ASTANA IT UNIVERSITY

Blockchain Technology 1

Final Project

Group: SE-2419, SE-2420

Authors: Aibek Nazarbek, Sergei Nurtilek, Beisenbek Nazly

Date: 12 February 2026

Astana, 2026

Contents

Abstract

This report presents the design, development, implementation, and evaluation of a decentralized crowdfunding platform built on the Ethereum blockchain. The system integrates Solidity smart contracts, Hardhat development framework, automated testing, gas optimization techniques, CI/CD workflows, and an Angular-based frontend interface. The purpose of the project is to demonstrate practical application of blockchain technology in decentralized finance (DeFi) systems and crowdfunding ecosystems. Specifically, "GreenPulse" aims to create a platform for environmental projects, rewarding contributors with "LEAF" tokens.

The platform enables users to create campaigns, contribute cryptocurrency (ETH), monitor funding progress, and securely withdraw funds through smart contract-controlled logic. The architecture ensures transparency, immutability, and trust minimization without relying on centralized intermediaries.

This document provides an in-depth explanation of system architecture, contract design, testing methodology, gas analysis, frontend integration, security considerations, scalability challenges, and future enhancements.

Introduction

Crowdfunding has become a widely adopted method of raising capital for startups, social initiatives, and creative projects. Traditional crowdfunding platforms rely on centralized intermediaries that manage funds, enforce rules, and charge service fees. However, centralized systems introduce trust issues, single points of failure, censorship risks, and potential misuse of funds.

Blockchain technology introduces decentralization, transparency, and immutability. Smart contracts can automate fundraising logic without requiring third-party control [1]. This project explores the implementation of a decentralized crowdfunding system using Ethereum smart contracts and a modern web interface.

The main goal of this project is to design and implement a secure, scalable, and efficient blockchain-based crowdfunding platform that demonstrates both theoretical knowledge and practical development skills.

Literature Review and Theoretical Background

Blockchain is a distributed ledger technology that ensures transparency, immutability, and decentralized consensus [2]. Ethereum extends blockchain functionality by supporting programmable smart contracts written in Solidity.

Smart contracts are self-executing programs stored on the blockchain. They automatically enforce rules defined in code, eliminating the need for intermediaries.

Crowdfunding models include:

- Donation-based crowdfunding
- Reward-based crowdfunding
- Equity-based crowdfunding
- Debt-based crowdfunding

This project focuses on donation-based crowdfunding, where contributors send funds without expecting financial return.

Ethereum transaction execution requires gas. Gas represents computational cost and ensures network security. Efficient smart contract design reduces gas consumption and improves user experience.

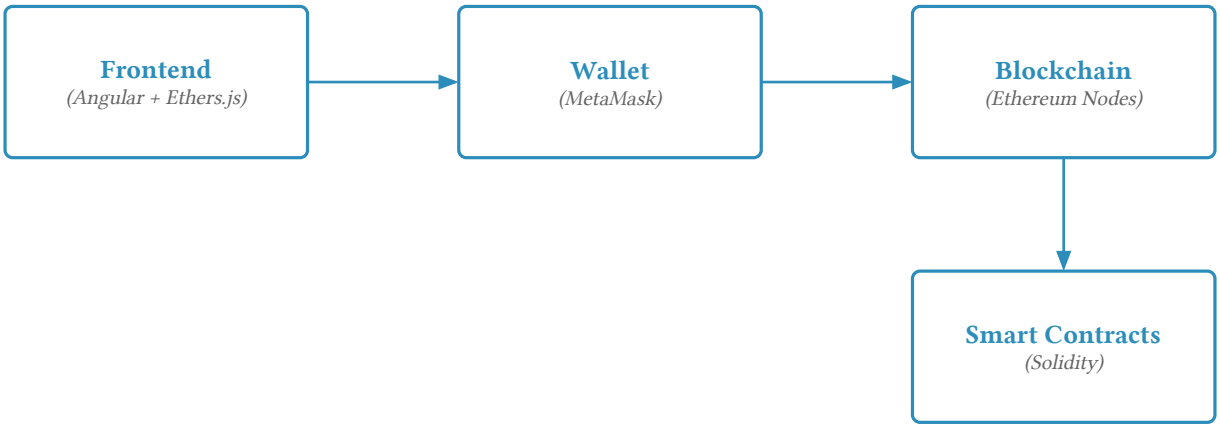
System Architecture

The system consists of three primary layers:

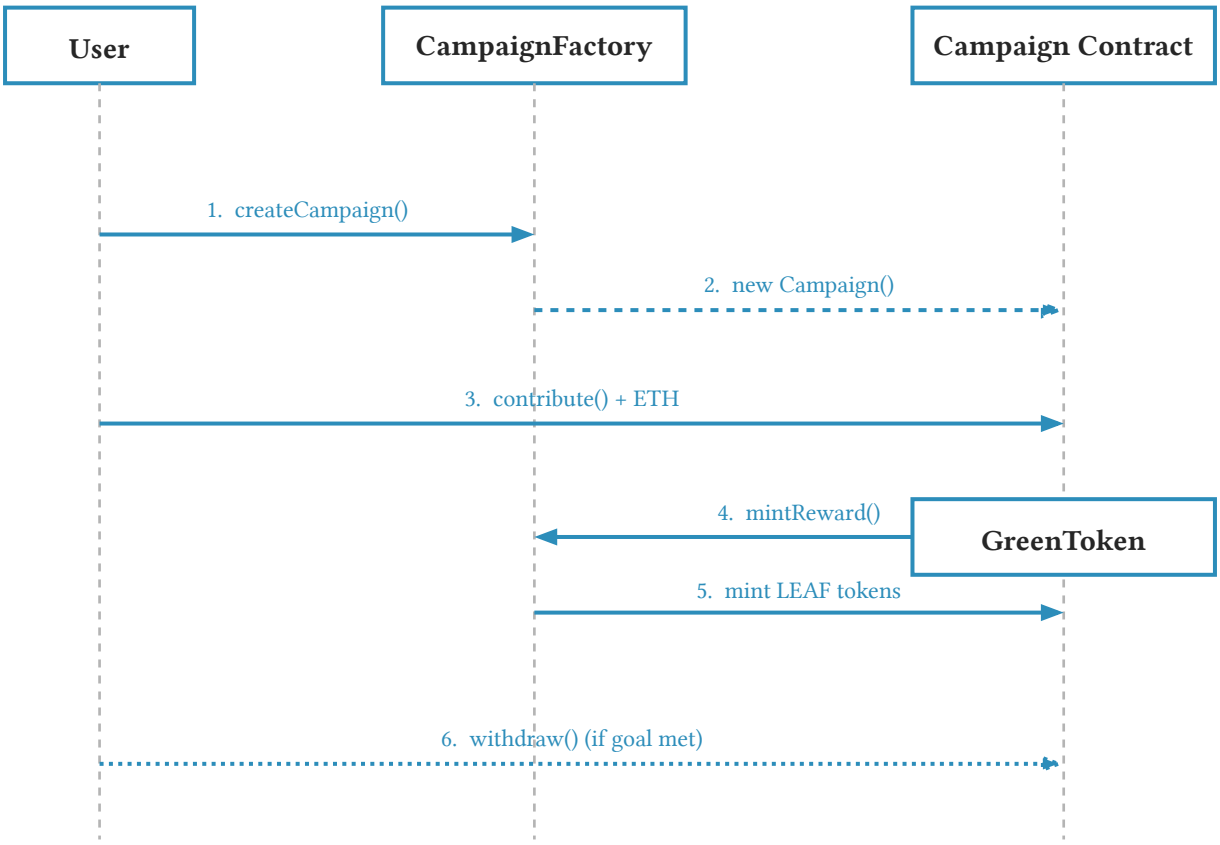
1. **Smart Contract Layer (Solidity)**
   - `Campaign.sol`
   - `CampaignFactory.sol`
   - `GreenToken.sol`
   - `Lock.sol`
2. **Development and Testing Layer**
   - Hardhat framework
   - Mocha & Chai testing
   - Gas reporting
   - Ignition deployment modules
3. **Frontend Layer**
   - Angular (TypeScript) with **Angular Signals** for state management
   - Web3 integration using `ethers.js`
   - MetaMask wallet connection
   - Tailwind CSS & **Material Tailwind** UI framework

The architecture follows modular design principles. Each campaign is deployed as an independent contract, ensuring scalability and isolation.

System Architecture Diagram



Interaction Flow



Sequence Description:

1. User calls the Factory to create a new campaign.
2. Factory deploys an independent Campaign contract.
3. Users contribute ETH directly to the Campaign.
4. Campaign triggers GreenToken logic to notify Factory.
5. Factory mints LEAF tokens to contributor.
6. Creator withdraws funds after goal is reached.

Smart Contract Design

The smart contract architecture is divided into three main components, designed for modularity and security.

Campaign Contract (`Campaign.sol`)

The `Campaign` contract is the core of the fundraising logic. Each campaign is a separate instance of this contract. **Key Functionalities:**

- **Contribution Management:** The `contribute()` function accepts ETH and updates the `contributions` mapping. It automatically mints 100 reward tokens per 1 ETH contributed by calling the factory contract.
- **Withdrawal Logic:** The `withdraw()` function allows the campaign creator to transfer raised funds only if the funding goal is met. It uses `ReentrancyGuard` to prevent reentrancy attacks.
- **Refund Mechanism:** If a campaign fails to meet its goal by the deadline, contributors can call `refund()` to reclaim their ETH.
- **State Retrieval:** The `getSummary()` function returns a tuple of all critical campaign statistics in a single call, optimizing frontend data fetching.

```solidity
// Campaign.sol partial code
contract Campaign is ReentrancyGuard {
    function contribute() external payable nonReentrant {
        require(block.timestamp < deadline, "Campaign has ended");
        require(msg.value > 0, "Contribution must be greater than 0");

        contributions[msg.sender] += msg.value;
        raisedAmount += msg.value;

        // Reward 100 tokens per 1 ETH
        uint256 rewardAmount = msg.value * 100;

        try factory.mintReward(msg.sender, rewardAmount) {} catch {}

        emit ContributionMade(msg.sender, msg.value);

        if (raisedAmount ⩾ goal) {
            deadline = block.timestamp;
            emit GoalReached(raisedAmount);
            emit CampaignEnded(block.timestamp);
        }
    }
}
```

Campaign Factory (`CampaignFactory.sol`)

To ensure scalability and trust, users do not deploy `Campaign` contracts directly. Instead, they use the `CampaignFactory` . **Responsibilities:**

- Deploys new instances of `Campaign` using the `new` keyword.
- Maintains a registry of all deployed campaigns (`Campaign[] public campaigns`).
- Acts as the central authority for minting `GreenToken` rewards, ensuring only valid campaigns can issue tokens.

```Solidity
// CampaignFactory.sol
contract CampaignFactory is Ownable {
    Campaign[] public campaigns;
    GreenToken public token;
    mapping(address => bool) public isCampaign;

    function createCampaign(uint256 _goal, uint256 _duration, string memory _title, string memory _description) external {
        Campaign newCampaign = new Campaign(msg.sender, _goal, _duration, address(this), _title, _description);
        campaigns.push(newCampaign);
        isCampaign[address(newCampaign)] = true;
        emit CampaignCreated(address(newCampaign), msg.sender, _goal, _title);
    }
}
```

Green Token (`GreenToken.sol`)

This contract implements the ERC-20 standard using OpenZeppelin's library [3].

- **Symbol:** LEAF
- **Role:** It serves as a reward token for contributors.
- **Access Control:** Only the `CampaignFactory` (owner) is authorized to mint new tokens when a valid contribution occurs.

contributors receive **LEAF** tokens as rewards for their "Green Impact".

Security Measures:

- **Reentrancy Protection:** `ReentrancyGuard` from OpenZeppelin is applied to `contribute` , `withdraw` , and `refund` functions.
- **Checks-Effects-Interactions Pattern:** Utilized in withdrawal functions to prevent state inconsistencies.
- **Access Control:** `Ownable` modifier restricts sensitive administrative functions in the Factory and Token contracts.

Development Methodology

The project followed iterative development methodology:

- **Phase 1:** Smart Contract Design
- **Phase 2:** Unit Testing
- **Phase 3:** Gas Optimization
- **Phase 4:** Frontend Integration (Migration from React to Angular)
- **Phase 5:** Deployment & CI Configuration

Hardhat was used for compilation, deployment, and testing. Automated testing ensures contract reliability and prevents regressions.

Testing and Validation

Testing was performed using Mocha and Chai frameworks within Hardhat environment.

Test cases included:

- Campaign creation verification
- Contribution validation and reward minting checks
- Withdrawal permission checks (creator only, goal reached check)
- **Time-shifting tests:** Using `evm_increaseTime` to simulate the passage of time and verify that `refund` functions work correctly only after the deadline.
- Edge cases (zero contribution, unauthorized withdrawal)
- Full workflow simulation

Comprehensive testing improves system reliability and demonstrates professional development standards.

Gas Usage and Optimization

Gas efficiency is crucial for user adoption and contract sustainability. The following specific optimizations were implemented:

1. **Mapping for Contributions:** Instead of iterating through an array of contributors to track balances (which would cost $O(n)$ gas), a `mapping(address ⇒ uint256)` is used. This ensures that lookups and updates always cost constant $O(1)$ gas, regardless of the number of contributors.
2. **External Visibility:** Functions meant to be called from outside the contract (like `contribute`, `withdraw`, `getSummary`) are declared as `external` instead of `public`. `external` functions can read arguments directly from calldata, which is cheaper than copying them to memory.
3. **Batched Data Retrieval:** The `getSummary()` function in `Campaign.sol` returns multiple state variables (minimum contribution, balance, requests count, approvers count, manager) in a single function call. This significantly reduces the number of JSON-RPC calls the frontend must make, improving perceived performance and reducing network load, although it doesn't strictly save execution gas on writes.
4. **Memory Variables:** In loops or complex calculations, state variables are cached in `memory` to avoid expensive storage `SLOAD` operations.
5. **Error Strings:** Short, concise error strings are used in `require` statements to slightly reduce deployment bytecode size.
6. **Compiler Configuration:** The project uses Solidity `0.8.28` with the optimizer enabled (200 runs). The `evmVersion` is set to `paris` in `hardhat.config.cjs` to ensure compatibility across different Ethereum-compatible networks and avoid `PUSH0` opcode tracing bugs common in older tooling versions.

Frontend Implementation & Blockchain Communication

The Angular frontend serves as the gateway for users to interact with the Ethereum blockchain. Communication is handled primarily through `ethers.js`, which bridges the gap between standard web protocols and Ethereum's JSON-RPC interface.

1. The Communication Bridge (Ethers.js & MetaMask)

The DApp uses `ethers.BrowserProvider` to tap into the `window.ethereum` object injected by MetaMask. This setup allows the application to:

- Read state from the blockchain without user permission.
- Request account access for signing transactions.
- Listen for network and account changes.

```typescript
// Initialization in web3.service.ts
private async initReadOnly() {
  if (typeof window !== 'undefined' && (window as any).ethereum) {
    const provider = new ethers.BrowserProvider((window as any).ethereum);
    this.provider.set(provider);

    // Create a read-only instance of the Factory
    const factory = new ethers.Contract(
      this.factoryAddress,
      CampaignFactoryArtifact.abi,
      provider
    );
    this.factoryContract.set(factory);
  }
}
```

2. Provider vs. Signer

A critical distinction in Web3 development is the difference between a **Provider** and a **Signer**:

- **Provider:** A read-only connection to the blockchain. It can fetch balances, block data, and call `view` functions.
- **Signer:** An abstraction of an Ethereum account. It has the ability to sign messages and send transactions (which cost gas).

3. Executing Smart Contract Operations

A. Read Operations (Call)

Read operations do not require gas or user confirmation. They are used to fetch campaign details using the `provider`.

```typescript
// Reading campaign data (0 gas cost)
```

```
const campaign = new ethers.Contract(addr, CampaignArtifact.abi, provider);
const summary = await campaign['getSummary']();
return {
  goal: summary[1],
  raisedAmount: summary[2],
};
```

B. Write Operations (Transaction)

Write operations modify the blockchain state, require gas fees, and must be signed by the user via MetaMask. The DApp uses the `signer` to execute these.

```typescript
// Sending a contribution (Transaction)
const signer = await provider.getSigner(); // Request permission to sign
const contract = new ethers.Contract(address, CampaignArtifact.abi, signer);

// 1. Transaction is sent to MetaMask for approval
const tx = await contract['contribute']({
  value: parseEther(amount.toString()),
});

// 2. Wait for the transaction to be mined (included in a block)
await tx.wait();
```

4. Reactive State Management (Angular Signals)

To ensure the UI updates immediately when the blockchain state changes (e.g., after a wallet connects or a transaction finishes), the project utilizes **Angular Signals**.

Unlike traditional variables, Signals notify all dependent UI components when their value changes.

- `public account: WritableSignal<string | null> = signal(null);`
- `public tokenBalance: WritableSignal<string> = signal('0');`

When `fetchBalances()` updates the `tokenBalance` signal, the Navbar automatically re-renders the new LEAF token count without a full page refresh.

MetaMask integration enables wallet connection and transaction signing. The UI displays campaign progress, contribution forms, and real-time updates.

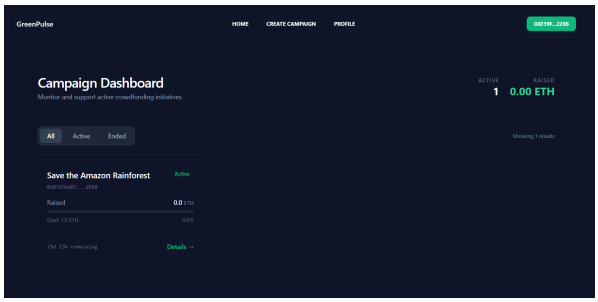


Figure 1: GreenPulse Home Page - Campaign Selection

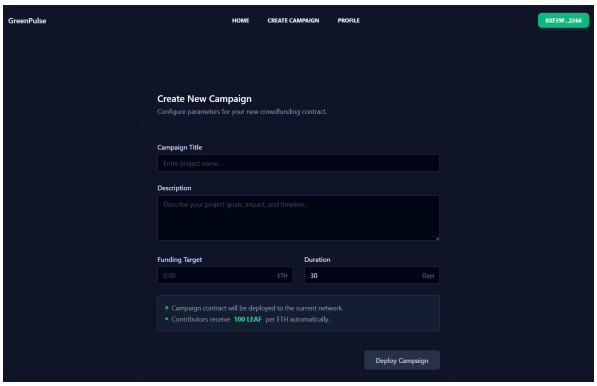


Figure 2: Campaign Creation Interface

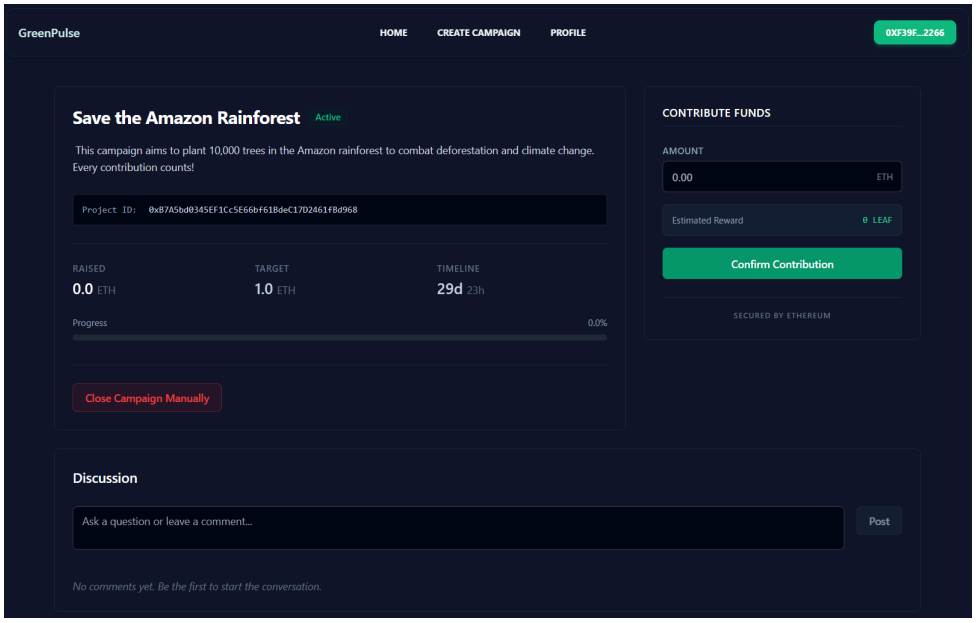


Figure 3: Detailed Campaign View with Contribution tracking

Security Considerations

Security is critical in blockchain systems.

Potential vulnerabilities:
- Reentrancy attacks
- Integer overflow/underflow
- Access control misconfiguration
- Front-running attacks

Mitigation strategies:
- Solidity ^0.8 built-in overflow protection
- Proper modifier usage
- Minimal external calls
- Testing for edge cases

Deployment and DevOps

Deployment scripts were implemented using Hardhat.

CI pipeline via GitHub Actions ensures:

- Automated testing
- Continuous integration
- Code stability verification

Deployment process:

1. Compile contracts
2. Deploy to local network
3. Verify deployment
4. Run tests

Performance and Scalability Analysis

Scalability considerations include:

- High gas fees during network congestion
- Blockchain throughput limitations
- Need for Layer-2 solutions

Future scalability solutions:

- Optimistic Rollups
- zk-Rollups
- Sidechains
- Polygon integration

Future Improvements

Planned enhancements:

- Governance voting system for fund management
- IPFS integration for decentralized storage of campaign media
- Multi-network deployment (e.g., L2 support)
- Backend indexing via The Graph to improve search capabilities

Discussion

The project demonstrates integration of blockchain backend and web frontend technologies. It highlights the importance of testing, gas optimization, and security awareness in decentralized systems.

The modular architecture ensures extensibility. The system can serve as foundation for real-world DeFi or Web3 fundraising platforms.

Conclusion

This project successfully implements a decentralized crowdfunding platform using Ethereum smart contracts and Angular frontend.

It demonstrates:

- Smart contract engineering
- Secure blockchain development
- Full-stack Web3 integration
- Automated testing practices
- Gas optimization awareness

The system provides a strong academic and practical example of modern decentralized application development.

Bibliography

[1] Ethereum Foundation, “Introduction to Smart Contracts.” [Online]. Available: https://ethereum.org/en/developers/docs/smart-contracts/

[2] I. Bashir, *Blockchain Consensus: An Introduction to Classical, Blockchain, and Quantum Consensus Protocols*. Apress, 2020.

[3] OpenZeppelin, “OpenZeppelin Contracts Documentation.” [Online]. Available: https://docs.openzeppelin.com/contracts/