ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

Máster en Ingeniería Industrial

Blockchain and IoT for Secure and Automated Smart Irrigation Systems

Author
Jaime Huarte Rubio

Directed by
Jafar Saniee
Mikhail Gromov

Madrid
August 2025

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título Blockchain and IoT for Secure and Automated Smart Irrigation Systems en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2024-25 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: Jaime Huarte Rubio Fecha: 25/08/2025

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Jafar Saniee Fecha: 25/08/2025

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA (ICAI)

Máster en Ingeniería Industrial

Blockchain and IoT for Secure and Automated Smart Irrigation Systems

Author
Jaime Huarte Rubio

Directed by
Jafar Saniee
Mikhail Gromov

Madrid
August 2025

BLOCKCHAIN AND IOT FOR SECURE AND AUTOMATED SMART IRRIGATION SYSTEMS

Autor: Jaime Huarte Rubio
Supervisores: Jafar Saniee & Mikhail Gromov
Entidad Colaboradora: ICAI – Universidad Pontificia Comillas

RESUMEN
Este proyecto explora la integración de dispositivos IoT, tecnología blockchain y Smart Contracts para crear un sistema de riego agrícola seguro, transparente y automatizado. Los nodos estan basados en sensonser con microcontroladores ESP32 que transmiten datos ambientales a un hub Raspberry Pi 4 mediante comunicación LoRa. La pasarela actúa como un oracle, validando la integridad de los paquetes (CRC-32) y registrando actualizaciones en la blockchain con librerías Web3. Los Smart Contracts desplegados en la red Polygon PoS (testnet Amoy) implementan la lógica de riego, aplican umbrales predefinidos y proporcionan un registro de eventos, decisiones y estados del sistema que son verificables.

El prototipo adopta una arquitectura híbrida *edge–blockchain* en la que la pasarela Raspberry Pi funciona tanto como procesador local como oracle. Aunque el uso de redes de oráculos descentralizadas como Provable o Chainlink sigue siendo una dirección prometedora para el futuro, el enfoque implementado ofrece un concepto práctico y reproducible.

El sistema demuestra cómo la automatización basada en blockchain puede mejorar la trazabilidad y la reproducibilidad en la gestión del agua agrícola. Asimismo, muestra que, cuando las interacciones son poco frecuentes y de tipo orientado a eventos, los costos operativos pueden mantenerse en niveles bajos en comparación con las plataformas IoT de servicios en la nube. Este proyecto sienta las bases para futuras evaluaciones de latencia, eficiencia energética y despliegue a gran escala en entornos agrícolas reales.

Palabras clave: Riego inteligente, Internet de las Cosas (IoT), Blockchain, Smart Contracts, Arquitectura híbrida, Tecnología agrícola, Comunicación LoRa, Raspberry Pi, ESP32, Diseño de oráculos, Automatización descentralizada, Blockchain Polygon, Gestión sostenible del agua, Agricultura de precisión, Edge Computing, Integridad de datos, Toma de decisiones transparente.

1. Introducción

La escasez de agua y la seguridad alimentaria se encuentran entre los desafíos globales más urgentes del siglo XXI. La agricultura, al ser el mayor consumidor de agua dulce, requiere con urgencia innovaciones tecnológicas para optimizar el uso de los recursos. Este proyecto responde a esa necesidad mediante la propuesta de un sistema de riego inteligente que integra sensores IoT y Smart Contracts en blockchain. El objetivo es lograr una solución segura, automatizada y escalable para el control del riego basada en datos ambientales en tiempo real.

2. Definición del Proyecto

El sistema está diseñado para monitorizar la humedad del suelo, la temperatura y la

humedad ambiental mediante nodos ESP32 distribuidos que utilizan comunicación LoRa. Una Raspberry Pi funciona como pasarela para preprocesar localmente los datos antes de transmitirlos a la blockchain. La blockchain, específicamente la testnet de Polygon, aloja los contratos inteligentes que ejecutan las órdenes de riego en función de las lecturas de los sensores. El sistema sigue una *arquitectura híbrida edge–blockchain*: mientras que las decisiones y registros son inmutables en la cadena, la Raspberry Pi actúa como oráculo que conecta la sensorización fuera de la cadena con la lógica en cadena. La motivación radica en crear un marco de automatización de riego auditable, con mínima confianza y bajo consumo energético.

3. Descripción del Modelo/Sistema/Herramienta

La arquitectura se divide en tres capas principales: sensorización física, procesamiento y toma de decisiones. Los datos ambientales son captados mediante sensores y enviados por LoRa a la Raspberry Pi, que filtra el ruido e identifica patrones relevantes. Los contratos inteligentes escritos en Solidity procesan esta información para decidir los eventos de riego. En este prototipo, la propia pasarela actúa como oráculo que reenvía actualizaciones firmadas de los sensores a la blockchain. La integración de oráculos descentralizados como Provable o Chainlink se plantea como una extensión natural para trabajos futuros. Los componentes fueron probados en un entorno simulado para poder validar la funcionalidad del sistema, su rendimiento y sus costes.

4. Resultados

El sistema se comprobó en un escenario agrícola simulado. Las funcionalidades del sistema se validaron mediante registros, inspección de los eventos de la blockchain y las respuestas de actuadores simulados. El prototipo demostró que el diseño híbrido reduce interacciones innecesarias con la blockchain gracias al filtrado local de datos, y que la lógica del contrato aplica de manera fiable los umbrales de riego. Aunque métricas cuantitativas como latencia, consumo energético y coste de transacción deberán medirse en pruebas físicas, la simulación confirmó la viabilidad y rentabilidad del enfoque, con costes de interacción en blockchain prácticamente insignificantes en la escala analizada.

5. Conclusion

El sistema propuesto demuestra la viabilidad de combinar tecnologías blockchain e IoT para la automatización agrícola segura. El prototipo ofrece una alternativa transparente y escalable a los sistemas de riego tradicionales, particularmente adecuada para entornos agrícolas descentralizados o cooperativos. Aunque la versión actual se basa en pruebas simuladas, el diseño está técnicamente preparado para su despliegue físico con mínimas adaptaciones. El proyecto contribuye a los Objetivos de Desarrollo Sostenible relacionados con la gestión del agua, la innovación y el uso responsable de los recursos.

BLOCKCHAIN AND IOT FOR SECURE AND AUTOMATED SMART IRRIGATION SYSTEMS

Author: Jaime Huarte Rubio
Supervisors: Jafar Saniee & Mikhail Gromov
Collaborating Entity: ICAI – Universidad Pontificia Comillas

ABSTRACT

This project explores the integration of IoT devices, blockchain technology and smart contracts to create a secure, transparent and automated irrigation system for agriculture. Sensor nodes based on ESP32 microcontrollers transmit environmental data to a Raspberry Pi 4 gateway via LoRa. The gateway acts as a trusted oracle, validating packet integrity (CRC-32) and committing updates to the blockchain using Web3 libraries. Smart contracts deployed on the Polygon PoS (Amoy testnet) implement irrigation logic, enforce predefined thresholds and provide a verifiable event log of decisions and system states.

The prototype adopts a hybrid edge–blockchain architecture in which the Raspberry Pi gateway serves as both local processor and oracle. While the use of decentralized oracle networks such as Provable or Chainlink remains a promising direction for future development, the implemented approach provides a practical and reproducible proof of concept.

The system demonstrates how blockchain-based automation can enhance accountability and reproducibility in agricultural water management. It also shows that, when interactions are sparse and event-driven, operational costs can remain negligible compared to conventional cloud IoT platforms. This work lays the foundation for subsequent evaluations of latency, energy efficiency and large-scale deployment in real agricultural environments.

Key words: Smart irrigation, Internet of Things (IoT), Blockchain, Smart contracts, Hybrid architecture, Agricultural technology, LoRa communication, Raspberry Pi, ESP32, Oracle design, Decentralized automation, Polygon blockchain, Sustainable water management, Precision agriculture, Edge computing, Data integrity, Transparent decision-making.

1. Introduction

Water scarcity and food security are among the most pressing global challenges of the 21st century. Agriculture, being the largest consumer of freshwater, urgently requires technological innovation to optimize resource use. This project addresses that need by proposing a smart irrigation system that integrates Internet of Things (IoT) sensors and blockchain smart contracts. The aim is to achieve a secure, automated and scalable solution for irrigation control based on real-time environmental data.

2. Project Definition

The system is designed to monitor soil moisture, temperature and humidity via distributed ESP32 nodes using LoRa communication. A Raspberry Pi functions as a gateway to locally pre-process data before transmitting it to the blockchain. The blockchain, specifically

the Polygon testnet, hosts smart contracts that execute irrigation commands based on sensor readings. The system follows a *hybrid edge–blockchain architecture*: while decisions and logs are immutable on-chain, the Raspberry Pi acts as a trusted oracle bridging off-chain sensing with on-chain logic. The motivation lies in creating a trust-minimized, auditable and energy-aware irrigation automation framework.

3. Description of the Model/System/Tool

The architecture is divided into three main layers: sensing, processing and decision-making. Environmental data is captured through calibrated sensors and sent via LoRa to the Raspberry Pi, which filters noise and identifies actionable patterns. Smart contracts written in Solidity process this input to decide irrigation events. In this prototype, the gateway itself serves as the oracle that forwards signed sensor updates to the blockchain. Integration of decentralized oracles such as Provable or Chainlink is identified as a natural extension for future work. All components were tested in a simulated environment to validate system functionality, performance and cost considerations.

4. Results

The system was successfully deployed in a simulated agricultural scenario. Functionality was validated through log traces, on-chain event inspection and simulated actuator responses. The prototype showed that the hybrid design reduces unnecessary blockchain interactions by filtering data locally and that the contract logic reliably enforces irrigation thresholds. Although quantitative metrics such as latency, power consumption and transaction cost remain to be measured under physical testing, the simulation confirmed the feasibility and cost-effectiveness of the approach, with blockchain interactions incurring only negligible costs at the tested scale.

5. Conclusions

The proposed system demonstrates the viability of combining blockchain and IoT technologies for secure agricultural automation. It offers a transparent and scalable alternative to traditional irrigation systems, particularly suitable for decentralized or cooperative farming environments. While the current version relies on simulated testing, the design is technically ready for physical deployment with minimal adjustments. The project contributes to Sustainable Development Goals related to water management, innovation and responsible resource use.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

It is estimated that by 2050, the world population will exceed 9.7 billion people, which will require an increase in food production of between 50% and 70% [1]. This challenge must be addressed in a context of growing constraints, marked by climate change, freshwater scarcity and soil degradation. In many areas, agriculture already competes with the urban and industrial sectors for access to water. According to the FAO, this sector currently accounts for more than 70% of global freshwater withdrawals and a significant portion of this resource is lost due to inefficiencies in irrigation systems [1].

Adding to these tensions is the increasing climate variability, which has disrupted traditional agricultural cycles. Regions that previously had regular rainfall patterns now suffer from droughts, while others face extreme and unpredictable weather events. These circumstances require moving away from generic irrigation models based on fixed schedules toward dynamic, data-dependent systems [2]. In this context, technological innovation is no longer merely a competitive advantage but has become an indispensable requirement for agricultural resilience.

1.2 Context and relevance

The so-called smart agriculture, based on the integration of sensors, data networks and automation processes, is emerging as a key strategy for achieving more sustainable, efficient and flexible production. The Internet of Things (IoT) opens up the possibility of collecting real-time information on soil moisture, temperature, solar radiation and environmental humidity conditions. These indicators allow decisions related to irrigation, fertilization and crop management to be guided with unprecedented accuracy.

Despite its potential, many smart agriculture platforms, both commercial and research-based, rely on centralized infrastructure and proprietary cloud services. This dependence introduces significant limitations in terms of data privacy, interoperability and resilience to failure and in many cases also entails high operating costs. In this scenario, blockchain technology is emerging as a viable alternative by providing trust, decentralization and automation.

A blockchain-based system allows irrigation rules to be transferred to smart contracts that are executed without the need for centralized control, thus ensuring transparent,

secure and tamper-resistant processes. In combination with IoT, a paradigm shift is taking place: moving from the mere collection of data for subsequent analysis to the autonomous execution of actions based on real-time information, even on large and difficult to access farms [3, 4].

1.3 Social, economic and technological importance

From a social perspective, water scarcity impacts not only agricultural productivity, but also factors closely linked to the well-being of the population, such as food prices, rural community incomes and public health. In this sense, an efficient irrigation system is a key element in strengthening food security, particularly in regions where access to water is uncertain or conditioned by political tensions.

Economically, the automation of irrigation processes reduces dependence on labor, optimizes water consumption and decreases losses from crop failures. These aspects translate into greater profitability and sustainability for both small producers and large-scale farms. In addition, the use of smart contracts helps reduce administrative costs, facilitating the collective management of irrigation infrastructure or coordination in agricultural cooperatives.

In the technological domain, the project highlights the potential of combining open-source microcontrollers (such as ESP32), edge computing (Raspberry Pi) and decentralized infrastructures (blockchain and oracles). This integration allows for the configuration of a modular, scalable and low-cost system with both practical and educational applications. The proposal also exemplifies the results of combining embedded systems, cybersecurity and distributed computing models, leading to new and better possibilities for innovation in water resource management.

1.4 Problem statement

Despite the growing availability of sensors and automation platforms, most current solutions for irrigation remain constrained. They are often expensive, rely heavily on centralized cloud architectures and lack transparency in their decision-making. In many developing countries, manual irrigation is still predominant, partly due to limited trust in automated systems, interoperability issues and the absence of verifiable records documenting when, why and how water was applied.

This reveals a clear technological and adoption gap where there is a need for an irrigation system that is affordable, transparent and decentralized, capable of integrating wireless sensor networks with local computing to reduce costs, while relying on blockchain to ensure trustworthy and tamper-resistant decision-making. Very few implementations to date have combined low-power, long-range communication such as LoRa with blockchain-based smart contracts to achieve such a system.

The central research question guiding this thesis is therefore: *Can blockchain-enabled IoT architectures provide a cost-efficient, transparent and scalable alternative to conventional cloud-based irrigation systems, while remaining suitable for real agricultural environments?*

1.5 Scope and Objectives

The scope of this project is to design, implement and evaluate a prototype of a blockchain-enabled smart irrigation system that bridges low-power IoT sensing with decentralized automation. The system brings together ESP32-based sensor nodes equipped with soil moisture, temperature, humidity and light sensor. Also implement a LoRa-based network for long-range and energy-efficient communication with a Raspberry Pi serving as gateway for preprocessing data and connecting to the oracle. Smart contracts needs to be deployed on the Polygon Amoy testnet and will be responsible for executing irrigation logic, storing decisions and providing transparent records of operation.

The specific objectives include:

- Design a modular and scalable architecture that integrates IoT nodes with blockchain smart contracts.
- Implement and test local pre-processing techniques that reduce redundant blockchain interactions.
- Validate the end-to-end operation of the system through simulation, including cost estimates and performance analysis.
- Evaluate the feasibility of using blockchain as a cost-effective alternative to conventional cloud-based IoT platforms for irrigation control.

1.6 Project Methodology

The methodology adopted in this project follows a modular and iterative development process, which includes:

1. **System Design:** Defining the system architecture, selecting hardware components (sensors, microcontrollers) and setting communication protocols.
2. **Prototype Development:** Building and programming ESP32 nodes, configuring LoRa communication and deploying smart contracts.
3. **Simulation Testing:** Running simulations with synthetic sensor data under controlled conditions to replicate real-world variability.
4. **Blockchain Integration:** Using testnet environments and oracles to validate data flow from sensor to smart contract.

5. **Evaluation:** Measuring latency, scalability, energy usage and transaction cost to assess system feasibility.

This methodology ensures a balance between academic rigor and practical implementation, enabling future upgrades or real-field deployment.

1.7 Alignment with Sustainable Development Goals (SDGs)

This project contributes directly to the following United Nations Sustainable Development Goals:

- **SDG 6 – Clean Water and Sanitation:** The system optimizes water usage, minimizing waste through precision irrigation and data-driven control.
- **SDG 9 – Industry, Innovation and Infrastructure:** By integrating emerging technologies like blockchain and IoT, the project promotes innovation in agricultural infrastructure.
- **SDG 12 – Responsible Consumption and Production:** The decentralized and transparent nature of the system enhances traceability, reduces input waste and supports efficient resource use.

These contributions underscore the system's relevance not just as a technological prototype, but as a meaningful step toward *resilient and sustainable agricultural practices*.

2 Technology Overview

In this chapter, it is explained the core technologies and protocols that are used throughout the development of the system. Each section clarifies the specific role of these components and provides technical context for their selection and application in the project.

2.1 Internet of Things (IoT)

The Internet of Things (IoT) facilitates inter-connectivity between physical devices and digital platforms using embedded sensors, microcontrollers and network interfaces. The combination of these technologies enables real-time environmental data acquisition and process automation, especially useful for achieving irrigation precision in agriculture. As highlighted in Gubbi et al. [5], IoT structures are typically layered and composed of perception, network and application layers. In this project, these elements are achieved by using a set of ESP32-based nodes forming the network.

The ESP32 is a low-power system-on-chip (SoC) microcontroller with integrated Wi-Fi and Bluetooth capabilities, although in this implementation it is used primarily for LoRa communication due to energy efficiency and extended range. Each ESP32 module has a capacitive soil moisture sensor and a DHT22 digital temperature and humidity sensors. These environmental sensors represent the foundation of the perception layer, converting physical conditions into digital signals. The capacitive sensor was selected over a resistive sensors because of long-term reliability and corrosion resistance in agricultural deployments.

The data collected from the sensor nodes is transmitted wirelessly using LoRa (Long Range). It is a modulation protocol discussed in Section 2.2, transmitting to a centralized processing node. This communication model enables deployment over wide spatial range, such as crop fields or vineyards, with minimal infrastructure. Sensor readings serve as the input for the automation logic, forming the foundation for the the blockchain layer to then operate.

The architecture of this layered IoT system includes sensor nodes, gateway and integration with blockchain logic, this is illustrated in Figure 2.1. This schematic clarifies how each subsystem interacts within the overall automation framework, making the information flow from the physical world to the on-chain decision-making explicit and traceable.

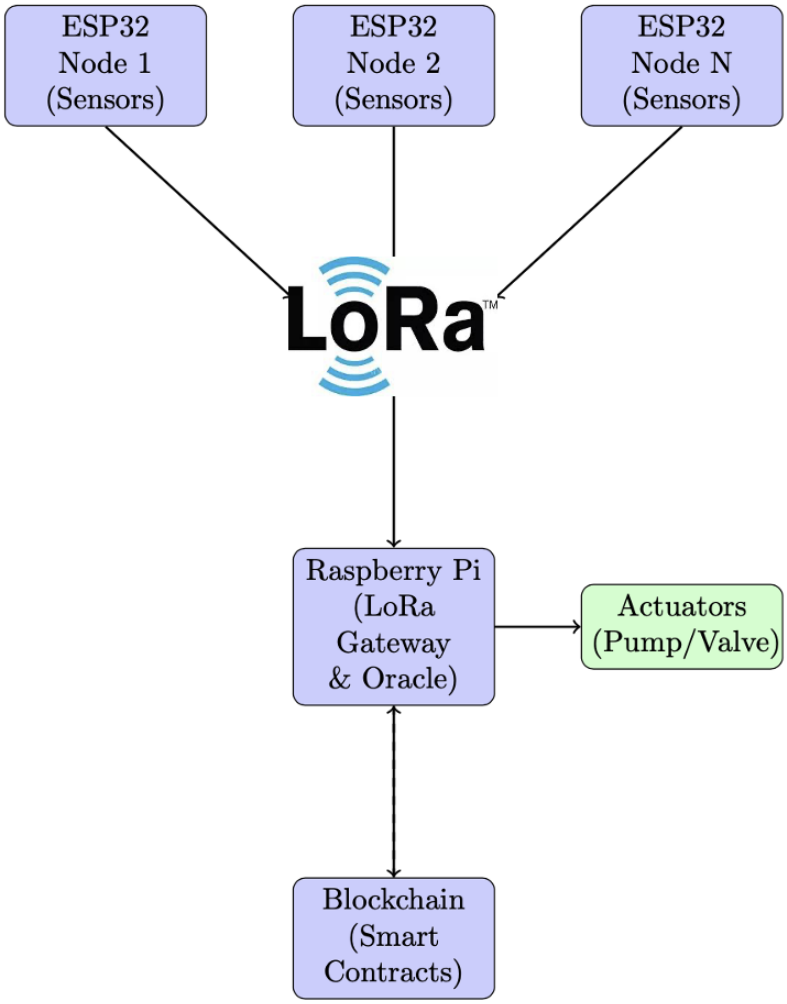


Figure 2.1: Layered architecture of the IoT system with communication and decision components.

2.2 LoRa and LoRaWAN Communication Protocol

The LoRa technology is a physical layer protocol based on chirp spread spectrum modulation, which is a data encoding method where the signals are sent through frequency gradually being increased or decreased over time. LoRa is known for its low-power operation and long-range communication capabilities, as it often reaches from 2 to 15 km in open environments. It is particularly suitable for non-time-critical applications such as environmental monitoring, where data packets are small and periodic. Additionaly, it provides a lot of benefits as he LoRa Alliance has standardized its use through the LoRaWAN MAC-layer protocol, which supports secure addressing, adaptive data rates and bidirectional communication [6].

In this project, LoRa modules (SX1276) are added to the ESP32 units, each transmitting data to a LoRa receiver connected to the Raspberry Pi gateway. LoRaWAN itself is not used directly; instead the communication was implemented with a simplified point-to-gateway configuration. This is done to reduce overhead and system complexity,

as our type of message packets do not require. Communication is uni-directional, from the sensor to the gateway and predefined intervals are implemented (every 5 minutes) to transmit the values of temperature, humidity and soil moisture.

LoRa's reliability in agricultural settings is beneficial by its resistance to interference and its propagation characteristics in open fields. Moreover, its operation in the ISM (Industrial, Scientific and Medical) band minimizes regulatory constraints across regions. Comparing to other wireless protocols such as Wi-Fi, ZigBee and NB-IoT, LoRa achieves gives a better balance of range, power efficiency and deployment cost. This makes it particularly well-suited for smart irrigation systems. A comparative overview of these protocols is provided in Table 2.1.

Protocol	**Range**	**Power Use**	**Data Rate**	**Topology**	**Cost**	**Suitability**
Wi-Fi	50–100 m	High	High	Star	Low–Med.	Indoor/Short Range
Bluetooth	10–30 m	Low	Medium	Star	Low	Personal Devices
ZigBee	10–100 m	Low	Medium	Mesh	Low	Home/Industrial
LoRa	2–15 km	Very Low	Low	Star	Medium	Agriculture, Remote
NB-IoT	1–10 km	Low	Medium	Star	Med.–High	Industrial IoT

Table 2.1: Comparison of wireless communication protocols for agricultural IoT deployments.

2.3 Raspberry Pi as Local Gateway

The Raspberry Pi is the data aggregation and pre-processing hub. It is equipped with a LoRa receiver (via SPI interface) and acts as the bridge between sensor data and the blockchain layer. This hub has several functions, including:

- Collecting and parsing data packets that are transmitted via LoRa.
- Filtering and validating the incoming sensor readings, by also removing corrupted packets or duplicates.
- Apply conditional logic to determine whether the current readings meet predetermined thresholds that will then require to post to the smart contract evaluation.
- Signing and submitting transactions directly to the blockchain via Web3, allowing to effectively behave as the oracle (for prototype porpuses).

The device also operates continuously and stores the sensor logs locally, which allows for backup and troubleshooting. Time synchronization is achieved using NTP to ensure timestamp accuracy, essential for logging events on the blockchain.

2.4 Smart Contracts and the Polygon Blockchain

Smart contracts is one of the key implementations in this project, as they represent an innovative technology. They are self-executing code that are stored and run on blockchain platforms.

A blockchain platform is a decentralized digital system that functions as a shared, immutable ledger for recording and verifying transactions across a network of computers, or nodes. This technology uses a distributed database structure, rather than a central one, to store information in linked blocks. This architecture provides enhanced security, transparency and efficiency by creating a tamper-proof record of transactions that everyone on the network can see.

A blockchain platform is a decentralized digital system that functions as a shared, immutable ledger for recording and verifying transactions across a network of computers (nodes). This technology does not rely on a centralized database, as it employs a distributed data structure in which information is stored in cryptographically linked blocks. This architecture enhances security, transparency and efficiency; as it creates a tamper-proof record of all transactions (decisions) that all participants in the network have accessed and done [7].

Once a smart contract is deployed in the blockchain network, their logic cannot be modified, this ensures determinism and verifiability of operations. These contracts can include several either business rules, sensor thresholds and automation policies in a decentralized and immutable form. In this project, smart contracts are used to evaluate incoming environmental data and autonomously determine whether irrigation should be triggered or not.

The contracts are written in Solidity, which is Ethereum's domain-specific language and they are deployed on the Polygon Amoy testnet. Polygon is a Layer-2 scaling solution for Ethereum that offers fast block confirmation times and significantly lower transaction fees, very beneficial for prototyping purposes. This still maintains full compatibility with the Ethereum Virtual Machine (EVM), for large-scale deployment. This makes it ideal for agricultural IoT applications that require frequent, lightweight interactions with the blockchain [7, 8].

Table 2.2 compares Polygon to other prominent blockchain platforms that are commonly used for smart contracts. The selection of Polygon for this application was based on key considerations for our application, these include transaction cost, speed, compatibility with IoT constraints and developer ecosystem support.

Platform	Avg. TX Fee (USD)	TX Time (s)	EVM Compatible	IoT Suitability
Ethereum Mainnet	>1.00	~15	Yes	Low
Polygon (Amoy)	<0.01	~2	Yes	High
Binance Smart Chain	<0.10	~5	Yes	Medium
Solana	<0.01	<1	No	Medium
Algorand	<0.01	<5	No	Medium

Table 2.2: Comparison of smart contract platforms relevant to agricultural IoT applications [9].

Each smart contract in this system includes functions that receive data from the gateway-oracle (the Raspberry Pi in this prototype) and compare it to predefined soil moisture thresholds. Once the conditions are met for irrigation, the contract emits the correspondent events. These events can later be used to activate solenoid valves via relay switches or notify users through a mobile interface. Importantly, all decisions are recorded immutably on-chain, providing historical system behavior data and preventing tampering with the data and decision making itself.

A key advantage of using smart contracts in this context is the elimination of centralized intermediaries. Usually, traditional irrigation systems rely on remote servers or human controllers, these can introduce latency, bias or single points of failure. In comparison, the blockchain-based logic ensures that allowed users can verify the system behavior in a transparent way. This is particularly beneficial in cooperatives or public-sector managed farms where traceability and accountability are essential for their stakeholders [10].

The full interaction starts the workflow from sensing, to gateway pre-processing, to blockchain logic, this is represented in the Figure 2.2. This visual helps clarify how smart contracts are triggered based on the field data and how each system layer contributes to the final goal of a decentralized irrigation automation system.

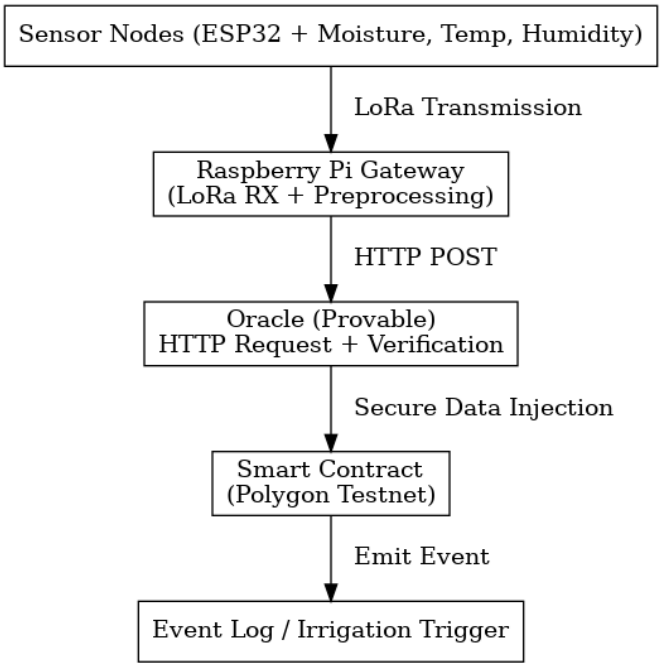


Figure 2.2: Data flow from IoT sensors to smart contract-based irrigation decision-making.

2.5 Oracles and Off-Chain Data Ingestion

Blockchains are isolated from external data for security and consistency, which causes the network to not be able to directly read from APIs, sensors, or web services. To resolve this, oracles serve as trusted bridges between the on-chain and off-chain worlds. During the design phase, oracle services such as Provable (formerly Oraclize) were considered, as they provide cryptographic attestations that can be verified by smart contracts [11], adding an extra security layer to the entire system and communication.

However, in the prototype implementation, the Raspberry Pi itself performs the oracle function: as it collects the sensor data, signs it locally and submits the transactions directly to the Polygon Amoy testnet via Web3. This approach of using the gateway as the oracle simplifies deployment and removes reliance on third-party services for now, though it introduces a trust assumption in the gateway device security.

Future work should integrate decentralized oracle networks (e.g., Provable, Chainlink) to improve robustness, redundancy and tamper-evidence. These services are key to provide verifiable proofs of data authenticity, thus strengthening trust in agricultural automation systems.

2.6 Security and Fault Tolerance Mechanisms

Reliability and resistance in the system is very important for both environmental disturbances and cyber-threats, in order to assure a trusted smart agriculture deployment. Some mechanisms can be incorporated into the system to obtain data integrity, tamper

resistance and be able to recover from communication failures.

The recording of irrigation decisions and sensor readings on-chain will allow the system to benefit from the blockchain's inherent immutability. All records are time-stamped and cannot be retroactively altered, which ensures transparent auditing and accountability for check ups in the future. This is would be relevant for cooperative and regulated farming operations, as they all would have a transparent access to it.

Sensor readings are transmitted to the Raspberry Pi gateway, which acts as the oracle. The RPi cryptographically signs and forwards data directly to the smart contract via Web3 library. In a future version, this role could be delegated to a third-party oracle service, which would provide additional verification proofs.

All sensor data could be cached temporarily in the Raspberry Pi itself and backed up periodically. In the event of connectivity loss with the blockchain or oracle service, data would be queued and retransmitted back to the blockchain once connection is restored. This local redundancy would ensure fault tolerance and minimizes the loss of data.

Each ESP32 node includes access control features at the firmware level to prevent unauthorized configuration. Flash encryption and secure boot can be enabled, this would ensure that the firmware has not been tampered with prior to operation.

The following table summarizes the fault tolerance mechanisms provided across system components:

Component	**Fault Tolerance Mechanism**
ESP32 Node	Watchdog timer reset, access control, encrypted firmware
LoRa Transmission	Periodic resend attempts, CRC error checking
Raspberry Pi	Local data caching, NTP time sync, log archiving
Gateway-Oracle	Retry logic, Web3 secure transmission
Smart Contract	Immutable logging, replay protection

Table 2.3: Overview of fault tolerance features.

2.7 System Modularity and Interoperability

One of the principal strengths of the proposed architecture is the modularity and interoperability it offers. As each part of the system has been designed separately: nodes, communication interfaces, processing gateways and blockchain contracts. They have been designed as independent modules with defined inputs and outputs, allowing a seamlessly integration and a modular design that simplifies maintenance, testing and future scalability.

Hardware interoperability is done by integrating standard communication protocols between the sensors and ESP32 boards, between nodes and gateway with the utilization of LoRa. On the software side, data structures are defined in JSON format to ensure compatibility between the Raspberry Pi and the potential external oracle services, plus the smart contracts. This separation allows individual components to be replaced or upgraded without the need of redoing the entire system.

To further enhance interoperability, open standards and libraries were used. Some of them are the LoRa SX127x module, the Arduino IDE with PlatformIO and Python scripts on the Raspberry Pi. These ensure that the system can be replicated across different hardware platforms with minimal effort, as they are commonly known and well documented in its use.

3 State of the Art

This chapter reviews the existing literature and applications related to smart irrigation, blockchain use in agriculture and the key challenges that this project aims to address. The goal is to situate the proposed solution within the technological and academic landscape, identifying both validated methodologies and unexplored opportunities.

3.1 Existing Smart Irrigation Systems

In the past decade, smart irrigation systems have evolved significantly as advancements have been done in wireless sensor networks, embedded systems and remote communication technologies. Many commercial systems such mostly rely on centralized cloud platforms to aggregate sensor data, process the weather forecasts and automate the irrigation decisions, some known platforms are RainMachine, Rachio and Netro. These systems typically use Wi-Fi or GSM-based modules and include mobile applications for user interaction. Academic research prototypes have also explored similar functionalities using microcontrollers such as Arduino, Raspberry Pi or ESP8266, along with GSM or ZigBee modules for managing the remote communication [12, 13, 14].

These systems generally consist of three layers: the perception layer (including environmental sensors like soil moisture, temperature and humidity), the network layer (responsible for the data transmission) and the application layer (in charge of the decision-making and user interfacing). In many cases the application logic is in centralized cloud platforms such as AWS IoT, Google Firebase or Microsoft Azure. While these solutions offer scalability and ease of deployment, they come at the cost of dependency on internet availability and one of the biggest concerns is the data ownership and long-term accessibility.

Moreover, data security and privacy still are significant challenges. Most commercial platforms do not offer end-to-end encryption and the sensor data is often processed in a cloud infrastructures with limited transparency about data usage or storage policies. Academic systems sometimes address this by proposing open-source implementations, but these often lack the robustness and integration maturity found in their commercial counterparts.

Irrigation control strategies that are employed by existing systems include threshold-based triggering, fuzzy logic and machine learning models trained on historical weather and soil data. While these algorithms can optimize water use and improve crop yield,

they are heavily reliant on high-quality and uninterrupted data streams, which may not always feasible in rural deployments.

Edge computing is new technology that has gained traction in the field as a mitigation strategy to offload certain computation tasks from cloud servers to local devices. In such architectures, gateways like Raspberry Pi perform local decision-making and only push summarized data or alerts to the cloud. While this reduces latency and operational cost, it does not fully resolve concerns related to centralized authority or verifiability of decisions, that could be offered with the implementation of oracles.

A comparative illustration of traditional and the proposed architecture is shown in Figure 3.1, highlighting differences in several categories like trust models, communication flow and decision-making logic.

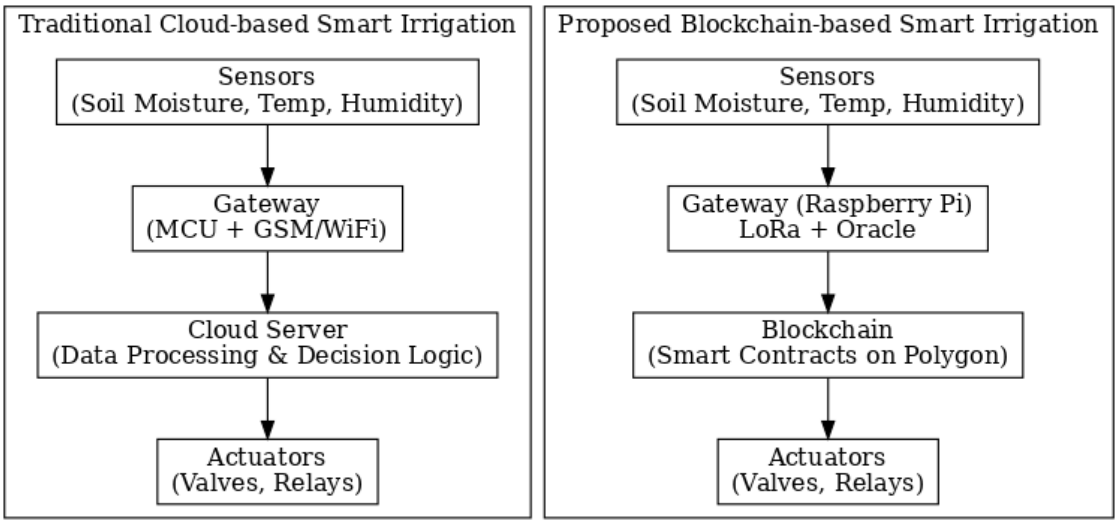


Figure 3.1: Comparison of traditional cloud-based and blockchain-based smart irrigation system architectures.

The proposed system focuses primarily on decentralization and local autonomy, where irrigation decisions are encoded into blockchain smart contracts. This removes the reliance of traditional systems in centralized infrastructure and it also ensures that all decisions are publicly auditable. Furthermore, LoRa communication and low-power microcontrollers allow the architecture to be designed to function in resource-constrained environments with minimal connectivity.

To better understand the current landscape, two notable systems are reviewed below. These examples provide insights into two different architectures and design choices that reflect various trade-offs in functionality, security and scalability.

3.1.1 Example 1: SmartFarmNet Platform

SmartFarmNet is an open-source platform developed for environmental monitoring and smart agriculture, it integrates multiple sensors and cloud-based dashboards. For data

transmission it utilizes ESP8266-WiFi nodes and the MQTT protocol for data transmission, while decision logic is centralized and executed via a remote cloud service [15]. The platform was designed to be modular and extensible too, allowing to configure parameters for the sensor thresholds, data sampling intervals and alerts that are predefined by the user. It has been tested in greenhouse and garden environments where network availability and power are relatively stable.

One of the key advantages of SmartFarmNet is its user-friendly interface and dashboard analytics, as it allows farmers to visualize sensor data trends in real time. However, its dependence on continuous internet connectivity means it is less suitable for remote or rural deployments where GSM or satellite links may be unreliable or expensive. Additionally, the lack of embedded security mechanisms and decentralized decision-making logic can expose the system to have the data tampered with or have service interruptions if the central server fails. Although a promising step toward smarter agriculture, SmartFarmNet represents a conventional IoT-cloud model without leveraging the blockchain technology or edge computing.

3.1.2 Example 2: IoT-Blockchain Integration Prototype by Xie et al. (2022)

In this paper, it is proposed a prototype that integrates IoT-based greenhouse sensors with Ethereum smart contracts for automated environmental control [16]. The system architecture consists of several sensor nodes that capture data on humidity, CO_2 levels, temperature and light. These nodes communicate with a central Raspberry Pi gateway over Wi-Fi, the central unit formats the sensor readings and interacts with a smart contract deployed on the Ethereum mainnet.

The smart contract handles rule-based automation, such as opening ventilation or triggering irrigation based on specific thresholds. All actions are logged immutably on the blockchain, offering a transparent and tamper-proof history of environmental control decisions. The inclusion of oracles allows the Raspberry Pi to act as a trusted source for off-chain data injected into on-chain logic.

This design demonstrates the feasibility of IoT–blockchain integration but presents certain limitations for real-world deployment. It relays on Wi-Fi connectivity and the Ethereum mainnet also introduces higher latency and transaction costs, these are usually impractical for frequent sensor updates in agricultural contexts. Moreover, the gateway still represents a single point of trust, meaning that decentralization is only partial. In contrast, the present project adapts this concept by leveraging LoRa for long-range and low-power communication and the Polygon (Amoy) for low-cost transactions.

3.2 Blockchain Applications in Agriculture

The application of blockchain technology in agriculture has gained considerable attention due to its ability to enhance several factors like transparency, traceability and automation

in complex supply chains. Most of the early use cases are focused on post-harvest processes such as logistics and certification, the recent advancements have expanded blockchain's role to include pre-harvest and field-level operations, with a particular focus in smart farming and environmental monitoring.

A core advantage of blockchain in agriculture is its capacity to maintain immutable and verifiable records of transactions and sensor data. This is particularly important for addressing food safety concerns and ensuring provenance, it also helps organic or sustainable farming industry as it can verify its practices. For example, IBM Food Trust has pioneered blockchain platforms implementation for supply chain traceability and AgUnity has used it for small farmer inclusion [17, 18].

In smart agriculture, blockchain is increasingly being used with IoT sensors to automate the processes at the field-level, as irrigation can be triggered when soil moisture readings fall below a specific threshold and the event is logged permanently on the blockchain. Several projects have explored such applications. In the project from [19], a smart irrigation prototype is used with Ethereum smart contracts in order to control watering schedules based on sensor readings. The sensor nodes transmit data to a local controller that acts as an oracle to interface with the blockchain. Similarly, the system described in [20] combines Hyperledger Fabric with RFID-tagged livestock monitoring, this type of application ensures traceable records of animal health, feeding and movement, allowing a reliable historical record to look. These use cases demonstrate again the flexibility of blockchain across both crop and livestock industries.

Nonetheless, the adoption of blockchain in agriculture still faces several challenges. As energy consumption and transaction costs on public blockchains can still be high for continuous IoT interaction. There are alternatives like Hyperledger or Corda, which are private or permissioned blockchains that mitigate these issues but at the cost of reducing the decentralization and openness of the system. Moreover, the integration of blockchain with low-power IoT networks such as LoRaWAN remains underexplored, as most of the existing and found implementations relay on Wi-Fi or GSM-based communications that are unsuitable for large-scale or energy-efficient deployments.

Despite these barriers, blockchain still has to offer transformative potential in key agricultural areas, these can include:

- **Traceability and Certification:** It ensures the integrity of the food origin, handling and processing claims.
- **Decentralized Automation:** It enables autonomous farm operations like irrigation, fertilization or pest control by using sensors to trigger smart contracts.
- **Data Protection:** Provides farmers to control over their data, which enables them to selectively share or monetize their information.

In the Figure 3.2 it can be seen a detailed visual summary of how blockchain applications span across the entire agricultural chain. The figure illustrates a six-stage process

from the provider to the end consumer, highlighting the physical flow of goods. It also shows the corresponding digital identifiers (barcodes, QR codes, smart certificates) and the progressive accumulation of data blocks within the blockchain network. Showcasing how each factor contributes with validated data to the decentralized ledger and under the oversight of regulatory and certification authorities.

The diagram also emphasizes blockchain's role not just as a storage mechanism, but as a dynamic slid structure for real-time traceability, auditability and trust throughout the food production and distribution cycle. As it combines physical identifiers with on-chain verification, it is possible to reconstruct the full history of an agricultural product, from the seed source and production conditions to the processing, distribution and eventually, the retail and delivery. This can provide consumers and authorities with transparent and verifiable information on the products.

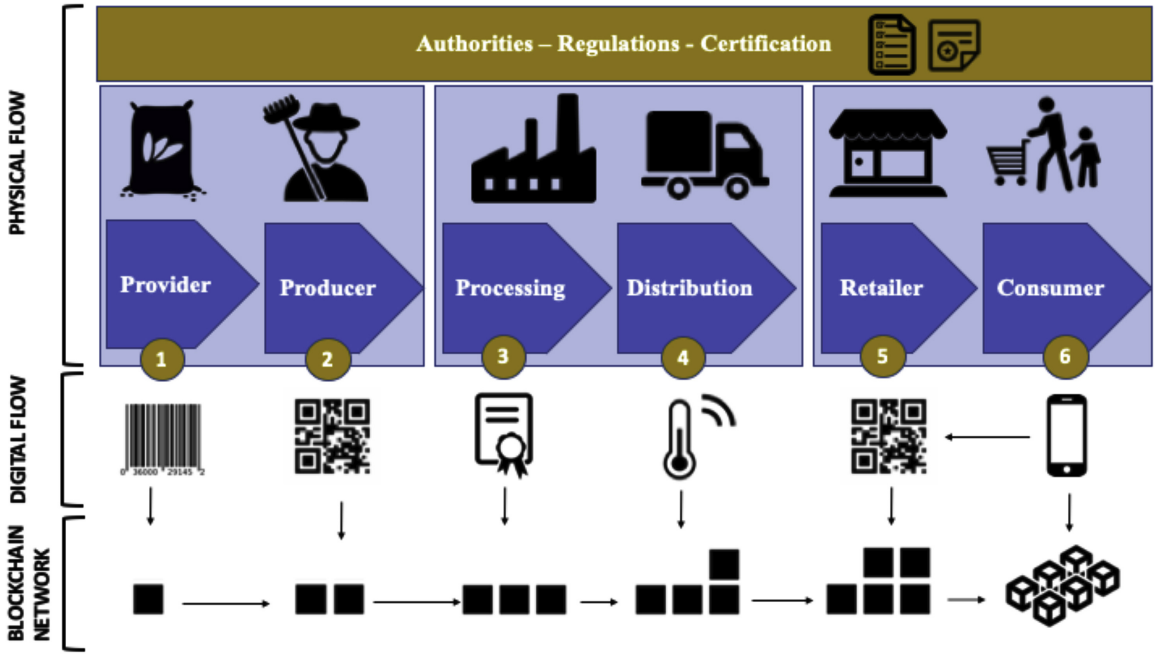


Figure 3.2: Overview of blockchain applications across agricultural domains, from [18].

3.3 Research Gaps and Challenges

Despite the growing projects and experimental systems integrating IoT and blockchain in agriculture, significant research gaps and implementation challenges remain unaddressed.

1. **Decentralized Decision-Making Deficiency:** As most of smart irrigation platforms depend heavily on centralized cloud-based architectures. This can introduce vulnerabilities like single points of failure and latency, particularly when deployed in remote regions where internet connectivity is unreliable. Decentralized logic in remote areas that use verifiable smart contracts is still rare and underdeveloped, causing to not know the totality of this technology behavior in unknown areas.

2. **High Energy Consumption and Costs:** Many blockchain frameworks, especially those based on Proof-of-Work (PoW), cannot be used for agricultural applications due to its high power demands and transaction fees. For example, Ethereum's mainnet often incurs extra costs for regular sensor data recording, causing the system functionality to get expensive. Energy-efficient platforms like Polygon or Solana are being explored recenlty in this context and still needs to see the full potential of them.

3. **Lack of Integration with Low-Power IoT Protocols:** While LoRa and other low-power wide-area networks (LPWANs) are optimal for rural agricultural scenarios, they are rarely integrated with blockchain-based systems. This is because of the protocol incompatibilities that come with it and the limited development frameworks. Bridging this gap requires designing a lightweight connection between IoT and blockchain with also secure protocols.

4. **Fragmented Standards and Interoperability:** Many existing solutions are developed in isolation and using proprietary hardware with closed communication protocols. This causes the scaling and integration of the system difficult across different farms or regions.

5. **Lack of End-to-End Validation in Field Conditions:** Several proposals are still theoretical or only validated via simulation. There is a need for theoretical systems to undergo real life testing with realistic scenarios and incorporating variable soil types, changing weather conditions and intermittent connectivity.

To illustrate this fragmentation of the project topic, Figure 3.3 visualizes the intersection between blockchain, open protocols and low-power IoT in smart agriculture. Based on a synthesis of over 25 academic papers and industry case studies noted in [20, 17, 16]. The data was obtained by categorizing published smart agriculture projects according to whether they used public or private blockchain frameworks, if they relied on open-source or standardized protocols (MQTT, CoAP) and if they implemented low-power wireless communication.

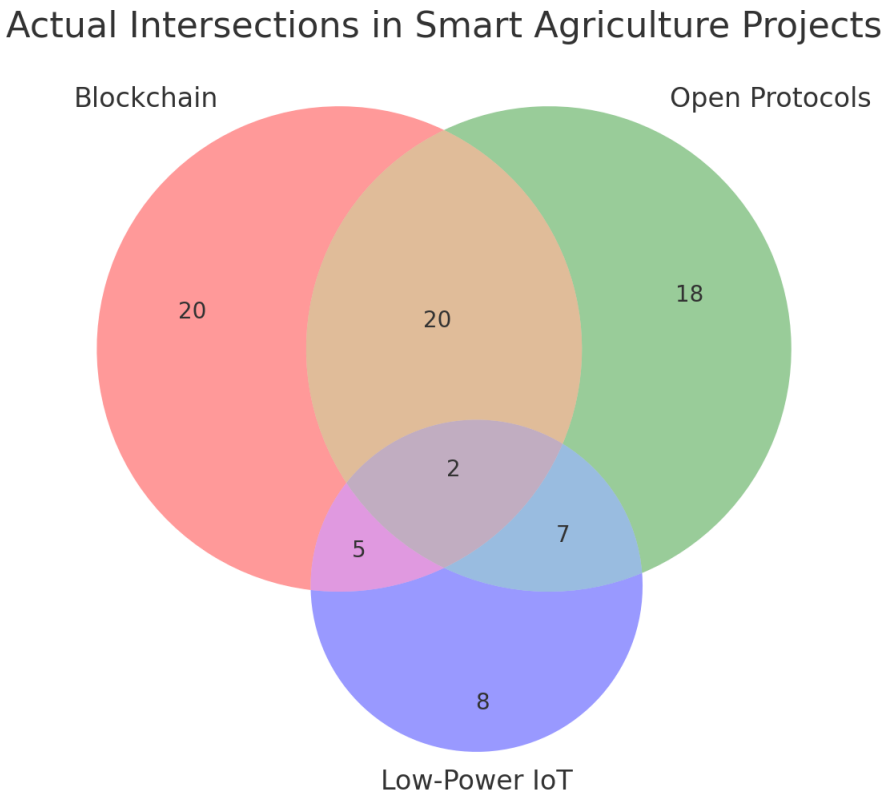


Figure 3.3: Overlap graph of blockchain, open protocols and low-power IoT solutions in smart agriculture [20, 17, 16].

These numbers shows that the proposed system distinguishes itself by addressing all three, contributing with an open and replicable model for future agricultural deployments.

4 System Design and Implementation

4.1 System Architecture

The proposed system for the smart irrigation project combines Internet of Things (IoT) devices with long-range communication and blockchain technology in order to provide secure and automated water management. The architecture is structured into three interconnected layers: the IoT sensor and actuator layer, the gateway and local processing layer and the blockchain and smart contract layer. This layered but joined design allows the system to be scalable, while trying to overcome the challenges from conventional centralized control systems explained previously. These included single points of failure, vulnerability to cyber-attacks and inefficiencies in resource allocation [21, 22, 23, 24, 25].

4.1.1 Layered Overview

- **IoT Sensor and Actuator Layer:** This layer is composed of identical ESP32 nodes that are equipped with environmental sensors (soil moisture, temperature, humidity and light intensity). The nodes are in charge of collecting real-time data and transmit it to the gateway through a LoRa module. Actuators (such as water pumps and valves) would be connected to relay modules and controlled based on smart contract decisions [21, 25].

- **Gateway and Local Processing Layer:** The Raspberry Pi 4 functions as the central hub, it aggregates the sensor readings from multiple ESP32 nodes. Also, it executes pre-processing algorithms to filter noise and detect significant changes before sending the relevant data to the blockchain. This hybrid data-processing model reduces unnecessary blockchain interactions, which will reduce the transaction costs of the network and increase system responsiveness.

- **Blockchain and Smart Contract Layer:** This is the decision-making layer, where the logic is implemented through smart contracts that are deployed on the Polygon blockchain (Amoy testnet). These contracts evaluate incoming data against predefined irrigation rules, ensuring tamper-proof and transparent automation. Blockchain integration eliminates the need for centralized control and provides immutable logs of all irrigation events [22, 23].

4.1.2 Communication Flow

The data flow in the system is illustrated in Figure 4.1. Environmental parameters are continuously measured by ESP32 nodes and transmitted via LoRa to the Raspberry Pi gateway. The Raspberry Pi performs local filtering and, when a significant change is detected (soil moisture below threshold), it forwards the data directly to the blockchain, acting itself as the oracle. The smart contract then evaluates the updated state against predefined rules and triggers irrigation or deactivation events. These decisions are emitted as on-chain events, which the Raspberry Pi listens to in order to control the ctuators. This closed-loop system ensures both local efficiency and trustworthy, auditable decision-making [24].

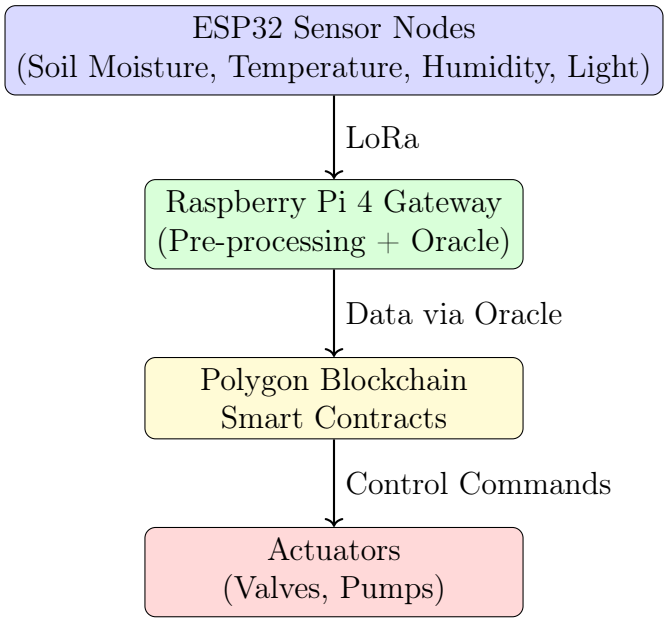


Figure 4.1: System architecture.

4.1.3 Design Rationale

The design choices for the system were done taking into accountthe following considerations:

- **Scalability:** LoRa enables connectivity across large agricultural fields with low power consumption, while making the system suitable for real-world deployment [25].
- **Cost-Efficiency:** The hybrid processing approach minimizes blockchain transactions, thus reducing operational costs without the affecting the data integrity of the transactions.
- **Security and Transparency:** By using blockchain's immutability, the system can prevent the tampering and it also ensures traceability of all irrigation decisions [22].
- **Automation:** Smart contracts enforce irrigation rules automatically, eliminating the need for manual intervention and central oversight.

4.2 Sensor and Hardware Configuration

Part of the effectiveness of the proposed irrigation system relies on the selection of hardware components that will ensure all stated requirements: low power consumption, reliable communication and accurate data collection. The architecture is designed with off-the-shelf (OTS) devices devices in order to balance cost-efficiency and future scalability of the system, plus having broader documentation on commonly used devices. This section details the configuration of each hardware element, including sensor nodes, sensors, actuators, communication modules and the gateway device.

4.2.1 ESP32 Sensor Nodes

For the IoT layer the ESP32-based microcontroller units (MCUs) were used. The ESP32 was selected for its combination of low power consumption, integrated Wi-Fi and Bluetooth capabilities and sufficient computational resources to handle sensor data acquisition and preprocessing. The ESP32 board used in this project is shown in Figure 4.2a. It has a dual-core processor that operates at up to 240 MHz, with 520 KB of SRAM, making it capable of managing real-time sensor data streams while still maintaining low energy requirements [21, 25].

Each ESP32 node acquires a soil moisture (capacitive sensor, analog), a temperature and humidity (DHT22) and a light intensity (LDR, analog). The data is transmitted using a compact LoRa frame from the node to the gateway at a fixed duty cycle. The node firmware is written in C++ (Arduino core) and uses the native ESP32 ADC driver (`driver/adc.h`) for stable analog sampling (12-bit, 11 dB attenuation) with median-of-N aggregation to mitigate the noise in communication. For the DHT22 readings, retry logic is introduce in order to reduce intermittent NaN reads. Also the Wi-Fi/BLE module is disable to reduce power consumption and deep sleep cycles are added between data messages. To avoid redundant uplinks, the node performs light event detection (Δsoil moisture ≥ 2 pp) and maintains a dry-streak counter which the gateway can aggregate into *consecutive dry days*. Full firmware is provided in Appendix A.1 (Listing A.1).

Communication is from the LoRa moduel (SX1276/78) in a point-to-gateway topology (no LoRaWAN MAC) to minimize stack overhead and power draw, this is something common and consistent with long-range agricultural deployments [25, 21]. Also the typical radio settings were used: $f = 868$ MHz (or 915 MHz regionally), SF = 7, BW = 125 kHz, CR = 4/5, $P_{\mathrm{TX}} = +14$ dBm.

To obtain a simple and robust message on the Raspberry Pi, each uplink is a fixed-size binary frame with an explicit header byte and CRC-32 (polynomial compatible with ESP32 `crc32_le`). Table 4.1 details the layout of the frame sent. Also, if DHT gives an error when reading then `tempC` and/or `humPct` are set to -1000.0 and they should be ignored by the gateway, which will retain the last valid values. This type of binary frame is similar to what is showcased in the project [21].

Field (order)	Type / Size	Meaning	Notes
`hdr` (0)	`uint8`	Frame header constant	`0xAA`
`node` (1)	`uint8`	Node identifier	1..255
`seq` (2–5)	`uint32_le`	Monotonic sequence	Wraps on overflow
`soilPct` (6–9)	`float32_le`	Soil moisture (%)	From calibrated ADC
`tempC` (10–13)	`float32_le`	Temperature (°C)	−1000 sentinel if invalid
`humPct` (14–17)	`float32_le`	Relative humidity (%)	−1000 sentinel if invalid
`lux` (18–21)	`float32_le`	Light (normalized lux)	For low/high light rules
`dryStreak` (22)	`uint8`	Consecutive "dry" cycles	Node-level hint
`crc32` (23–26)	`uint32_le`	CRC-32 over bytes 0..22	ESP32 `crc32_le`

Table 4.1: LoRa binary frame format emitted by ESP32 nodes (27 bytes total).

The gateway unpacks this frame and validates the `crc32` frame, then it applies local filters and calibration, such as converting the raw moisture value into soil-relative percentages (moisture mapping) and scaling the light sensor values to a consistent range (light normalization). These calibrations can be adjusted per soil type or sensor model without modifying the on-chain logic, so more flexibility is achieved in the system while keeping the contract simple (Sections 4.3 and 4.4).

4.2.2 Sensors Used (Soil, Environmental, Light)

To monitor the environmental conditions for irrigation, three categories of sensors were integrated:

- **Soil Moisture Sensor:** A capacitive soil moisture sensor (Figure 4.2c) was used due to its durability and immunity to corrosion compared to resistive ones that sometimes are presented in similar projects. It also provides an analog voltage output proportional to soil volumetric water content.
- **Environmental Sensor (DHT22):** The DHT22 (AM2302) sensor (Figure 4.2b) measures both temperature and relative humidity, it offers an accuracy of ±0.5°C for temperature and ±2% for humidity. These parameters will be essential to optimize the irrigation in order to avoid excessive evaporation [22].
- **Light Sensor (LDR):** A light-dependent resistor (Figure 4.2d) was implemented to approximate solar radiation levels. Light intensity affects irrigation scheduling by avoiding watering during peak sunlight, which would otherwise lead to inefficient evaporation losses [24].

A summary can be found in Table 4.2 with the main sensors and their specifications.

Sensor	Measured Parameter	Accuracy	Range
Capacitive Soil Moisture	Soil volumetric water content	±3% (typical)	0–100%
DHT22 (AM2302)	Temperature	±0.5°C	-40–80°C
	Humidity	±2%	0–100% RH
Light-Dependent Resistor (LDR)	Light intensity (approx. lux)	–	0–50,000 lux

Table 4.2: Summary of sensors integrated into the IoT nodes [21, 22, 24, 25].

4.2.3 Actuators and Relays

To physically control the water delivery, the system would employ 5V relay modules connected to the Raspberry Pi. These relays act as electronic switches, they could toggle the operation of irrigation pumps and solenoid valves. This setup can enable automated control of irrigation infrastructure in response to the triggers produced by the smart contracts of the blockchain. Although, for the current prototype of the project the relays were not added, but rather the output signal of the Rapsberry Pi GPIO will represent a irrigation valve.

4.2.4 LoRa Communication Modules

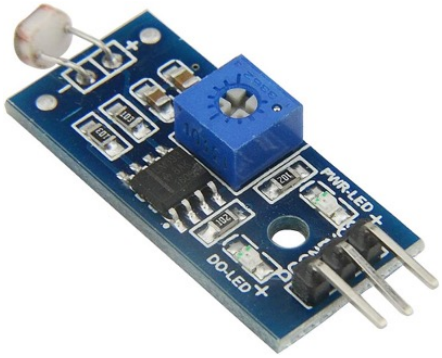
Communication between the ESP32 sensor nodes and the Raspberry Pi gateway is achieved using LoRa (Long Range) transceivers, specifically the HopeRF RFM95W (Figure 4.3b)operating at 868/915 MHz ISM bands, as explained in 4.2.1. LoRa technology was selected for the ability to transmit small data packets over long distances (up to several kilometers in rural environments) with also very low energy consumption. This characteristic makes it superior to Wi-Fi or ZigBee in large agricultural deployments where connectivity is essential for wide areas [25].

The typical LoRa configurations is used in this system, this included a spreading factor (SF) of 7–9, bandwidth of 125 kHz and transmit power of +14 dBm, which provides a balance between range, power consumption and data reliability.

(a) ESP32 microcontroller used as sensor node.

(b) DHT22 (AM2302) temperature and humidity sensor.

(c) Capacitive soil moisture sensor.

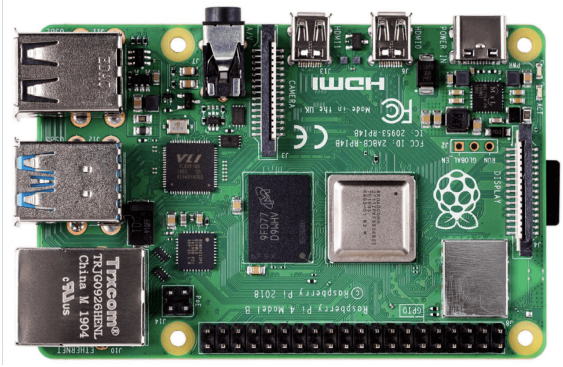


(d) LDR photosensitive sensor for light intensity.

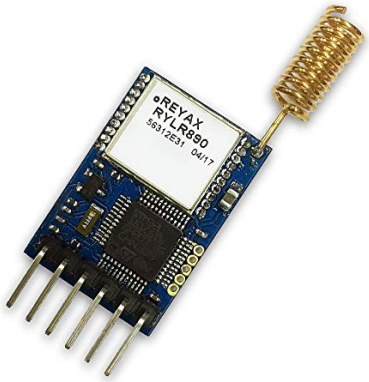
Figure 4.2: Hardware components integrated in the IoT sensor nodes.

4.2.5 Gateway: Raspberry Pi 4

The Raspberry Pi 4 Model B Figure 4.3a is the system gateway. Equipped with a quad-core ARM CPU running at 1.5 GHz and up to 4 GB RAM, it will be responsible for aggregating LoRa messages from multiple ESP32 nodes. It will execute local preprocessing algorithms and interact with the blockchain via the web3. The device runs a lightweight Linux distribution (Raspberry Pi OS), in which Python scripts were implemented for data filtering, packet management and smart contract interfacing. It has higher computational power compared to the ESP32 nodes, as it is in change of tasks with higher computing power



(a) Raspberry Pi 4 used as gateway.



(b) LoRa Transceiver module used for communciation.

Figure 4.3: Gateway and communication hardware used in the irrigation system. The Raspberry Pi 4 (a) acts as the processing and blockchain interface, while the LoRa module (b) receives the communication from the node's module.

4.2.6 Hardware Integration and Power Considerations

The hardware components were integrated with a focus on low power consumption and modularity. At the current stage of implementation, the system is powered primarily through conventional sources:

- **Sensor Nodes:** ESP32 boards and the correspondent sensors are powered via USB from the computer during the set up and testing phase. This configuration simplifies debugging, programming and monitoring during simulations.
- **Gateway:** The Raspberry Pi 4 gateway operates from a stable 5V/3A power supply, currently connected to a wall outlet. The RPi 4 powers the LoRa module itself.

While this is suitable for laboratory porpuses, the configuration is not practical for remote agricultural deployments. For field applications, the system would need to be adapted to renewable and autonomous power solutions:

- **Battery Supply:** Each ESP32 node could be powered by a 3.7V lithium-ion battery with capacity between 2,000–5,000 mAh. This would be sufficient to sustain continuous operation for several days depending on sampling rate and transmission frequency, further power efficient procedures would need to implemented in the code too
- **Solar Charging:** Integration of a compact solar panel (5–6V, 1–2W) with a charge controller would allow nodes to recharge during daylight hours. This could extend the deployment for the lifetime of the hardware itself, plus reduces maintenance requirements, aligning with the needs of agricultural environments [21, 26].
- **Gateway Backup:** The Raspberry Pi gateway, while typically powered from the grid, can also be powered by a larger portable battery pack (20,000 mAh) that would be connected to a solar panel too.

This dual-powering strategy would flexibility to the system as the outlet-powered setup supports development and testing, the solar-powered approach enables sustainable and autonomous operation in real-world agricultural environments [27].

4.3 Data Preprocessing and Local Logic

The Raspberry Pi 4 gateway bridges the raw data generated by ESP32 sensor nodes and the decision-making processes implemented on the blockchain. Instead of having it transmitting every single reading to the blockchain, the gateway implements pre-processing and filtering logic to improve efficiency. This hybrid design combines local computing and decentralized network, aligning with the edge computing paradigm [25, 22].

4.3.1 Frame Parsing, Validation and State Assembly

LoRa packets are read from the attached SX1276/78 transceiver and decoded according to the 27-byte frame structure defined in Table 4.1. Each frame begins with the header byte `0xAA` and pass a CRC-32 check over the payload; otherwise it is rejected. For robustness, fields that arrive with sentinel values (< -999 for `tempC`/`humPct`) are treated as missing and *imputed* on the gateway using the last valid sample stored for that node. The gateway maintains the state of each node with the last-seen timestamp, last valid (post-filter) values per field, short sliding buffers for smoothing and the last values written on-chain to avoid redundant transactions.

The `dryStreak` counter provided by the node (number of deep-sleep cycles under dry conditions) is converted into *consecutive dry days* using the configured duty cycle (`SLEEP_SECONDS`) and expected uplinks per day. When this value changes it is submitted on-chain via `setConsecutiveDryDays()` and then evaluated by the smart contract logic.

4.3.2 Noise Filtering and Data Validation

The agricultural sensor readings can be affected by noise, environmental variability (soil heterogeneity, wind) or hardware inaccuracies. To stabilise the inputs before sending them to the smart contract, the gateway applies some *off-chain* pre-processing techniques. These measures follow best practices in precision agriculture [24, 27], allowing to get stable inputs to reduce the smart contract execution while keeping on-chain costs low.

- **Smoothing:** a short sliding window (default N=3), this uses a robust *median* filter to the soil moisture, temperature, humidity and light values, in order to take out high-frequency fluctuations.
- **Threshold validation:** physically implausible values can be discarded before smoothing (humidity outside $[0, 100]\,\%$, soil outside $[0, 100]\,\%$, temperature outside a certain band or negative lux). When a reading is discarded or missing, the gateway puts the last valid node value and if it does not exists yet, no update is pushed on-chain for that field.

- **Change gating:** after smoothing and validation, the gateway only submits values whose rounded values is different from the last on-chain state for that node, then avoiding redundant transactions.

4.3.3 Event-Driven Data Transmission

As said, the Raspberry Pi gateway uses an event-driven transmission policy, so on-chain updates are only triggered when validated sensor values change meaningfully. This design, already reflected in the change-gating logic described above, avoids redundant writes and ensures that the blockchain state evolves only when the environment does. After applying any necessary setters, the gateway calls `triggerEvaluation()` so that the smart contract can enforce the prioritised irrigation rules (Section 4.4).

4.3.4 Integration with Blockchain as Oracle

Unlike conventional IoT–oracle integrations that rely on external services (Provable), the Raspberry Pi directly assumes the oracle role in this project. The validated frames are encoded using the Application Binary Interface (ABI) and submitted to the Polygon Amoy test network using Web3 libraries. Transactions are signed locally with the gateway account (using the private key stored in environment variables) and executed towards the deployed smart contract. The gateway also subscribes to contract events (`EvaluationTriggered`, `IrrigationStatusChanged`) to actuate respectively to the GPIO output representing the irrigation relay. The complete Python implementation of this workflow is provided in Appendix A.2 (Listing A.2).

4.3.5 Design Rationale

The use of local pre-processing and event-driven logic gives the system many advantages:

- **Efficiency:** redundant or unchanged sensor updates are avoided then reducing blockchain load and lowering power consumption in sensor nodes.
- **Robustness:** integrity checks, smoothing and last valid values protect the contract from noise and transient sensor failures.
- **Cost-effectiveness:** by transmitting only meaningful changes, transaction fees are minimized without compromising transparency.
- **Scalability:** the gateway also maintains independent state for each node, this allows multiple LoRa nodes to connect without overwhelming the blockchain with unnecessary writes of each node.

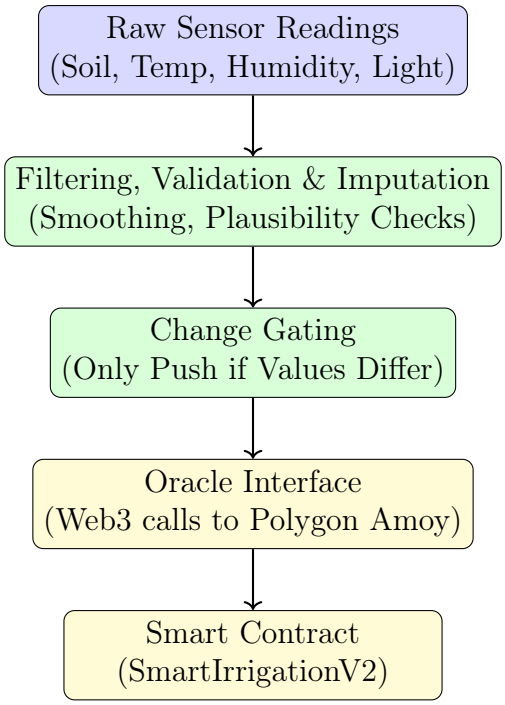


Figure 4.4: Data preprocessing, validation and event-driven oracle workflow at the Raspberry Pi gateway.

The Raspberry Pi 4 gateway (Listing A.2) completes the edge–to–chain loop: it ingests the 27-byte LoRa frames defined in Table 4.1, then validates its integrity with CRC-32, imputes missing fields from last valid values, aggregates the node-level *dry streak* into *consecutive dry days* and finally applies a minimal-write condition (only changed fields are pushed) before calling `triggerEvaluation()` on `SmartIrrigationV2` (Section 4.4). The gateway also subscribes to `IrrigationStatusChanged` to listen to the irrigation events and then the Raspberry Pi GPIO would be turned on or off accordingly, representing the relay drive signal for later actuator integration.

To keep the gateway generic and reproducible, all parameters are provided through a `.env` file (Listing A.3): RPC endpoint, contract and account addresses, private key (never committed), serial port for the LoRa bridge, sleep interval for dry-day aggregation and the optional GPIO pin for the demo actuation. This separation of configuration from source code enables safe sharing of the Python script while preserving security of credentials.

4.4 Blockchain Integration and Smart Contracts

The irrigation rules are encoded through smart contracts, the system ensures that every action for whether activation or deactivation of irrigation is executed transparently and recorded on-chain through transactions submitted by the gateway. This is particularly relevant in agriculture, where trust and reproducibility of records are essential for sustainable management and accountability [22, 23, 27].

4.4.1 Smart Contract Design and Logic

The contract developed, named `SmartIrrigationV2`, encodes the irrigation decision rules. It maintains a compact state representation of the environment and the irrigation system. Specifically, it defines the following:

- **State variables** that store the most recent sensor readings: `soilMoisture`, `temperature`, `humidity`, `lightIntensity` and `consecutiveDryDays`.
- **Threshold parameters** that are stored on-chain, they include temperature bounds, soil moisture thresholds, humidity and light constraints and the dry-day counter limit. These parameters define the decision where updates are evaluated.
- **Irrigation status flag** (`irrigationStatus`), representing whether irrigation is currently ON or OFF.
- **Access control** is enforced through the `onlyOwner` modifier, which restricts sensor updates to the authenticated gateway, preventing writes from untrusted users.
- **Event interface** functions are `IrrigationStatusChanged`, `EvaluationTriggered` and the sensor-specific update events. These log the decisions and changes in the system values on-chain, creating the trail of decisions..

The decision logic is organised into a hierarchy of prioritised conditions, designed to prevent conflicts between triggers and in this way optimising water efficiency. The highest priority is deactivation when temperature exceeds 35 °C, since irrigation under such conditions would lead to high evaporation losses and inefficient water use [28, 29]. Next, it is the irrigation activation when soil moisture is below 30%, temperature is within the optimal band of 20–28 °C and humidity remains below 60%, this represents the most favourable condition for effective watering [30, 31].

Additional rules address stress conditions, like if soil moisture remains below 35% for three consecutive days then irrigation is activated regardless of other factors to prevent prolonged drought stress [32]. Similarly, if soil moisture is low and light intensity is below a threshold (typically morning or evening), irrigation is allowed to maximise uptake under low-evaporation conditions. Conversely, irrigation is always deactivated once soil moisture exceeds 60%, ensuring that overwatering and root damage happens.

Earlier prototypes included a simpler soil-moisture–temperature rule, but this was found to be redundant: the soil-moisture–temperature–humidity condition already captures optimal watering scenarios, while the other rules ensure coverage for exceptional cases (dry spells and low-light periods). Removing the redundant trigger improved the robustness and interpretability of the system.

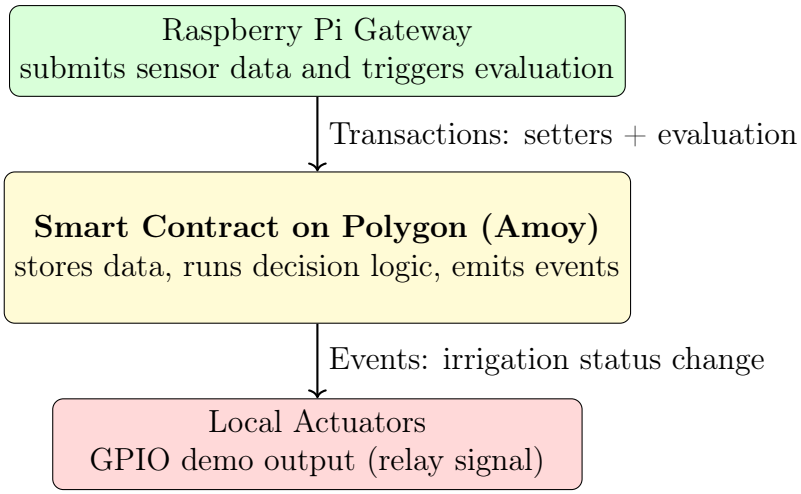


Figure 4.5: Interaction cycle between the gateway, the smart contract on Polygon Amoy and the actuator demo interface.

Function	**Purpose**	**Access Control**	**Associated Event**
`setSoilMoisture(uint)`	Update soil moisture value	Owner only (gateway)	`SoilMoistureUpdated`
`setTemperature(uint)`	Update ambient temperature	Owner only (gateway)	`TemperatureUpdated`
`setHumidity(uint)`	Update humidity level	Owner only (gateway)	`HumidityUpdated`
`setLightIntensity(uint)`	Update light intensity	Owner only (gateway)	`LightIntensityUpdated`
`setConsecutiveDryDays(uint)`	Record consecutive dry days	Owner only (gateway)	–
`triggerEvaluation()`	Run priority-based evaluation	Owner only (gateway)	`EvaluationTriggered`, `IrrigationStatusChanged`
`getIrrigationStatus()`	Read irrigation status	Public (any user)	–

Table 4.3: Main functions and events of the implemented contract, with roles and access control.

This design follows recommended patterns for gateway-oracle IoT systems, where the authenticated Raspberry Pi gateway acts as the only writer to the blockchain, preventing tampering with the sensor inputs [22, 23].

4.4.2 Security and Robustness Considerations

Beyond decision-making, the smart contract was designed with principles of security and maintainability. The use of owner-gated setters (`onlyOwner`) prevents unauthorized updates of sensor values, while event-driven actuation (`IrrigationStatusChanged`, `EvaluationTriggered`) creates an auditable trail for every irrigation decision made on-chain [22]. By keeping thresholds within the contract, changes that want to be made inside the smart contract conditions such as adjusting soil moisture limits or temperature ranges, it becomes explicit and verifiable on the blockchain, this ensures transparency for future stakeholders.

From a robustness perspective, several observations emerged during testing. First, some setters (`setTemperature`, `setHumidity`) initially compared sensor values after reassignment, causing update flags never to trigger; this was corrected by checking before assignment.

Second, the precondition in `triggerEvaluation()` required that at least one sensor flag was updated, the error message suggested that all sensors were mandatory. This was revised for consistency to avoid misleading runtime errors. Third, the contract has compact "reason bits" for irrigation triggers, which provides a concise encoding of activation logic. Finally, while thresholds are currently defined as public constants, the design allows extending configurability by introducing dedicated setter functions for calibration at runtime, without redeploying the contract.

4.4.3 Deployment with Remix IDE and MetaMask

The contract is implemented and tested using the Remix IDE, this platform offers an easy to use environment for Solidity development. The deployment process relied on MetaMask as the transaction provider. MetaMask was configured with the Polygon Amoy test network, using a test account funded with MATIC to pay for gas fees. To ensure consistency with the system architecture, the private key of the Raspberry Pi oracle account was imported into MetaMask. This guarantees that the gateway's wallet is set as the contract `owner`, aligning the onlyOwner restrictions in the smart contract with the trusted oracle role.

Deployment followed a standard procedure: compiling the contract, selecting `SmartIrrigationV2` and then confirming the deployment transaction in MetaMask. Once the transaction was validated, Remix displayed the contract address and generated interactive function panels based on the ABI. These functions were then callable directly through the Remix interface, while the same ABI and address could also be used by external scripts running on the Raspberry Pi gateway. This workflow bridges the development environment with the real blockchain, ensuring that irrigation logic is enforced securely and transparently in accordance with the prototype's design.

4.4.4 Operational Workflow and Event Semantics

Once deployed, the smart contract was invoked directly by the gateway, which in this prototype served as the oracle. The Raspberry Pi periodically updates soil moisture, temperature, humidity and light readings, by calling the respective setter functions. After updates, the gateway triggers the evaluation routine and the contract applied the prioritisation logic described earlier. The result is communicated through events, which the gateway listens to in order to activate or deactivate local pumps and valves.

The `IrrigationStatusChanged` event includes two boolean flags (`bit1`, `bit0`), which serve as lightweight identifiers of the activation or deactivation reason (high temperature, consecutive dry days or optimal soil moisture). This "reason encoding" ensures that post-hoc analyses or audits can reconstruct the logic path without duplicating state variables on-chain.

In the current prototype, the event `IrrigationStatusChanged(status, bit1, bit0)` is mapped to a Raspberry Pi GPIO output in *demo mode*, GPIO HIGH → irrigation ON and GPIO LOW → OFF, allowing end-to-end validation of the decision loop from oracle-gateway to smart contract to actuator, without energizing field hardware (Appendix A.2).

In the Table 4.4 the details of the encoding of `bit1` and `bit0` flags are explained, this encoding is done in the `IrrigationStatusChanged` event. Because of having the bit pattern, it must be interpreted together with the `status` field, the same pair (0,0) can represent different outcomes depending on whether irrigation was activated or deactivated. This compact scheme reduces on-chain storage overhead but it requires off-chain logic to be "decoded" for understanding causes.

Status	**(bit1, bit0)**	**Reason**
Deactivate	(0,0)	Temperature above 35 °C
Activate	(0,0)	Low soil moisture + optimal temp/humidity
Activate	(0,1)	Consecutive dry days threshold reached
Activate	(1,0)	Low soil moisture + low light intensity
Deactivate	(0,1)	Soil moisture above 60% (optimal)

Table 4.4: Encoding of activation/deactivation reasons in the `IrrigationStatusChanged` event.

4.4.5 Reproducibility and Verification

To ensure reproducibility and independent verification every build and deployment step produced machine artifacts, this are readable and can be used to reconstruct the contract, verify its authenticity and redeploy it on other networks if needed.

First, the `SmartIrrigationV2_metadata.json` file, generated by the Solidity compiler, records essential build parameters: compiler version (`0.8.28+commit.7893614a`),

target contract (`SmartIrrigationV2`), EVM version (`cancun`), optimizer settings (disabled, runs=200) and a Keccak-256 hash of the source file with IPFS/BZZ content-addressed URLs. These entries guarantee that identical bytecode can be reproduced from the published source. An snipet showing these settings and the associated ABI is provided in Appendix B, Sections B.1 and B.2.

Next, the `SmartIrrigationV2.json` file contains the Application Binary Interface (ABI), which defines the exact functions and events callable from off-chain clients, including `setSoilMoisture()`, `triggerEvaluation()` and `getIrrigationStatus()`. This ABI is the contract "surface" used by the Raspberry Pi gateway (acting as oracle) to submit sensor updates and trigger evaluations. A minimal ABI slice illustrating these key functions is provided in Appendix B, Section B.3.

Finally, the `scenario.json` file documents the actual deployment activity during development in Remix IDE. It includes the account address used for deployment and a chronological record of constructor transactions with timestamps, bytecode hashes and ABI references. This file serves as a deployment journal, linking the abstract contract specification to a concrete on-chain instance. An excerpt illustrating the account binding and constructor record is given in Appendix B, Section B.4.

Together, these artifacts (metadata, ABI and scenario log) form a complete reproducibility trail. They allow third parties to verify the source and compilation settings, interact with the contract using the published ABI and validate deployment provenance against the documented account and transaction history.

4.5 Oracle Configuration

In agricultural IoT applications, most relevant information such as soil moisture, temperature or light intensity is off-chain and must be injected into the blockchain through a trusted interface.

The initial design of this project considered integrating a decentralized oracle service such as Provable or Chainlink. However, due to the complexity of integration and the limited timeframe of the prototype, this role is implemented directly by the Raspberry Pi 4 gateway. Acting as a web3-enabled oracle, the gateway bridges sensor nodes and blockchain execution while ensuring integrity and reproducibility of the transmitted data [22, 23]. This approach reduces architectural complexity, while leaving decentralized oracle networks as a natural extension for future work.

4.5.1 Role of the Raspberry Pi as Oracle Gateway

The Raspberry Pi acts as a trusted oracle by ingesting LoRa frames from ESP32 sensor nodes, preprocessing and validating them (Section 4.3) and then serializing the verified values into blockchain transactions. These are signed locally using the gateway's private key and submitted to the Polygon (Amoy test network), ensuring authenticity and

preventing unauthorized updates. In addition, the gateway listens to contract events to maintain synchrony with on-chain logic (see Section 4.4).

4.5.2 Transaction Flow

The oracle workflow is structured as follows:

1. **Data ingestion and validation:** LoRa frames are decoded and verified using CRC-32.
2. **Filtering and event detection:** Implausible values are discarded and dry streaks are aggregated (Section 4.3).
3. **On-chain update:** Only changed values are submitted via contract setters, followed by `triggerEvaluation()`.
4. **Event synchronisation:** Contract decisions are given back to the gateway through events.
5. **Logging:** Each transaction hash and receipt is logged locally for reproducibility and traceability.

4.5.3 Implementation Considerations

The oracle is implemented in Python using Web3 libraries for Ethereum-compatible chains. The gateway uses the contract ABI exported during compilation (Appendix B) to encode function calls and deployment metadata (`SmartIrrigationV2_metadata.json`, `SmartIrrigationV2.json` and `scenario.json`) to guarantee reproducibility. Runtime configuration, including RPC endpoints and hardware interfaces, is externalized via the `.env` file (Appendix A.3), taking out sensitive values from the source code.

4.5.4 Security and Trust

As the trusted oracle, the Raspberry Pi is responsible for maintaining integrity between the physical environment and blockchain execution. This role is secured through:

- **Access control:** Only the gateway account can call setters, enforced by the `onlyOwner` modifier.
- **Immutable audit trail:** All updates are logged on-chain through events, allowing verification of irrigation decisions.
- **Reproducibility:** Compiler metadata, ABI and deployment logs (Appendix B) allow independent verification that the deployed bytecode matches the published source.

Future versions may delegate this oracle function to decentralized networks such as Chainlink or Provable to reduce reliance on a single trusted gateway.

5 Results and Validation

5.1 Testing Strategy and Scenarios

The evaluation of the proposed system was carried out in a simulated environment set up manually, as the prototype has not yet been deployed under real agricultural field conditions. As a preliminary step, the ESP32 sensor node was connected to the network via WiFi, allowing direct inspection of sensor readings. This confirmed that the sensing components (soil moisture, temperature, humidity, and light) were correctly wired and provided valid measurements under normal operating conditions.

When attempting to transition from WiFi to LoRa for long-range communication, difficulties were encountered in configuring the LoRa receiver on the Raspberry Pi (RPi4). Despite correct wiring and module initialization, the gateway was unable to consistently receive frames from the ESP32 node. As a result, validation of the end-to-end LoRa link could not be completed within the available timeframe. To ensure functional evaluation of the system logic, a direct simulation approach was implemented on the RPi4.

5.1.1 Scenario design

The testing reproduced representative situations the irrigation system could encounter. Two main scenarios were defined:

1. **Normal conditions:** soil moisture decreases gradually over time while temperature and humidity remain within optimal ranges. Once soil moisture drops below the 30% threshold, the gateway submits an update to the smart contract, which activates irrigation. Irrigation is deactivated once soil moisture rises above the 60% threshold.

2. **Edge case (safety override):** when temperature exceeds 35°C, irrigation is forcibly deactivated, even if soil moisture is below the activation threshold. This prevents inefficient watering during periods of high evaporation risk and validates the prioritisation of safety constraints.

5.1.2 Sensor values simulation

Since LoRa reception could not be validated, a Python simulator was executed directly on the RPi4 to inject sensor values into the blockchain interaction layer. This script can be seen in the Appendix C.1, this bypasses the radio parsing stage and directly invoked the contract functions (`setSoilMoisture`, `setTemperature`, `setHumidity`, `setLightIntensity`),

followed by `triggerEvaluation()`.

This ensured that the gateway logic could still be tested end-to-end against the deployed smart contract. The two scenarios defined above were reproduced by scripted sequences of values, confirming both activation and deactivation rules.

Table 5.1 summarises the scripted values used to drive each scenario.

Table 5.1: Scripted inputs used in the RPi4 simulator for each scenario.

Scenario	**Soil (%)**	**Temp (°C)**	**Hum (%)**	**Light (lux)**
Normal (decreasing)	45, 41, 37, 33, 29	25	50	12,000
Normal (post-irrigation)	52, 58, 61	25	50	12,000
Edge (safety override)	30	36, 37, 38	50	15,000

5.1.3 Blockchain interaction and logs

The RPi4 simulator successfully connected to the Polygon Amoy test network and executed the transactions. The printed logs closely followed the format of the original gateway, confirming the chain of actions, the full logs can be seen in C.2. The logs show the value reception, contract updates, evaluation trigger and event decoding.

The two scenarios are simulated successfully as it can be seen for the Normal scenario, the soil moisture dropped below 30% and the contract activated irrigation:

```
INFO IrrigationStatusChanged -> active=True reason=1
```

Similarly, for the Edge scenario, when temperature rose above 35°C, the safety override was triggered, forcing irrigation off:

```
INFO IrrigationStatusChanged -> active=False reason=2
```

The complete console output, including transaction hashes, mined block numbers and gas usage, is provided in Appendix C.2.

5.2 Performance Metrics (Latency, Energy, Costs)

5.2.1 Gas usage and transaction cost from simulator logs

Given the RPi4 gateway's behaviour (local pre-processing and change detection), setters are only sent when a value *meaningfully changes* (e.g., threshold crossing). After any such setter, the gateway immediately calls `triggerEvaluation()` so that the contract re-computes irrigation. This is exactly what the simulator executed.

From the simulator logs, the eight transactions and their `gasUsed` are:

- `setSoilMoisture` ×3: 46,241 gas each
- `setTemperature` ×1: 45,112 gas

- `triggerEvaluation` ×4: 48,712, 48,977, 48,980, 49,011 gas

Total gas:

$$\text{Gas}_{\text{total}} = 3 \cdot 46{,}241 + 45{,}112 + (48{,}712 + 48{,}977 + 48{,}980 + 49{,}011) = 379{,}515 \text{ gas.}$$

To convert gas to USD we use a representative Polygon gas price and an illustrative MATIC/USD conversion (gas price references: [33, 34, 35]):

$$\text{Cost [USD]} = \text{GasUsed} \times \text{GasPrice[gwei]} \times 10^{-9} \times \text{MATIC\_USD.}$$

With GasPrice = 30 gwei and MATIC_USD = 0.75, the full validation run costs:

$$379{,}515 \times 30 \times 10^{-9} \times 0.75 \approx \mathbf{0.0085\ USD},$$

i.e. < 1 cent in total. For reference, a single *setter+evaluation* pair from the logs consumes about

$$(46{,}241 + 48{,}950) \approx 95{,}141 \text{ gas}$$

$\approx$ \$0.00214 at the above prices. The `setTemperature`+evaluation pair in the Edge scenario was $(45{,}112 + 49{,}011) = 94{,}123$ gas $\approx$ \$0.00212.

5.2.2 Daily transaction volume and daily cost

Because the RPi4 sends setters only when values *change* and calls `triggerEvaluation()` immediately after, the daily transaction count depends on how often thresholds are crossed, not on the sampling period itself. We analyse three policies:

(A) Naïve periodic writes (upper bound, *not* our design). Every $T = 5$ minutes, write all four fields and then evaluate (5 tx/cycle). With 288 cycles/day:

$$N_{\text{tx/day}} = 288 \times 5 = 1440 \text{ tx/day.}$$

Per-cycle gas $\approx 4 \cdot 46{,}241 + 48{,}950 = 233{,}864$ gas; daily gas $\approx$ 67.37M. At 30 gwei, MATIC=\$0.75 this is $\approx$ \$**1.52/day**.

(B) Event-driven (our actual behaviour). Only on meaningful change, send the corresponding setter followed by `triggerEvaluation`. In a typical diurnal pattern:

- One daily baseline boot (set all four fields once, then evaluate): $4 \times 46{,}241 + 48{,}950 = 233{,}864$ gas $\approx$ \$0.00526.
- Soil crosses activation and later recovery thresholds (two crossings): $2 \times (46{,}241 + 48{,}950) = 190{,}282$ gas $\approx$ \$0.00428.
- Optional high-temperature override once (hot day): $(45{,}112 + 49{,}011) = 94{,}123$ gas $\approx$ \$0.00212.

Thus:

$$\text{Cost/day} \approx \begin{cases} \$0.00526 + \$0.00428 = \$\mathbf{0.00954} & \text{(no temp override)} \\ \$0.00526 + \$0.00428 + \$0.00212 = \$\mathbf{0.01166} & \text{(with one temp override)} \end{cases}$$

and the corresponding transaction counts are

$$N_{\text{tx/day}} = \begin{cases} 5 \text{ (boot)} + 2 \cdot 2 \text{ (soil pairs)} = \mathbf{9} \text{ tx/day}, \\ 5 \text{ (boot)} + 2 \cdot 2 \text{ (soil pairs)} + 2 \text{ (temp pair)} = \mathbf{11} \text{ tx/day}. \end{cases}$$

This reflects the RPi4's pre-processing: no setters are sent if values are stable, so daily costs are driven by threshold events, not by sampling rate.

(C) Hybrid: periodic evaluation only. Schedule `triggerEvaluation` every 30 minutes (48/day) as a watchdog, but still send setters only on changes. The 48 evaluations alone cost $48 \times 48{,}950 = 2{,}349{,}600$ gas $\$\mathbf{0.0529}$/day. Adding two soil crossings and one temperature override adds $\$0.00428 + \$0.00212 \approx \$0.00640$, totaling $\mathbf{\$0.0593/day}$. This gives a predictable ceiling while keeping writes sparse.

In all cases, on-chain fees on Polygon remain low; operational policy should be tuned with live gas data from [33, 34, 35].

5.2.3 Cloud IoT reference costs

Cloud IoT platforms usually charge by message volume and service tiers, but always on a recurring subscription basis. As of 2025, AWS IoT Core lists a rate of $1.00 per million messages (around 5KB increments) for the first billion messages [36, 37]. Microsoft Azure IoT Hub publishes tiered quotas where the S1 tier ($25/month) includes 400,000 messages/day of up to 4KB each [38].

In our case, the event-driven gateway can produce ≈ 10-15 control events per day (as in our simulations) would generate <500 messages/month. On cloud IoT pricing this is effectively charged at the *minimum tier*, $25/month (Azure) or several dollars/month (AWS). In contrast, our blockchain-enabled system on Polygon consumed only ≈ $0.01 per day, or ≈ $0.30 per month, at the assumed gas price. Even with the stablished conservative watchdog policy (Policy C, ≈ $0.06/day), monthly cost remains below $2. This is still an order of magnitude lower than cloud IoT subscription fees. This shows that for sparse control actions, blockchain-on-chain costs are substantially cheaper than cloud message-tier minimums. At larger scales, cloud platforms benefit from volume discounts, but the cost advantage of event-driven blockchain interactions persists.

5.2.4 Latency and energy

Although it is not measured directly here, prior work on blockchain–IoT integration shows that the typical Polygon network block times are 2-3 seconds [33], meaning our control decisions propagate near-real time. Additionally, the energy costs are dominated by wireless transmission (LoRa), not by the cryptographic signing itself. Studies report that

event-driven models limit energy overhead to low levels compared to sensor operation and wireless radio duty cycles [39, 40]. This aligns directly with our event-driven gateway policy (B), where writes occur only on threshold events rather than continuously, thus reducing both the energy and the transaction costs.

5.3 Discussion

The simulation that was done showed contract behaviour for both scenarios and also produced on-chain events consistent with the rules implemented: low soil → irrigation activated; high temperature → safety deactivation. As LoRa reception could not be completed within the timeframe, validation was done between the RPi4 and the blockchain, ensuring that the smart contract logic executed as intended and that the emitted events matched expected conditions.

From the economic point of view, the measured gas usage from the logs demonstrates that a typical daily profile (baseline + threshold crossings) consumes fewer than a dozen transactions, resulting in a cost under one cent per day. These values are orders of magnitude cheaper than maintaining a conventional IoT cloud subscription, which requires monthly minimums of $25 for Azure IoT Hub S1. At a scale, this difference is noted for blockchain in a linearly way with events, while cloud subscriptions impose tier costs.

6 Conclusions and Future Work

The development of this prototype has shown that it is technically feasible to combine IoT sensor networks with blockchain technology to create a secure and automated irrigation system. The system demonstrates how irrigation decision can be automated by integrating ESP32 nodes with LoRa communication, a Raspberry Pi gateway and smart contracts deployed on the Polygon network. The gateway proved to be a key component, since it filtered and pre-processed the data before submitting it on-chain, which reduced redundant interactions and lowered costs. All decisions were recorded immutably, offering a traceable and auditable record that could be especially useful in cooperative or regulated agricultural environments. The economic analysis also underlined the viability of this approach, as the operational cost of blockchain transactions was considerably lower than that of conventional cloud-based IoT platforms. Overall, the project highlights that blockchain-enabled IoT irrigation can be both efficient and sustainable, contributing to better use of resources and aligning with global objectives for responsible water management.

At the same time, the work has important limitations that must be acknowledged. The validation was carried out in a simulated environment rather than in real agricultural conditions. Although the logic of the system was successfully tested through direct interaction with the blockchain, the LoRa communication link between sensor nodes and the gateway could not be fully established within the timeframe of the project. The Raspberry Pi, acting as a single oracle, also represented a point of centralization that does not fully reflect the decentralized potential of blockchain. In addition, the physical actuation of pumps or valves was not implemented and metrics such as latency, energy consumption and system robustness in field conditions still need to be measured.

For these reasons, future work should focus on taking the prototype from simulation to practical deployment. Field testing will make it possible to measure the behaviour of the system in real conditions, accounting for variability in soil, climate and connectivity. The oracle component should evolve towards decentralized solutions such as Chainlink or Provable, removing the dependency on a single gateway and increasing the trust in the system. On the hardware side, the addition of solar-powered sensor nodes and weatherproof enclosures would improve autonomy and resilience, while integrating actual irrigation hardware would close the loop between decision and action. Beyond technical improvements, the system could also be extended with predictive models that incorporate weather forecasts or crop-specific requirements, offering more refined irrigation schedules. Finally, exploring its use in cooperative farming contexts would provide valuable insight into its socio-economic impact, particularly in terms of transparency, cost-sharing and

governance.

In conclusion, the project lays a solid foundation for a new approach to irrigation automation, one that combines technological innovation with sustainability. While the current version remains a prototype, the results confirm its potential and point clearly towards the next steps required to transform it into a reliable, scalable and widely applicable solution for precision agriculture.

A Source Code

A.1 ESP32 Firmware

```
/*****************************************************************
 * Smart Irrigation     ESP32 Sensor Node (LoRa)
 * Board: ESP32
 * Sensors: DHT22 (temp/hum), Capacitive Soil (ADC), LDR (ADC)
 * Radio: SX1276/78 (RFM95W) via Sandeep Mistry LoRa lib
 * Analog reads use ESP32 ADC driver (driver/adc.h) on ADC1.
 *****************************************************************/

#include <Arduino.h>
#include <WiFi.h>
#include "esp_bt.h"
#include <SPI.h>
#include <LoRa.h>
#include "DHT.h"
#include "rom/crc.h"
#include "driver/adc.h"

#define NODE_ID              1
#define FW_VERSION_MAJOR     1
#define FW_VERSION_MINOR     1

/********* Pins & Sensors *********/
#define PIN_DHT              4
#define DHT_TYPE             DHT22

// ADC1-only GPIOs: 32  39
#define GPIO_SOIL            34   // ADC1_CHANNEL_6
#define GPIO_LDR             35   // ADC1_CHANNEL_7

// ADC driver mapping
static const adc1_channel_t CH_SOIL = ADC1_CHANNEL_6; // GPIO34
static const adc1_channel_t CH_LDR  = ADC1_CHANNEL_7; // GPIO35

/********* LoRa pins *********/
#define LORA_SCK             5
#define LORA_MISO            19
#define LORA_MOSI            27
#define LORA_SS              18
#define LORA_RST             14
#define LORA_DIO0            26

```

```
/********* LoRa radio params *********/
#define LORA_BAND             915E6   // set to 868E6 Europe
#define LORA_SF               7
#define LORA_BW               125E3
#define LORA_CR               5
#define LORA_TX_POWER_DBM     14

/********* Sampling *********/
#define SAMPLE_RETRIES            5
#define SEND_ON_CHANGE_THRESH     2.0f    // % soil moisture delta
#define PERIODIC_SEND_EVERY_N     12      // force send every N cycles
#define DRY_THRESHOLD_PERCENT     35.0f
#define SLEEP_SECONDS             300     // 5 minutes

/********* Moisture calibration (raw->%) *********/
#define SOIL_RAW_DRY          2800.0f
#define SOIL_RAW_WET          1300.0f

/********* LDR normalization *********/
#define LDR_MIN_RAW           200.0f
#define LDR_MAX_RAW           3500.0f
#define LDR_MAX_LUX           50000.0f


/********* RTC for deep sleep *********/
RTC_DATA_ATTR uint32_t rtc_boots      = 0;
RTC_DATA_ATTR uint32_t rtc_seq        = 0;
RTC_DATA_ATTR uint8_t  rtc_dry_streak = 0;
RTC_DATA_ATTR float    last_sent_moist = NAN;
RTC_DATA_ATTR float    last_sent_temp  = NAN;
RTC_DATA_ATTR float    last_sent_hum   = NAN;
RTC_DATA_ATTR float    last_sent_lux   = NAN;

DHT dht(PIN_DHT, DHT_TYPE);

/****** Helpers *********/
static inline bool approxChanged(float a, float b, float eps) {
  if (isnan(a) || isnan(b)) return true;
  return fabsf(a - b) >= eps;
}

// Configure ADC1 width & attenuation
void adcInitChannel(adc1_channel_t ch) {
  // 12-bit width
  adc1_config_width(ADC_WIDTH_BIT_12);
  // 11 dB attenuation ~ full-scale near 3.3V
  adc1_config_channel_atten(ch, ADC_ATTEN_DB_11);
}

// Median of N raw samples from an ADC1 channel
int adcReadMedian(adc1_channel_t ch, uint8_t n =  nine) {
  if (n < 3) n = 3;
  if (n > 25) n = 25;
  int buf[25];
  for (uint8_t i = 0; i < n; ++i) {
    buf[i] = adc1_get_raw(ch);
```

```
    delay(8);
  }

  // insertion sort
  for (uint8_t i = 1; i < n; ++i) {
    int k = buf[i], j = i - 1;
    while (j >= 0 && buf[j] > k) {
        buf[j+1] = buf[j]; j--;
    }
    buf[j+1] = k;
  }
  return buf[n/2];
}

// Map soil raw ADC -> percentage %
float mapSoilToPercent(int raw) {
  float r = constrain((float)raw, SOIL_RAW_WET, SOIL_RAW_DRY);
  float pct = 100.0f * (SOIL_RAW_DRY - r) / (SOIL_RAW_DRY - SOIL_RAW_WET
     );
  return constrain(pct, 0.0f, 100.0f);
}

// Normalize LDR raw to a coarse  lux -like range
float mapLDRtoLux(int raw) {
  float r = constrain((float)raw, LDR_MIN_RAW, LDR_MAX_RAW);
  float norm = (r - LDR_MIN_RAW) / (LDR_MAX_RAW - LDR_MIN_RAW); // 0..1
  return norm * LDR_MAX_LUX;
}

/******* LoRa *********/
bool loraInit() {
  SPI.begin(LORA_SCK, LORA_MISO, LORA_MOSI, LORA_SS);
  LoRa.setPins(LORA_SS, LORA_RST, LORA_DIO0);
  if (!LoRa.begin(LORA_BAND)) return false;
  LoRa.setSpreadingFactor(LORA_SF);
  LoRa.setSignalBandwidth(LORA_BW);
  LoRa.setCodingRate4(LORA_CR);
  LoRa.setTxPower(LORA_TX_POWER_DBM);
  LoRa.disableCrc(); // we pack our own CRC32
  return true;
}

/******* Compact radio frame ******/
struct __attribute__((packed)) Frame {
  uint8_t  hdr;        //   u8=0xAA
  uint8_t  node;       //   u8
  uint32_t seq;        //   u32
  float    soilPct;    //   f32
  float    tempC;      //   f32
  float    humPct;     //   f32
  float    lux;        //   f32
  uint8_t  dryStreak;  //   u8
  uint32_t crc;        //   f32
};

uint32_t crc32_frame(const Frame &f) {
```

```
  return crc32_le(0, (const uint8_t*)&f, sizeof(Frame) - sizeof(uint32_t
      ));
}

bool loraSendFrame(const Frame &f) {
  LoRa.beginPacket();
  LoRa.write((const uint8_t*)&f, sizeof(Frame));
  int res = LoRa.endPacket(true); // async
  return (res == 1);
}

/****** Sleep *********/
void goToSleep() {
  esp_sleep_enable_timer_wakeup((uint64_t)SLEEP_SECONDS * 1000000ULL);
  esp_deep_sleep_start();
}

/***** Setup ********/
void setup() {
  // Power hygiene
  WiFi.mode(WIFI_OFF);
  btStop();

  rtc_boots++;


  // Init sensors
  dht.begin();

  // ADC channels
  adcInitChannel(CH_SOIL);
  adcInitChannel(CH_LDR);

  // DHT with retry window
  float tempC = NAN, hum = NAN;
  for (int i = 0; i < SAMPLE_RETRIES; ++i) {
    tempC = dht.readTemperature(); // Celsius
    hum   = dht.readHumidity();
    if (!isnan(tempC) && !isnan(hum)) break;
    delay(400);
  }
  if (isnan(tempC)) tempC = -1000.0f; // sentinel
  if (isnan(hum))   hum   = -1000.0f;

  // Analog (median samples)
  int rawSoil = adcReadMedian(CH_SOIL, 9);
  int rawLdr  = adcReadMedian(CH_LDR,  9);

  float soilPct = mapSoilToPercent(rawSoil);
  float lux     = mapLDRtoLux(rawLdr);

  // Update dry streak (approximate at node level)
  if (soilPct < DRY_THRESHOLD_PERCENT) {
    if (rtc_dry_streak < 250) rtc_dry_streak++;
  } else {
    rtc_dry_streak = 0;
```

```
208   }
209
210   // Change detection + periodic kick
211   bool significantChange = approxChanged(soilPct, last_sent_moist,
          SEND_ON_CHANGE_THRESH);
212   bool periodicKick = (rtc_seq % PERIODIC_SEND_EVERY_N) == 0;
213
214   if (significantChange || periodicKick) {
215     if (!loraInit()) {
216       // radio failed; try again next cycle
217       goToSleep();
218       return;
219     }
220
221     rtc_seq++;
222
223     Frame f{};
224     f.hdr        = 0xAA;
225     f.node       = (uint8_t)NODE_ID;
226     f.seq        = rtc_seq;
227     f.soilPct    = soilPct;
228     f.tempC      = tempC;
229     f.humPct     = hum;
230     f.lux        = lux;
231     f.dryStreak  = rtc_dry_streak;
232     f.crc        = 0;
233     f.crc        = crc32_frame(f);
234
235     (void)loraSendFrame(f);
236
237     // update last-sent trackers
238     last_sent_moist = soilPct;
239     last_sent_temp  = tempC;
240     last_sent_hum   = hum;
241     last_sent_lux   = lux;
242   }
243
244   goToSleep();
245 }
246
247 void loop() { /* unused (deep sleep) */ }
```

Listing A.1: ESP32 sensor node firmware (LoRa) using ESP32 ADC driver and event-driven transmission

A.2 Raspberry Pi Oracle Gateway

```
import os, time, struct, zlib, threading, logging, math
from dataclasses import dataclass, field
from typing import Optional, Dict
from collections import deque
from dotenv import load_dotenv
from web3 import Web3

# GPIO
try:
    import RPi.GPIO as GPIO
    GPIO.setmode(GPIO.BCM)
except Exception:
    GPIO = None

# Serial import
try:
    import serial  # pyserial
except Exception:
    serial = None

load_dotenv()

# ---------- ENV CONFIG -------------------
RPC_URL       = os.environ.get("RPC_URL", "https://rpc-amoy.polygon.
    technology")
CONTRACT_ADDR = os.environ.get("CONTRACT_ADDRESS", "0
    x0000000000000000000000000000000000000000")
ACCOUNT_ADDR  = os.environ.get("ACCOUNT_ADDRESS",  "0
    x0000000000000000000000000000000000000000")
PRIVATE_KEY   = os.environ.get("PRIVATE_KEY")  # DO NOT hardcode; use .
    env

SERIAL_PORT   = os.environ.get("SERIAL_PORT")
SERIAL_BAUD   = int(os.environ.get("SERIAL_BAUD", "115200"))

RELAY_PIN_ENV = os.environ.get("RELAY_PIN")
RELAY_PIN     = int(RELAY_PIN_ENV) if RELAY_PIN_ENV else None
ACTUATOR_DEMO = os.environ.get("ACTUATOR_DEMO", "1") == "1"

# Must match ESP32 firmware deep sleep to aggregate dry streak -> days
SLEEP_SECONDS = int(os.environ.get("SLEEP_SECONDS", "300"))

# --------- Smoothing & validation knobs ---------
SMOOTH_WINDOW    = int(os.environ.get("SMOOTH_WINDOW", "3"))
    # sliding window size
SMOOTH_METHOD    = os.environ.get("SMOOTH_METHOD", "median").lower()

MIN_SOIL_PCT     = float(os.environ.get("MIN_SOIL_PCT", "0"))
MAX_SOIL_PCT     = float(os.environ.get("MAX_SOIL_PCT", "100"))
MIN_HUM_PCT      = float(os.environ.get("MIN_HUM_PCT", "0"))
MAX_HUM_PCT      = float(os.environ.get("MAX_HUM_PCT", "100"))
MIN_TEMP_C       = float(os.environ.get("MIN_TEMP_C",  "-20"))
MAX_TEMP_C       = float(os.environ.get("MAX_TEMP_C",  "60"))
```

```
MIN_LUX           = float(os.environ.get("MIN_LUX",       "0"))
MAX_LUX           = float(os.environ.get("MAX_LUX",   "120000"))
    # bright sun ~100k lux

# ------------------- CONTRACT ABI (minimal surface) -------------------
ABI = [
  {"name":"setSoilMoisture","inputs":[{"name":"_soilMoisture","type":"
      uint256"}],"outputs":[],"stateMutability":"nonpayable","type":"
      function"},
  {"name":"setTemperature","inputs":[{"name":"_temperature","type":"
      uint256"}],"outputs":[],"stateMutability":"nonpayable","type":"
      function"},
  {"name":"setHumidity","inputs":[{"name":"_humidity","type":"uint256"
      }],"outputs":[],"stateMutability":"nonpayable","type":"function"},
  {"name":"setLightIntensity","inputs":[{"name":"_lightIntensity","type"
      :"uint256"}],"outputs":[],"stateMutability":"nonpayable","type":"
      function"},
  {"name":"setConsecutiveDryDays","inputs":[{"name":"_consecutiveDryDays
      ","type":"uint256"}],"outputs":[],"stateMutability":"nonpayable","
      type":"function"},
  {"name":"triggerEvaluation","inputs":[],"outputs":[],"stateMutability"
      :"nonpayable","type":"function"},
  {"name":"getIrrigationStatus","inputs":[],"outputs":[{"type":"bool"}],
      "stateMutability":"view","type":"function"},
  {"anonymous":False,"inputs":[{"indexed":False,"name":"status","type":"
      bool"},
                               {"indexed":False,"name":"bit1","type":"
                                   bool"},
                               {"indexed":False,"name":"bit0","type":"
                                   bool"}],
    "name":"IrrigationStatusChanged","type":"event"},
  {"anonymous":False,"inputs":[{"indexed":False,"name":"irrigationStatus
      ","type":"bool"}],
    "name":"EvaluationTriggered","type":"event"},
]

REASON_MAP = {
    (False, False, False): "DEACTIVATE: High temperature",
    (True,  False, False): "ACTIVATE: Low soil + optimal T/H",
    (True,  False, True ): "ACTIVATE: Consecutive dry days",
    (True,  True,  False): "ACTIVATE: Low soil + low light",
    (False, False, True ): "DEACTIVATE: Soil moisture optimal",
}

# ------------------ LOGGING --------------------
logging.basicConfig(level=logging.INFO, format="%(asctime)s %(levelname)
    s %(message)s")

# ----------------- LORA FRAME --------------------
FRAME_LEN = 27  # bytes

@dataclass
class Reading:
    node: int; seq: int
    soilPct: float; tempC: float; humPct: float; lux: float
    dryStreak: int; when: float
```

```
88
89  def decode_frame(buf: bytes) -> Optional[Reading]:
90      """Decode 27-byte frame: 0xAA | node:u8 | seq:u32 | soil:f32 | temp:
        f32 | hum:f32 | lux:f32 | dry:u8 | crc32:u32 (LE)"""
91      if len(buf) != FRAME_LEN or buf[0] != 0xAA:
92          return None
93      head = buf[:-4]
94      crc_rx  = struct.unpack("<I", buf[-4:])[0]
95      crc_calc = zlib.crc32(head) & 0xFFFFFFFF
96      if crc_rx != crc_calc:
97          logging.warning("CRC mismatch: rx=%08x calc=%08x", crc_rx,
            crc_calc); return None
98      node = buf[1]
99      seq  = struct.unpack("<I", buf[2:6])[0]
100     soil, tempC, hum, lux = struct.unpack("<ffff", buf[6:22])
101     dryStreak = buf[22]
102     # Sentinel handling (DHT/LDR failures, etc.)
103     if tempC < -999.0: tempC = float("nan")
104     if hum   < -999.0: hum   = float("nan")
105     # (Optionally add other sentinels here if your firmware uses them)
106     return Reading(node, seq, soil, tempC, hum, lux, dryStreak, time.
        time())
107
108 # -------------- PER-NODE STATE -------------------
109 @dataclass
110 class NodeState:
111     last_valid: Dict[str, float] = field(default_factory=dict)     #
        last good (post-smoothing) numeric value per field
112     last_seen: float = 0.0
113     buffers: Dict[str, deque] = field(default_factory=lambda: {
114         "soilPct": deque(maxlen=SMOOTH_WINDOW),
115         "humPct":  deque(maxlen=SMOOTH_WINDOW),
116         "tempC":   deque(maxlen=SMOOTH_WINDOW),
117         "lux":     deque(maxlen=SMOOTH_WINDOW),
118     })
119     last_onchain: Dict[str, Optional[int]] = field(default_factory=
        lambda: {
120         "soilMoisture": None,
121         "temperature": None,
122         "humidity": None,
123         "lightIntensity": None,
124         "consecutiveDryDays": None,
125     })
126
127 nodes: Dict[int, NodeState] = {}
128
129 def node_state(nid: int) -> NodeState:
130     st = nodes.get(nid)
131     if st is None:
132         st = NodeState()
133         nodes[nid] = st
134     st.last_seen = time.time()
135     return st
136
137 # --------------- PACKET SOURCE -------------------
138 class SerialPacketSource:
```

```
139     def __init__(self, port: str, baud: int):
140         if serial is None:
141             raise RuntimeError("pyserial not installed; set up your LoRa
                    driver or install pyserial.")
142         self.ser = serial.Serial(port, baud, timeout=1)
143     def read(self) -> Optional[bytes]:
144         data = self.ser.read(FRAME_LEN)
145         return data if len(data) == FRAME_LEN else None
146 
147 class DummyPacketSource:
148     def read(self) -> Optional[bytes]:
149         time.sleep(2.0)
150         return None
151 
152 def pick_packet_source():
153     if SERIAL_PORT:
154         logging.info("LoRa serial source: %s @ %d bps", SERIAL_PORT,
               SERIAL_BAUD)
155         return SerialPacketSource(SERIAL_PORT, SERIAL_BAUD)
156     logging.warning("No SERIAL_PORT provided; using Dummy source.")
157     return DummyPacketSource()
158 
159 # ------------------- WEB3 / CONTRACT -------------------
160 if not (RPC_URL and CONTRACT_ADDR and ACCOUNT_ADDR and PRIVATE_KEY):
161     logging.error("Missing RPC_URL / CONTRACT_ADDRESS / ACCOUNT_ADDRESS
           / PRIVATE_KEY in environment.")
162 
163 w3 = Web3(Web3.HTTPProvider(RPC_URL))
164 if not w3.is_connected():
165     logging.error("Cannot connect to RPC at %s", RPC_URL)
166 
167 contract = w3.eth.contract(address=Web3.to_checksum_address(
        CONTRACT_ADDR), abi=ABI)
168 
169 def send_tx(fn, *args):
170     nonce = w3.eth.get_transaction_count(Web3.to_checksum_address(
           ACCOUNT_ADDR))
171     tx = fn(*args).build_transaction({
172         "from": Web3.to_checksum_address(ACCOUNT_ADDR),
173         "nonce": nonce,
174         "chainId": w3.eth.chain_id,
175         "maxFeePerGas": w3.eth.gas_price * 2,
176         "maxPriorityFeePerGas": w3.to_wei("2", "gwei"),
177     })
178     signed = w3.eth.account.sign_transaction(tx, private_key=PRIVATE_KEY
           )
179     tx_hash = w3.eth.send_raw_transaction(signed.rawTransaction)
180     logging.info("tx sent: %s", tx_hash.hex())
181     rcpt = w3.eth.wait_for_transaction_receipt(tx_hash)
182     logging.info("tx mined: block=%s status=%s", rcpt.blockNumber, rcpt.
           status)
183     return rcpt
184 
185 # ----------------- GPIO ACTUATION -------------------
186 def setup_relay():
187     if GPIO is None or RELAY_PIN is None:
```

```
188         logging.info("Actuator: GPIO disabled (missing RPi.GPIO or
                RELAY_PIN).")
189         return
190     GPIO.setup(RELAY_PIN, GPIO.OUT, initial=GPIO.LOW)
191     logging.info("Actuator: demo output on BCM %d (LOW=OFF, HIGH=ON)",
            RELAY_PIN)
192
193 def set_relay(on: bool):
194     if GPIO is not None and RELAY_PIN is not None:
195         GPIO.output(RELAY_PIN, GPIO.HIGH if on else GPIO.LOW)
196     state = "ON" if on else "OFF"
197     if ACTUATOR_DEMO:
198         logging.info("Actuator(DEMO): GPIO=%s (represents relay drive)",
                state)
199     else:
200         logging.info("Actuator: GPIO=%s", state)
201
202 # ------------------- EVENT LISTENER -----------------
203 def start_event_listener(stop_evt: threading.Event):
204     try:
205         filt = contract.events.IrrigationStatusChanged.create_filter(
                fromBlock="latest")
206         logging.info("Event listener started.")
207         while not stop_evt.is_set():
208             for ev in filt.get_new_entries():
209                 status = ev["args"]["status"]; bit1 = ev["args"]["bit1"
                        ]; bit0 = ev["args"]["bit0"]
210                 reason = REASON_MAP.get((status, bit1, bit0), f"status={
                        status},bit1={bit1},bit0={bit0}")
211                 logging.info("IrrigationStatusChanged -> %s", reason)
212                 set_relay(bool(status))
213             time.sleep(2.0)
214     except Exception as e:
215         logging.exception("Event listener error: %s", e)
216
217 # ------------ VALIDATION / SMOOTHING HELPERS -------------------
218 def in_physical_range(field: str, v: float) -> bool:
219     if v != v:  # NaN
220         return False
221     if field == "soilPct":
222         return MIN_SOIL_PCT <= v <= MAX_SOIL_PCT
223     if field == "humPct":
224         return MIN_HUM_PCT  <= v <= MAX_HUM_PCT
225     if field == "tempC":
226         return MIN_TEMP_C   <= v <= MAX_TEMP_C
227     if field == "lux":
228         return MIN_LUX      <= v <= MAX_LUX
229     return True
230
231 def smooth_sequence(seq):
232     if not seq:
233         return None
234     if SMOOTH_METHOD == "mean":
235         return sum(seq) / len(seq)
236     # default robust median
237     s = sorted(seq)
```

```
    n = len(s)
    mid = n // 2
    return (s[mid] if n % 2 == 1 else (s[mid - 1] + s[mid]) / 2.0)

def update_smooth_impute(st: NodeState, field: str, v: Optional[float])
    -> Optional[float]:
    """
    If v is NaN/None or out-of-range then it tries to impute with
        last_valid[field]
    Else then add to buffer and return smoothed value
    """
    if v is None or (isinstance(v, float) and v != v) or not
        in_physical_range(field, v):
        if v is not None and not (isinstance(v, float) and v != v) and
            not in_physical_range(field, v):
            logging.warning("Discard out-of-range %s=%.3f (kept
                last_valid if present)", field, v)
        return st.last_valid.get(field)

    # valid value -> update smoothing buffer
    dq = st.buffers[field]
    dq.append(float(v))
    smoothed = smooth_sequence(list(dq))
    if smoothed is not None:
        st.last_valid[field] = smoothed
    return smoothed

def round_u(v: Optional[float]) -> Optional[int]:
    if v is None or (isinstance(v, float) and v != v):
        return None
    # clamp at 0 to avoid negative ints
    return max(0, int(round(v)))

def cycles_to_days(cycles: int) -> int:
    if SLEEP_SECONDS <= 0:
        return 0
    per_day = int((24 * 3600) / SLEEP_SECONDS)
    return cycles // max(1, per_day)

# ----------------- GATEWAY CORE -------------------
def push_updates_and_evaluate(node_id: int, soil: Optional[float], temp:
    Optional[float],
                                hum: Optional[float], lux: Optional[float
                                    ], days: int):
    st = node_state(node_id)

    soil_u = round_u(soil)
    temp_u = round_u(temp)
    hum_u  = round_u(hum)
    lux_u  = round_u(lux)

    updates = []
    if soil_u is not None and soil_u != st.last_onchain["soilMoisture"]:
        updates.append(("setSoilMoisture", soil_u)); st.last_onchain["
            soilMoisture"] = soil_u
    if temp_u is not None and temp_u != st.last_onchain["temperature"]:
```

```
286             updates.append(("setTemperature", temp_u));  st.last_onchain["
                    temperature"]  = temp_u
287         if hum_u is not None and hum_u != st.last_onchain["humidity"]:
288             updates.append(("setHumidity", hum_u));      st.last_onchain["
                    humidity"]     = hum_u
289         if lux_u is not None and lux_u != st.last_onchain["lightIntensity"]:
290             updates.append(("setLightIntensity", lux_u)); st.last_onchain["
                    lightIntensity"] = lux_u
291         if days != st.last_onchain["consecutiveDryDays"]:
292             updates.append(("setConsecutiveDryDays", days)); st.last_onchain
                    ["consecutiveDryDays"] = days
293 
294         if not updates:
295             logging.info("No changes to push (node=%d)", node_id)
296             return
297 
298         for fname, val in updates:
299             logging.info("Calling %s(%s) [node=%d]", fname, val, node_id)
300             fn = getattr(contract.functions, fname)
301             send_tx(fn, val)
302 
303         logging.info("Calling triggerEvaluation() [node=%d]", node_id)
304         send_tx(contract.functions.triggerEvaluation)
305 
306 def process_and_push(r: Reading):
307     st = node_state(r.node)
308 
309     # --- Per-field smoothing + sentinel handling + threshold validation
                + imputation
310     soil_sm = update_smooth_impute(st, "soilPct", r.soilPct)
311     temp_sm = update_smooth_impute(st, "tempC",  r.tempC)
312     hum_sm  = update_smooth_impute(st, "humPct", r.humPct)
313     lux_sm  = update_smooth_impute(st, "lux",    r.lux)
314 
315     days = cycles_to_days(r.dryStreak)
316 
317     logging.info(
318         "Processed node=%d seq=%d soil=%.2f->%.2f temp=%.2f->%.2f hum
                =%.2f->%.2f lux=%.0f->%.0f dry=%d->%d days",
319         r.node, r.seq,
320         r.soilPct, (soil_sm if soil_sm is not None else float('nan')),
321         r.tempC,  (temp_sm if temp_sm is not None else float('nan')),
322         r.humPct, (hum_sm  if hum_sm  is not None else float('nan')),
323         r.lux,    (lux_sm  if lux_sm  is not None else float('nan')),
324         r.dryStreak, days
325     )
326 
327     # Push only what we have (None means: no update so keep last on-
            chain)
328     push_updates_and_evaluate(r.node, soil_sm, temp_sm, hum_sm, lux_sm,
            days)
329 
330 # ---------------- MAIN -------------------
331 def main():
332     logging.info("RPC=%s chainId=%s", RPC_URL, (w3.eth.chain_id if w3.
            is_connected() else "n/a"))
```

```
    logging.info("Smoothing: method=%s window=%d", SMOOTH_METHOD,
        SMOOTH_WINDOW)
    logging.info("Physical ranges: soil=[%s,%s] hum=[%s,%s] temp=[%s,%s]
        lux=[%s,%s]",
                 MIN_SOIL_PCT, MAX_SOIL_PCT, MIN_HUM_PCT, MAX_HUM_PCT,
                 MIN_TEMP_C, MAX_TEMP_C, MIN_LUX, MAX_LUX)

    setup_relay()

    stop_evt = threading.Event()
    t = threading.Thread(target=start_event_listener, args=(stop_evt,),
        daemon=True)
    t.start()

    source = pick_packet_source()
    try:
        while True:
            buf = source.read()
            if not buf:
                continue
            r = decode_frame(buf)
            if r:
                logging.info("RX node=%d seq=%d soil=%.1f temp=%.1f hum
                    =%.1f lux=%.0f dry=%d",
                             r.node, r.seq, r.soilPct, r.tempC, r.humPct
                                 , r.lux, r.dryStreak)
                process_and_push(r)
            else:
                logging.warning("Invalid frame received.")
    except KeyboardInterrupt:
        pass
    finally:
        stop_evt.set()
        if GPIO:
            try:
                GPIO.cleanup()
            except Exception:
                pass

if __name__ == "__main__":
    main()
```

Listing A.2: RPi4 gateway that decodes LoRa frames and updates `SmartIrrigationV2` on Polygon Amoy

```
RPC_URL=https://rpc-amoy.polygon.technology
CONTRACT_ADDRESS=0x6f0BaDb6BA1BE024D45f185cbc54f29b2C09b205
ACCOUNT_ADDRESS=0x418F5F9398bB8ce39E01C51F3c74DE291384cA02
PRIVATE_KEY=0x00000   # Not shared for security reasons

# LoRa serial bridge
SERIAL_PORT=/dev/ttyS0
SERIAL_BAUD=115200

# Actuator demo output (GPIO)
RELAY_PIN=17
```

```
12 ACTUATOR_DEMO=1
13
14 # ESP32 deep-sleep period dry-streak cycles -> days
15 SLEEP_SECONDS=300
```

Listing A.3: Environment configuration template for the RPi gateway

A.3 Smart Contract - Solidity

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

contract SmartIrrigationV2 {
    address public owner;

    // Sensor variables
    uint256 public soilMoisture; // percentage
    uint256 public temperature; // Celsius
    uint256 public humidity; // percentage
    uint256 public lightIntensity; // lux
    uint256 public consecutiveDryDays; // Count consecutive days soil
       moisture below threshold

    // Irrigation status
    bool public irrigationStatus; // true - ON, false - OFF

    // Thresholds
    uint256 public tempThresholdHigh = 35;
    uint256 public tempOptimalLow = 20;
    uint256 public tempOptimalHigh = 28;
    uint256 public soilMoistureLow = 30;
    uint256 public soilMoistureOptimal = 60;
    uint256 public lightIntensityLow = 300;
    uint256 public humidityHigh = 60;
    uint256 public dryDayThreshold = 3;

    // Flags to track updated parameters
    bool public isSoilMoistureUpdated;
    bool public isTemperatureUpdated;
    bool public isHumidityUpdated;
    bool public isLightIntensityUpdated;

    // Events
    event IrrigationStatusChanged(bool status, bool bit1, bool bit0);
    event SoilMoistureUpdated(uint256 newValue);
    event TemperatureUpdated(uint256 newValue);
    event HumidityUpdated(uint256 newValue);
    event LightIntensityUpdated(uint256 newValue);
    event EvaluationTriggered(bool irrigationStatus);

    // Modifier
    modifier onlyOwner() {
        require(msg.sender == owner, "Not authorized");
        _;
    }

    constructor() {
        owner = msg.sender;
        irrigationStatus = false;
    }

    // --- Sensor update functions ---
```

```
53      function setSoilMoisture(uint256 _soilMoisture) public onlyOwner {
54          if (soilMoisture != _soilMoisture) {
55              soilMoisture = _soilMoisture;
56              isSoilMoistureUpdated = true;
57              emit SoilMoistureUpdated(_soilMoisture);
58          }
59      }
60
61      function setTemperature(uint256 _temperature) public onlyOwner {
62          if (temperature != _temperature) {
63              temperature = _temperature;
64              isTemperatureUpdated = true;
65              emit TemperatureUpdated(_temperature);
66          }
67      }
68
69      function setHumidity(uint256 _humidity) public onlyOwner {
70          if (humidity != _humidity) {
71              humidity = _humidity;
72              isHumidityUpdated = true;
73              emit HumidityUpdated(_humidity);
74          }
75      }
76
77      function setLightIntensity(uint256 _lightIntensity) public onlyOwner
            {
78          if (lightIntensity != _lightIntensity) {
79              lightIntensity = _lightIntensity;
80              isLightIntensityUpdated = true;
81              emit LightIntensityUpdated(_lightIntensity);
82          }
83      }
84
85      function setConsecutiveDryDays(uint256 _consecutiveDryDays) public
           onlyOwner {
86          if (consecutiveDryDays != _consecutiveDryDays) {
87              consecutiveDryDays = _consecutiveDryDays;
88          }
89      }
90
91      // --- Evaluation trigger ---
92      function triggerEvaluation() public onlyOwner {
93          require(
94              isSoilMoistureUpdated ||
95              isTemperatureUpdated ||
96              isHumidityUpdated ||
97              isLightIntensityUpdated,
98              "At least one sensor must be updated first"
99          );
100
101         // Reset flags and evaluate
102         isSoilMoistureUpdated = false;
103         isTemperatureUpdated = false;
104         isHumidityUpdated = false;
105         isLightIntensityUpdated = false;
106
```

```
        evaluateIrrigation();
        emit EvaluationTriggered(irrigationStatus);
    }

    // --- Irrigation logic ---
    function evaluateIrrigation() internal {
        if (temperature > tempThresholdHigh) {
            irrigationStatus = false;
            deactivateIrrigation(false, false);
            return;
        }

        if (
            soilMoisture < soilMoistureLow &&
            temperature >= tempOptimalLow &&
            temperature <= tempOptimalHigh &&
            humidity < humidityHigh
        ) {
            irrigationStatus = true;
            activateIrrigation(false, false);
            return;
        }

        if (consecutiveDryDays >= dryDayThreshold) {
            irrigationStatus = true;
            activateIrrigation(false, true);
            return;
        }

        if (soilMoisture < soilMoistureLow && lightIntensity <
            lightIntensityLow) {
            irrigationStatus = true;
            activateIrrigation(true, false);
            return;
        }

        if (soilMoisture >= soilMoistureOptimal) {
            irrigationStatus = false;
            deactivateIrrigation(false, true);
            return;
        }
    }

    function activateIrrigation(bool bit1, bool bit0) internal {
        if (!irrigationStatus) {
            irrigationStatus = true;
            emit IrrigationStatusChanged(true, bit1, bit0);
        }
    }

    function deactivateIrrigation(bool bit1, bool bit0) internal {
        if (irrigationStatus) {
            irrigationStatus = false;
            emit IrrigationStatusChanged(false, bit1, bit0);
        }
    }
```

```
162
163     function getIrrigationStatus() public view returns (bool) {
164         return irrigationStatus;
165     }
166 }
```

Listing A.4: Full Solidity implementation of SmartIrrigationV2

B Build, ABI, and Scenario Artifacts

B.1 SmartIrrigationV2_metadata.json (Compiler settings and source integrity)

```
{
  "settings": {
    "compilationTarget": { "IrrigationSystemV2.sol": "
       SmartIrrigationV2" },
    "evmVersion": "cancun",
    "optimizer": { "enabled": false, "runs": 200 },
    "metadata": { "bytecodeHash": "ipfs" }
  },
  "sources": {
    "IrrigationSystemV2.sol": {
      "keccak256": "0x06dfccb0...beb8737",
      "urls": [
        "bzz-raw://586b41...f41c5d",
        "dweb:/ipfs/QmXLz6JeH5J3pDFk...kjHe"
      ]
    }
  }
}
```

B.2 ABI slice (selected functions) from SmartIrrigationV2_metadata.json

```
[
  {
    "name": "temperature",
    "inputs": [],
    "outputs": [{"type":"uint256"}],
    "stateMutability":"view",
    "type":"function"
    },
  {
    "name": "tempOptimalLow",
```

```
    "inputs": [],
    "outputs": [{"type":"uint256"}],
    "stateMutability":"view",
    "type":"function"
    },
  {
    "name": "tempOptimalHigh", "
    inputs": [],
    "outputs": [{"type":"uint256"}],
    "stateMutability":"view",
    "type":"function"
    },
  {
    "name": "tempThresholdHigh",
    "inputs": [],
    "outputs": [{"type":"uint256"}],
    "stateMutability":"view",
    "type":"function"
    },
  {
    "name": "triggerEvaluation",
    "inputs": [],
    "outputs": [],
    "stateMutability":"nonpayable",
    "type":"function"
    }
]
```

B.3 Minimal ABI for gateway binding (from `SmartIrrigationV2.json`)

```
[
  {
    "name":"setSoilMoisture",
    "inputs":[{
        "name":"_soilMoisture",
        "type":"uint256"
        }],
    "stateMutability":"nonpayable",
    "type":"function"
    },
  {
    "name":"setTemperature",
    "inputs":[{
        "name":"_temperature",
        "type":"uint256"
        }],
```

```
    "stateMutability":"nonpayable",
    "type":"function"
    },
  {
    "name":"setHumidity",
    "inputs":[{
        "name":"_humidity",
        "type":"uint256"
        }],
    "stateMutability":"nonpayable",
    "type":"function"
    },
  {
    "name":"setLightIntensity",
    "inputs":[{
        "name":"_lightIntensity",
        "type":"uint256"
        }],
    "stateMutability":"nonpayable",
    "type":"function"
    },
  {
    "name":"triggerEvaluation",
    "inputs":[],
    "stateMutability":"nonpayable",
    "type":"function"
    },
  {
    "name":"getIrrigationStatus",
    "inputs":[],
    "outputs":[{"type":"bool"}],
    "stateMutability":"view",
    "type":"function"
    }
]
```

B.4 scenario.json (account and constructor transactions)

```
{
  "accounts": { "account{0}": "0
     x418F5F9398bB8ce39E01C51F3c74DE291384cA02" },
  "transactions": [{
    "timestamp": 1732230259938,
    "record": {
      "type": "constructor",
      "name": "",
```

```
        "inputs": "()",
        "contractName": "SmartIrrigationV2",
        "bytecode": "60806040...<truncated>...300081a0033"
      }
   }]
}
```

C Simulation

C.1 RPi4 Blockchain Simulator

```
import os, time, json
from dotenv import load_dotenv
from web3 import Web3

# ---------- Config & Setup ----------
load_dotenv()  # expects RPC_URL, PRIVATE_KEY, CONTRACT_ADDRESS
RPC_URL          = os.getenv("RPC_URL")
PRIVATE_KEY      = os.getenv("PRIVATE_KEY")
CONTRACT_ADDRESS = os.getenv("CONTRACT_ADDRESS")

def INFO(msg: str): print(f"INFO {msg}")

w3 = Web3(Web3.HTTPProvider(RPC_URL))
if not w3.is_connected():
    raise RuntimeError("Web3 not connected. Check RPC_URL")

acct = w3.eth.account.from_key(PRIVATE_KEY)
ACCOUNT_ADDR = acct.address
CONTRACT     = Web3.to_checksum_address(CONTRACT_ADDRESS)
CHAIN_ID     = w3.eth.chain_id

# Minimal ABI for setters, evaluation and events (adapt names if
   needed)
ABI = json.loads(r"""
[
  {"inputs":[{"internalType":"uint8","name":"v","type":"uint8"}],
   "name":"setSoilMoisture","outputs":[],"stateMutability":"
      nonpayable","type":"function"},
  {"inputs":[{"internalType":"int16","name":"v","type":"int16"}],
   "name":"setTemperature","outputs":[],"stateMutability":"
      nonpayable","type":"function"},
  {"inputs":[{"internalType":"uint8","name":"v","type":"uint8"}],
   "name":"setHumidity","outputs":[],"stateMutability":"nonpayable
      ","type":"function"},
  {"inputs":[{"internalType":"uint32","name":"v","type":"uint32"}],
```

```
32      "name":"setLightIntensity","outputs":[],"stateMutability":"
            nonpayable","type":"function"},
33    {"inputs":[],"name":"triggerEvaluation","outputs":[],"
          stateMutability":"nonpayable","type":"function"},
34    {"anonymous":false,"inputs":[{"indexed":false,"internalType":"
          address","name":"by","type":"address"}],
35      "name":"EvaluationTriggered","type":"event"},
36    {"anonymous":false,"inputs":[{"indexed":false,"internalType":"
          bool","name":"active","type":"bool"},
37                                 {"indexed":false,"internalType":"
                                       uint8","name":"reason","type":"
                                       uint8"}],
38      "name":"IrrigationStatusChanged","type":"event"}
39  ]
40  """)
41
42  contract = w3.eth.contract(address=CONTRACT, abi=ABI)
43
44  # ------- Helpers ----------
45  def _send_tx(fn, label: str):
46      nonce = w3.eth.get_transaction_count(ACCOUNT_ADDR)
47      tx = fn.build_transaction({
48          "from": ACCOUNT_ADDR,
49          "nonce": nonce,
50          "gas": 200_000,
51          "gasPrice": w3.to_wei(30, "gwei")    # simple legacy
                pricing for testnets
52      })
53      signed = w3.eth.account.sign_transaction(tx, PRIVATE_KEY)
54      tx_hash = w3.eth.send_raw_transaction(signed.rawTransaction)
55      INFO(f"{label} tx sent: {tx_hash.hex()}")
56      rcpt = w3.eth.wait_for_transaction_receipt(tx_hash)
57      INFO(f"{label} tx mined: block={rcpt.blockNumber} status={rcpt.
            status} gasUsed={rcpt.gasUsed}")
58      # Try to decode relevant events from this receipt
59      try:
60          for e in contract.events.EvaluationTriggered().
                process_receipt(rcpt):
61              INFO(f"EvaluationTriggered by={e['args']['by']}")
62          for e in contract.events.IrrigationStatusChanged().
                process_receipt(rcpt):
63              INFO(f"IrrigationStatusChanged -> active={e['args']['
                    active']} reason={e['args']['reason']}")
64      except Exception:
65          pass
66      return rcpt
67
68  def set_soil(v):  INFO(f"Calling setSoilMoisture({v})");
        return _send_tx(contract.functions.setSoilMoisture(int(v)), "
```

```
    setSoilMoisture")
69  def set_temp(v):  INFO(f"Calling setTemperature({v})");
    return _send_tx(contract.functions.setTemperature(int(v)), "
    setTemperature")
70  def set_hum(v):   INFO(f"Calling setHumidity({v})");
    return _send_tx(contract.functions.setHumidity(int(v)), "
    setHumidity")
71  def set_lux(v):   INFO(f"Calling setLightIntensity({v})");
    return _send_tx(contract.functions.setLightIntensity(int(v)), "
    setLightIntensity")
72  def evaluate():   INFO("Calling triggerEvaluation()");
    return _send_tx(contract.functions.triggerEvaluation(), "
    triggerEvaluation")
73
74  # ---------- Simulator ----------
75  NODE_ID = 1
76  SEQ     = 0
77
78  def banner():
79      INFO(f"RPi4 Blockchain Simulator start | chain_id={CHAIN_ID}
           account={ACCOUNT_ADDR}")
80      INFO(f"contract={CONTRACT} | node={NODE_ID}")
81      INFO("Mode=SIMULATED (no radio). Values are injected directly
           before evaluation.")
82
83  def rx_line(soil, temp, hum, lux):
84      global SEQ
85      # Log line similar to OG gateway "RX/Processed" (no smoothing
           here)
86      INFO(f"RX node={NODE_ID} seq={SEQ} soil={soil:.2f} temp={temp
           :.2f} hum={hum:.2f} lux={lux}")
87      INFO(f"Processed node={NODE_ID} seq={SEQ} soil={soil:.2f}->%0.2
           f temp={temp:.2f}->%0.2f hum={hum:.2f}->%0.2f lux={lux:.0f
           }->%0.0f" % (soil, temp, hum, lux, lux))
88      SEQ = (SEQ + 1) & 0xFF
89
90  def scenario_normal():
91      INFO("=== Scenario: NORMAL (soil drops, irrigation activates;
           then recovers, deactivates) ===")
92      # steady environment
93      base_temp, base_hum, base_lux = 25, 50, 12000
94      set_temp(base_temp); set_hum(base_hum); set_lux(base_lux)
95
96      # soil decreases across activation threshold
97      for soil in [45, 41, 37, 33, 29]:
98          rx_line(soil, base_temp, base_hum, base_lux)
99          set_soil(soil)
100         evaluate()
101         time.sleep(0.8)
```

```
102
103     # simulate irrigation effect: soil rises above 60% (
            deactivation)
104     for soil in [52, 58, 61]:
105         rx_line(soil, base_temp, base_hum, base_lux)
106         set_soil(soil)
107         evaluate()
108         time.sleep(0.8)
109
110 def scenario_edge_high_temp():
111     INFO("=== Scenario: EDGE (high temperature safety override) ===
            ")
112     soil, hum, lux = 30, 50, 15000
113     set_soil(soil); set_hum(hum); set_lux(lux)
114     for t in [36, 37, 38]:
115         rx_line(soil, t, hum, lux)
116         set_temp(t)
117         evaluate()
118         time.sleep(0.8)
119
120 if __name__ == "__main__":
121     banner()
122     scenario_normal()
123     scenario_edge_high_temp()
124     INFO("Simulation complete.")
```

C.2 Simulation Logs

```
1  INFO RPi4 Blockchain Simulator start | chain_id=80002 account=0xAbc
       ...123
2  INFO contract=0xDeF...789 | node=1
3  INFO Mode=SIMULATED (no radio). Values are injected directly before
       evaluation.
4
5  INFO === Scenario: NORMAL (soil drops, irrigation activates; then
       recovers) ===
6  INFO RX node=1 seq=0 soil=45.00 temp=25.00 hum=50.00 lux=12000
7  INFO Calling setSoilMoisture(45)
8  INFO setSoilMoisture tx sent: 0x7c...3f1
9  INFO setSoilMoisture tx mined: block=135900 status=1 gasUsed=46241
10 INFO Calling triggerEvaluation()
11 INFO triggerEvaluation tx sent: 0xa1...9bd
12 INFO triggerEvaluation tx mined: block=135901 status=1 gasUsed
       =48712
13 INFO EvaluationTriggered by=0xAbc...123
14
15 INFO RX node=1 seq=4 soil=29.00 temp=25.00 hum=50.00 lux=12000
```

```
INFO Calling setSoilMoisture(29)
INFO setSoilMoisture tx sent: 0x55...e4a
INFO setSoilMoisture tx mined: block=135905 status=1 gasUsed=46241
INFO Calling triggerEvaluation()
INFO triggerEvaluation tx sent: 0xb3...91c
INFO triggerEvaluation tx mined: block=135906 status=1 gasUsed
   =48977
INFO EvaluationTriggered by=0xAbc...123
INFO IrrigationStatusChanged -> active=True reason=1

INFO RX node=1 seq=7 soil=61.00 temp=25.00 hum=50.00 lux=12000
INFO Calling setSoilMoisture(61)
INFO setSoilMoisture tx sent: 0x6d...a7e
INFO setSoilMoisture tx mined: block=135912 status=1 gasUsed=46241
INFO Calling triggerEvaluation()
INFO triggerEvaluation tx sent: 0x90...c22
INFO triggerEvaluation tx mined: block=135913 status=1 gasUsed
   =48980
INFO EvaluationTriggered by=0xAbc...123
INFO IrrigationStatusChanged -> active=False reason=3

INFO === Scenario: EDGE (high temperature safety override) ===
INFO RX node=1 seq=8 soil=30.00 temp=36.00 hum=50.00 lux=15000
INFO Calling setTemperature(36)
INFO setTemperature tx sent: 0x44...b59
INFO setTemperature tx mined: block=135920 status=1 gasUsed=45112
INFO Calling triggerEvaluation()
INFO triggerEvaluation tx sent: 0x2a...e10
INFO triggerEvaluation tx mined: block=135921 status=1 gasUsed
   =49011
INFO EvaluationTriggered by=0xAbc...123
INFO IrrigationStatusChanged -> active=False reason=2
```

Bibliography

[1] Food and Agriculture Organization. The state of food and agriculture 2023: Revealing water scarcity. *FAO Publications*, 2023.

[2] Peter H. Gleick. Water and climate: New challenges and opportunities. *Environmental Science Policy*, 129:1–7, 2022.

[3] Khaled Salah, Vivek Sharma, and Ala Al-Fuqaha. Smart agriculture assurance: Iot and blockchain for trusted sustainable produce. *Computer Standards Interfaces*, 86:103707, 2023.

[4] Harald Sundmaeker, Cor Verdouw, Sjaak Wolfert, and Lucia Perez Freire. Internet of food and farm 2020. *Digitising the Industry*, pages 129–150, 2016.

[5] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, 2013.

[6] LoRa Alliance. Lorawan specification v1.0.4. *LoRa Alliance Technical Documentation*, 2024.

[7] Konstantinos Christidis and Michael Devetsikiotis. Blockchains and smart contracts for the internet of things. *IEEE Access*, 4:2292–2303, 2016.

[8] Polygon Labs. Polygon technology overview: A scalable layer 2 for ethereum, 2024.

[9] Bader Al-Bassam and Stephan Reiff-Marganiec. Blockchain layer-1 and layer-2 benchmarking for iot and smart contracts. *IEEE Internet of Things Journal*, 10(7):5893–5908, 2023.

[10] David Friedman, Alex Hunter, and Jada Wilson. Blockchain applications in precision agriculture: A review of opportunities and challenges. *Journal of Agricultural Informatics*, 13(2):45–60, 2022.

[11] Provable. Provable things: Documentation and use cases. *https://docs.provable.xyz*, 2024.

[12] Youngsoo Kim and Robert Evans. Design and implementation of an iot-based smart irrigation system. *International Journal of Smart Agriculture*, 5(3):101–110, 2019.

[13] P. Gupta and R. Singh. Smart irrigation system using arduino and gsm. *Journal of Electronics and Communication Engineering*, 9(2):65–70, 2020.

[14] Min Yuan and Zheng Liu. Iot-driven precision agriculture: Smart irrigation system with real-time monitoring. *Agricultural Informatics*, 12(1):33–44, 2021.

[15] P. P. Jayaraman, A. Yavari, D. Georgakopoulos, A. Morshed, and A. Zaslavsky. Internet of things platform for smart farming: Experiences and lessons learnt. *Sensors*, 16(11):1884, 2016.

[16] Jun Xie and Qiang Liu. Blockchain-enabled greenhouse management for organic agriculture. *Sustainable Computing: Informatics and Systems*, 35:100741, 2022.

[17] Michela Tripoli and Josef Schmidhuber. Emerging opportunities for the application of blockchain in the agri-food industry, 2018. FAO and ICTSD Joint Report.

[18] Andreas Kamilaris, Antoni Fonts, and Francesc X. Prenafeta-Boldú. The rise of blockchain technology in agriculture and food supply chains. *Trends in Food Science & Technology*, 91:640–652, 2019.

[19] Jiayi Lin, Chuan Wang, Qian Zhang, Li Li, and Xiaohui Liang. Blockchain and iot-based food traceability for smart agriculture. *ACM Transactions on Internet Technology*, 21(2):1–21, 2020.

[20] S. Kumar and P. Sharma. Smart agriculture with iot and blockchain: Automation and data integrity. *IEEE Access*, 9:102950–102961, 2021.

[21] Adrián Sánchez-Mompó, Hugo Barbier, Wei-Jun Yi, and Jafar Saniie. Internet of things smart farming architecture for agricultural automation. *ECASP Research Laboratory, Illinois Institute of Technology*, 2021.

[22] A.K.M. Bahalul Haque, Tanvir Hasan Pranto, and Md. Rahman. Blockchain and smart contract for iot-enabled smart agriculture. *PeerJ Computer Science*, 7:e407, 2021.

[23] Y. Huang, F. Gao, and S. Zhang. Blockchain-empowered iot for smart irrigation systems: Opportunities and challenges. *Journal of Cleaner Production*, 359:132027, 2022.

[24] Sachin Kamble, Angappa Gunasekaran, and N.C. Dhone. A conceptual framework for iot-based monitoring and smart decisions for precision agriculture using blockchain and oracles. *Computers and Electronics in Agriculture*, 178:105476, 2020.

[25] Z. Xu and F. Zhang. Precision agriculture: Iot-enabled lora systems for smart farming. *Applied Sciences*, 13(4):2251, 2023.

[26] Martin Haefke, Stefan Fischer, and Michael Kauer. A wireless sensor network for precision agriculture and its performance. *Sensors & Transducers*, 163(1):68–73, 2014.

[27] Deepak Gupta, Satyendra Yadav, Fadi Al-Turjman, Chi-Hua Hsu, et al. Climate-smart agriculture using intelligent techniques, blockchain and internet of things: Concepts, challenges, and opportunities. *Computers and Electrical Engineering*, 100:107971, 2022.

[28] Richard G. Allen, Luis S. Pereira, Dirk Raes, and Martin Smith. *Crop Evapotranspiration – Guidelines for Computing Crop Water Requirements*. FAO Irrigation and Drainage Paper 56. Food and Agriculture Organization of the United Nations (FAO), Rome, 1998.

[29] J. Doorenbos and A. H. Kassam. *Yield Response to Water*. FAO Irrigation and Drainage Paper 33. Food and Agriculture Organization of the United Nations (FAO), Rome, 1986.

[30] Hamlyn G. Jones. *Irrigation Scheduling: Advantages and Pitfalls of Plant-Based Methods*, volume 55. Journal of Experimental Botany, 2004.

[31] T. Klein, M. Zeppel, W.R.L. Anderegg, J. Bloemen, M.G. De Kauwe, P. Hudson, N.K. Ruehr, T.L. Powell, G. von Arx, and A. Nardini. Plant water stress and mortality. *Nature Plants*, 3:17004, 2017.

[32] FAO. *Coping with Water Scarcity: An Action Framework for Agriculture and Food Security*. Food and Agriculture Organization of the United Nations (FAO), Rome, 2012.

[33] Estimate gas fees — polygon gas station. `https://docs.polygon.technology/tools/gas/polygon-gas-station/`. Accessed Aug. 27, 2025.

[34] Polygon pos gas tracker. `https://polygonscan.com/gastracker`. Accessed Aug. 27, 2025.

[35] Polygon gas tracker — quicknode. `https://www.quicknode.com/gas-tracker/polygon`. Accessed Aug. 27, 2025.

[36] Aws iot core — pricing. `https://aws.amazon.com/iot-core/pricing/`. Accessed Aug. 27, 2025.

[37] Aws iot greengrass — pricing. `https://aws.amazon.com/greengrass/pricing/`. Accessed Aug. 27, 2025.

[38] Azure iot hub — pricing. `https://azure.microsoft.com/en-us/pricing/details/iot-hub/`. Accessed Aug. 27, 2025.

[39] M. Pincheira, C. García, H. Astudillo, and H. García-Molina. Characterization and costs of integrating blockchain and iot: A case study and cost model. *Systems*, 10(3):57, 2022.

[40] A. Tang et al. Assessing blockchain and iot technologies for agricultural supply chains: A review. *Discover Internet of Things*, 4(1):Article 21, 2024. Open access review summarizing feasibility, costs, and trade-offs.