Design Document and Project Plan

Earth Observation and Blockchain Integration with Syngenta Foundation's Agri-Entrepreneur Program

Project Swaminathan

May 30th, 2025

Contents

1 Overview

This document outlines the design and project plan for a proof-of-concept (PoC) oracle system built on the Cardano blockchain to empower smallholder farmers in India. The system integrates earth observation, farm, and market data to facilitate trusted data exchange among farmers, Agri-Entrepreneurs (AEs), buyers, and government agencies. The focus is on the data structure, blockchain interaction, component architecture, technical specifications for the oracle contract, integration strategy, unit testing plan, and scalability measurement criteria.

The solution leverages Cardano blockchain for immutable data storage, Gamma Earth satellite APIs for crop insights, and provides a comprehensive mobile-first experience for agricultural entrepreneurs.

1.1 Data Flow Architecture

Authentication Flow

- User enters credentials in mobile app
- Backend validates against MongoDB user collection
- JWT token issued and stored securely (Keychain/Keystore)
- Token used for all subsequent API requests

Farmer Registration Flow

- Agent submits farmer data through mobile app
- Backend saves farmer record to MongoDB
- Cardano transaction created with metadata:

```
{
  "farmerId": "123",
  "farmerName": "John Doe",
  "timestamp": "2025-05-28T10:00:00Z"
}
```

- Transaction hash stored for audit trail

Field Registration Flow

- Agent submits field polygon (GeoJSON format)
- MongoDB stores with 2dsphere geospatial indexing
- Optional blockchain transaction for field verification
- Integration with Gamma Earth API for satellite data

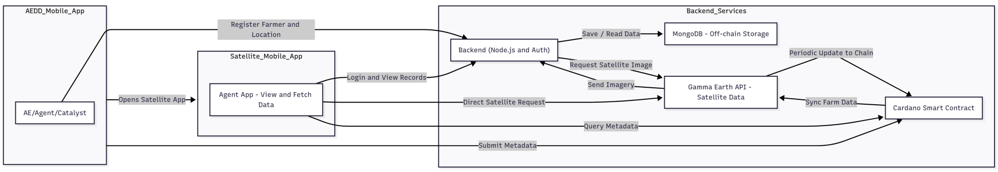
Crop Insights Flow

- System queries Gamma Earth API with field coordinates
- Receives NDVI data, crop health metrics, and satellite imagery
- Data processed and cached in MongoDB
- Results displayed in mobile app with visual analytics

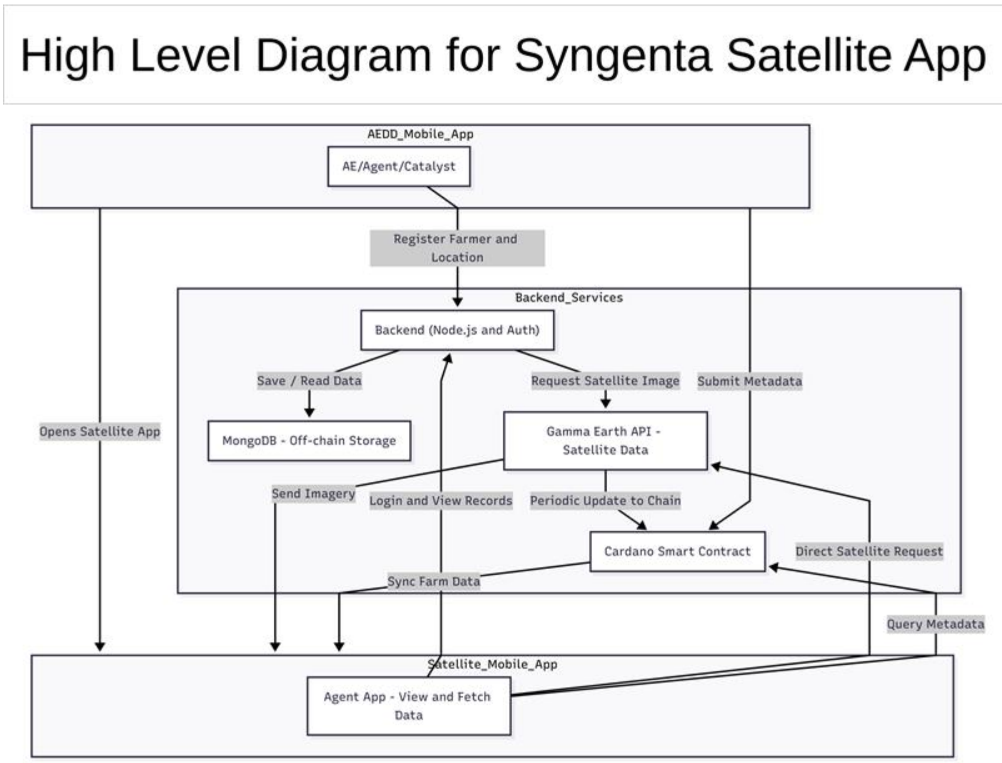
View Registered Farmer Data Flow

- Agent/AE/Catalyst requests a list of registered farmers or fields.
- The backend:
  - Retrieves this data directly from MongoDB.
  - Returns it to the app for display.
- No interaction with the blockchain is required for this operation.

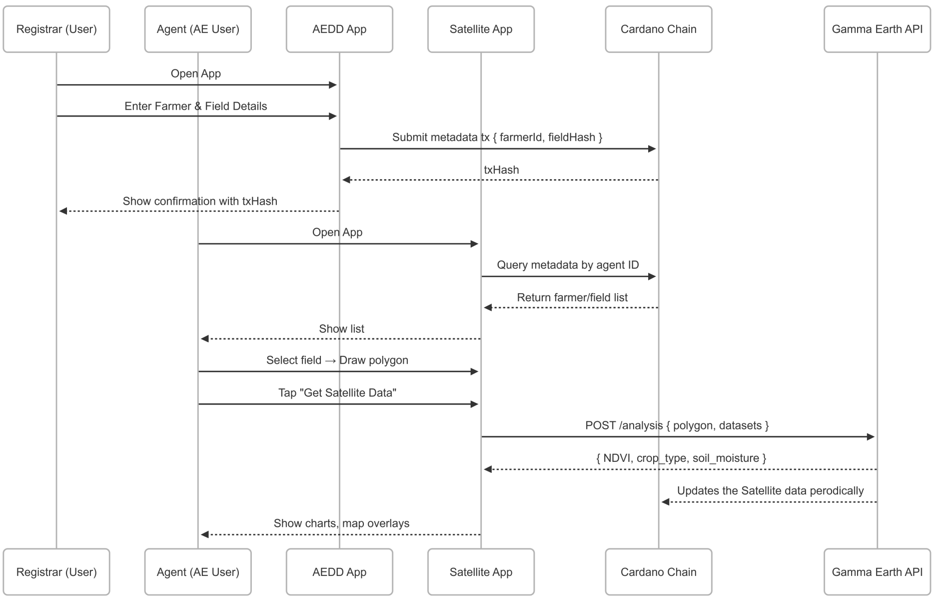
Data Flow Diagram



High Level Diagram



Low Level Diagram



2 User Interface (UI) Design

2.1 Core Objectives

The system is designed to provide real-time farm metrics stored on-chain using the Cardano blockchain, enabling transparent and immutable agricultural data management for farmers, admins, and agricultural entrepreneurs (AEs).

2.2 Technical Specification

This section outlines the technical requirements and integration design for the project leveraging the Cardano blockchain.

- **UI Mockups**: Provided by design team (Figma link to be added)
- **Implementation**:
  - React Native with EXPO Framework
  - Mobile-first design for Agent/AE/Catalyst workflows

2.3 Screen Reference

AEDD application with Oracle Satellite app deep linking

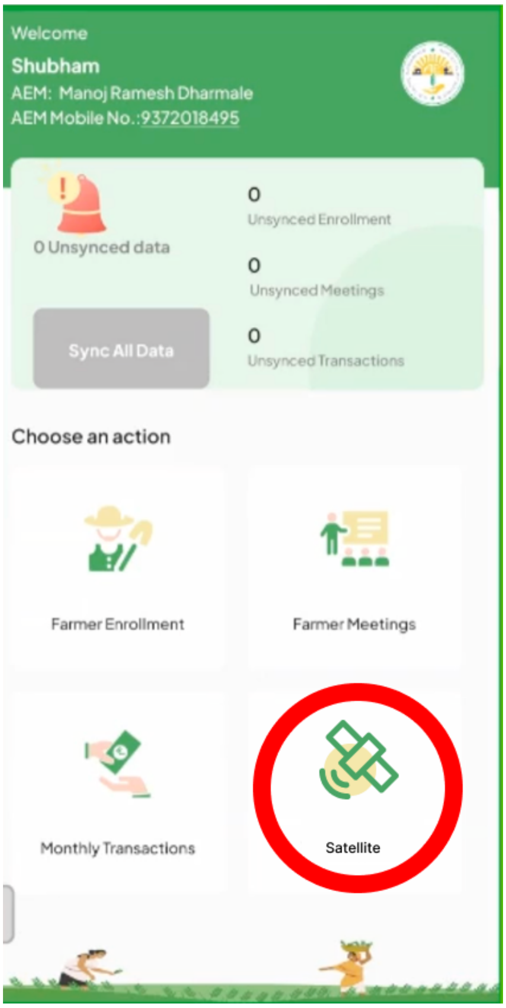


Figure 1: AEDD application with Oracle Satellite app deep linking

App splash screen

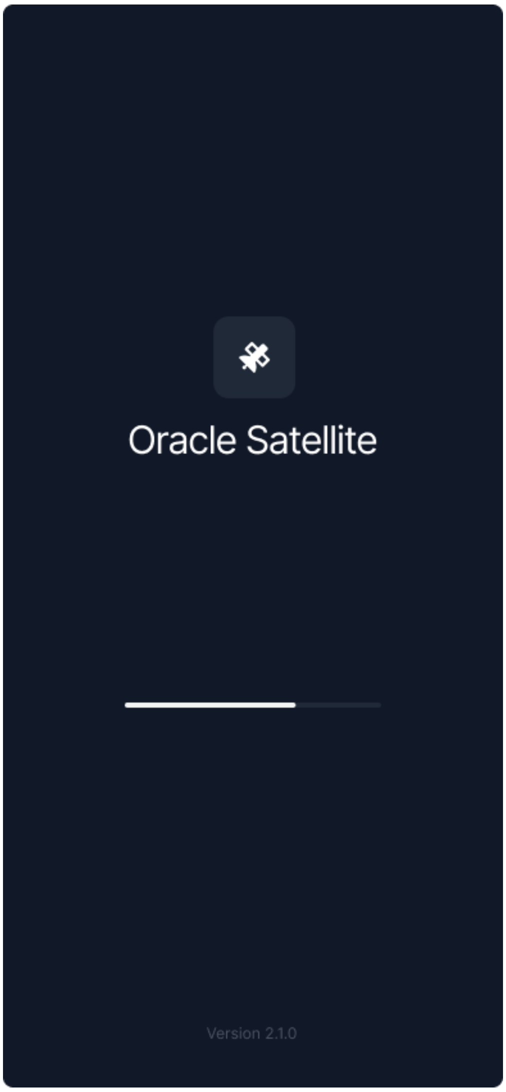


Figure 2: App splash screen

FieldView search screen

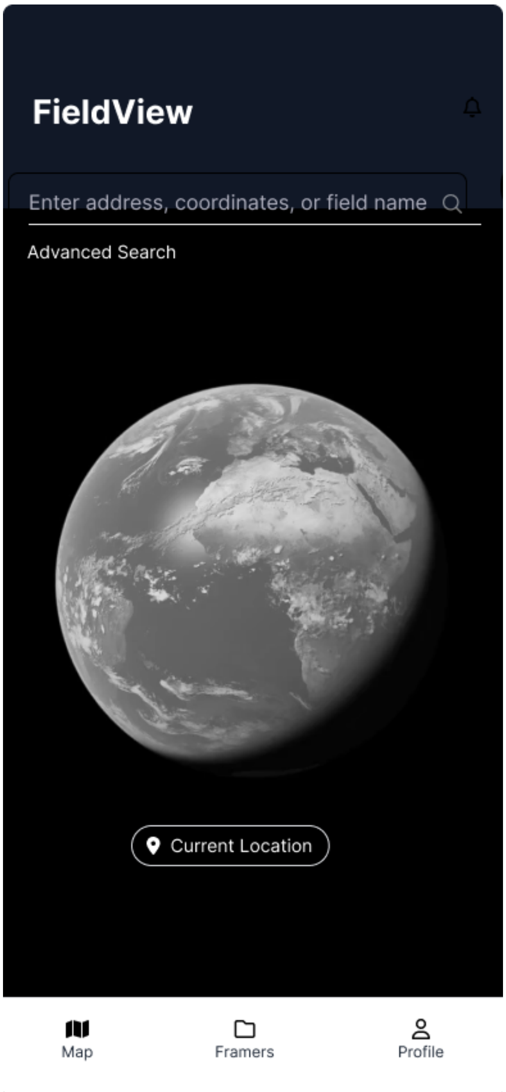


Figure 3: FieldView search screen

Detailed search filters

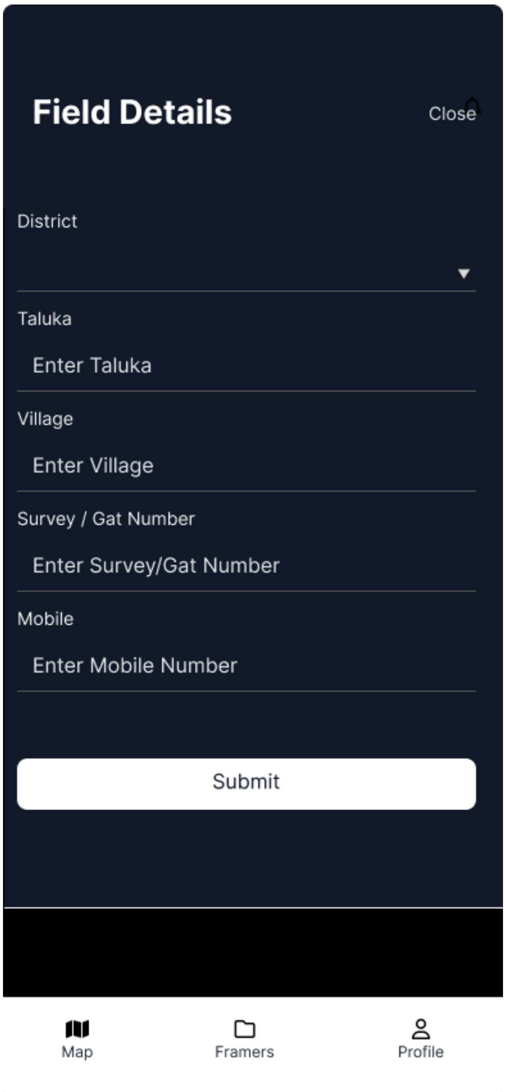


Figure 4: Detailed search filters

Interactive map display

Figure 5: Interactive map display

Boundary drawing tool

Figure 6: Boundary drawing tool

Precision editing interface

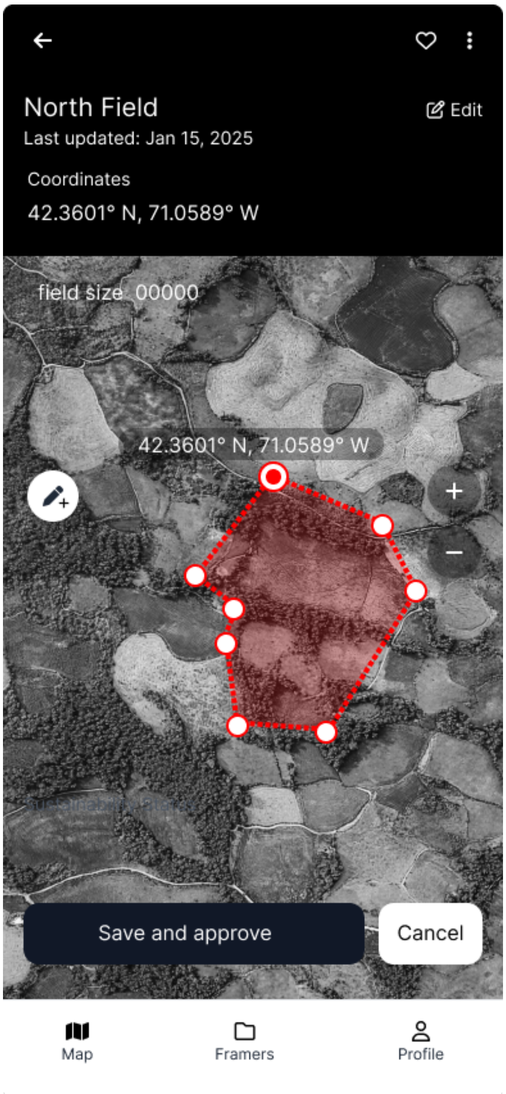


Figure 7: Precision editing interface

Shape modification screen

Figure 8: Shape modification screen

Data export options

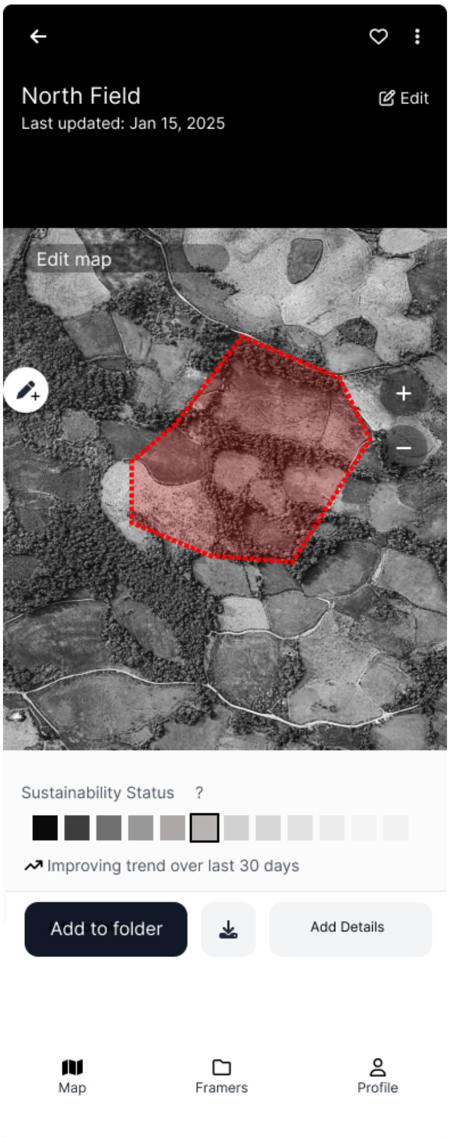


Figure 9: Data export options

Extended metadata entry

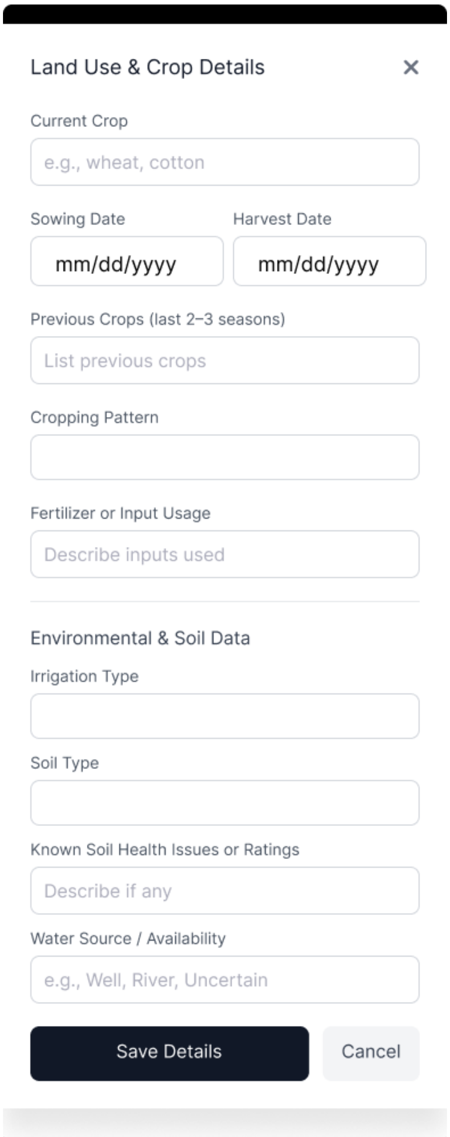


Figure 10: Extended metadata entry

Field details screen

Figure 11: Field details screen

2.4 User Management

Agricultural property dashboard

Figure 12: Agricultural property dashboard

Registered farmer list

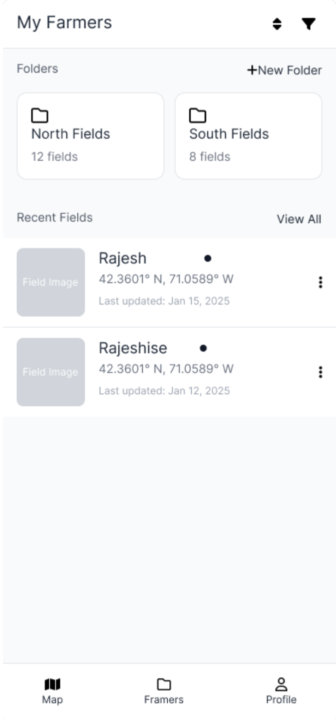


Figure 13: Registered farmer list

Individual farmer details

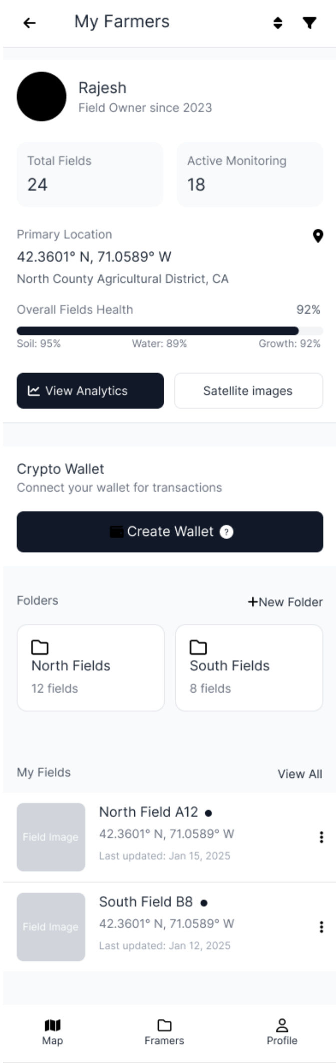


Figure 14: Individual farmer details

Cardano wallet authentication (Additional Implementation-Optional)

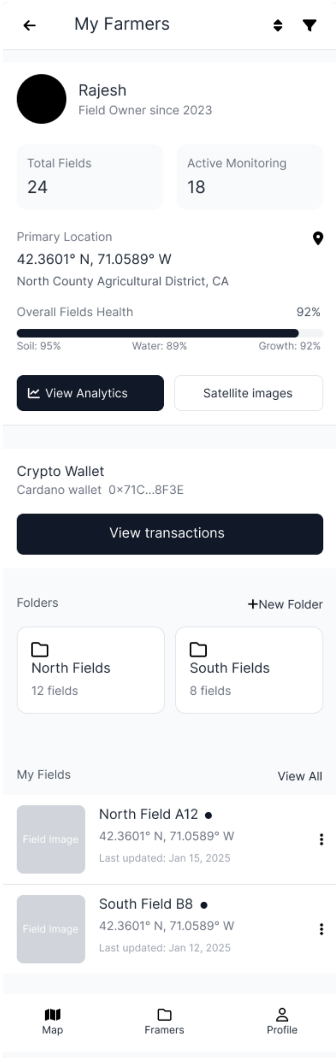


Figure 15: Cardano wallet authentication (Additional Implementation-Optional)

Digital asset interface (Additional Implementation-Optional)

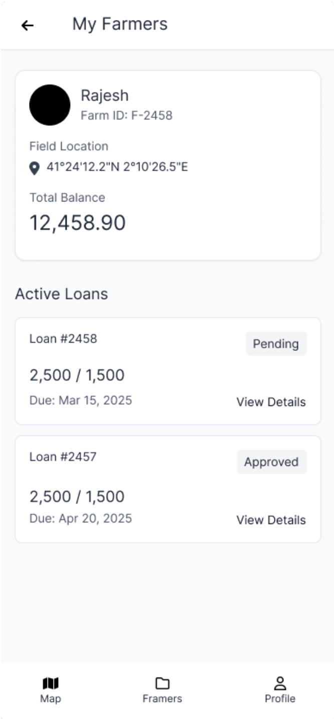


Figure 16: Digital asset interface (Additional Implementation-Optional)

Agri Entrepreneur profile screen

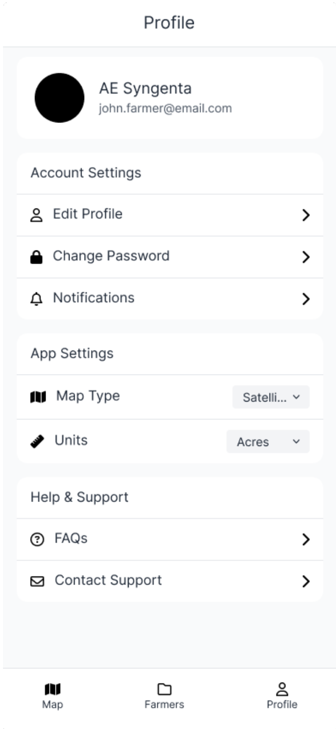


Figure 17: Agri Entrepreneur profile screen

3 DApp API Documentation

Task	Method	Path	Request (Body / Params)	Response

Post Admin	POST	`/admins/`	**Body:**
			{
			"admin_id": "admin003",
			"first_name": "ABC",
			"last_name": "DEF",
			"email": "aa.ee@example.com",
			"phone_number": "9876543210",
			"password_hash": "hashedpassword123",
			"role": "agent"
			}
				{
				"message": "Admin created successfully",
				"admin_id": "admin003"
				}

Admin Login	POST	`/admins/login`	**Body:**
			{
			"email": "aa.ee@example.com",
			"password": "hashedpassword123"
			}
				{
				"message": "Login successful",
				"token": "eyJ...YGe4",
				"admin": {
				"admin_id": "admin003",
				"role": "agent",
				"email": "aa.ee@example.com",
				"first_name": "ABC",
				"last_name": "DEF"
				}
				}

Get All Admins	GET	`/admins/`	*No body or params*	[{ "_id": "68411a1113feffc7d0ba674f", "admin_id": "admin003", "first_name": "ABC", "last_name": "DEF", ... }]

Get Admin by Id	GET	`/admins/{id}`	**Param:** `id = 683ed0a4ddd14fd5ef86fe42`	{
				"_id": "683ed0a4ddd14fd5ef86fe42",
				"admin_id": "admin003",
				"first_name": "ABC",
				"last_name": "CDE",
				...
				}

Update Admin	PUT	`/admins/{id}`	**Param:** `id = 683ecfb3ddd14fd5ef86fe3b`
			Body:
			{
			"admin_id": "admin001",
			"first_name": "Ravi",
			"last_name": "Kumar Show",
			"email": "ravi.kumar@example.com",
			"phone_number": "9876543210",
			"password_hash": "hashedpassword123",
			"role": "agent"
			}
				{
				"message": "Admin updated successfully",
				"_id": "683ecfb3ddd14fd5ef86fe3b",
				"admin_id": "admin001",
				"first_name": "Ravi",
				...
				}

Post Farmer Details	POST	`/farmers/`	**Body:**
			{
			"farmer_id": "farmer002",
			"first_name": "Roy",
			"last_name": "Sada",
			"phone_number": "9123456789",
			"registor": "68411a1113feffc7d0ba674f"
			}
				{
				"farmer_id": "farmer002",
				"first_name": "Roy",
				...
				}

Get Farmers by Registor	GET	`/farmers/registor/{regId}`	**Param:** `regId = 68411a1113feffc7d0ba674f`	[{ "_id": "684131c1c07d8a2fc6891aa6", "farmer_id": "farmer002", "first_name": "Roy", ... }]
Get All Farmers	GET	`/farmers/`	*No body or params*	[{ "_id": "684131c1c07d8a2fc6891aa6", "farmer_id": "farmer002", ... }, { ... }]

Get Farmer by Id	GET	`/farmers/{id}`	**Param:** `id = 683ed9b3ab48f2289bcd1662`	{
				"_id": "683ed9b3ab48f2289bcd1662",
				"farmer_id": "farmer001",
				...
				}

Update Farmer Details	PUT	`/farmers/{id}`	**Param:** id = 683ed9b3ab48f2289bcd1662
			Body:
			{
			"farmer_id": "farmer001",
			"first_name": "Sita",
			"last_name": "Devi",
			"phone_number": "1234567890",
			"registor": "60faad9cf6c1f50015c9d456"
			}
				{
				"_id": "683ed9b3ab48f2289bcd1662",
				"farmer_id": "farmer001",
				...
				}

Post Field	POST	`/fields/`	**Body:**
			{
			"field_id": "field789",
			"farmer_id": "64abd1234ef5a67b89011",
			"field_name": "North Plot",
			"field_location": {
			"type":"Polygon",
			"coordinates":[[
			[20.1,35.1],[20.2,35.1],
			[20.2,35.2],[20.1,35.2],
			[20.1,35.1]
			]]
			}
			}
				{
				"field_id": "field7893",
				"farmer_id": "64abd1234ef5a67b89011234",
				...
				}

Get All Fields	GET	`/fields/`	*No body or params*	[{ ... }, { ... }]

Get Field by Id	GET	`/fields/{id}`	**Param:** id = 683edd3e082c56add9fe443a	{
				"_id": "683edd3e082c56add9fe443a",
				"field_id": "field789",
				...
				}

Update Field Details	PUT	`/fields/{id}`	**Param:** id = 683edd3e082c56add9fe443a
			Body: same as GET response
				{
				"_id": "683edd3e082c56add9fe443a",
				"field_id": "field789",
				...
				}

Post Crop Data	POST	`/cropData/`	**Body:** `{ "field_id": "64abe5678cd4f90ab1234", "cropInformation": { "kharif":[{...}], "rabi":[], "zaid":[] }, "soilTesting": {...}, "ndvi_index":[{...}], "soil_moisture":[{...}], "crop_health_classification":[{...}], "source":"Satellite" }`	`{ "_id": "68415f0cd0247f709dac29a5", "field_id": "683edd3e082c56add9fe443a", ... }`
Get All Crop Data	GET	`/cropData/`	*No body or params*	`[ { ... } ]`
Get Crop Data by Id	GET	`/cropData/{id}`	**Param:** `id = 683ede84082c56add9fe4440`	`{ "_id": "683ede84082c56add9fe4440", "field_id": "64abe5678cd4f90ab1234567", ... }`
Update Crop Data by Id	PUT	`/cropData/{id}`	**Param:** `id = 683ede84082c56add9fe4440` **Body:** same as POST response	`{ "_id": "683ede84082c56add9fe4440", "cropInformation":{...}, ... }`

4 Data Structure and Blockchain Interaction

4.1 Key Components

<table>
<tr><th>Component</th><th>Technology</th><th>Purpose</th></tr>
<tr><td>Mobile Frontend</td><td>React Native + Expo</td><td>User interface for data entry
<pre><code>import { Lucid, Blockfrost } from "lucid-cardano";

const lucid = await Lucid.new(
  "https://cardano-mainnet.blockfrost.io/
  process.env.BLOCKFROST_API_KEY
  ),
  "Mainnet" // or "Preprod" for testing
);
</code></pre>
ization</td></tr>
<tr><td>Backend API</td><td>Node.js + Express</td><td>Business logic and blockchain integration</td></tr>
<tr><td>Database</td><td>MongoDB</td><td>Application data storage with geospatial indexing</td></tr>
<tr><td>Blockchain</td><td>Cardano + Blockfrost API</td><td>Immutable data verification and audit trail</td></tr>
<tr><td>Satellite Data</td><td>Gamma Earth API</td><td>Real-time crop health and NDVI analytics</td></tr>
</table>

4.2 User Ecosystem

- **Agents/AEs/Catalysts**: Register farmers and manage field data
- **Farmers**: Benefit from crop insights and blockchain-verified records
- **System Administrators**: Monitor and maintain the platform

4.3 Oracle Datum Structure

The oracle datum is a critical component stored on-chain to represent farm-related data. It is designed to be lightweight, extensible, and interoperable with Cardano's Plutus smart contracts.

Datum Fields

1. **Farm Land Area (`Integer`)**:
   - Represents the area of the farm in square yards, with one square yard encoded as 1,000,000 units to ensure precision, and to facilitate standard smart contract interfaces for decimal precision operations.

- Example: A farm of 500 square yards is represented as 500,000,000 (500 * 1,000,000).

2. **IPFS Hash for Farm Borders (`ByteString`)**:

   - A 46-byte IPFS hash linking to a JSON file containing geospatial coordinates (latitude/longitude points) defining the farm's boundaries.
   - Stored as a Cardano `ByteString` for compatibility with Plutus.
   - Example: `QmXyZ123...` (IPFS CIDv0 hash).

3. **Arbitrary Data (`BuiltinData`)**:

   - A flexible field for additional arbitrary metadata (e.g., soil health metrics, crop type, or yield predictions).
   - Uses Cardano's `BuiltinData` type to support arbitrary Plutus-compatible data structures.
   - Example: `{ "cropType": "rice", "soilPH": 6.5 }` would be encoded as the `BuiltinData` representation of the type:

Listing 26: Oracle Datum Schema

```
data OracleDatum = OracleDatum
  { farmArea :: Integer
  , ipfsHash :: BuiltinByteString
  , arbitraryData :: BuiltinData
  }
```

Listing 27: Example Arbitrary Data Type

```
-- | Example arbitrary data that can be included in the oracle
  ↪ datum.
data CropInfo =
   { cropType :: BuiltinByteString
   , soilPH :: Rational
   }
```

4.4 Blockchain Interaction

To ensure maximum usability, the system supports two oracle architectures on Cardano: Reference UTxO Oracle Architecture and Signed Message Oracle Architecture. Each has distinct trade-offs, balancing data availability, permissionless access, and operational efficiency.

Reference UTxO Oracle Architecture

In the reference UTxO oracle architecture, the oracle data is submitted to the chain directly as UTxOs that can be consumed as reference inputs by transactions that seek to interact with the oracle data. For our use-case, we model oracles as CIP-68 NFTs where the CIP-68 user-token is the farm parcel/area NFT, and the corresponding CIP-68 reference-token is the UTxO with the oracle data provided for the associated farm parcel/area.

There are a few key parameters for the system:

```haskell
-- | The public key hash of the party authorized to issue farm
    ↪ parcel/area DID NFTs.
issuanceOperator :: PubKeyHash
issuanceOperator = ...

-- | A list of public key hashes of parties authorized to
    ↪ update the oracle data
-- associated with the farm parcels/areas.
farmParcelOracleProviders :: [PubKeyHash]
farmParcelOracleProviders = [...]
```

Oracle Management Spending Validator:

- **On-Chain**:
  - The `oracleManagementValidator` is a Plutus spending validator that manages oracle data updates and access control.
  - The lifecycle of all the CIP-68 reference-tokens are managed by this script, each oracle UTxO lives at the `oracleManagementScript` and contains a CIP-68 reference-token corresponding to the relevant farm parcel/area.
  - The validation logic of this script ensures that only relevant parties (`farmParcelOracleProviders`) are able to spend the Oracle UTxOs at this script, and enforces that they must produce a continuing output which preserves the reference-tokens and produces a valid oracle datum.
- **Off-Chain**:
  - The oracle backend server fetches satellite data (via Gamma Earth APIs) and farmer inputs (via AE dashboards).
  - At regular intervals, the backend server submits transactions to update the oracle datum with newly fetched data.

DID NFT Minting Policy

The DID NFT minting policy is a Plutus minting policy which manages the issuance of DIDs for individual farm parcels/areas.

- **On-Chain**:
  - The `farmParcelNFTMintingPolicy` validation logic enforces:
  - Exactly one user-token and reference-token pair is minted per farm parcel/area, ensuring DID uniqueness.
  - The reference-token is included in an output to the oracle management validator's script address.
  - The initial oracle datum stored in this output is structurally valid.
  - Issuance of the DID is only possible with the authorization of the `issuanceOperator` (their signature must be present in any minting transaction).
- **Off-Chain**:
  - Upon successful registration / onboarding of a new farm parcel/area, the backend will submit a transaction (signed by the `issuanceOperator` public key) that mints the user-token & reference-token pair corresponding to the farm parcel/area.
  - The transaction includes an output UTxO with the user-token sent to the farm parcel/area owner's wallet.
  - Another output UTxO with the reference-token and a valid oracle datum is sent to the oracle management validator's script address.
- **DID NFTs**:
  - Each CIP-68 user-token acts as a DID for an individual farm parcel/area, granting smallholders both ownership and control over their data.
  - The DIDs assign verifiable identities to farm parcels/areas.

This architecture guarantees the data-availability of oracle data and ensures that the data can be consumed by dApps in a permissionless manner.

Signed Message Oracle Architecture

The Signed Message Oracle Architecture is an alternative approach where data is not directly stored on-chain but is instead signed by a trusted oracle operator and passed into transactions that need to verify the data's authenticity. This pattern optimizes for low-cost, high-frequency data updates by minimizing on-chain footprint.

Key Components:

- **Off-Chain**:
  - The oracle operator backend periodically collects farm data, uses that data to construct the oracle datum message, and signs the message with their private key.

- Each signed message includes:
- The oracle data (same schema as the on-chain `OracleDatum`) encoded as cborHex.
- Concretely the oracle data in this message is the result of `serialiseData oracleDatum` where `oracleDatum` is some value of the type `OracleDatum`.
- Two timestamps to prevent replay attacks:
- `validityStart` - the start of the validity range of this oracle data.
- `validityEnd` - the end of the validity range of this oracle data.

Smart contract protocols can consume these messages as follows:

- A Plutus smart contract (e.g., for dApps consuming the oracle data) is provided with:
  - The oracle data and validity range timestamps.
  - The signature of the oracle data message.
- The validator verifies that:
  - The signature is valid over the provided data.
  - The signing key matches the expected oracle operator's public key.
  - The data timestamp or nonce is within an acceptable range (to prevent stale inputs).

Advantages:

- **Lower Oracle Operational Costs**: Since no state is stored or mutated on-chain, the oracle provider does not incur the costs of transaction fees required to publish the data to the chain periodically.
- **High Frequency**: Oracle data can be can be updated frequently without the constraints of onchain throughput / bandwidth.
- **Dynamic Input**: Multiple dApps can consume the same oracle data simultaneously without regard for the potential of UTxO contention. In the reference input architecture, if an oracle datum is updated in the same block where a dApp is attempting to consume the datum, there is a potential that the dApp's transaction will fail as the oracle UTxO may already have been spent by the update transaction.

Trade-offs:

- **No Native Data Availability**: Oracle data must be provided by the transaction creator; there is no guaranteed on-chain storage.

- **Higher Off-Chain Responsibility**: Applications must fetch and store the oracle data themselves.

This architecture complements the Reference UTxO Oracle architecture by serving use cases that demand real-time updates or when cost sensitivity outweighs on-chain persistence.

4.5 Integration Strategy with PoC Oracle

Oracle Data Structure

The Cardano smart contract utilizes the following OracleDatum structure:

```
data OracleDatum = OracleDatum
  { farmArea :: Integer
  , ipfsHash :: BuiltinByteString
  , arbitraryData :: BuiltinData
  }
```

Typescript Representation:

```
interface OracleDatum {
  farmArea: number;
  ipfsHash: string;          // Hex or base58 encoded
  arbitraryData: any;        // Parsed JSON or raw bytes
}
```

4.6 Blockchain Integration Implementation

Blockfrost Configuration

```
import { Lucid, Blockfrost } from "lucid-cardano";

const lucid = await Lucid.new(
  new Blockfrost(
    "https://cardano-mainnet.blockfrost.io/api/v0",
    process.env.BLOCKFROST_API_KEY
  ),
  "Mainnet" // or "Preprod" for testing
);
```

4.7 API Endpoints for Oracle Integration

Endpoint	Method	Description	Response
/api/oracle/:address	GET	Fetch latest Oracle-Datum	JSON with decoded data
/api/oracle/history/:address	GET	Historical oracle data	Array of timestamped entries
/api/oracle/verify/:txHash	GET	Verify specific transaction	Verification status

5 Gamma Earth Satellite Integration

5.1 S2DR3 RISC API – Version 0.2

Important: The API is asynchronous. A POST request launches the processing job and immediately returns metadata. The actual processed imagery appears in the corresponding GCP bucket a few minutes later.

A. Job Submission Endpoint

```
POST
    https://s2dr3-job-20250428-862134799361.europe-west1.run.app/{USER_ID}
```

Request Example:

```
{
  "date": "2023-05-01",
  "aoi": "19.93 49.28 20.00 49.35"  // minx, miny, maxx, maxy
}
```

The AOI should not exceed 150 sq.km (roughly a 12km x 12km box).

Response Example

```
{
  "ISO": "UA",
  "MGRS": "35UQR",
  "PID": "T35UQR-20230810-u8600146",
  "aoi_overlap": "1.0",
  "bbox": "30.85 50.39 30.88 50.42",
  "date": "20230810",
  "job_id": "e26bb408-d330-11ef",
  "save_path_MS":
    "gs://sentinel-s2dr3/UA/T35UQR/T35UQR-u8600146/S2L2Ax10_T35UQR-u860014
  "save_path_TCI":
    "gs://sentinel-s2dr3/UA/T35UQR/T35UQR-u8600146/S2L2Ax10_T35UQR-u860014
}
```

Parameter Descriptions

Field	Description
`save_path_TCI`	Download URL for the super-resolved true-colour dataset
`save_path_MS`	Download URL for the super-resolved multi-spectral

B. Check Job Status

Endpoint:

```
GET
    ↪ https://s2dr3-job-20250428-862134799361.europe-west1.run.app/{USER_ID}/{
```

Response Example:

```
{
  "PID": "T35UQR-u8600146-20230810",
  "State": "completed",
  "jobID": "e26bb408-d330-11ef"
}
```

C. Tile Server

```
https://kgbbmarmdgv53gdb47pls2v2oe0qotcs.lambda-url.eu-central-1.on.aws/cog/t

https://kgbbmarmdgv53gdb47pls2v2oe0qotcs.lambda-url.eu-central-1.on.aws/cog/t

https://kgbbmarmdgv53gdb47pls2v2oe0qotcs.lambda-url.eu-central-1.on.aws/cog/t

PID = "T35UQR-860014671-20230810"
MGRS = "T35UQR"
SID = "860014671"
```

D. Download Raw GeoTIFFs with wget

IRP Image Request Example:

```
wget
    ↪ https://sentinel-s2dr3.s3.eu-central-1.amazonaws.com/PL/T34UDV/T34UDV-2
```

Response Example:

```
--2025-05-30 13:44:46--
    ↪ https://sentinel-s2dr3.s3.eu-central-1.amazonaws.com/PL/T34UDV/T34UDV-2
Resolving sentinel-s2dr3.s3.eu-central-1.amazonaws.com...
    ↪ connected.
HTTP request sent, awaiting response... 200 OK
Length: 22504275 (21M) [binary/octet-stream]
Saving to: 'S2L2Ax10_T34UDV-20240501-ucc1a562_IRP.tif'

100%[=================================================>] 21.46M  19.7MB/s
    ↪   in 1.1s

2025-05-30 13:44:48 (19.7 MB/s) -
    ↪ 'S2L2Ax10_T34UDV-20240501-ucc1a562_IRP.tif' saved
    ↪ [22504275/22504275]
```

TCI Image Request Example:

```
!wget
    ↪ https://storage.googleapis.com/sentinel-s2dr3/PL/T34UDV/T34UDV-84632e954
```

Response Example:

```
--2025-05-30 13:44:46--
    ↪ https://sentinel-s2dr3.s3.eu-central-1.amazonaws.com/PL/T34UDV/T34UDV-2
Resolving sentinel-s2dr3.s3.eu-central-1.amazonaws.com
    ↪ (sentinel-s2dr3.s3.eu-central-1.amazonaws.com)...
    ↪ 3.5.136.188, 3.5.136.206
Connecting to sentinel-s2dr3.s3.eu-central-1.amazonaws.com
    ↪ (sentinel-s2dr3.s3.eu-central-1.amazonaws.com)|3.5.136.188|:443...
    ↪ connected.
HTTP request sent, awaiting response... 200 OK
Length: 22504275 (21M) [binary/octet-stream]
Saving to: 'S2L2Ax10_T34UDV-84632e954-20230502_TCI.tif'
S2L2Ax10_T34UDV-84632e954-20230502_TCI.tif
    ↪ 100%[====================>] 21.46M 19.7MB/s in 1.1s
2025-05-30 13:44:48 (19.7 MB/s) -
    ↪ 'S2L2Ax10_T34UDV-84632e954-20230502_TCI.tif' saved
    ↪ [22504275/22504275]
```

5.2 Application Implementation

To interact with the `S2DR3 RISC API` in React Native and render `.tif`(GeoTIFF) satellite imagery using something like geotiff.js, we need to:

A. Submit a Job (React Native API Call)

```
// SubmitSatelliteJob.ts
```

```
export async function submitSatelliteJob(userId: string, date:
  ↪ string, aoi: string) {
  const url =
    ↪ `https://s2dr3-job-20250428-862134799361.europe-west1.run.app/${userI

  const payload = {
    date,
    aoi, // format: "minx miny maxx maxy"
  };

  try {
    const res = await fetch(url, {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify(payload),
    });

    const data = await res.json();
    if (!res.ok) throw new Error(data.message || 'Job
      ↪ submission failed');

    return data; // contains PID, job_id, and save paths
  } catch (err) {
    console.error(err);
    throw err;
  }
}
```

B. Poll Job Status

```
export async function getJobStatus(userId: string, jobId:
  ↪ string) {
  const url =
    ↪ `https://s2dr3-job-20250428-862134799361.europe-west1.run.app/${userI

  try {
    const res = await fetch(url);
    const data = await res.json();

    if (!res.ok) throw new Error(data.message || 'Status check
      ↪ failed');

    return data; // should include "State": "completed"
  } catch (err) {
    console.error(err);
```

```
      throw err;
    }
}
```

6 Testing Plan

To ensure the correctness, reliability, and upgradability of the oracle system, a comprehensive unit testing strategy is defined across all smart contract components, datum structures, and integration workflows. The testing plan is divided into three primary layers: Datum validation, Script validation, and Integration tests.

6.1 Datum Integrity Tests

Unit tests will focus on serialization, schema integrity, and forward compatibility of the `OracleDatum` and any embedded custom data types (e.g., `CropInfo`).

Test Cases:

- **Signed Message Structural Integrity**:
  - Validate oracle datum message: `serialiseData oracleDatum <> integerToByteString validityStart <> integerToByteString validityEnd  oracleDatumMessage` for valid `oracleDatum :: OracleDatum`.
- **Precision Integrity**:
  - Verify that farm areas represented with decimal-like scaling (e.g., `500 * 1_000_000`) maintain expected interpretation on-chain over common arithmetic operations.
- **Arbitrary Data Robustness**:
  - Fuzz test `arbitraryData` field with:
  - Valid known schema (e.g., correct `CropInfo`)
  - Unexpected but Plutus-compatible structures
  - Edge cases (empty maps, deeply nested structures, large byte strings)
- **Boundary Validation**:
  - Maximum field lengths for `ipfsHash`
  - Minimum and maximum land area edge cases (0, 1, 2^63-1)

6.2 Script Unit Tests

All Plutus validator and minting policy scripts will undergo isolated scenario-based testing.

Oracle Management Validator:

- **Positive Tests**:
  - Authorized oracle provider updates a datum with valid structure and preserves reference-token.
  - Continuity of UTxO validated (same asset class, correct script address).
- **Negative Tests**:
  - Unauthorized key attempts to spend an oracle UTxO.
  - Output datum is malformed or missing required fields.
  - Reference-token is removed, burned, or incorrectly transferred.

DID NFT Minting Policy:

- **Positive Tests**:
  - Correct issuance by `issuanceOperator` with matching user and reference-token minted.
  - Oracle UTxO correctly initialized with reference-token and valid datum.
- **Negative Tests**:
  - Attempt to mint without operator signature.
  - Minting multiple user-tokens or reference-tokens.
  - Oracle output not created, contains structurally invalid oracle datum, or does not contain the minted reference-token.

Signed Message Validation (for consuming contracts):

- **Positive Tests**:
  - Valid signature over CBOR-encoded oracle data with correct public key and valid timestamp window.
- **Negative Tests**:
  - Invalid signature.
  - Timestamps outside acceptable validity window.
  - Message tampering (mismatched data and signature).

6.3 Application Unit Testing

Backend Testing Framework

Testing Stack

- **Framework:** Jest
- **Mocking:** Jest mocks
- **API Testing:** Supertest

Test Categories

- **API Endpoint Testing**

```
describe('Oracle Data API', () => {
  beforeEach(async () => {
    await setupTestDatabase();
  });

  test('GET /api/oracle/:address - valid address', async () =>
    ↪ {
    const mockUtxo = createMockUtxo();
    jest.spyOn(lucid.fetcher,
      ↪ 'fetchUTxOs').mockResolvedValue([mockUtxo]);

    const response = await request(app)
      .get('/api/oracle/addr_test123...')
      .expect(200);

    expect(response.body).toHaveProperty('farmArea');
    expect(response.body).toHaveProperty('ipfsHash');
  });

  test('GET /api/oracle/:address - invalid address', async ()
    ↪ => {
    const response = await request(app)
      .get('/api/oracle/invalid-address')
      .expect(400);

    expect(response.body.error).toBe('Invalid Cardano
      ↪ address');
  });
});
```

- **Blockchain Integration Testing**

```
describe('Cardano Integration', () => {
  test('should decode OracleDatum correctly', () => {
    const mockDatum = 'hex_encoded_datum';
    const decoded = parseOracleDatum(mockDatum);

    expect(decoded.farmArea).toBe(100);
    expect(decoded.ipfsHash).toBe('QmTest123...');
  });

  test('should handle invalid datum gracefully', () => {
    const invalidDatum = 'invalid_hex';
    expect(() =>
      ↪ parseOracleDatum(invalidDatum)).toThrow('Invalid
      ↪ datum format');
  });
});
```

6.4 Frontend Testing Framework

Testing Stack

- **Framework:** Jest + React Testing Library or E2E testing through Maestro
- **Component Testing:** @testing-library/react-native

Test Examples

```
describe('FarmCard Component', () => {
  test('renders oracle data correctly', () => {
    const mockData = {
      farmArea: 42,
      ipfsHash: 'QmTest123',
      ndvi: 0.67
    };

    const { getByText } = render(<FarmCard data={mockData} />);

    expect(getByText('42 hectares')).toBeTruthy();
    expect(getByText('NDVI: 0.67')).toBeTruthy();
  });

  test('handles loading state', () => {
```

```
    const { getByTestId } = render(<FarmCard loading={true}
      ↪ />);
    expect(getByTestId('loading-spinner')).toBeTruthy();
  });
});
```

7 Security Testing Approach

7.1 Backend Security Measures

- **Authentication & Authorization**

```
// JWT Implementation with security headers
const jwt = require('jsonwebtoken');
const rateLimit = require('express-rate-limit');

// Rate limiting
app.use('/api/', rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // requests per window
  message: 'Too many requests from this IP'
}));

// Input validation
const { check, validationResult } =
    ↪ require('express-validator');

app.post('/api/farmers', [
  check('name').isLength({ min: 2, max: 50 }).escape(),
  check('email').isEmail().normalizeEmail(),
  check('phone').isMobilePhone(),
], (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }
  // Process request
});
```

- **Blockchain Security**

```javascript
// Cardano address validation
const validateCardanoAddress = (address) => {
  const addressRegex = /^addr1[a-z0-9]{98}$/; // Mainnet
  const testnetRegex = /^addr_test1[a-z0-9]{98}$/; // Testnet

  return addressRegex.test(address) ||
      ↪ testnetRegex.test(address);
};
```

7.2 Security Testing Checklist

API Security Tests

- Input Validation: SQL injection, XSS, command injection
- Authentication: JWT token validation, session management
- Authorization: Role-based access control
- Rate Limiting: DDoS protection, brute force prevention
- HTTPS: SSL/TLS configuration, certificate validation
- CORS: Cross-origin request policies

Database Security Tests

- Access Control: User permissions and roles
- Encryption: Data at rest and in transit
- Backup Security: Encrypted backups, access logs
- Connection Security: SSL connections, firewall rules

Blockchain Security Tests

- Address Validation: Proper format checking
- Private Key Management: Secure key storage
- Smart Contract Interaction: Parameter validation

7.3 Frontend Security Measures

Secure Storage Implementation

```
import * as SecureStore from 'expo-secure-store';

// Store sensitive data
const storeToken = async (token) => {
  await SecureStore.setItemAsync('authToken', token, {
    keychainService: 'agritech-app',
    encrypt: true
  });
};

// Retrieve with validation
const getToken = async () => {
  try {
    const token = await SecureStore.getItemAsync('authToken');
    if (token && !isTokenExpired(token)) {
      return token;
    }
    return null;
  } catch (error) {
    console.error('Token retrieval failed:', error);
    return null;
  }
};
```

Network Security

```
// HTTPS enforcement
const API_BASE_URL = __DEV__
  ? 'https://dev-api.agritech.com'
  : 'https://api.agritech.com';

// Certificate pinning (production)
const secureAxios = axios.create({
  baseURL: API_BASE_URL,
  timeout: 10000,
  headers: {
    'Content-Type': 'application/json'
  }
});
```

8 Scalability Measurement Criteria

8.1 Performance & Memory Optimizations (React Native)

To ensure smooth performance and efficient memory usage in the React Native mobile application, the following best practices are implemented:

- **Efficient List Rendering**
  - Use **FlatList** or **SectionList** for scrollable data (e.g., farmers, fields).
  - These components recycle rows for constant memory usage.
  - Implement `getItemLayout` to avoid dynamic height calculations and improve scroll performance.
- **Hermes Engine**
  - Enable **Hermes**, the optimized JavaScript engine for React Native.
  - Hermes compiles JavaScript into bytecode ahead-of-time.
  - Benefits:
    * Faster app startup
    * Lower memory usage
    * Reduced runtime parsing overhead
- **Lazy Loading**
  - Use `React.lazy()` and `Suspense` to **lazy-load heavy components**.
  - Defer loading of large modules (e.g., map screens, chart visualizations) until needed.
  - This keeps the **initial bundle small**, improving cold start time.
- **Render Optimization**
  - Use **React.memo**, **PureComponent**, **useMemo**, and **useCallback** to:
  - Avoid unnecessary re-renders
  - Cache expensive computations
  - Ensure shallow comparison of props
  - Prefer **immutable data structures** to simplify change detection.
- **Image Optimization**
  - Use `expo-image` pakage for better image optimization.
- **Memory Management**
  - Clean up side effects:

        * Cancel timers, API calls, or event subscriptions in `useEffect` cleanup or `componentWillUnmount`.
    - Avoid large, persistent in-memory caches.
    - Use **React Native's Performance Monitor** to:
        * Detect frame drops
        * Identify memory spikes

Together, these techniques ensure smooth UI (60 FPS) and low memory usage even as data grows.

9 Deployment Plan

9.1 Deployment Overview

This deployment plan outlines the steps and strategies to deliver and maintain the mobile application across development, staging, and production environments. The mobile frontend is built using **React Native with Expo**, while the backend uses **Node.js with Express**. Data is managed via **MongoDB** and **PostgreSQL**. A future enhancement will enable integration with a parent app via **Single Sign-On (SSO)**.

9.2 Deployment Environments

- **Development Environment**
    - **Purpose**: Internal testing and rapid development.
    - **Deployment**: Expo Prebuild and EAS development profile build.
    - **Access**: Developers only.
- **Production Environment**
    - **Purpose**: End-user availability.
    - **Deployment**:
        * **Frontend**: Expo EAS build for Android (Google Play or Internal APK)
        * **Backend**: Node.js hosted on a cloud provider (e.g., AWS or Render)
        * **Database**: Managed services (MongoDB)

9.3 Frontend Deployment (React Native + Expo)

- **Development Build**: Via Expo Go (QR code scanning) or Expo Prebuild
- **Production Build**:
    - Use **Expo EAS Build** for custom builds
    - EAS Submit to deploy to:
        * **Google Play Console** (Android)

9.4 Backend Deployment (Node.js/Express)

- **Containerization**: Dockerize the Node.js app
- **CI/CD Pipeline**: GitHub for automated builds and deployments
- **Hosting Options**:
  - AWS EC2

9.5 Database Setup

- **MongoDB**:
  - **Development**: Local MongoDB or MongoDB Atlas free tier
  - **Production**: MongoDB Atlas with backups and scaling
- **PostgreSQL**:
  - **Development**: Local instance or Docker
  - **Production**: Managed PostgreSQL (e.g., AWS RDS)

9.6 Single Sign-On (SSO) Integration

- **Planned Integration**:
  - Enable app access via parent app using a centralized **SSO system**
  - Likely protocols: OAuth2 / OpenID Connect
- **Approach**:
  - Use parent app's authentication provider for secure token exchange
  - Share session/token via deep linking or secure API handshake
  - Token validation middleware on the backend
- **Dependency**:
  - Coordination with the parent app's team
  - Updated user/session models to support external identity providers

9.7 Rollback Strategy

- **Frontend**: Use EAS Update rollback to revert to last stable OTA
- **Backend**: Maintain last known working Docker image version
- **Database**: Point-in-time recovery via backups (MongoDB Atlas / PostgreSQL RDS)

10 Project Plan and Implementation Timeline

The overall implementation of the Satellite Oracle and digital service infrastructure has been structured across multiple phases. This phased design ensures feasibility in field settings, continuous feedback loops with stakeholders, and a smooth progression from technical development to full-scale deployment.

To enhance clarity and readability, the Gantt chart has been divided into two parts:

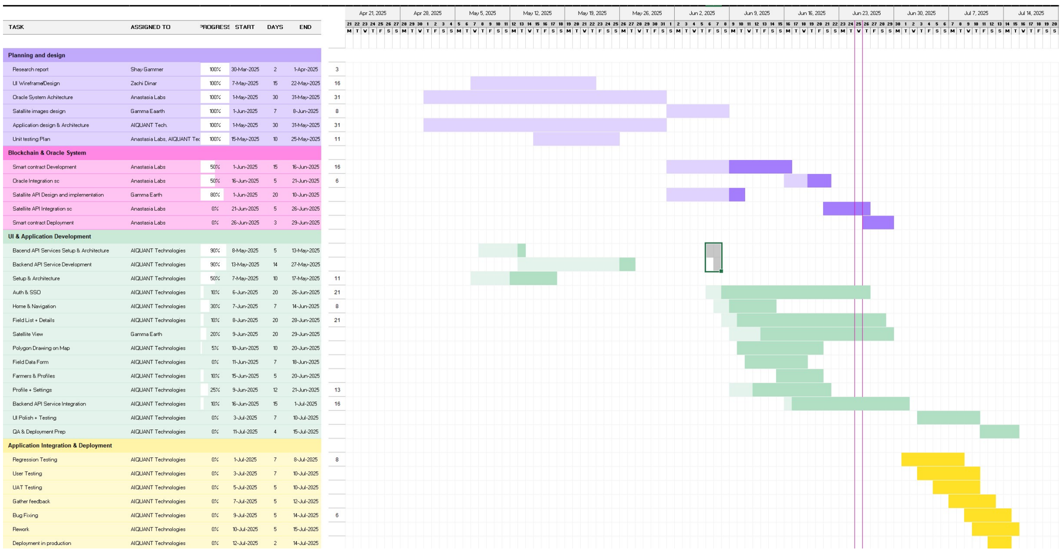


Figure 18: Project Timeline – Part 1: Planning, Development, and Integration

Part 1 outlines the foundational phases of the project, including:

- Technical architecture and wireframe design
- Oracle system integration and blockchain component prototyping
- Full-stack application development and UI deployment
- Preparation for integration and pilot testing

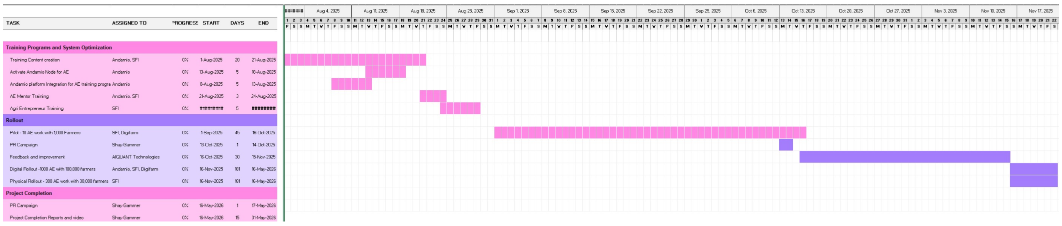


Figure 19: Project Timeline – Part 2: Rollout, Training, and Scaling

Part 2 transitions into:

- Training rollouts via Andamio Learn-to-Work Platform
- Field testing and user feedback loops
- Long-term sustainability monitoring and scaling phases

This plan allows for iterative feedback from the Agri-Entrepreneurs and partner institutions, while de-risking the final deployment through early validation and targeted adjustments.