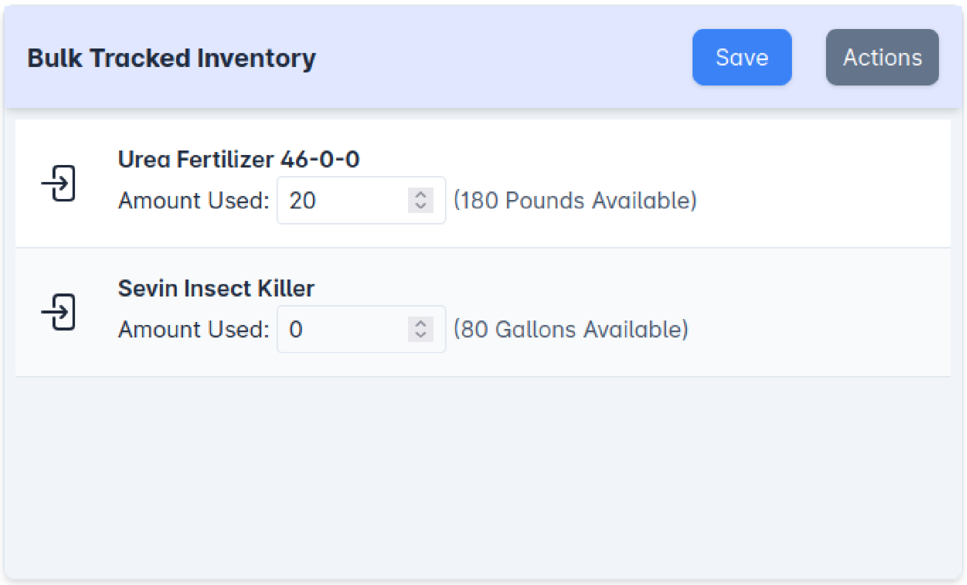


Urea Fertilizer 46-0-0
Amount Used: 0 (200 Pounds Available)

Sevin Insect Killer
Amount Used: 0 (80 Gallons Available)

Figure 32. Bulk tracked inventory unused.

Once the item has been used for the job and the user adjusts the amount, the available stock for that item goes down. we can see this in Figure **34**.

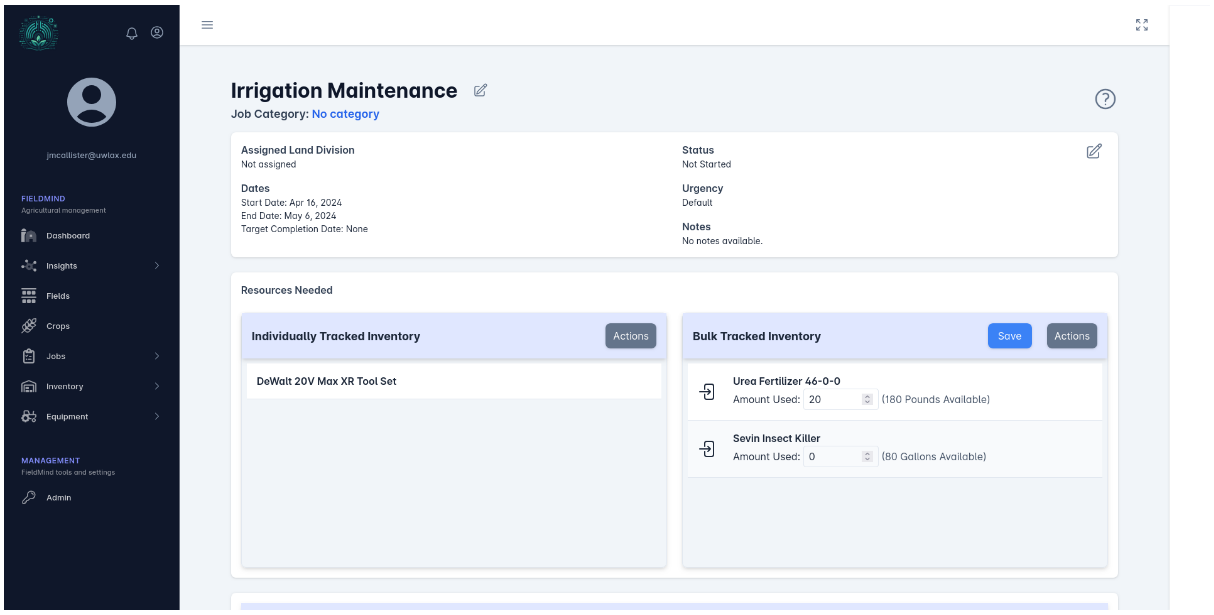


Urea Fertilizer 46-0-0
Amount Used: 20 (180 Pounds Available)

Sevin Insect Killer
Amount Used: 0 (80 Gallons Available)

Figure 33. Bulk tracked inventory used.

When a job has drawn down inventory, that job also cannot be deleted. Figure **34** shows that the "Delete" button which was previously in the top-right corner in Figure **30** before inventory had been drawn down is now a question mark icon, which is clickable.



Irrigation Maintenance

Job Category: No category

Assigned Land Division
Not assigned

Dates
Start Date: Apr 16, 2024
End Date: May 6, 2024
Target Completion Date: None

Status
Not Started

Urgency
Default

Notes
No notes available.

Resources Needed

Individually Tracked Inventory — Actions

DeWalt 20V Max XR Tool Set

Bulk Tracked Inventory — Save — Actions

Urea Fertilizer 46-0-0
Amount Used: 20 (180 Pounds Available)

Sevin Insect Killer
Amount Used: 0 (80 Gallons Available)

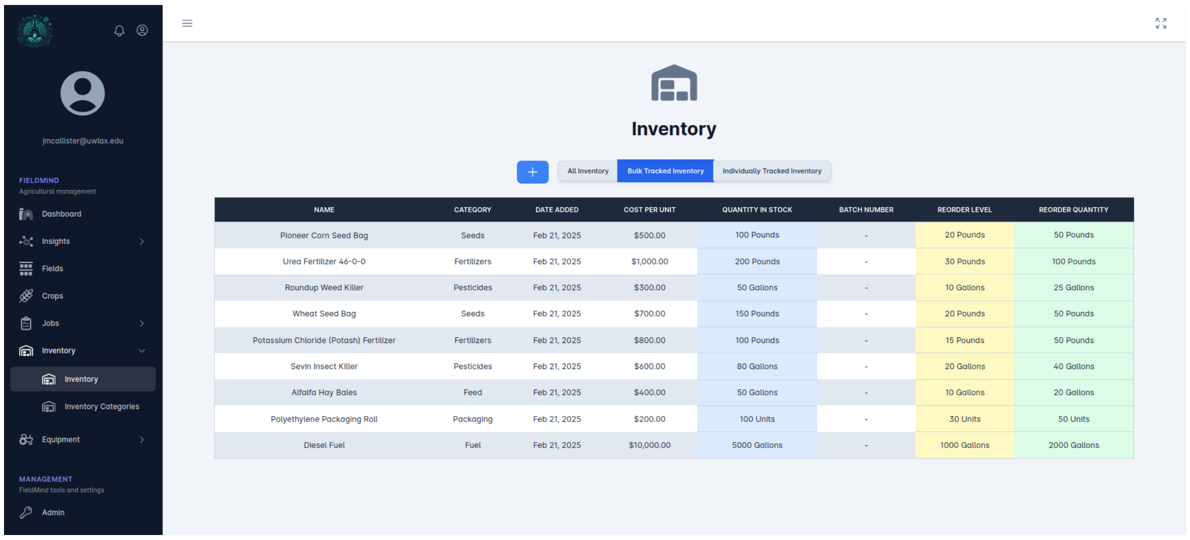
Figure 34. Bulk inventory has been drawn down, and thus the "Delete" button in the top-right corner of the job has changed to a clickable question mark icon.

When the question mark icon is clicked on, we are shown a dialog window which indicates the reason that the job cannot be deleted. Additionally, a user can only remove a `BulkTrackedItem` from a job if that item has not drawn down inventory. Otherwise, if a `BulkTrackedItem` which had drawn down inventory was allowed to be deleted from a job, there would be no record of where that inventory went. When the "Actions" button is clicked and the "Remove Item" option is selected, if that inventory item has drawn down inventory, instead of a red "x" icon being displayed, a question mark which is clickable is displayed. When that question mark icon is clicked on, a dialog box pops open giving the reason for not being able to remove that inventory item.

In this way, we have a basic inventory tracking system. I opted for this approach with deletion rather than just zero-ing out the inventory usages because it makes the decision to remove it more intentional by the user. There may be some alternative approaches to handle this situation, but this is the approach FieldMind has taken for this iteration.

4.7. Inventory

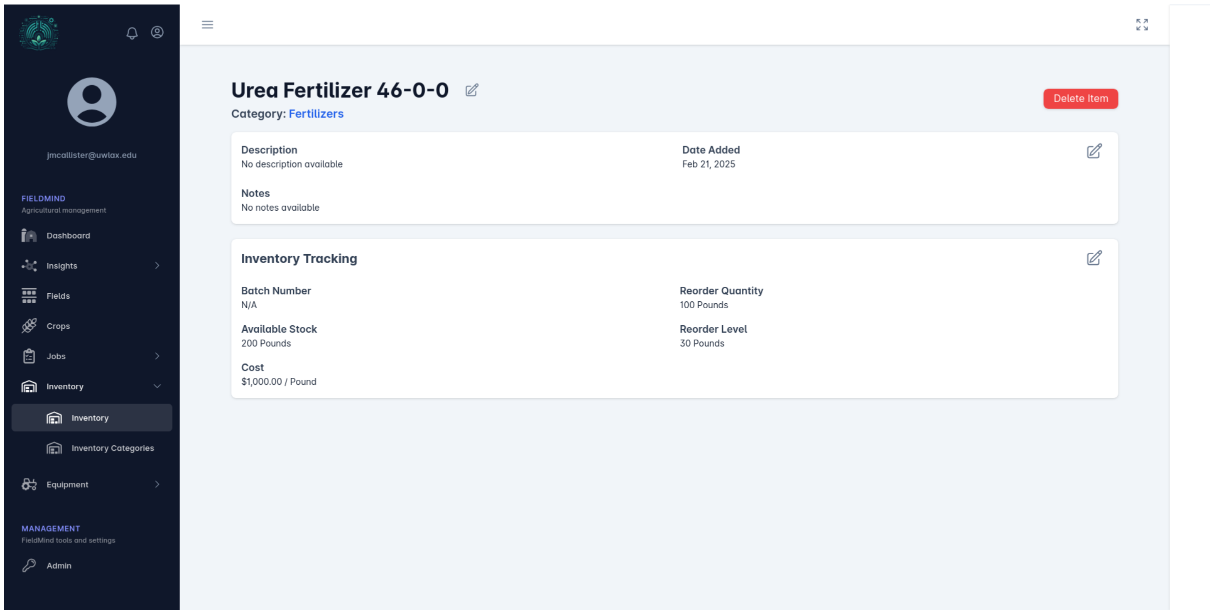
The inventory is divided into two broad data types: individually tracked inventory and bulk tracked inventory, which are backed by the `IndividuallyTrackedItem` and `BulkTrackedItem` entities respectively, each implementing an `IInventory` interface. Furthermore, every inventory item is also associated with an `InventoryCategory` entity as well.



NAME	CATEGORY	DATE ADDED	COST PER UNIT	QUANTITY IN STOCK	BATCH NUMBER	REORDER LEVEL	REORDER QUANTITY
Pioneer Corn Seed Bag	Seeds	Feb 21, 2025	$500.00	100 Pounds	-	20 Pounds	50 Pounds
Urea Fertilizer 46-0-0	Fertilizers	Feb 21, 2025	$1,000.00	200 Pounds	-	30 Pounds	100 Pounds
Roundup Weed Killer	Pesticides	Feb 21, 2025	$300.00	50 Gallons	-	10 Gallons	25 Gallons
Wheat Seed Bag	Seeds	Feb 21, 2025	$700.00	150 Pounds	-	20 Pounds	50 Pounds
Potassium Chloride (Potash) Fertilizer	Fertilizers	Feb 21, 2025	$800.00	100 Pounds	-	15 Pounds	50 Pounds
Sevin Insect Killer	Pesticides	Feb 21, 2025	$600.00	80 Gallons	-	20 Gallons	40 Gallons
Alfalfa Hay Bales	Feed	Feb 21, 2025	$400.00	50 Gallons	-	10 Gallons	20 Gallons
Polyethylene Packaging Roll	Packaging	Feb 21, 2025	$200.00	100 Units	-	30 Units	50 Units
Diesel Fuel	Fuel	Feb 21, 2025	$10,000.00	5000 Gallons	-	1000 Gallons	2000 Gallons

Figure 35. Inventory table with the "Bulk Tracked Inventory" option selected.

In Figure **35**, the inventory page is shown with the table of all of the inventory across the farm. There are three buttons at the top which allow the user to look through either all inventory, bulk tracked, or individually tracked. When either the "Bulk Tracked Inventory" or "Individually Tracked Inventory" are selected, there is a blue "+" button which appears which allows a user to add either bulk tracked or individually tracked inventory to the farm. Based on which option was selected when the "add" button was clicked, the appropriate form will be displayed for the inventory item to add.



Urea Fertilizer 46-0-0

Category: Fertilizers

Description
No description available

Date Added
Feb 21, 2025

Notes
No notes available

Inventory Tracking

Batch Number
N/A

Reorder Quantity
100 Pounds

Available Stock
200 Pounds

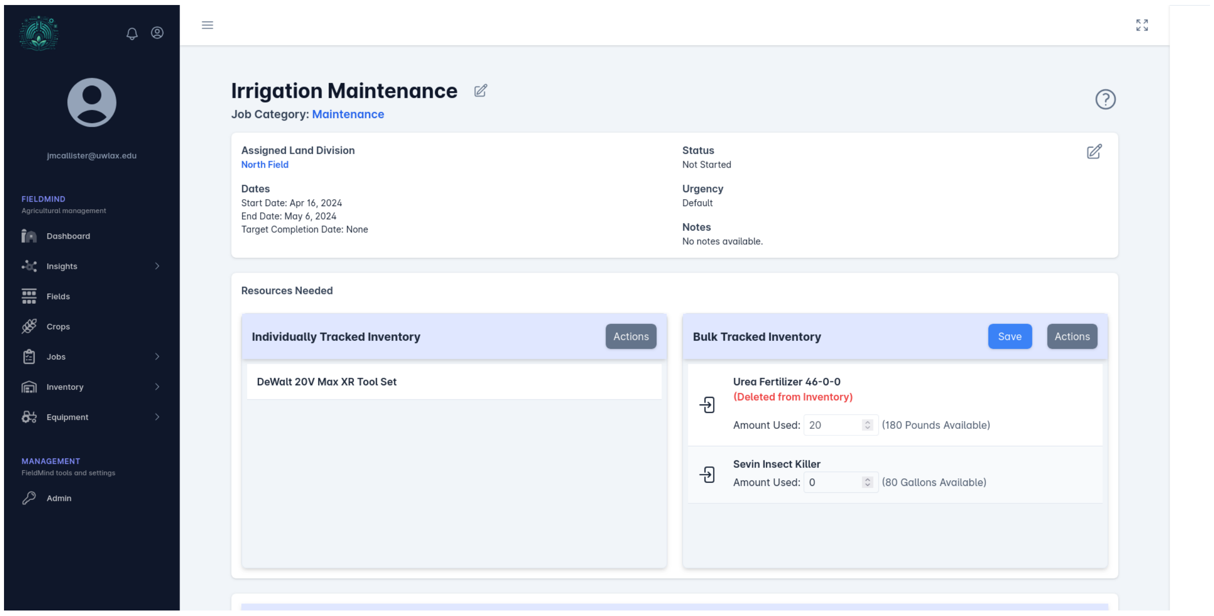
Reorder Level
30 Pounds

Cost
$1,000.00 / Pound

Figure 36. Bulk tracked inventory details page.

Any row in the inventory table can be clicked on, and the details for that inventory item will be brought up. This details page is also what will be shown if an inventory item is selected on a job as well. On this details page, various metadata can be viewed and manipulated, including the deletion of the inventory item.

If a user chooses to delete an inventory item, a dialog box will appear to confirm this choice and inform the user of the consequences of this action. As Figure **37** shows, when an inventory item is deleted, all jobs which have that item on the job will no longer be able to edit the quantity used for that item.



Irrigation Maintenance
Job Category: Maintenance

Assigned Land Division
North Field

Dates
Start Date: Apr 16, 2024
End Date: May 6, 2024
Target Completion Date: None

Status
Not Started

Urgency
Default

Notes
No notes available.

Resources Needed

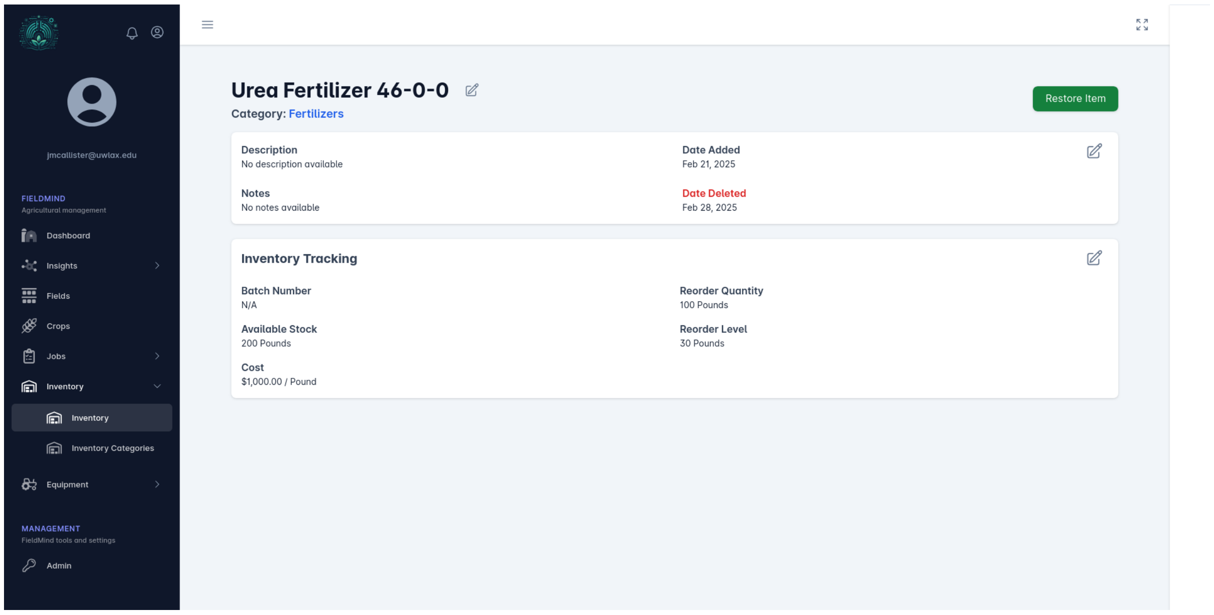
Individually Tracked Inventory
DeWalt 20V Max XR Tool Set

Bulk Tracked Inventory
Urea Fertilizer 46-0-0
(Deleted from Inventory)
Amount Used: 20 (180 Pounds Available)
Sevin Insect Killer
Amount Used: 0 (80 Gallons Available)

Figure 37. Job details in which the job has used a bulk tracked inventory item which has subsequently been deleted from inventory.

The same conditions apply regarding the deletion of an inventory item from a job – if no inventory has been drawn down, the item may be removed; however, if inventory has been deducted, deletion is not permitted, as it would compromise the integrity of the inventory tracking system. In the case where the job has drawn down inventory, as is the case with Figure **37**, the inventory item will indicate that it has been deleted from inventory, and the user will be prevented from changing the amount used for that item.

When adding new items any item which has been deleted will no longer show up as an option to add to the job. Deletions for inventory are currently implemented as a soft delete. Therefore, after an inventory item is deleted, it gives the option to restore it, as shown in Figure **38**.



Urea Fertilizer 46-0-0

Category: Fertilizers

Description
No description available

Date Added
Feb 21, 2025

Notes
No notes available

Date Deleted
Feb 28, 2025

Inventory Tracking

Batch Number
N/A

Reorder Quantity
100 Pounds

Available Stock
200 Pounds

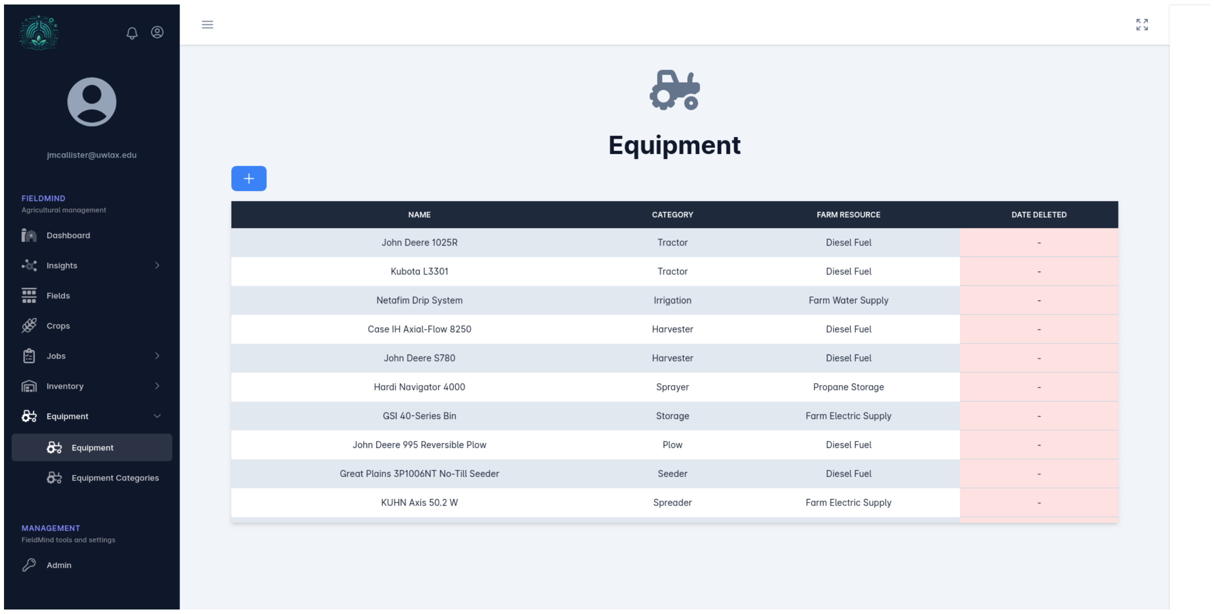
Reorder Level
30 Pounds

Cost
$1,000.00 / Pound

Figure 38. Bulk tracked inventory details page with restore option.

4.8. Equipment

FieldMind also tracks equipment that is on a farm. When the “Equipment” option is selected from the navigation menu, the table shown in Figure **39** is displayed.



NAME	CATEGORY	FARM RESOURCE	DATE DELETED
John Deere 1025R	Tractor	Diesel Fuel	-
Kubota L3301	Tractor	Diesel Fuel	-
Netafim Drip System	Irrigation	Farm Water Supply	-
Case IH Axial-Flow 8250	Harvester	Diesel Fuel	-
John Deere S780	Harvester	Diesel Fuel	-
Hardi Navigator 4000	Sprayer	Propane Storage	-
GSI 40-Series Bin	Storage	Farm Electric Supply	-
John Deere 995 Reversible Plow	Plow	Diesel Fuel	-
Great Plains 3P1006NT No-Till Seeder	Seeder	Diesel Fuel	-
KUHN Axis 50.2 W	Spreader	Farm Electric Supply	-

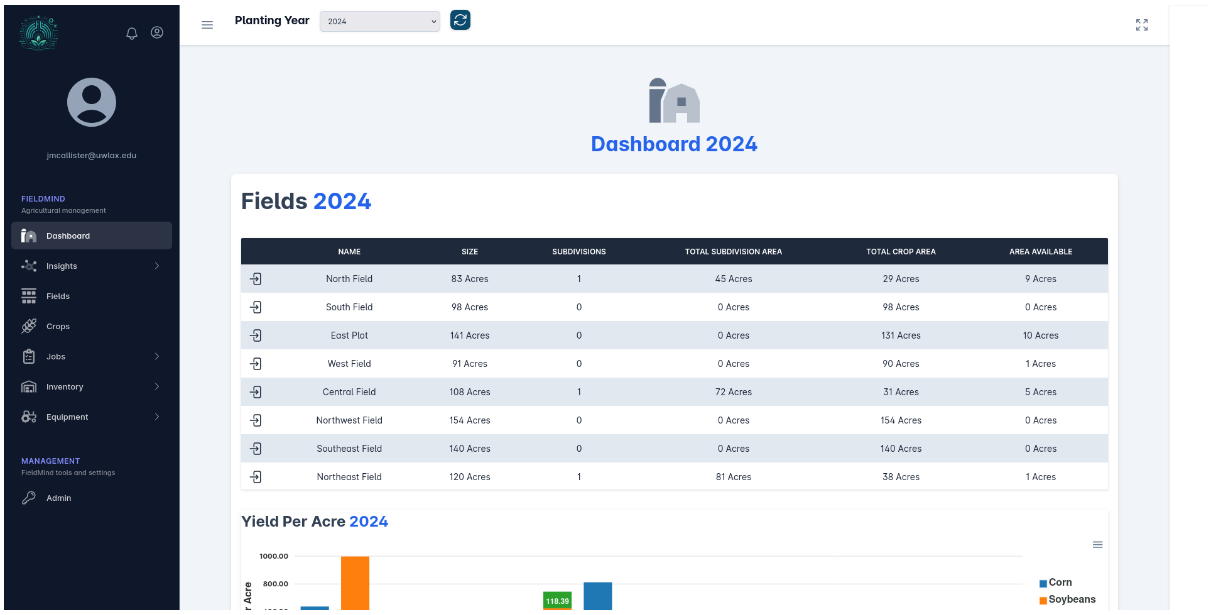
Figure 39. Table displaying all of the equipment on the farm.

When a row is clicked on from a table, the details page for that piece of equipment is displayed. Currently, the main distinction between `Equipment` and an `IndividuallyTrackedItem` would be that `Equipment` can have a `FarmResource` associated with it. This resource would be whatever it needs in order to operate, such as diesel fuel.

As with the inventory, equipment can also be deleted. When equipment is deleted, any jobs currently using that equipment will have that equipment listed as deleted from the farm, just as with the inventory. Unlike with inventory though, since nothing is being tracked by FieldMind to be depleted, the equipment can always be deleted from the job. Rather than doing this automatically, I decided to have that be something that the user does for this implementation. This is largely due to the soft deletion which is performed on equipment, just as with inventory; and, so if equipment is restored, the state of jobs which were using that equipment is maintained.

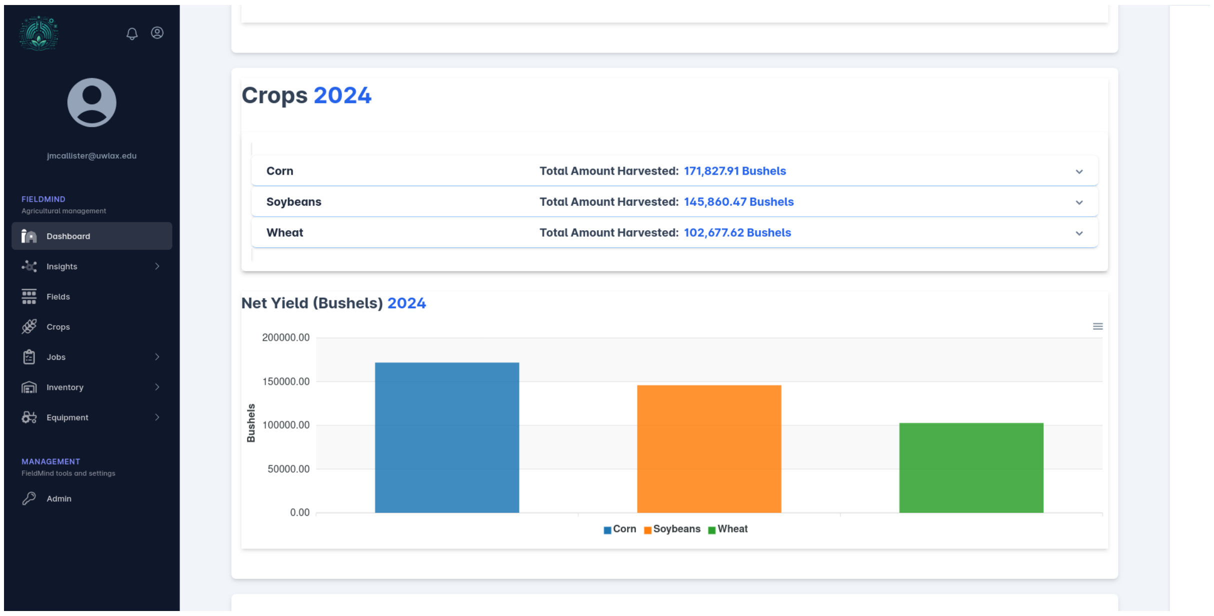
4.9. Dashboard

The “Dashboard” page contains some of the important charts from each of the “Fields”, “Crops”, and “Jobs” pages. There is a section for each of these on the dashboard to house their charts.



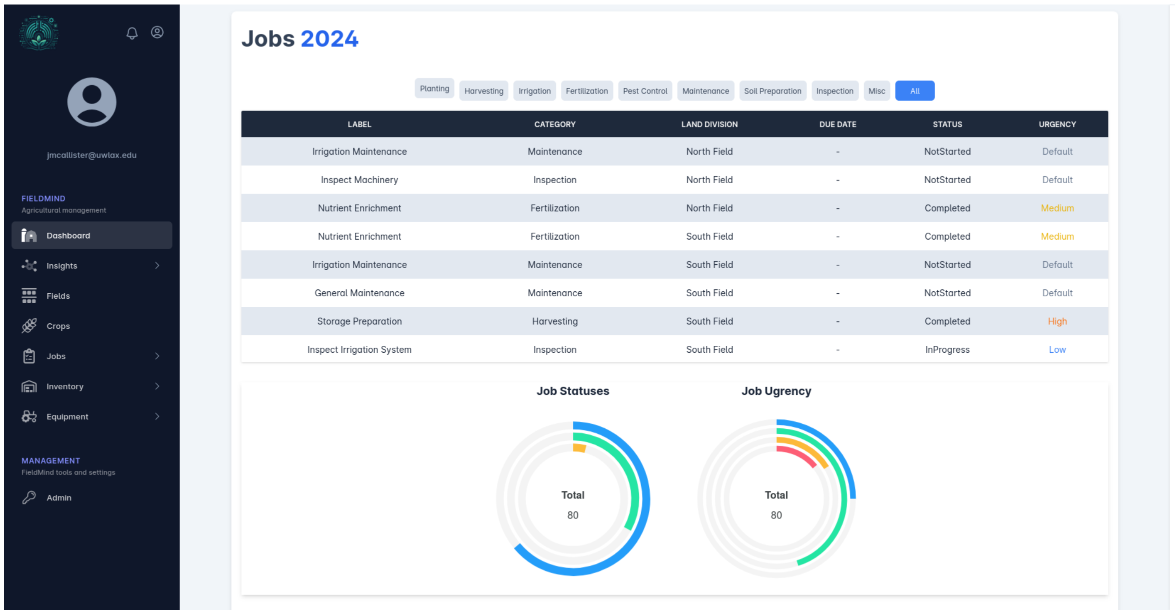
NAME	SIZE	SUBDIVISIONS	TOTAL SUBDIVISION AREA	TOTAL CROP AREA	AREA AVAILABLE
North Field	83 Acres	1	45 Acres	29 Acres	9 Acres
South Field	98 Acres	0	0 Acres	98 Acres	0 Acres
East Plot	141 Acres	0	0 Acres	131 Acres	10 Acres
West Field	91 Acres	0	0 Acres	90 Acres	1 Acres
Central Field	108 Acres	1	72 Acres	31 Acres	5 Acres
Northwest Field	154 Acres	0	0 Acres	154 Acres	0 Acres
Southeast Field	140 Acres	0	0 Acres	140 Acres	0 Acres
Northeast Field	120 Acres	1	81 Acres	38 Acres	1 Acres

Figure 40. Fields portion of Dashboard page.



Corn Total Amount Harvested: 171,827.91 Bushels
Soybeans Total Amount Harvested: 145,860.47 Bushels
Wheat Total Amount Harvested: 102,677.62 Bushels

Figure 41. Crops portion of Dashboard page.



Jobs 2024

LABEL	CATEGORY	LAND DIVISION	DUE DATE	STATUS	URGENCY
Irrigation Maintenance	Maintenance	North Field	-	NotStarted	Default
Inspect Machinery	Inspection	North Field	-	NotStarted	Default
Nutrient Enrichment	Fertilization	North Field	-	Completed	Medium
Nutrient Enrichment	Fertilization	South Field	-	Completed	Medium
Irrigation Maintenance	Maintenance	South Field	-	NotStarted	Default
General Maintenance	Maintenance	South Field	-	NotStarted	Default
Storage Preparation	Harvesting	South Field	-	Completed	High
Inspect Irrigation System	Inspection	South Field	-	InProgress	Low

Figure 42. Jobs portion of Dashboard page.

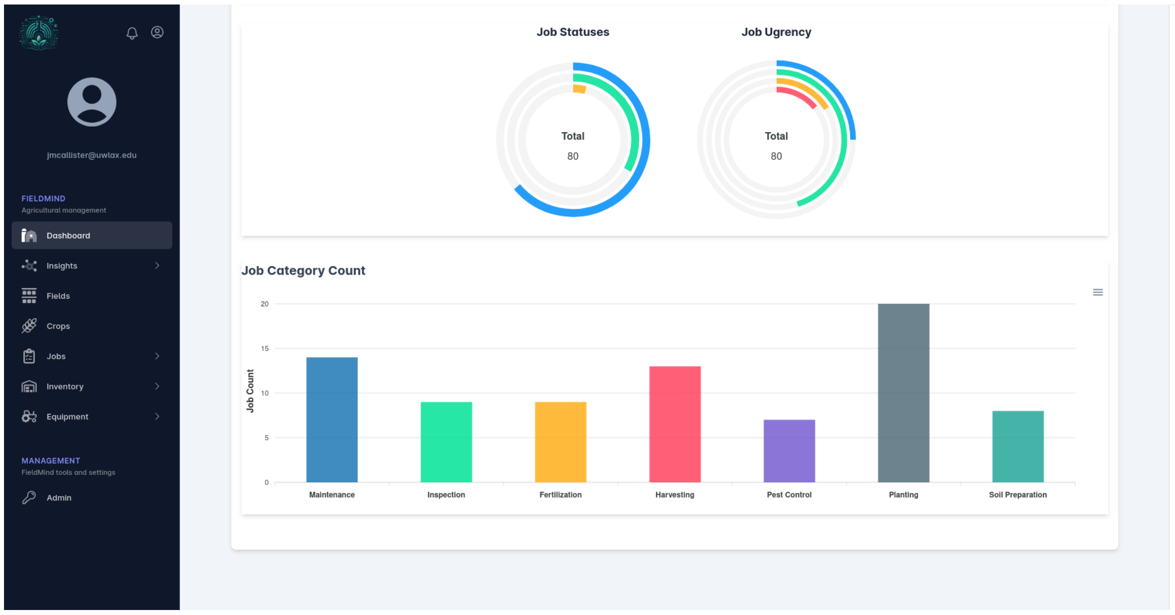


Figure 43. Jobs portion of Dashboard page.

Like the “Fields”, “Crops”, and “Jobs” pages, a specific planting year can be selected on the “Dashboard” page. The dashboard gives a high-level overview of the farm for a planting year across the major entities it tracks. In future iterations, I will add additional charts and options to this page that are unique which specifically combines data from multiple entities across the farm. In this way it can be part of a hub for how things are connected.

4.10. Insights

Like the prior portions of FieldMind, the “Insights” also focuses on the three major entities of “Fields”, “Crops”, and “Jobs”. Figure 44 shows that each of these can be selected through the “Insights” dropdown in the navigation.

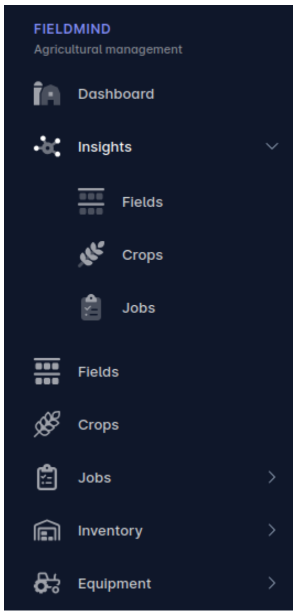


Figure 44. Insights option toggled open in the navigation menu.

The unique perspective of the options in the “Insights” portion of FieldMind is that, rather than just looking at one planting year, each of the insights pages allows the user to select a year **range**. Charts are then generated to look at trends over time for each of the entities.

Currently, FieldMind presents insights for each entity at the farm-wide level, comparing all `LandDivisions` across the entire farm over the selected time range, rather than focusing on individual entities over time. For example, in the “Fields” insights section, trends are analyzed by comparing multiple `LandDivisions` rather than tracking changes within a single `LandDivision`. In future iterations, I plan to introduce the ability to select a specific `LandDivision` and generate charts that display trends exclusively for that entity over the chosen year range. Additionally, I aim to expand the ”Insights” section by incorporating a broader range of charts for each entity to provide deeper analytical capabilities.

4.10.1. Fields

Due to crop rotation, which is where farmers will seasonally change what is planted in a field,it is important to have a distinction when viewing bushels per acre for `LandDivisions` and to only show data for `LandDivisions` for a given year in which it had that crop planted in it. In Figure X, it can be noticed that both the bar chart and the candle stick chart are showing harvest data for all `LandDivisions` based on the crop type which has been selected from the top bar.

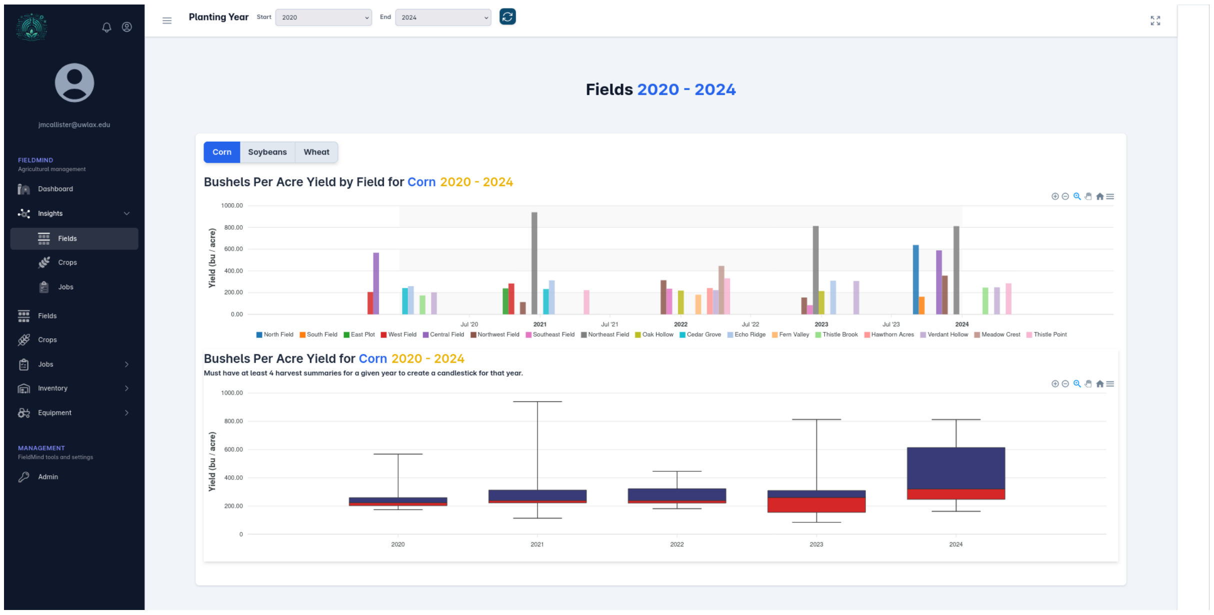


Figure 45. Charts for Land Divisions generated from data over the years of 2020 through 2024.

The top chart is a bar chart showing the yield in bushels per acre for each `LandDivision` which had the selected crop type planted in that year. The bottom chart is a candle stick chart which visualizes a five-number summary for each year based on the yield by each of the `LandDivisions` planted that year. When the user hovers over one of the candle sticks, a tooltip is shown which shows the information for the five-number summary for that year. This can be seen in Figure **46**.

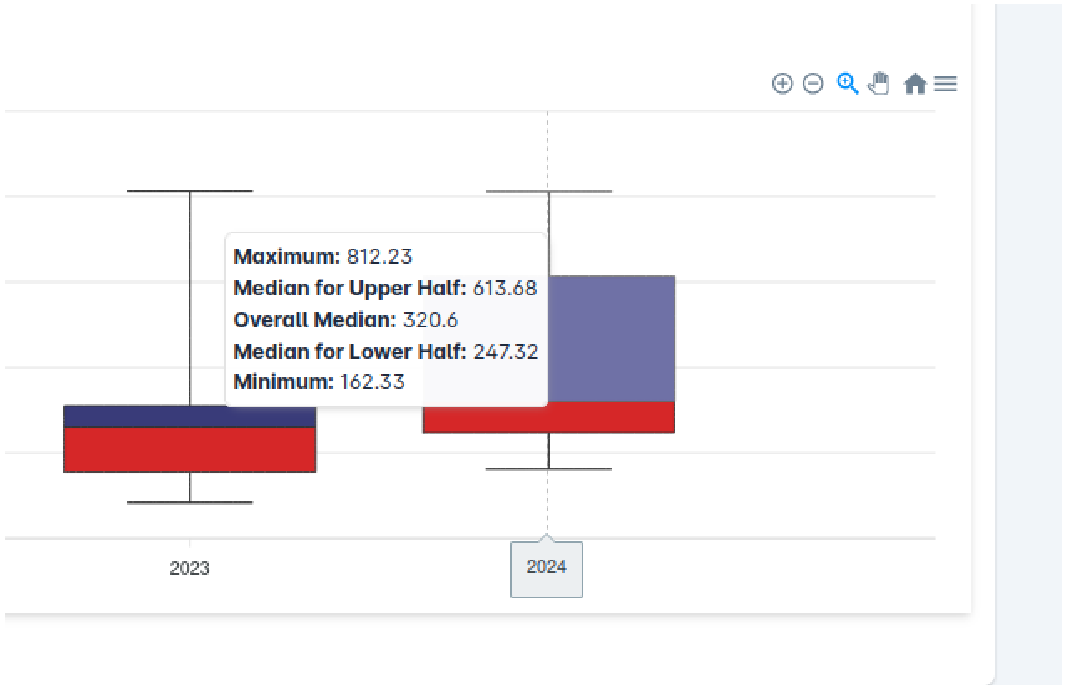
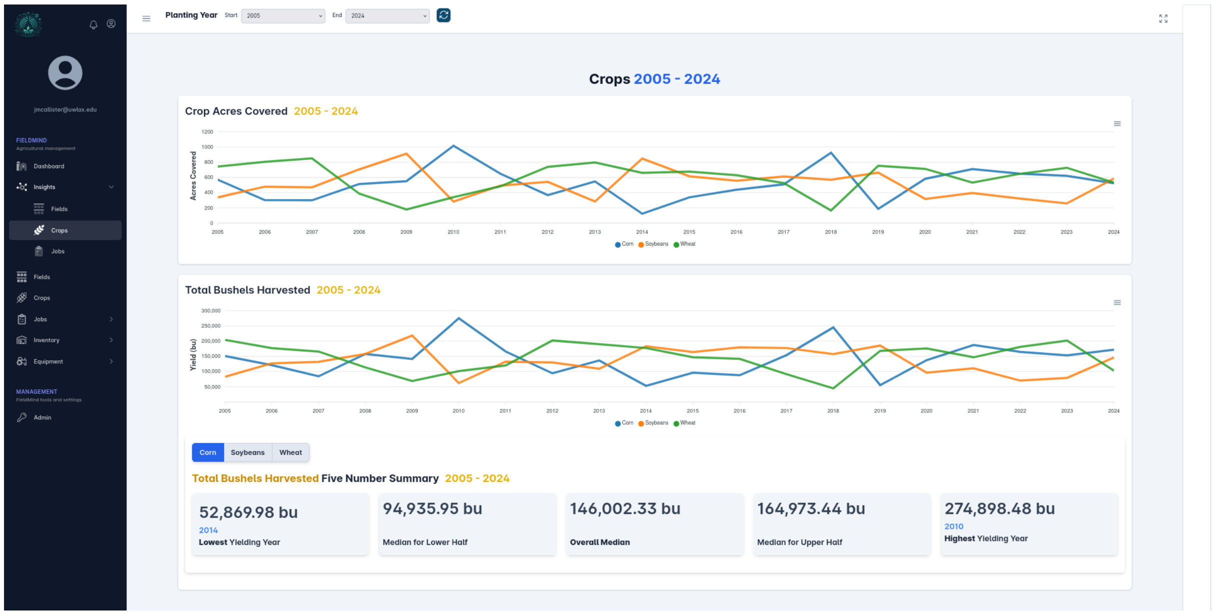


Figure 46. Tooltip for the candle stick chart.

4.10.2. Crops

The “Crops” insights contain two line charts: one for viewing the acres covered by each crop, and another showing the total bushels harvested for each crop. There is a line for each crop type on each chart. Additionally, underneath the line charts, there is a five-number summary on the total bushels harvested for each year based on the crop type that the user selects.



Total Bushels Harvested Five Number Summary 2005 - 2024

52,869.98 bu	94,935.95 bu	146,002.33 bu	164,973.44 bu	274,898.48 bu
2014				2010
Lowest Yielding Year	Median for Lower Half	Overall Median	Median for Upper Half	Highest Yielding Year

Figure 47. Charts for viewing various metrics on crops from the year range of 2005 through 2024.

4.10.3. Jobs

Currently, the only chart displayed in the "Jobs" insights section is a stacked bar chart that visualizes the count of job categories created for each year within the selected time range. As FieldMind's functionality expands, the `JobTask` entity can be enhanced to include additional attributes, such as tracking which jobs experience the most mechanical issues or identifying the most costly job categories. These additions would enable the generation of more detailed insights and visualizations for job-related trends.

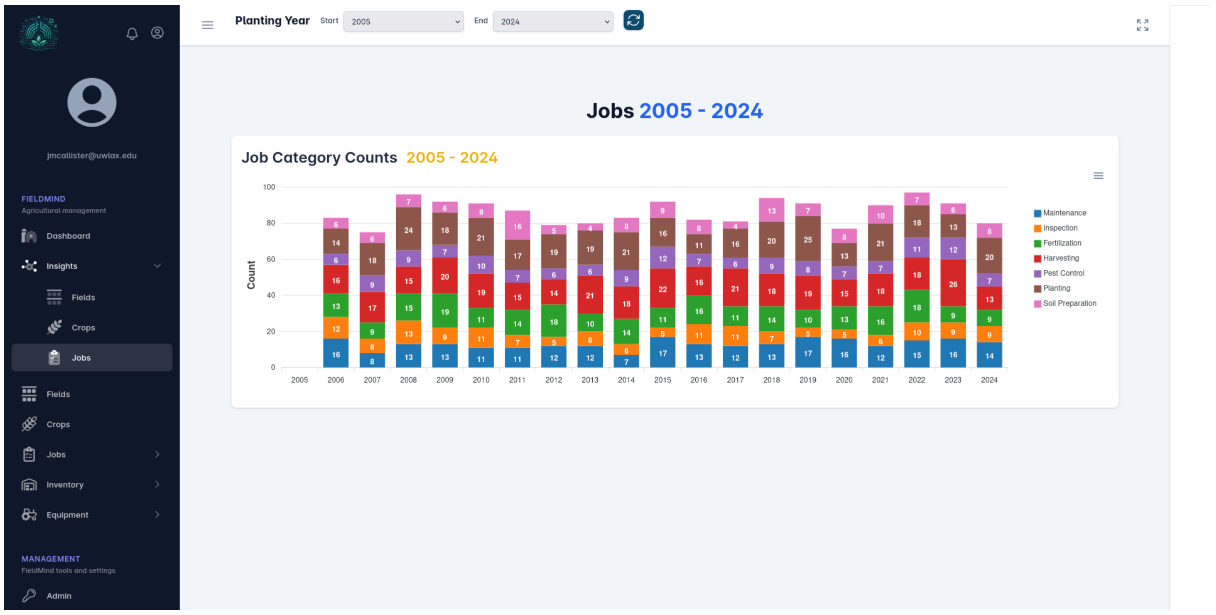


Figure 48. Stacked bar chart showing the breakdown of the count of jobs by category for the year range of 2005 through 2024.

5. Implementation

FieldMind is a web application which uses ASP.NET Core along with Entity Framework Core (EF Core) and Identity framework, following a Model-View-Controller (MVC) workflow for the backend.

On the frontend, AngularJS is used for the frontend framework along with the Angular Fuse Template as a foundation for the UI, and TailwindCSS is used for the styling. For the creation of the charts, I utilized open-sourced examples from Observable HQ, which uses D3.js in order to create intricate data visualizations. Additionally, ApexCharts is another library that was used in order to create charts seen throughout FieldMind.

The choice overall to use .NET and Angular is because of the experience that I have with these technology stacks. I have worked with these technologies in industry, and have also taken and taught classes using these technologies.

ASP.NET EF Core provides a powerful and flexible object-relational mapping (ORM) system. It allows developers to define entity models as standard classes, and based on the configurations registered in the database context file, the framework automatically generates the necessary tables. Each table's columns correspond to the properties of the entity classes, while relationships between tables are determined by how the models reference each other in their definitions. For example, take the following simplified (a number of properties irrelevant for this example have been omitted) `Equipment` and `EquipmentCategory` classes:

```csharp
public class Equipment {
    [Key]
    public int Id { get; set; }

    [ForeignKey("EquipmentCategory")]
    public int EquipmentCategoryId { get; set; }

    public virtual EquipmentCategory? EquipmentCategory { get; set; }
}
```

Listing 1. C# Equipment Class

```csharp
public class EquipmentCategory {
    [Key]
    public int Id { get; set; }
}
```

Listing 2. C# EquipmentCategory Class

Through the use of what .NET calls a "navigation property" of `EquipmentCategory`, because `EquipmentCategory` is another entity, EF Core knows that by including this navigation property here (namely that the **data type** is of the `EquipmentCategory` entity), that there is, in this case, a many-to-one relationship from 'Equipment' to `EquipmentCategory`. This will be reflected in the database schema. If one were to have two models, each with a property which had a `List` of the other, EF Core would know that this represents a many-to-many relationship between the two, and a subsequent junction table would be made between these entities, creating a composite primary key from the foreign keys of each table.

This ORM system makes defining the entity models, and the relationships between them, far more intuitive and streamlined. Additionally, when more customized configurations are needed, EF Core offers an API called the Fluent API in order to have more fine-grained control of how the relationships between entities should be created.

Additionally, .NET has something called Language Integrated Query (LINQ) which allows one to interact with collections and database entities, and can be used in conjunction with EF Core to perform database queries. LINQ provides a powerful way to interact with these types of objects allowing for filtering, mapping, and other manipulations, in a consistent way. The syntax can follow a SQL-like form, or it can also use a method syntax as well. When writing database queries, the queries follow a delayed execution which optimizes their performance. Using LINQ for database access, this also creates a standard and safe way for database queries to be built and executed.

SQLite was chosen for the database because it stores the entire database in a single file. This is convenient for development as it allows the database to be simple and be pushed to the project's GitHub repository with the code. This way, if developing on multiple machines, or for deployment of the application, I would not additionally have to setup a database server along with the application and connect them.

Angular was chosen primarily because of my familiarity with it on other projects and in industry. Additionally, Angular provides a very robust frontend framework for handling asynchronous events, components, and state management. One organizational aspect that I prefer with Angular over something like React, is that with Angular, each component has a specific file for its HTML, TypeScript, and CSS. I appreciate the separation of concerns within the component, as this design contributes to improved organization and maintainability.

The Fuse Angular Template was used as it provides a great foundation from which to base the UI on. Building the frontend of an app can often take a large amount of time due to the frontend's need to handle user interaction as well as visual layout. Fuse provides a robust starting point which can be leveraged to take a lot of the initial tedium out of the frontend setup and create a professional and responsive layout with minimal effort.

This streamlined UI design is critical for the application's ultimate functional use by a farmer. The layout needs to be something that is intuitive and also quick to navigate. FieldMind needs to be something that works **with** the farmer, not against them. My anticipation is that there would not be a lot of patience for hiccups in the way that the application receives data from a farmer. I think with software products like this, where the product is aiming to be a central part of a critical business operation, that any small amount of friction becomes greatly magnified due to the significance of the task which the application is designed to support.

Fuse provided a strong foundation for both the workflow and aesthetic early in development. Leveraging this framework eliminated the need to build the design from scratch, significantly reducing the effort required and streamlining the process.

Fuse also provides support to handle and manage the use of JSON Web Tokens (JWTs) for the frontend authentication system. This is a modern approach to handling the user's logged in state and helps to mitigate threats such as Cross-Site Request Forgery (CSRF) attacks.

Rather than writing custom CSS for the styling, TailwindCSS was used. This choice was made for two main reasons: I have familiarity with this CSS library, and Fuse also uses this as its CSS library as well. Additionally, I also think that Tailwind provides a wide range of options for styling and that the use of this library is easy and intuitive to use, making the styling a far smoother process.

Taking an approach of using styling libraries, such as Tailwind, is something that I have taken from my work in industry where using such libraries is preferred. This preference of using libraries such as these rather than writing custom styles arises from the fact that these libraries are well-documented and the effects of their styles are known. If one were to use their own custom CSS, it is often not well-documented, and can also easily lead to situations where different teams write different custom styles which are incompatible once each team needs to integrate their piece into the overall project. By using styling libraries, such as Tailwind, projects can be more consistent and predictable, which allows for better scalability and maintainability.

Fuse includes Heroicons, a free, open-source collection of SVG icons maintained by the Tailwind Labs team (creators of TailwindCSS). In FieldMind, these icons are used alongside FontAwesome icons. I opted for the paid version of FontAwesome because it offers a wider selection, including agricultural icons that were not available in the Heroicons collection. These agricultural icons can be seen in the navigation menu of FieldMind, as well as on certain pages. Additionally, FontAwesome provides a larger overall library with more stylistic variety. All of these choices contribute to the quality of the user experience as they help in making the frontend more visually pleasing and intuitive to use.

5.1. Features and Components

When implementing features and components for FieldMind, I started with the structure of the models. I typically like to start here because, for me, everything starts from here. I thought generally how the frontend may need to look and function, and used this to determine the general structure of the models. The model that I spent the most amount of time thinking about, by far, was the `LandDivision`. The `LandDivision` model is the one which the entire application revolves around, and all other models are based around. A lot of thought went into how this model should be structured in order to allow for the subdivisions, new crops each year, and the association of harvest data and jobs. The development of this took a number of iterations, as issues with a given manifestation were seen as other parts of the application were designed or created.

After the creation of the entity models (the model which is mapped to a database table), I would create the relevant Data Transfer Object (DTO) model. This model would contain a subset or superset of properties from the entity. For example, a superset of the `LandDivision` entity was used when sending down a `LandDivision` to the frontend. Along with all of the `LandDivision` entity's properties, the crops, harvested crops, harvest summary information, job information, and various hierarchical information about its parent division were also included - all of these properties are not part of the `LandDivision` entity, but are instead part of the DTO for the `LandDivision`. This is because some of these properties, such as harvest summary information, are derived; while others, such as the crops on the `LandDivision`, are properties which change year by year and thus must be gathered based on the selected year. An example of when a subset would be used would be with the entity representing the user. Here, a DTO would be used in order to hide some sensitive data, such as the password hash.

Once these models were defined, I created the relevant controller and service for a given model. The service is where the DTO gets constructed with the combination of mapping profiles for simple property mapping, along with service methods which do more rigorous work for getting values for the derived properties mentioned. Through the creation of the controller and the service, sometimes the entity and DTO models needed to be refined.

From there, I proceeded with creating the frontend component(s) and services which interacted with and display that given model. Once the frontend received the data and I started to build the layout, there again were times in which I realized that I needed to add certain properties to either the entity or DTO model in order to facilitate things like navigation to various routes, display certain information, or to perform various operations.

6. Testing and Verification

Because FieldMind is a web application, much of the testing was done through the user interface. As features were added to the application, those features were tested through the user interface, and the previous features were also tested to verify that the newly added features did not break any already implemented features.

There are a number of places where the yield, weight, and area calculations are done. During development, I kept the data seeded in the database at a manageable level so that I could verify that the math was done correctly. I systematically set up various situations to represent usual cases as well as edge cases along with their expected outputs so that I could verify the proper outputs were being produced and correct any issues as they arose.

6.1. Manual Testing Strategy

FieldMind's testing approach incorporates manual testing to ensure the accuracy and reliability of the application. The primary focus was on the following:

1. **Functional Testing** - Ensuring that each feature behaves as expected.
2. **Regression Testing** - Confirming that new updates do not introduce unintended issues.
3. **Edge Case Testing** - Verifying how the system handles unusual or extreme input conditions.
4. **Data Validation Testing** - Ensuring the integrity of calculations, such as yield and area usage metrics.

Because FieldMind is a web application, most testing was conducted through the user interface, validating how the system behaves under real-world usage conditions.

A structured manual testing process was followed for key features. The table below outlines some of the test cases covering different aspects of FieldMind:

Feature	**Initial State**	**Action Performed**	**Expected Outcome**
User Authentication	No user logged in	Enter valid credentials and log in	User is authenticated and redirected to the dashboard
User Authentication	Incorrect password entered	Attempt login with wrong password	Error message displayed, login attempt denied
Data Entry - Field Creation	No fields in the database	User creates a new field with valid inputs	Field successfully saved and displayed in the list
Data Entry - Crop Management	Existing field with no crops	User adds a crop with valid data	Crop is correctly associated with the field
Data Entry - Crop Management	Field already contains a crop for the year	User attempts to add a duplicate crop	System prevents duplicate entry, error message displayed
Data Retrieval	Several years of farm data exist	User selects a year from the toggle	Data updates to reflect selected year
Data Visualization	Various land divisions contain recorded data	User generates a yield comparison chart	Chart displays correct yield data per division
Field Deletion	Field contains subdivisions and crop data	User deletes the field	System prompts for confirmation and prevents deletion if dependent data exists
System Performance	User interacts with large datasets	User loads multi-year farm insights	Application loads and renders data without significant delay
Error Handling	Invalid data input for job creation	User enters an invalid date format	System prevents invalid input and displays an error message

Table 1. Manual Test Cases

6.2. Automated Testing Considerations

In addition to manual testing, automated tests could be implemented using unit tests and integration tests for core components:

- **Unit Tests**: Used for functions like yield calculations to verify correctness under controlled conditions.
- **Integration Tests**: Validate interactions between different components, such as database queries and API responses.

Future development could incorporate automated UI testing with **Selenium** or **Cypress** to reduce reliance on manual testing for repeated scenarios. Automated testing was not implemented in the current iteration due to time constraints. Given the broad scope and feature set of FieldMind, development efforts were strategically focused on building out core functionalities to ensure a working application could be delivered within the project timeline. While automated testing offers long-term benefits in terms of maintainability and reliability, the initial development phase prioritized the implementation of critical features to meet project objectives and provide a functional application for demonstration and evaluation.

7. Validation

Through the demos that I did with Glenn, the project was able to fit many of the practical needs of a typical crop farmer in the Midwest. The current state of the project is something that could be used in a production setting with a grain crop farmer in the Midwest. I credit this to Glenn's contributions through his expertise on the problem domain. Through continued conversations with Glenn, I was able to hone in on the right questions to ask and identify which features would be of most value to farmers like him.

The current state of the project represents a minimum viable product (MVP) release. FieldMind fulfills the core requirements of allowing users to track fields, crops, jobs, inventory, and equipment on a farm, while also supporting a rich suite of data visualizations derived from this data. Each of these functional areas has been implemented to meet the initial specifications, and the application enables users to interact with them in a manner that is intuitive and practical for Midwestern crop farmers. This usability has been validated through demonstrations and feedback from Glenn.

While the project offers a wide range of functionalities that make it practical for use by crop farmers, there are still areas for improvement that could further extend its capabilities and enhance its overall utility.

One such item would be the ability to edit certain aspects of a `LandDivision` and have that property only change for that year and not every year. For example, currently, a user can only edit size information on a `LandDivision` with no history, but not for any `LandDivision`s that do have a history of planting and harvesting of crops. This is because the `LandDivision` entity has its size as a property, and because the `LandDivision` exists as a single entity in the database, when this property is changed, all years will have this change. And so, this can break some of the rules of allowing a user to adjust the size of a `LandDivision` to a dimension that perhaps is large enough to fit all of the crops and subdivisions for that given year in which the user is editing, but this may make the `LandDivision` too small for other years given the crops and subdivisions it has during those years. For example, if a `LandDivision` is 80 acres in size and in 2023 has 40 acres used for subdivisions and and 30 acres used for crops, for that year, the user would be able to reduce the `LandDivision`'s size by up to 10 acres, as this is the amount of unused space there is. Though this may be fine for 2023, if the system has that same `LandDivision`'s configuration in 2024 as 40 acres of subdivisions and 40 acres of crops, that reduction made for the 2023 planting year now makes the `LandDivision` too small for 2024. The solution for this would be to create an additional table which holds properties, such as size, for a `LandDivision`, and associates it with a given planting year. And then when a given year is being viewed, the state of the `LandDivision` for that year can be retrieved and edited for that year only without effecting prior or subsequent years.

In the final meeting with Glenn, he said that the current state of the application has enough functionality to make it a practical product for a farmer to use. FieldMind represents the major operational aspects of crop farming to a degree which would add value to a farmer.

Glenn confirmed that the terminology aligns with what crop farmers use, and the workflow of the application as well as the overall design are intuitive and user-friendly.

Based on the initial project goals, FieldMind successfully meets the intended objectives for a minimum viable product (MVP) release. While there are additional features and refinements I would like to pursue, the core functionality aligns well with the original vision.

8. Security

Given that FieldMind is a web application, it has a very wide accessibility. This access can come in the form of both legitimate and illegitimate actors. Because FieldMind stores detailed information on a farm's operations, it is essential to store this data in a safe way and prevent unauthorized access of critical data.

8.1. Authentication

FieldMind uses ASP.NET Core Identity Framework which provides support for a wide variety of authentication methods such as password-based authentication, two-factor authentication (using SMS, email, or authenticator apps), external authentication through OAuth 2.0, and token-based authentication. There is also a verbose setup for authorization as well, which can be enforced on an entire controller, and also on an endpoint-to-endpoint basis.

FieldMind currently uses password authentication on login, and then issues a JSON Web Token (JWT) to persist the user's login for a configurable duration of time that they are active on the site. The Angular Fuse Template comes expecting this authentication flow and will work with the JWT issued by ASP.NET in order to facilitate a smooth user login experience.

FieldMind is an application which requires input from users. In the future, FieldMind may facilitate interaction between workers and the farmer. In this case, certain data created by one user may be viewed by others, thus the protection against attacks such as Cross-Site Scripting (XSS) is a serious concern. In this same spirit, Cross-Site Request Forgery (CSRF) attacks are also a major concern.

Starting with XSS, Angular has robust defenses against this. Angular will automatically escape potentially dangerous characters in templates and will instead render such dangerous characters as plain text rather than executable JavaScript. Angular also sanitizes HTML when using data bindings in the markup. In order to have Angular **not** perform this sanitization, a programmer would have to explicitly set the `bypassSecurityTrustHtml` bypass function.

As for CSRF attacks, these often rely on a browser's default behavior of sending cookies associated with a given origin with any requests made to that origin. However, JWTs are not automatically sent by browsers. Instead, JWTs are typically stored in `localStorage` (as is the case with FieldMind) or in `sessionStorage`. This requires the **application itself** to explicitly add the JWT to the request headers. Because this must be manually done by the application, this greatly reduces the risk of a cross-origin request automatically including authentication tokens.

8.2. Authorization

ASP.NET also offers authorization packages as well. Part of this is the use of attributes, such as `[Authorize]`. This attribute requires that a user is logged in before they will able to access the controller or endpoint which it resides over. The `[Authorize]` attribute can be placed over a controller as a whole, meaning that every endpoint is only reachable by a logged in user; or, you can use the `[Authorize]` attribute over specific endpoints within the controller, and guard only those in this fashion.

This `[Authorize]` attribute is employed on all of the controllers throughout the application, providing an exception to only certain endpoints which do not require the user to be logged-in order to access such as the `sign-in` and `sign-up` endpoints.

However, just being logged in is not enough to secure routes in our case. For example, many backend controllers are reached with the inclusion of the farm ID that a user is desiring to get information about or modify. An example is the following route to reach the `LandDivisionController`:

```
"api/farms/{farmId}/land-divisions"
```

As seen below, though the `[Authorize]` attribute will protect this route to a certain extent ensuring that the user is logged in, it does **not** enforce that the user has rights to the farm with the farm ID of `farmId`.

```
[Authorize] // Just Authorize is not enough to secure this controller.
[Route("api/farms/{farmId}/land-divisions")]
[ApiController]
public class LandDivisionsController : ControllerBase {
    /* Controller code */
}
```

This means that a logged in user would be able to merely manipulate the part of the URL string which has the farm ID to be whatever farm that they wanted, and they would be allowed access to data for farms which they are not a part of. This is clearly not an acceptable action to allow.

The following is an example of manipulating the URL to view data on arbitrary farms:

```
"api/farms/1/land-divisions" // Farm with ID of 1
"api/farms/8/land-divisions" // Farm with ID of 8
```

In addition to the enforcement of the user being logged in, we also need an authorization policy which can enforce that the farm ID that is being requested by the logged in user is indeed a farm to which this user has access. In order to do this, I created a custom policy which uses various claims (information about the logged-in user) made available via Identity Framework which checks for and enforces exactly this. The following block of code shows the **use** of said policy, though I have defined this policy elsewhere in the application (the details of its definition are not needed for this description).

```
[Authorize(Policy = "FarmAccessPolicy")]
[Route("api/farms/{farmId}/land-divisions")]
[ApiController]
public class LandDivisionsController : ControllerBase {
    /* Controller code */
}
```

Listing 3. C# LandDivisionsController

This policy will provide the access control that is required to ensure that farm data is only given to users which are authorized to view that data. Currently, FieldMind is built for use of a farm run by a single person, and therefore this person has no privilege restrictions on the farm. However, in future iterations of FieldMind, the user base may expand and there may be a more tiered structure on access privileges. This approach used the `FarmAccessPolicy`, and other approaches similar to this can be leveraged in order to include more intricate access privilege structures.

8.3. Error Handling

Security of an application also extends to the proper handling of errors which may occur during its operation. One of these critical error checks is ensuring proper checks for `null`. The C# language has a rather verbose ability to allow for nullable types (marked with a `?` such as `int?`) and provides convenient accesses to know whether or not a value is `null` and to retrieve the actual value which a nullable variable holds.

There were times throughout the construction of the FieldMind where errors cropped up and debugging was needed. Because I had taken care to check for various error states and produce appropriate error messages when said errors occurred, when these issues occurred, I was able to quickly locate the source of the issue and correct it. If not for those error checks and subsequent descriptive messages, finding the errors would have been much more time-consuming.

For the backend code, there are controllers and various services which facilitate the gathering and modification of data. The flow of the system is that the controllers handle requests, and will then use services and call methods of those various services in order to perform whatever transformations or more intricate manipulations of data are necessary. This separation of concerns keeps the code more streamlined and organized; but, there also needs to be a way for the controller to handle an error that may have occurred deep within a service. The controller must have an indication of whether or not a service was able to complete a given action, and if it was not, to know what went wrong.

In order to facilitate this requirement, the following `ValidationResult` data type was created.

```
public class ValidationResult<T> {
    public T? Result { get; set; }
    public string? ErrorMessage { get; set; }
    public bool HasError => !string.IsNullOrEmpty(ErrorMessage);
}
```

Listing 4. C# ValidationResult Class

Notice how the `ValidationResult` data type uses a generic for its payload. This structure allows for the payload to be very dynamic as far as its type, while simultaneously having guarantees about the overall structure of the `ValidationResult` type as a whole.

Below, we can see how the `ValidationResult<T?>` is used within the `GetAllLandDivisionsForYear(int farmId)` method. The generic type is leveraged to allow a variety of types for the payload that will be returned while also maintaining the consistency of indicating if an error occurred, and if one did, the message of what that error was. In the below example, if an error does occur in one the the subsequent method calls made by this `GetAllLandDivisionsForYear(int farmId)` method, we merely place that error message into the `validationResult` and then return that `validationResult`. In this way, we are able to propagate errors up the stack and also prevent further execution of a method if an error occurs. This prevents unexpected crashing and also lends the way for much faster problem diagnosis and resolution.

```
public async Task<ValidationResult<ICollection<LandDivision>>>
   GetAllLandDivisionsForYear(int farmId) {
    ValidationResult<ICollection<LandDivision>> validationResult = new
        ValidationResult<ICollection<LandDivision>>();

    if (!_yearContextService.StartYear.HasValue) {
        validationResult.ErrorMessage = "No planting year has been
            selected. Planting year is required.";

        return validationResult;
    }

    ValidationResult<Expression<Func<LandDivision, bool>>> predicateResult
        = this.AllLandDivisionsForYearPredictate(farmId);

    if (predicateResult.HasError) {
        validationResult.ErrorMessage = predicateResult.ErrorMessage;

        return validationResult;
    }

    if (null == predicateResult.Result) {
        validationResult.ErrorMessage = "Expected to have queryable
            predicate for parent divisions, but got null instead.";

        return validationResult;
    }

    /*
        Other code continuing here.
    */

    return validationResult;
}
```

Listing 5. C# GetAllLandDivisionsForYear Method

The reason that this flow is desired over merely throwing an exception is because we are dealing with controllers for a web application. Therefore, ultimately, we want the controller to be able to respond to the client with an HTTP response. The application gives error messages which the frontend code parses in order to respond with a redirection and/or appropriate error message. The extra time invested into having verbose error messages paid for itself in dividends of time which was recovered in diagnosing and fixing bugs when they arose.

I make such a significant mention of this because the handling of errors is often something that can fall by the wayside when trying to move quickly in building an application. And though this mention of error messages is in the security section of this thesis, proper checking and handling of errors has far wider implications than just security. As mentioned, when errors occurred, because of the care given to checking for these issues and then producing descriptive messages which are indicative as to **what** happened and **where** something happened, these errors were able to be quickly tracked to their source and corrected.

When working on more complex systems, the need for the thoughtful detection and handling of errors makes itself manifest. There is an upfront cost to creating these mechanisms, but the benefits are very quickly realized. Even if there are often no errors which crop up, there is a peace of mind in knowing that checks are being done and that invalid states of various kinds are being handled. It gives far more confidence that the progress one is making is actually progress, and that there is less of a likelihood of some unanticipated "trap doors" lurking somewhere.

8.4. Database Interaction

Database accesses are another area which can introduce significant security vulnerabilities. FieldMind uses LINQ, which stands for Language Integrated Query, in order to retrieve data from the database and also to perform updates. LINQ is a feature of .NET languages, such as C#, which allows for the querying and manipulation of data from various sources such as collections, databases, and others, in a unified, readable, and type-safe syntax.

When using LINQ, there are a number of benefits over writing raw SQL queries. LINQ performs compile-time checks on the code that is written rather than being just a raw string, as is the case with SQL. The queries that are generated with LINQ are also parameterized, which means that the input is treated as data, not executable SQL code. LINQ also optimizes the queries that it generates based on the database provider that the application is using.

By using LINQ, there is a standardized way to access the database - and interact with any collection for that matter - and therefore, this will be a common, well-documented API for other developers as well. This makes the use of LINQ much more scalable and maintainable. The LINQ code that is written by the developer is also database agnostic, so if the project were to change its database provider, none of the LINQ code would need to change since it will generate the appropriate SQL code based on the database that it needs to communicate with.

8.5. Concurrency

One of the important features of FieldMind is its ability to provide aggregated data over a span of years. Throughout the application, there are a number of pieces of data which need to be retrieved from the database, and various transformations that need to be done on each entity in order to put it in the desired form for either further manipulations or for serving to the frontend. The use of asynchronous code is critical for these operations, particularly with a user base that grows, so as to keep the server able to handle this kind of load being placed upon it. However, asynchronous code on its own is not enough to reduce bottlenecks. In conjunction with the asynchronous code, for operations such as transformations which must be done for each element in a set of elements, running these tasks **concurrently** in an asynchronous fashion provides the desired efficiency.

```
public async Task<D3HierarchyData> FarmAreaUsageTreeMapData(ICollection<
    LandDivision> landDivisions) {
    D3HierarchyData rootNode = new D3HierarchyData {
        Name = "Farm",
        Children = new List<D3HierarchyData>()
    };

    if (0 < landDivisions.Count) {
        // Create tasks for each Land Division to execute concurrently.
        IEnumerable<Task<D3HierarchyData>> landDivisionTasks
            = landDivisions
            .Select(landDiv => this.AreaUsageTreeMapNode(landDiv));

        /**
            Using Task.WhenAll() allows for all Land Division tasks
            to run concurrently.
            This approach is more efficient than awaiting each
            task sequentially, as it waits only for the
            longest-running task, not the sum of all tasks.
        */
        ICollection<D3HierarchyData> childNodes
            = await Task.WhenAll(landDivisionTasks);
        rootNode.Children.AddRange(childNodes);
    }

    return rootNode;
}
```

Listing 6. C# FarmAreaUsageTreeMapData Method

Take the above block of code, for example, which demonstrates the use of the `Task.WhenAll()` approach, a common pattern throughout the application. The `Task.WhenAll()` method can accept either a collection of asynchronous tasks, as shown here, or multiple asynchronous tasks as separate arguments. Instead of awaiting each task sequentially, `Task.WhenAll()` enables them to execute concurrently, ensuring that the total wait time is only as long as the longest-running task rather than the sum of all tasks.

The `FarmAreaUsageTreeMapData` method processes a collection of `LandDivision` entities to build an interactive tree map representing the farm's area usage. To determine area usage, it retrieves relevant data for each `LandDivision` within the selected year, including associated crops and subdivisions, as well as any nested subdivisions and their corresponding data. Since this requires multiple database look-ups, execution time can vary depending on the number of crops and subdivisions at each level.

To maximize efficiency, all necessary database queries and processing tasks are initiated simultaneously. By gathering all `Task` objects and tracking them with `Task.WhenAll()`, the method ensures that data retrieval is completed in the shortest possible time - dictated by the longest-running task rather than the cumulative execution time of individual tasks. This significantly improves performance by leveraging concurrent execution.

Care has been taken throughout the backend to apply this approach wherever possible, optimizing CPU utilization and making the application more scalable and responsive to user interactions.

9. Deployment

FieldMind is deployed using Azure cloud services, chosen primarily because the backend is built as an ASP.NET web application. Given that ASP.NET and Azure are both Microsoft products, I anticipated that Azure would provide a streamlined deployment process. Research confirmed this assumption, and I also received guidance from coworkers experienced with this technology stack.

For domain registration, I purchased `fieldmind.io` through Porkbun and configured it to connect to the web app on Azure during deployment.

9.1. Preparing FieldMind for Deployment

Before deployment, a **production build** was required for both the **backend** and **frontend**:

1. **Backend (.NET) Build**
    - A production build was generated using the `dotnet publish` command, which compiles the application into its binaries and places them in a `Publish` directory inside the `bin` folder.
    - The location and naming of this output directory are customizable in the command.
2. **Frontend (Angular) Build**
    - The Angular portion was compiled using the `ng build --configuration=production` command, creating a `dist` directory containing the built frontend files.
    - Since the project uses the Angular Fuse Template, the output files were placed inside a `fuse` subdirectory within `dist`.
3. **Integrating the Angular Frontend with the .NET Backend**
    - By default, .NET serves frontend files from its `wwwroot` directory.
    - Since FieldMind uses an Angular frontend, the existing contents of `wwwroot` were removed.
    - The compiled files from Angular's `dist/fuse` directory were then copied into `wwwroot` so that .NET could serve them properly.
4. **Configuration in `Program.cs`**
    - Additional configurations were required in `Program.cs`, which serves as the entry point for the .NET application.
    - These included setting up environment variables, defining database paths, and ensuring that the SQLite database would be created on startup if it didn't already exist.

9.2. Setting Up Azure Resources

- **Resource Group** – A logical container managing all related Azure services. This functions as a virtual server environment.
- **App Service Plan** – Defines the computing resources allocated for running the application.
- **App Service (Web App)** – The primary hosting service for the ASP.NET backend and Angular frontend. Multiple web applications can be hosted within a single resource group.
- **Storage and Database Considerations** – Since **SQLite** was used as the database, it was stored in Azure's persistent storage (`/home/`) instead of `wwwroot`, which could be overwritten during deployments. Proper write permissions were granted to the `/home/` directory to ensure data persistence.

9.3. Deploying FieldMind to Azure

Once the necessary Azure resources were created, the application environment was configured:

1. **Setting Environment Variables**
   - Sensitive information such as JWT authentication keys and database connection strings were securely stored as environment variables in the Azure portal.
2. **Database Configuration**
   - The SQLite database path was updated to be stored in `/home/` for data persistence across deployments.
   - A configuration was also added to ensure that FieldMind automatically creates the database on startup if one does not already exist.
3. **Uploading the Application to Azure**
   - The Azure Command Line Interface (CLI) was used to push the application to the web server.
   - Once uploaded, I accessed the server via SSH through the Azure portal and navigated to `/home/site/wwwroot/`.
   - Inside this directory, the `Publish` folder (containing the compiled .NET application) was moved to `wwwroot`, and the now-empty `Publish` directory was removed.
4. **Starting the Application**
   - A start command (or restart command, if updating an existing deployment) was issued via the Azure CLI to launch the application.

5. **Configuring the Domain**

    - After verifying that the application was running, I configured DNS records in Azure to associate `fieldmind.io` with the IP address assigned by Azure.

One potential improvement would be migrating from SQLite to an Azure SQL Database for the production environment. SQLite operates as an embedded database and relies on file-based storage, which can become inefficient for a data-intensive application like FieldMind. As the database grows, query performance can degrade since SQLite loads entire datasets into memory rather than leveraging optimized indexing and query execution strategies used by full-fledged database management systems. By transitioning to Azure SQL Database, FieldMind could benefit from better scalability, optimized query performance, and improved memory management, ensuring a more efficient and responsive application.

Overall, deploying FieldMind to Azure required careful preparation, from generating production builds for both the backend and frontend to configuring the hosting environment and ensuring data persistence. Azure's compatibility with ASP.NET, along with its suite of cloud services, made it a logical choice for hosting FieldMind, providing a streamlined deployment process with scalability for future enhancements. The use of environment variables, database persistence strategies, and structured resource management ensures that the application runs efficiently and remains secure. With fieldmind.io now fully operational, future improvements can focus on optimizing performance, expanding feature sets, and refining user interactions to better serve the needs of Midwestern crop farmers.

10. Challenges

Early on, in the spirit of scalability, I was trying to make FieldMind applicable for any kind of crop farm that you could think of - whether that be corn and soybeans, or something like bananas or cherries. However, as I built FieldMind with this aim, I began to realize that this mindset was causing me to become deadlocked in not knowing the proper manner in which to proceed. I would question whether my approach was general enough to fit all cases or if I was being too specific. And I had no way of knowing either way. I also did not even know the questions that I should be asking for these other farming situations either. The intentions were good - I was trying to make something that was not too specific for a given situation. In other words, I did not want to overfit a specific problem domain. I soon came to the conclusion though that, by trying to do **everything**, I was not being able to do **anything**. What I have is a consultant for a Midwest crop farm for grains. Therefore, that is what the application should be built for. Perhaps there would need to be some reworking and expanding of certain aspects of FieldMind in order to accommodate other kinds of farming, but those features and challenges are something that need to be addressed when tangible, actionable knowledge has been gained about those problem domains. Otherwise, I would be truly just making shots in the dark.

By narrowing the scope to a Midwestern crop farmer, whom I had as a consultant, I could create an application which was well-suited for this purpose. In doing this, this allowed me tangible implementations of not just aspects that manifest itself currently in the application, but also a setup that is still expandable to other kinds of crops as well. The implementation details for these other kinds of crops are left for future versions of the application because, just as with the situation of a Midwestern crop farm, a consultant(s) would be needed in order to get the details right. By getting the details right for the tangible situation of a Midwestern crop farm, many of these design choices set the stage for what would be needed for other kinds of farms once that domain knowledge is accessible by way of a consultant.

For example, yield calculations for Midwestern grain farms are given in bushels. Furthermore, to assess the efficiency of a field, the bushels gathered for a given crop are divided by the area covered by that crop to give a "bushels per acre" measurement for the field. There are surely other measurement units and divisions done on other crop types as well. Though the conversions and specifics of this can be acquired through research, having the research be vindicated and contextualized by a farmer who farms that crop is a necessary requirement in order to have proper implementation of not just the aforementioned conversions, but also so that the data can be gathered, aggregated, and displayed in a way which makes sense for those kinds of farmers.

The idea of overfitting also embodied itself early in the design as well when I was first gathering requirements and building the initial models. I *did* overfit a description from Glenn about how farms subdivide their fields and how they label these subdivisions. Glenn had described how a farmer can subdivide their field and that it worked in the following way:

- A **field** can be subdivided in half, each being called a **section**.
- A **section** can be divided in half, each being called a **section quarter**.
- Finally, a **section quarter** can be subdivided into a **plot**.
- A plot is the smallest unit.

With this information, I initially created one entity for each of these different divisions, and based on whether that division could be divided, it would have a property of a collection of the entity lower in the hierarchy. For example, a `Field` had an `ICollection<Section> Sections`, the `Section` entity had an `ICollection<SectionQuarter> SectionQuarters`, the `SectionQuarter` entity had an `ICollection<Plot> Plots`, and then the `Plot` did not have such a property since it always functioned as a leaf node in this structure.

However, as indicated, this design was overfitted to its description and actually created a number of problems as I began to think more about the necessary mechanics of the application. First of all, each of the above types really would only differentiate by name and by what - and if - it had a collection of children divisions. Otherwise, at each level, there was the option to hold crops. But, if say, the `Field` entity, was divided into two `Section` entities, since this would mean that the `Field` was divided in half, that would mean that the `Field` cannot hold any crops since that would be delegated to the `Section`. And then what if the `Section` was divided? And what if those `SectionQuarters` were divided? How do we know where the planted crops should be attributed? And then furthermore, we would also have to limit the size of the collections as well. This becomes unnecessarily complicated to track and enforce.

Additionally, imagine if a farmer wanted to use a small part of the farm for a seasonal pumpkin patch, or something of this sort. If it was small, the size of a `Plot`, then in this setup, the system would force the farmer to create a `Section`, then `SectionQuarter`, before ultimately being able to create a `Plot`. This does not make sense, and also can create an organizational nightmare for the farmer for wanting to do such a simple action of creating a small plot for one thing or another.

The deletion of an entity within this hierarchy presents significant challenges. Consider a scenario where a farmer has a fully structured hierarchy down to the `Plot` level - a `Field` divided into two `Sections`, each further divided into `SectionQuarters`, which in turn contain `Plots`. If one `Section` were to be deleted, the system would need to determine how to reassign the remaining `SectionQuarters`. Since a `Field` can only contain two `Sections`, one of the `SectionQuarters` would have to be promoted to a `Section` to maintain the required structure, while the other might need to be converted into a `Plot`. This restructuring is neither intuitive nor practical, leading to inconsistencies and unnecessary complexity.

Furthermore, all entities within this hierarchy share more similarities than differences. Their primary distinction lies in their assigned entity names (‘Field‘, ‘Section‘, ‘SectionQuarter‘, or ‘Plot‘) and the specific rules governing their subdivision relationships. Given these structural similarities, enforcing rigid hierarchical constraints introduces unnecessary complications.

Instead of creating all of these different entity types with these rules, I created an general `LandDivision` entity. This `LandDivision` entity functions just as a regular tree node in computer science, wherein every node is the same as all others and is only differentiated by where it is in the tree and by what value(s) it carries. Each `LandDivision` entity has a `Subdivision` property, which is just an `ICollection<LandDivision>` for its data type, and it has a nullable `ParentDivisionId` which contains the ID of the `LandDivision` which is its parent. If it is a root `LandDivision`, then the value of `ParentDivisionId` will be `null`. In order to represent the `Field`, `Section`, `SectionQuarter`, and `Plot` idea, each `LandDivision` has a `LandDivisionType` property which is just a category for the `LandDivision`. This `LandDivisionType` is user-defined and therefore allows the farmer to decide how they would like to have any sort of hierarchy organized. But these decisions will be placed in the hands of the farmer. This satisfies the organizational structure that Glenn had described, while also allowing this to be general enough to fit more than just the described hierarchy. This simultaneously frees FieldMind from having to micromanage this organization - a task which would have likely only caused excruciating overhead with very little, if any, payoff.

One of the other major challenges that was faced was the management of data over time. Maintaining the integrity of historical data while still allowing the user to make edits to the various entities is something that needed to be handled delicately. And I think that there are still some places where some improvements can be made in addition to the already mentioned `LandDivision` size adjustments. For example, does it make sense that a user deletes a given entity and then later restores it? In certain cases, perhaps the answer to this is “yes”, but in others, perhaps the answer is “no”. What would be the deciding criteria for this? If someone mistakenly deletes something, I think that, in general, they should be allowed to undo this action. However, in what ways should this deletion happen and what limits should be imposed on them? If a user is interacting with a `LandDivision` and then goes back a certain number of years on that `LandDivision` and wants to delete a subdivision - if that subdivision persisted for the years **after** the year currently being viewed... should that subdivision only be deleted for that year? Should it be deleted for all years after that? And what of its harvest information and all other data associated with it such as jobs and resource consumption - what should be done with all of that data?

It is questions like these that I found presented the most challenges. And to highlight - these questions are difficult not because there is a lack of knowledge for how to technically execute one solution or another; instead, the difficulty comes from knowing which path should be taken, or which **combination** of paths should be taken. These kinds of questions are really design questions, and are also questions about, psychologically, what the user is going to expect when certain actions are performed - and whether or not these actions should even be **allowed** to be performed.

I think that when building systems like these, when the technical know-how is there, much more attention turns to **what** should be done rather than the technical **how** - at least for the kinds of problems I'm describing. And again, I would say that this is where the consultation with domain experts is critical, because perhaps you are looking at a problem in the wrong way; perhaps you are fretting about situations which will not occur and thus should not be **allowed** to occur in the application. Knowing the bounds of what the application should concern itself with and what it should not is where the answers to some of these questions are, and where the breadcrumbs of the path to take can be found.

There is also nothing wrong with deciding that an application does not support certain features for a given release either. This circles back to the idea of wanting to do one thing well. Not everything is going to be able to be accomplished in one step. Perhaps the conclusion is reached that at a given point in time, the "right" decision is just to not have whatever given feature or action is in question. Those features perhaps are better introduced when the application is more mature and when a better understanding is held by the programmer and/or team.

This is the exact conclusion that I came to when thinking about the span of agriculture and software features that I wanted FieldMind to be able to support for this MVP release. I strongly believe that making these decisions in the right places can actually make an application's use more broad because, ironically, it is not trying to solve all problems on a given release. You have to make something specific enough so that it can transition from idea to reality. From there it can grow.

I would say that the greatest challenges when developing FieldMind were these exact problems rather than the technical execution of them per se. I knew **how** to create everything that I wanted to create, but it came down to questions of **what** should be implemented, and in what ways the user should interact with these aspects. The manner in which these features would be displayed and controlled by the user was another matter as well.

11. Conclusions and Future Work

I've worked in industry as a software engineer for about seven years. Each of the companies I worked for were startups, and they all had significant codebases that I contributed to. Since they were startups, I led the projects I worked on and was responsible for implementing major features. These included integrating merchant processors—financial services that handle electronic payments between businesses and customers—to overhaul how client companies paid their entire staff, adding multilingual capabilities to a telephony system, developing a Google Home agent for contextual conversations, integrating QuickBooks' API with an existing accounting system, and other projects.

All of the aforementioned experiences have caused me to grow tremendously as a software engineer and pushed forward my programming and problem-solving skills in major ways. I would say that this project with FieldMind has grown my software development skills in ways that I did not anticipate.

As with any project, there is an initial underestimation of the complexity and the challenges. And with this project, there were a number of instances where I "did not know the question to ask" because even though I'd worked on large systems, I'd never built them from scratch.

There is an inclination to dismiss away concerns about the UI and how that will shape up and be interacted with by the user. The rationale for this being that these things are rather superficial and are more "design" aspects and not the "hard engineering" aspects which are the "real work" of the project. However, I would give much more attention to drawing out what the UI would be if I were to do a project like this again. The UI ultimately is what is going to trigger the backend machine to turn its gears in the desired ways. The UI offers the levers which will make the machine function, and in turn tells you the functionality that you must have. As the project progressed, this is why I made my to-do lists based on the frontend navigation menu - because this is what was going to inform me of what I needed to do for all of the functionality mechanics, and it kept all of this very nicely organized.

As I say these things, it may sound obvious. However, I would argue that many of us, as programmers, can have a tendency to have this exact sort of attitude, which in turn can decrease user-friendliness, and in turn, the functionality that actually ends up being used or implemented, for a given application.

For example, I mentioned earlier in this paper about how the `LandDivision` entity currently has a restriction of not being able to edit the size, and some other details, if there is any historical data on that `LandDivision` due to the fact that changing those properties would change it for all years since there is not a separation of those properties year-by-year: when you change it, you change it for the single instance of the `LandDivision`. This oversight happened because I was focused on looking at a `LandDivision`, and indeed the entire application, for **one year** for the initial design. I did not want to focus on a UI feature such as looking at data over a span of years. However, had I paid more attention to some core features of how the UI would flow and what the user should be allowed to do, this flaw could have been seen and addressed earlier in the process. This issue would have required far less testing to verify its correctness early on because every feature would have been built with this design already a part of the app, rather than now, having to retroactively add this in and then address any issues caused by this adjustment.

This example is also what I mean when I say that being dismissive of the UI design elements can indeed affect the actual implementation of an application. If you build too many things up without considering a more holistic picture of how it all fits together, UI and all, this may end up pushing out some features for a given release because of the risk of introducing them at the stage in which they enter your foreground of your thoughts and attention. Even if the given features do get implemented, if attention is not paid to how the UI is structured, they may never be used anyway - and that is about as good as not implementing it at all.

Lingering on the design of the `LandDivision`, which is a core entity of FieldMind, I initially thought of this as being an entity which would have properties such as the crops, harvest, and yield information as part of its definition. But, as I began to see, this would not work since this information would change year-to-year. This year-to-year change was more obvious earlier in FieldMind's development because of the role that this data plays in the application - these are more central to many of the charts. Instead, this data, along with other pieces of data, on the `LandDivision` are actually **not** a part of the `LandDivision` entity, but are instead part of the Data Transfer Object (DTO) for the `LandDivision`. This means that the `LandDivision` that is ultimately displayed is actually more assembled by the application upon request rather than existing in the database as the user sees it. We "put Humpty Dumpty together" every time that we request and interact with a `LandDivision`.

This "scattering" of the properties like this also happens because there are a number of properties, such as all of the yield information, which are **derived** and thus not something that you would want to store in the database. These ideas were illustrative for me about how all of these components fit together. When building simpler projects, you do not necessarily see how all of these pieces come together, and how it is not any **one** part which does everything, but how the "machine" itself is only whole once it becomes comprised of all of the parts which make it up. An example of this would be how the `LandDivision` entity actually does **not** store all information about the `LandDivision`, but instead only a fraction. If you want to look at a `LandDivision` in the way that you would understand it with crops, harvest, yield information, and so forth, whether for a single year or span of years, you have to use the various services for the `LandDivision`, `Crops`, `HarvestedCrops`, and `Yield`, in order to assemble the whole picture. I picture this like a vehicle in the real-world - a car is not any one of its parts, but it is the sum of them; and, each part is dependent upon the others. Seeing this connection only reveals itself when a project grows to a certain level of complexity. With simpler projects, one part may dominate more and not rely so heavily on all of the others.

For future work with FieldMind, I would like to extend its ability to track more details about the existing entities. I would also like to incorporate 3rd party APIs, such as ones from the USDA, so that a farmer can see how their farm compares to overall trends seen across the United States. Expanding on more charting features can also provide a richer set of insights for farmers as well, further adding to the value that FieldMind provides. Additionally, the incorporation of physical sensors which can be placed out in a field and then have that data sent to FieldMind and displayed to the farmer would add another layer of depth which would be unique to each farm. I would also like for FieldMind to be able to allow for workers on a farm to have an account and the appropriate permissions as well. These workers could work on just one farm, or multiple farms, and have all of this be facilitated through FieldMind.

FieldMind could also allow for the tracking of `LandDivision`s owned by a certain farm and then rented by another. This would allow for a deeper network of connections to be made through the application. On this note, a given company or person may own multiple farms, and the ability to manage and switch between these would be something for future development as well.

The ease of use of the application by having as little manual entry of data as possible for common tasks would greatly increase the user-friendliness of the application. For the entry of scale ticket information, I would like this to be done via the user taking a picture of the scale ticket they received, and then having FieldMind fill in the data from that photo. This way the user would only need to audit this data in the application. In this same spirit of reducing that amount of manual work that would need to be done, if a farmer would like to enter in data from previous years before signing up for FieldMind, I would like the application to allow for the farmer to be able to upload their harvest summary data for previous years. This way these totals for the harvest summary can be used and the main charts showing performance of a field over time can still be built without forcing the farmer to enter in scale tickets for previous years.

A number of these features have some foundations already setup in FieldMind, but would need some more significant work in order to come to full fruition. These are features that I would like to continue to develop with the application. Agriculture is such a crucial part of our society, and I think that there is a need and opportunity here to bring value to farmers in a way that puts their concerns first.

12. References

[1] U.S. Department of Agriculture, National Agricultural Statistics Service, "2022 census of agriculture data now available." https://www.nass.usda.gov/Newsroom/2024/02-13-2024.php, 2024. Press release announcing the release of the 2022 Census of Agriculture data.

[2] United States Department of Agriculture, National Agricultural Statistics Service, "Table 52. selected operator characteristics: 2022 census of agriculture - iowa." https://www.nass.usda.gov/Publications/AgCensus/2022/Full_Report/Volume_1,_Chapter_1_State_Level/Iowa/st19_1_052_052.pdf, 2024. Provides demographic information about Iowa farm operators, including age, sex, and experience.

[3] A. Plastina *et al.*, "Iowa farm costs and returns 2019–2023 (agdm file c1-10)." https://www.extension.iastate.edu/agdm/wholefarm/pdf/c1-10.pdf, 2023. Supports the cropland figure: full-time Iowa farms averaged 587 crop acres in 2023. This sample includes primarily larger operations.

[4] United States Department of Agriculture, National Agricultural Statistics Service, "Quick stats: Iowa state agriculture overview." https://www.nass.usda.gov/Quick_Stats/Ag_Overview/stateOverview.php?state=IOWA, 2024. Provides key statistics on Iowa agriculture, including acreage, production, and economic value across various commodities.

[5] USDA National Agricultural Statistics Service, "Farm production expenditures – iowa 2023." https://www.nass.usda.gov/Statistics_by_State/Iowa/Publications/Economics/2024/IA-Farm-Production-Expenditures-07-24.pdf, 2024.

[6] United States Department of Agriculture, National Agricultural Statistics Service, "Iowa state agricultural statistics." https://www.nass.usda.gov/Statistics_by_State/Iowa/index.php, 2024.

[7] U.S. Department of Agriculture, Economic Research Service, "Farm income and wealth statistics - state-level farm income statement." https://www.ers.usda.gov/data-products/farm-income-and-wealth-statistics/, 2023.

[8] United States Department of Agriculture, National Agricultural Statistics Service, "2022 census of agriculture." https://www.nass.usda.gov/AgCensus/, 2024.

[9] Iowa State University Extension and Outreach, "Costs and returns – ag decision maker." https://www.extension.iastate.edu/agdm/cdcostsreturns.html#yields, 2024. General reference page for cost and return estimates.

[10] USDA National Agricultural Statistics Service, "Usda nass publications." https://www.nass.usda.gov/Publications/, 2024.

[11] United States Department of Agriculture, National Agricultural Statistics Service, "2022 census of agriculture - iowa state data (volume 1, chapter 1)." https://www.nass.usda.gov/Publications/AgCensus/2022/Full_Report/Volume_1,_Chapter_1_State_Level/Iowa/, 2024. Landing page for 2022 Census of Agriculture - Iowa state-level reports.

13. Appendices

13.1. Thoughts on Efficiency

One of the major takeaways that I've had from this project has been a maturing of my view of software design - particularly how to write well-written code and balancing this with moving at an effective rate of progress. The question of **what** should be implemented, as mentioned, is certainly the first pressing question that must be answered. After this has been decided, the decision must now be made with the way in which you would like to build something. With the aim of building something that is robust, it can be easy to over-commit to a certain approach which must only be redone later. Or perhaps making something robust becomes more of a distraction and instead slows down progress towards the overall objective. And so the question becomes how to balance these two seemingly opposing requirements - writing software that is robust, while at the same time not over-engineering a certain aspect or over-committing to a certain approach only to find later that it was not ideal.

A commonly used quote among programmers, by Donald Knuth, is that "premature optimization is the root of all evil." This quote aims to encourage simplicity of a sort, and so there is an irony in dissecting it for its details; but there is a nuanced approach nonetheless. I'll make the argument both for and against the spirit of this quote, while ultimately settling on the need for a balanced take on its interpretation. But I think the journey to this conclusion can yield some fruitful insights.

Often times, I think this quote gets used as an excuse to write poorly organized code. While functionality is the primary goal, a system that merely 'works' is often far more fragile than we'd like to admit. However, for something to be in a state of "working" is often more fragile than we would like to admit.

In the pursuit to get our digital machine "working", we can, justifiably, pick the simplest path to that destination. We may notice certain patterns and opportunities to "gather supplies" on our journey, but choose to ignore them until we reach our destination. And once the destination is reached, we decide that rather than refining the path that we took, or our "inventory", that we should instead continue on. It is only later in our journey, that we may find ourselves confronted with an obstacle - a metaphorical "river" that we must cross. Perhaps a different path may have avoided us coming to this "river", or perhaps all paths led to this; but, maybe some more careful consideration of "gathering supplies" on our journey may have left us more prepared for what we are confronted with now.

The "path" that we will consider is the gathering of `LandDivision` entities. When first creating the functionality for FieldMind, I chose, justifiably, to focus only on a single year's worth of data. This simplified the approach and allowed me to implement functionality for entering in data which would **allow** for viewing over a multiple-year range.

While implementing the necessary endpoints, I noticed that there were recurring instances in the controller in which I would retrieve all `LandDivisions`. The resultant code was something like this:

```
var landDivisions = await _context.LandDivisions
  .Where(landDiv => landDiv.OwnedByFarmId == farmId)
  .Include(landDiv => landDiv.Subdivisions)
  .Include(landDiv => landDiv.LandDivisionType)
  .Include(landDiv => landDiv.SoilType)
  .Include(landDiv => landDiv.SizeUnit)
  .ToListAsync();
```

Listing 7. C# Query for Retrieving Land Divisions

Seeing this, there was a desire in me to create a method in the `LandDivisionService` which **could** "get all land divisions" for me (more on this tendency to pattern-match later). However, I stymied this action in myself at the time, attributing the method creation to a "premature optimization", and that, furthermore, it would provide unnecessary "nesting" of method calls - creating a "Russian doll" kind of situation. After all, these `Include()` methods and database accesses with EF Core are themselves methods encapsulating logic, and are meant to be called when needed. I wanted to be cautious of encapsulating methods inside of methods just for the sake of doing it. I reasoned that each controller can make its call with the EF Core methods shown above and that this is precisely what they are for.

This reasoning is not wrong in that "that is what those methods are for". However, once the job of implementing the functionality of looking at multiple years is turned to, some issues emerge. The issues emerge when we consider the `Where()` method and the inclusion of subdivisions.

When we move from looking at data for the year `2019` to instead look at data for `2022`, there are a number of things that may have changed in that amount of time. We may have created or deleted `LandDivisions` in that time. When viewing the `2022` data, we should only see `LandDivisions` which are in existence at that time - meaning that they were either created before the year that we are looking at and were not deleted; or, if they have been deleted, they are deleted **after** the year that we are looking at. Our `Where()` condition now changes to the following.

```
var landDivisions = await _context.LandDivisions
  .Where(landDiv => landDiv.OwnedByFarmId == farmId
          // Created before or during the plantingYear we are looking at
       && landDiv.CreatedAt.Value.Year <= plantingYear
       // And has not been deleted
       && (!landDiv.DateDeleted.HasValue
       // Or if it has been deleted, it has been deleted after the
          plantingYear we are looking at.
       || landDiv.DateDeleted.Value.Year > plantingYear)
   )
  .Include(landDiv => landDiv.Subdivisions)
  .Include(landDiv => landDiv.LandDivisionType)
  .Include(landDiv => landDiv.SoilType)
  .Include(landDiv => landDiv.SizeUnit)
  .ToListAsync();
```

Listing 8. C# Query for Retrieving Land Divisions with Filtering

You can notice also that the `Include(landDiv => landDivSubdivisions)` has also been highlighted for consideration as well. And indeed, our collection of subdivisions in the way the code above was written presents the same issues that we have aimed to correct with our `Where()` clause. The issue is the same - we **do not** want every subdivision associated with a given `LandDivision`, but instead we want only the subdivisions which were in existence during the year in which we are looking at.

And so now we have an issue which rears its head for single years, but also will roar just as loudly, but in some unique ways of its own, when we look at `LandDivisions` over a range of years as well. At this point, we can appreciate our need for the additional inventory of specialized methods to retrieve this data and allow us to be prepared to cross the "river" which is before us. This "additional inventory" is needed as going through all of our endpoints and implementing this fix would not only be tedious, but it is very error-prone in the action itself, and it also increases the likelihood of writing code which is inconsistent as more endpoints are written. This inconsistent code will lead to bugs which will appear in only circumstances which use the code paths calling the methods which contain the errors from the copy-pasting or re-implementation of the fix multiple times (one for each endpoint).

Due to the need of the reuse for this functionality combined with its critical role of being foundational and depended upon to retrieve the correct data in order for subsequent functionality to work, refactoring this code to be more modular is well-justified.

The code would be well-adjusted to be broken down in the following way: One method each for getting `LandDivisions` for a specific year, and one for getting `LandDivisions` for a range of years. We will also create the same for subdivisions and for root `LandDivisions` specifically.

```
GetAllLandDivisionsForYear(int farmId, int plantingYear)

GetAllLandDivisionsForYearRange(
    int farmId,
    int startPlantingYear,
    int endPlantingYear
)

GetAllRootLandDivisionsForYear(int farmId, int plantingYear)

GetAllSubdivisionsForYear(
    int farmId,
    int parentLandDivisionId,
    int plantingYear
)

GetAllSubdivisionsForYearRange(
    int farmId,
    int parentLandDivisionId,
    int startPlantingYear,
    int endPlantingYear
)
```

Listing 9. C# Method Signatures for Retrieving Land Divisions

Furthermore, we will create the following methods, and inside the implementation of the each of the above, we will call the respective following method which will handle the creation of the appropriate predicate.

```
AllLandDivisionsForYearPredictate(int farmId, int plantingYear)

AllLandDivisionsForYearRangePredictate(
    int farmId,
    int startPlantingYear,
    int endPlantingYear
)

AllRootLandDivisionsForYearPredictate(int farmId, int plantingYear)

AllSubivisionsForYearPredictate(
    int farmId,
    int parentLandDivisionId,
    int plantingYear
)

AllSubivisionsForYearRangePredictate(
    int farmId,
    int parentLandDivisionId,
    int startPlantingYear,
    int endPlantingYear
)
```

Listing 10. C# Predicate Methods for Filtering Land Divisions

The reason for the breakdown of not just the gathering of the collection of relevant `LandDivisions`, but also the predicate used to gather them is twofold - there are further complexities which are needed when gathering collections of `LandDivisions`, and it is advantageous to segregate this with the pivotal functionality of crafting the appropriate predicate for gathering what we need. Creating these distinct methods also allows us to be better prepared for a future "river" we may encounter where we need to gather a collection of `LandDivisions` but under a different predicate. In this way, we have a centralized, streamlined approach to adjust what needs to be adjusted while not affecting the other pieces of the code which will remain the same aside from the change in the predicate.

This is an example where the path to getting things "working" quickly actually slowed down the process, and in fact made the "working" state far more fragile of an element than it would have otherwise been had more attention been paid to the pattern which was revealing itself.

And this brings about an important point in which the "premature optimization is the root of all evil" seeks to illuminate: Sometimes, you are not going to know ***the path*** until you start walking **a path**.

In order to describe a pattern, that pattern must first reveal itself. For example, if I were to present to you a sequence which starts with the following numbers: `2, 1`, and then asked you to tell me the next number in the sequence, this is not something that you would likely be able to do with confidence as a pattern has not made itself readily manifest - at least not without some context.

If I were to first give you the pattern `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...` , I think you would see the pattern that we add the `0` and `1` to get the next number in the sequence, and then add that number, `1`, with the previous number in the sequence, `1`, to get the next number, `2`, and that we continue on in that fashion adding the previous number to the most recently generated one, for as long as we like. You may even recognize this sequence as the famous Fibonacci Sequence.

If I were to again present to you the numbers `2, 1` and ask you the next number, I think you may now be inclined to say `3`. And that the next number after that would be `4`, and then `7`, and so on - continuing on in this same fashion as before. And indeed you would be correct in this pattern that I have in mind - the Lucas Sequence.

This example serves as an analogy to the insight that comes with programming experience. When one is presented with the initial numbers of a sequence such as `2, 1`, someone could could give any multitude of defensible answers for the next number in the sequence. This is analogous to being confronted with a programming problem for the first time. Ruminating on trying to find "the" answer can stand in the way of the achievement of this goal in the first place. This is part of what is meant by "premature optimization is the root of all evil." No matter how long you sit and ponder the next number from `2, 1`, at the end of the day, if this is your first encounter of a problem like this, it will serve you better to just pick a path which you think is best, and proceed with that and see how your choice lines up with what you find out is **actually** the next element in the sequence. The key thing here is that the proper adjustments are made after your knowledge increases.

Once the pattern manifests, whether in the problem you find yourself currently tackling or in a similar one, the next time you encounter either the same problem, or a derivation of it, **now** is the time where it would be the more strategic move to implement your lessons learned from the past and to make your go at readying yourself to walk **the path** rather than merely trudging ahead, full brunt towards **a path**. You now have more context of what is likely to lie ahead, and at this point to neglect in making the proper preparations **may** be ill-informed to do.

To make the counter argument to walking **the path**... sometimes even **the path** is not the **right path**. Let's dive into this. When building anything, and particularly when looking at things from an engineering perspective, everything has a tradeoff. This is where engineering experience and insight can lend itself to another approach to the kinds of problems presented above.

When readying yourself for the journey ahead, you may know that you are going to approach a "river", but perhaps instead of building a sturdy bridge to cross it, you know that a rope bridge will do for your case. You're confident of this **because** of your experience, and you are fully aware that this solution is not going to be one that necessarily "scales." However, it will hasten the journey, and you therefore opt for this.

The cliche of "it is the journey, not the destination" is something that applies here with our decision of approach. The tradeoff with walking **the path** and building that "bridge" which you know is scalable, is that this does require more upfront cost in time and resources which must be immediately paid. Depending on the circumstance you find yourself in, paying this cost at the beginning may not be ideal. Perhaps you need to quickly get the project to a workable state for a demo or some kind of launch. Or perhaps, this is a project which is not meant to last for a long time as a long-term project. These are cases in which **the path** is not necessarily the **right path**. In either case, there is technical debt which can be paid immediately through creating a system which is more robust, or it can be paid later in favor of immediate speed. The key thing to note is that at some point, if the project persists, that technical debt **must** be paid. It is just a matter of **when**.

13.2. More Can Be Less

A mention was made earlier in our exploration of efficiency about pattern-matching. As a programmer, one has a tendency to look for, and see patterns, and seek to find a way to leverage that pattern into some kind of structure. This structure can manifest itself by way of methods, classes, interfaces, etc., as well as design patterns like the factory pattern or others. In general, this tendency will lead us to writing higher quality software. However, this tendency can sometimes lead us to over-correcting or "correcting" too early. These can be made in an attempt to create more streamlined software, but this can easily turn on itself.

Let us consider an example of the following `FormatDataForStackedBarChart` method. The important piece for this example is the `Dictionary` value for the return type, and the `Dictionary` for the `data` parameter of the method. The value will be a `List` which is either a list of nullable integers (`List<int?>`) or a list of nullable decimal values (`List<decimal?>`). Both the return type and the parameter type will match—so if the return type is a `List<int?>`, then the parameter type will also be a `List<int?>`. Other than this, the method's implementation is **exactly the same**.

Because the implementation is exactly the same, the idea of copy and pasting one implementation just to change the list's data type can scream to a programmer's mind to refactor this method to perhaps accept a generic type for the list as `List<T?>` and thus we can save the coding real estate and just use the one method in both cases because no implementation details change. However, due to the requirements of the chart building, the only data types that we can accept for `T` are either an `int` or a `decimal`. This will lead us to impose some restrictions on `T` in our method implementation, as well as a runtime check to specifically verify that `T` is either an `int` or a `decimal`. This would lead us to something like the following:

```
private Dictionary<string, List<T?>> FormatDataForStackedChart<T>(
    Dictionary<string, List<T?>> data,
    Dictionary<string, Dictionary<string, T>> landDivDictionary
)
where T : struct, IComparable, IConvertible, IComparable<T>, IEquatable<T>
{

    if (typeof(T) != typeof(int) && typeof(T) != typeof(decimal)) {
        throw new InvalidOperationException("Only int and decimal are
            supported.");
    }

    // Other logic here
}
```

Listing 11. Generic Method for Formatting Data in a Stacked Chart

We can see here that in our attempt to simplify, we have instead written something complicated. Though we may have a single method which can be used in either case for the data types aforementioned, we are far better off overloading the method than writing this generic method. We can see the overloaded `FormatDataForStackedChart` method below.

```
private Dictionary<string, List<decimal?>> FormatDataForStackedChart(
        Dictionary<string, List<decimal?>> data,
        Dictionary<string, Dictionary<string, decimal>> landDivDictionary
    ) {

        foreach (string name in data.Keys) {

            foreach (
                KeyValuePair<string, Dictionary<string, decimal>> item
                in landDivDictionary
            ) {
                var itemsForLandDivDictionary = item.Value;

                if (
                    itemsForLandDivDictionary.TryGetValue(name, out
                        decimal value)
                ) {
                    data[name].Add(value);

                } else {
                    // insert null
                    data[name].Add(null);
                }
            }
        }

        return data;
    }
```

Listing 12. Method for Formatting Data in a Stacked Chart (Decimal Version)

```
private Dictionary<string, List<int?>> FormatDataForStackedChart(
        Dictionary<string, List<int?>> data,
        Dictionary<string, Dictionary<string, int>> landDivDictionary
    ) {

        foreach (string name in data.Keys) {

            foreach (
                KeyValuePair<string, Dictionary<string, int>> item
                in landDivDictionary
            ) {
                var itemsForLandDivDictionary = item.Value;

                if (
                    itemsForLandDivDictionary.TryGetValue(name, out int
                       value)
                ) {
                    data[name].Add(value);

                } else {
                    // insert null
                    data[name].Add(null);
                }
            }
        }

        return data;
    }
```

Listing 13. Method for Formatting Data in a Stacked Chart (Integer Version)

Yes, there is some repeated code above, but, think of all of the code that is happening under the hood to enforce those checks for the generic method implementation we had before. And also think of how unreadable the generic version of the method is - you have to contemplate for a bit to decipher what it wants. And ultimately, if you give it something that is not an `int` or a `decimal`, you will have a **runtime** error. Whereas with the overloading of the method, you get a much preferred **compilation** error and a far more direct message of what is wrong. Additionally, the overloading approach is far more readable - it does not require the pondering and deciphering that the generic implementation of the method requires. Because of this, the overloaded method is more maintainable, and is safer from bugs due to its being more understandable.

There is an irony in the pursuit to reduce lines of code, sometimes we can leverage **more** code and resources due to the underlying features that we use. And just because we reduce the lines of code does not necessarily mean that we have increased efficiency either. Efficiency can take a number of forms in addition to the number of lines of code. Sometimes more is less.

In total, what we ultimately want to build is a digital machine that will suit our needs. In order to do this, we leverage (and build) our skills in order to create products which better suit the needs of the circumstance that we find ourselves in. Looking at projects as iterative, growing things - like a seed which grows into a tree - can help to inform these decisions. A certain stage in the "growth" may necessitate certain approaches for that stage.

Throughout the building process, I think that it is important to build the best product you can for the moment in time that you are in, given all contexts. If what is called for happens to be a "rope bridge" rather than a fortified "concrete" one, then one should make that rope bridge have the best knots that you can and not neglect the craftsmanship that goes into every project. I think that the signs of a good engineer should show through whether it is a "rope bridge" or a "concrete bridge".

13.3. Thoughts on Use of Internet and AI for Research

At a time where people boast about the availability of information on the Internet and the proliferation of various artificial intelligence (AI) tools, it is important to be mindful that these resources are only as useful as what a **person** has made available at some point on the Internet. There are certain topics and certain industries which are more opaque than others. For example, many farmers are not necessarily blogging or writing articles about their day-to-day operations and struggles. Therefore, there is less information on the Internet as a result of this.

Even information that is found on the web, does not necessarily map well to what is true for the day-to-day operations of what, in my case, a Midwestern crop farmer is going to be doing. To find out what common methods and concerns are, I think that you do need a consultant who is in the industry to relay that information to you. In one way or another, the information needs to be conveyed to you. With Internet research, that can get you to a certain level, but it may leave some dots undiscovered or unconnected. You could be missing some very important context. It is not just about seeing the dots, but it is about connecting them. And sometimes, you may not even know the right questions to ask at the outset.

As much as the impression can be given that we all can just retreat to our "cave", and with only an electronic device and connection to the Internet be in touch with the world, I do not think that this is true. Ultimately, that human interaction, and content which is generated ultimately by humans, is not so easily substituted or replaced. There can be a lot of pride garnered from doing research on your own via the Internet or books. Whether you are reading information on the Internet or from books, you are receiving information from an expert in some manner.

For FieldMind, there were questions which would have been very difficult to articulate to an Internet search to get meaningful results, because what I was aiming to get was some very specific information or feedback for the process in which crops would be planted, a harvest would be executed, or yield would be calculated. For example, the information which gets put onto a scale ticket and how that information will ultimately relate back to the field's yield statistics would have been difficult to gather. This would have been difficult to gather because though I'm sure that I could have found something indicative of a scale ticket, to further **verify** that what I found through Internet searches was the common format and information for scale tickets for grain crops throughout the Midwest would have been a bit on shaky ground. Having the direct vindication from a farmer is really what was needed.

Furthermore, when looking into some various other ways that yield is calculated for specific types of corn or soybeans, I had asked ChatGPT which types of corn use the standard weight of 56 lbs / bushel. I was given an answer for this and asked Glenn about the list that was created - which also included some descriptions of the corn type. Some of the items on the list were correct, but others were specialty crops which are not often planted, and when they are planted, the yield is not calculated in bushels but rather by the ton for some of these. Others on the list Glenn did not have any knowledge of, and even when he'd done some research attempting to find what measurement units were used and the harvest details, he struggled to find that information. I think this speaks volumes about the limits of Internet research on certain topics: being that Glenn is an experienced farmer, he is also aware of **where** the official sources of some of this information is, and he still came back empty-handed.

There is one other issue that comes with finding this information as well for an outsider to the industry like me - knowing **where** the official sources for this information are. Additionally, again, even knowing **what** information is crucial to know and include (the "questions to ask") is not always clear. For certain industries the "how-to" tutorials are not out there. Instead prerequisite knowledge is required.

Though undoubtedly the Internet combined with tools like AI are powerful and open up access to information not previously accessible, there is still the challenge of verifying that the information that is gathered is actually representative of the current reality. One must also find the hubs of information which are used by the industry as well in an official manner, and navigate all of the various intricacies of the industry as well.