



TerraByte

# System Requirements Specification (SRS)

A project for  by  GeekGurusUnion

Demo 1

3 June 2024

# System Requirements Specification (SRS)

[Introduction](#)

[User Stories & User Characteristics](#)

[Functional Requirements](#)

[Requirements](#)

[Use Case Diagrams](#)

[Architectural Requirements](#)

[Quality Requirements](#)

[Service Contracts](#)

[Class Diagrams](#)

[Architectural Patterns](#)

[Design Patterns](#)

[Constraints](#)

[Technology Requirements](#)

[Bibliography](#)

## 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 input my farm's specific data into the system so that I can receive accurate crop yield predictions.

- Description: A farmer will enter data such as soil quality, crop variety, planting dates, and other relevant information into the system.
- Acceptance Criteria:
  - The system should provide a user-friendly form for data input.
  - The system should validate the data for completeness and correctness.
  - The system should store the data securely in the backend database.
  - Upon submission, the system should acknowledge receipt of the data and initiate the prediction process.

### User Story 2: Farm Manager

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

- Description: A farm manager will access the system to view data and reports on crop health and yield.
- Acceptance Criteria:
  - The reports should be accessible through a user-friendly dashboard.
  - The reports should include visual aids (graphs, charts) for easy interpretation.

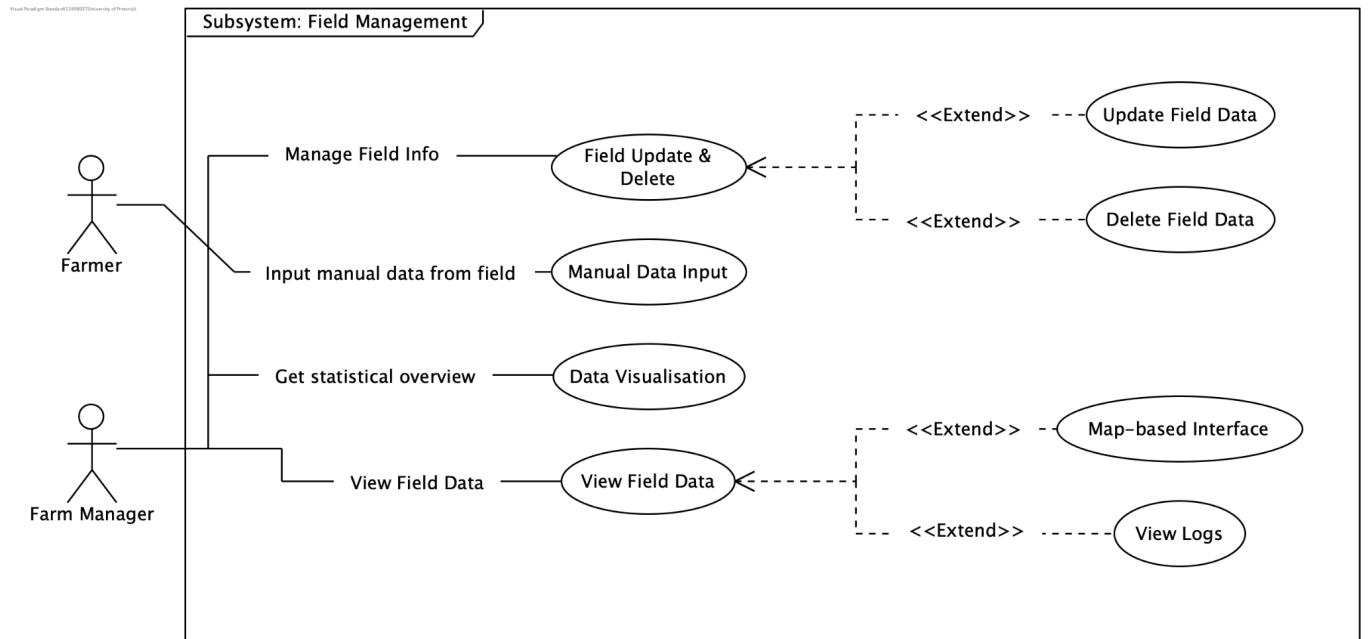
# Functional Requirements

## Requirements

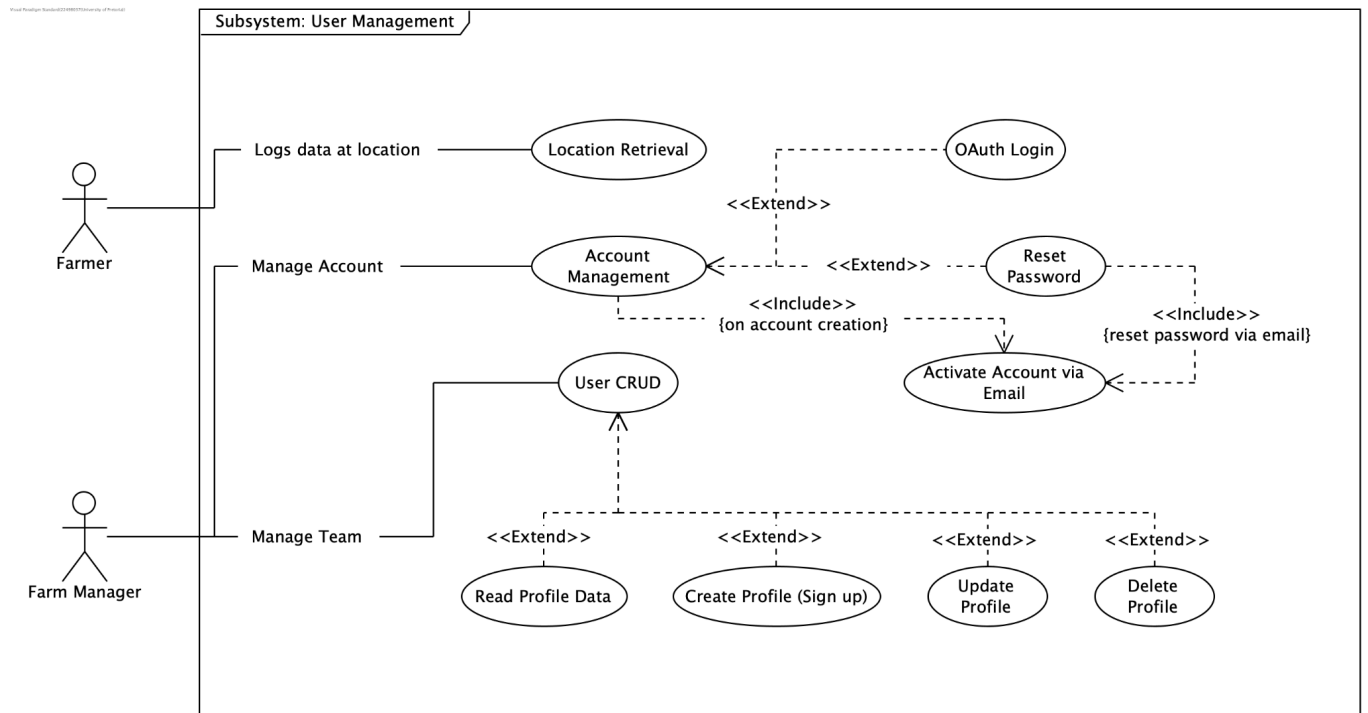
1. Field Management
  - 1.1. Manual Data Input
    - 1.1.1. Users should be able to manually input data related to crops, weather conditions, and other relevant factors
    - 1.1.2. System should be able to securely store the data while maintaining data integrity.
  - 1.2. CRUD Operations
    - 1.2.1. Users should be able to insert, update, read and delete data.
  - 1.3. Data Visualisation
    - 1.3.1. Representation: Data should be represented to the user with short explanations and graphs.
  - 1.4. View Field Data
    - 1.4.1. The map-based interface should generate a map and show the user's current farms. Field data should be colour-coded and vectorised and shown on the map. When clicked on, it should show more details about the specific field.
      - 1.4.1.1. A user should be able to draw and map out their field on the map.
    - 1.4.2. View Logs: Users should be able to view past entered data and be able to correct them if necessary.
2. User Management
  - 2.1. Map-based Interface
    - 2.1.1. User location should be retrieved, but not stored, to be able to show their current location and let them view the fields closeby.
  - 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.

# Use Case Diagrams

## Field Management Subsystem



## User Management Subsystem



# Architectural Requirements

## Quality Requirements

\* denotes future implementation.

### Performance

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.

#### Quantification

Performance is quantified by timing the amount of time it takes for our app to start up and show actionable results. Timing and performance metrics will also be gathered on ML models to ensure optimal performance. We'll keep track of the metrics while developing the model. This is defined by:

1. ML model training time\*
2. App loading time
3. API response time

#### Targets

1. Model training time should be less than 5 minutes.\*
2. App loading time should be less than 10 seconds.
3. API response time should be less than 5 seconds.

### Testability

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.

#### Quantification

Automated testing and coverage reports will be used to test individual components (unit tests), interactions between units (integration tests) and system flow testing (end-to-end testing). We broke the testing into separate parts to be able to uniquely identify the layer (in our layered architecture):

1. Presentation Layer: UI component unit testing.
2. Logic Layer: function and class unit testing.

3. Data Layer: ensure consistency in the database to serve accurate and representative data.
4. Between layers: integration testing.
5. System-wide: end-to-end testing.

## Targets

1. Make use of Jest to write unit tests for our frontend.
2. Make use of PyTest to write unit tests for our backend.
3. Make use of PyTest to test the database operations.
4. Make use of Jest and Postman to test our interactions between layers.
5. Make use Cypress to write end-to-end tests simulating system flow.
6. 60% coverage overall on Codecov.

## Efficiency

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.

## Quantification

Service measures will be used to identify the bottlenecks in our system. This can be reported by the SaaS (such as Digital Ocean) or defined by our code. This will be quantified by:

1. Graphs provided by our Digital Ocean droplet dashboard (budget, resource utilisation and response time)
2. Timing function calls and output for our ML model.\*
3. Reporting analytics to a CSV file to analyse bottlenecks.\*

## Targets

1. Checking the Digital Ocean dashboard on a regular basis.
2. Setting timers in Python\*

## Compatibility

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.

## Quantification

When a farmer wants to use the app on the go, it should be applicable to various devices, regardless of system performance. Users should be able to access the app offline, giving them the ability to view their yield data even if they are not connected. How will we achieve this?

1. Making use of a PWA to ensure offline capability.\*
2. User-friendly buttons.

3. Fetching new data when connected to the internet.\*
4. Saving new data temporarily while the app is not connected to the internet.\*

## Targets

1. Have an installable PWA-app available to all users.\*
2. Provide a consistent user interface using component libraries.
3. Set up runners that will periodically check for internet connection.\*
4. Make use of local storage to temporarily store values while not connected to the internet.\*

# 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.

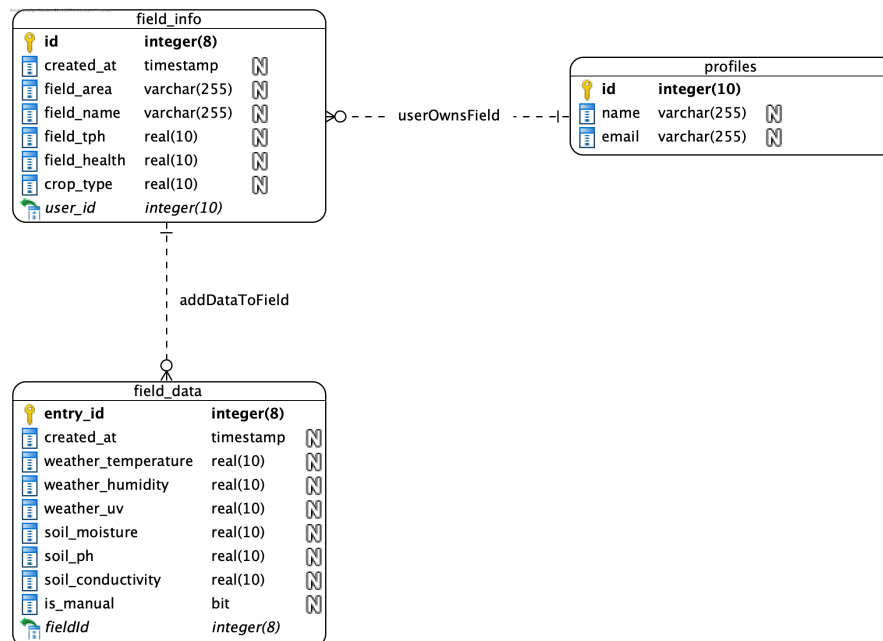
1. CreateUserAccount: Creates a new user account.



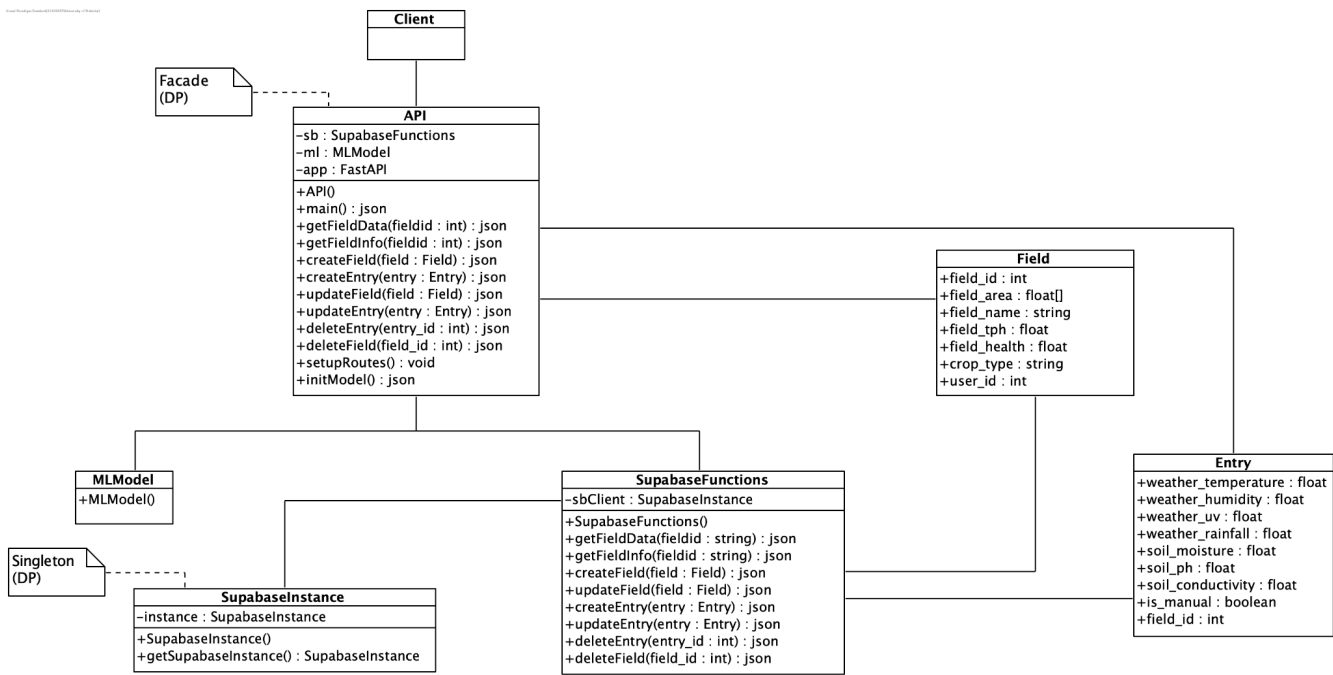
- Input: UserAccountData (includes email, password, personal details)
- Output: Acknowledgement (confirmation of account creation)
- 2. AuthenticateUser: Authenticates a user with their credentials.
  - Input: Email, Password
  - Output: AuthenticationToken
- 3. ResetPassword: Allows users to reset their 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

## Class Diagrams

### Database Entity Relationship Diagram



## Backend Class Diagram



## Architectural Patterns

### Layered Architectural 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, 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: Handles user interactions, facilitates user inputs, and communicates with the logic layer to enable backend functionalities. This layer ensures a seamless user experience and manages all client-side operations.
  - 1.2. Quantification:
2. Logic Layer:
  - 2.1. API Gateway:
    - 2.1.1. Functionality: Acts as a single-entry point for the presentation layer to access the logic layer. The API gateway will be used for external API calls, and routing requests based on functionality.

- 2.2. Core Business Logic:
  - 2.2.1. Functionality: Contains the main application logic, processes user requests, and applies business rules. This includes data processing, computation, and other core functionalities essential for the application.
  - 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 personalised forecasts for each farm. These predictions empower farmers to optimise resource use and boost their profitability.
- 2.3. Authentication:
  - 2.3.1. Functionality: The logic layer will handle our authentication system by providing logical routing between the database layer (where the users are stored) and the presentation layer.
- 3. Data Layer:
  - 3.1. A managed database will store application data in a centralised location, accessible by the logic layer for efficient data sharing and consistency. This layer provides data access mechanisms and interacts with the logic layer to store and retrieve information as needed.

## 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 layered 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)

## 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.

## 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.

## Bibliography

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

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