The Rainy-Day Fund: Decentralized Parametric Insurance for Smallholder Farmers in Kenya

Group A: Ellena, Sabina, Noah, Vincent

University of Basel, Blockchain Challenge 25

September 25, 2025

Abstract

Smallholder farmers in Kenya face serious challenges due to climate variability, as more than 98% of agriculture depends on rain-fed systems and much of the country's arable land lies in arid and semi-arid regions (ASALs). Frequent droughts, unpredictable rainfall, and extreme weather events have reduced crop yields, threatened food security, and put rural livelihoods at risk. Traditional risk mitigation strategies, including crop diversification and off-farm income, remain insufficient, while irrigation development is limited due to infrastructural and environmental constraints. Weather index insurance (WII) has emerged as a practical solution because it provides affordable, quick, and reliable payouts based on objective weather data, while reducing administrative costs and moral hazard. Building on this idea, the Rainy-Day Fund introduces a decentralized, blockchain-based parametric insurance system. By using smart contracts and mobile payment platforms, it can deliver timely payouts to farmers, strengthen their resilience, and help them reinvest in their farms. This approach not only protects farmers from the financial shocks of extreme weather but also offers a transparent and efficient way to support sustainable agricultural livelihoods.

Keywords: Smallholder farmers; Kenya; climate change; weather index insurance; parametric insurance; blockchain; agricultural resilience; arid and semi-arid lands.

Contents

1 Introduction and Motivation

The goal of the Rainy Day Fund is to design a decentralized parametric insurance solution that can provide smallholder farmers with affordable, fast, and trustworthy protection against climate-related risks. By leveraging blockchain technology, the project seeks to overcome mistrust, high administrative costs, and inefficiencies that have limited adoption of weather index insurance. The motivation lies in addressing urgent climate vulnerabilities, reducing poverty traps, and creating scalable financial safety nets in one of the world's most underserved insurance markets.

Agriculture is the backbone of Kenya's economy, contributing approximately 21.3% to the country's GDP in 2024 (Bank 2024). It is the largest employer in the country, providing livelihoods for over 40% of the total population and more than 70% of the rural population (Food and Organization 2024b). Smallholder farmers form the majority of agricultural producers, yet they remain highly vulnerable to climate variability.

Kenya's Arid and Semi-Arid Lands (ASALs) cover more than 80% of the country, host over 70% of livestock, and are home to around 36% of the population (IUCN 2021; (NDMA) 2021; DHI 2021). Farmers in these regions depend almost entirely on rain-fed crops and livestock, making them extremely vulnerable to droughts and erratic rainfall.

The drought between 2020 and 2023, the worst in four decades, illustrates the severity of this risk: yields declined by up to 70%, 2.6 million livestock were lost, and 4.4 million people required urgent food assistance (Star 2024; (NDMA) 2024). These shocks highlight the fragility of rural livelihoods and the absence of effective financial safety nets to protect farmers.

Our guiding research question is: *How can blockchain-based weather insurance provide affordable, transparent, and automatic protection for rural farmers in Africa who are exposed to increasing climate risks?*

1.1 Problem Analysis

Smallholder farmers in Kenya are uniquely vulnerable to climate risks. About 98% of Kenya's agricultural systems are rain-fed (Kenya 2017), while irrigation development is limited due to infrastructural and environmental constraints (Wairimu n.d.). Traditional coping strategies such as crop diversification, off-farm work, and borrowing are insufficient to withstand the growing severity of climate shocks. As a result, households often fall into poverty traps, selling livestock or assets after droughts and struggling to recover in subsequent seasons. The recent 2020–2023 drought, the worst in four decades, illustrates the magnitude of the challenge: in some regions, crop yields dropped by up to 70%, more than 2.6 million livestock died, and 4.4 million people required urgent food assistance (OCHA 2023; Star 2024). These shocks do not only affect farmers individually but also ripple across Kenya's economy, since agriculture contributes over 20% of national GDP and supports the majority of rural livelihoods (Bank 2024).

Conventional agricultural insurance is largely absent in Kenya, and where it does exist, it suffers from deep structural weaknesses. High administrative costs make premiums unaffordable for smallholder farmers (Dominguez 2024). Claims processing is slow and heavily manual, which delays payouts and exacerbates farmers' financial vulnerability during shocks (Chainlink 2021). A lack of transparency around pricing and claims fosters widespread mistrust (Dominguez 2024). Moreover, coverage remains minimal: in Sub-Saharan Africa, fewer than 3% of farmers are insured, leaving more than 97% unprotected (Bank 2022).

Weather Index Insurance (WII) has emerged as a potential tool to address these challenges, because it relies on measurable weather data such as rainfall or temperature thresholds to trigger payouts. This reduces delays, administrative costs, and moral hazard compared to traditional indemnity-based insurance (Baagoe 2020; Sibiko 2018). Studies show that WII adoption can reduce poverty, improve household welfare, and encourage investment in improved seeds and fertilizers. Yet despite this potential, uptake remains very limited. Farmers often lack awareness or financial literacy, making WII appear too complex (Janzen 2020). Basis risk remains a major concern, as mismatches between weather station data and on-farm realities can result in payouts that do not reflect actual losses, undermining trust (Jensen 2016). Affordability is another barrier, since many farmers lack liquidity at the beginning of the planting season, precisely when premiums are due.

1.2 Benefits of Blockchain in Insurance

Blockchain technology, with its decentralized ledger and automated smart contracts, is particularly well-suited for microinsurance solutions targeting smallholder farmers. By eliminating intermediaries, automating claims, and ensuring transparency, blockchain can reduce operational costs, increase trust, and make insurance accessible even in remote areas (Dominguez 2024; Shetty 2022). Moreover, parametric microinsurance, where payouts are triggered by measurable weather events, benefits from blockchain's immutable and auditable infrastructure, ensuring rapid and verifiable payments. ...

2 Business Model and Perspective

Smallholder farmers in Kenya, particularly those living in the Arid and Semi-Arid Lands (ASALs), represent the primary target market for decentralized parametric insurance. These regions cover more than 80% of the country's land area, host over 70% of the livestock population, and are home to roughly 36% of the national population (IUCN 2021; DHI 2021). With more than 98% of agriculture dependent on rain-fed systems, smallholder farmers are disproportionately exposed to climate variability and shocks (Kenya 2017). Agriculture employs around 40% of Kenya's total population and over 70% of the rural population, yet fewer than 1% of farmers currently purchase agricultural insurance, leaving the vast majority unprotected (Food and Organization 2024a; Agriculture 2023).

2.1 Target Market

The stakeholders in Kenya's agricultural insurance ecosystem are diverse and interdependent. Farmers are the primary end-users. Insurers and micro-insurers underwrite and distribute weather-index products, while global reinsurers provide the capital buffers needed to make

large-scale coverage feasible (Artemis 2017; BASIS 2017). The Government of Kenya, through its National Agricultural Insurance Policy (NAIP), and regulatory agencies play a critical role in shaping policy and supervising products (Initiative 2024; Agriculture 2023). International donors and development partners fund pilots, subsidize premiums, and provide technical assistance, as seen in the Kenya Livestock Insurance Program (KLIP) (Bank 2022). Mobile money providers such as Safaricom's M-Pesa enable efficient premium collection and direct payouts(Oxford 2017).

2.2 High-Level Concept

...

3 Implementation

Overview of implementation goals, strategy and architecture.

The goal for this project was providing a showcase prototype . Therefore, the focus was on providing the most important features, a neat, easy UI and a good test-setup to showcase the functionality and laying the groundwork for possible extensions. These extensions and some of the more advanced features that were not provided in the prototype will be discussed in sections 3.5 and 3.6 It is important to note that the project is not a complete MVP (minimum viable product), due to the constraints when it comes to funding for example Chainlink as provider for weather data or using actual currency as a payment method.

3.1 Development Process & Strategy

The strategy for the implementation was largely based on a learning process. At first the focus was mainly on exploring the basics of Solidity based Smart Contracts and creating an early working prototype. Once this was achieved the focus shifted to code quality, finding efficient and well established solutions to tackle current challenges and at the end, integrating all of this into an easy-to-use UI .

The most important guidelines and ideas for the implementation were:

1 **Division of Work:** The work was divided mostly into two parts. One part was the overarching architecture and product design, including specifically the smart contract development using Solidity as well as researching existing libraries and standards to implement and use. The other part, equally important was researching and setting up the tools and frameworks for everything else: The project and testing setup using Hardhat and writing the actual test-code, implementing the UI, using React, Managing the Repository, and bug-fixing. This division was important for numerous reasons, but especially because it allowed for mutual verification and simultaneously the possibility to truly focus on specific topics.

2 **Testing Setup:** The Testing-Setup within Hardhat and using Node.js made finding bugs and testing after changes much more efficient and also allows for neat and compact showcases of the features and working prototype code. An excerpt from this testing setup and what it shows can be seen in figure 3.

3 **Research:** Research was an important, continuous process throughout the project, for finding technical solutions but also possible issues with the current state of the project.

4 **Usage of AI:** AI was used throughout the project for research purposes, when trying to find fitting libraries, tools and standards, as well as making repetitive tasks, like writing test code more efficient.

An important challenge was to align technical progress with the economic modeling. Once a working prototype was achieved, communication between these two distinct work streams came into focus. Daily synchronization and change management helped in finding ways to implement the most important parts of the business model, whilst simplifying other parts, like using a mock weather oracle, instead of Chainlink as a source of weather data. The strategy during the development process was to implement based on an objective hierarchy:

1 Working Smart Contract with deterministic payouts

2 Investment logic

3 Investor incentive model (yield / compound interest)

4 Tests and Code quality

5 Working frontend for showcases and manual testing

6 ...

These are just the most important tasks that are included in the final prototype. For further necessary steps towards a finished product refer to 3.6.

3.2 System (Macro) Architecture

The macro architecture of the system is centered around the RainyDayFund smart contract, which acts as the core of the decentralized insurance platform. It is designed to be extensible for integration with oracles (such as Chainlink), `ERC20 tokens` as payment and is integrated within a frontend built with React. The architecture emphasizes modularity, and ease of use for both farmers and investors, as well as neat and easy testing and showcasing.

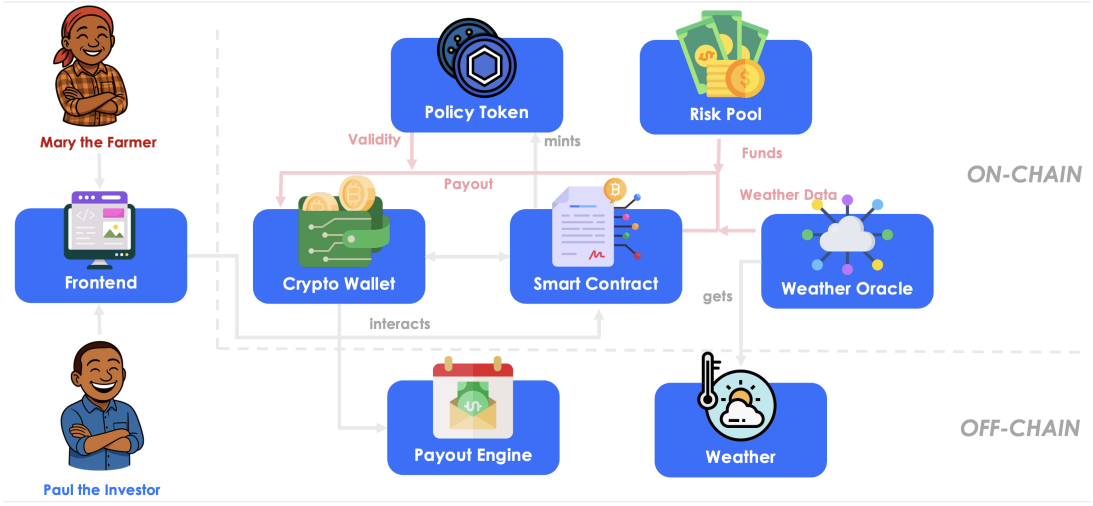

Figure 1: Architectural Overview
Source: Author's own.

As shown in figure 1 the users (both farmers and investors) own a crypto wallet and connect this to the frontend. The frontend then allows them to easily invest or buy policies for the current season with their crypto funds, by programmatically interacting with the Smart Contract. The smart contract then mints a policy token for farmers or deposits the investors funds in the vault, so that they can receive a yield. All funds are stored in a Risk-Pool (in our case the ERC4626 vault). The farmers can claim their policies at the end of the season and, if all conditions are met, i.e. the weather oracle gets and returns weather data, below the threshold and the farmer making the claim actually owns the policy tokens, receive their coverage funds. Investors may withdraw their funds, receiving all the accumulated yield. The `Payout Engine` symbolizes the possibility of transferring out funds from the crypto wallet to widely used mobile-money services, like M-Pesa. This is already a well established process and not included in the project itself, should however be noted, because it allows the farmers to actually immediately use their payouts in their everyday life.

The concrete timeline for the process is based on the crop season and has 5 phases:

1. **Active Phase:** In the active phase policies can be bought and investments can be made. The premium for the current season is already set at the beginning through the auction-based system explained in section 2 check wether this links correctly!. No claims or withdrawals can be made during this phase.

2. **Inactive Phase:** Now the policies can no longer be bought, but investments can still be made. This allows for a speculative market for the insurance policies, while keeping the possibility of introducing more liquidity from capital investors.

3. **Claim Phase:** The claim phase starts upon the actual crop season ending. Farmers may now try to claim their policies and receive their coverage payouts, while investors have to wait until this phase is over.

4. **Withdrawal Phase:** Now investors may choose to withdraw their capital and receive their yield, while farmers wait until the next season is started.

5. **Finished Phase:** The season is over, unclaimed policies are now invalid and not-withdrawn investments are used as liquidity for the next season, earning yield on the current investment (including the earned yield), creating a compounding effect.

3.3 Smart Contract Design (Micro)

The project is built around a Solidity smart contract using the Ethereum Virtual Machine. This central RainyDayFund contract can be called using the front end and holds all the exposed functionality for the consumer. Investors use the `invest()` and `withdraw()` methods while insurees can buy and claim their policies in batches using the `buyPolicy()` and `claimPolicies()` functions. The contract leverages the ERC1155 token standard for policy tokens, enabling efficient batch operations and flexible policy management.

As shown in Figure 2, the initial contract made use of the `1155 token standard` for the `policyTokens`, to take advantage of the cost-efficient batch operations (Ethereum 2025). To help with testing the (Mock)-MUSCD was created, implementing the ERC20 interface, to emulate the behavior of common stablecoins like USDC, which are used as payment and funding

for the riskpool. This allowed for the free minting and full control over the coins and made testing the functionality of the main contract possible, even within the bounds of Remix VM . Therefore, testing was quicker and easier at the beginning, without the need to always deploy on the Sepolia Testnet or an immediate, complete setup, for example with hardhat.

Initially there were multiple issues:

1 **Mapping for storing addresses:** Initially we used a mapping to store who had bought policies and was thereby eligible to claim them. However, there were some issues with this: Firstly, the policies lost all their value when they were resold, because the address of the new owner would not be tracked in the mapping. Moreover, storing all of this data was very gas-inefficient and redundant, because ownership of the policyToken is proof of ownership for the policy.

2 **No proper incentive for investors:** We had already planned to give the investors some incentive to keep their funds in the riskpool, even after the season would end, to increase liquidity for the next season. The idea to achieve this, was, to use compound interest or create some other form of yield over time. However, implementing this with the initial setup proved difficult and created vastly unfair results Hier vllt nochmal genauer erklären

3 **Weather oracle:** The initial mock-up for the weather oracle was just a number that could be accessed and stored on the contract directly, as an initial mock-up for testing purposes. Nevertheless, this was far from the goal of using chainlink as a decentralized oracle to reliably and transparently access the data.

4 **Attack vectors:** Lastly there were some attack-vectors, especially for the owner. The `startNewSeason()` function could be used at any point during the season, thus rendering unclaimed policyTokens worthless and not allowing for investment-withdrawals to be made.

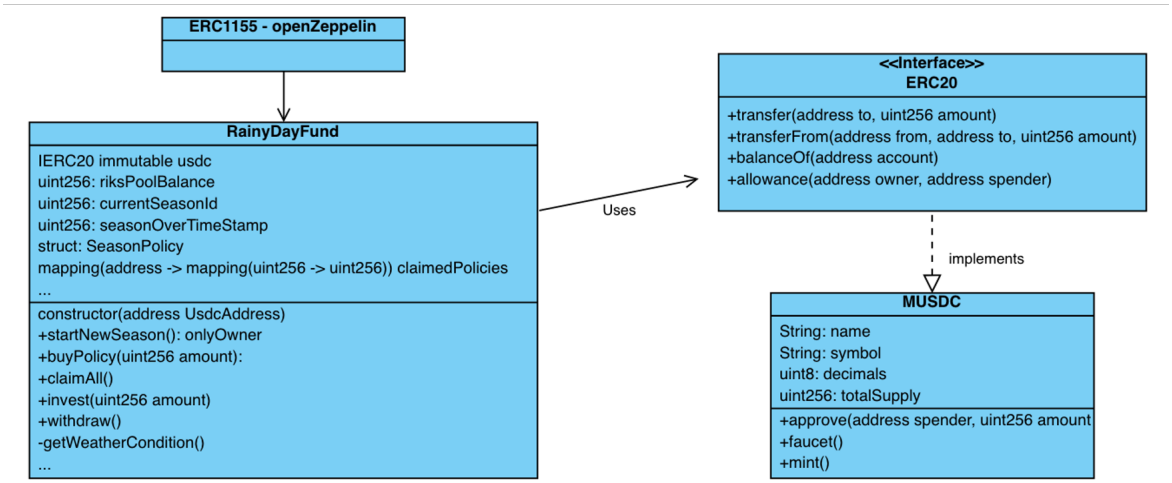

Figure 2: Initial Contract Design
Source: Author's own.

// Solving of issues -> explain new structure

3.4 Development Tools & Infrastructure

The development process utilized a modern toolchain centered around the Hardhat framework. We used this to develop, deploy and test our smart contracts. In detail, we used a javascript file to deploy our contracts. All of our tests were written in Typescript, and use the Node.js framework. Additionaly, we wrote a React based frontend. It provides an interface to manually test our contracts. Throughout our project, we used version control via git and explictly a GitHub repository. This made collaboration between team members flawless, and provided accountability measures in case of errors. For the smart contract development, we first used the Remix IDE. It allowed for easy deployment and manual testing. For anything that wasn't written in solidity, we used Neovim. Another big part of the project is the network which was used for the contract deployment. We initially planned on using the Sepolia testnet, but in the final sprint decided to only deploy it locally. The reason for this is a multitude of problems arising when we tried it via Sepolia. The furthest we got was deploying, but interacting turned out too difficult in this timeframe. The compomise was made: use local network only, but in return we managed to get a fully working prototype.

The test setup, as shown in Figure 3, highlights the importance of robust testing and code coverage in the project.

```
RainyDayFund
  Deployment
    ✔ Should set the right owner
    ✔ Should set the correct USDC address
    ✔ Should set the correct weather oracle
    ✔ Should initialize with season 1
    ✔ Should set correct initial season parameters
    ✔ Should initialize in ACTIVE state
    ✔ Should be in testing mode by default
    ✔ Should initialize season over timestamp correctly
    ✔ Should reject zero addresses in constructor
  Testing Mode Functions
    ✔ Should allow owner to advance through all phases
    ✔ Should correctly calculate time offset for each phase
    ✔ Should allow owner to toggle testing mode
    ✔ Should not allow non-owner to use testing functions
    ✔ Should return correct current time in testing mode
  Policy Purchase
    ✔ Should allow buying policies in ACTIVE state
    ✔ Should return correct season ID when buying policy
    ✔ Should reject zero amount purchase
    ✔ Should reject purchase when not in active period
    ✔ Should handle multiple policy purchases
    ✔ Should handle insufficient USDC balance
    ✔ Should handle insufficient allowance
  Weather Data and Claims
    ✔ Should read weather data from oracle
    ✔ Should allow claiming when weather conditions are met
    ✔ Should not allow claiming with good weather
    ✔ Should test weather threshold boundary
    ✔ Should not allow claiming in wrong periods
    ✔ Should not allow claiming without policies
    ✔ Should handle insufficient funds for claims
    ✔ Should prevent double claiming
  Investment Functions (ERC4626)
    ✔ Should allow investments using invest wrapper in ACTIVE state
    ✔ Should allow investments using direct ERC4626 deposit
    ✔ Should allow investments in INACTIVE period
    ✔ Should not allow investments outside ACTIVE/INACTIVE periods
    ✔ Should reject zero amount investment
    ✔ Should allow withdrawal during WITHDRAW period using redeemShares
    ✔ Should allow withdrawal using direct ERC4626 redeem
    ✔ Should not allow withdrawal outside WITHDRAW period
    ✔ Should calculate proportional returns with premium income
    ✔ Should handle ERC4626 preview functions correctly
    ✔ Should calculate correct share to asset ratios

  63 passing (3s)

----------------------|----------|----------|----------|----------|----------------|
File                  |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
----------------------|----------|----------|----------|----------|----------------|
 contracts/           |     98.7 |    92.31 |      100 |      100 |                |
  MockUSDC.sol        |      100 |      100 |      100 |      100 |                |
  MockWeatherOracle.sol |      100 |      100 |      100 |      100 |                |
  RainyDayFund.sol    |    98.65 |    92.31 |      100 |      100 |                |
----------------------|----------|----------|----------|----------|----------------|
All files             |     98.7 |    92.31 |      100 |      100 |                |
----------------------|----------|----------|----------|----------|----------------|
```

(a) Passing Tests and Coverage

```
describe("Deployment", function () {
  it("Should set the right owner", async function () {
    expect(await rainyDayFund.owner()).to.equal(owner.address);
  });

  it("Should set the correct USDC address", async function () {
    expect(await rainyDayFund.usdc()).to.equal(await mockUSDC.getAddress());
  });

  it("Should set the correct weather oracle", async function () {
    expect(await rainyDayFund.weatherFeed()).to.equal(await mockWeatherOracle.getAddress());
  });

  it("Should initialize with season 1", async function () {
    expect(await rainyDayFund.currentSeasonId()).to.equal(1);
  });

  it("Should set correct initial season parameters", async function () {
    const policyInfo = await rainyDayFund.seasonPolicies(1);
    expect(policyInfo.premium).to.equal(PREMIUM);
    expect(policyInfo.payoutAmount).to.equal(PAYOUT);
    expect(policyInfo.totalPoliciesSold).to.equal(0);
    expect(policyInfo.creationTimestamp).to.be.greaterThan(0);
  });

  it("Should initialize in ACTIVE state", async function () {
    expect(await rainyDayFund.getSeasonState()).to.equal(0); // ACTIVE
  });

  it("Should be in testing mode by default", async function () {
    expect(await rainyDayFund.testingMode()).to.equal(true);
    expect(await rainyDayFund.testingTimeOffset()).to.equal(0);
  });

  it("Should initialize season over timestamp correctly", async function () {
    const seasonOverTime = await rainyDayFund.seasonOverTimeStamp();
    const currentTime = await rainyDayFund.getCurrentTime();
    expect(seasonOverTime).to.be.greaterThan(currentTime);
  });

  it("Should reject zero addresses in constructor", async function () {
    const RainyDayFundFactory = await ethers.getContractFactory("RainyDayFund");
```

(b) Test Code

Figure 3: Test Setup. (a) Passing tests and coverage; (b) Excerpt of test code.
Source: Author's own.

3.5 Quality Assurance & Security

We started with Remix IDE because it's straightforward for learning Solidity - you can write, deploy, and test contracts right in the browser without any setup. This made it easy to experiment with basic functionality and get familiar with smart contract development. Once we had a working prototype, we moved to Hardhat. It comes with a local blockchain that simulates real network conditions, which made testing reliable. For testing, we used TypeScript to catch errors early and make the code easier to maintain. The tests run on Node.js with Mocha and Chai for assertions. We also created MockUSCD, an ERC20 token that behaves like USDC, so we could test everything without needing real money or external services. The frontend is built with React, and we used Git with GitHub for version control. This made collaboration smooth - we could work on different parts simultaneously without stepping on each other's toes. We originally planned to deploy on Sepolia testnet, but ran into issues with wallet connections and unpredictable gas costs that we couldn't solve in time. So we stuck with local deployment through Hardhat's network. This actually worked out well for demonstrations since everything runs predictably without external dependencies. Our test coverage reaches nearly 100% on critical functions, which helped catch bugs early when they were still easy to fix. The test setup shown in Figure 3 demonstrates how we can verify the contract works correctly across different scenarios.

3.6 Outlook & Next Steps

The most important next step is integrating real Chainlink oracles for weather data. Right now we're using mock data, but production needs reliable, tamper-proof weather information from multiple sources. We'll also need to deploy on a real network. Layer 2 solutions like Polygon or Arbitrum make sense because gas costs are much lower - important when your users are price-sensitive farmers. We might also look into cross-chain deployment to give users more options. The smart contract architecture needs some work for production. We'll want upgradeability mechanisms so we can fix bugs and add features, but we need to balance that with immutability for user trust. Gas optimization is another priority - the current contract works but isn't optimized for cost. For additional features, we're thinking about crop-specific insurance products and multi-season coverage. The current single-season model is too limiting for real-world agriculture. We'd also like to support group policies for farmer cooperatives. The frontend needs to be much more user-friendly, especially for people with limited digital experience. Mobile-first design is essential since most users will access this through smartphones. We'll need offline capabilities too since internet connectivity can be spotty in rural areas. Technical priorities include:

1. Chainlink integration for real weather data
2. Layer 2 deployment for lower costs
3. Gas optimization and contract upgrades
4. Mobile-optimized frontend with offline support
5. Multi-language support and simplified interfaces

We're also planning more comprehensive testing on testnets once we sort out the deployment issues. The current local testing is good for development, but we need real network conditions to catch problems we haven't thought of.

4 Conclusion

...

References

(NDMA), National Disaster Management Authority (2021). *NDMA 2021 (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

— (2024). *NDMA 2024 report (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

Agriculture, Ministry of (2023). *MoA 2023 (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

Artemis (2017). *Artemis 2017 (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

Baagoe (2020). *Baagoe 2020 (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

Bank, World (2022). *World Bank 2022 (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

— (2024). *Kenya Agriculture GDP*. Accessed 2024-09-19. URL: `https://data.worldbank.org/indicator/NV.AGR.TOTL.ZS?locations=KE`.

BASIS (2017). *BASIS 2017 (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

Chainlink (2021). *Chainlink 2021 (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

DHI, UNEP / (2021). *UNEP DHI 2021 (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

Dominguez (2024). *Dominguez 2024 (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

Ethereum (2025). *ERC-1155 Multi-Token Standard*. Last accessed: 20.09.2025. URL: `https://ethereum.org/de/developers/docs/standards/tokens/erc-1155/`.

Food and Agriculture Organization (2024a). *FAO 2024 (second reference) (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

— (2024b). *FAO 2024 report (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

Initiative, African Climate (2024). *African Climate 2024 (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

IUCN (2021). *IUCN 2021 report (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

Janzen (2020). *Janzen 2020 (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

Jensen (2016). *Jensen 2016 (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

Kenya, Government of (2017). *GoK 2017 document (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

OCHA (2023). *OCHA 2023 report (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

Oxford (2017). *Oxford 2017 (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

Shetty (2022). *Shetty 2022 (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

Sibiko (2018). *Sibiko 2018 (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

Star, The (2024). *The Star 2024 article (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

Wairimu (n.d.). *Wairimu (n.d.) (PLACEHOLDER)*. PLACEHOLDER: replace with full citation details.

Appendices

Add any supplementary material here.