



**VIT<sup>®</sup>**  
**Vellore Institute of Technology**  
(Deemed to be University under section 3 of UGC Act, 1956)

# **Blockchain Technology**

## **Project Report**

### **EnergySim Marketplace**

**Team Members:**

- Sushen Grover (23BCE1728)
- Advit Bhutani (23BCE1883)

**Course Code:** BCSE324L

**Slot:** G2+TG2

**Date:** 04/11/2025

**Submitted To:** Dr. Anubha Pearline S

# Table of Contents

1. Abstract
2. Introduction
  - 2.1. Problem Statement
  - 2.2. Project Objectives
  - 2.3. Scope of Work
3. System Design & Architecture
  - 3.1. High-Level Architecture
  - 3.2. Technology Stack
4. Implementation Details
  - 4.1. The Blockchain Layer (Smart Contracts)
    - 4.1.1. EnergyToken.sol & CarbonCreditToken.sol
    - 4.1.2. Marketplace.sol
  - 4.2. The Backend Layer (Simulation)
    - 4.2.1. simulator.py: The Simulation Engine
    - 4.2.2. run.py: Backend Server
  - 4.3. The Frontend Layer (User Interface)
    - 4.3.1. Wallet Connection (App.jsx)
    - 4.3.2. Real-Time Balances (Dashboard.jsx)
    - 4.3.3. P2P Trading Flow (SellForm.jsx & MarketplaceListings.jsx)
5. Results & Demonstration
  - 5.1. End-to-End System Flow
  - 5.2. Key Functionalities
6. Challenges Faced
7. Conclusion & Future Work
8. References

# 1. Abstract

This report details the design, development, and implementation of the "EnergySim Marketplace," a decentralized application (dApp) that simulates a peer-to-peer (P2P) energy trading system using blockchain technology. The project creates a secure, transparent, and autonomous ecosystem where users (prosumers) who generate excess renewable energy can tokenize it and sell it directly to other users (consumers). The system also incentivizes green energy production by automatically minting and awarding tokenized carbon credits. The project consists of three main components: a set of Solidity smart contracts deployed on a local Hardhat blockchain to manage tokenization and trade logic; a Python backend that simulates energy generation and consumption, automatically minting new tokens; and a modern React frontend that allows users to connect their Web3 wallets, view real-time token balances, and participate in the P2P marketplace. This report documents the complete end-to-end functionality, from simulated energy generation to the successful execution of a decentralized trade.

## 2. Introduction

### 2.1. Problem Statement

The traditional energy grid is highly centralized, relying on a few large-scale producers and intermediaries to manage distribution and sales. This model leads to inefficiencies, higher costs for consumers, and a lack of transparency. Furthermore, it creates significant barriers for small-scale renewable energy producers (prosumers) who wish to sell their excess energy back to the grid, as they are often given uncompetitive, fixed prices. There is also a growing need for a reliable and transparent system to track and trade carbon credits to incentivize sustainable practices.

### 2.2. Project Objectives

The primary goal of this project is to design and build a functional prototype of a blockchain-based system that addresses these problems. The core objectives are to:

- **Enable Fair Energy Access:** Allow users to trade energy directly with one another without a central authority, promoting a fair and open market.
- **Reward Green Energy:** Automatically issue tokenized carbon credits (\$CCT) to users who generate verified renewable energy, creating a direct financial incentive for sustainability.
- **Ensure Trust & Transparency:** Utilize smart contracts to automate and secure all transactions, making the entire trading process transparent, auditable, and trustless.
- **Simulate a Closed-Loop Economy:** Create a complete end-to-end system, from data simulation to token minting, wallet integration, and decentralized trading.

### 2.3. Scope of Work

This project implements a full-stack simulation of the proposed system. Instead of integrating

with physical IoT smart meters (which is outside the scope), this project uses a Python-based backend simulator to mimic energy generation/consumption data. The project's scope includes:

- Developing and deploying smart contracts for energy and carbon credit tokens (ERC-20 standard).
- Developing and deploying a smart contract for the P2P marketplace, including escrow and payment logic.
- Building a Python backend to simulate prosumer data and interact with the smart contracts to mint tokens.
- Creating a React-based frontend (dApp) for users to connect their MetaMask wallets, view balances, and trade tokens.
- Testing the full end-to-end flow on a local Hardhat blockchain network.

## 3. System Design & Architecture

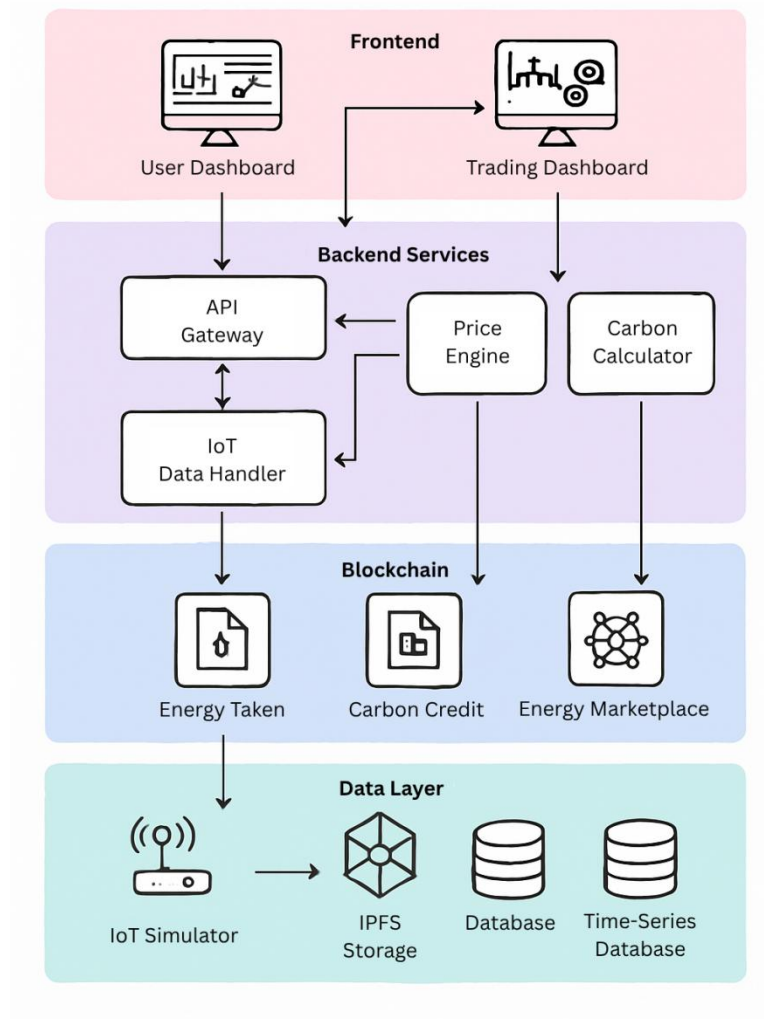
### 3.1. High-Level Architecture

The system is designed as a three-tier decentralized application, separating the core logic (blockchain), business logic (backend), and presentation (frontend).

- **Blockchain Layer:** The foundation of the system. It runs on a local Hardhat (Ethereum-like) network and hosts the Solidity smart contracts that define the rules of the economy. This layer is the "source of truth" for token ownership and trade execution.
- **Backend Layer:** A Python application that serves two purposes:
  1. It runs the `simulator.py` script to act as a trusted oracle, simulating data that would otherwise come from real-world IoT devices.
  2. It uses the "Owner" account's private key to call the mint function on the token contracts, creating new tokens for the simulated prosumers.
- **Frontend Layer:** A React (Vite + Tailwind CSS) single-page application that runs in the user's browser. It uses the Ethers.js library to connect to the user's MetaMask wallet and interact directly with the smart contracts on the blockchain.

## P2P Energy Trading & Carbon Credit Tracking System

Proposed Architecture Diagram



Architecture Diagram

### 3.2. Technology Stack

- **Frontend:** React.js, Vite (build tool), Ethers.js (blockchain interaction), Tailwind CSS (styling).
- **Backend:** Python, Flask (web server framework), Web3.py (blockchain interaction).
- **Blockchain:** Solidity (smart contract language), Hardhat (development environment, local node), OpenZeppelin Contracts (secure ERC-20 templates).
- **Wallet:** MetaMask (Web3 provider and wallet).

## 4. Implementation Details

This section details the code and logic behind each layer of the application.

## 4.1. The Blockchain Layer (Smart Contracts)

We developed three core smart contracts in Solidity.

### 4.1.1. EnergyToken.sol & CarbonCreditToken.sol

To represent energy and carbon credits as tradable assets, we created two standard ERC-20 tokens. Both contracts are nearly identical, inheriting from the secure OpenZeppelin ERC20 and Ownable contracts. The key feature is a custom mint function, which is restricted to the contract's "owner" (our backend server) to control the creation of new tokens.

#### EnergyToken.sol (Core Logic):

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

contract EnergyToken is ERC20, Ownable {
    /**
     * @dev Sets the token's name, symbol, and initial owner.
     */
    constructor(address initialOwner)
        ERC20("Energy Token", "ETKN")
        Ownable(initialOwner)
    {}

    /**
     * @dev Creates `amount` new tokens for the `to` address.
     * Can only be called by the contract owner.
     */
    function mint(address to, uint256 amount) public onlyOwner {
        _mint(to, amount);
    }
}
```

### 4.1.2. Marketplace.sol

This is the most complex contract, acting as the decentralized "stock exchange" for our tokens. It manages all trade logic in a trustless manner.

- **Sale struct:** A custom data structure to store the details of each listing (seller, token amount, price in ETH, active status).
- **createSale(.):** A function that allows users to list their tokens. It requires two transactions from the user:

1. An approve() call to the token contract, giving the marketplace permission to move their tokens.
2. The createSale() call, which uses transferFrom() to pull the tokens into the marketplace contract for escrow, and then lists the sale.
- **executeSale(..):** A function that allows a buyer to purchase a listing. It is marked payable, meaning it can receive ETH. It automatically:
  1. Verifies the correct amount of ETH was sent.
  2. Transfers the escrowed tokens (\$ETKN) to the buyer.
  3. Transfers the received ETH to the seller.
  4. Marks the sale as inactive.

### Marketplace.sol (Core Logic Snippets):

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;
```

```
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
```

```
contract Marketplace is Ownable {
    IERC20 public immutable energyToken;
    IERC20 public immutable carbonCreditToken;
```

```
    struct Sale {
        uint256 id;
        address seller;
        address tokenContract;
        uint256 amount;
        uint256 price; // Price in WEI
        bool active;
    }
```

```
    mapping(uint256 => Sale) public sales;
    uint256 public nextSaleId;
```

```
    event SaleCreated(uint256 id, address seller, uint256 amount, uint256 price);
    event SaleCompleted(uint256 id, address buyer);
```

```
    constructor(...) { ... }
```

```
    function createSale(address _tokenContract, uint256 _amount, uint256 _price) external {
        require(_amount > 0, "Amount must be greater than zero");
        require(_price > 0, "Price must be greater than zero");
```

```
        // Pull tokens from seller into this contract (escrow)
        IERC20(_tokenContract).transferFrom(msg.sender, address(this), _amount);
```

```

    // Create and save the new sale
    sales[nextSaleId] = Sale(nextSaleId, msg.sender, _tokenContract, _amount, _price, true);

    emit SaleCreated(nextSaleId, msg.sender, _amount, _price);
    nextSaleId++;
}

function executeSale(uint256 _saleId) external payable {
    Sale storage sale = sales[_saleId];

    require(sale.active, "Sale is not active");
    require(msg.sender != sale.seller, "Seller cannot buy their own sale");
    require(msg.value == sale.price, "Incorrect Ether value sent");

    sale.active = false;

    // Transfer tokens to buyer
    IERC20(sale.tokenContract).transfer(msg.sender, sale.amount);

    // Transfer ETH payment to seller
    (bool success, ) = sale.seller.call{value: msg.value}("");
    require(success, "Failed to send Ether to seller");

    emit SaleCompleted(_saleId, msg.sender);
}

// ... cancelSale logic ...
}

```

## 4.2. The Backend Layer (Simulation)

The Python backend connects to the Hardhat node using web3.py and the node's RPC URL.

### 4.2.1. simulator.py: The Simulation Engine

This script is the heartbeat of our simulation. It runs in a continuous loop to mimic real-world energy prosumers.

1. It defines a list of "Prosumer" addresses (Hardhat accounts #1 and #2).
2. Every 5-10 seconds, it loops through these prosumers and generates random energy\_generated and energy\_consumed values.
3. If generated > consumed, it calculates the excess\_energy and calls the mint\_tokens function.
4. The mint\_tokens function builds and signs a transaction using the "Owner" (Account #0) private key, calling the mint function on both the EnergyToken and CarbonCreditToken

contracts to send the newly created tokens to the prosumer.

#### **simulator.py (Core Logic Snippet):**

```
def mint_tokens(prosumer_address, amount, contract):
    """Builds, signs, and sends a transaction to call the mint function."""
    try:
        symbol = contract.functions.symbol().call()
        print(f"Minting {amount / (10**18)} {symbol} for {prosumer_address[:10]}...")

        # Build the transaction
        tx = contract.functions.mint(prosumer_address, amount).build_transaction({
            'from': OWNER_ACCOUNT.address,
            'nonce': w3.eth.get_transaction_count(OWNER_ACCOUNT.address),
            'gas': 300000,
            'gasPrice': w3.eth.gas_price
        })

        # Sign the transaction
        signed_tx = w3.eth.account.sign_transaction(tx, private_key=OWNER_PRIVATE_KEY)

        # Send the transaction
        tx_hash = w3.eth.send_raw_transaction(signed_tx.raw_transaction)

        # Wait for the transaction to be mined
        receipt = w3.eth.wait_for_transaction_receipt(tx_hash)
        print(f" -> Minting successful! Tx: {receipt.transactionHash.hex()}")

    except Exception as e:
        print(f" -> An error occurred during minting: {e}")
```

#### **4.2.2. run.py: Backend Server**

This is a simple Flask server that initializes the connection to the blockchain and verifies the contract instances. While not heavily used in this simulation, it serves as the foundation for any future REST API (e.g., providing historical data to the frontend).

### **4.3. The Frontend Layer (User Interface)**

The frontend is a React application built with Vite and styled with Tailwind CSS for a modern, professional appearance.

#### **4.3.1. Wallet Connection (App.jsx & Header.jsx)**

The App.jsx component manages the core state of the application, including the user's account and signer object. The connectWallet function uses ethers.BrowserProvider(window.ethereum) to request access to the user's MetaMask, get their account, and create a signer object, which

is necessary for all subsequent transactions.

#### 4.3.2. Real-Time Balances (Dashboard.jsx)

Once the user is connected, the Dashboard component is rendered. It uses a `useEffect` hook to fetch and display the user's token balances.

- It creates instances of the `EnergyToken` and `CarbonCreditToken` contracts using the signer.
- It calls the `balanceOf(account)` function on both contracts.
- It formats the returned (18-decimal) `BigInt` value using `ethers.formatUnits()` into a readable string.
- It uses `setInterval` to re-fetch the balances every 10 seconds, ensuring the UI updates automatically as the simulator mints new tokens.

#### Dashboard.jsx (Core Logic Snippet):

```
useEffect(() => {
  const fetchBalances = async () => {
    if (signer && account) {
      try {
        // Create ethers contract instances
        const etknContract = new ethers.Contract(...);
        const cctContract = new ethers.Contract(...);

        // Fetch balances for the connected account
        const etknBal = await etknContract.balanceOf(account);
        const cctBal = await cctContract.balanceOf(account);

        // Format and set balances
        setEtknBalance(ethers.formatUnits(etknBal, 18));
        setCctBalance(ethers.formatUnits(cctBal, 18));
      } catch (error) { ... }
    }
  };

  fetchBalances();
  // Set up an interval to refetch balances every 10 seconds
  const interval = setInterval(fetchBalances, 10000);
  return () => clearInterval(interval); // Cleanup
}, [signer, account]);
```

#### 4.3.3. P2P Trading Flow (SellForm.jsx & MarketplaceListings.jsx)

This is the complete trading loop implemented in React.

- **SellForm.jsx:** Provides the UI for a user to sell their tokens. It implements the two-step

ERC-20 approval process:

1. **handleApprove:** Calls the approve() function on the EnergyToken contract, prompting MetaMask.
  2. **handleCreateSale:** Calls the createSale() function on the Marketplace contract, prompting MetaMask again.
- **MarketplaceListings.jsx:** Fetches and displays all active sales.
    1. **fetchSales:** Loops through all sale IDs on the Marketplace contract and displays the active ones.
    2. **handleBuy:** Allows a *different* user to buy a listing. It calls the executeSale(sale.id, { value: priceInWei }) function, passing the required ETH payment with the transaction, which MetaMask prompts the user to confirm.

## 5. Results & Demonstration

The project successfully achieves a full, end-to-end simulation of a P2P energy marketplace. All components work in harmony as demonstrated in the implementation review.

### 5.1. End-to-End System Flow

1. **System Startup:** All servers (Hardhat Node, Flask, Simulator, React) are started.
2. **Simulation:** The simulator.py script begins running, calculating excess energy for test accounts (Account #1 and #2) and successfully minting new \$ETKN and \$CCT tokens to them.
3. **User Connection:** A user opens the React app, connects their MetaMask wallet, and imports one of the test accounts (e.g., Account #1).
4. **Real-Time Data:** The Dashboard component appears, fetches the balances for Account #1, and displays them. The user can watch these balances increase every 10 seconds as the simulator mints new tokens.
5. **Seller Creates Listing:** The user (Account #1) uses the "Sell Form" to Approve and then Create Sale for 10 \$ETKN at a price of 0.1 ETH. The listing appears in the "Marketplace" section.
6. **Buyer Purchases Listing:** The user switches to a different account in MetaMask (e.g., Account #3). They see the listing from Account #1 and click "Buy Now."
7. **Decentralized Trade:** MetaMask prompts Account #3 to confirm the payment of 0.1 ETH. Upon confirmation, the Marketplace smart contract automatically sends the 10 \$ETKN to Account #3 and the 0.1 ETH to Account #1.
8. **UI Update:** The sales listing disappears (as it is no longer active), and the balances for both users are updated.

### 5.2. Key Functionalities

The following screenshots demonstrate the working application:

```

(base) PS C:\Users\grove\Desktop\coding\Blockchain\blockchain> npx hardhat node
Started HTTP and WebSocket JSON-RPC server at http://127.0.0.1:8545/

Accounts
=====

WARNING: Funds sent on live network to accounts with publicly known private keys WILL BE LOST

Account #0: 0xf39fd6e51aad88f6f4ce6ab8827279cfff92266 (10000 ETH)
Private Key: 0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80

Account #1: 0x70997970c51812dc3a010c7d01b50e0d17dc79c8 (10000 ETH)
Private Key: 0x59c6995e998f97a5a0044966f0945389dc9e86dae88c7a8412f4603b6b78690d

Account #2: 0x3c44cdddb6a900fa2b585dd299e03d12fa4293bc (10000 ETH)
Private Key: 0x5de4111afa1a4b94908f83103eb1f1706367c2e68ca870fc3fb9a804cdab365a

Account #3: 0x90f79bf6eb2c4f870365e785982e1f101e93b906 (10000 ETH)
Private Key: 0x7c852118294e51e653712a81e05800f419141751be58f605c371e15141b007a6

Account #4: 0x15d34aaf54267db7d7c367839aaf71a00a2c6a65 (10000 ETH)
Private Key: 0x47e179ec197488593b187f80a00eb0da91f1b9d0b13f8733639f19c30a34926a

Account #5: 0x9965507d1a55bcc2695c58ba16fb37d819b0a4dc (10000 ETH)
Private Key: 0x8b3a350cf5c34c9194ca85829a2df0ec3153be0318b5e2d3348e872092edffba

Account #6: 0x976ea74026e726554db657fa54763abd0c3a0aa9 (10000 ETH)
Private Key: 0x92db14e403b83dfe3df233f83dfa3a0d7096f21ca9b0d6d6b8d88b2b4ec1564e

Account #7: 0x14dc79964da2c08b23698b3d3cc7ca32193d9955 (10000 ETH)
Private Key: 0x4bbb8f5ce3377467afe5d46f804f221813b2bb87f24d81f60f1fcd9f7cbf4356

Account #8: 0x23618e81e3f5cdf7f54c3d65f7fbc0abf5b21e8f (10000 ETH)
Private Key: 0xdbda1821b80551c9d65939329250298aa3472ba22feea921c0cf5d620ea67b97

Account #9: 0xa0ee7a142d267c1f36714e4a8f75612f20a79720 (10000 ETH)
Private Key: 0x2a871d0798f97d79848a013d4936a73bf4cc922c825d33c1cf7073dff6d409c6

Account #10: 0xbcd4042de499d14e55001cbb24a551f3b954096 (10000 ETH)
Private Key: 0xf21af2b2cd398c806f84e317254e0f0b801d0643303237d97a22a48e01628897

Account #11: 0x71be63f3384f5fb98995898a86b02fb2426c5788 (10000 ETH)
Private Key: 0x701b615bbdfb9de65240bc28bd21bbc0d996645a3dd57e7b12bc2bdf6f192c82

Account #12: 0xfabb0ac9d68b0b445fb7357272ff202c5651694a (10000 ETH)

```

```

(base) PS C:\Users\grove\Desktop\coding\Blockchain\blockchain>

```

```

DeployModule#CarbonCreditToken - 0x5FbDB2315678afecb367f032d93F642f64180aa3
DeployModule#EnergyToken - 0xe7f1725E7734CE288F8367e1Bb143E90bb3F0512
DeployModule#Marketplace - 0x9fE46736679d2D9a65F0992F2272dE9f3c7fa6e0

```

```

DeployModule#CarbonCreditToken - 0x5FbDB2315678afecb367f032d93F642f64180aa3

```

```

DeployModule#CarbonCreditToken - 0x5FbDB2315678afecb367f032d93F642f64180aa3
DeployModule#EnergyToken - 0xe7f1725E7734CE288F8367e1Bb143E90bb3F0512
DeployModule#Marketplace - 0x9fE46736679d2D9a65F0992F2272dE9f3c7fa6e0

```

```

(base) PS C:\Users\grove\Desktop\coding\Blockchain\blockchain>

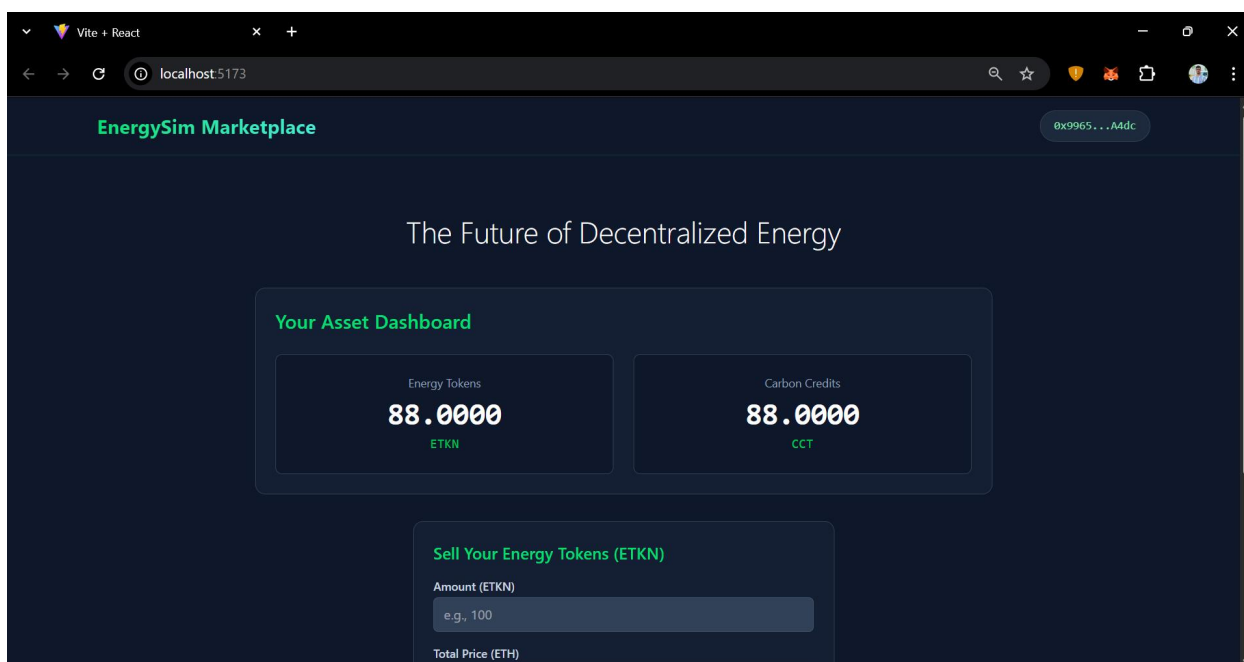
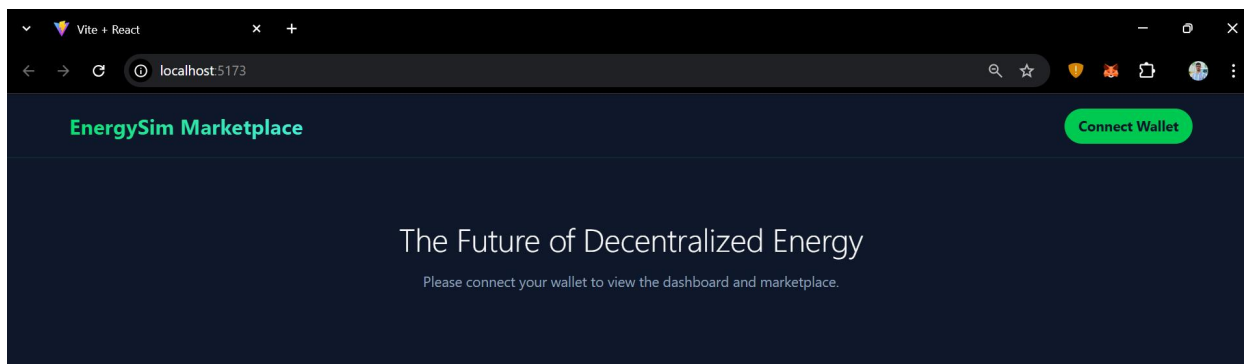
```

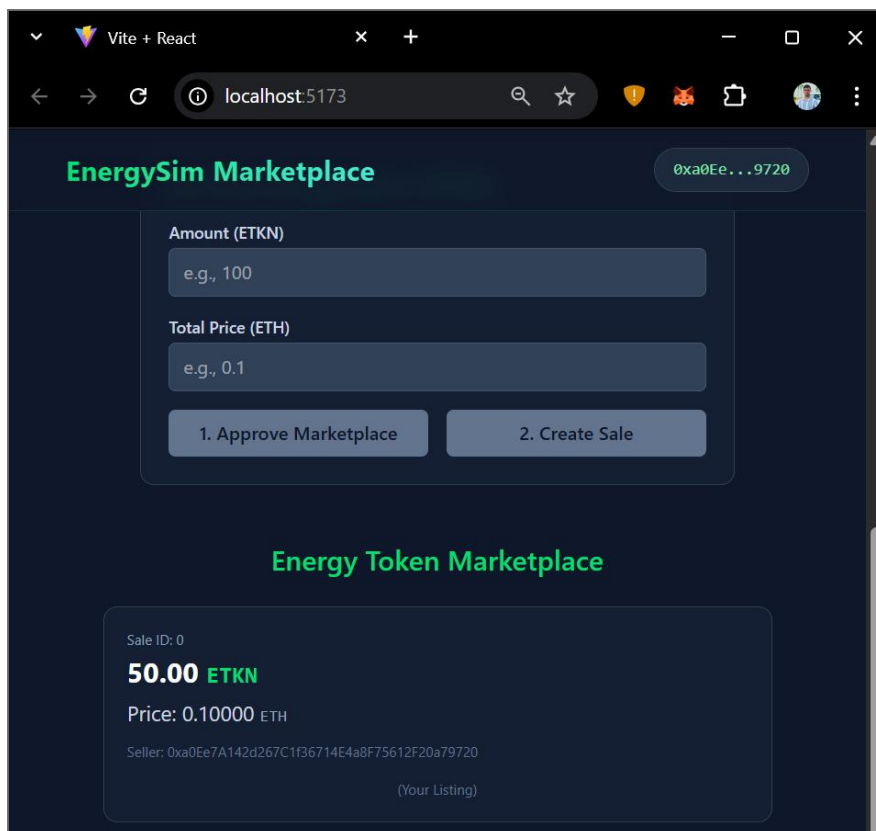
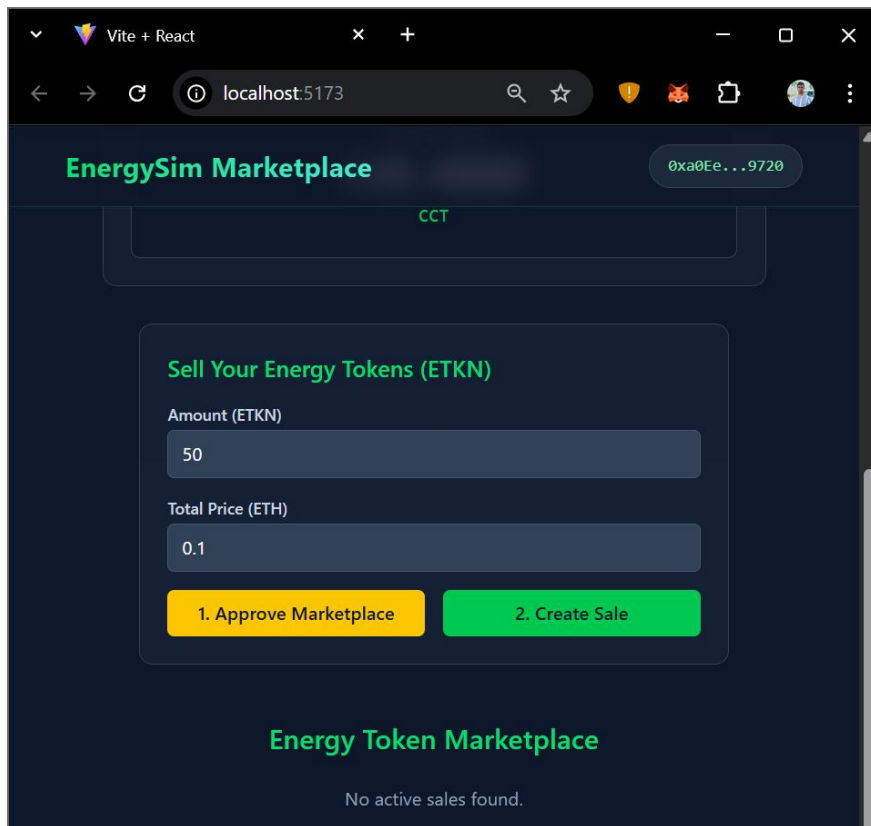
```

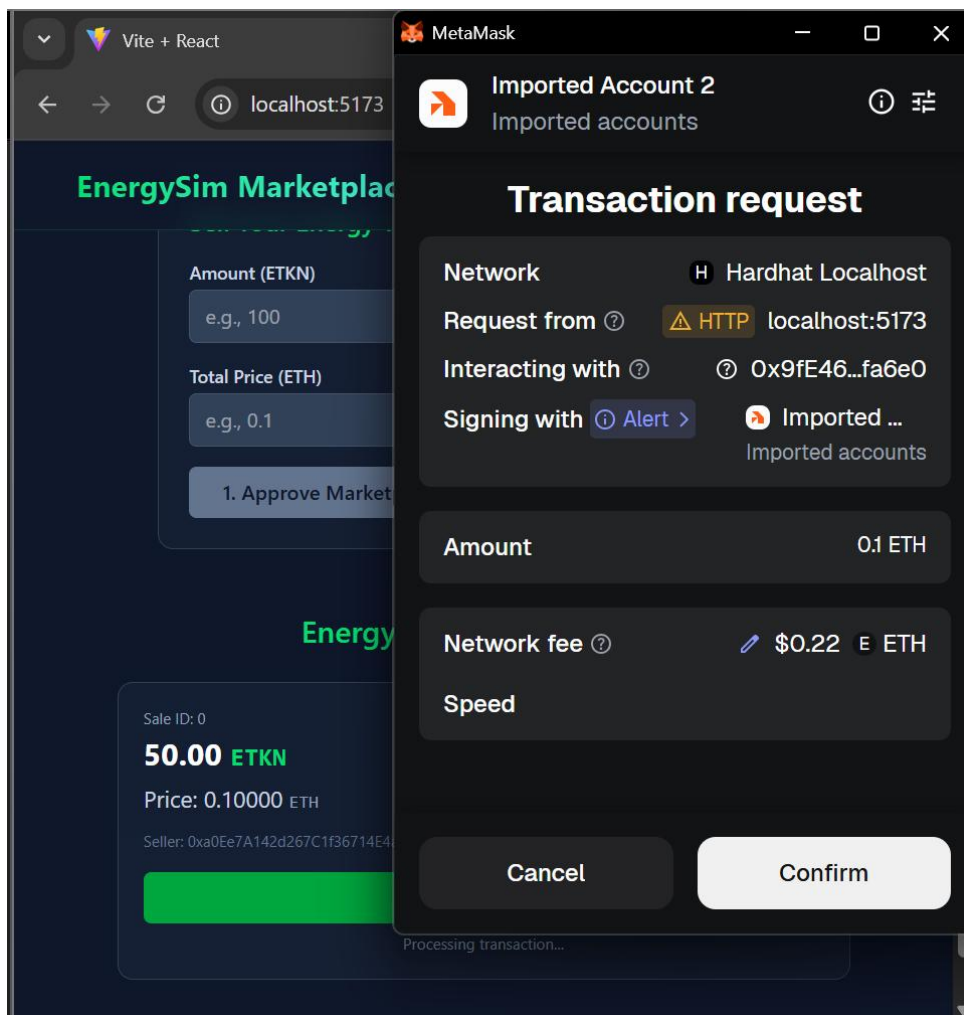
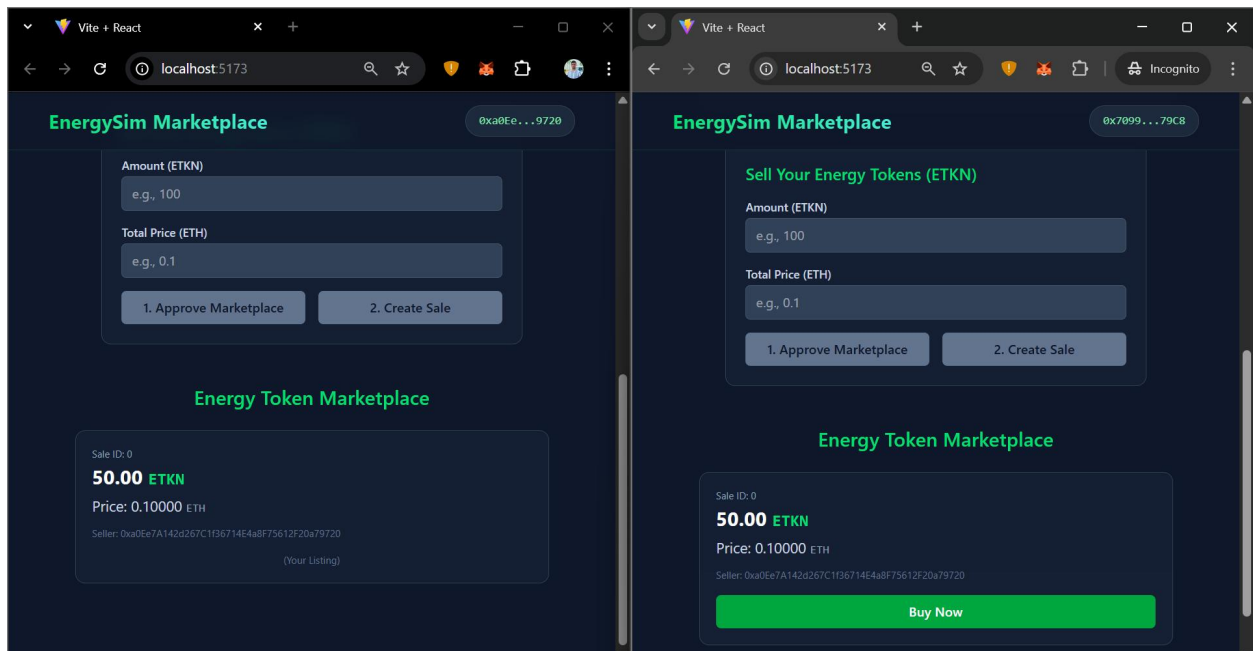
(base) PS C:\Users\grove\Desktop\coding\Blockchain> cd backend
(base) PS C:\Users\grove\Desktop\coding\Blockchain\backend> python simulator.py
Simulator connecting to blockchain...
Simulator connected.

--- Running simulation cycle ---
Prosumer 0xf39Fd6e5... | Generated: 17 kWh | Consumed: 7 kWh
-> Excess energy: 10 kWh. Minting tokens...
Minting 10.0 ETKN for 0xf39Fd6e5...
-> Minting successful! Tx: 1a574c19079a28d48f3e70b1c70faf489521e9ccfd8b0a3722d18efad89ac3ac
Minting 10.0 CCT for 0xf39Fd6e5...
-> Minting successful! Tx: 841d0dfbd16e56974d95cbf03d17ea349c5a69ed932d239b6f25c1357ad1573b
Prosumer 0x70997970... | Generated: 10 kWh | Consumed: 4 kWh
-> Excess energy: 6 kWh. Minting tokens...
Minting 6.0 ETKN for 0x70997970...
-> Minting successful! Tx: 639891e8f579b65c6c85fa548464704be670b1f8d92d3db3207aac839f88741c
Minting 6.0 CCT for 0x70997970...
-> Minting successful! Tx: 20cea942b48037a62391393303edf9b206a35be9fa5acb87a726ae21d7aa5e2a

```







## 6. Challenges Faced

Several technical challenges were encountered and overcome during development:

1. **Frontend Build Configuration:** The initial setup of the React (Vite) frontend with Tailwind CSS was complex. We faced several configuration errors related to postcss and npm executables, which were resolved by manually creating the configuration files (tailwind.config.js, postcss.config.js) and ensuring the correct PostCSS compatibility packages (@tailwindcss/postcss) were installed.
2. **Blockchain Desynchronization:** The most significant challenge was a persistent code=BAD\_DATA error in the frontend. This error indicated that the frontend was trying to call contracts at addresses that were empty. We diagnosed this as a synchronization issue between the newly-deployed contract addresses (after a Hardhat node restart) and the static addresses stored in our backend (.env) and frontend (contractConfig.js) configuration. This was solved by implementing a strict startup routine: **Node -> Deploy -> Update All Configs -> Start Servers.**
3. **web3.py Library Versioning:** The simulator.py script initially failed with errors related to the SignedTransaction object. This was due to a breaking change between web3.py library versions (v5 vs. v6), where the raw transaction data attribute was renamed from signed\_tx.rawTransaction to signed\_tx.raw and back again in different versions. We resolved this by ensuring a stable, up-to-date version of web3.py and using the correct signed\_tx.raw\_transaction attribute.

## 7. Conclusion & Future Work

This project successfully demonstrates the viability of a decentralized P2P energy trading system. By leveraging Solidity smart contracts, we created a trustless, transparent, and autonomous marketplace. The React frontend provides a modern, user-friendly interface for wallet interaction, and the Python backend serves as a robust simulation engine.

The core objectives of tokenizing energy, rewarding prosumers, and enabling P2P trading were all met. The final application proves the end-to-end flow of a simulated decentralized economy.

Future work on this project could expand in several exciting directions:

- **Integration with Real IoT Devices:** Replace the Python simulator with an API that reads data from real-world smart meters.
- **AI-Based Dynamic Pricing:** Integrate an AI model to suggest optimal pricing in the "Sell Form" based on real-time grid demand, weather, and time of day.
- **Carbon Credit NFTs:** Evolve the \$CCT from a fungible ERC-20 token to a Non-Fungible Token (NFT, ERC-721), where each token represents a unique, verifiable batch of carbon credits from a specific source and time, increasing its traceability and value.
- **Layer-2 Scaling:** Deploy the contracts on a Layer-2 scaling solution (like Polygon, Arbitrum, or Optimism) to dramatically reduce gas fees and increase transaction speed for a real-world application.

## 8. References

- [Research Papers Drive Link](#) (Used in Review 1)
- [Solidity Documentation](#)
- [React Documentation](#)
- [Ethers.js Documentation](#)
- [Hardhat Documentation](#)
- [OpenZeppelin Contracts Documentation](#)
- [MetaMask Documentation](#)