Technical Report — ChainProof

Students:

- Syphax LAKHDARCHAOUCHE
- Kebe MANIANG
- Mamady Saiba Oulare
- Layba Oulare

A web-based supply chain tracking platform enabling secure product validation, real-world event confirmation, and transparent journey visualization

MONARK - UQAC

Date 19–12-2025

CONTENTS

1. TECHNICAL OVERVIEW

2.1 Project Objective

The objective of **ChainProof** is to provide a clear, reliable, and user-friendly solution for verifying product-related information using blockchain technologies.
The platform focuses on **ensuring trust and transparency** throughout the verification process while keeping the user experience simple and intuitive. Specifically, the project aims to:

- Enable users to **instantly verify** the authenticity of a product by scanning a QR code.
- Allow administrators to **manage products, transactions, and associated data** in a secure and organized way.
- Reliably access **trusted blockchain data** to confirm that information is accurate and immutable.
- Establish a foundation for **future expansion**, including additional roles, richer transaction histories, and enhanced interfaces for different actors in a supply chain.

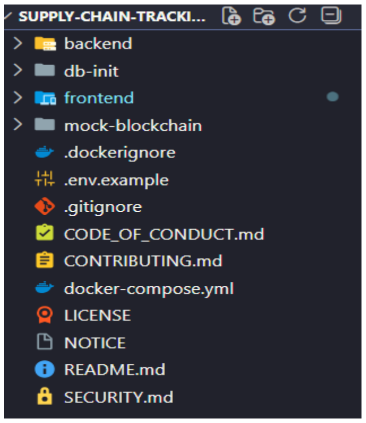
The **minimum viable product (MVP)** focuses on the core functionality necessary to achieve these goals, providing a **working, testable prototype** without including all potential features. By concentrating on essential features first, the project ensures:

- The main user flow is complete, functional, and understandable.
- Admin and user interactions are seamless and coherent.
- Blockchain integration demonstrates the **value of verified and immutable data**.

This approach allows the system to deliver a **functional and demonstrable solution** while leaving room for further improvements and open-source contributions.

2.2 System Architecture

The ChainProof platform follows a **web-based client-server architecture** that separates the responsibilities of the frontend, backend, and blockchain integration. This design ensures **scalability, security, and maintainability**.


SUPPLY-CHAIN-TRACKI...
backend
db-init
frontend
mock-blockchain
.dockerignore
.env.example
.gitignore
CODE_OF_CONDUCT.md
CONTRIBUTING.md
docker-compose.yml
LICENSE
NOTICE
README.md
SECURITY.md

Frontend (User Interface)

The frontend is a web application that provides:

- **Public user interface:**
  - Allows users to scan a QR code.
  - Displays verified product transactions retrieved from the blockchain.
  - Presents information in a simple and readable format.
- **Admin interface:**
  - Manages products, transactions, and associated contacts.
  - Secures access through authentication.
  - Provides a clear navigation structure for CRUD operations.

Technologies / Components:

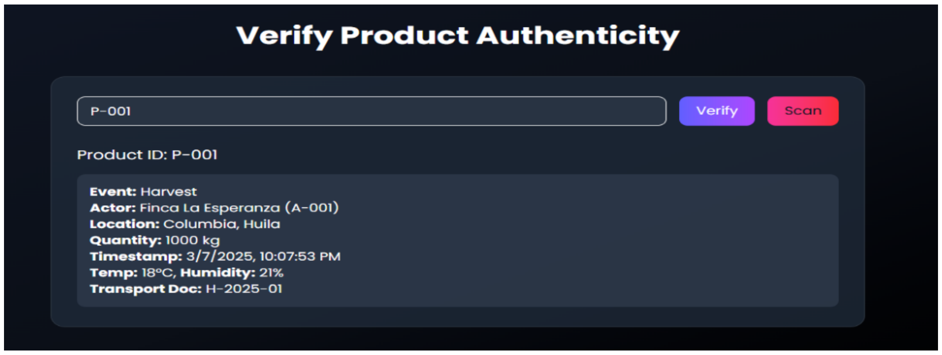
- Next.js + js + TypeScript for the interface
- QR scanning component
- State management for dynamic data display
- Communication with backend via RESTful APIs
- Authentication with auth0

Backend / Services Layer

The backend layer is responsible for **business logic, data persistence, and the orchestration of off-chain ↔ on-chain data.** Its main role is to provide **structured and verifiable responses** to the frontend based on identifiers and to maintain consistency between the admin database and blockchain reference

Main Responsibilities

- Manage core entities: products, actors, categories, contacts, trace events, and blockchain references.
- Expose REST API endpoints for both public and admin interfaces (CRUD, trace, transaction lookup).
- Resolve a transactionId (or blockchain_tx_uid) into readable data then process and normalize on-chain logs/events (ABI decoding) and return them in a human-readable format, including:
  - Event
  - Actor
  - Location
  - Quantity
  - Etc.



Product ID: P-001

Event: Harvest
Actor: Finca La Esperanza (A-001)
Location: Columbia, Huila
Quantity: 1000 kg
Timestamp: 3/7/2025, 10:07:53 PM
Temp: 18°C, **Humidity:** 21%
Transport Doc: H-2025-01

Technologies & Libraries

- **Language & Runtime:** Node.js + TypeScript
- **Database:** PostgreSQL (pg) – connection handled via a pool defined in db.ts
- **Environment & Configuration:** dotenv for loading .env / .env.local files
- **CORS:** cors to allow frontend access in development and production
- **API Documentation:** swagger-jsdoc + swagger-ui-express for OpenAPI documentation generated from route annotations
- **Development Tools:** ts-node, ts-node-dev, TypeScript

Smart contract artifacts:

- ABI stored in SupplyChain.json to decode on-chain events and map them to off-chain data

DevOps & Deployment

- **Containerization & Orchestration:** Dockerfiles and docker-compose.yml at the root to run the full stack (frontend, backend, and database).
- **Hosting:** Render
  - Backend: https://backend-supply-chain-tracking-uqac.onrender.com/
  - Frontend: https://supply-chain-tracking-uqac.onrender.com/

2.3 Data Flow

User QR Scan (Verification Flow)

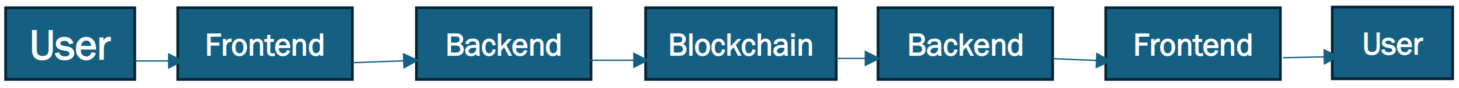
1. A user scans a **QR code** on a product using the public interface.
2. The frontend sends the **product ID** to the backend.
3. The backend queries the **product_transactions table** to find the associated transactionId.
4. The backend fetches the transaction **directly from the blockchain provider or indexer**.
5. The backend **parses and normalizes** the transaction data (timestamp, actor, location, note, proof) into a readable format.
6. The frontend displays the verified **transaction history** to the user, ensuring transparency and trust.

Admin Flow (Management & CRUD)

- The admin logs into the system via **authenticated access** (Auth0).
- Once authenticated, the admin can access the **dashboard and navigation panel** which includes all tools needed to manage the system:
  - Products
  - Contacts
  - Actors / Suppliers
  - Transactions
- In the **Products section**, each product provides three main actions:
  - **Transactions:** view the transaction(s) associated with this product.
  - **Details:** see detailed information about the product.
  - **Delete:** remove the product from the system.
  - Additionally, admins can **add new products.**
- All actions interact with the **backend**, which:
  - Ensures consistency between the **off-chain database** and blockchain references.
  - Retrieves and structures transaction data for display.
  - The admin interface allows **full CRUD management** while maintaining security and integrity, without modifying blockchain data directly.

Summary of Data Flow

The end-to-end flow is now:



2.4 Code Structure

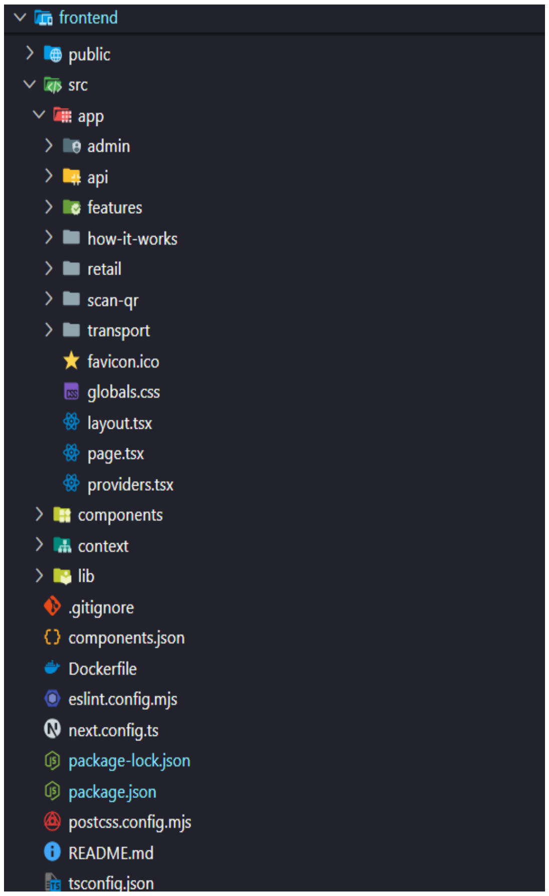
Frontend Structure:

The frontend of ChainProof is organized as follows:

1. app/

- Contains Next.js routing and page components for public and admin sections (e.g., scan-qr, admin/*).
- Organizes layouts, pages, and providers.

2. components/
   - Reusable and app-specific UI elements, including icons, buttons, navbar, admin components, and design system (ui/*, admin/*, icons.tsx, LoginButton.tsx, site-navbar.tsx).
   - Centralizes shared components.
3. lib/
   - Centralizes configuration and utilities (environment variables, helper functions, light network functions like wakeBackend) (env.ts, utils.ts, wakeBackend.ts).
   - Recommendation: add a services/ folder (e.g., services/api.ts) for backend REST calls.
4. context/
   - Global state management via React Context (products-context.tsx).
   - Shares product/session state between pages and components, acting as a client-side cache abstraction.
5. hooks/ *(to create if needed)*
   - For reusable hook-based logic (fetching, useAuth, useProducts).
   - Factorizes state and side-effects in the UI.
6. lib/utils.ts *(or utils/ folder)*
   - Utility functions for data formatting and processing (dates, event mapping, QR helpers).
   - Avoids duplication in components.
7. public/
   - Static assets served directly by Next.js (images, icons).
8. styles/ / globals.css
   - Global CSS, theme, and variables configuration.

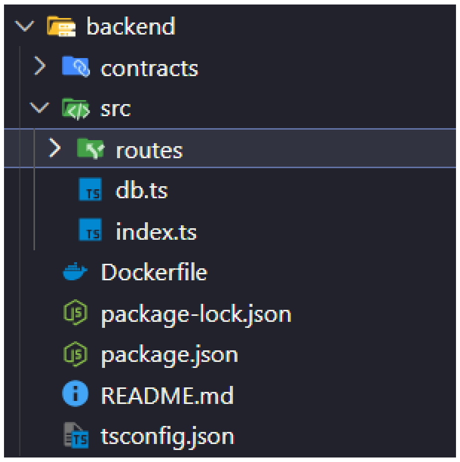


```
frontend
  public
  src
    app
      admin
      api
      features
      how-it-works
      retail
      scan-qr
      transport
      favicon.ico
      globals.css
      layout.tsx
      page.tsx
      providers.tsx
    components
    context
    lib
  .gitignore
  components.json
  Dockerfile
  eslint.config.mjs
  next.config.ts
  package-lock.json
  package.json
  postcss.config.mjs
  README.md
  tsconfig.json
```

Backend Structure:

The backend of ChainProof is organized as follows:

1. Root files
   - package.json — manages scripts and dependencies.
   - Dockerfile & docker-compose.yml — containerization and orchestration for backend, frontend, and database.
   - db-init/ — SQL scripts for database initialization.
2. src/
   - Contains all **TypeScript source code**, including server setup and business logic.
3. index.ts
   - Express entry point.
   - Configures middlewares (JSON parsing, CORS), sets up Swagger, and mounts routes.
4. db.ts
   - PostgreSQL pool configuration.
   - Provides a helper q() for executing queries.
   - Loads environment variables from .env.
5. src/routes/
   - Contains all HTTP routes for public and admin endpoints.
6. contracts/



```
backend
  contracts
  src
    routes
    db.ts
    index.ts
  Dockerfile
  package-lock.json
  package.json
  README.md
  tsconfig.json
```

- Smart contract artifacts (ABI).
- Example: SupplyChain.json — used to decode on-chain logs and events.

7. Swagger / OpenAPI
   - Documentation generated from route annotations and served at /swagger

Mock-Blockchain Structure

The mock-blockchain folder provides a development blockchain environment for testing and demonstrating on-chain interactions. Its structure and purpose are as follows:

Root files
- package.json — development dependencies and scripts (Hardhat, Viem, TypeScript).
- hardhat.config.ts — Hardhat configuration (Solidity compiler, networks, .env reading).
- tsconfig.json — TypeScript configuration.
- README.md — usage instructions and notes for the development blockchain.
- .gitignore — ignored files.

contracts/
- Contains Solidity smart contract sources.
- SupplyChain.sol — main traceability contract defining the structure of on-chain transactions and emitted events.

scripts/
- Utility scripts to deploy and interact with the contract:
  - deploy.ts — deploys the contract using Viem/wallet client, reading ABI/bytecode, and environment variables for RPC/keys.
  - sendTransaction.ts — reads data/event.json and calls addTransaction on the contract to send an on-chain event.
  - getTransaction.ts — retrieves a transaction on-chain by hash and decodes it using Viem client.

data/
- event.json — example payload used by sendTransaction.ts to send a test transaction.

Artifacts / build (expected)
- After Hardhat compilation, contract artifacts (ABI/bytecode) are stored in artifacts/, used by deploy.ts and sendTransaction.ts.
- SupplyChain.json in the repo is read by scripts for contract interaction.

Global Role

The mock-blockchain provides a **development blockchain environment** (Hardhat + scripts) to:
1. Compile and deploy the smart contract.
2. Produce test transactions.
3. Demonstrate reading and decoding on-chain transactions.

Scripts interact with the blockchain via **Viem RPC client**, using environment variables for **RPC URL, private key, contract address, and transaction hash**.

2. CHALLENGES AND SOLUTIONS

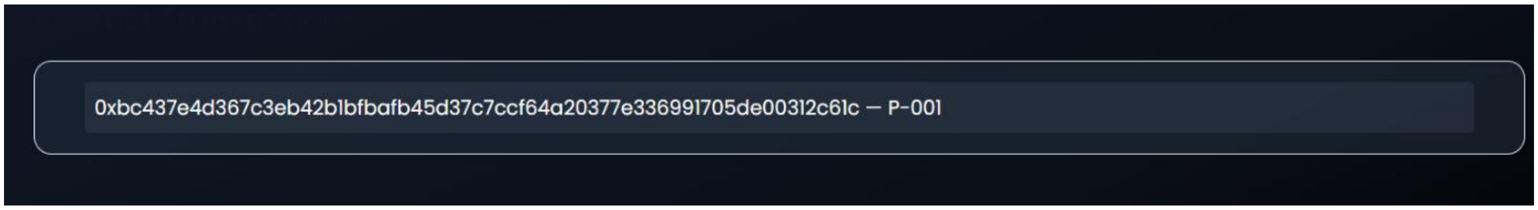
During the development of ChainProof, several technical challenges were encountered and addressed:

Challenge 1: Linking Products to On-Chain Transactions

- **Problem:** Once transactions were stored on the blockchain, it was challenging to link them back to their corresponding products. The blockchain only contains transaction data, and there is no inherent mapping to off-chain product identifiers.
- **Solution:** We created a dedicated products_transactions table in the backend database. This table stores pairs of product_id and transaction_id, allowing the system to efficiently fetch all blockchain transactions related to a specific product when requested by the frontend. This approach ensures accurate verification while keeping the product data structured and easily queryable.

Challenge 2: Confirming Smart Contract Deployment via ABI

- **Problem:** During development, automatically confirming the smart contract deployment and interaction on Tenderly was not working as expected. We needed to ensure that the smart contract was correctly deployed and accessible for reading and writing transactions.
- **Solution:** We manually confirmed the smart contract using its ABI. By decoding and verifying the contract's events on-chain, we ensured that it was properly deployed and functioning as intended. This manual confirmation provided reliability until full automation could be implemented.



```
0xbc437e4d367c3eb42b1bfbafb45d37c7ccf64a20377e336991705de00312c61c — P-001
```

3. ROLES & CONTRIBUTIONS

Team Member 1: Syphax Lakhdarchaouche

DevOps

- **Containerization:** Created Dockerfile and docker-compose.yml to containerize frontend, backend, and database in container.
- **Hosting & Deployment:** Deployed the full stack on Render, ensuring that both frontend and backend are publicly accessible.
- **Environment Management:** Managed .env files for both frontend and backend, ensuring secure handling of credentials and configuration.
- **Documentation:** Wrote README instructions for setup, running, and interacting with the system.

Blockchain

- Set up the development blockchain environment using Hardhat and Viem.
- Developed and deployed the Solidity smart contract SupplyChain.sol.
- Created scripts for sending test transactions (sendTransaction.ts) and reading them (getTransaction.ts).
- Ensured manual verification of smart contract deployment using the ABI, resolving automatic confirmation issues.

Frontend

- **Admin Dashboard:** Completed the admin dashboard including routing, CRUD functionality, and integration with backend endpoints.
- **Authentication:** Implemented Auth0 authentication for admin access.
- **API Integration:** Consumed backend routes in the admin interface to manage products, transactions, and contacts, Etc.
- **User Interface:** Improved design and usability of the public user interface,

Team Member 2 – Kebe Maniang

Backend

- **Backend Architecture & Routing:** Designed and implemented the backend routing structure using Express. Organized routes for products, transactions, actors, contacts, and trace events to ensure maintainable and scalable API endpoints.
- **Database Integration & Management:** Connected the backend to PostgreSQL, designing efficient queries and ensuring data consistency. Implemented the products_transactions table to link products with their blockchain transactions.
- **REST API Development:** Built multiple REST endpoints to handle CRUD operations for the admin interface and public endpoints for users. Ensured endpoints return structured and verifiable data.
- **Middleware & Configuration:** Configured essential middlewares including JSON parsing, CORS, and environment variable loading to support secure and smooth API interactions.
- **Swagger Documentation:** Added OpenAPI annotations to routes to automatically generate Swagger documentation for backend APIs.
- **Integration Testing & Validation**
- **Advanced debugging & backend stabilization**

Team Member 3 – Layba Oulare

Frontend – User Side

Contributed mainly during the early phase of the project to the initial design and setup of the client-side interface. This work focused on establishing the first public-facing structure of the platform and implementing the early informational pages.

- Developed the first version of the public website pages, including:
  - Home page (platform introduction)
  - Features page (overview of key capabilities)
  - How It Works page (high-level explanation of the verification process)
- Helped define the initial navigation structure and presentation of the client interface.
- Participated in the initial Front-end setup and early UI direction.
- Contributed to separating public-facing content from other platform areas.
- Did not complete the full functional implementation, as the architecture later evolved and responsibilities were redistributed.

Summary

Main contribution consisted of:

- establishing the first version of the client UI,
- developing early informational pages,
- supporting the initial design direction before later technical iterations.

Team Member 4 – Mamady Saiba Oulare

Design

Contributed to the first mockup version of the website design but did **not** complete the full UI/UX or the full admin implementation.

Frontend

Contribution to the Admin:

- Initially started the admin UI and created an early draft of the admin dashboard interface.
- Proposed the first visual idea for the structure of the admin area (sidebar + pages concept).

5.PERSPECTIVES / NEXT STEPS

Automated Smart Contract Interaction

- Implement automatic confirmation of smart contract deployment and transactions, removing the need for manual verification.
- Integrate with blockchain monitoring tools to trigger events and notifications in real-time.

Enhanced Product History

- Visualize detailed product journeys, showing a chronological map of all events associated with a product.
- Include more metadata for each transaction (location, actor role, timestamp, proof) to provide richer traceability.

Secure & Actionable Scanning

Currently, scanning primarily allows users to verify product information.
In future versions, scanning will also enable **secure event creation**, meaning actors will be able to:

- confirm product reception,
- validate transitions between supply chain stages,
- trigger blockchain-recorded events directly.

This transforms QR scanning from a passive feature into a secure action mechanism.

Multi-Actor Supply Chain Interfaces

In the future, the system will not only be for the admin and users.
Each actor in the supply chain will have their own dedicated interface with role-based permissions.
For example:

- **Suppliers** will be able to register or declare products.
- **Transporters** will update product status during shipment.
- **Distributors** will manage items during their handling stage.
- **Retailers** will handle the final delivery/availability stage.

Each actor will only see what they need, with the right permissions.
This makes the platform much more realistic and aligned with how real supply chains operate, while creating a richer and more trustworthy transaction history.

4. RETROSPECTIVE ANALYSIS – WHAT WE WOULD DO DIFFERENTLY

Future Improvements & Perspectives
To further enhance reliability, scalability, and realism, several improvements can be introduced in future iterations of the system:

- Full On-Chain Integration
  Currently, the system uses a mock-blockchain to simulate transactions. In the future, the platform can be fully migrated to a real blockchain network to ensure immutable, transparent, and verifiable traceability across the entire supply chain.
- Caching System for Better Performance
  Implementing a caching layer (such as Redis) will significantly reduce response times, minimize repeated blockchain and database calls, and improve the overall system performance and scalability, especially when handling large datasets.
- **Enhanced Security & Vulnerability Hardening**
  Security will be strengthened through multiple improvements, including:
  - Fixing backend and authentication vulnerabilities
  - Securing API endpoints and routes
  - Better management and encryption of environment variables
  - Improving access control and enforcing stricter role-based permissions
  - Strengthening protection against common vulnerabilities (Injection, XSS, CSRF, etc.)
    These enhancements will ensure the system remains robust, trustworthy, and safe for real-world use.

5. CONCLUSION

This project enabled the development of an end-to-end decentralized application integrating a blockchain-based smart contract layer with a scalable backend architecture, a structured database system, and a responsive frontend interface. Throughout the implementation process, we designed and deployed secure on-chain logic, established reliable backend routing and API communication, ensured efficient database connectivity and data consistency, and developed a user-friendly interface to interact with the system.
The clear separation of responsibilities allowed each team member to contribute effectively while maintaining cohesion across the system’s components. This collaborative workflow strengthened our understanding of distributed systems, smart contract integration, backend service orchestration, and full-stack application design. The resulting platform is functional, maintainable, and extensible, providing a solid foundation for future improvements and additional features.