1.6 Technical Specification Document

Index

1. Title Page

- Project Name: Plastiks powering plastic waste recovery for Danone and other leading brands with CARDANO built plastic credit marketplace
- Date: February 2025
- Version: 1

---

2. Executive Summary

- Objective: Integrate Cardano to the existing application, enabling connectivity with Cardano wallets, issuing of plastic and CO2 credits on Cardano and some UI updates to enhance UX and data extraction. BE work will also be done to enhance performance, security and scalability.
- Scope: Backend development, smart contract migration, updated UI, performance and scalability.
  - Smart contract functionalities on Cardano will mirror or enhance existing logic for issuing, managing, and verifying plastic and CO2 credits.

3. Functional Requirements

Core Features

3.1 Feature 1: Plastic Credit Image Redesign

- **Purpose:**
  To create a more professional and visually appealing representation of plastic credits that better reflects the professionality and data stored.
- **User Actions:**
  - View redesigned images of plastic credits in the app interface and in the NFT section on a web3 wallet.
  - Recognize relevant information visually from the redesigned image.
- **System Actions:**
  - Generate a new credit image dynamically based on our methodology and user feedback.
  - Update existing credits with the redesigned image format. (New image will not be changed for old credits in web3 wallets)
  - Ensure compatibility with different device resolutions for consistent visual quality.

User Stories for Feature 1: Plastic Credit Image Redesign

User Story 1: Viewing Redesigned Images

As a user of the Plastiks app,

I want to see visually updated and professional credit images in the app and my web3 wallet,

so that I can better understand the credit's data at a glance and trust its professional presentation.

User Story 2: Dynamic Image Generation

As a recovery entity creating plastic credits,

I want the system to generate professional and data-driven images dynamically based on the credit details I provide,

so that the image accurately reflects the methodology and adds credibility to the credit.

3.2 Feature 2: Enhanced Plastic Credit Detail Screen

- **Purpose:**
  To provide users with more detailed and transparent information about the plastic credit, improving trust and decision-making.
- **User Actions:**
  - Navigate to the plastic credit detail screen.
  - View detailed data such as:
    - Geolocation of recovery.
    - Point of origin and destination of recycled plastic.
    - The type and amount of plastic recovered.
    - Information about the entity recovering.
    - Link to blockchain details
- **System Actions:**
  - Retrieve and display data tied to the credit.
  - Fetch origin and destination points of the recycled plastic from the database.
  - Lik all of the data to the credit screen and display it with an easy-to-understand UI

User Stories for Enhanced Plastic Credit Detail Screen

User Story 1: Viewing Detailed Credit Information

As a buyer or app user,

I want to navigate to the plastic credit detail screen,

so that I can view comprehensive data about the credit, including geolocation, point of origin, destination, type of plastic, and recovery entity details.

User Story 2: Accessing Blockchain Details

As a user,

I want to access a link to the blockchain record associated with a specific credit,

so that I can verify its authenticity and track its data on the blockchain transparently.

3.3 Feature 3: New Create Credit Screen

- **Purpose:**
  To empower recovery entities to create higher-quality credits by allowing them to include more detailed information during credit creation.
- **User Actions:**

    - Access the new "Create Credit" screen.
    - Input detailed information about the credit, such as:
        - Type and amount of plastic.
        - Recovery location and timestamp.
        - Destination of recycled plastic.
        - Supporting documentation (e.g., photos, invoices).
    - Submit the credit for verification.
- **System Actions:**
    - Validate user-input data for completeness and accuracy.
    - Store supporting documentation securely.
    - Link all data to the credit token and issue it to the blockchain.

User Stories for New "Create Credit" Screen

User Story 1: Enhanced Credit Creation Input

As a recovery entity,

I want to access the new "Create Credit" screen to input detailed information such as plastic type, recovery location, timestamp, and destination,

so that I can generate high-quality credits with complete and transparent data.

User Story 2: Upload Supporting Documentation

As a recovery entity,

I want to upload supporting documentation, such as photos, invoice and waybills during credit creation,

so that the credit can meet verification requirements and enhance its credibility for buyers.

User Story 3: Blockchain Integration for Credit Issuance

As a recovery entity,

I want the system to validate my inputs, securely store my documentation, and link the detailed data to the credit token,

so that the credit is issued accurately and transparently on the blockchain.

3.4 Feature 4: Upgraded Sustainability Dashboard

- **Purpose:**
  To give users a clearer understanding of their environmental impact by displaying more comprehensive metrics and enabling downloadable reports.
- **User Actions:**
  - Access the upgraded sustainability dashboard.
  - View impact metrics presented in:
    - Graphs (e.g., recovery trends, KG savings, historical recovery data).
    - Other datasets (e.g., recovery site, roadmap information)
  - Generate and download impact reports.
- **System Actions:**
  - Fetch updated impact metrics (e.g., KG savings, volume of plastic recovered) from the database.
  - Dynamically generate visualizations (graphs and diagrams) using the latest data.
  - Compile and export detailed impact reports in PDF format.

User Stories for Enhanced Sustainability Dashboard

User Story 1: Viewing Expanded Impact Metrics

As a user,

I want to access the upgraded sustainability dashboard to view detailed metrics like recovery trends, KG savings, historical recovery data, and recovery site information,

so that I can gain a clearer understanding of my environmental impact.

User Story 2: Generating and Downloading Impact Reports

As a user,

I want to generate and download detailed impact reports containing metrics and roadmap data,

so that I can easily share or present my sustainability contributions to stakeholders.

User Story 3: Dynamic Visualizations of Recovery Data

As a user,

I want the system to dynamically generate graphs and datasets using the latest recovery metrics and roadmap information,

so that I can interact with accurate and up-to-date visual representations of my contributions.

3.5 Feature 5: Credit Collections Creation

- **Purpose:**
  To allow users to organize and manage their plastic and CO2 credits by storing the credits in collections, enhancing usability and user experience.
- **User Actions:**
  - Navigate to the “Credit” section.
  - Filter by type of credits by:
    - Selecting credits (plastic, CO2, or both) to include.
    - Filter by old to new and high to low (kg)
- **System Actions:**
  - Ensure credits can belong to multiple collections without duplication.
  - Dynamically update collections when credits are added, sold, or modified.

User Stories for Organizing Credits into Collections

User Story 1: Managing Credit Collections

As a user,

I want to navigate to the “Credit” section and filter my plastic and CO2 credits in my collection,

so that I can better manage and categorize my credits for easier access.

User Story 2: Filtering and Selecting Credits

As a user,

I want to filter my credits by type (plastic, CO2, or both) and sort them by attributes like date (old to new) or weight (high to low),

so that I can quickly find and include the credits I need in specific collections.

User Story 3: Dynamic Collection Updates

As a user

I want the system to dynamically update my collections when credits are added, sold, or modified,

so that my collections always reflect the current status of my credits without requiring manual adjustments.

3.6 Feature 6: CO2 Credit Generation and Differentiation

IMPORTANT: We will make the app ready for the issuance of CO2 credits, but we will not start the issuing before 2026 due to the work related to creating the methodology.

- **Purpose:**
  To enable the system to generate CO2 credits based on specific datapoints and ensure users can differentiate between plastic and CO2 credits where applicable. This enhances functionality and transparency for users managing diverse credit types.
- **User Actions:**
  - Input relevant datapoints (e.g., CO2 savings, source information, and methodology validation) during credit creation.
  - View generated CO2 credits in the dashboard alongside plastic credits.
  - Differentiate between plastic and CO2 credits based on visual or label indicators.
- **System Actions:**
  - Validate user-input datapoints and calculate CO2 savings according to predefined formulas or methodologies.
  - Display CO2-specific data points (e.g., CO2 savings) on the credit detail screen to differentiate it from plastic credits.
  - Ensure both plastic and CO2 credits are properly labeled and visually distinct within the platform for clear identification.

User Stories for CO2 Credit Generation and Differentiation

User Story 1: Generating CO2 Credits

As a user,

I want to input relevant datapoints to generate CO2 credits, so that I can contribute to reducing carbon emissions and track my impact alongside plastic recovery.

User Story 2: Differentiating Between Plastic and CO2 Credits

As a user,

I want to easily differentiate between plastic and CO2 credits in the system,

so that I can manage both types of credits effectively and understand their respective impacts.

User Story 3: Viewing CO2 Credit Details

As a user,

I want to view detailed information about CO2 credits, including CO2 savings and origin,

so that I can assess the value and impact of each CO2 credit alongside plastic credits.

4. Non-Functional Requirements

Non-functional requirements define the operational characteristics that the system must meet to ensure optimal performance, security, usability, scalability, and reliability.

4.1 Performance

- The system must ensure a response time of less than 5 seconds for user interactions, including loading credit details and generating reports.
- Smart contract execution on Cardano should be optimized for efficient transaction validation and minimal processing delays.
- API response times should be within acceptable latency thresholds to maintain a seamless user experience.
- Image rendering for redesigned plastic credits must be dynamic and efficient without significantly affecting performance.

4.2 Scalability

- The system architecture should support increasing numbers of users and transactions without performance degradation.
- Blockchain interactions should be designed to handle high transaction volumes efficiently, preventing congestion or excessive fees.
- The backend (Ruby on Rails + MySQL) must be optimized for horizontal scaling, ensuring smooth operation even as the user base grows.

- Efficient data caching and indexing should be employed to handle the expected growth of stored plastic and CO2 credit data.

4.3 Security

- Data Protection: All sensitive data, including user and transaction records, must be encrypted both in transit and at rest.
- Authentication & Access Control: Secure authentication via Web3 wallet login (e.g., Nami, Eternl, Yoroi)should be enforced for blockchain interactions.
- Smart Contract Security: Cardano smart contracts (Plutus scripts and monetary policy scripts) must undergo regular external audits to mitigate risks of vulnerabilities and exploits.
- Geolocation Data: If included, user geolocation data should be anonymized and securely stored to comply with GDPR and other data privacy regulations.

4.4 Usability

- The user interface must be fully responsive across desktop, tablet, and mobile browsers to provide a seamless experience.
- The system should be intuitive and user-friendly, ensuring that users can easily navigate features such as credit issuance, report generation, and sustainability dashboard insights.
- Accessibility compliance (WCAG 2.1) should be considered, ensuring readability, contrast, and usability for all users, including those with disabilities.
- Users should receive clear feedback and error messages in case of transaction failures, missing data, or blockchain sync issues.

4.5 Reliability

- The system must maintain 99.9% uptime, ensuring that credit issuance, transaction tracking, and report generation remain consistently available.
- A disaster recovery plan should be implemented, with regular automated backups of transaction data, MySQL records, and IPFS-stored documents.
- Automated monitoring and logging should be set up to detect anomalies in system performance, security breaches, and transaction failures, triggering alerts for rapid resolution.
- Blockchain interactions must include failover mechanisms to retry failed transactions or notify users in case of network congestion.

5. Technical Requirements

This section outlines the technology stack, system architecture, and third-party integrations required for the successful implementation of the Plastiks platform enhancements with Cardano blockchain integration.

5.1 Platform

- Web application built with Ruby on Rails (server-side rendering) for an optimized and scalable backend.
- Fully responsive UI to ensure seamless experience across desktop, tablet, and mobile browsers.
- No standalone mobile application, but mobile accessibility is prioritized.

5.2 Database

- MySQL is the primary relational database for structured data storage (user accounts, credit transactions, metadata).
- IPFS (InterPlanetary File System) for decentralized storage of immutable plastic and CO2 credit documentation, images, and supporting records.
- Caching & Performance Enhancements: Redis or similar caching mechanism to optimize database queries and reduce latency.

5.3 APIs & Integrations

- Web3 Wallet Integration: Support for Cardano-compatible wallets including Nami, Eternl, and Yoroi, enabling user authentication and transaction signing.
- Sustainability Data Sources: APIs for verified external sustainability data providers to enhance reporting and validation.
- Cardano Blockchain Explorer APIs: Real-time integration with blockchain explorers for tracking and verifying issued plastic and CO2 credits.
- Payment Gateway Integration: Stripe is integrated into the platform, allowing users to make payments via credit or debit card.
- Reporting & Analytics Tools: Integration with third-party visualization tools for generating detailed sustainability impact reports.

5.4 Blockchain Implementation

- **Blockchain:** Cardano (Native Implementation) for issuing and verifying plastic credits as NFTs.
- **Smart Contracts:**
    - Plutus scripts for executing on-chain logic related to plastic credit issuance and validation.

    - Monetary Policy Scripts for defining the NFT minting rules, ensuring compliance with sustainability verification standards.
- **Tokenization:**
    - Plastic credits will be issued as NFTs on Cardano, embedding immutable metadata that includes both plastic recovery data and CO2 savings information.
    - Each NFT will contain structured metadata stored on IPFS, linking it to its respective sustainability data.

6. Assumptions and Constraints

6.1 Assumptions:

- Users will have access to Cardano-compatible Web3 wallets (e.g., Nami, Eternl, Yoroi) for authentication and blockchain interactions.
- The Cardano blockchain will remain stable and operational, ensuring smooth deployment and transaction processing.
- Smart contract execution costs (transaction fees) on Cardano will remain within reasonable and predictable limits.
- The integration with Stripe will continue functioning as expected, allowing seamless fiat payments for marketplace transactions.
- Sustainability data sources and reporting APIs will remain accessible and operational for real-time sustainability tracking.
- Users will have basic familiarity with blockchain technology or will be provided with sufficient onboarding materials to interact with the platform effectively.

6.2 Constraints:

- Budgetary Limitations: Development, testing, and deployment resources are constrained by predefined financial limitations, impacting the scope of additional features or enhancements.

- Time Constraints: The project follows a strict timeline with phased rollouts, limiting the possibility of incorporating non-critical additional features within the initial release.
- Resource Availability: The development team's capacity is limited, and prioritization of core functionalities is required to meet deadlines.
- Regulatory Considerations: Compliance with financial regulations, sustainability reporting requirements, and blockchain-related policies may affect feature implementation.
- Blockchain Transaction Speed & Fees: Cardano's network conditions, including congestion and fluctuating fees, may impact the efficiency of smart contract execution and NFT issuance.
- External Dependencies: The system relies on third-party services (e.g., Stripe, Cardano Explorer APIs, IPFS) that may introduce potential points of failure or service disruptions beyond the control of the development team.

7. Acceptance Criteria

- Feature 1: New plastic credit image design, more professional and serious look. A new image must be autogenerated when the user creates a credit.
- Feature 2: Plastic credit detail screen, add datapoints: Geolocation, point of origin and destination. Data can now be viewed in the credit detail screen.
- Feature 3: Redesign of the “create credit” screen for credit issuers. The redesigned screen can be implemented for use. Add fields for geolocation, point of origin and destination.
- Feature 4: Sustainability dashboard redesign, users can now extract impact reports and view more sustainability impact metrics.
- Feature 5: New collections to store credits with filtering options.
- Feature 6: Enable users to create CO2 credits through a screen. (NB. No credits will be issued until methodology is established, early 2026)

8. Milestones and Timeline

`8.1 Phase 1: April – June 2025 (Core Platform Enhancements & CO2 Credit Framework)

- **April 2025:** Final UI designs for Feature 1 (new plastic credit image), Feature 2 (credit detail screen enhancements), and Feature 3 (redesigned credit creation flow) will be completed and reviewed.
- **May – June 2025:** Development phase for these features, ensuring integration with the existing platform.
- **By End of June 2025:**
  - All three features will be deployed and fully operational.
  - CO2 credit creation functionality will be introduced, allowing users to create CO2 credits in the same format as plastic credits, pending the finalized methodology.

8.2 Phase 2: June – September 2025 (Collections & Filtering Functionality)

- **June 2025:** UI design for Feature 5 (credit collections with filtering options) will be delivered and approved.
- **July – September 2025:** Development, testing, and optimization of collection storage and filtering functionalities.
- **By End of September 2025:**
  - Users will be able to save credits into collections and apply advanced filtering for streamlined access.

8.3 Phase 3: September – October 2025 (Sustainability Impact & Reporting Enhancements)

- **September 2025:** UI design for Feature 4 (redesigned sustainability dashboard) will be finalized.
- **September – October 2025:** Development, integration, and testing of the new dashboard with expanded sustainability impact metrics and reporting capabilities.
- **By End of October 2025:**
  - The new sustainability dashboard will be launched, allowing users to extract impact reports and analyze broader sustainability data.

9. Risks and Mitigation

9.1 Feature 1: New Plastic Credit Image Design

Risks

- **Design Complexity:** The autogenerated design may require additional processing power or storage.
- **Performance Issues:** If the image generation process is slow, it may impact credit creation speed.
- **UI/UX Challenges:** Users may not like the new design or may find it confusing.
- **Potential Bugs:** Errors in the automation could result in missing or incorrect images.

Mitigation:

- Use efficient image generation libraries to optimize performance.
- Implement caching to store generated images and reduce processing load.
- Conduct user testing to refine the design before launch.
- Implement fallback logic in case of failed image generation.

9.2 Feature 2: Plastic Credit Detail Screen – Adding Geolocation, Point of Origin, and Destination

Risks:

- **Data Integrity:** Inaccurate or missing location data could affect credit credibility.
- **Security Concerns:** Geolocation data might expose sensitive business information.
- **UI Overload:** Adding multiple data points could clutter the interface.

Mitigation:

- Validate and verify geolocation data before storage.
- Allow users to choose to share geolocation data optionally.
- Design a clean and structured UI to prevent clutter.

9.3 Feature 3: Redesign of "Create Credit" Screen with Additional Fields

Risks:

- Increased Complexity: More data fields might make the credit creation process slower or more difficult for users.
- Validation Issues: Errors in entering geolocation, origin, or destination could impact data accuracy.
- Backend Changes Required: The database structure might need updates to store new data fields.

Mitigation:

- Optimize form usability with auto-fill suggestions and validation checks.
- Provide clear instructions and tooltips to help users understand new fields.
- Ensure backward compatibility to prevent breaking existing credits.
- Implement field validation to prevent incorrect data input.

9.4 Feature 4: Sustainability Dashboard Redesign

Risks:

- **Data Accuracy:** Incorrect impact metrics could damage credibility.
- **Performance Issues:** Large datasets may slow down dashboard responsiveness.
- **User Adoption:** Users might resist change if the new interface is too different from the current one.
- **Integration Challenges:** New impact metrics may require backend changes

Mitigation:

- Verify all data calculations with external experts before launch.
- Optimize database queries and caching to enhance performance.
- Conduct usability testing to ensure the redesigned dashboard remains intuitive.
- Develop the dashboard in parallel with backend updates to ensure smooth integration.

9.5 Feature 5: Collections with Filtering Options

Risks:

- **Storage Challenges:** Large numbers of saved credits may require additional storage capacity.
- **Filtering Performance:** Poorly optimized filters could slow downloading times.
- **User Confusion:** Users may struggle to understand how to organize and retrieve collections.

Mitigation:

- Optimize database queries for efficient retrieval of saved credits.
- Implement dynamic indexing to improve filtering speed.
- Provide a user guide or tutorial on how to use the collection feature effectively.
- Test various filter parameters to ensure the feature remains user-friendly.

9.6 Feature 6: CO2 Credit Creation (No Issuance Until 2026)

Risks:

- **Unclear Methodology:** Without a finalized methodology, the feature might require major changes later.
- **User Confusion:** Users may attempt to issue CO2 credits before the methodology is approved.
- **Integration Issues:** CO2 credits may require different validation rules compared to plastic credits.

Mitigation:

- Clearly label the CO2 credit creation feature as "Experimental" or "Coming Soon."
- Ensure flexibility in the system to accommodate future methodological changes.
- Provide a user guide explaining that issuance will only happen after methodology approval.
- Work closely with sustainability experts (ECOTA) to align the feature with industry standards.

10. Smart Contract Technical Specification

10.1 Introduction:

The Smart Contract Technical Specification is the Integration of the Plastik Eco system migrating from Celo to Cardano Blockchain. Its Specifies the Cardano Overview, the migration of Smart Contract from the Celo Blockchain to Cardano Blockchain, the Structure and scope for the Haskell and Plutus Code along with Flow Diagram.

Smart Contract:
Solidity Code Explanation
Key Functionalities Implemented in Solidity

Feature	Description
Lazy Minting	NFTs are not minted upfront but instead generated when purchased using a signed voucher.
Token Sale	Users can buy NFTs or ERC-20 tokens using other tokens or ETH.
Lockup Mechanism	Airdropped tokens can be locked in a specified period to prevent instant sales.
Access Control	Contracts use Ownable and AccessControl to manage permissions.
Royalty Mechanism	Ensures creators receive a percentage of resale earnings using ERC-2981.
Verification System	Certain addresses must be verified before they can interact with specific functions.

- The contracts are well-structured, using OpenZeppelin's security features (Ownable, AccessControl, ERC-2981).
- Lazy minting optimizes gas costs, ensuring NFTs are only minted when needed.
  The token sale system supports multiple cryptocurrencies, making the contract flexible and scalable.
- The verification system prevents unauthorized users from interacting with the contract.
- The lockup system helps control token supply and prevents dumping after airdrops.

Translates to Cardano (Plutus & Haskell)

No Global Storage → Instead of mappings (mapping(address => bool)), Cardano uses UTXOs with Datum to store verification data.
 No ERC-20 or ERC-721 → Uses Native Tokens and Validator Scripts to enforce ownership and transfers.
 No Direct Transfers → Uses UTXO-based spending conditions instead of transferFrom().
 More Secure Execution → No risk of reentrancy attacks, thanks to the UTXO model.

Differences Between Solidity (Ethereum) and Haskell/Plutus (Cardano)

Feature	Solidity (Ethereum)	Haskell/Plutus (Cardano)	Notes
Execution Model	Account-based	EUTxO-based	Different transaction processing mechanisms.
Smart Contract State	Stateful (persistent storage)	Stateless (logic tied to UTXOs)	Needs explicit state transition design.
Function Calls	Direct contract calls	Script validation via spending transactions	No direct function calls; relies on spending rules.
Gas Model	Dynamic gas fee	Predictable execution cost	No unexpected execution fees in Plutus.
Events & Logs	emit events	No direct equivalent	Logs need to be stored in Datum
Global Variables	msg.sender, msg.value, block.timestamp	No direct equivalent	Must be provided as transaction inputs.
Loops & Recursion	Allowed	Limited	Transactions must be stateless; recursion is avoided.
Storage	Contract stores state	Uses UTXOs & Datum	Requires a different design pattern.

10.2 Cardano Start Guide

This guide provides a structured breakdown of Cardano blockchain fundamentals, transactions, consensus, fees, and smart contracts to understand how it works.

10.2.1 What is Cardano?

- Cardano is a blockchain, similar to Bitcoin and Ethereum.
- It is a distributed ledger, meaning it records transactions permanently and transparently.
- Decentralized – No single entity controls the network; transactions are verified by users worldwide.
- Immutable – Once a transaction is added, it cannot be edited or deleted.

- Permissionless – Anyone can participate by sending transactions or running a node.

10.2.2 What Makes Cardano Unique?

- Energy Efficient – Uses Proof of Stake (PoS) instead of energy-intensive mining (Proof of Work).
- Scalability – Built to handle high transaction volumes with low fees.
- Security & Research-Driven – Developed using peer-reviewed research to ensure long-term stability.
- Dual-Layer Architecture – Separates transactions and smart contracts for better efficiency.
- EUTXO Model – Uses an Extended UTXO (Unspent Transaction Output) model instead of Ethereum's account model.

10.2.3 Sending & Receiving ADA (Cardano's Native Currency)

Steps to Send ADA

1. Install a Cardano wallet (Lace, Nami).
2. Create a wallet and receive ADA (Get funds from an exchange or another wallet).
3. Enter the recipient's address and amount.
4. Pay a small transaction fee and confirm.
5. Transaction gets added to the blockchain.

Transaction Components

- Sender Address – Who is sending the ADA.
- Recipient Address – Who is receiving the ADA.
- Amount – How much ADA is being sent.
- Digital Signature – Proves ownership of funds.

10.2.4 Understanding Blockchain Transactions

- A transaction is just data that gets sent across the network.
- Before it is added to the blockchain, it is just a digital message stored on your device.
- Transactions must be signed using your private key before they are valid.
- Security – Only the person with the private key can spend their ADA.

10.2.5 Who Adds Transactions to Blockchain?

Block Producers (Stake Pool Operators)

- Special nodes that validate transactions and create new blocks.
- Receive transaction fees as a reward.
- Ensure network security and prevent fraud.

Consensus Mechanism (Ouroboros Proof-of-Stake)

- Randomly selects stake pool operators to validate transactions.
- Ensures that only valid transactions are added.
- More energy-efficient than Bitcoin's Proof-of-Work.

10.2.6 Transaction Fees in Cardano

- Every transaction requires a fee to prevent spam.
- Fees go to the stake pool operators who validate transactions.
- Automatically calculated based on network usage and transaction size.
- Lower than Ethereum fees due to an efficient design.

10.2.7 Cardano's Blockchain Layers

Cardano has two separate layers

1. **Cardano Settlement Layer (CSL)**
    a. Responsible for processing and securing transactions.
    b. Uses a distributed ledger and the Ouroboros PoS mechanism.
2. **Cardano Computation Layer (CCL)**
    a. Handles smart contracts and decentralized applications (DApps).
    b. Allows developers to build applications on Cardano.

Key Advantage – This separation improves scalability and flexibility.

10.2.8 Smart Contracts on Cardano

- Smart contracts are self-executing programs on the blockchain.
- Cardano uses Plutus (Haskell-based language) for writing smart contracts.
- More secure than Ethereum due to functional programming principles.
- Uses the EUTXO model instead of Ethereum's account-based model.

10.2.9 How Smart Contracts Work in Cardano

1. A user submits a transaction that interacts with a smart contract.
2. The contract processes the transaction based on predefined rules.
3. The result is recorded on the blockchain permanently.

10.2.10 How Transactions Are Processed in Cardano?

1. Users send a transaction from their wallet.
2. The transaction is broadcasted to the network.
3. Stake Pool Operators validate it using the Ouroboros protocol.
4. If valid, the transaction is added to a new block.
5. The new block is distributed across all nodes, updating the ledger.

Once added, the transaction is final and irreversible.

10.2.11 Extended UTXO Model (EUTXO) vs Ethereum's Account Model

What is UTXO?

- UTXO = Unspent Transaction Output
- Similar to cash transactions (you spend specific "coins").
- Each transaction uses UTXOs as inputs and creates new UTXOs as outputs.

EUTXO Benefits

- More secure – Each transaction must specify exact inputs and outputs.
- Predictable execution – No hidden costs, unlike Ethereum's gas fees.
- Parallel execution – Multiple transactions can run simultaneously, improving speed.

10.2.12 How Cardano Handles Smart Contract Transactions?

- A transaction can carry Plutus scripts instead of just transferring funds.
- Smart contract logic executes off-chain, reducing network congestion.
- Uses the EUTXO model to track funds and contract execution state.

10.2.13 Guide to Using Cardano

1. Install a Cardano Wallet → Lace, Eternl.
2. Get ADA → Buy from an exchange (Binance, Coinbase) and transfer to your wallet.

3. Explore the Cardano Blockchain → Use Cardano Scan to check transactions.
4. Stake Your ADA → Earn rewards by delegating your ADA to a stake pool.

10.2.14 Smart Contract Specifications:

Libraries to be incorporated in the Haskell File:

Libraries	Purpose	Key Features
Plutus.V2.Ledger.Api	Core data types and logic for Plutus scripts	Validator, Datum, Redeemer
Plutus.V2.Ledger.Contexts	Access transaction context for validation	ScriptContext, TxInfo
Plutus.V2.Ledger.Scripts	Compile and manage validator scripts	mkValidatorScript, Validator
PlutusTx	Compile Haskell code to Plutus Core and manage data types	compile, unstableMakeIsData
PlutusTx.Prelude	Provides arithmetic, logical operators, and debugging functions for Plutus Core compatibility	traceIfFalse, (+), div
Ledger	Interact with Cardano ledger, manage addresses and hashes	pubKeyHash, unPaymentPubKeyHash
Ledger.Typed.Scripts	Ensures type safety for smart contracts	TypedValidator, mkTypedValidator

All the Smart Contract is designed to manage Plastik Eco System on the Cardano Blockchain. It ensures transparency, immutability, and trust by implementing the following functionalities:

10.3 Technical Breakdown & Migration Plan: NFTStoreFrontV4 from Solidity (Ethereum) to Plutus (Cardano)

10.3.1 Introduction

Your Solidity smart contract, NFTStoreFrontV4, is a marketplace facilitating NFT purchases, lazy minting, fee distribution, and PRG (Plastic Recovery Grant) token integration. Migrating this to Cardano's Plutus framework requires significant changes due to architectural differences between Ethereum (account-based) and Cardano (UTXO-based).

Key Features & Required Changes

Feature	Ethereum (Solidity)	Cardano (Plutus - Haskell)	Migration Strategy
Lazy Minting	safeLazyMint() for NFTs	No direct equivalent	Pre-mint NFTs & track with UTXOs
NFT Ownership	Stored on contract storage	Stored in UTXO datum	Use Datum to track ownership
NFT Transfers	Uses safeTransferFrom()	No direct function calls	Handled via spending validators
PRG Token Attachment	Custom attachPRGToNFT() function	No inter-contract calls	Link PRG token metadata in UTXO datum

Fee Distribution	Platform, NGO, Royalties	No global state	Define fee logic in spending scripts
ERC20 Payments	Uses transferFrom()	No ERC20 equivalent	Handle via Native Assets & Constraints
Signature Verification	Uses msg.sender & ECDSA	No msg.sender	Use transaction signatures
Dynamic Fee Updates	Uses setPlatformServiceFee()	No contract state updates	Store in off-chain logic
Event Logs	Uses emit Event(...)	No native event system	Store logs in UTXOs

10.3.2 Migration Plan

1. **NFT Minting:**
    a. Replace safeLazyMint() with Cardano Native Tokens & Minting Policy.
    b. Use UTXO datum to track NFT ownership.
2. **NFT & PRG Transactions:**
    a. Use off-chain code to construct transactions.
    b. Validate ownership & signatures using Plutus validator scripts.
3. **Fee Distribution:**
    a. Store fees in UTXO datum and distribute upon spending.
4. **Payment Handling:**
    a. Use Plutus constraints to ensure correct token transfers.

10.3.3 NFT Marketplace Functional Modules

The NFTStoreFrontV4 contract needs to be broken into multiple modules on Cardano:

Module	Functionality
NFT Minting Policy	Controls minting of NFTs
NFT Marketplace Validator	Handles buying/selling logic
Fee & Royalty Distribution	Distributes Platform Fee, NGO Fee, Royalties
Chain Logic	Handles listing, price verification, PRG token linking

Code Structure:

NFT Marketplace Contract

The marketplace logic needs:

1. **Datum → Stores NFT details.**
   NFT Marketplace Datum
```
data NFTDatum = NFTDatum
  { owner   :: PaymentPubKeyHash  -- Seller's wallet
  , price    :: Integer          -- NFT price
  , token    :: AssetClass     -- NFT AssetClass
  , fees      :: Integer          -- Platform fee
  , prgToken  :: Maybe AssetClass   -- Optional PRG token attachment
  } deriving Show
```
2. **Redeemer → Defines buy/sell conditions.**

data NFTAction = BuyNFT

3. **Validator Script → Ensures correct transaction execution.**

10.3.4 Limitations of Migration

Feature	Ethereum (Solidity)	Cardano (Plutus)	Solution
Lazy Minting	safeLazyMint()	Not Supported	Pre-mint NFTs & use UTXO tracking
Function Calls	Cross-contract calls	Not Supported	chain logic handles execution
State Storage	Mappings & variables	Stateless	Encode state in UTXOs
Event Logs	emit Event(...)	No event system	Store logs in UTXOs
Fee Distribution	Stored in contract	No global storage	Processed via spending scripts

10.4 Technical Specification for Migrating PlasticRecoveryProjects from Solidity to Cardano (Plutus)

10.4.1 Overview

The PlasticRecoveryProjects contract on Ethereum manages a list of verified plastic recovery projects using an admin-controlled mapping (address => bool). Since Cardano uses the UTXO model, we restructure this contract to store verified projects in UTXOs instead of Solidity's persistent storage.

10.4.2 Code Deep Dive

The PlasticRecoveryProjects contract in Solidity is designed to maintain a registry of verified plastic recovery projects with admin-controlled access. Let's analyze its components:

Features Ethereum (Solidity)

Feature	Description
Access Control	Uses OpenZeppelin's AccessControlEnumerable to manage admin roles.
Storage	Uses mapping (address => bool) to track verified plastic recovery projects.
Verification Process	Admins can add or remove a project with addPlasticRecoveryProject().
Event Emission	Emits PlasticRecoveryProjectAdded or PlasticRecoveryProjectRemoved.
Query Function	isPlasticRecoveryProject() checks if an address is a verified project.

Cardano (Plutus)

Feature	Description
Admin Role Management	Admin role is enforced by transaction signature verification (instead of OpenZeppelin).

Storage Management	Verified projects are stored as UTXOs instead of mappings.
Verification Process	Admins add or remove projects by creating/spending UTXOs.
Event Emission	No native event system → Events are stored in transaction metadata.
Function Call for Querying	Query UTXOs at script address to check project verification.

Solidity vs. Plutus (Comparison)

Feature	Ethereum (Solidity)	Cardano (Plutus - Haskell)	Migration Strategy
Contract Storage	Uses mapping (address => bool)	No persistent storage	Store verified projects in UTXOs.
Role-Based Access Control	Uses OpenZeppelin AccessControl	No built-in roles	Verify admin's signature on transactions.
Function Calls	isPlasticRecoveryProject() returns true/false	No direct function calls	Query UTXOs at script address.
Event Logging	emit PlasticRecoveryProjectAdded()	No event system	Store logs in transaction metadata.

Contract Structure in Cardano

Component	Description	Implementation Type
NFTDatum	Stores verified project address.	Datum (On-Chain)
NFTAction	Defines actions: AddProject and RemoveProject.	Redeemer (On-Chain)
Validator Script	Validates admin authorization and project addition/removal.	Smart Contract (On-Chain)
Chain Logic	Handles transaction building for adding/removing projects.	Off-Chain (Frontend)
Query Function	Fetches verified projects from UTXOs.	Off-Chain (Frontend API)

10.4.3 Data Structure: Datum & Redeemer

Datum:

```
Datum: Stores the verified project address
data ProjectDatum = ProjectDatum
  { projectAddress :: PaymentPubKeyHash -- Address of verified project
  } deriving Show
```

Redeemer:

Redeemer: Defines action (Add or Remove)

```
data ProjectAction = AddProject | RemoveProject
```

10.4.4 Migration to Cardano

Advantage	Description
Enhanced Security	Transactions are cryptographically signed and immutable.
Decentralized Verification	No centralized storage → Verification is tracked in UTXOs.
No Reentrancy Attacks	Plutus contracts are stateless, preventing reentrancy vulnerabilities.
Deterministic Execution	No gas-based execution → Predictable transaction costs.

10.4.5 Limitations & Workarounds

Limitation	Ethereum (Solidity)	Cardano (Plutus - Haskell)	Workaround
Persistent Storage (mapping)	Yes	No	Store verified projects in UTXOs.
Role-Based Access Control	Yes (AccessControl)	No	Verify admin signatures in transactions.
Function Calls (isPlasticRecoveryProject())	Yes	No	Query UTXOs at script address.
Event Logging (emit Event())	Yes	No	Store logs in transaction metadata.

10.5 Specification for Migrating PlastikCryptoV2 from Solidity to Cardano (Plutus)

10.5.1 Overview

The PlastikCrypto contract in Solidity is responsible for:
Verifying EIP-712 signatures for NFT vouchers
Ensuring authenticity of NFT vouchers before minting
Recovering signer addresses from digital signatures

Since Cardano does not support EIP-712, we must rebuild the signature verification process.

10.5.2 Solidity Contract Key Features

Feature	Solidity Implementation
Signature Verification	Uses EIP712 to verify off-chain signatures.
NFT Voucher Validation	Hashes voucher data, signs using ECDSA, and recovers signer address.
Ensuring Authenticity	Prevents unauthorized NFT minting by verifying creator's signature.

10.5.3 Cardano (Plutus) Migration Plan

Feature	Plutus Implementation
Signature Verification	Uses Ed25519 signatures (instead of EIP-712).
NFT Voucher Validation	Hashes voucher data using Blake2b-256 (instead of keccak256).
Ensuring Authenticity	Uses Off-Chain Code to verify Ed25519 signatures before minting NFTs.

10.5.4 Detailed Contract Structure in Plutus

Modules Required

Component	Description	Plutus Implementation
NFT Validator Script	Ensures NFT is minted only by authorized creators.	On-Chain
Datum Storage	Stores NFT metadata & verification details.	On-Chain

Datum:

NFT Voucher Data (Replacing EIP-712)

```
data NFTDatum = NFTDatum
  { tokenAddress   :: BuiltinByteString  -- NFT Contract Address
  , tokenId        :: Integer            -- NFT Token ID
  , tokenURI       :: BuiltinByteString  -- Metadata URI
  , creatorAddress :: PaymentPubKeyHash  -- NFT Creator's Address
  , royalty        :: Integer            -- NFT Royalty Fee
  , signature      :: BuiltinByteString
  } deriving Show
```

Redeemer:

Redeemer: Defines NFT Minting or Transfer

```
data NFTAction = MintNFT | TransferNFT
```

- **Stores NFT metadata & creator details in Datum.**
- **Signature verification is handled off-chain.**
- **Ensures only authorized signers can mint NFTs.**

10.6 Migration of PlastikToken Smart Contract to Cardano (Plutus)

10.6.1 Introduction

This document outlines the technical migration of the PlastikToken contract from Ethereum (Solidity ERC-20) to Cardano (Plutus & Native Assets).

The goals of this migration:

Implement token minting and transfer with lockup.

Enforce time-based token lockup for specific addresses.

Replace ERC-20 features with Cardano Native Assets & UTXO Model.

Implement pausable token transfers controlled by an admin.

Features of PlastikToken

The Solidity contract **extends ERC-20** with:

Feature	Solidity Implementation
Token Minting	Uses _mint() to create tokens at deployment.
Pausable Transfers	Uses _pause() and _unpause() to enable or disable transfers.
Token Locking	Uses _locks[address] mapping to restrict transfers before a timestamp.
Lockers Management	Maintains _lockers list to manage accounts that can lock tokens.
Transfer Restrictions	_beforeTokenTransfer() ensures locked wallets cannot transfer.

10.6.2 Migration Plan for Cardano (Plutus) Differences Between Solidity & Plutus

Solidity Feature	**Cardano Alternative**	**Implementation**
ERC-20 Standard	Cardano Native Assets	Minting policy instead of _mint() function.
Pausable Transfers	Validator Script Restriction	Uses Datum to enable/disable transfers.
Token Locking	Datum-based time-lock	Uses transaction constraints before allowing transfers.
Transfer Restrictions	UTXO Model & Validator Logic	Transfers restricted based on on-chain data.

Plutus Contract Components

Component	**Description**
Minting Policy	Defines token issuance rules (Admin can mint tokens).
Validator Script	Enforces transfer restrictions based on lockup time.
Datum Storage	Stores token lockup expiry for each address.
Off-Chain Logic	Handles pausing/unpausing and role management.

Datum:

```
data PlastikDatum = PlastikDatum
{ owner :: PaymentPubKeyHash -- Owner of the tokens
, isPaused :: Bool -- Flag to indicate if transfers are paused
} deriving Show
```

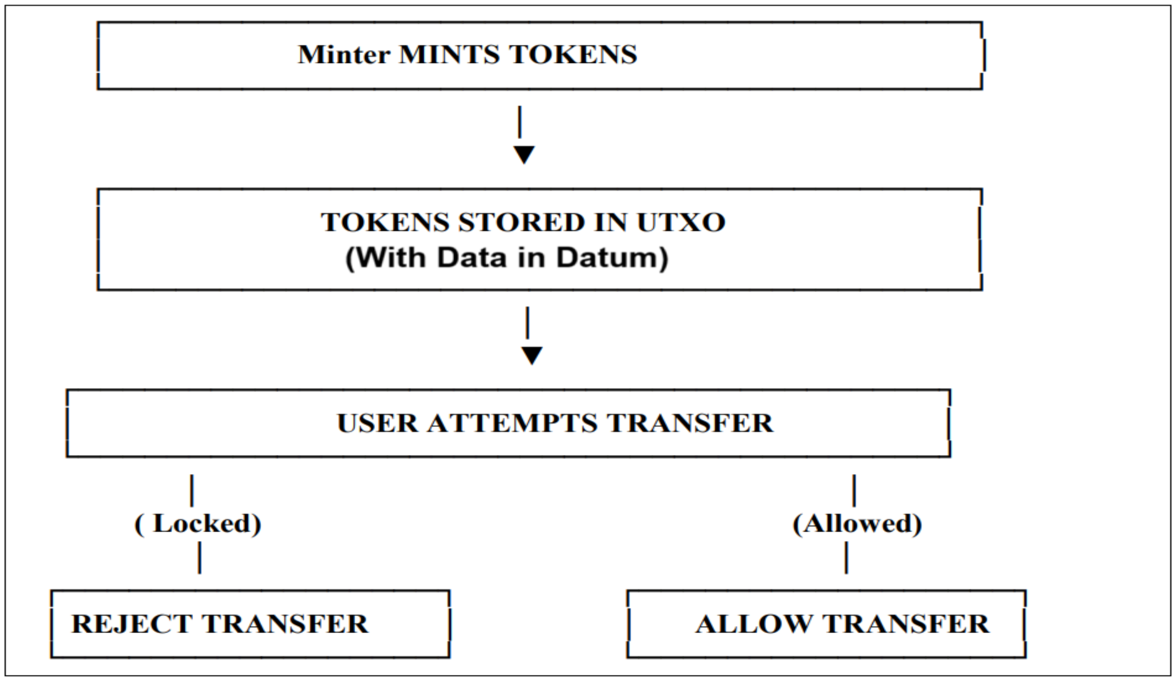
Redeemer
data PlastikAction = Transfer | Pause | Unpause

10.6.3 Migration Challenges & Solutions

Solidity Feature	Cardano Alternative	Implementation
ERC-20 Token Transfers	Native Tokens & UTXO-based Validation	Uses Plutus scripts for controlled transfers.
pause() & unpause()	Datum-based pause flag	Transfers are restricted to script level.

10.6.4 Data Flow of PlastikToken in Cardano

Data Flow Diagram



10.7 Specification Document for PlastikPRGV3 Migration to Cardano (Plutus)

10.7.1 Overview

This document provides a detailed technical specification for migrating the Ethereum-based PlastikPRGV3 smart contract (written in Solidity) to the Cardano blockchain using Plutus. The purpose of this migration is to:

- Implement multi-token NFT functionality on Cardano.
- Enable batch minting & metadata storage.
- Maintain lazy minting without EIP-712 signatures.

- Enforce NFT recycling verification rules.
- Replace ERC-1155 with Cardano's Native Assets and Validators.

The migration will focus on converting Solidity smart contracts into Plutus-based Cardano smart contracts while preserving:

- NFT minting with metadata
- Batch minting and verification
- Lazy minting (vouchers)
- Recycling validation & PRG token attachment

10.7.2 Solidity Contract Summary

The PlastikPRGV3 contract extends ERC-1155, implementing:

Feature	Solidity Implementation
Multi-Token Standard	Uses ERC1155 to handle multiple NFTs.
Batch Minting	Uses _mintBatch() to create multiple NFTs in one transaction.
Metadata Storage	Uses _setURI() for metadata links.
Lazy Minting	Uses safeLazyMint() with NFT Vouchers and EIP-712 signature verification.
Recycling Verification	Uses verifiedAccounts.isVerified(to) to ensure recyclers are valid.
Whitelist Management	Uses whiteListSenders mapping to restrict transfers.
PRG Token Attachment	Allows NFTs to store additional PRG token data.

10.7.3 Cardano Migration Plan

Since Cardano does not support ERC-1155, we use:

Feature	Plutus Implementation
Multi-Token Standard	Uses Native Assets (Multi-Asset Ledger) instead of ERC-1155.
Batch Minting	Uses Minting Policies
Metadata Storage	Store metadata in Datum.
Lazy Minting	Uses Datum-based NFT Vouchers with Ed25519 Signatures.
Recycling Verification	Enforced via Transaction Constraints & Validator Scripts.
Whitelist Management	Implemented using Policy Constraints.
PRG Token Attachment	Uses Datum storage to track PRG assignments.

10.7. 4 Plutus Smart Contract Components

Component	Description
NFT Minting Policy	Defines the rules for minting NFTs.
NFT Validator Script	Ensures NFT transfers follow ownership and recycling verification rules.

Datum Storage	Stores NFT metadata, ownership details, and PRG token data.
Chain Logic	Handles minting, batch transactions, and lazy minting.

Storage Model

Solidity Storage	Plutus Equivalent
mapping(uint256 => string) tokenURI	Datum (NFT metadata)
mapping(address => bool) whiteListSenders	Minting Policy Constraints
mapping(address => bool) verifiedAccounts	Transaction Constraints (Verification Check)
mapping(uint256 => mapping(address => mapping(uint256 => uint256))) attachedPRGToNFT	Datum Storage for PRG attachment

Datum (NFT Metadata & PRG Storage)

```
data NFTDatum = NFTDatum
  { owner    :: PaymentPubKeyHash
  , metadata  :: BuiltinByteString
  , prgAmount :: Integer
  } deriving Show
```

Redeemer

```
data NFTAction = MintNFT | TransferNFT | AttachPRG Integer
```

Minting NFT

```
mintNFT :: PaymentPubKeyHash -> BuiltinByteString -> Integer -> Contract w s Text ()
mintNFT owner metadata prgAmount = do
```

10.8 Specification Document for PlastikRoleV2 Migration to Cardano (Plutus)

10.8.1 Introduction

This document provides a detailed technical specification for migrating the Ethereum-based PlastikRoleV2 smart contract (written in Solidity) to the Cardano blockchain using Plutus.

The purpose of this migration is to:
Implement Role-Based Access Control (RBAC) on Cardano.
Ensure only authorized users can perform minting operations.
Replace Ethereum's AccessControl with Plutus signature verification.
Maintain security and decentralization.

Scope

The migration will replace Ethereum's AccessControl mechanism with Plutus-based on-chain logic, including:

Admin Role (Owner of the contract)
Minter Role (Authorized minters can mint NFTs)
Role Verification using Signature Constraints

10.8.2 Solidity Contract Summary

The PlastikRoleV2 contract implements role-based access control (RBAC) with the following functionalities:

Feature	Solidity Implementation
Admin Role	Uses DEFAULT_ADMIN_ROLE for admin permissions.
Minter Role	Uses Constants.MINTER_ROLE for minting permissions.
Role Management	Uses grantRole() to assign minter roles.
Role Verification	Uses hasRole() to verify if a user has the correct role.

10.8.3 Cardano Migration Plan

Since Cardano does not support AccessControl, we use:

Feature	Plutus Implementation
Admin Role	Implemented using Public Key Hash (PKH) verification.
Minter Role	Implemented using Transaction Signature Constraints.
Role Management	Controlled via On-Chain Validator Scripts.
Role Verification	Enforced via Plutus Validators & Transaction Constraints.

Functional Requirements

Role Management

- The contract should assign admin and minter roles.
- Only admins can grant minter roles.

Minter Role Verification

- Only minters should be able to mint new NFTs.
- The system should reject unauthorized minting requests.

Security & Constraints

- Signature verification should prevent unauthorized access.
- The admin should have full control over granting/removing minters.

10.8.4 Plutus Smart Contract Components

Component	Description
Admin Verification	Uses Public Key Hash (PKH) to verify admin role.
Minter Role Verification	Uses transaction signatures to verify authorized minters.
Validator Script	Ensures only authorized users can mint NFTs.
Chain Logic	Handles role assignment & verification.

10.8.5 Cardano (Plutus) Flow

1. **Contract Deployment**
    a. RoleDatum stores the admin's public key and a list of minters.
2. **Granting a Minter Role**
    a. The admin submits a transaction with GrantMinter action.
    b. The validator script ensures that only the admin can perform this action.

3. **Verifying Minter Role**
    a. Before an NFT is minted, verifyMinter() checks if the transaction is signed by an authorized minter.

Datum:

```
data RoleDatum = RoleDatum
  { admin    :: PaymentPubKeyHash
  , minters  :: [PaymentPubKeyHash]
  } deriving Show
```

Stores the admin's public key and a list of authorized minters.

Redeemer:

```
data RoleAction = GrantMinter PaymentPubKeyHash | RevokeMinter PaymentPubKeyHash
Defines two actions: GrantMinter (adds a new minter) and RevokeMinter (removes a minter).
```

Validation: (Ensuring Only Admin Can Modify Roles)

```
        GrantMinter newMinter -> traceIfFalse "Only admin can grant roles" isAdmin
    RevokeMinter oldMinter -> traceIfFalse "Only admin can revoke roles" isAdmin
```

Full Flow

Contract Deployment

- Stores admin's public key and empty minter list in Datum.

Granting a Minter Role

- The admin submits a transaction with GrantMinter.
- The validator checks if the admin signed the transaction.
- If valid, the new minter is added to the Datum.

Verifying Minter Role (Before Minting)

- When a user tries to mint an NFT, the validator:
    - Checks if their public key is in the minter list.
    - Allows minting only if they are authorized.

Revoking a Minter Role

- The admin submits a transaction with RevokeMinter.
- The validator removes the minter from the list.

10.8.6 Migration Challenges & Solutions

Solidity Feature	Cardano Alternative	Implementation
AccessControl (Roles)	On-Chain Role Storage	Stores roles in Datum.
grantRole() Function	Validator Script Constraints	Only admin can modify roles.
hasRole() Function	Plutus Validator & Queries	Checks role before minting.

10.9 Migration of VerifiedAccounts Smart Contract to Cardano (Plutus & Haskell)

10.9.1 Overview

This document explains the data flow and administrative workflow for minting and verifying accounts in the Cardano ecosystem using Plutus. It compares how Solidity manages verification and how this will be handled in Cardano's UTXO-based model.

Functions

Role	Functionality	Equivalent in Cardano
Admin (Owner)	Assigns validators, manages verification UTXOs, and mints native tokens.	Uses Minting Policy Scripts & Validator Scripts to control verification.
Validators	Mark accounts as verified or unverified.	Only allowed to create or spend UTXOs that store verification data.
Users	Check if an address is verified.	Look up UTXOs on the blockchain to determine verification status.
Smart Contract	Stores verified accounts in a mapping (Solidity).	Stores verification data inside a UTXO as a datum.

10.9.2 Differences Between Solidity & Plutus Implementation

Concept	Solidity (Ethereum)	Plutus (Cardano)
Storage Model	Uses mapping (address => bool) for verification.	No storage; uses UTXO-based verification records.
Access Control	Uses AccessControl for admin/validators.	Uses Validator Script & PubKeyHash-based verification.
Role-Based Authorization	Role-based permissions via hasRole().	Uses Plutus Validator scripts to check authorization.
State Modifications	Updates contract storage.	Uses UTXO creation & spending to track verification status.

10.9.3 Data Flow of Verified Accounts in Cardano

Step 1: Adding a Verified Account

Admin submits a transaction to verify an address.
A Plutus Validator Script checks if the sender has permission.
If valid, a new UTXO is created with the verified address stored in the datum.

Step 2: Checking Verification Status

A Plutus script scans the UTXO set.
If a UTXO exists with the given address in its datum, the address is verified.

Step 3: Removing a Verified Account

Admin submits a transaction to revoke verification.
A Plutus Validator Script verifies the admin's identity.
If valid, the verification UTXO is spent, removing the address from the list.

10.9.4 Detailed Contract Structure in Plutus

Datum:

```
data VerifiedDatum = VerifiedDatum
  { verifiedAddress :: PaymentPubKeyHash  -- Address being verified
  , validator      :: PaymentPubKeyHash  -- Validator who verified it
  } deriving Show
```

10.9.5 Migration Challenges & Solutions

Feature in Solidity	Can't Be Done in Cardano	Alternative Solution
Mapping for Verified Addresses	Cardano does not store mappings.	Uses UTXOs with datum to store verified accounts.
Access Control via AccessControl	No built-in role management.	Uses Plutus Validator Scripts to check if the sender is a validator.
Checking Verification On-Chain	On-chain logic can't loop over all addresses.	Uses UTXO lookups to check verification.
Removing an Address Instantly	In Solidity, the state updates immediately.	Uses UTXO spending to remove verification.

10.10 Migrating PlastikBurner Solidity Code to Cardano (Plutus & Haskell)

10.10.1 Introduction

This document provides a technical analysis and migration plan for converting the PlastikBurner Solidity smart contract into a Plutus smart contract for deployment on the Cardano blockchain.

The goal is to replicate the burn mechanism of Plastik Token while ensuring compatibility with Cardano's Extended UTXO (EUTXO) model and Plutus programming constraints.

Objectives

- Convert the PlastikBurner contract from Ethereum to Cardano.
- Ensure the contract maintains core functionalities, including token burning, tracking burn rates, and updating max yearly burns.
- Implement Plutus contracts using Haskell, ensuring they comply with Cardano's UTXO-based model.
- Identify limitations and workarounds due to differences between Solidity and Plutus.

Key Functionalities

1. **Tracks Burn Rate –** Determines how many tokens are burned per kg of plastic.

2. **Limits Annual Burn –** Ensures token burns do not exceed a set maximum per year.
3. **Allows Configuration –** Admin can modify the burn rate and maximum burn per year.
4. **Ensures Sustainability –** The contract dynamically adjusts burning rates based on production.

Key Variables & Calculations

- **maxBurnYear –** The maximum number of tokens that can be burned in a year.
- **plasticProduction –** The estimated yearly plastic production (300B KG).
- **burnPerKg –** The rate at which Plastik tokens are burned per kg of plastic.

10.10.2 Migration to Cardano (Plutus & Haskell)

Challenges in Migration from Solidity to Plutus

Feature	Solidity (Ethereum)	Plutus (Cardano)
Execution Model	Account-based model	UTXO-based model
State Storage	Uses contract state storage	Uses UTXOs to track data
Gas Fees	Paid per execution step	Fixed per transaction
Concurrency	Sequential transactions	Parallel transactions possible with UTXO model
Upgradability	Smart contract versions can be replaced	Scripts are immutable once deployed

10.10.3 Plutus (Haskell) Implementation

Design Approach

Plutus uses a different execution model where data is stored in UTXOs (Unspent Transaction Outputs) and modified via Datum and Redeemer scripts.

To replicate PlastikBurner, we will:

1. Store contract state in Datum (instead of Solidity's contract storage).
2. Use Redeemer for authorized updates to burn rates and plastic production.
3. Enforce access control via cryptographic signature validation.

Execution Flow on Cardano

Step	Description
1. Deploy Smart Contract	Deploy the Plutus script on Cardano.
2. Store Initial State	Create a UTXO with maxBurnYear and plasticProduction in Datum.
3. Submit Update Transaction	A user submits a transaction with a Redeemer to update burn settings.
4. Validation by Script	Plutus validator checks if the update is valid before allowing the transaction.
5. Finalization	If valid, the UTXO is updated with the new state.

10.10.4 Plutus Contract Structure

The Cardano version of PlastikBurner will include:

1. Datum – Stores the maxBurnYear, plasticProduction, and burnPerKg.
2. Redeemer – Allows updates to the burn settings (only by an admin).
3. Validator Script – Ensures only the correct conditions allow burning.

Datum:

```
data PlastikBurnDatum = PlastikBurnDatum
  { maxBurnYear :: Integer
  , plasticProduction :: Integer
  , burnPerKg :: Integer
  } deriving Show
```

Redeemer:

```
data PlastikBurnRedeemer = UpdateBurnRate Integer | UpdatePlasticProduction Integer
  deriving Show
```

Validator Script:

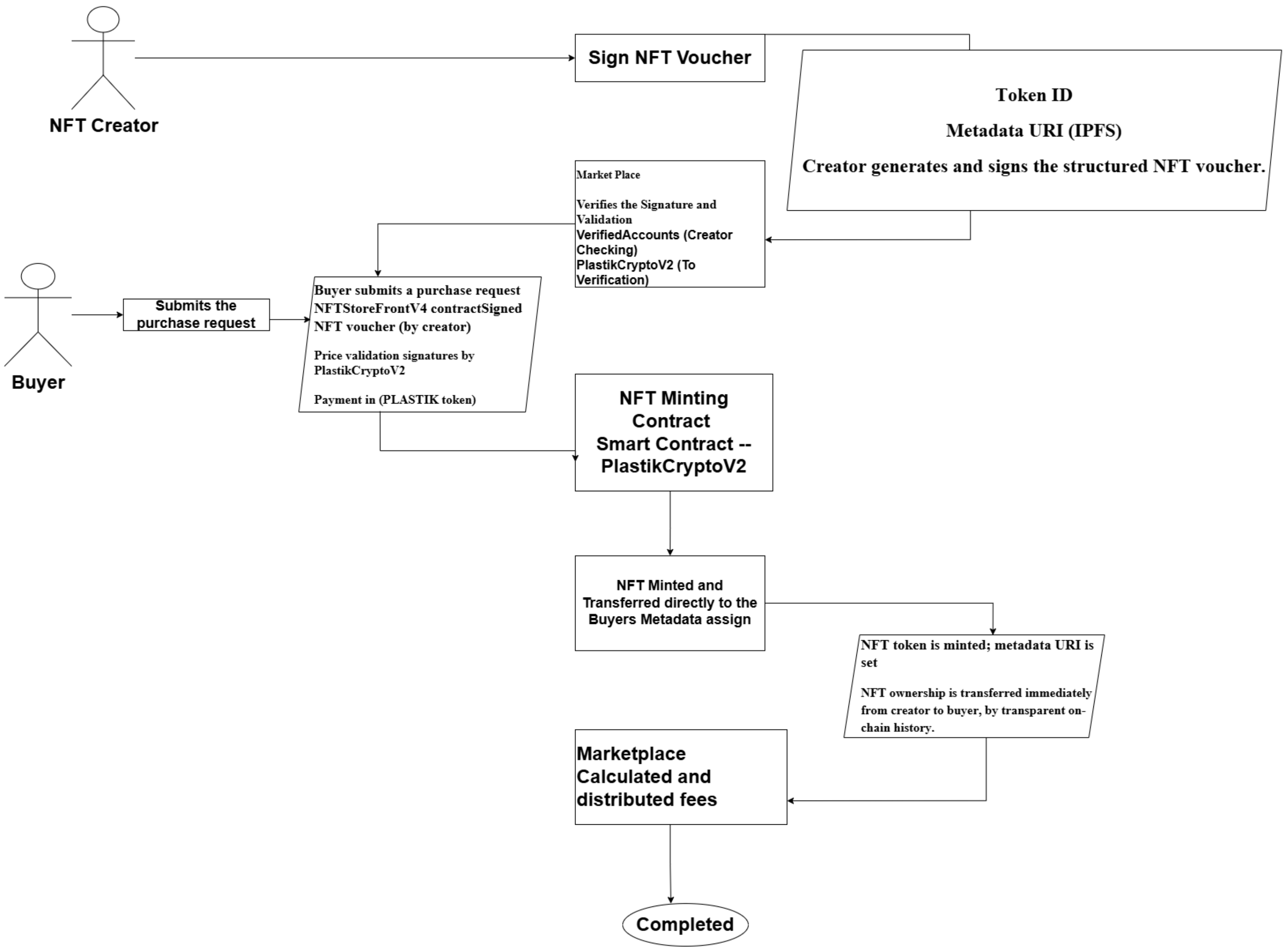
```
PlastikBurnDatum -> PlastikBurnRedeemer -> ScriptContext -> Bool
```

Limitations in Migration

Ethereum (Solidity)	Cardano (Plutus)	Challenge
Contract stores variables	Uses UTXO-based state (Datum)	No permanent storage; state updates require new UTXOs.
Modifiers (onlyOwner)	Signature validation in validator script	Need custom logic to restrict actions to admins.
transfer() function for tokens	Requires native token policies	Token burning requires a minting policy to burn tokens.

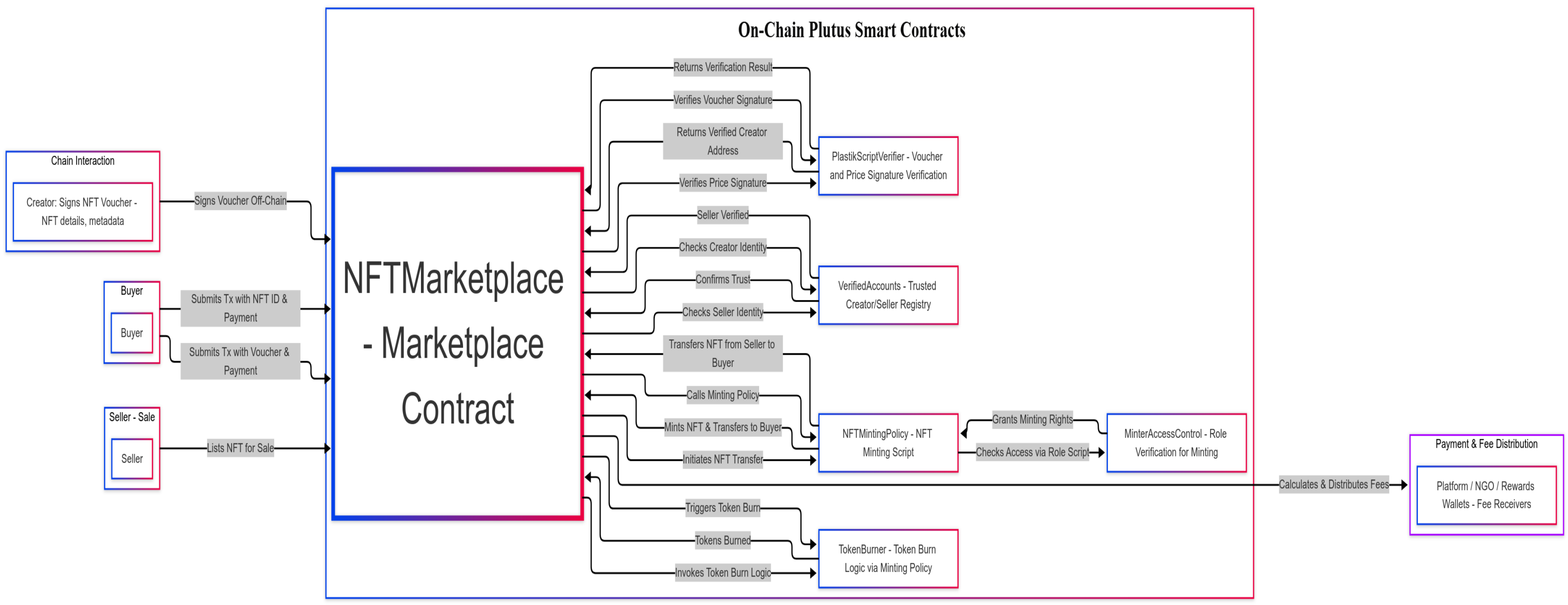
10.11 Data Flow Diagrams (DFDs) & Architectural Flowcharts

10.11.1 System-Level Data Flow Diagram (DFD)

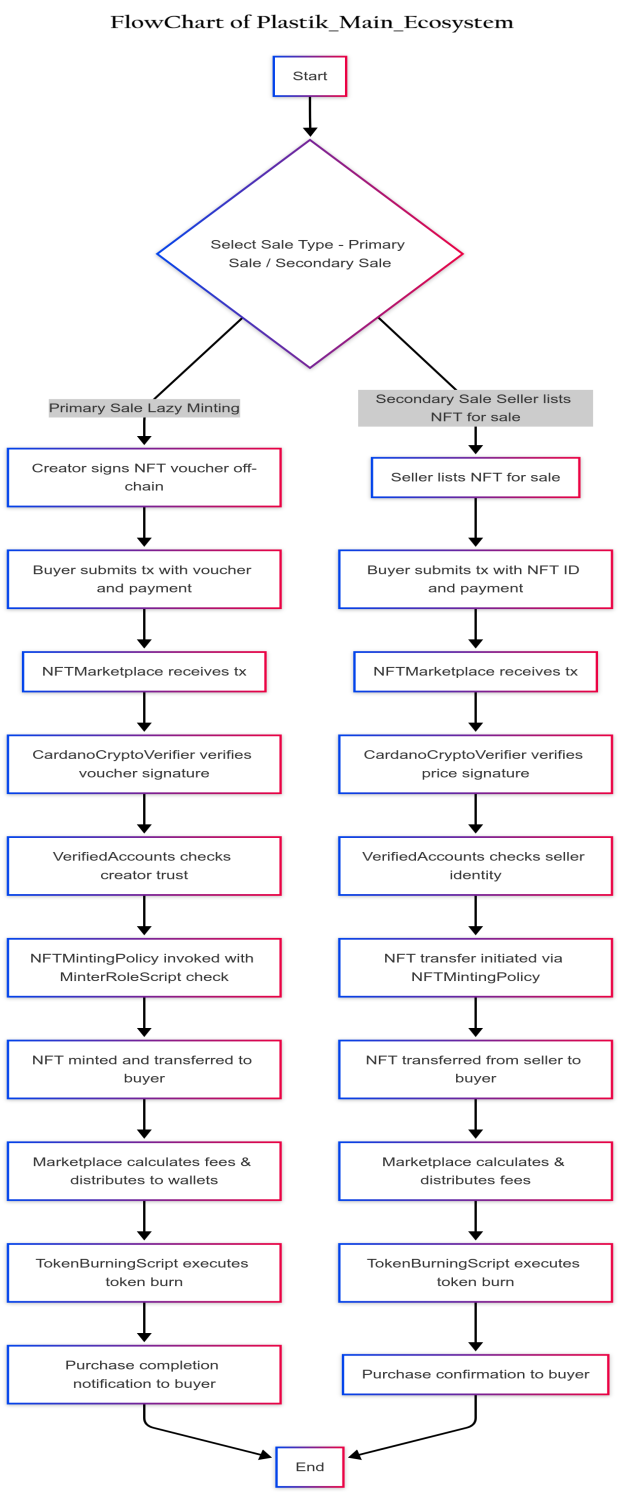

NFT Creator
Sign NFT Voucher
Token ID
Metadata URI (IPFS)
Creator generates and signs the structured NFT voucher.
Market Place
Verifies the Signature and Validation
VerifiedAccounts (Creator Checking)
PlastikCryptoV2 (To Verification)
Buyer
Submits the purchase request
Buyer submits a purchase request NFTStoreFrontV4 contractSigned NFT voucher (by creator)
Price validation signatures by PlastikCryptoV2
Payment in (PLASTIK token)
NFT Minting Contract Smart Contract -- PlastikCryptoV2
NFT Minted and Transferred directly to the Buyers Metadata assign
NFT token is minted; metadata URI is set
NFT ownership is transferred immediately from creator to buyer, by transparent on-chain history.
Marketplace Calculated and distributed fees
Completed

10.11.2 Smart Contract Execution Flow

Plastik's Eco System Smart Contracts Detailed Data Flow Diagram


On-Chain Plutus Smart Contracts
Chain Interaction
Creator: Signs NFT Voucher - NFT details, metadata
Signs Voucher Off-Chain
Buyer
Buyer
Submits Tx with NFT ID & Payment
Submits Tx with Voucher & Payment
Seller - Sale
Seller
Lists NFT for Sale
NFTMarketplace - Marketplace Contract
Returns Verification Result
Verifies Voucher Signature
Returns Verified Creator Address
Verifies Price Signature
PlastikScriptVerifier - Voucher and Price Signature Verification
Seller Verified
Checks Creator Identity
Confirms Trust
Checks Seller Identity
VerifiedAccounts - Trusted Creator/Seller Registry
Transfers NFT from Seller to Buyer
Calls Minting Policy
Mints NFT & Transfers to Buyer
Initiates NFT Transfer
NFTMintingPolicy - NFT Minting Script
Grants Minting Rights
Checks Access via Role Script
MinterAccessControl - Role Verification for Minting
Calculates & Distributes Fees
Payment & Fee Distribution
Platform / NGO / Rewards Wallets - Fee Receivers
Triggers Token Burn
Tokens Burned
Invokes Token Burn Logic
TokenBurner - Token Burn Logic via Minting Policy

10.11.3 Scope of Plastik Ecosystem Flowchart


FlowChart of Plastik_Main_Ecosystem
Start
Select Sale Type - Primary Sale / Secondary Sale
Primary Sale Lazy Minting
Creator signs NFT voucher off-chain
Buyer submits tx with voucher and payment
NFTMarketplace receives tx
CardanoCryptoVerifier verifies voucher signature
VerifiedAccounts checks creator trust
NFTMintingPolicy invoked with MinterRoleScript check
NFT minted and transferred to buyer
Marketplace calculates fees & distributes to wallets
TokenBurningScript executes token burn
Purchase completion notification to buyer
Secondary Sale Seller lists NFT for sale
Seller lists NFT for sale
Buyer submits tx with NFT ID and payment
NFTMarketplace receives tx
CardanoCryptoVerifier verifies price signature
VerifiedAccounts checks seller identity
NFT transfer initiated via NFTMintingPolicy
NFT transferred from seller to buyer
Marketplace calculates & distributes fees
TokenBurningScript executes token burn
Purchase confirmation to buyer
End

11. Cardano Microservice for Plastiks Marketplace

We have designed a REST-based microservice to abstract blockchain interactions and simplify integration with the marketplace. This document details the structure, API endpoints, data models, and core considerations for building this microservice.

11.1 Objectives

The goals for this microservice include:

- Ensuring a seamless transition of Plastic Credits (NFTs) to Cardano.
- Abstracting the complexities of Cardano's blockchain interactions.
- Maintaining existing marketplace functionalities with minimal disruption.
- Providing clear and intuitive RESTful APIs.

11.2 Technology Stack

For implementing this microservice, we propose:

- Node.js (Express.js)
- Cardano SDK (Blockfrost API, Cardano Serialization Library)
- Docker (Containerization)
- Kubernetes (Google Kubernetes Engine - GKE)
- Google Cloud Platform (GCP) for infrastructure

11.3 REST API Endpoints

Here's a refined version of the REST API documentation, clearly defining endpoints and payload structures aligned directly with your NFTStoreFrontV4.sol smart contract. This version explicitly shows that the microservice will receive complete data structures (such as vouchers and sell requests) rather than IDs, given it won't have direct access to the database.

11.4 Cardano Microservice REST API Specification for Plastiks Marketplace

Introduction

As the software architect designing the integration of Plastiks Marketplace with Cardano, I've structured this REST API microservice to interact seamlessly with the Cardano blockchain. The service receives fully structured payloads (vouchers, sell requests, signatures) from the marketplace frontend to perform blockchain transactions. It mirrors the data interactions and logic defined within the existing NFTStoreFrontV4.sol.

REST API Endpoints and Data Structures

11.4.1. Lazy Mint Voucher Creation (Signature)

This step happens; recovery entities sign a voucher with metadata. This signed voucher is then passed by the marketplace to the microservice.

11.4.2. Create Sell Request for Lazy Minted Plastic Credit

Endpoint:
POST /plastic-credits/sell-request

Request Payload:

```
{
  "voucher": {
    "tokenPolicyId": "policy_id_of_native_token",
    "tokenName": "Plastik_Token",
    "metadata": {
      "documentsHashes": {
        "invoiceHash": "md5_invoice_hash",
        "proofOfDeliveryHash": "md5_delivery_hash",
        "additionalDocuments": [
          {"type": "certificate", "hash": "md5_certificate_hash"}
        ]
      },
      "plasticRecoveredKg": 1000,
      "co2SavedKg": 200,
      "projectDetails": "Cleanup project in Mexico"
    }
  },
  "sellRequest": {
    "sellerAddress": "addr_recovery_entity",
    "paymentTokenPolicyId": "policy_id_of_payment_token",


"paymentTokenName": " Plastik_Token",
"price": 2500,
    "ngoFeePct": 500,
    "tokenPolicyId": "policy_id_of_native_token",
    "tokenName": " Plastik_Token"

  },
  "sellRequestSignature": "signed_sell_request"
}
```

Response:

```
{
 "sellRequestHash": "generated_hash_of_sellrequest",
 "status": "listed"
}
```

11.4.3. Purchase Plastic Credit & Lazy Mint NFT on Cardano

Endpoint:

POST /plastic-credits/purchase-and-mint

Request Payload:

```
{
  "buyerWalletAddress": "addr_buyer_wallet",
  "voucher": {
   "policyId": "cardano_policy_id",
   "assetName": "Plastik_Token",
   "metadata": {
    "documentsHashes": {
     "invoiceHash": "md5_invoice_hash",
     "proofOfDeliveryHash": "md5_delivery_hash",
     "additionalDocuments": [
      {"type": "certificate", "hash": "md5_certificate_hash"}
     ]
    },
    "plasticRecoveredKg": 1000,
    "co2SavedKg": 200,
    "projectDetails": "Cleanup project in Mexico"
   }
  },
  "voucherSignature": "signed_voucher",
  "sellRequest": {
   "sellerAddress": "addr_recovery_entity",
   "paymentCurrency": "lovelace",
   "price": 2500,
   "ngoFeePct": 500,
   "policyId": "policy_id",
   "assetName": "Plastik_Token",
  },
  "sellRequestSignature": "signed_sell_request",
  "sellPrice": {
   "currency": “ADA”,
   "ratio": 1,
   "decimals": 6
  },
  "signaturePrice": "signed_sell_price",
  "amountNft": 1,
  "isMultiNft": true,
  "sendTo": "addr_buyer_wallet"
 }
```

Response:

```
{
 "transactionId": "tx_cardano_hash",
 "plasticCreditId": "minted_plastic_credit_id",
 "status": "minting",
 "blockchain": {
  "txHash": "tx_cardano_hash",
  "blockNumber": 123456
 }
}
```

11.4.4. Retrieve Minted Plastic Credit Information

Endpoint:

GET /plastic-credits/{plasticCreditId}

BlockFrost API -

```
curl -H "project_id: YOUR_API_KEY"

https://cardano-xxxxnet.blockfrost.io/api/vx/assets/{plasticCreditId}/{policyId}{assetName}
```

Response:

```
{
  "plasticCreditId": "minted_plastic_credit_id",
  "ownerWalletAddress": "addr_buyer_wallet",
  "policyId": "Policy_Id",
  "assetName": " Plastik_Token ",
  "metadata": {
   "documentsHashes": {
    "invoiceHash": "md5_invoice_hash",
    "proofOfDeliveryHash": "md5_delivery_hash",
    "additionalDocuments": [
     {"type": "certificate", "hash": "md5_certificate_hash"}
    ]
   },
   "plasticRecoveredKg": 1000,
   "co2SavedKg": 200,
   "projectDetails": "Cleanup project in Mexico"
  },
  "status": "minted",
  "blockchain": {
   "txHash": "tx_cardano_hash",
   "mintedTimestamp": "2025-03-01T13:00:00Z",
   "blockNumber": 123456
  }
}
```

11.4.5. Check Blockchain Transaction Status

Endpoint:

GET /transaction/{transactionId}

BlockFrost API -

```
curl -H "project_id: YOUR_API_KEY"

https://cardano-xxxxnet.blockfrost.io/api/vx/ api/vx/transaction/{transactionId}
```

Response:

```
{
  "transactionId": "tx_cardano_hash",
  "status": "confirmed",
  "blockchainDetails": {
    "hash": "25f1c1c2d0xxxxxxxxe0aeddc4450625xxxxxxxx92f9ee70f",
    "block": "ada6617e1f77fb7a9f4bd6e7dee76d276b25abb3b4933",
   "block_height": 2685156,
   "block_time": 1726224343,
   "slot": 70541143,
   "index": 3,
    "output_amount": [
      {
        "unit": "lovelace",
        "quantity": "15274517"
      },
      {
        "unit":
"436941ead56c61dbf9xxxxxxxxb5f566cac08f8c957f28f0xxxxxxbd60d4b5041xxxx54e54e",
        "quantity": "724290"
      }
    ],
    "fees": "169581",
    "deposit": "0",
    "size": 308
  }
}
```

11.5 Key Data Structures Used

- **Voucher:**

Includes metadata with document hashes and project-specific details, prepared and signed by the recovery entity.

- **SellRequest:**

Includes listing details like seller address, Cardano Native Token currency, price, and NGO fees.

- Signatures:
- voucherSignature: Signed voucher by the recovery entity.
- sellRequestSignature: Signature of the sell request.
- SellPrice:

Represents currency, ratio, and decimals for transaction validation.

Security Practices

- JWT Authentication for API calls.
- Input sanitization and strict payload validation.
- Use of Google Secret Manager for secure handling of keys.

Scalability & Reliability

- Kubernetes-based deployment with auto-scaling.
- Efficient caching mechanisms.
- Robust handling of blockchain transaction confirmations.

Monitoring & Logging

- Comprehensive logging for debugging and compliance.
- Google Cloud Operations Suite and Sentry integration for monitoring.

Approval Section

- **Approved By: Daniel Garcia CTO**

[illegible]