TerraByte

System Requirements Specification (SRS)

A project for EPI-USE® by GeekGurusUnion

Demo 4

30 September 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, 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: Farm Manager

As a farm manager, I want to see detailed reports on predicted crop yields, crop health, and sprayability so I can plan resources and operations well. I also want the ability to manually update data if necessary and manage my team's access to the system.

Description: The farm manager will use the system to access and download reports related to crop health, predicted yield, and sprayability. The system will allow the manager to manually update critical data if needed and manage team members' roles and permissions within the platform.

Acceptance Criteria:

1. **Reports and Dashboard**
   1.1. The system should provide daily data on the dashboard for the field's health, predicted yield, sprayability, and environmental factors.
   1.2. Reports should include visual aids like graphs and charts for easy understanding.
2. **Manual Data Updates**
   2.1. The system should allow the manager to override or correct specific data if needed, ensuring flexibility in managing unforeseen issues or incomplete data.
   2.2. The system should allow the farm manager to manually update data, such as crop type, soil conditions, or other relevant field information.
3. **Field Registration and Updating**
   3.1. The system should allow the farm manager to add and register new fields, entering field-specific details such as name, location, and crop type.
   3.2. The system should allow the farm manager to edit or update field information as necessary.
   3.3. The system should provide an overview of all registered fields and their data.

3.4. The system should validate and confirm field registration, ensuring accurate field tracking.

4. **Team Management**
   4.1. The farm manager should be able to invite new team members by sending email invitations.
   4.2. The system should allow the farm manager to assign roles to team members (e.g., Farmer, Data Analyst).
   4.3. The farm manager should have the ability to modify or revoke team members' access rights based on their role.

User Story 2: 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 the system to handle any missing or incomplete data that I am unable to collect.

Description: The farmer will enter essential information such as soil temperature and moisture, crop type, and other necessary details into the system. The system will also allow the farmer to register and manage fields, track field-specific data, and provide daily predictions on crop yield and actionable insights.

Acceptance Criteria:

1. **Data Entry**
   1.1. The system should have an intuitive and easy-to-use form for data entry.
   1.2. The system should validate the data for completeness and accuracy before submission.
   1.3. The system should store the data safely in the backend database.
   1.4. After submitting, the system should confirm it received the data and prepare the models for the prediction process.
2. **Field Registration and Updating**
   2.1. The system should allow the farmer to add and register new fields, entering details such as the field name, location, and crop type.
   2.2. The system should allow the farmer to edit or update field information when necessary.
   2.3. The system should notify the farmer of successful registration or any errors during the field registration process.

User Story 3: Data Analyst

As a data analyst, I want to access detailed reports, export data, and analyze field and revenue information using various dashboards so I can provide actionable insights based on the farm's performance.

Description: The data analyst will use the system to view and analyze existing field data, revenue-related data, and export logs in CSV format. The data analyst can generate reports that include charts and visualizations for further analysis but does not have permissions to update or create fields.

Acceptance Criteria:

1. **Market Dashboard (Revenue Data)**
   - 1.1. The data analyst should have access to the Market Dashboard to view revenue-related data for various crops.
   - 1.2. The system should display revenue trends, including predicted and past revenue data, through interactive charts.
   - 1.3. The data analyst should be able to analyze revenue trends over time using the provided reports and visual aids.
2. **Field Dashboard (Field-Specific Information)**
   - 2.1. The system should allow the data analyst to view detailed information about specific fields on the Field Dashboard, such as crop health, yield predictions, soil moisture, sprayability, and environmental factors.
   - 2.2. The field data should be presented with interactive graphs and charts for easier analysis and interpretation.
   - 2.3. The data analyst should be able to filter and explore the data for specific fields, but without the ability to edit or update field data.
3. **Log Data (Exporting CSV)**
   - 3.1. The data analyst should be able to export field-related logs and data in CSV format for further external analysis.
   - 3.2. The system should provide filtering options in the Logs Page, allowing the data analyst to select specific time periods, fields, or datasets for export.
   - 3.3. The exported data should include all relevant information for analysis, such as temperature, precipitation, predicted crop yield, field health, sprayability, temperature, soil moisture and temperature, pressure, humidity, dew point, and uv index.
4. **Report Generation**
   - 4.1. The system should allow the data analyst to generate reports that include charts and data visualizations for individual fields.
   - 4.2. The data analyst should be able to generate reports that combine field-specific data and charts, making them available for download.
   - 4.3.
5. **Access Limitation**
   - 5.1. The data analyst should not have the ability to update, create, or delete fields.
   - 5.2. The data analyst's access should be read-only for all field and revenue data, ensuring they can analyze but not modify any data in the system.

Functional Requirements

Requirements

1. Field Management
    - 1.1. Manual Data Update
        - 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. CRON Schedule
        - 1.3.1. OpenWeather API automatically updates and inserts data.
        - 1.3.2. Gemini API creates a summary from the data.
    - 1.4. Data Visualisation
        - 1.4.1. Data should be represented to the user with short explanations and relevant graphs.
    - 1.5. View Field Data
        - 1.5.1. Map-based: A clickable, drawable, and colour-coded map should show the user's current farms.
            - 1.5.1.1. User location should be retrieved to show their current location.
            - 1.5.1.2. The map should show a farmer's current fields colour-coded with their respective health data.
        - 1.5.2. View Logs: Users should be able to view past entered data and be able to correct them if necessary.
        - 1.5.3. Export/Print Logs: Users should be able to select entries and print or export the data to be analysed on another platform.
        - 1.5.4. Generate Report: Users should be able to generate reports based on the current data and yield predictions to print out.
2. Team Management
    - 2.1. Team Chat
        - 2.1.1. Chat Functionality: Users from the same team should be able to communicate safely and securely with each other.
        - 2.1.2. Send Messages: Users should be able to send and receive messages within a team space.
    - 2.2. User Accounts and Authentication
        - 2.2.1. Login: Users should be able to log in using their credentials or via OAuth.
        - 2.2.2. Reset Password: Users should be able to reset their password via email.

2.2.3. Activate Account via Email: Users should receive an email upon account creation to activate their account.
2.2.4. Join Team: Users should be able to join a team if a join link is sent to them via email.
2.2.5. Log out: Users should be able to log out of the system.

2.3. CRUD operations

2.3.1. Users should be able to update their account details in settings.
2.3.2. Update User Roles: Farm Managers should be able to invite new team members.
2.3.3. Invite User: Farm Managers should be able to update user roles.
2.3.4. Remove from Team: Farm Managers should be able to remove users from the team.

3. Crop Prediction Model Subsystem

3.1. Model training

3.1.1. A model is trained on every field individually to have personalised and accurate predictions based on their location.

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 manually retrain their personalised model for each field.
3.2.3. Using automated jobs (scheduler), every field's model will be retrained daily in order to keep the predictions relevant to the current field situation.

3.3. Weather Data Improvement

3.3.1. Using automated jobs (scheduler), weather will be fetched daily in order to ensure the user receives up to date information on their fields.
3.3.2. Weather is fetched daily from an external weather API.

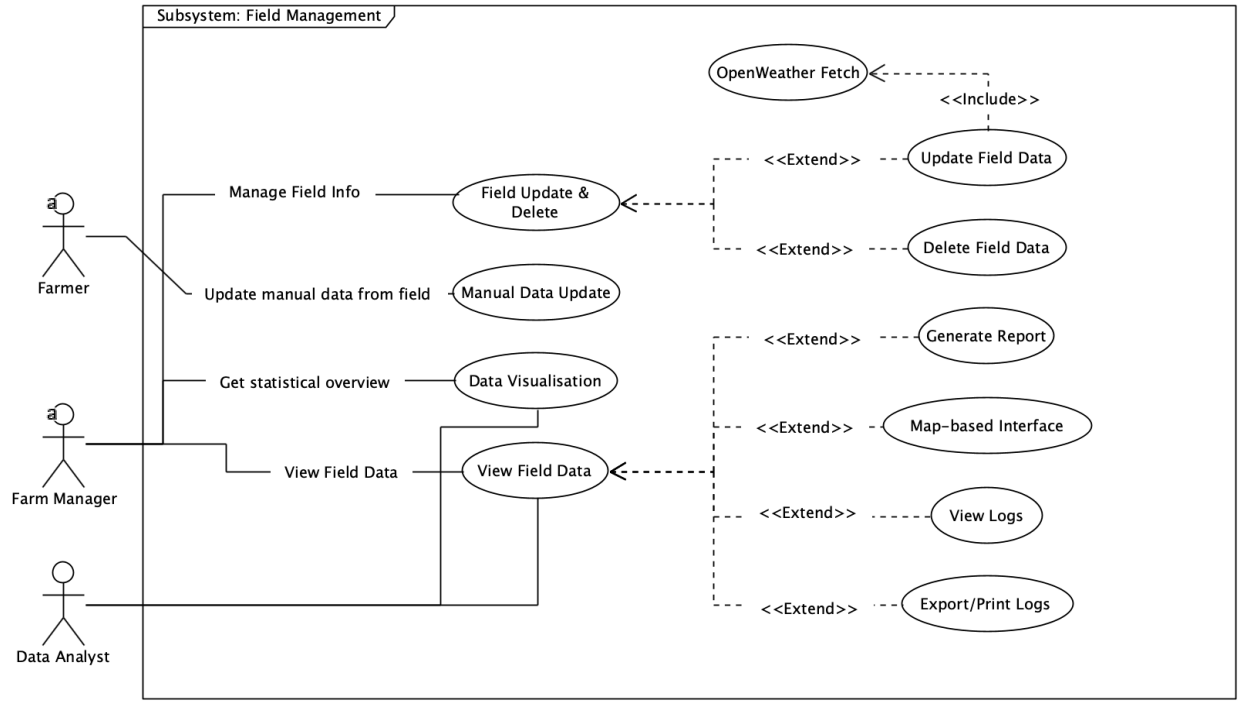
4. Revenue Subsystem

4.1. Market Data Visualisation

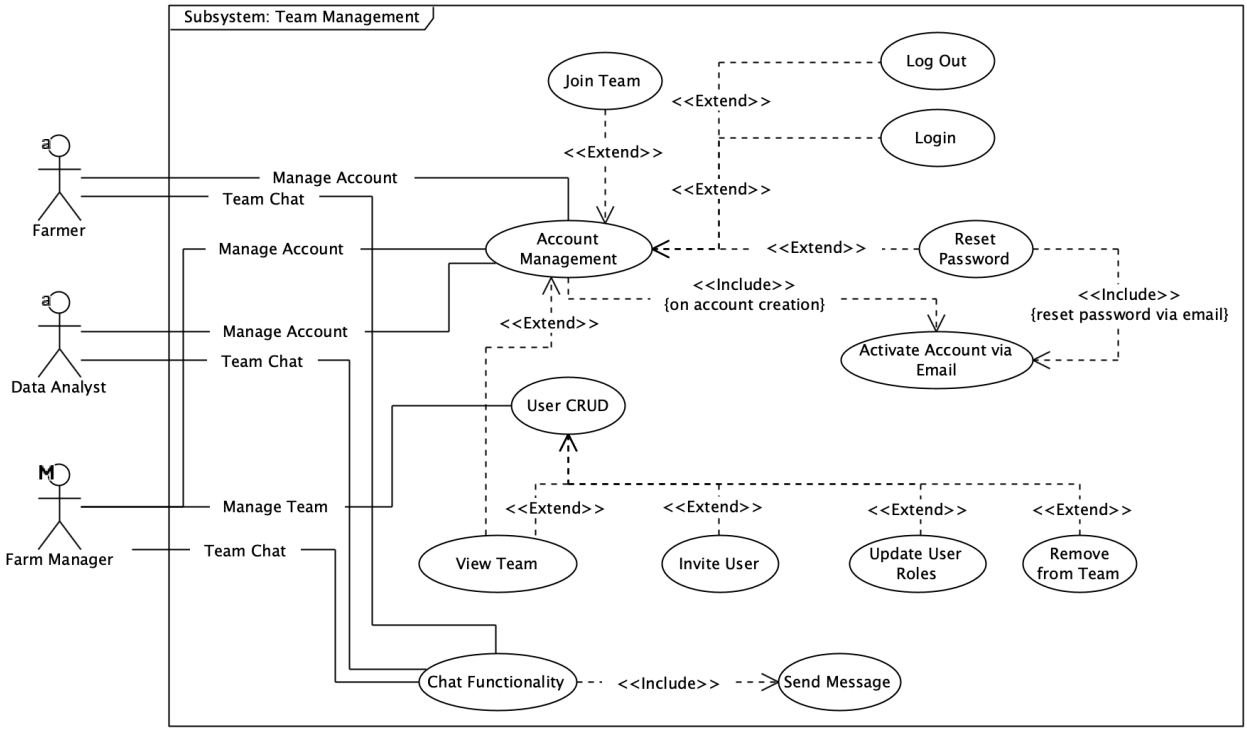
4.1.1. Fetch Team Yields: upon field selection, team yields will be calculated by crop type and used for revenue calculation.
4.1.2. Fetch Market Data: upon field selection, the market data will be fetched.

Use Case Diagrams

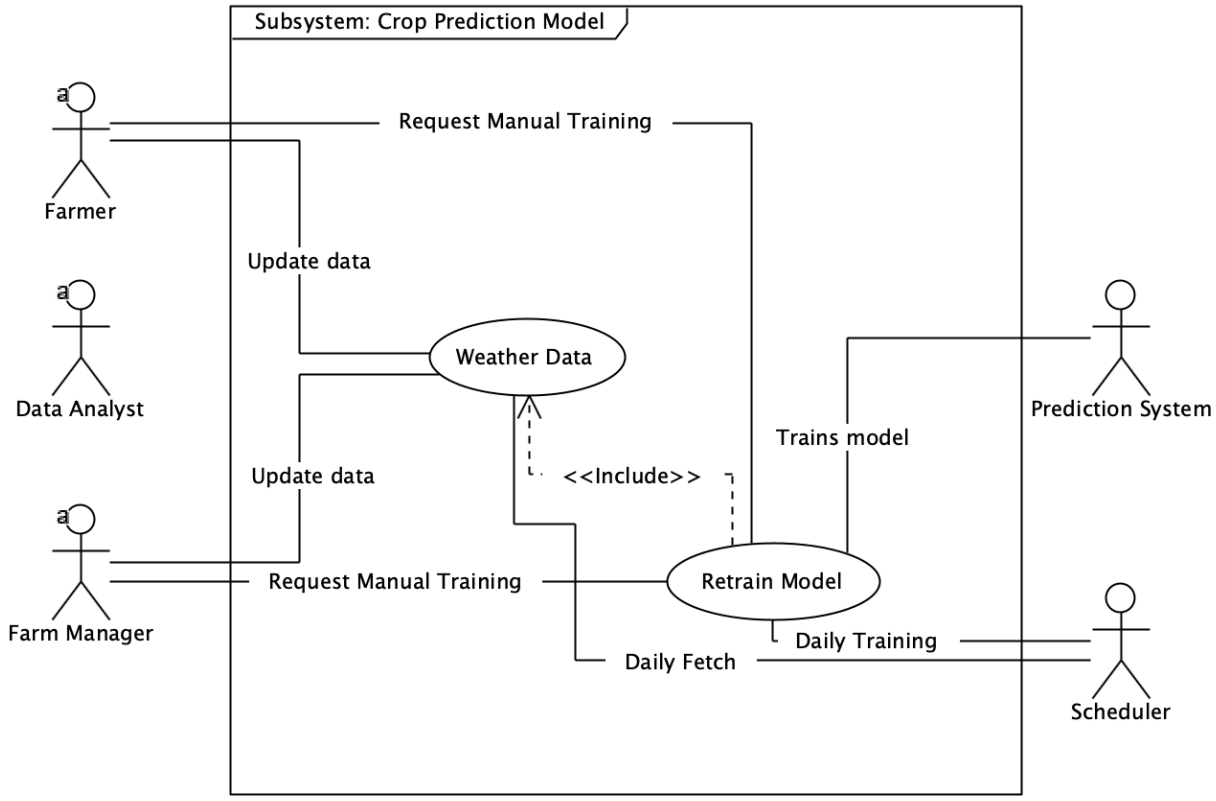
Field Management Subsystem



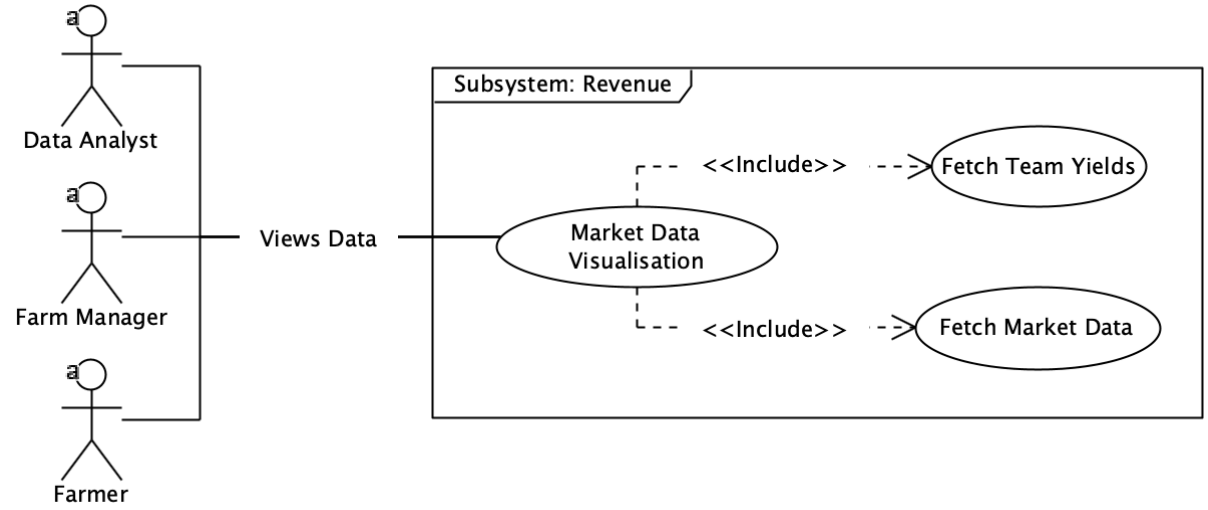
Team Management Subsystem



Crop Prediction Model Subsystem



Revenue 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 10 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. 70%+ overall code coverage 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.
4. Compatibility
    4.1. The app should be easily accessible via modern web browsers on various devices commonly used by farmers, such as smartphones, tablets, and desktops.
    4.2. Quantification

4.2.1. Provide a consistent user interface using component libraries.
4.2.2. Ensure a consistent and responsive user interface using a component library such as TailwindCSS. Prioritize mobile responsiveness for an optimal experience across different screen sizes.

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 Contracts

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

1. **getFieldInfo**: Retrieves detailed information about a specific field.
   - Input: field_id
   - Output: Field metadata (name, location, size, crops, etc.)
2. **getFieldData**: Fetches all the historical and current data related to a field.
   - Input: field_id
   - Output: Full field data, including crop health, soil information, etc.
3. **createField**: Creates a new field entry.
   - Input: field_area, field_name, crop_type, team_id
   - Output: Success message and field ID
4. **updateField**: Updates the details of an existing field.

    - Input: field_area, field_name, crop_type, team_id
    - Output: Acknowledgement (Success message)
5. **deleteField**: Deletes a specific field entry
    - Input: field_id
    - Output: Success message
6. **updateEntry**: Updates specific field data (like rainfall, temperature, etc.)
    - Input: field_id with entry-related data (such as rainfall, temperature, etc.)
    - Output: Success message
7. **getPastYieldAvg**: Retrieves past yield averages for the field.
    - Input: field_id
    - Output: Average yield data for the field over past seasons.

Weather Data Service Contracts

This service fetches the latest weather data for all fields managed by the user or team. The weather data is collected from an external weather API and processed in the background. This ensures that the system has the most up-to-date information for all fields, which is critical for field management and crop prediction.

1. **fetchWeather**: Fetches the latest weather data for all fields.
    - Input: None
    - Output: Success (Background Task)
2. **fetchSummary**: Retrieves a summary of the weather data for all fields.
    - Input: None
    - Output: Success (Background Task)

Team Management Service Contracts

The Team Management Service facilitates collaboration and communication among team members involved in field management. This service provides functionalities for managing team members, fields, and team-specific interactions, ensuring seamless coordination and information sharing. It allows users to communicate effectively, manage roles, and access shared resources efficiently.

1. **getTeamFields**: Retrieves all fields managed by the team.
    - Input: team_id
    - Output: List of fields managed by the team.
2. **getTeamFieldData**: Fetches data related to all team-managed fields.
    - Input: team_id
    - Output: Data for all fields owned by the team.
3. **addToTeam**: Adds a user to a specific team.

- Input: team_id, user_id
- Output: Success message

4. **removeFromTeam**: Removes a user from a specific team.
    - Input: team_id, user_id
    - Output: Success message
5. **updateRoles**: Updates the role of a team member.
    - Input: user_id, role
    - Output: Success message
6. **getTeamDetails**: Retrieves detailed information about the team.
    - Input: team_id
    - Output: Details of the team (members, fields, etc.)
7. **getTeamId**: Retrieves the unique ID for the team.
    - Input: user_id
    - Output: team_id
8. **sendMessage**: Sends a message in the team's chat.
    - Input: team_id, user_email, user_name, message
    - Output: Success message
9. **getTeamMessages:** Retrieves all messages from the team chat.
    - Input: team_id
    - Output: id, created_at, user_name, email, team_id, message

User Management Service Contracts

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

1. **updateUser:** Updates the user account details.
    - Input: id, full_name
    - Output: success
2. **getUser:** Retrieves user information for a specific user.
    - Input: user_id
    - Output: id, email, created_at, full_name, team_id, role

Crop Prediction Model Service Contracts

1. **train:** Trains a new prediction model for a specific field.
    - Input: field_id, batch, waitForCompletion
    - Output: If waitForCompletion is true, status, model, StageModel, YieldOnlyModel, training_time_s, prediction are returned.

Revenue Management Service Contracts

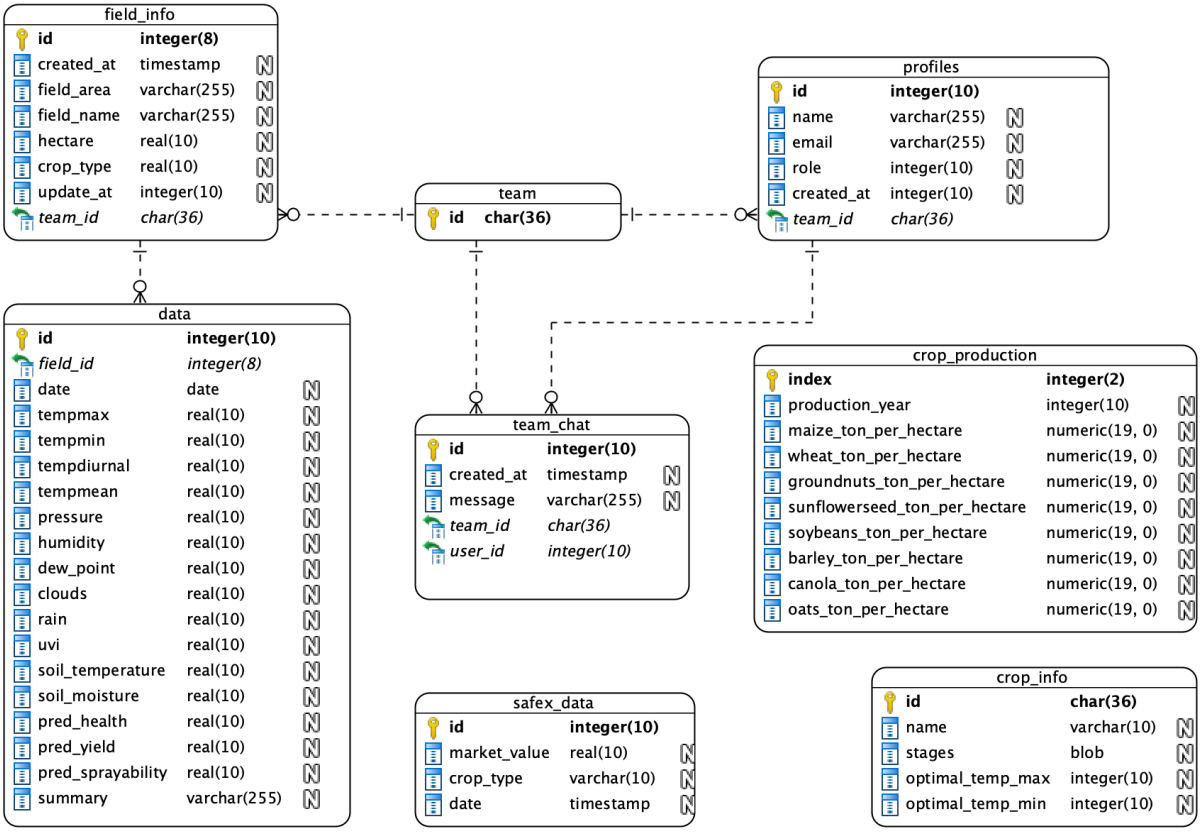
Description: This service handles all operations related to market and revenue data.

1. **Market:** Fetches market data for all crop types.
    - Input: crop
    - Output: Current market data for crops.
2. **getTeamYield**: Fetches all the combined crop type sizes and yield for fields.
    - Input: team_id
    - Output: Yield data for the team's fields

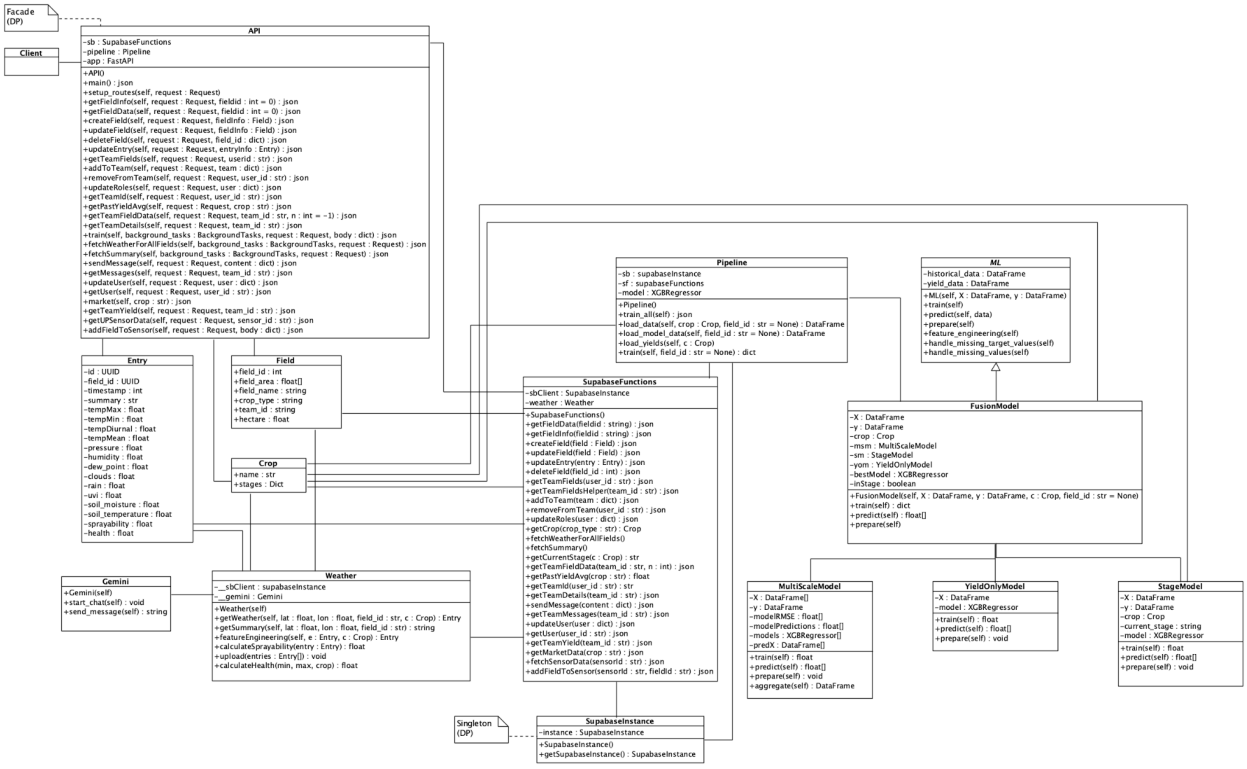
Each of these contracts represents how your service should handle requests and responses for the endpoints you provided. This structure can guide the implementation and maintenance of the API as well as help with documentation or integrations. Let me know if you need further details or adjustments!

Class Diagrams

Database Entity Relationship Diagram



Backend Class Diagram



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

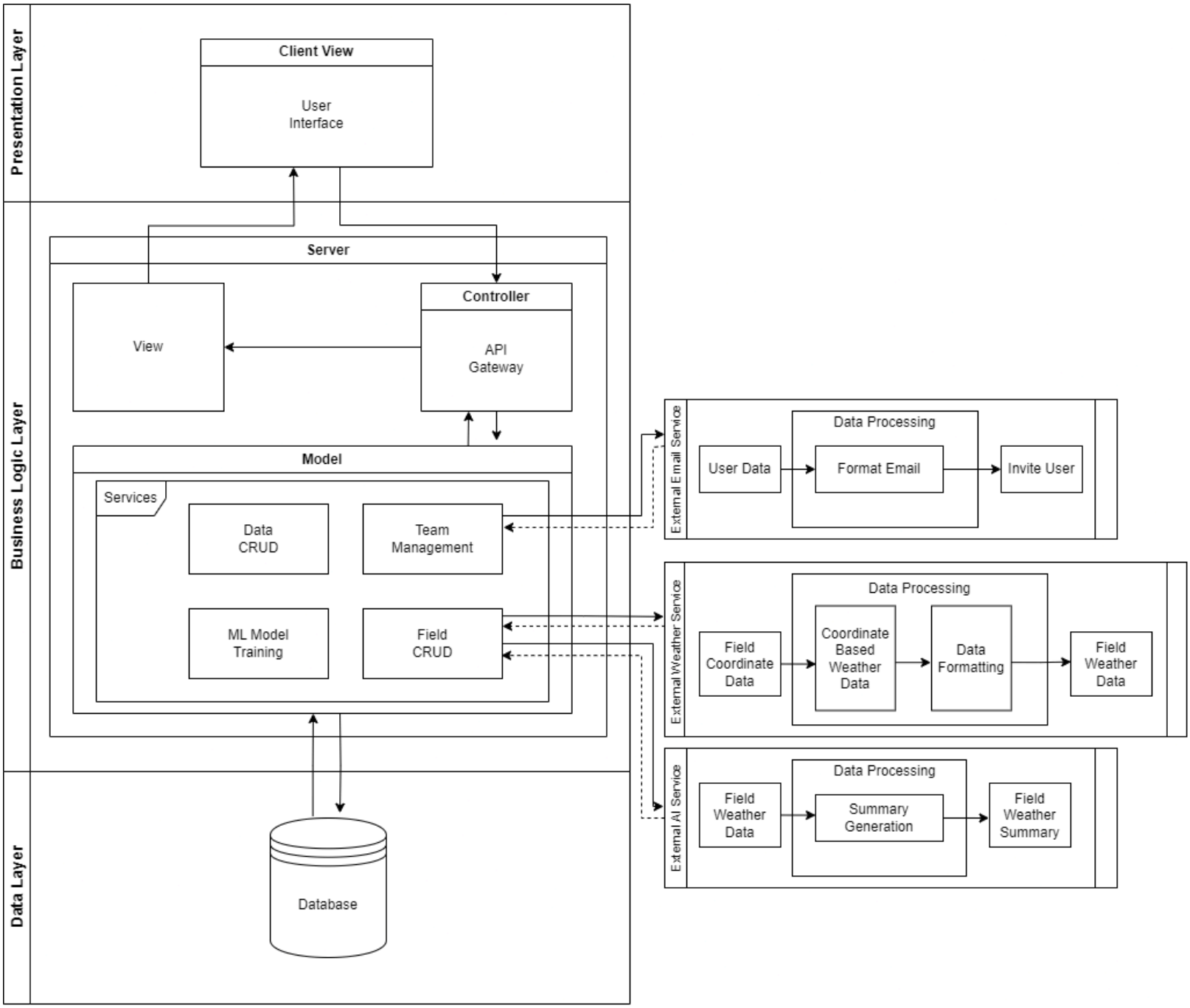
- Limited processing power and memory on devices used by farmers (e.g., older smartphones and tablets).

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.

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. Team Management Service: Manages user profiles and team-related functionalities, allowing for the creation, updating, and maintenance of user information. In addition to profile management, this service handles team creation, inviting new members, and managing team roles. Team managers can invite users via email, assign roles (such as Farm Manager, Farmer, or Data Analyst), and control access levels. The service also supports the removal of team members, ensuring smooth collaboration within the team structure.
        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. User Data: This filter gathers data on a user that needs to be sent to the external email service. The data includes the user's email address and selected role (such as Farm Manager, Farmer, or Data Analyst).
      1.1.2. Format Email: This filter formats the email details to be used by the external email service for sending invites. This ensures a smooth collaboration with the external email service.
      1.1.3. 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.
      1.1.4. Coordinate-Based Weather Data: After obtaining the field coordinates, this filter processes the data to retrieve weather information specific to those coordinates.
      1.1.5. 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.
      1.1.6. 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. Team Management Service: Manages user profiles and team-related functionalities, including the creation, updating, and

maintenance of user information, team creation, member invitations, role assignment, and access management.

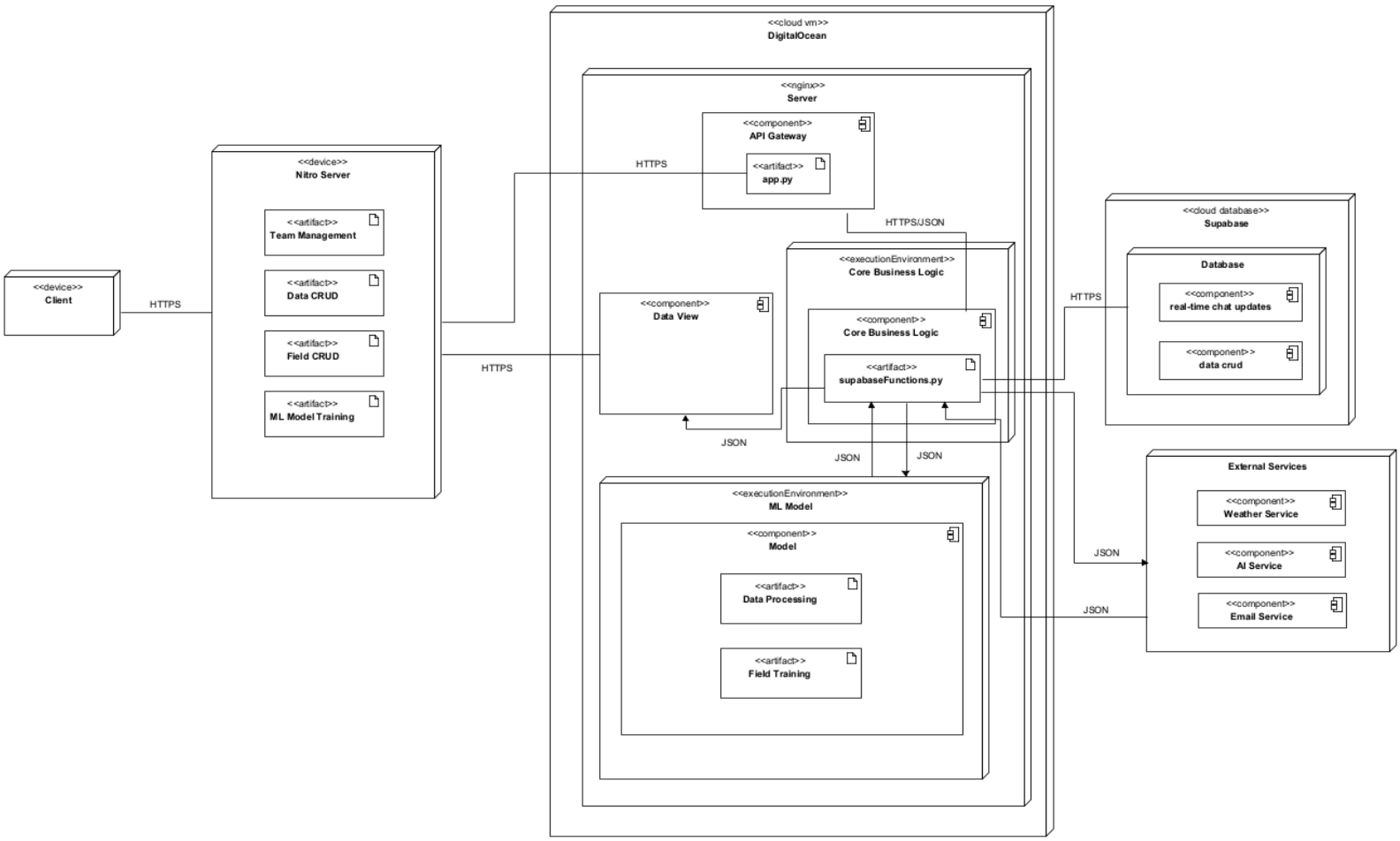
2.1.2.2. Data CRUD Service: Handles the creation, reading, updating, and deletion of data within the system.

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

The live deployed system is deployed onto 3 separate domains:

- The landing page is currently deployed on https://terrabyte.software. This is hosted on Netlify.
- The app itself is currently deployed on https://app.terrabyte.software. This is also hosted on Netlify and monitored by Uptime Robot.
- The API is deployed on https://api.terrabyte.software. This is hosted on DigitalOcean as a Virtual Machine droplet and uses Assertible to monitor uptime.

The entire deployment, as described in the deployment model, uses automatic deployments via Github Actions.

Bibliography

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

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