TerraByte

System Requirements Specification (SRS)

A project for EPI·USE by GeekGurusUnion

Demo 3

12 August 2024

System Requirements Specification (SRS)

Introduction

TerraByte is poised to lead the digital transformation within the South African farming industry by harnessing the power of artificial intelligence and data analytics. Our mission is to turn inefficiency into precision, empowering farmers with the tools they need to optimise their operations and maximise their yields.

Problem Statement

Every harvesting season, farmers go through a lot of stress about their crops, will they harvest enough? Are their crops healthy? They might take soil moisture samples, and water the crops that same day, but the next day it rains, so the crops get overfed. What if they can minimise the risk to farm more precisely? What if they can be provided with a user-friendly interface that gives actionable statistics and smart recommendations?

How will we solve this?

At its core, TerraByte aims to bridge the gap between traditional farming practices and a future where crop production is driven by data and predictive modeling. This initiative involves several key components:

1. Data Collection: We gather extensive data on weather patterns, soil quality, crop varieties, and historical yields. This data forms the backbone of our predictive analytics model.

2. Machine Learning Model: Our sophisticated AI model (will) analyse this data to predict future crop health and yield with remarkable accuracy. This model continuously learns and adapts, improving its predictions over time.
3. User-Friendly Interface & Backend: Farmers can easily input their farm-specific data through an intuitive interface. Meanwhile, a robust backend system ensures secure data processing and storage.

The AI Crop Prediction system represents a significant leap forward in agricultural technology for South Africa. By leveraging AI, this system enables farmers to make informed, data-driven decisions, minimising risk and enhancing efficiency and profitability.

User Stories & User Characteristics

User Story 1: Farmer

As a farmer, I want to enter my farm's data into the system so I can get accurate crop yield predictions. I also want to be ensured that the system handles the data that I am unable to collect in a timely manner.

Description: The farmer will enter information like soil quality, crop type, planting dates, and other details into the system. He will then receive daily predictions on his yield as well as actionable insights on his crops.

Acceptance Criteria:

1. The system should have an easy-to-use form for entering data.
2. The system should check the data to make sure it's complete and correct.
3. The system should store the data safely in the backend database.
4. After submitting, the system should confirm it received the data and start the prediction process.

User Story 2: Farm Manager

As a farm manager, I want to see detailed reports on predicted crop yields so I can plan resources and operations well.

Description: The farm manager will use the system to see data and reports on crop health and yield.

Acceptance Criteria:

1. The reports should be easy to access through a user-friendly dashboard.
2. The reports should include visual aids like graphs and charts for easy understanding.

3. The home page should be updated with the latest data.

User Story 3: Data Analyst

As a data analyst, I want to download CSV-files and analyse it to give additional insights on top of the ML-model implemented.

Description: The data analyst will use the system to see the current data logged and analyse the results to give actionable insights to the farm manager.

Acceptance Criteria:

1. The reports should be downloadable on selection from the dashboard
2. The data analyst should be able to see all the graphs and data related to the field

Functional Requirements

Requirements

* denotes future/currently partial implementation.

1. Field Management
    - 1.1. Manual Data Input
        - 1.1.1. Users should be able to manually edit recorded data related to crops.
        - 1.1.2. System will store data securely.
    - 1.2. CRUD Operations
        - 1.2.1. Users should be able to update data.
    - 1.3. Data Visualisation
        - 1.3.1. Data should be represented to the user with short explanations and relevant graphs.
    - 1.4. View Field Data
        - 1.4.1. Map-based: A clickable, drawable, and colour-coded map should show the user's current farms.
        - 1.4.2. View Logs: Users should be able to view past entered data and be able to correct them if necessary.
        - 1.4.3. Export/Print Logs: Users should be able to select entries and print or export the data to be analysed on another platform.
2. User Management
    - 2.1. Map-based Interface
        - 2.1.1. User location should be retrieved to show their current location.
        - 2.1.2. The map should show a farmer's current fields colour-coded with their respective health data.

2.2. User Accounts and Authentication

2.2.1. Users should be able to log in using their credentials or via OAuth.

2.2.2. Users should be able to reset their password via email.

2.2.3. Users should receive an email upon account creation to activate their account.

2.3. CRUD operations

2.3.1. Users should be able to update their account details.

2.3.2. Farm Managers should be able to add team members.*

3. Crop Prediction Model Subsystem

3.1. Model training

3.1.1. Generic Model: A generic ML model should be able to be trained on historical data either provided from APIs or manual uploads.

3.1.2. Personalised Model: from the generic model, a model is trained on every field individually to have personalised and accurate predictions.

3.2. Model updating

3.2.1. Models should be able to be updated and re-deployed to a user to ensure up-to-date predictions.

3.2.2. Farmers should have the choice to retrain their personalised model per upload.

3.2.3. Using automated jobs (CRON jobs), every field's model will be retrained weekly in order to keep the predictions relevant to the current field situation.

3.3. Weather Data Improvement

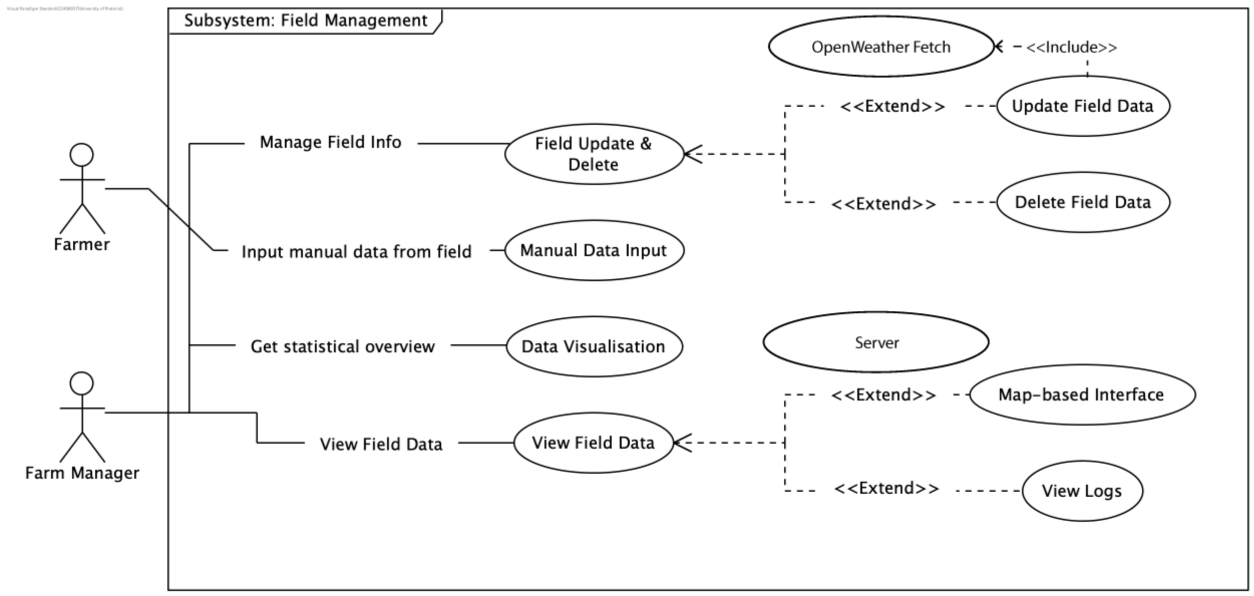
3.3.1. Using sensors, such as an anemometer and a thermometer, actual data can be compared against an external weather API service to provide more accurate and personalised weather predictions.*

3.3.2. Using automated jobs (CRON jobs), weather will be fetched daily in order to ensure the user receives up to date information on their fields.

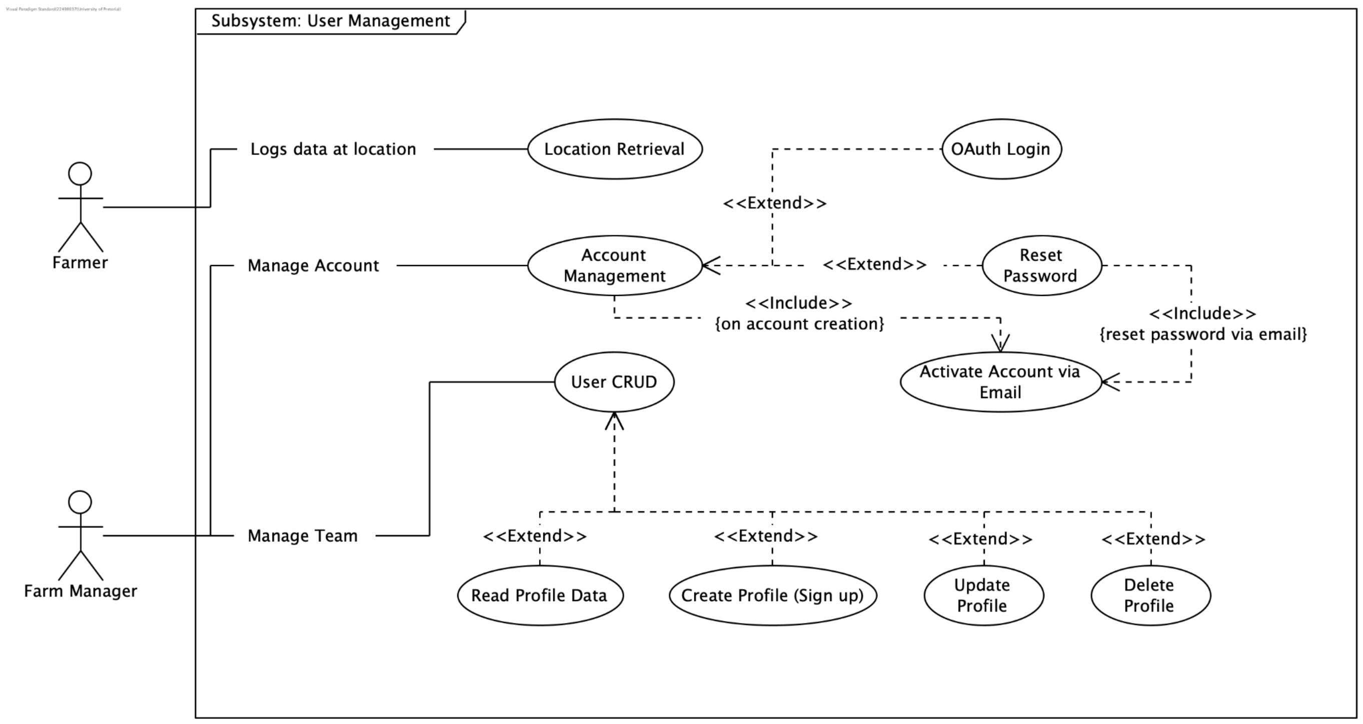
3.3.3. Weather is fetched from an external weather API.

Use Case Diagrams

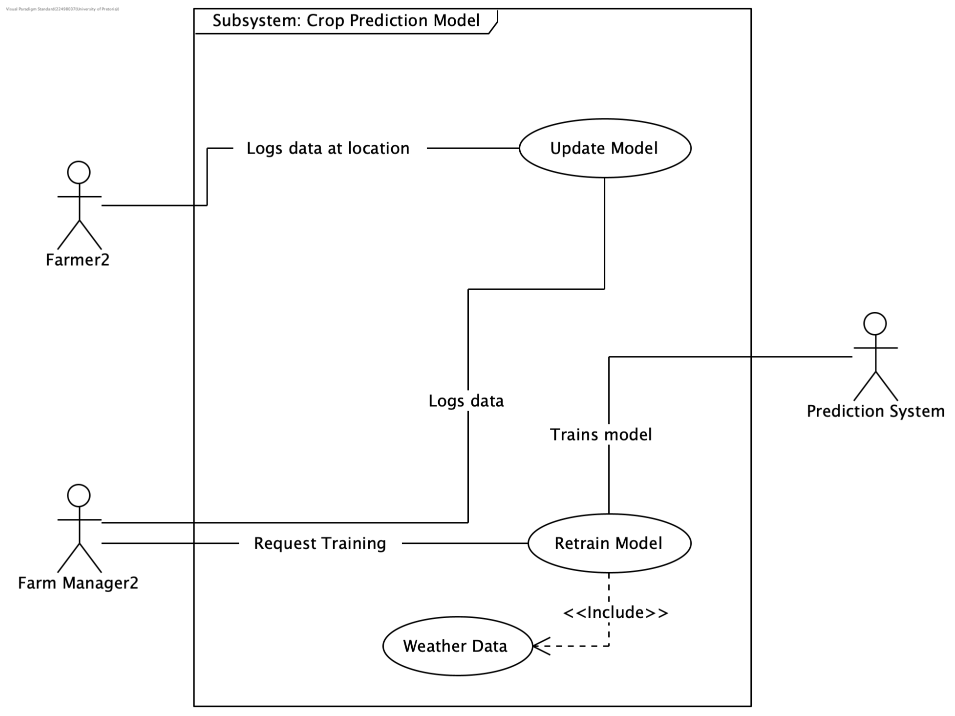
Field Management Subsystem



User Management Subsystem



Crop Prediction Model Subsystem



Architectural Requirements

Quality Requirements

1. Performance
    1.1. Performance is considered a crucial underlying factor for our system, highlighting the importance of processing power and all-round data analysis capabilities. On the frontend side, effective and rapid rendering is crucial for real time statistics provided to the end user, while being backed by a solid backend system with a rugged API.
    1.2. Quantification
        1.2.1. ML model training time should be less than 5 minutes.
        1.2.2. App loading time should be less than 10 seconds.
        1.2.3. API response time should be less than 5 seconds.
2. Testability
    2.1. Our system's testability measures our test coverage and the ability to place a clear distinction between failing and passing units in our code. This will be required to provide a robust system that works under various circumstances.
    2.2. Quantification
        2.2.1. Presentation Layer: Make use of Vitest to write unit tests for our frontend.
        2.2.2. Logic Layer: Make use of PyTest to write unit tests for our backend.

        - 2.2.3. Data Layer: Make use of PyTest to test the database operations and to ensure consistency in the database to serve accurate and representative data.
        - 2.2.4. Between layers: Make use of Vitest and Postman to test our interactions between layers.
        - 2.2.5. System-wide: Make use of Cypress to write end-to-end tests simulating system flow.
        - 2.2.6. 60% coverage overall on Codecov.

3. Efficiency
    - 3.1. Because we run sophisticated services in the background on a constrained budget, having a system that avoids wasting efforts, money, and time (i.e. resources) while performing a specific task is crucial.
    - 3.2. Quantification
        - 3.2.1. Graphs provided by our Digital Ocean droplet dashboard (budget, resource utilisation and response time)
        - 3.2.2. Timing function calls and output for our ML model.*
        - 3.2.3. Reporting analytics to a CSV file to analyse bottlenecks.*
4. Compatibility
    - 4.1. The app should be designed for easy installation on a variety of devices commonly used by farmers. The app should be configured for automatic updates over WiFi whenever possible.
    - 4.2. Quantification
        - 4.2.1. Make use of a PWA to ensure offline capability.*
        - 4.2.2. Provide a consistent user interface using component libraries.
        - 4.2.3. Fetching new data when connected to the internet by setting up runners that will periodically check for internet connection.*
        - 4.2.4. Saving new data temporarily while the app is not connected to the internet by making use of local storage to temporarily store values while not connected to the internet.*
5. Usability
    - 5.1. A system designed to help farmers or agricultural professionals predict the yield and potential success of their crops based on various factors like weather, soil conditions, and historical data. Usability refers to how easy and enjoyable it is for users (in this case, farmers or agricultural professionals) to use the system.
    - 5.2. Quantification
        - 5.2.1. The system should help users achieve their goals (e.g., accurately predicting crop yields) by
            - 5.2.1.1. Providing a clean, intuitive interface with user-friendly navigation.
            - 5.2.1.2. Ensure consistency in the design by using established design systems and component libraries.

5.2.2. The system should be easy and quick to use, allowing users to complete tasks without unnecessary effort or time by providing an easy-to-access help menu within the app for users to get help when needed.*

5.2.3. The system should be user-friendly and enjoyable to interact with, leaving the users feeling satisfied with their experience by

5.2.3.1. Simplify data entry and interaction processes to minimize user effort.

5.2.3.2. Provide clear, concise instructions and tooltips to guide users through complex tasks.

Service Contracts

Field Management Service Contract

Description: This service handles all operations related to field management including manual data input, data visualization, and CRUD operations.

1. InputFieldData: Allows users to manually input data related to crops, weather conditions, and other relevant factors.
    - Input: FieldData (includes crop type, soil quality, and/or weather conditions)
    - Output: Acknowledgement (user alert)
2. UpdateFieldData: Allows users to update existing field data.
    - Input: FieldDataID, UpdatedFieldData
    - Output: Acknowledgement
3. GetFieldData: Retrieves field data for a given field.
    - Input: FieldDataID
    - Output: FieldData
4. DeleteFieldData: Deletes field data for a given field.
    - Input: FieldDataID
    - Output: Acknowledgement
5. ViewFieldData: Provides a visual representation of field data on a map.
    - Input: UserID
    - Output: MapData (includes color-coded and vectorized field data)
6. ViewLogs: Allows users to view and correct past entered data.
    - Input: UserID
    - Output: FieldDataLogs

User Management Service Contract

This service handles user account operations, including authentication, CRUD operations on user accounts, and managing user sessions.

1. CreateUserAccount: Creates a new user account.
    - Input: UserAccountData (includes email, password, personal details)
    - Output: Acknowledgement (confirmation of account creation)
2. AuthenticateUser: Authenticates a user with their credentials.
    - Input: Email, Password
    - Output: AuthenticationToken
3. ResetPassword: Allows users to reset their password via email.
    - Input: Email
    - Output: Acknowledgement (confirmation of password reset email sent)
4. UpdateUserAccount: Updates user account details.
    - Input: UserID, UpdatedUserAccountData
    - Output: Acknowledgement
5. GetUserAccount: Retrieves user account details.
    - Input: UserID
    - Output: UserAccountData
6. DeleteUserAccount: Deletes a user account.
    - Input: UserID
    - Output: Acknowledgement

Crop Prediction Model Subsystem Contract

This service handles operations related to the training, updating, storage, and utilization of crop prediction models. It includes functionalities for managing both generic and personalized models, integrating historical and weather data, and ensuring data security.

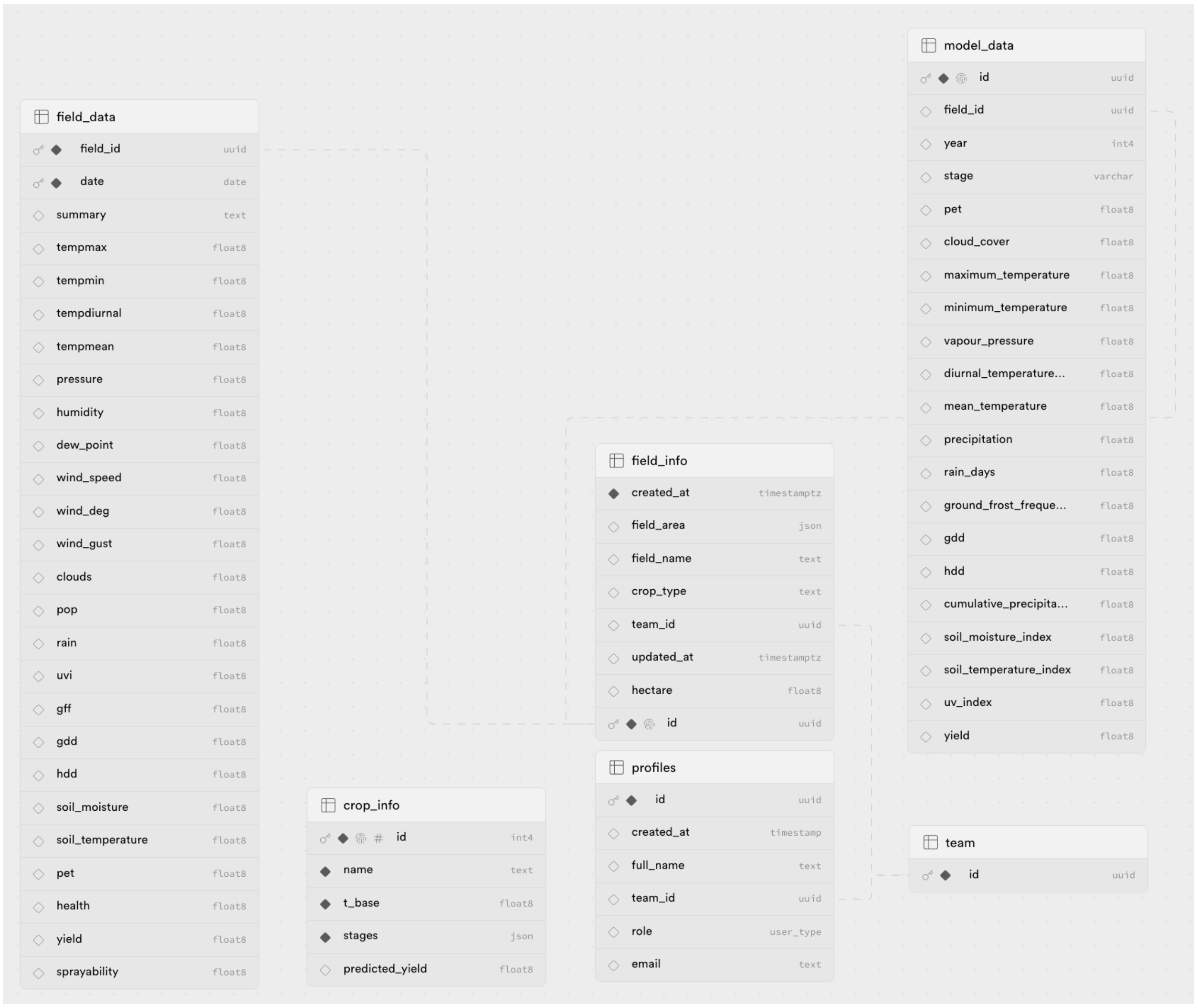
Operations:

1. TrainGenericModel: Trains a generic crop prediction model using historical data.
    - Input: HistoricalData
    - Output: ModelTrainingAcknowledgement (confirmation of model training completion)
2. TrainPersonalisedModel: Trains a personalized crop prediction model for a specific field using the generic model as a base.
    - Input: UserID, PersonalisedTrainingData
    - Output: ModelTrainingAcknowledgement (confirmation of model training completion)
3. RetrainModel: Updates and redeploys a crop prediction model (either generic or personalized).
    - Input: ModelID, UpdatedModelData
    - Output: ModelUpdateAcknowledgement (confirmation of model update and redeployment)

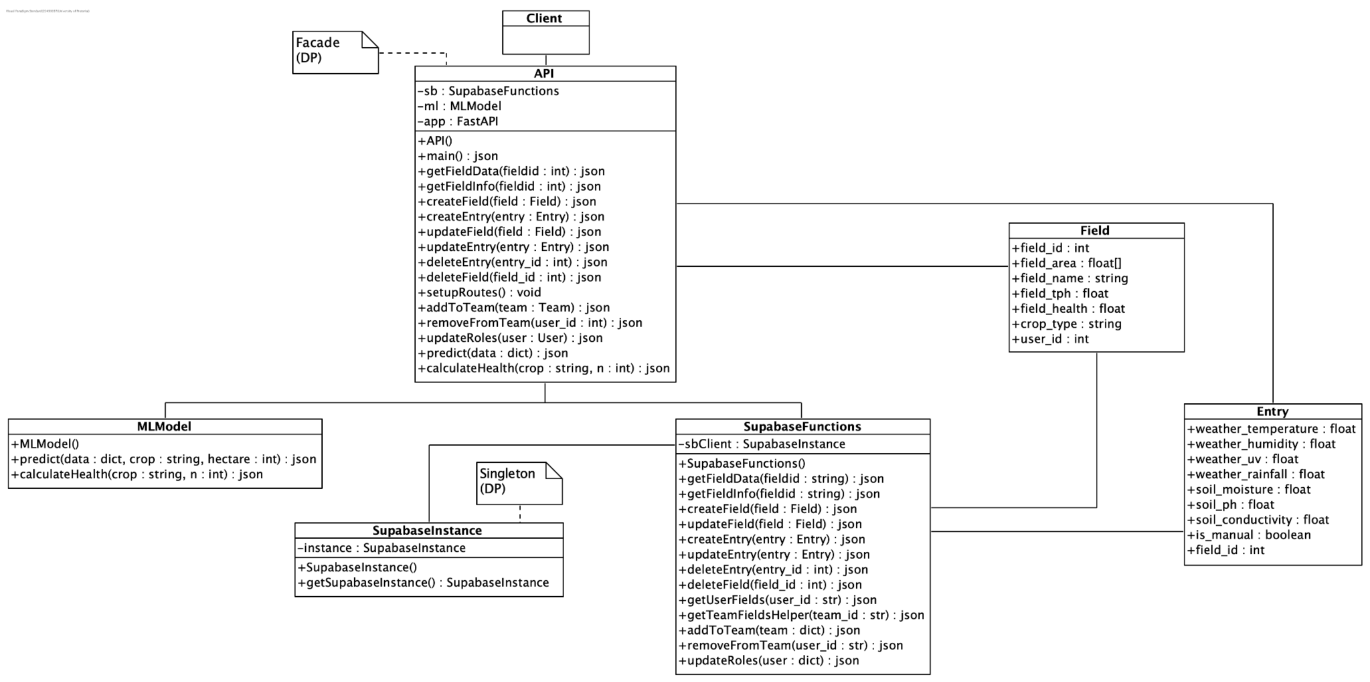
4. IntegrateWeatherData: Integrates sensor data with external weather API data to improve weather predictions.
    - Input: SensorData, ExternalWeatherAPIData
    - Output: WeatherDataIntegrationAcknowledgement (confirmation of data integration)
5. GetWeatherPrediction: Provides weather predictions based on integrated data.
    - Input: Location, DateRange
    - Output: WeatherPredictionData

Class Diagrams

Database Entity Relationship Diagram



Backend Class Diagram


Facade (DP)
Client
API
-sb : SupabaseFunctions
-ml : MLModel
-app : FastAPI
+API()
+main() : json
+getFieldData(fieldid : int) : json
+getFieldInfo(fieldid : int) : json
+createField(field : Field) : json
+createEntry(entry : Entry) : json
+updateField(field : Field) : json
+updateEntry(entry : Entry) : json
+deleteEntry(entry_id : int) : json
+deleteField(field_id : int) : json
+setupRoutes() : void
+addToTeam(team : Team) : json
+removeFromTeam(user_id : int) : json
+updateRoles(user : User) : json
+predict(data : dict) : json
+calculateHealth(crop : string, n : int) : json
Field
+field_id : int
+field_area : float[]
+field_name : string
+field_tph : float
+field_health : float
+crop_type : string
+user_id : int
Entry
+weather_temperature : float
+weather_humidity : float
+weather_uv : float
+weather_rainfall : float
+soil_moisture : float
+soil_ph : float
+soil_conductivity : float
+is_manual : boolean
+field_id : int
MLModel
+MLModel()
+predict(data : dict, crop : string, hectare : int) : json
+calculateHealth(crop : string, n : int) : json
Singleton (DP)
SupabaseInstance
-instance : SupabaseInstance
+SupabaseInstance()
+getSupabaseInstance() : SupabaseInstance
SupabaseFunctions
-sbClient : SupabaseInstance
+SupabaseFunctions()
+getFieldData(fieldid : string) : json
+getFieldInfo(fieldid : string) : json
+createField(field : Field) : json
+updateField(field : Field) : json
+createEntry(entry : Entry) : json
+updateEntry(entry : Entry) : json
+deleteEntry(entry_id : int) : json
+deleteField(field_id : int) : json
+getUserFields(user_id : str) : json
+getTeamFieldsHelper(team_id : str) : json
+addToTeam(team : dict) : json
+removeFromTeam(user_id : str) : json
+updateRoles(user : dict) : json

Design Patterns

Singleton

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. We utilised this pattern to manage global resources or configurations, such as a centralised connection to the database.

Benefits

- Global Access
- Centralised Management
- Resource Management
- Consistency (all components access the same instance of the singleton class)

Factors worth considering

- Alterations to the singleton object not possible (e.g. different parameters).

Facade

With our current architecture, we ensure a single entrypoint to our (backend) system. This ensures that we connect a simple interface with a complex interface (*Facade*, n.d.) without exposing or compromising underlying functionality.

Benefits

- Simplified Interface
- Decoupling (Clients only interact with the facade and are unaware of the internal workings of the subsystems)

Factors worth considering

- Maintenance Overhead (Changes to subsystems may require corresponding changes to the facade)
- Performance Overhead (Additional layer of abstraction may incur a slight performance overhead)

Chain of Responsibility with Pipe-and-Filter Pattern

In our current architecture, we implement the Chain of Responsibility pattern combined with the Pipe-and-Filter pattern. This approach allows for a series of processing steps to be applied to incoming data in a sequential manner. Each step, or filter, is responsible for a specific task, such as fetching data, cleaning it, or performing feature engineering. The Chain of Responsibility ensures that each step operates on the data in a pipeline, enabling a clear and manageable flow of data processing.

Benefits

1. Modularity promotes separation of concerns.
2. Extensibility allows new processing steps that can be added to the pipeline with minimal disruption to existing functionality.

Factors Worth Considering

1. Maintenance Overhead (changes in the data processing requirements may necessitate modifications to the filters and their sequence).
2. Errors encountered in one filter may need to be managed and propagated through the chain, and requires a careful approach to handle errors.

Constraints

Budget Constraints

- Limited financial resources for initial development, deployment, and maintenance.
- Preference for open-source tools and free-tier services to minimise costs.

Hardware Constraints

- Compatibility with common farming hardware, including sensors and IoT devices.

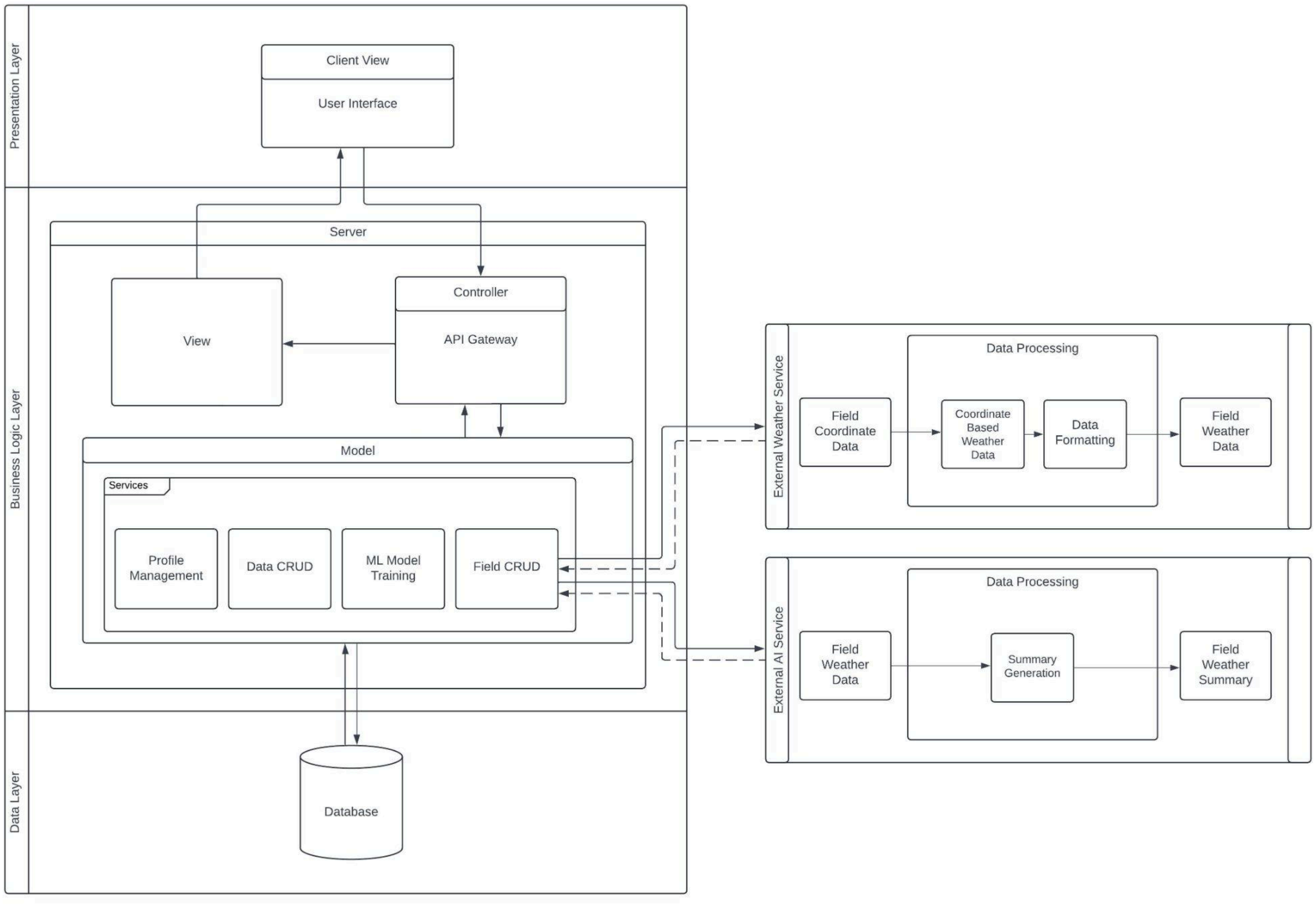
- Limited processing power and memory on devices used by farmers (e.g., older smartphones and tablets).

Network Constraints

- Ensuring functionality in areas with limited or intermittent internet connectivity.
- Efficient data transfer protocols to minimise bandwidth usage.

Architectural Strategy

Architectural Diagram



Architectural Patterns

Layered Pattern

Our project will leverage the use of a layered architecture. We firmly believe that a layered architecture will be the most optimal choice for this project with regards to cost efficiency, system effectiveness, testability, and overall performance. Developing these layers independently provides us with the ability to ship a powerful project without having to compromise on hardware and software limitations related to each layer.

Our Layered Architecture Breakdown:

1. Presentation Layer
    - 1.1. Functionality: This layer contains our Client-View User Interface. It handles user interactions, facilitates user inputs, and communicates with the server-side Controller on the Business logic layer to enable backend functionalities and communicate with the Server Model. This layer ensures a seamless user experience and manages all client-side operations.
2. Business Logic Layer:
    - 2.1. API Gateway Controller:
        - 2.1.1. Functionality: Acts as a single-entry point for the presentation layer client-view to access the business logic layer. The API gateway will be used for external API calls, and routing requests based on functionality to the server model.
    - 2.2. Core Business Logic:
        - 2.2.1. Functionality: Contains the main model application logic services to process user requests, and applies business rules. This includes data processing, computation, CRUD operations, external pipe-and-filter processes, and ML Model training services.
        - 2.2.2. Quantification: The Crop Prediction Model is the main model behind AI Crop Prediction, analysing vast datasets to predict crop yields. It considers weather, soil conditions, and historical data to create personalized forecasts for each farm. These predictions empower farmers to optimise resource use and boost their profitability.
3. Data Layer:
    - 3.1. A managed database will store application data in a centralised location, accessible by the business logic layer for efficient data sharing and consistency. This layer provides data access mechanisms and interacts with the business logic layer to store and retrieve information as needed.

MVC (Model-View-Controller) Pattern

The MVC pattern enhances separation of concerns and modularity within our system by breaking down application functionality into three interconnected components.

1. Model:
    - 1.1. Functionality: The model represents the application's data and encompasses all of the system's core business logic. It processes requests from the Controller, interacts with the respective service, and communicates with the database. The business logic is distributed across four primary services that the model utilizes:
        - 1.1.1. Profile Management Service: Manages user profiles, allowing for the creation, updating, and maintenance of user information.
        - 1.1.2. Data CRUD Service: Handles the creation, reading, updating, and deletion of data within the system.
        - 1.1.3. ML Model Training Service: Facilitates the training and retraining of machine learning models efficiently.
        - 1.1.4. Field CRUD Service: Manages the creation, reading, updating, and deletion of farmer field data. This service also integrates two external services using the pipe-and-filter pattern to collect field data and generate AI-driven field condition summaries.
2. View:
    - 2.1. Functionality: The views in the system facilitate user interactions by sending gestures to the controller and allow the controller to choose the appropriate view to render. The views represent the User Interface (UI) of the system. The system includes two types of views:
        - 2.1.1. Client View: Represents the client-side user interface, responsible for sending user requests to the controller on the server and receiving formatted data from the server view.
        - 2.1.2. Server View: Represents data received from the controller's response after selecting and processing the relevant model service. This view formats the data and sends it to the client view, enabling it to present the information to the users.
3. Controller:
    - 3.1. Functionality: The Controller manages user requests, determines the appropriate model behavior, and updates the views accordingly. It acts as the entry point for the Client View to interact with the server side of the system and communicates with the system view through an API gateway:
        - 3.1.1. API Gateway: The Client View sends requests to the controller via unique endpoints. These endpoints specify which model service should be invoked to either modify or retrieve data. Once the model service processes the request, the data is sent back to the server view for representation.

Pipe-and-Filter Pattern

The Pipe and Filter pattern in the system enables efficient data processing by passing data through a sequence of filters, each performing specific operations. These filters are connected by pipes, which facilitate the flow of data from one filter to the next. This pattern is particularly effective for integrating external services and processing data in a modular and scalable manner.

1. Filters:
    - 1.1. Functionality: Filters are responsible for processing the data received from external services. Each filter performs a specific task, transforming the data into a format suitable for further processing or storage. In our system, we employ the following filters:
        - 1.1.1. Field Coordinate Data: This filter gathers field coordinate data from an external weather service, which is then used to fetch relevant weather data for the field.
        - 1.1.2. Coordinate-Based Weather Data: After obtaining the field coordinates, this filter processes the data to retrieve weather information specific to those coordinates.
        - 1.1.3. Data Formatting: This filter formats the weather data received, ensuring it is structured correctly for storage and further use by other components of the system.
        - 1.1.4. Summary Generation: This filter processes the weather data to generate a summary, which is used to provide AI-driven insights about the field's condition.
2. Pipes:
    - 2.1. Functionality: Pipes serve as the conduits that connect the filters, ensuring that data flows smoothly from one processing stage to the next. In our system, pipes are responsible for:
        - 2.1.1. Data Transfer: Facilitating the movement of data between filters, such as transferring field coordinate data to the weather data processing filter.
        - 2.1.2. Integration with External Services: Managing the interaction between our system and external services, ensuring that data flows into the appropriate filters for processing.

This structured approach allows the system to handle complex data processing tasks with high modularity and scalability, enabling the seamless integration of external services and the efficient handling of large volumes of data.

Client-Server Pattern

The Client-Server pattern in our system establishes a clear division between the client and server components, enabling scalable and efficient communication. This pattern ensures that the client, responsible for the user interface and user interactions, communicates with the server, which processes requests, handles business logic, and manages data.

1. Client:
   - 1.1. Functionality: The client manages the user interface and facilitates user interactions. It serves as the entry point for user requests, which are then sent to the server for processing. In our system, the client is composed of the following:
     - 1.1.1. Client View: This view represents the client-side user interface, where users interact with the system. The Client View sends requests to the API Gateway (which serves as the Controller) on the server and receives formatted data to be displayed to the user. It ensures that users have an intuitive and responsive interface for interaction.
2. Server:
   - 2.1. Functionality: The server processes requests from the client, executes business logic, and interacts with the database and external services. The server component in our system is structured as follows:
     - 2.1.1. API Gateway (Controller): The API Gateway serves as the Controller, managing all incoming requests from the Client View. It routes these requests to the appropriate model services and ensures that the correct business logic is applied. The API Gateway is the main entry point for client requests, dictating which model service should be used to process the data.
     - 2.1.2. Model: The Model encapsulates the business logic and handles data processing. It interacts with the respective services to perform operations based on the request from the Controller. The business logic is split across several key services:
       - 2.1.2.1. Profile Management Service: Manages user profiles, allowing for creation, updating, and maintenance of user information.
       - 2.1.2.2. Data CRUD Service: Handles the creation, reading, updating, and deletion of data within the system.
       - 2.1.2.3. ML Model Training Service: Facilitates the training and retraining of machine learning models.
       - 2.1.2.4. Field CRUD Service: Manages the creation, reading, updating, and deletion of farmer field data, and integrates external services using the Pipe and Filter pattern to process field-related data.
     - 2.1.3. Server View: After processing by the Model, the data is passed to the Controller and then to the Server View, which formats it for transmission

back to the Client View. This ensures that the data is prepared correctly for display to the user.

Deployment Model

In this deployment model, each service is used for a different layer of the application:

- Netlify for the Presentation Layer
- Digital Ocean for the Logic Layer
- Supabase for the Data Layer

This approach leverages the strengths of each platform to build a robust, scalable, and efficient agricultural monitoring system.

Components of the Deployment Model

1. Presentation Layer (Frontend) on Netlify:
    1.1. Hosting: Netlify is used for hosting the static frontend application, built with frameworks like React, Vue.js, or Angular.
    1.2. Benefits:
        1.2.1. Automatic build and deploy from Git repositories.
        1.2.2. Continuous Integration/Continuous Deployment (CI/CD) pipeline.
        1.2.3. Global Content Delivery Network (CDN) for fast load times.
        1.2.4. Custom domain support with SSL/TLS.
2. Logic Layer (Backend) on Digital Ocean:
    2.1. Hosting: Digital Ocean provides scalable infrastructure for deploying the backend services, such as APIs and business logic.
    2.2. Benefits:
        2.2.1. Easy scalability with virtual machines (Droplets) or Kubernetes clusters.
        2.2.2. Load balancing and high availability.
        2.2.3. Flexible environment setup with various operating systems and tools.
        2.2.4. Managed services for simplified infrastructure management.
3. Data Layer on Supabase:
    3.1. Hosting: Supabase is a backend-as-a-service platform providing a scalable and secure PostgreSQL database along with authentication, storage, and real-time capabilities.
    3.2. Benefits:
        3.2.1. Managed PostgreSQL database with automatic backups and scaling.
        3.2.2. Built-in authentication and authorization.
        3.2.3. Real-time subscriptions for live updates.
        3.2.4. Simple integration with frontend and backend.

Technology Requirements

Users are required to have at least:

- A mobile device or desktop/laptop.
- Basic internet connectivity.
- An email that can be accessed and used to log in to the dashboard.

Optional requirements include data measurement instruments to enter manual data.

Live Deployed System

Our application is currently deployed on https://terrabyte.software.
It is set to automatically deploy on a pull request to the main-branch and is hosted via Netlify and monitored by Uptime Robot.

Bibliography

Facade. (n.d.). Refactoring.Guru. Retrieved May 21, 2024, from

https://refactoring.guru/design-patterns/facade