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

Group A: Ellena, Sabina, Noah, Vincent

*University of Basel, Blockchain Challenge 25*

October 20, 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 Goals 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.

## 2    Introduction

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 (FAO 2024). 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 about 89% of the country's land area, host over 70% of livestock, and are home to around 36% of the population (IUCN n.d. Authority 2021; UNEP-DHI n.d.). 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 as farmers faced severe yield losses in many regions, 2.6 million livestock were lost, and 4.4 million people required urgent food assistance (The-Star 2023).

These shocks highlight the fragility of rural livelihoods and the absence of effective financial safety nets to protect farmers. This paper proposes a decentralized, blockchain-based parametric insurance model as a potential solution to bridge this gap. 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?

## 3    Problem Analysis

Smallholder farmers in Kenya are uniquely vulnerable to climate risks. About 98% of Kenya's agricultural systems are rain-fed (Government of Kenya and Fisheries 2017), while irrigation development is limited due to infrastructural and environmental constraints (Wairimu et al. 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 later seasons. The drought between 2020 and 2023, considered the worst in four decades, illustrates the magnitude of the challenge as farmers experienced severe yield losses in affected areas, more than 2.6 million livestock died, and about 4.4 million people required urgent food assistance (OCHA 2023). 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 Anguiano and Parte 2024). Claims are processed slowly and often manually, which delays payouts and worsens farmers' financial stress during crises (Chainlink 2021). A lack of transparency around pricing and claims fosters mistrust (Dominguez Anguiano

3

and Parte 2024). Moreover, coverage remains minimal, as fewer than 3% of farmers in Sub-Saharan Africa 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 (Jensen et al. 2016; Sibiko et al. 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 et al. 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 et al. 2016). Affordability is another barrier, since many farmers lack liquidity at the beginning of the planting season, precisely when premiums are due.

These systemic shortcomings are reflected in the lived experiences of farmers like *Mary*, a 38-year-old smallholder living in a semi-arid region of Kenya. Mary is married with three children and owns a basic Android phone, but she has no formal insurance. For her, every drought threatens both her harvest and her household income. When she does consider insurance, she struggles to afford premiums upfront, finds products too complex to understand, and doubts that payouts would be delivered fairly or on time. Even if she were to purchase a policy, she fears that unreliable weather data would not capture the realities of her farm. As she puts it, "Why should I pay if they don't measure my reality?" Mary's skepticism mirrors the sentiments of millions of smallholder farmers across Kenya who remain excluded from meaningful risk protection.

## 4 Benefits of Blockchain in Insurance

### 4.1 Blockchain & Decentralized Microinsurance

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 Anguiano and Parte 2024; Shetty et al. 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. Despite early adoption challenges, integrating blockchain into agricultural insurance provides a scalable, efficient, and reliable solution to climate risk, bridging gaps in coverage for underserved populations.

### 4.2 Problems with Traditional Insurance

Traditional insurance faces several critical challenges that limit its effectiveness, particularly in agricultural and microinsurance contexts. Transparency is often lacking, so policyholders cannot easily see how premiums are set, how claims are assessed, or why payouts are delayed. This lack of clarity fosters distrust between insurers and customers (Dominguez Anguiano and Parte 2024). Claims are also slow and mostly manual, requiring extensive paperwork and human assessment. These delays can worsen farmers' financial stress during climate shocks (Chainlink 2021; Shetty

et al. 2022). High administrative costs associated with record-keeping, fraud checks, and staffing are often passed on as expensive premiums, making insurance unaffordable for smallholder farmers (Dominguez Anguiano and Parte 2024). The reliance on self-reported or incomplete data further raises the risk of fraud and moral hazard, while detecting and preventing fraud requires significant resources (Chainlink 2021; Shetty et al. 2022). Finally, coverage remains extremely limited. In Sub-Saharan Africa, less than 3% of farmers are insured due to logistical, financial, and trust barriers, and the centralized structure of traditional insurance often makes it difficult for rural farmers to access services (Alsdorf and Berkun 2024; Bank 2022).

## 4.3 Blockchain Advantages

Blockchain technology helps address many of the weaknesses in traditional insurance by providing a decentralized, tamper-proof ledger that builds transparency and trust without relying on a central authority (Shetty et al. 2022). Smart contracts can automate claims and payouts, reducing the need for human involvement and lowering administrative costs (Chainlink 2021; Dominguez Anguiano and Parte 2024). Because records on the blockchain are immutable, they cannot be tampered with, which prevents fraudulent claims. Decentralized oracles also supply reliable and verifiable data for triggering insurance events. In addition, blockchain can connect easily with mobile wallets, allowing farmers in remote or underserved areas to access insurance services directly (Dominguez Anguiano and Parte 2024).

## 4.4 Benefits for Parametric & Microinsurance

Applying blockchain to parametric insurance enables automated, event-based payouts, reducing basis risk, enhancing speed, and improving trust among stakeholders (Chainlink 2021; *How blockchain is revolutionizing parametric insurance* 2025; Shetty et al. 2022). For microinsurance, blockchain improves cost efficiency by lowering policy management costs, making it possible to expand coverage to underserved farmers and to explore flexible pricing models (Dominguez Anguiano and Parte 2024; *Blockchain in insurance (preprint)* 2023). Real-world examples such as Etherisc and Arbol already show how this can work in practice, with thousands of farmers receiving rapid payouts without the need for manual claims assessments (Alsdorf and Berkun 2024; Dominguez Anguiano and Parte 2024; *How blockchain is revolutionizing parametric insurance* 2025). The combination of decentralization, automation, and verifiable data ensures that parametric microinsurance is transparent, scalable, and trustworthy (Shetty et al. 2022).

## 5 Target Market & Stakeholders

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 about 89% of the country's land area, host over 70% of the livestock population, and are home to roughly 36% of the national population (IUCN n.d. Authority 2021; UNEP-DHI n.d.). With more than 98% of agriculture dependent on rain-fed systems, smallholder farmers are disproportionately exposed to climate variability and shocks (Government of Kenya and Fisheries 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 (FAO 2024; Ministry of Agriculture 2023).

The stakeholders in Kenya's agricultural insurance ecosystem are diverse and interdependent. Farmers are the primary end-users, seeking affordable and trustworthy protection against climate risks. Insurers and micro-insurers, such as APA Insurance, UAP, CIC, Jubilee, and Kenya Orient, underwrite and distribute weather-index products, while global reinsurers like Swiss Re provide the financial backing needed for large-scale coverage (Artemis 2017; BASIS 2017). The Government of Kenya, through its National Agricultural Insurance Policy (NAIP), and regulatory agencies such as the Insurance Regulatory Authority (IRA) and the National Drought Management Authority (NDMA), sets regulations and coordinates drought response (African-Climate-Foundation 2024; Ministry of Agriculture 2023). International donors and development partners, including the World Bank, IFC, and FAO, subsidize premiums, and provide technical support, as seen in the Kenya Livestock Insurance Program (KLIP) (Artemis 2017; Bank 2022). Finally, mobile money services such as Safaricom's M-Pesa make it possible to collect premiums and deliver payouts directly to farmers (*Seeds of growth: Agriculture presents a range of options for insurers* 2017).

# 6 Market Research & Competitor Analysis

Agricultural insurance penetration in Kenya remains extremely low, even though agriculture forms the backbone of the economy. The Government's National Agricultural Insurance Policy (NAIP) highlights that less than 1% of farmers and pastoralists purchase agricultural insurance, leaving the vast majority of producers exposed to climate shocks without any formal safety net (Ministry of Agriculture 2023). Across Sub-Saharan Africa, the picture is similar. There are about 48 million smallholder farmers, yet fewer than 1.5 million are insured. This means that only around 3% have coverage, leaving 97% unprotected, despite an annual demand estimated at USD 10–14 billion compared to a premium pool of only USD 1–3 billion (Africa 2020). These figures highlight an enormous protection gap, both in Kenya and across the region, where millions of rural households face severe climate risks without reliable financial buffers.

Despite these low figures, recent pilots suggest that demand exists when products are designed to be affordable, transparent, and accessible. For instance, the World Bank–supported One Million Farmers Platform has enabled over 70,000 smallholders across 15 counties to access weather-index insurance, primarily through digital delivery channels and bundled products. This suggests that the barrier is not farmer willingness but the way insurance is designed and distributed (Bank 2022).

On the supply side, Kenya already offers both livestock and crop index solutions. The Kenya Livestock Insurance Program (KLIP) is a leading public–private partnership providing drought insurance in ASAL counties. It is run by a consortium of local insurers led by APA (together with UAP, CIC, Jubilee, Heritage, Amaco, and Kenya Orient) and supported by Swiss Re as reinsurer. KLIP has successfully delivered multi-million-shilling payouts when drought thresholds were met (Artemis 2017; BASIS 2017). KLIP demonstrates proof of concept for index insurance in Kenya, showing that government support, consortium underwriting, and reinsurer participation can combine to deliver payouts at scale.

In the crop sector, ACRE Africa and Pula are the leading intermediaries. ACRE Africa has developed weather-index products that are increasingly bundled with agronomic services, such as

advisory tools and crop-cutting experiments, to improve product value and data accuracy (Africa 2024). Pula, by contrast, has built scale across Africa by bundling insurance with seeds, fertilizer, and other farm inputs, often distributed through governments and value-chain partners. Its use of satellite and remote sensing data has enabled broad coverage, though it continues to face challenges with basis risk and farmer liquidity constraints at planting (*Pula learning session: Zambia case* 2022; UNSGSA 2023).

More recently, blockchain-based models have entered the Kenyan market. Etherisc, in collaboration with ACRE Africa, piloted a "Bima Pima" crop cover in 2021, reaching over 17,000 farmers and delivering the first blockchain-based payouts in Kenya. By embedding smart contracts and decentralized oracles into the insurance process, Etherisc was able to automate claim settlement and reduce administrative overhead (Etherisc 2021). Although small in scale, this pilot shows how blockchain can help reduce long-standing problems in agricultural insurance, especially those related to trust, transparency, and timely payouts.

OKO Finance, for example, has demonstrated in Mali and Uganda how mobile-first, index-based agriculture insurance can be structured using satellite or remote sensing data and delivered through digital channels (ADA and Finance 2021). Although OKO is not known to be active in Kenya at the moment, its model suggests how digital ecosystems could lower distribution costs and expand access to underserved farming populations.

In short, agricultural insurance in Kenya is still underdeveloped, with less than 1% of farmers insured. However, programs like KLIP and private-sector innovators such as ACRE Africa and Pula show that growth is possible. Farmers still face major barriers, including basis risk (Jensen et al. 2016), limited awareness and perceived complexity (Janzen et al. 2020), and upfront liquidity constraints all reduce trust and limit uptake. Early blockchain pilots such as Etherisc's indicate promising pathways to reduce administrative costs and improve transparency, but scaling such solutions will depend on robust data infrastructure, effective oracles, and farmer-centric design. Overall, the market reflects both a large protection gap and a significant opportunity, as innovative, technology-enabled models that combine affordability, transparency, and mobile integration have the potential to transform risk management for Kenya's smallholder farmers.

# 7 Business Model

## 7.1 Overview

The Rainy Day Fund is a two-sided parametric insurance marketplace. Farmers obtain seasonal drought protection through ERC-20 policy tokens, while investors provide liquidity to an ERC-4626 vault and receive proportional shares. A smart contract manages issuance, pooling and payouts. The design reduces administrative overhead and shifts pricing and risk allocation to a fully market-based mechanism.

## 7.2 Problem and Customer Segments

Smallholder farmers in Kenya face recurrent rainfall shocks but lack access to formal insurance. High distribution costs, weak trust in insurers and limited actuarial data prevent traditional coverage. At the same time, impact-oriented investors seek transparent, uncorrelated instruments.

The marketplace connects both groups, while cooperatives, NGOs and telcos provide distribution and trust infrastructure (Cole et al. 2013).

## 7.3 Value Propositions

- **Farmers**: affordable coverage, predictable payouts triggered by rainfall indices, and mobile transactions via M-Pesa.

- **Investors**: standardized exposure to agricultural risk with measurable yield, diversifying traditional portfolios.

- **Partners**: reduced leakage and transparent performance metrics for evaluation and reporting.

## 7.4 Solution Design

Farmers purchase ERC-20 tokens as digital proof of coverage. Investors deposit capital into an ERC-4626 vault and receive fungible shares. Oracle-based rainfall indices determine payout eligibility. Premiums and yields are discovered through a pre-season sealed-bid uniform-price auction functioning as a prediction market: farmer bids and investor offers jointly reveal the market-implied drought probability $\hat{\pi}$ and equilibrium margin (Hanson 2003; Wolfers and Zitzewitz 2004). In the current prototype, the premium is owner-set at season start (no on-chain auction). The frontend abstracts blockchain logic and is designed for fiat–stablecoin integration.

## 7.5 Revenue Model

Premiums collected from farmers fund the risk pool. Investor returns correspond to the surplus of premiums over payouts, distributed proportionally through vault shares. Operational costs are deducted transparently by the protocol *after* settlement. Because investors anticipate this deduction when bidding, premiums remain entirely market-determined. This ensures that the auction mechanism itself is unaffected by fees, while the platform sustains infrastructure, audits, and compliance.

The operator may also allocate its own capital into the pool. This dual role enhances credibility and creates an additional revenue stream if claims remain below expectations, though it increases exposure to volatility. Optional reinsurance transfers catastrophic tail risk to external providers and stabilizes investor yields (Cummins 2009).

ERC-4626 shares accumulate value as premiums flow in and payouts are deducted. Surpluses are automatically reinvested into vault NAV, producing a compounding effect similar to conventional finance. Payouts require a minimal on-chain claim, after which the contract settles automatically if the rainfall threshold is breached (see Claim Process and Payout Logic).

## 7.6 Cost Structure

Key cost drivers are oracle services, secure data feeds, contract audits, regulatory engagement, payment integration, and farmer outreach. Automation keeps administrative costs substantially lower than in conventional microinsurance. As scale increases, marginal costs decline, generating economies of scale.

## 7.7 Channels and Partnerships

Cooperatives, NGOs, and agro-dealers provide farmer onboarding and trust. M-Pesa integration enables direct premium and payout transactions. Oracle providers deliver rainfall indices (Egberts 2017). Reinsurers supply capacity for catastrophic events. Regulators and development agencies support compliance and legitimacy.

## 7.8 Scalability

The model can expand from Kenyan maize to other crops, regions, and perils such as livestock mortality or flood. ERC standards allow interoperability with DeFi protocols and secondary markets for risk exposure. The auction mechanism improves with scale, as more participants sharpen price discovery. Modular system design supports adaptation across regulatory environments.

## 7.9 Claim Process and Payout Logic

Farmers must file a minimal on-chain claim (notice-of-loss) during a specified window $[t_0, t_1]$ after oracle publication. The contract verifies:

1. ownership: policy tokens $q > 0$ for the epoch,

2. uniqueness: no prior claim from the same address/epoch,

3. eligibility: oracle index $I$ satisfies $I \leq \tau$.

If (1)–(3) hold, the contract transfers $qL$ to the claimant's wallet; otherwise the call reverts. In the analytical model, pro-rata rationing applies; in the prototype, claims are settled directly without deductions. No off-chain adjusters or discretionary assessments occur.[1]

# 8 Mechanism Design

*Scope note: The following mechanism specifies the market design used in our economic model. The prototype does not implement on-chain bidding (commit–reveal) or automated auction clearing; seasonal premiums are owner-set in the current implementation.*

Premiums and investor yields are discovered through a sealed-bid uniform-price auction. This mechanism functions as a prediction market for drought risk while remaining implementable under smallholder constraints (Wilson 1979; Ausubel and Cramton 2002).

## 8.1 Notation

Let $x$ denote the number of insurance tokens sold, with $Q$ referring to the total seasonal quantity ($Q = x$ at equilibrium). Each token promises a payout $L$ if drought occurs, represented by $X \in \{0, 1\}$. Farmers pay a premium $p$ per token.

We define $\hat{\pi}$ as the *market-implied probability of drought*, inferred endogenously from the auction clearing. Premiums decompose into $\hat{\pi}L$ and the risk margin $\mu$. Investor capital is $K$, realized return $R$, and expected return $R^\star$. Farmer inverse demand is $p_d(x)$; investor supply is $C_s(R)$. Required reserves are $C_{\text{req}}(x)$ with prudential factor $\theta \in (0, 1]$.

---

[1]A claim call is required to (i) prevent automatic transfers to inactive wallets, (ii) avoid dust payouts below gas costs, and (iii) let farmers select their preferred payout rail (e.g. M-Pesa bridge).

## 8.2 Auction Process and Clearing Mechanism

Farmers submit bids $(q_i, p_i)$; investors submit $(k_j, R_j)$. Bids are committed via a commit–reveal protocol (Blum 1983; Buterin 2015; Parkes and Seuken 2014), ensuring confidentiality until revelation.

Feasibility requires

$$C_{\text{req}}(x) = xL\theta \le S(R), \quad x \le D(p).$$

Investor cashflow is

$$CF = pQ - XQL,$$

with realized return

$$R = \frac{CF}{K}, \quad R^\star = \frac{pQ - \hat{\pi}QL}{K}.$$

The clearing premium is

$$p^* = \min\{p_i : \sum_i q_i \ge x^*\},$$

and equilibrium return $R^*$ follows from the cashflow identity. All accepted farmers pay $p^*$, all accepted investors receive $R^*$. If demand exceeds feasible coverage, allocations are scaled proportionally:

$$q_i^{\text{alloc}} = \rho q_i, \qquad \rho = \frac{x^*}{\sum_i q_i}.$$

## 8.3 Premium Decomposition and the Risk Margin

The clearing premium decomposes ex post into

$$p^* = \hat{\pi}L + \mu,$$

where $\mu$ is the margin implied by bids. It follows that

$$\mu = \frac{R^\star K}{Q}.$$

After settlement, the protocol may deduct a fee $F$ from the total pool for operational costs. This deduction does not affect price formation or investor bidding; it is applied ex-post and anticipated in bids.

## 8.4 Equilibrium Properties

Auction outcomes satisfy

$$p^* = p_d(x^*), \qquad C_s(R^*) = C_{\text{req}}(x^*).$$

**Monotonicity.** If farmer demand increases, $p^*$ weakly increases. If investor supply increases, $R^*$ weakly decreases.

**Existence and uniqueness.** Under standard monotonicity, the auction admits a unique clearing outcome $(x^*, p^*, R^*)$.

**Efficiency.** The full coverage rule maximizes feasible farmer coverage given investor constraints.

**Incentives.** Uniform-price auctions are not fully strategy-proof; large bidders may attempt demand reduction (Ausubel and Cramton 2002). Mitigation includes sealed bids, per-farmer caps, and pro-rata rationing.

## 8.5 Limitations and Future Research

Limitations remain. Functional forms for demand and supply require calibration. $\hat{\pi}$ must be validated against actuarial or climate models. Uniform-price auctions remain vulnerable to demand reduction and collusion. Commit–reveal protocols face risks of non-reveal failures. Future work should calibrate with field data, add anti-collusion safeguards, and test dynamic or continuous models as counterfactuals.

# 9 Implementation

Overview of implementation goals, strategy and architecture.

The goal for this project is 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 9.7 and 9.9. 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.

## 9.1 Prototype Scope: Model vs. Implementation

To avoid ambiguity, this subsection summarizes the economic model versus the implemented prototype.

## 9.2 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]See: https://v2.hardhat.org/docs

Table 1: Economic Model vs. Prototype Implementation

| Feature | Economic Model | Prototype (This Repo) |
|---|---|---|
| Premium discovery | Sealed-bid uniform-price auction (commit–reveal) | Owner-set per season (no on-chain auction) |
| Claims flow | Minimal claim; pro-rata possible; fees deducted | Explicit claim call; no pro-rata; no fees; revert if insufficient funds |
| Payout automation | Event-based automatic settlement | Automated settlement upon claim call when oracle threshold met |
| Oracle data | Chainlink or equivalent, multi-source | Mock oracle (AggregatorV3Interface-compatible) |
| Policy tokens | ERC-20 per season | ERC-20 per season (transferable) |
| Risk pool | ERC-4626 vault | ERC-4626 vault implemented |
| Investor redemption | Flexible per design | Restricted to WITHDRAW phase |
| Fiat/mobile rails | M-Pesa bridge integration | Not implemented (UI designed for future integration) |
| Reinsurance | Optional | Not implemented |
| Deployment | Public testnet/mainnet | Local Hardhat network[2] for prototype |

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 **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 9.9.

## 9.3 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.
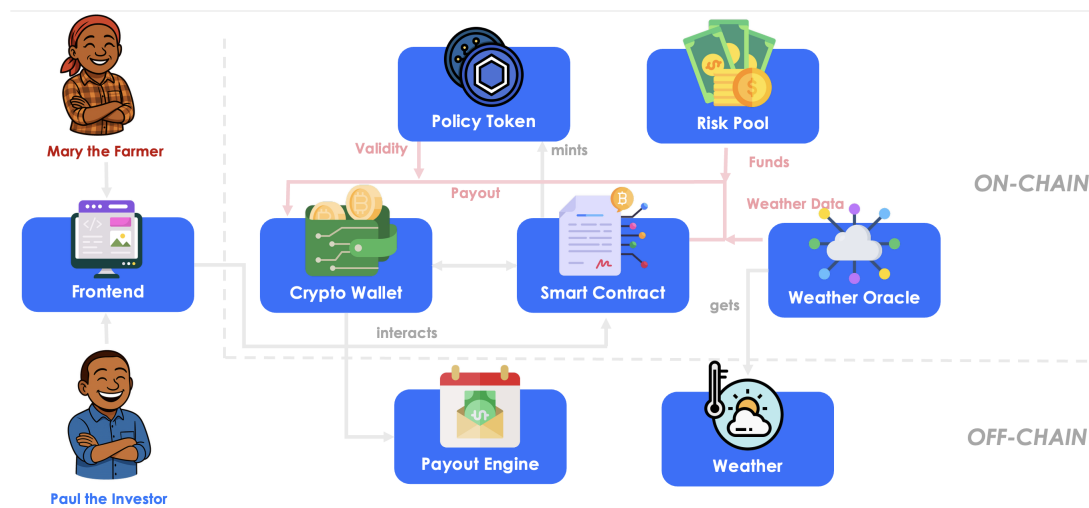


Figure 1: Architectural Overview
*Source: Author's own.*

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 ERC-4626 vault). The farmers can claim their policies at the end of the season and, if all conditions are met, i.e. the weather oracle 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 set at the beginning by the owner. 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.

## 9.4 Development Tools & Infrastructure

The development process utilized a modern toolchain centered around the Hardhat framework for smart contract development, deployment, and testing. Initial contract development was performed in the Remix IDE, which enabled rapid deployment and manual testing directly in the browser without requiring local setup. After a working prototype was established, development transitioned to Hardhat, providing a more robust environment with a local blockchain that simulates real network conditions. The frontend was built using React, serving as an interface to test smart contract interactions and demonstrate overall system functionality.

Version control was managed through Git, with collaboration facilitated via a GitHub repository. This setup ensured transparency, accountability, and effective team coordination throughout the project. Deployment was initially planned for the Sepolia Testnet[3], but the final sprint used only a local network due to recurring issues with contract interaction on Sepolia. This compromise allowed for the delivery of a fully functional prototype that operates predictably without external dependencies.

Development environments included the Remix IDE[4] for Solidity-based work and Neovim for general-purpose coding tasks. All deployment tasks were executed through JavaScript scripts within the Hardhat framework.

## 9.5 Initial Contract Design

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 `redeemShares()` methods while insurees can buy and claim their policies in batches using the `buyPolicy()` and `claimPolicies()`

---

[3]Test network for Ethereum; for more information refer to https://getblock.medium.com/what-is-sepolia-a-beginners-guide-to-ethereum-test-networks-866663a26698

[4]An online IDE for developing solidity smart contracts. For documentation see: https://remix-ide.readthedocs.io/en/latest/run.html#remix-vm, the IDE can be found at: https://remix.ethereum.org

functions. The contract leverages ERC-20 season policy tokens, enabling simple transfers and season-scoped policy management.

As shown in Figure 2, the initial contract made use of the `ERC-1155 token standard` for the `policyTokens`, to take advantage of the cost-efficient batch operations (Ethereum 2025). To help with testing the (Mock) USDC[5] was created, implementing the ERC20 interface, to emulate the behavior of common stablecoins like USDC, which are used as payment and funding for the risk pool. 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.
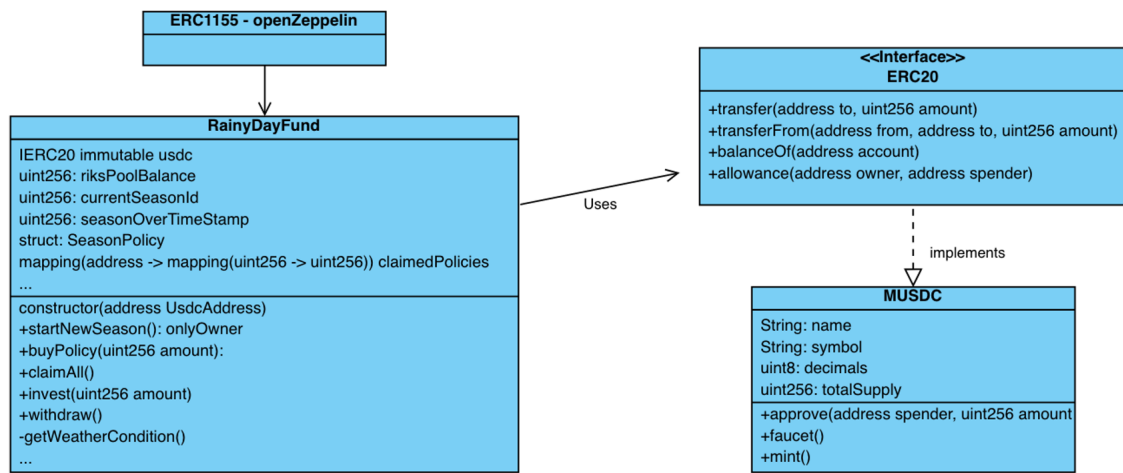


Figure 2: Initial Contract Design
*Source: Author's own.*

---

[5]One of the largest stablecoins worldwide, well accessible and low cost, see: https://developers.circle.com/stablecoins/what-is-usdc

Initially there were multiple issues:

1 **Mapping for storing addresses:** Initially, a mapping was used 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:** It was already planned to give investors an incentive to keep their funds in the risk pool, 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.

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. 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 policy tokens worthless and not allowing for investment withdrawals to be made.

Addressing these issues of the initial prototype the contract was restructured. The main fixes were:

Table 2: Major Issues and Corresponding Fixes

| Issue | Fix |
|---|---|
| Mapping for storing addresses was gas-inefficient and did not track policy resales | Removed the mapping |
| No proper incentive for investors and difficulties with implementation | Switched from self-made investment tracking to using the `ERC4626 vault` |
| Using Token Standards ERC4626, ERC20 and ERC1155 simultaneously, causing errors with inheritance of the contract | Switched from ERC1155 to only using ERC20 for the policy tokens |
| Weather Oracle / Chainlink integration | Implementing the proper Chainlink standards for the mock-up weather oracle (Chainlink n.d.) |
| Attack Vector for Owner | Fixed some issues by including modifiers and properly implementing a state machine (refer to enumeration 5) |

## 9.6 Implementation: Final State of the project

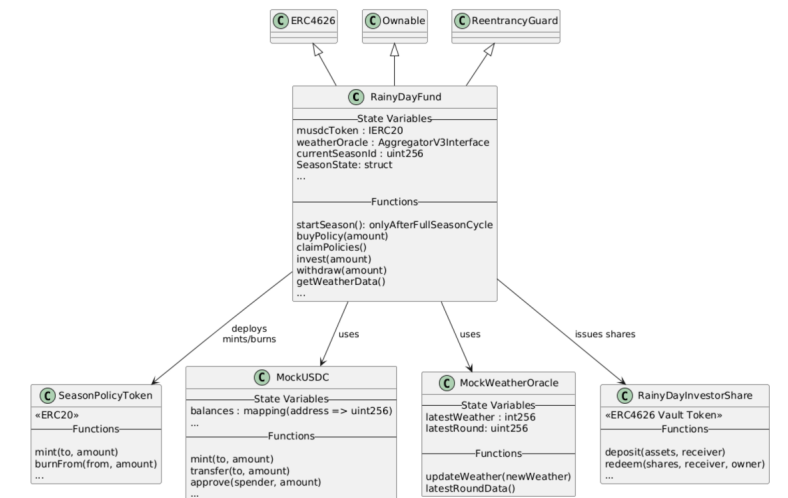The final state of the contract can be seen in Figure 3



Figure 3: Final Contract Design
*Source: Author's own.*

This class diagram only shows the most important functions and fields of the contract. The full code can be found in the appendix A along with some notes on its functionality.

The constructor takes 2 arguments:

(1) **usdcAddress:** The address of the stablecoin that will be used for the transactions, here this would be the address of the deployed MockUSDC contract

(2) **weatherOracle:** The address of the weather data oracle service, here this would be the address of the deployed MockWeatherOracle contract, but in production should be the corresponding Chainlink address.

Both are required to not be zero and used to set the corresponding fields for further use. Then the first season is initialized.

As shown in listing 1, the contract utilizes a SeasonState enum to keep track of the phase that is currently active in the state machine. The `getSeasonState()` method is used as the state machine and utilizes the `getCurrentTime()` method. The choice to implement the time this way, instead of simply using the `block.timestamp` was made to allow for time manipulation in testing. That is why there is a testingTimeOffset being added to the timestamp. This offset is calculated and set in a method of its own, that is only used for testing and showcasing reasons and would be removed for production.

```solidity
enum SeasonState { ACTIVE, INACTIVE, CLAIM, WITHDRAW, FINISHED }
function getCurrentTime() public view returns (uint256) {
  if (testingMode) {
    return block.timestamp + testingTimeOffset;
  }
  return block.timestamp;
}
  function getSeasonState() public view returns (SeasonState) {
      uint256 currentTime = getCurrentTime();
      if (currentTime < seasonOverTimeStamp - timeUnit) {
          return SeasonState.ACTIVE;
      } else if (currentTime < seasonOverTimeStamp) {
          return SeasonState.INACTIVE;
      } else if (currentTime < seasonOverTimeStamp + timeUnit) {
          return SeasonState.CLAIM;
      } else if (currentTime < seasonOverTimeStamp + 2 * timeUnit) {
          return SeasonState.WITHDRAW;
      } else {
          return SeasonState.FINISHED;
      }
      }
  modifier onlyAfterFullSeasonCycle() {
      require(getSeasonState() == SeasonState.FINISHED, "Season not fully
 finished yet");
      _;
      }
  function startNewSeason(uint256 _premium) external onlyOwner
 onlyAfterFullSeasonCycle {
      currentSeasonId++;
      premium = _premium;
      seasonOverTimeStamp = getCurrentTime() + 2 * timeUnit;
      _initializeSeason(currentSeasonId);
      }

```

Listing 1: Excerpt: RainyDayFund.sol - StateMachine

One example of how this gets used, can be seen in the modifier `onlyAfterFullSeasonCycle()`, which requires that the current season state is "FINISHED". This modifier is for example used for the `startNewSeason()` function, which initializes a new season with the given premium. The function should only be called once the last season has finished, to avoid policies not being claimable or investors not being able to withdraw their funds.

The most important functions for the farmers are `buyPolicy()` and `claimPolicies()`. They allow for purchasing and filing claims while checking all necessary conditions.

```solidity
function buyPolicy(uint256 _amount) external nonReentrant returns (uint256 seasonId) {
    require(_amount > 0, "Amount > 0");
    require(getSeasonState() == SeasonState.ACTIVE, "Not in active period");

    SeasonPolicy storage policy = seasonPolicies[currentSeasonId];

    uint256 totalPremium = policy.premium * _amount;
    require(usdc.transferFrom(msg.sender, address(this), totalPremium), "Transfer failed");

    SeasonPolicyToken(address(policy.policyToken)).mint(msg.sender, _amount);
    policy.totalPoliciesSold += _amount;

    emit PolicyBought(msg.sender, currentSeasonId, _amount, totalPremium);
    return currentSeasonId;
    }

    function claimPolicies() external nonReentrant {
        require(getSeasonState() == SeasonState.CLAIM, "Not in claim period");

        SeasonPolicy storage policy = seasonPolicies[currentSeasonId];
        SeasonPolicyToken token = SeasonPolicyToken(address(policy.policyToken));
        uint256 amount = token.balanceOf(msg.sender);
        require(amount > 0, "No policies to claim");

        (,int256 weather,) = getWeatherData();
        require(uint256(weather) < 10, "Weather not bad enough");

        uint256 totalPayout = policy.payoutAmount * amount;
        require(usdc.balanceOf(address(this)) >= totalPayout, "Insufficient funds");

        require(usdc.transfer(msg.sender, totalPayout), "Payout failed");
        token.burnFrom(msg.sender, amount);

        emit ClaimMade(msg.sender, currentSeasonId, amount, totalPayout);
    }
```

Listing 2: Excerpt: RainyDayFund.sol - Farmer Functions

The `buyPolicy()` function takes in the amount of `policyTokens` the user wants to buy and returns the current seasonId. This id could be used to display metadata about the season in the front end. The amount needs to be above zero and the season needs to be in the active phase. The premium gets calculated based on the amount and the premium stored in the SeasonPolicy struct, that gets set at the beginning of the season. After the transfer of the funds the `policyTokens` get minted to the user's wallet and the corresponding event gets emitted.

The `claimPolicies()` function requires the season to be in the claim phase and claims all available seasonPolicy tokens in the user's wallet. It checks whether the weather condition is met through the weather oracle with the `getWeatherData()` function and calculates the total payout, based on the fixed payout per token and the amount of tokens. Lastly the payout is made, the tokens are burned and an event gets emitted.

Both functions are secured by a reentrancy guard as a security best practice, because unlike in the prototype setup, in production the stablecoin contract used is not directly controllable and there might be some malicious code or even potential flash loan attacks, where the attacker attempts to manipulate the contract state through rapid sequential interactions. In both cases the tokens are minted/burned only after the transfer was successful to avoid minting/burning should the transfer fail.

Lastly the investment logic is simplified a lot by using the `ERC4626 token vault` standard:

```
1    // ERC4626 investment logic
2    function invest(uint256 assets) external nonReentrant onlyDuringSeason {
3      require(assets > 0, "Amount > 0");
4      deposit(assets, msg.sender);
5      emit InvestmentMade(msg.sender, assets);
6      }
7
8      function redeemShares(uint256 shares) external nonReentrant {
9        require(getSeasonState() == SeasonState.WITHDRAW, "Not in withdrawal
   period");
10       uint256 assets = redeem(shares, msg.sender, msg.sender);
11       emit InvestmentWithdrawn(msg.sender, assets);
12       }
13
```

Listing 3: Excerpt: RainyDayFund.sol - Investor Functions

The `invest()` function simply deposits the assets in the vault, requiring the amount to be above zero and the `redeemShares()` function redeems the user's shares again. All the transfers are handled by the vault implementation. The main customization is the `onlyDuringSeason` modifier for the `invest()` function and the requirement for the "WITHDRAW" phase in the `redeemShares()` function, making sure investments are not made during the claim or withdraw phases and the withdrawals can only happen in the corresponding phase.

Some information about the `SeasonPolicyToken`, `MockUSDC` and `MockWeatherOracle` contracts can be found in the appendix B.

## 9.7 Quality Assurance & Security

Quality assurance focused on comprehensive testing and systematic validation of contract functionality. All tests were written in TypeScript using the Node.js framework with Chai for assertions, enabling early error detection and improving maintainability. To ensure complete testing without reliance on external dependencies, a mock ERC20 token, `MockUSDC`, was implemented to emulate USDC behavior. This approach enabled full control over the testing environment and facilitated the verification of payment and funding logic within isolated virtual machine conditions.

The test setup, shown in Figure **??**, emphasizes the importance of robust testing and code coverage in the project. Test coverage reaches nearly 100% for critical functions, allowing bugs to be identified and resolved early in the development cycle. The comprehensive test suite verifies correct contract behavior across a variety of scenarios, including edge cases and potential attack vectors.



(a) Passing Tests and Coverage
*Source: Author's own.*

## 9.8 Problems and Remaining Questions

During implementation of the RainyDayFund contract, a subtle interaction between policy purchases and investor deposits was identified, stemming from the OpenZeppelin ERC4626 vault's design for handling external asset inflows.

Specifically, when policies are purchased before any investments, the USDC premiums are transferred directly into the vault via `buyPolicy()`, increasing `totalAssets()` without minting corresponding RDIS shares. This leaves `totalSupply()` at zero.

Subsequent calls to `invest()`, invoking ERC4626's `convertToShares()` function to compute share issuance:

$$\text{shares} = \text{assets} \times \frac{\text{totalSupply()} + 10^{\text{decimals offset}}}{\text{totalAssets()} + 1}$$

where the decimals offset is 12 (RDIS[6]at 18 decimals, USDC at 6). The $10^{12}$ term acts as a "virtual supply" to bootstrap fair pricing and normalize decimals.

With no prior shares, this heavily dilutes the first deposit: for a 1 USDC investment after 9 USDC in premiums, shares $\approx 1.11 \times 10^{11}$ raw units, or $\sim 1.11 \times 10^{-7}$ whole RDIS tokens—appearing as "quasi-null" in the React UI . Later investments face similar dilution until real supply dominates the virtual offset.

This is not a bug but intended ERC4626 behavior (per EIP-4626[7]) to prevent windfalls from external transfers, ensuring atomic fairness: depositors receive exactly their assets' worth upon immediate redeem. However, it creates order dependence in this flow—invest-first avoids the issue—and cosmetic UI problems, misaligning with the goal of seamless pooling where premiums back the fund and investments capture retained upside (NAV[8]$> 1$ on low claims) or share dilution (NAV $< 1$ on payouts).

To address this, the following measures are proposed:

1. In `buyPolicy()`, after transferring premiums, mint proportional protocol shares to the owner at the bootstrap rate ($10^{12}$ per USDC), ensuring whole-scale issuance for all deposits.

2. In `redeemShares()` (WITHDRAW phase only), auto-trigger a one-time redemption of protocol shares on the first investor withdrawal, donating proceeds back as external assets (no new shares minted).

This realizes upside trustlessly: the first withdrawer enables full NAV for all, with negligible extra gas ($\sim$20k).

Trade-offs include: if no withdrawals occur, retained premiums remain unrealized (fund rolls over); concurrent first withdrawals are serialized fairly by Ethereum. Implementation is deferred due to time constraints and audit needs, and the order dependence is documented and avoided in deployment.

## 9.9 Future Work

The most important next step is the integration of real Chainlink oracles for weather data. The current implementation uses mock data, but production requires reliable, tamper-proof weather information from multiple sources. Deployment on a real network is also necessary. Layer 2[9] solutions such as Polygon or Arbitrum [10] are suitable due to significantly lower gas costs, which is essential for price-sensitive users such as farmers. Cross-chain deployment should also be explored to provide users with additional flexibility.

---

[6]RainyDayFund Investor Shares - the shares investors get for their capital, as implemented in the ERC-4626 standard

[7]See: https://eips.ethereum.org/EIPS/eip-4626

[8]Net Asset Value, refer to e.g. https://www.investopedia.com/terms/n/nav.asp

[9]Refer to: (Song et al. 2024)

[10]https://blog.arbitrum.io/a-guide-to-understanding-blockchain-layer-2s/

The smart contract architecture requires further refinement for production readiness. Upgradeability mechanisms must be introduced to enable future bug fixes and feature additions while maintaining a balance with immutability to preserve user trust. Gas optimization remains a key priority, as the current version functions correctly but is not cost-efficient.

Future functionality may include crop-specific insurance products and multi-season coverage, addressing the limitations of the current single-season model. Additional features such as group policies for farmer cooperatives would further increase system usability.

The frontend requires enhanced usability, particularly for users with limited digital experience. A mobile-first design is essential, as most users will access the platform via smartphones. Offline functionality should also be included to accommodate rural regions with inconsistent internet connectivity.

**Technical priorities include:**

1. Chainlink integration for real weather data

2. Layer 2 deployment for reduced transaction costs

3. Gas optimization and upgradeable contract architecture

4. Mobile-optimized frontend with offline support

5. Multi-language support and simplified user interfaces

More extensive testing on public testnets is planned once deployment issues are resolved. While the current local testing setup provides an effective development environment, testing under real network conditions is essential to uncover issues that may not appear in isolated simulations.

# 10 Conclusion

Blockchain-enabled parametric microinsurance is not a silver bullet for climate risk, but this work shows it can remove long-standing frictions that kept protection out of reach for Kenyan smallholders. The approach begins from a simple premise: if pricing, capital formation, and payouts are automated around objective weather signals, then insurance can become cheaper, faster, and more trustworthy. The design proposes a sealed-bid, uniform-price mechanism for seasonal price discovery alongside tokenized coverage and a pooled ERC-4626 vault, and a test-covered prototype was implemented that executes end-to-end flows with deterministic, auditable logic. The auction mechanism remains part of the model and is not yet implemented on-chain.

Three results stand out.

First, automation meaningfully reduces operational complexity: premiums, coverage, and payouts are settled by code with transparent rules and verifiable data, minimizing discretion and delays.

Second, market-based clearing aligns farmer affordability and investor return ex ante, translating beliefs about drought into prices and capacity in a way that scales with participation.

Third, modularity matters: by standardizing payments, tokens, and oracles, the system can plug into mobile rails (e.g., M-Pesa bridges), real weather feeds, and reinsurance capacity without redesigning core contracts.

Important limitations remain. Basis risk must be reduced through better indices (multi-source rainfall, satellite vegetation, crop models) and local calibration. Real oracle infrastructure, L2 deployment, gas optimization, and upgrade paths are prerequisites for production. Governance and consumer protection (fair disclosures, dispute windows, orderly season transitions) must be specified and audited. Finally, the ERC-4626 interaction between external premiums and share issuance requires a production-grade policy to avoid confusing UX and to preserve fair NAV accounting.

The path forward is practical and testable: pilot one crop and county with a cooperative partner, integrate Chainlink (or equivalent) multi-source weather feeds, settle on a conservative claim threshold with an appeal buffer, and distribute through trusted local channels. Use donor or public co-funding to subsidize early seasons, measure realized basis risk and take-up, and iterate the auction parameters with real demand data. On-chain transparency should be paired with mobile-first UX and multilingual support to build confidence.

In sum, the Rainy-Day Fund demonstrates that decentralized infrastructure can make parametric insurance simpler, fairer, and more credible for farmers like Mary—converting climate shocks into predictable cashflows when they matter most. With robust data, careful governance, and field partnerships, the prototype can evolve into a resilient safety net that scales beyond a single crop or region while remaining transparent and accountable by design.

The entire project can be found in the GitHub repository.

# References

ADA and OKO Finance (2021). *Lessons learned: OKO Mali.* Accessed 2025-09-25. ADA. URL: https://www.ada-microfinance.org/sites/default/files/2022-09/Lessons%20learned_OKO%20Mali_EN.pdf.

Africa, ACRE (2020). *Reimagining agriculture insurance using blockchain technology.* Accessed 2025-09-25. URL: https://acreafrica.com/reimagining-agriculture-insurance-using-blockchain-technology/.

— (2024). *Building agricultural resilience among smallholder farmers through crop insurance.* Accessed 2025-09-25. URL: https://acreafrica.com/building-agricultural-resilience-among-smallholder-farmers-through-crop-insurance/.

African-Climate-Foundation (2024). *Kenya Agricultural Insurance Policy Review.* Accessed 2025-09-25. African Climate Foundation. URL: https://africanclimatefoundation.org/wp-content/uploads/2024/12/KenyaFinalVersion2DecSend.pdf.

Alsdorf, G. and J. Berkun (2024). "Is blockchain the next big thing for insurance companies?" In: *Reuters.* URL: https://www.reuters.com/legal/legalindustry/is-blockchain-next-big-thing-insurance-companies-2024-10-09/ (visited on 10/09/2024).

Artemis (2017). *Kenya Livestock Insurance Program Delivers Payouts.* Accessed 2025-09-25. URL: https://www.artemis.bm/news/world-bank-swiss-re-backed-kenya-livestock-index-insurance-pays-out/.

Ausubel, Lawrence M. and Peter Cramton (2002). "Demand reduction and inefficiency in multi-unit auctions". In: *Working Paper, University of Maryland*.

Authority, National Drought Management (2021). *Annual Report 2021*. Accessed 2025-09-25. NDMA. URL: https://ndma.go.ke/twende/?utm_source=chatgpt.com.

Bank, World (2022). *Disruptive innovations boost uptake of agriculture insurance solutions in Kenya*. URL: https://www.worldbank.org/en/news/feature/2022/06/15/disruptive-innovations-boost-uptake-of-agriculture-insurance-solutions-in-kenya.

— (2024). *World Bank open data*. Accessed 2025-09-25. URL: https://data.worldbank.org.

BASIS (2017). *Government of Kenya partners with insurers to make record insurance payouts to livestock herders facing drought*. URL: https://basis.ucdavis.edu/news/government-kenya-partners-insurers-make-record-insurance-payouts-livestock-herders-facing (visited on 02/20/2017).

*Blockchain in insurance (preprint)* (2023). DOI: 10.36227/techrxiv.24006237.

Blum, Manuel (1983). "Coin flipping by telephone". In: *Advances in Cryptology*.

Buterin, Vitalik (2015). *Commit-Reveal Schemes*. Ethereum Blog.

Chainlink (2021). *Blockchain in insurance*. URL: https://blog.chain.link/blockchain-insurance/ (visited on 12/14/2021).

— (n.d.). *AggregatorV3Interface v0.2.2 API Reference*. Accessed 2025-09-25. URL: https://docs.chain.link/chainlink-local/api-reference/v022/aggregator-v3-interface/.

Cole, Shawn, Xavier Giné, and James Vickery (2013). "Barriers to household risk management: Evidence from India". In: *Quarterly Journal of Economics* 128.2, pp. 709–764.

Cummins, J. David (2009). "Catastrophe risk modeling and catastrophe bonds". In: *Journal of Insurance Issues* 32.2, pp. 1–28.

Dominguez Anguiano, T. and L. Parte (2024). "The state of art, opportunities and challenges of blockchain in the insurance industry: A systematic literature review". In: *Management Review Quarterly* 74.2, pp. 1097–1118. DOI: 10.1007/s11301-023-00328-6.

Egberts, Alexander (2017). "The Oracle Problem in Smart Contracts". In: *Ledger* 2, pp. 1–10.

Ethereum (2025). *ERC-1155 Multi-Token Standard*. Accessed 2025-09-25. URL: https://ethereum.org/en/developers/docs/standards/tokens/erc-1155/.

Etherisc (2021). *Etherisc and ACRE Africa announce first blockchain-based payouts*. Accessed 2025-09-25. URL: https://blog.etherisc.com/etherisc-update-etherisc-and-acre-africa-announce-first-payouts-through-blockchain-based-platform-a0c5194214f4.

FAO (2024). *Kenya country profile: Hand-in-Hand initiative*. URL: https://www.fao.org/hand-in-hand/previous-editions/hih-IF-2024/kenya/en.

Government of Kenya Ministry of Agriculture, Livestock and Fisheries (2017). *Kenya climate smart agriculture strategy 2017–2026*. Accessed 2025-09-25. URL: https://www.adaptation-undp.org/sites/default/files/resources/kenya_climate_smart_agriculture_strategy.pdf.

Hanson, Robin (2003). "Combinatorial information market design". In: *Information Systems Frontiers* 5.1, pp. 107–119.

*How blockchain is revolutionizing parametric insurance* (2025). Accessed 2025-09-25. URL: `https://parametricinsured.com/how-blockchain-is-revolutionizing-parametric-insurance/` (visited on 08/08/2025).

IUCN (n.d.). *Kenya's arid and semi-arid lands (ASALs)*. Accessed 2025-09-25. URL: `https://iucn.org/our-work/topic/ecosystem-restoration/restoration-initiative/projects/kenya-asal`.

Janzen, S. et al. (2020). "Experimental games to teach farmers about weather index insurance in Kenya". In: DOI: `10.23846/TW13FE12`.

Jensen, N. D., C. B. Barrett, and A. G. Mude (2016). "Index insurance quality and basis risk: Evidence from northern Kenya". In: *American Journal of Agricultural Economics* 98.5, pp. 1450–1469. DOI: `10.1093/ajae/aaw046`.

Ministry of Agriculture (2023). *National Agricultural Insurance Policy 2024*. Accessed 2025-09-25. Government of Kenya.

OCHA (2023). *Kenya 2022 drought response in review*. URL: `https://www.unocha.org/publications/report/kenya/kenya-2022-drought-response-review` (visited on 02/15/2023).

Parkes, David C. and Sven Seuken (2014). "Economic mechanisms for online platforms". In: *ACM Conference on Economics and Computation*, pp. 679–695.

*Pula learning session: Zambia case* (2022). Accessed 2025-09-25. URL: `https://www.munichre-foundation.org/content/dam/munichre/foundation/publications/inclusive-insurance/2022-learning-sessions/20220721_LS_Zambia_Session6_Joyce_Pula.pdf`.

*Seeds of growth: Agriculture presents a range of options for insurers* (2017). Accessed 2025-09-25. URL: `https://oxfordbusinessgroup.com/reports/kenya/2017-report/economy/seeds-of-growth-agriculture-presents-a-range-of-options-for-insurers` (visited on 05/21/2017).

Shetty, A. et al. (2022). "Block chain application in insurance services: A systematic review of the evidence". In: *SAGE Open* 12.1. DOI: `10.1177/21582440221079877`.

Sibiko, K. W., P. C. Veettil, and M. Qaim (2018). "Small farmers' preferences for weather index insurance: Insights from Kenya". In: *Agriculture & Food Security* 7.1, p. 53. DOI: `10.1186/s40066-018-0200-6`.

Song, Han, Zhongche Qu, and Yihao Wei (2024). *Advancing Blockchain Scalability: An Introduction to Layer 1 and Layer 2 Solutions*. arXiv: `2406.13855` [cs.CR]. URL: `https://arxiv.org/abs/2406.13855`.

The-Star (2023). *2.6m livestock deaths reported due to prolonged drought – NDMA*. Accessed 2025-09-25. URL: `https://www.the-star.co.ke/opinion/2023-02-22-26m-livestock-deaths-reported-due-to-prolonged-drought-ndma`.

UNEP-DHI (n.d.). *Rapid integrated assessment of ASAL counties in Kenya*. Accessed 2025-09-25. URL: `https://unepdhi.org/rapid-integrated-assessment-asal-counties-kenya/`.

UNSGSA (2023). *Empowering Kenyan smallholder farmers: Pula's game-changing digital insurance*. Accessed 2025-09-25. URL: `https://www.unsgsa.org/stories/empowering-kenyan-smallholder-farmers-pulas-game-changing-digital-insurance`.

Wairimu, E., G. Obare, and M. Odendo (n.d.). *Factors affecting weather index-based crop insurance in Laikipia County, Kenya.*

Wilson, Robert (1979). *Auctions of shares.*

Wolfers, Justin and Eric Zitzewitz (2004). "Prediction markets". In: *Journal of Economic Perspectives* 18.2, pp. 107–126.

# A    Smart Contract Code

This appendix contains the full source code for the smart contracts implemented in this project. These contracts form the core of our decentralized parametric insurance system.

## A.1    RainyDayFund.sol

The main contract that implements the decentralized parametric insurance system.It handles policy issuance, claims processing, and investment management.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/ERC4626.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol";

interface AggregatorV3Interface {
  function latestRoundData() external view returns (
    uint80 roundId,
    int256 answer,
    uint256 startedAt,
    uint256 updatedAt,
    uint80 answeredInRound
  );
}

contract RainyDayFund is ERC4626, Ownable, ReentrancyGuard {
  IERC20 public immutable usdc;

  uint256 public currentSeasonId;
  uint256 public seasonOverTimeStamp;
  uint256 public constant timeUnit = 30 days;
  uint256 premium = 9 * 10**6; // 9 USDC

  AggregatorV3Interface public weatherFeed;

  // Testing variables for time control
  uint256 public testingTimeOffset;
  bool public testingMode;

  enum SeasonState { ACTIVE, INACTIVE, CLAIM, WITHDRAW, FINISHED }

  struct SeasonPolicy {
```

```solidity
37          uint256 creationTimestamp;
38          uint256 payoutAmount;
39          uint256 premium;
40          uint256 totalPoliciesSold;
41          ERC20 policyToken;
42        }
43
44      mapping(uint256 => SeasonPolicy) public seasonPolicies;
45
46      event PolicyBought(address indexed farmer, uint256 seasonId, uint256
    amount, uint256 totalPremium);
47      event ClaimMade(address indexed farmer, uint256 seasonId, uint256 amount,
     uint256 totalPayout);
48      event InvestmentMade(address indexed investor, uint256 amount);
49      event InvestmentWithdrawn(address indexed investor, uint256 amount);
50      event NewSeasonStarted(uint256 seasonId, uint256 premium, uint256
    payoutAmount);
51      event TimeAdvanced(uint256 newTimestamp, SeasonState newState);
52
53      constructor(address _usdcAddress, address _weatherOracle)
54      ERC4626(IERC20Metadata(_usdcAddress))
55      ERC20("RainyDay Investor Shares", "RDIS")
56      Ownable(msg.sender)
57      {
58        require(_usdcAddress != address(0), "USDC address zero");
59        usdc = IERC20(_usdcAddress);
60
61        require(_weatherOracle != address(0), "Weather oracle zero");
62        weatherFeed = AggregatorV3Interface(_weatherOracle);
63
64        currentSeasonId = 1;
65        _initializeSeason(currentSeasonId);
66        seasonOverTimeStamp = getCurrentTime() + 2 * timeUnit;
67
68        // Enable testing mode by default for local testing
69        testingMode = true;
70      }
71
72      // Testing function to get current time (can be offset in testing mode)
73      function getCurrentTime() public view returns (uint256) {
74        if (testingMode) {
75          return block.timestamp + testingTimeOffset;
76        }
77        return block.timestamp;
78      }
79
80      // Testing function to advance time manually
81      function advanceToNextPhase() external onlyOwner {
82        require(testingMode, "Not in testing mode");
83
84        SeasonState currentState = getSeasonState();
85        uint256 nowBlock = block.timestamp;
```

```solidity
if (currentState == SeasonState.ACTIVE) {
    uint256 target = seasonOverTimeStamp - timeUnit + 1; // +1 to ensure
inside the next phase
    testingTimeOffset = target - nowBlock;
} else if (currentState == SeasonState.INACTIVE) {
    uint256 target = seasonOverTimeStamp + 1;
    testingTimeOffset = target - nowBlock;
} else if (currentState == SeasonState.CLAIM) {
    uint256 target = seasonOverTimeStamp + timeUnit + 1;
    testingTimeOffset = target - nowBlock;
} else if (currentState == SeasonState.WITHDRAW) {
    uint256 target = seasonOverTimeStamp + 2 * timeUnit + 1;
    testingTimeOffset = target - nowBlock;
} else {
    // already finished, do nothing
    emit TimeAdvanced(getCurrentTime(), getSeasonState());
    return;
}

    emit TimeAdvanced(getCurrentTime(), getSeasonState());
}

// Testing function to set testing mode
function setTestingMode(bool _enabled) external onlyOwner {
    testingMode = _enabled;
    if (!_enabled) {
        testingTimeOffset = 0;
    }
}

function _initializeSeason(uint256 seasonId) internal {
    SeasonPolicyToken policyToken = new SeasonPolicyToken(
        string(abi.encodePacked("RainyDay Policy Season ", _toString(seasonId
))),
        string(abi.encodePacked("RDP", _toString(seasonId))),
        address(this)
    );

    seasonPolicies[seasonId] = SeasonPolicy({
        creationTimestamp: getCurrentTime(),
        payoutAmount: premium * 4,
        premium: premium,
        totalPoliciesSold: 0,
        policyToken: policyToken
    });

    emit NewSeasonStarted(seasonId, premium, seasonPolicies[seasonId].
payoutAmount);
}

function getSeasonState() public view returns (SeasonState) {
    uint256 currentTime = getCurrentTime();
```

```solidity
135          if (currentTime < seasonOverTimeStamp - timeUnit) {
136            return SeasonState.ACTIVE;
137          } else if (currentTime < seasonOverTimeStamp) {
138            return SeasonState.INACTIVE;
139          } else if (currentTime < seasonOverTimeStamp + timeUnit) {
140            return SeasonState.CLAIM;
141          } else if (currentTime < seasonOverTimeStamp + 2 * timeUnit) {
142            return SeasonState.WITHDRAW;
143          } else {
144            return SeasonState.FINISHED;
145          }
146        }
147
148        modifier onlyAfterFullSeasonCycle() {
149          require(getSeasonState() == SeasonState.FINISHED, "Season not fully
      finished yet");
150          _;
151        }
152
153        modifier onlyDuringSeason() {
154          require(
155            getSeasonState() == SeasonState.ACTIVE ||
156            getSeasonState() == SeasonState.INACTIVE,
157            "Season not active aymore");
158          _;
159        }
160
161        function startNewSeason(uint256 _premium) external onlyOwner
      onlyAfterFullSeasonCycle {
162          currentSeasonId++;
163          premium = _premium;
164          seasonOverTimeStamp = getCurrentTime() + 2 * timeUnit;
165          _initializeSeason(currentSeasonId);
166        }
167
168        function buyPolicy(uint256 _amount) external nonReentrant returns (
      uint256 seasonId) {
169          require(_amount > 0, "Amount > 0");
170          require(getSeasonState() == SeasonState.ACTIVE, "Not in active period")
      ;
171
172          SeasonPolicy storage policy = seasonPolicies[currentSeasonId];
173
174          uint256 totalPremium = policy.premium * _amount;
175          require(usdc.transferFrom(msg.sender, address(this), totalPremium), "
      Transfer failed");
176
177          SeasonPolicyToken(address(policy.policyToken)).mint(msg.sender, _amount
      );
178          policy.totalPoliciesSold += _amount;
179
180          emit PolicyBought(msg.sender, currentSeasonId, _amount, totalPremium);
```

```solidity
181            return currentSeasonId;
182        }
183
184        function claimPolicies() external nonReentrant {
185            require(getSeasonState() == SeasonState.CLAIM, "Not in claim period");
186
187            SeasonPolicy storage policy = seasonPolicies[currentSeasonId];
188            SeasonPolicyToken token = SeasonPolicyToken(address(policy.policyToken)
    );
189            uint256 amount = token.balanceOf(msg.sender);
190            require(amount > 0, "No policies to claim");
191
192            (,int256 weather,) = getWeatherData();
193            require(uint256(weather) < 10, "Weather not bad enough");
194
195            uint256 totalPayout = policy.payoutAmount * amount;
196            require(usdc.balanceOf(address(this)) >= totalPayout, "Insufficient
    funds");
197
198            require(usdc.transfer(msg.sender, totalPayout), "Payout failed");
199            token.burnFrom(msg.sender, amount);
200
201            emit ClaimMade(msg.sender, currentSeasonId, amount, totalPayout);
202        }
203
204        function getWeatherData() public view returns (uint80 roundId, int256
    weather, uint256 timestamp) {
205            (roundId, weather,,timestamp,) = weatherFeed.latestRoundData();
206        }
207
208        // ERC4626 investment logic
209        function invest(uint256 assets) external nonReentrant onlyDuringSeason {
210            require(assets > 0, "Amount > 0");
211            deposit(assets, msg.sender);
212            emit InvestmentMade(msg.sender, assets);
213        }
214
215        function redeemShares(uint256 shares) external nonReentrant {
216            require(getSeasonState() == SeasonState.WITHDRAW, "Not in withdrawal
    period");
217            uint256 assets = redeem(shares, msg.sender, msg.sender);
218            emit InvestmentWithdrawn(msg.sender, assets);
219        }
220
221        function totalAssets() public view override returns (uint256) {
222            return usdc.balanceOf(address(this));
223        }
224
225        // minimal utility
226        function _toString(uint256 value) internal pure returns (string memory) {
227            if (value == 0) return "0";
228            uint256 temp = value;
```

31

```
229        uint256 digits;
230        while (temp != 0) { digits++; temp /= 10; }
231        bytes memory buffer = new bytes(digits);
232        while (value != 0) { digits--; buffer[digits] = bytes1(uint8(48 + value
     % 10)); value /= 10; }
233        return string(buffer);
234      }
235    }
236
237    contract SeasonPolicyToken is ERC20 {
238      address public immutable rainyDayFund;
239      modifier onlyRainyDayFund() { require(msg.sender == rainyDayFund, "Only
     fund"); _; }
240
241      constructor(string memory name, string memory symbol, address _fund)
     ERC20(name, symbol) {
242        rainyDayFund = _fund;
243      }
244
245      function mint(address to, uint256 amount) external onlyRainyDayFund {
     _mint(to, amount); }
246      function burnFrom(address from, uint256 amount) external onlyRainyDayFund
      { _burn(from, amount); }
247    }
248
```

Listing 4: RainyDayFund.sol - Main Insurance Contract

## A.2 MockUSDC.sol

A mock implementation of the USDC stablecoin for testing purposes. It implements the ERC20 interface and provides additional minting functionality.

```
1     // SPDX-License-Identifier: MIT
2     pragma solidity ^0.8.20;
3
4     import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
5
6     contract MockUSDC is ERC20 {
7         constructor() ERC20("Mock USDC", "mUSDC") {
8             _mint(msg.sender, 1_000_000 * 10 ** decimals());
9         }
10
11        function mint(address to, uint256 amount) external {
12            _mint(to, amount);
13        }
14    }
15
```

Listing 5: MockUSDC.sol - Mock Stablecoin for Testing

## A.3 MockWeatherOracle.sol

A mock implementation of a weather data oracle that conforms to Chainlink's AggregatorV3Interface for standardized data feeds. This contract simulates weather data for testing the insurance system.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

contract MockWeatherOracle {
    int256 public latestWeather;
    uint80 public latestRound;

    constructor(int256 _initialWeather) {
        latestWeather = _initialWeather;
        latestRound = 1;
    }

    function updatePrice(int256 _newWeather) external {
        latestWeather = _newWeather;
        latestRound += 1;
    }

    function latestRoundData()
        external
        view
        returns (
            uint80 roundId,
            int256 answer,
            uint256 startedAt,
            uint256 updatedAt,
            uint80 answeredInRound
        )
    {
        return (
            latestRound,
            latestWeather,
            block.timestamp,
            block.timestamp,
            latestRound
        );
    }
}
```

Listing 6: MockWeatherOracle.sol - Mock Weather Data Oracle

# B Contract Architecture Analysis

The smart contracts presented above form the foundation of our decentralized parametric insurance system. Each plays a specific role in the ecosystem:

## B.1 RainyDayFund

This is the core contract of the system and implements several key components:

1. **Policy Management**: Handles the issuance of policy tokens to farmers and tracks active policies.

2. **Season States**: Implements a state machine for managing different phases of the insurance season (active, inactive, claim, withdraw, finished).

3. **Claims Processing**: Processes claims by verifying policy ownership and weather conditions.

4. **Investment Management**: Allows investors to provide liquidity to the risk pool and earn returns.

5. **Weather Data Integration**: Interfaces with the weather oracle to retrieve data for claims verification.

The contract employs several important design patterns:

- **ERC4626 Tokenized Vault**: For managing investments and yield generation.

- **ERC20 Policy Tokens**: For representing insurance policies as transferable assets.

- **Non-reentrant Guards**: To prevent reentrancy attacks during financial transactions.

- **Owner Controls**: To manage system parameters and seasonal transitions.

- **State Machine**: To enforce the correct sequence of operations based on the current season state.

## B.2 MockUSDC

This contract simulates a stablecoin for testing purposes. In a production environment, this would be replaced with a real stablecoin like USDC. The mock implementation provides:

- Standard ERC20 functionality for token transfers and allowances

- Additional minting capability for testing purposes

- Initial supply of 1,000,000 tokens for the contract deployer

## B.3 MockWeatherOracle

This contract simulates a weather data oracle, conforming to Chainlink's AggregatorV3Interface standard. In production, this would be replaced with actual Chainlink oracles providing real weather data. The mock implementation provides:

- Storage for weather data values

- Functions to update weather data (for testing purposes)

- Standard interface conformance to ensure compatibility with the main contract

# C   AI Usage Overview

Table 3: AI Usage Table

| AI-based Tool | Type of Use | Affected Parts of the Work |
|---|---|---|
| **ChatGPT** | Proof-Reading, fixing typos and improving sentence structure as well as fixing LaTeX compilation errors | ResearchPaper.pdf |
| **ClaudeAI** | Finding and Fixing coding errors | Testing suite, App.jsx |
| **Grok/xAI** | Finding and Fixing coding errors, suggestions for tech-stack | Solidity Contracts |