**BCSE498J Project-II - Capstone Project**

**BLOCKFUND – WEB3-POWERED DECENTRALIZED CROWDFUNDING**

21BCI0234 NIKET SUCHAK

21BCI0313 KARTIK GUPTA

21BCI0360 VANDIT SHARMA

Under the Supervision of

**Dr. MANOOV R**

Assistant Professor Sr. Grade 1

School of Computer Science and Engineering (SCOPE)

**B.Tech.**

*in*

**Computer Science and Engineering**

**(with specialization in Information Security)**

**School of Computer Science and Engineering**

\

April, 2025

#### DECLARATION

I hereby declare that the **<Project-II >** entitled “BlockFund - Web3-Powered decentralized Crowdfunding " submitted by us, for the award of the degree of *Bachelor of Technology in Computer Science (with specialization in information security), School of Computer Science and Engineering* to VIT is a record of Bonafide work carried out by us under the supervision of Dr. Manoov R., Assistant Professor Grade 1, SCORE, VIT, Vellore>.

I further declare that the work reported in this Project-II has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university.

Place: Vellore

Date: 14/04/2025

**Signature of the Candidate**

#### CERTIFICATE

This is to certify that the Project entitled “BlockFund - Web3-Powered Crowdfunding” submitted by **Vandit Sharma (21BCI0360), Niket Suchak (21BCI0234) , Kartik Gupta (21BCI0313)**, SCORE, VIT, for the award of the degree of *Bachelor of Technology in Computer Science and Engineering* , is a record of Bonafide work carried out by him under my supervision during the period, 13. 12. 2024 to 23.04.2025, as per the VIT code of academic and research ethics.

The contents of this report have not been submitted and will not be submitted either in part or in full, for the award of any other degree or diploma in this institute or any other institute or university. The project fulfils the requirements and regulations of the University and in my opinion meets the necessary standards for submission.

Place: Vellore Date:

##### Signature of the VIT-SCOPE - Guide

**Internal Examiner External Examiner**

**Head of the Department**

**Department of Information Technology**

### ACKNOWLEDGEMENT

It is my pleasure to express with a deep sense of gratitude to my **Project- II** guide **<Dr. Manoov R. , Assistant Professor Grade 1>,** School of Computer Science Engineering and Information Systems, Vellore Institute of Technology, Vellore for **his** constant guidance, continual encouragement, in my endeavour. My association with **him** is not confined to academics only, but it is a great opportunity on my part to work with an intellectual and an expert in the field of Blockchain Technology**.**

I would like to express my heartfelt gratitude to Honourable Chancellor **Dr. G Viswanathan**; respected Vice Presidents **Mr. Sankar Viswanathan**, **Dr. Sekar Viswanathan**, Vice Chancellor **Dr. V. S. Kanchana Bhaaskaran**; Pro-Vice Chancellor **Dr. Partha Sharathi Mallick**; and Registrar **Dr. Jayabarathi T**.

My whole-hearted thanks to Dean **Dr. Daphne Lopez**, School of Computer Science Engineering and Information Systems, Head, Department of Information Technology **Dr.**

**Prabhavathy P**, B. Tech Project Coordinator **Dr. Suganya P**, SCORE School Project Coordinator **Dr. Thandeeswaran R**, all faculty, staff and members working as limbs of our university for their continuous guidance throughout my course of study in unlimited ways.

It is indeed a pleasure to thank my parents and friends who persuaded and encouraged me to take up and complete my **project-II** successfully. Last, but not least, I express my gratitude and appreciation to all those who have helped me directly or indirectly towards the successful completion of the **project-II**.

Place: Vellore

Date: 14/04/2025 **Vandit Sharma**

## Executive Summary

BlockFund is a decentralized crowdfunding platform built on the Ethereum blockchain to offer a secure, transparent, and intermediary-free fundraising experience. The platform leverages smart contracts written in Solidity and integrates seamlessly with MetaMask for wallet connectivity and user authentication. By eliminating centralized control, BlockFund ensures direct interaction between campaign creators and contributors, empowering users to manage funds with full transparency.

Users can launch crowdfunding campaigns by specifying key details like funding goals, deadlines, and descriptions. Contributors can fund these campaigns directly through their wallets, with all transactions handled via secure Ethereum smart contracts. Funds are held in escrow and released only when milestone conditions are met, while refund mechanisms automatically return contributions if goals aren't achieved, ensuring trust on both ends.

The frontend is developed using React.js and styled with Tailwind CSS, offering a clean and intuitive user interface. Campaign data can be optionally stored using decentralized services like IPFS to enhance data integrity. The platform also features real-time campaign progress, contributor tracking, and a seamless donation flow.

Currently 80% complete, BlockFund is tested locally using Ganache and progressing toward full deployment on Ethereum. It offers a scalable, user-centric solution for global crowdfunding with blockchain-backed accountability.

**CONTENTS Page No.**

**Acknowledgement** v

**Executive Summary** vi

**Table of Contents** vii

**List of Figures** ix

**List of Tables** x

**Abbreviations** xi

**Symbols and Notations** xii

1. **INTRODUCTION 13**
   1. Objective 13
   2. Motivation 14
   3. Background 23
2. **PROJECT-II DESCRIPTION AND GOALS 24**
3. **TECHNICAL SPECIFICATION 25**
4. **DESIGN APPROACH AND DETAILS (as applicable)**  27
   1. Design Approach / Materials & Methods 27
   2. Codes and Standards 29
   3. Constraints, Alternatives and Tradeoffs 46
5. **SCHEDULE, TASKS AND MILESTONES** 46
6. **PROJECT-II DEMONSTRATION 48**
   1. Sample Codes 48
   2. Sample Screen Shots 53
7. **SUMMARY** 56

61

1. **REFERENCES**

**APPENDIX A**

### List of Figures

**Figure No. Title Page No.**

4.1.1 Use Case 17

4.1.2 System 18

Architecture

4.1.3 Class Diagram 19

6.2 Sample 29

Screenshots

### List of Tables

|  |  |  |  |
| --- | --- | --- | --- |
| **Table No.** |  | **Title** | **Page No.** |
| 5.1 | Gantt Chart |  | 22 |

**List of Abbreviations**

|  |  |
| --- | --- |
| **dApp**: | Decentralized Application |
| **ETH**: | Ether (Ethereum's native cryptocurrency) |
| **IPFS**: | InterPlanetary File System |
| **UI**: | User Interface |
| **API**: | Application Programming Interface |
| **ORM**: | Object-Relational Mapping |
| **CSS**: | Cascading Style Sheets |
| **CLI**: | Command Line Interface |
| **SQL**: | Structured Query Language |

### Symbols and Notations

**Ξ**: Symbol for Ether (ETH), used in transactions on the Ethereum blockchain

**→**: Represents the flow of funds or data (e.g., from contributor to campaign)

**#**: Denotes a unique identifier for campaigns or transactions (e.g., campaign ID)

**{}**: Represents smart contract logic or JSON-like data structures in the codebase

**@**: Used in code (e.g., Tailwind CSS directives like @tailwind, or TypeScript decorators)

#### CHAPTER 1 INTRODUCTION

Crowdfunding has revolutionized the way individuals and organizations raise capital for creative, entrepreneurial, social, or charitable initiatives. Traditional funding methods often create barriers for new or small-scale ventures due to complex procedures, high interest rates, and the need for intermediaries such as banks or venture capitalists. Crowdfunding platforms emerged to democratize this process by allowing a large number of people to contribute small amounts toward a larger funding goal. Despite their success, these platforms face limitations, including centralization, high fees, transparency issues, and control over fund distribution.

BlockFund addresses these challenges by introducing a decentralized, blockchain-powered crowdfunding solution. By building on the Ethereum blockchain and leveraging smart contracts, BlockFund eliminates intermediaries, ensures transparent fund management, and provides users with full control over their contributions and campaigns. This introduction of Web3 technologies into the crowdfunding space aligns with the broader movement toward decentralized finance (DeFi) and trustless digital interactions.

1.1 Objective

The primary objective of BlockFund is to create a transparent, secure, and decentralized crowdfunding platform that empowers users through smart contract technology. It aims to:

* Facilitate direct transactions between campaign creators and contributors without third-party control.
* Offer a secure, immutable system where rules are predefined and automatically enforced.
* Build a user-friendly platform that bridges the gap between complex blockchain interactions and non-technical users.
* Support transparency and accountability through on-chain fund tracking and milestone-based disbursement.

1.2 Motivation

The idea for BlockFund stems from the increasing inefficiencies and trust-related issues in traditional crowdfunding platforms and financial fundraising systems. In today’s interconnected and digitally-enabled economy, crowdfunding has become a popular mechanism for individuals and organizations to raise capital from a large number of contributors. However, the very platforms that made crowdfunding possible have slowly evolved into centralized gatekeepers—levying high service fees, enforcing rigid policies, and lacking in financial transparency. This growing disconnect between contributors and campaign creators has inspired the development of an alternative model: one that leverages blockchain to create a decentralized, transparent, and trustless fundraising environment.

1.2.1 Centralization Issues in Traditional Crowdfunding

Most popular crowdfunding platforms today—Kickstarter, GoFundMe, Indiegogo, and others—operate under a centralized business model. While convenient, this model creates several critical issues:

* Intermediary Control: The platform controls the rules, enforces policies, and holds the funds. In many cases, platforms delay fund disbursement or freeze accounts due to perceived policy violations.
* Service Fees: These platforms charge between 5–12% in platform and payment processing fees, drastically reducing the final amount campaigners receive.
* Opaque Operations: Contributors often have no insight into how funds are used. Campaign creators, after receiving the funds, are not bound to prove milestone completion or actual utilization.
* Geographical Limitations: Certain platforms only allow creators from specific countries, restricting participation in global fundraising efforts.
* Censorship and Bias: Campaigns can be removed or hidden due to political, religious, or social bias—contradicting the principle of openness in public fundraising.

These limitations create a system that is not only inefficient but also disempowering, especially for users from underserved regions or those working on unconventional or politically sensitive causes.

1.2.2 The Promise of Blockchain and Decentralization

Blockchain technology offers a radically different approach to handling money, data, and trust. Ethereum, in particular, enables the use of smart contracts—self-executing code stored on a blockchain that runs automatically when certain conditions are met. This allows the creation of decentralized applications (dApps) that operate without a central authority, offering transparency, immutability, and verifiability.

For crowdfunding, this translates into a system where:

* The rules are enforced not by a platform but by code.
* Funds are held in escrow until project milestones are verifiably completed.
* Contributors can view transactions on-chain and hold campaigners accountable.
* Anyone with internet access and a Web3 wallet can participate or raise funds.

BlockFund embraces this philosophy by using Ethereum smart contracts to build a trustless system where no single entity can tamper with campaign rules or restrict fund flow arbitrarily.

1.2.3 Motivating Scenarios

Several real-world situations illustrate the need for such a system:

a) Delayed Fund Access in Emergencies:

Imagine a social activist starting a campaign to raise funds for a medical emergency. On centralized platforms, verification, approval, and fund disbursement can take days—even weeks. In contrast, a blockchain-based platform like BlockFund can release funds automatically upon reaching the required threshold without bureaucratic intervention. b) Misuse of Funds:

There have been instances where creators received funds but failed to deliver promised results—leading to a loss of contributor trust. With smart contracts, fund release can be tied to verifiable milestones, like uploading photos of progress or passing predefined deadlines.

1. Contribution Tracking:

In traditional systems, contributors rarely receive updates after contributing. With BlockFund, every contribution and fund movement is publicly recorded on-chain, giving contributors confidence in transparency and accountability.

1. Platform Censorship:

Numerous campaigns on centralized platforms have been removed due to political sensitivity or being flagged by bots. BlockFund ensures censorship resistance—no authority can delete or block a campaign once it’s deployed on the blockchain.

1.2.4 Empowering Global Participation

One of the foundational motivations behind BlockFund is to democratize access to capital. Traditional crowdfunding platforms, while revolutionary in the early 2010s, have often failed to deliver on their promise of universal access. These systems, built atop traditional finance rails, introduce constraints based on geography, banking infrastructure, regulatory frameworks, and currency systems. In contrast, BlockFund, by leveraging blockchain and decentralized identities, opens the door for a truly global and borderless crowdfunding system.

This section explores how BlockFund empowers global participation and eliminates financial and systemic roadblocks for underserved, unbanked, or excluded populations around the world.

⸻

1.2.4.1 The Global Financial Divide

Over 1.4 billion adults globally remain unbanked according to the World Bank (2021), with the highest concentrations in developing countries across Africa, South Asia, and Latin America. Many of these individuals, while financially excluded, own mobile phones and have access to the internet. Traditional banking systems, often gatekept by documentation requirements, minimum balances, or regulatory barriers, prevent these populations from participating in global financial systems.

Challenges faced by these users:

* No access to credit cards or payment gateways like PayPal or Stripe
* High remittance fees (~6–10% average in Africa)
* Currencies subject to hyperinflation or government seizure
* Crowdfunding platforms restricted to a few ‘supported countries’
* Risk of censorship or platform bans due to political affiliation or activism

BlockFund circumvents all these issues by providing an Ethereum-based fundraising system that only requires:

* An internet connection
* A browser or mobile dApp browser
* A free Ethereum wallet like MetaMask

There are no restrictions based on geography, language, or political status. This enables users from Nigeria, Bangladesh, Argentina, Ukraine, or any part of the world to raise funds, contribute to campaigns, and track transactions—all without needing a traditional bank account.

⸻

1.2.4.2 Wallet-Based Participation Over Identity Barriers

Traditional platforms require:

* Email-based registration
* Government-issued ID verification (KYC)
* Linking a valid credit/debit card or bank account

This inherently excludes:

* Refugees and displaced individuals
* Stateless persons without documentation
* Activists operating under political surveillance
* Youth and students without banking access

BlockFund uses wallet-based login—eliminating the need for personal identification altogether. A wallet address becomes your public identity. Funds are sent and received based on wallet ownership, verified cryptographically via MetaMask or similar tools.

This form of self-sovereign identity enables anonymous but secure participation.

Benefits:

* No user profiling or data collection
* No surveillance on campaign intent
* Full control over assets and actions
* Censorship-resistant access

Such a model is especially relevant in authoritarian regimes where financial activity can be criminalized or weaponized.

⸻

1.2.4.3 Use Cases Across Geographies

BlockFund’s borderless architecture supports a wide spectrum of use cases in both urban and rural communities across the globe. Below are examples of real-world scenarios where such a system can be life-changing:

1. Rural Healthcare in Africa:

A rural clinic in Uganda needs $500 in equipment repairs. Traditional banking or international crowdfunding is inaccessible. BlockFund allows the clinic to deploy a verified campaign, receive funds from global supporters, and withdraw via local P2P ETH-to-cash exchanges.

1. Flood Relief in Bangladesh:

During a flood crisis, a local group quickly launches a BlockFund campaign. Contributors worldwide send ETH directly to the smart contract. Funds are disbursed only as goals are met (e.g., tents acquired, food distributed). Photos are uploaded via IPFS as proof.

1. Educational Support in Latin America:

A student in Venezuela wants to crowdsource $1,000 to buy a laptop and software to study coding. Hyperinflation renders local currency useless. BlockFund enables crypto contributions from supporters abroad. The funds are traceable, secured, and instantly accessible.

4. Political Advocacy in Southeast Asia:

Activists raising funds for legal defense or campaign awareness face bans on centralized platforms. With BlockFund, campaigns are hosted on-chain, censorship-resistant, and donations are anonymized via decentralized wallets.

These examples demonstrate that global participation is not just a feature but a lifeline for inclusion and empowerment.

⸻

1.2.4.4 Eliminating Cross-Border Barriers

Cross-border payments remain one of the most expensive and inefficient segments of the global financial system. Traditional options involve:

* Intermediaries like SWIFT
* Conversion fees
* Delays of up to 3–5 business days
* Regulatory compliance layers

For small fundraising campaigns, these mechanisms are not only expensive but infeasible.

In contrast, Ethereum transactions:

* Are executed globally within ~10–15 seconds
* Cost a predictable gas fee
* Require no intermediaries or settlement layers

BlockFund campaigns, once deployed, are visible to anyone with an Ethereum node or dApp interface. Users in Germany, India, Canada, or South Africa can fund a campaign in seconds, with verifiable on-chain proof. This dramatically lowers the barrier to trust and time—both essential for time-sensitive causes like medical emergencies or natural disasters.

⸻

1.2.4.5 Multi-Language and Multicultural Accessibility

To empower participation beyond English-speaking or tech-savvy audiences, BlockFund is designed to support:

* Multilingual campaign descriptions using Unicode and auto-translation tools
* Visual feedback and icons that guide users in UX even without fluent reading skills
* Simple onboarding guides to install MetaMask, get test ETH (for testing), and launch campaigns

This multilingual and multicultural sensitivity is crucial for:

* Local NGOs
* Grassroots community organizers
* Minority language speakers

By focusing on universal accessibility, BlockFund removes the elitist gatekeeping of modern fintech platforms.

⸻

1.2.4.6 Transparency as a Tool for Accountability

In centralized crowdfunding, contributors have no insight into where the money goes after the goal is reached. They rely on updates (if any) and have no recourse if the creator fails to deliver. In contrast, BlockFund enforces transparency by design:

* All contributions are logged on-chain
* Campaign status (active, completed, failed) is stored immutably
* Milestone-based release ensures funds aren’t prematurely used
* Refunds are processed automatically when campaigns fail

This system is invaluable in countries where corruption or mismanagement of charity funds is widespread. International contributors can monitor the flow of funds, check on deliverables, and feel confident about their support being used effectively.

⸻

1.2.4.7 Towards Financial Sovereignty

BlockFund is more than a tool—it’s a statement. A statement that individuals, regardless of nationality, background, or economic status, deserve the right to raise capital, support causes, and receive funds—all without waiting for permission from a centralized platform or institution.

With blockchain:

* You are your own bank.
* You are your own gatekeeper.
* You are your own campaign manager.

BlockFund reflects this philosophy by enabling permissionless participation in one of the most empowering human experiences: supporting and building something together.

Conclusion

Empowering global participation is not just a technical milestone for BlockFund—it is the very soul of the platform. In a world divided by borders, platforms, and financial privilege, BlockFund opens a door for equality. It levels the playing field by letting ideas —not identity or geography—drive support. By removing gatekeepers, decentralizing trust, and enabling transparency, BlockFund ensures that no worthy cause is ever left unheard due to systemic exclusion.

As the world shifts toward decentralized digital infrastructure, tools like BlockFund will become essential for enabling cross-border cooperation, trustless philanthropy, and inclusive innovation.

1.2.5 Aligning with the Web3 Vision

Web3 is the vision of a decentralized, user-owned internet—where users control their data, identity, and financial transactions. BlockFund aligns perfectly with this ethos by:

* Using wallet-based authentication instead of centralized login systems.
* Putting campaign logic in immutable smart contracts.
* Allowing full visibility and traceability of every rupee/ETH contributed.
* Giving contributors and creators ownership of their interaction records.

In a world where large corporations increasingly dominate the web and financial landscape, tools like BlockFund give back control to individuals and communities.

1.2.6 Learning and Adoption Motivation

From an academic and technical perspective, building BlockFund is a deeply motivating challenge. It blends:

* Blockchain development (Solidity, Ethereum, smart contracts)
* Frontend engineering (React.js, Tailwind CSS, MetaMask APIs)
* Backend logic and security (Node.js, Express, PostgreSQL)
* Design thinking (User onboarding, trust signals, mobile accessibility)

It allows the developer to contribute to the emerging decentralized ecosystem while solving a relevant real-world problem. Moreover, it provides a hands-on opportunity to learn about gas optimization, transaction finality, decentralized storage, and wallet interoperability.

1.2.7 Building Trust in Digital Fundraising

In a post-pandemic world where online interactions dominate, digital trust is more important than ever. BlockFund builds trust not through policy documents or legal disclaimers, but through transparent code, public audit logs, and deterministic execution of promises.

Trust is established through:

* Verified wallet interactions
* Unchangeable campaign logic
* Automatic refunds for failed campaigns
* Publicly accessible contract state

1.3 Background

Blockchain, often described as a digital ledger, enables peer-to-peer transactions without centralized oversight. Ethereum enhances this model by supporting smart contracts— programs that execute automatically when certain conditions are met. This innovation allows for the creation of decentralized applications that can replace traditional intermediaries in finance, governance, and more.

MetaMask, a popular Ethereum wallet browser extension, simplifies user interaction with dApps by providing secure authentication and transaction signing. BlockFund integrates this tool to make blockchain-based crowdfunding intuitive and accessible.

#### CHAPTER 2 PROJECT DESCRIPTION AND GOALS

BlockFund serves as a decentralized fundraising solution that empowers individuals, startups, and organizations to launch campaigns directly on the blockchain. It removes intermediaries from the process and provides mechanisms to enforce campaign milestones, refund contributors when goals are unmet, and provide real-time updates to backers.

The architecture of BlockFund is divided into modular components—frontend, backend, smart contracts, and decentralized storage. This modularity supports maintainability, scalability, and the integration of new features in the future.

Functional Highlights:

* Campaign Creation: Users can set title, description, goals, and deadlines.
* Contributor Interaction: Users can fund campaigns securely via Ethereum and MetaMask.
* Milestone-Based Fund Disbursement: Funds are released in stages based on campaign progress.
* Automatic Refunds: Contributions are returned if campaigns don’t meet their objectives.
* Real-Time Tracking: Contributors can monitor funding progress and history.
* Security: Built-in verification mechanisms prevent fraud or misuse.

Project Goals

1. Design an intuitive user interface for campaign management and contribution.
2. Implement secure and auditable smart contracts to govern fund logic.
3. Integrate MetaMask for wallet authentication and transaction execution.
4. Use decentralized storage like IPFS for campaign-related metadata (optional).
5. Provide a robust backend for logging activity and managing non-sensitive data.
6. Optimize system performance by reducing redundant calls and minimizing gas consumption.
7. Deploy and test the application on both local environments (e.g., Ganache) and public testnets.
8. Document codebase, architecture, and user workflow to enable open-source adoption.

#### CHAPTER 3 TECHNICAL SPECIFICATION

The BlockFund platform is a full-stack decentralized application built on the Ethereum blockchain with a focus on modular architecture, transparency, and user-friendliness. The platform comprises the following major components:

Frontend

Technology used is React.js with Vite as the build tool. Styling is done using Tailwind CSS for responsive and modern UI. Wallet integration is achieved through MetaMask for user authentication and transaction signing. Routing is managed using React Router, and React state/hooks are used for user and session management. Additional libraries include Ethers.js for blockchain interaction, Framer Motion for animations, and Axios or TanStack Query for API communication.

Smart Contracts

Smart contracts are written in Solidity. Development and testing are done using Hardhat. The main contracts are CampaignFactory, which creates and manages multiple campaigns, and Campaign, which handles contributions, goal tracking, milestone-based fund release, and refunds. Security is ensured through escrow mechanisms, conditional fund releases, and automatic refunds if goals are not met. Ganache CLI is used for local testing, and the final deployment target is the Ethereum mainnet.

Backend

The backend is built using Node.js with Express. TypeScript is used for type safety and maintainable code. PostgreSQL is used as the database, connected using the pg library. drizzle- orm is used as the ORM for schema-safe SQL queries. While wallet-based identity is used primarily, optional session management is available through express-session. Environment variables are managed using a .env file for sensitive configuration. Vite and Express are connected through a custom server configuration file.

DevOps and Tooling

Version control is handled using Git and GitHub. npm is used for package management. Development scripts include npm run dev to start the Vite and Express server, and npm run build to build the client and server.

Optional Features

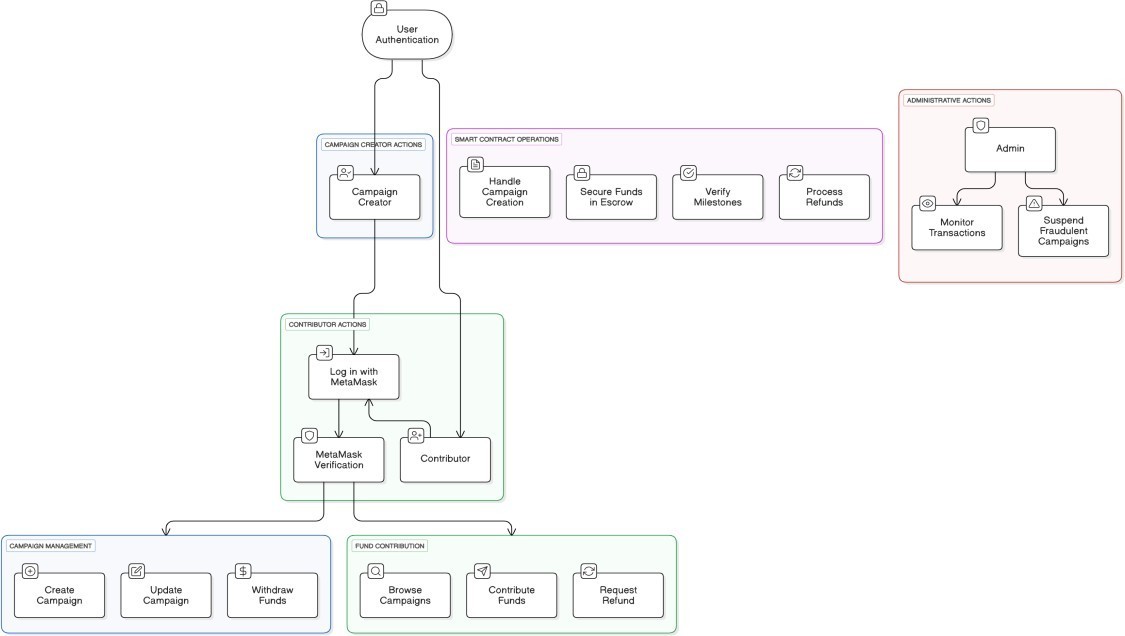
The platform plans to integrate IPFS or Filecoin for decentralized file storage. Client-side filtering is implemented to sort campaigns by goal, deadline, or popularity. Users can view a history of their contributions based on their wallet address. Real-time funding progress is displayed using polling or on-chain event listeners

.

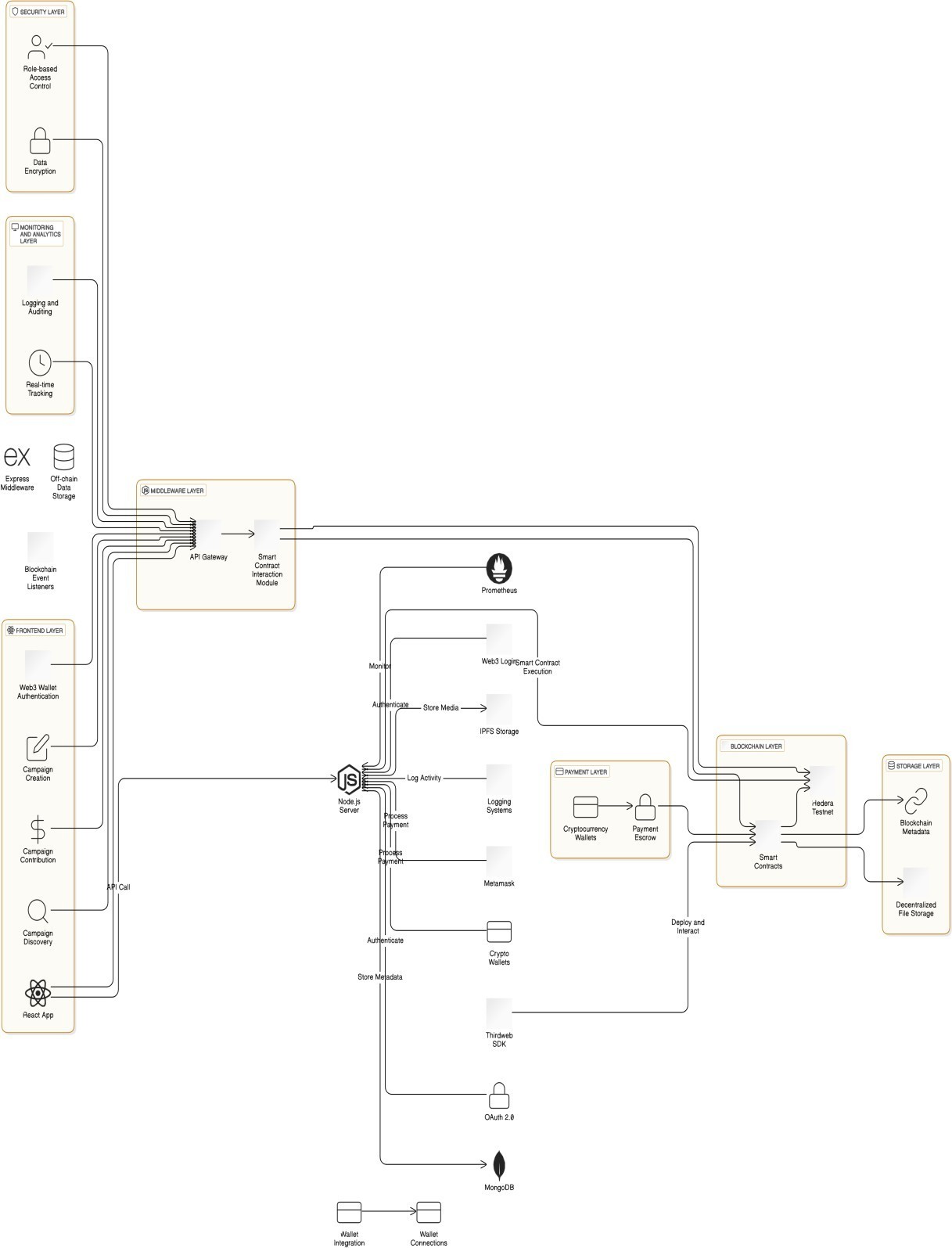
#### CHAPTER 4 DESIGN APPROACH AND DETAILS

4.1 Design Approach and Methods

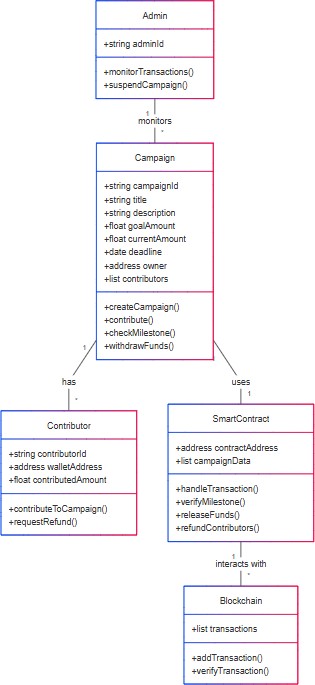
The design approach for BlockFund follows a modular and component-based architecture. The frontend is built using React.js, which allows for reusable components and efficient UI rendering. Tailwind CSS is used to ensure a clean and responsive design. The smart contracts are written in Solidity and follow a separation-of-concerns principle, with a factory contract handling campaign creation and individual campaign contracts managing logic and transactions. MetaMask is integrated to allow users to connect their wallets and interact with the Ethereum blockchain. The backend supports API services and data persistence through PostgreSQL and drizzle-orm. The system is designed to be scalable, testable using Ganache, and ready for deployment on Ethereum mainnet.



4.1.1 Use Case Diagram



4.1.2 System Architecture



4.1.3 Class Diagram

4.2 Codes and Standards

The project adheres to Web3 and Ethereum development best practices. Smart contracts are written using Solidity following the Ethereum Smart Contract Security Best Practices guidelines. Web development follows ECMAScript standards and TypeScript conventions. RESTful API principles are applied on the backend. PostgreSQL is used with structured and schema-based data modeling. MetaMask integration uses standard EIP-1193 and EIP-1102 interfaces for wallet interaction. Frontend code is structured according to React and Vite configuration standards.

App.txs import { Switch, Route } from "wouter"; import { Toaster } from "@/components/ui/toaster"; import NotFound from "@/pages/not-found"; import Home from "@/pages/Home"; import CampaignDetail from "@/pages/CampaignDetail"; import CreateCampaign from "@/pages/CreateCampaign"; import HowItWorks from "@/pages/HowItWorks"; import Header from "@/components/Header"; import Footer from "@/components/Footer"; import { Web3Provider } from "@/lib/web3";

function Router()

{ return (

<Switch>

<Route path="/" component={Home} />

<Route path="/campaigns/create" component={CreateCampaign} />

<Route path="/campaigns/:id" component={CampaignDetail} />

<Route path="/how-it-works" component={HowItWorks} />

<Route component={NotFound} />

</Switch>

);

}

import { ethers } from "ethers";

import { useWeb3Store, SUPPORTED\_NETWORKS } from "./web3"; import { create } from "zustand";

import { VoteOption, ContractInfo } from "@shared/schema"; import { toast } from "@/hooks/use-toast";

// Contract address (same on both networks) const CONTRACT\_ADDRESS = "0x4bdE3ca6f319274D531298Ea189cF857C1bb95bc";

// Blockchain explorer URLs by network ID

export const EXPLORER\_URLS: Record<number, string> = {

1: "https://etherscan.io",

11155111: "https://sepolia.etherscan.io" };

// ABI for the voting contract const CONTRACT\_ABI = [

// Functions

"function createCampaign(uint256[] options, uint256 startTime, uint256 endTime)",

"function createCategory(uint256 categoryId, uint256 weight)",

"function registerVoter(uint256 categoryId, address voter)",

"function vote(uint256 campaignId, uint256 optionIndex)",

"function renounceOwnership()",

"function transferOwnership(address newOwner)",

// View Functions

"function campaigns(uint256) view returns (uint256 startTime, uint256 endTime)",

"function categories(uint256) view returns (uint256 weight, bool exists)",

"function getCampaignResults(uint256 campaignId) view returns (uint256[])",

"function hasVoted(uint256, address) view returns (bool)",

"function nextCampaignId() view returns (uint256)",

"function owner() view returns (address)",

"function totalWeightedVotes(uint256, uint256) view returns (uint256)",

"function voters(address) view returns (uint256 categoryId, bool isRegistered)",

// Events

"event OwnershipTransferred(address indexed previousOwner, address indexed newOwner)"

];

// Contract state interface interface ContractState { contract: ethers.Contract | null; options: VoteOption[]; contractInfo: ContractInfo; selectedOptionId: number | null; isVoting: boolean; txHash: string | null; hasVoted: boolean; isLoading: boolean; errorMessage: string | null; initializeContract: () => Promise<void>; loadContractData: () => Promise<void>; selectOption: (optionId: number) => void; vote: () => Promise<void>; refreshResults: () => Promise<void>; checkIfVoted: () => Promise<void>;

createCampaign: (options: number[], startTime: number, endTime: number) => Promise<void>;

registerVoter: (categoryId: number, voterAddress: string) => Promise<void>; createCategory: (categoryId: number, weight: number) => Promise<void>; }

// Contract store with Zustand

export const useContractStore = create<ContractState>((set, get) => ({ contract: null, options: [],

contractInfo: {

address: CONTRACT\_ADDRESS,

networkId: 1,

networkName: "Ethereum Mainnet", totalVotes: 0, isOwner: false, isRegistered: false,

voterCategory: null

},

selectedOptionId: null, isVoting: false, txHash: null, hasVoted: false, isLoading: false, errorMessage: null,

initializeContract: async () => { try {

const { provider, signer, chainId, networkName, account } = useWeb3Store.getState();

if (!provider || !signer) {

throw new Error("No provider or signer available");

}

// Create contract instance with signer for write operations

const contract = new ethers.Contract(

CONTRACT\_ADDRESS, CONTRACT\_ABI,

signer

);

// Reset error message

set({ contract, errorMessage: null,

isLoading: true

}); try {

// Verify contract exists by calling a simple view function const nextCampaignId = await contract.nextCampaignId();

// Check if current user is the owner const ownerAddress = await contract.owner(); const isOwner = account && account.toLowerCase() === ownerAddress.toLowerCase();

// Update contract info with current network and contract data set({ contractInfo: {

address: CONTRACT\_ADDRESS,

networkId: chainId || 1,

networkName: networkName || "Ethereum Mainnet", totalVotes: 0,

nextCampaignId: Number(nextCampaignId),

isOwner: Boolean(isOwner), isRegistered: false, // Will be updated in loadContractData voterCategory: null // Will be updated in loadContractData

}

});

// Load contract data after verification await get().loadContractData(); await get().checkIfVoted(); } catch (verifyError: any) {

console.error("Contract verification error:", verifyError);

const errorMsg = "Smart contract not found or not compatible with this network. Please switch networks or verify contract address."; set({

errorMessage: errorMsg,

isLoading: false

});

toast({ title: "Contract Error", description: errorMsg,

variant: "destructive",

});

}

} catch (error: any) {

console.error("Contract initialization error:", error); set({

errorMessage: error.message || "Failed to initialize contract", isLoading: false

});

}

},

loadContractData: async () => {

try {

const { contract, contractInfo } = get();

const { account } = useWeb3Store.getState();

if (!contract) {

throw new Error("Contract not initialized");

}

if (!contractInfo.nextCampaignId) {

throw new Error("Contract info not initialized");

}

set({ isLoading: true });

// Get active campaigns

// The contract stores campaigns starting from ID 1 const campaignId = Number(contractInfo.nextCampaignId) - 1;

if (campaignId <= 0) { // No campaigns yet set({ options: [],

isLoading: false

}); return;

}

try {

// Get campaign details

const campaign = await contract.campaigns(campaignId);

// Get campaign results

const results = await contract.getCampaignResults(campaignId); // Create option objects with mock names (since the contract doesn't store names) const options: VoteOption[] = [];

let totalVotes = 0;

// Calculate total votes for (let i = 0; i < results.length; i++) { const votes = Number(results[i]); totalVotes += votes;

}

// Create options array for (let i = 0; i < results.length; i++) { const votes = Number(results[i]);

const percentage = totalVotes > 0 ? Math.round((votes / totalVotes) \* 100) : 0;

options.push({

id: i,

name:  Option ${i + 1} ,

description:  Voting option ${i + 1} for campaign ${campaignId} , votes: votes,

percentage: percentage

});

}

// Check if voter is registered and what category they belong to let voterInfo = null; if (account) { try {

voterInfo = await contract.voters(account);

} catch (error) {

console.log("User not registered as voter:", error);

}

}

// Check if the current account is the owner of the contract let isOwner = false;

try { if (account) {

const ownerAddress = await contract.owner();

isOwner = account.toLowerCase() === ownerAddress.toLowerCase();

}

} catch (error) {

console.error("Error checking contract owner:", error);

}

set({ options: options, contractInfo: { ...contractInfo, totalVotes: totalVotes, currentCampaignId: campaignId, campaignStartTime: Number(campaign.startTime), campaignEndTime: Number(campaign.endTime), voterCategory: voterInfo ? Number(voterInfo.categoryId) : null, isRegistered: voterInfo ? Boolean(voterInfo.isRegistered) : false, isOwner: isOwner

},

isLoading: false

});

} catch (error) {

console.error("Error loading campaign data:", error); set({ options: [],

isLoading: false

});

}

} catch (error: any) {

console.error("Loading contract data error:", error); set({

errorMessage: error.message || "Failed to load contract data", isLoading: false

});

toast({

title: "Error loading data",

description: error.message || "Failed to load contract data", variant: "destructive",

});

}

},

selectOption: (optionId: number) => {

set({ selectedOptionId: optionId });

},

vote: async () => { try {

const { contract, selectedOptionId, contractInfo } = get(); const { account, chainId } = useWeb3Store.getState();

if (!contract) {

throw new Error("Contract not initialized");

}

if (selectedOptionId === null) {

throw new Error("No option selected");

}

if (!account) {

throw new Error("Wallet not connected");

}

if (!contractInfo.currentCampaignId) { throw new Error("No active campaign found");

}

// Check if the user is registered as a voter if (!contractInfo.isRegistered) {

throw new Error("You are not registered as a voter. Please contact the administrator to register.");

}

// Check if campaign is active

const now = Math.floor(Date.now() / 1000); // Current time in seconds if (contractInfo.campaignStartTime && now < contractInfo.campaignStartTime) { throw new Error("Voting has not started yet");

}

if (contractInfo.campaignEndTime && now > contractInfo.campaignEndTime) { throw new Error("Voting has ended");

}

// Check if we're on a supported network

if (!SUPPORTED\_NETWORKS.some(network => network.id === chainId)) {

throw new Error( Network not supported. Please switch to $

{SUPPORTED\_NETWORKS.map(n => n.name).join(' or ')} );

}

set({ isVoting: true, errorMessage: null });

try {

// Send vote transaction with campaignId and optionIndex

const tx = await contract.vote(contractInfo.currentCampaignId, selectedOptionId); set({ txHash: tx.hash });

// Wait for transaction confirmation

await tx.wait();

// Refresh data after successful vote await get().loadContractData();

await get().checkIfVoted();

toast({

title: "Vote submitted successfully!",

description:  Transaction hash: ${tx.hash.substring(0, 10)}... ,

});

} catch (txError: any) {

console.error("Transaction error:", txError); let errorMessage = "Failed to submit vote. ";

// Check for common contract revert reasons if (txError.message.includes("execution reverted")) { errorMessage += "The transaction was rejected by the contract. Possible reasons include: already voted, voting closed, or invalid option.";

} else if (txError.message.includes("user rejected")) { errorMessage = "Transaction was rejected in your wallet."; } else if (txError.message.includes("insufficient funds")) {

errorMessage += "You don't have enough ETH to cover the transaction fee.";

} else {

errorMessage += txError.message;

}

set({ errorMessage });

toast({ title: "Voting Error", description: errorMessage,

variant: "destructive",

});

}

set({ isVoting: false }); } catch (error: any) {

console.error("Voting error:", error);

set({

errorMessage: error.message || "Failed to submit vote", isVoting: false

});

toast({

title: "Voting Error",

description: error.message || "Failed to submit vote", variant: "destructive",

});

}

},

refreshResults: async () => {

await get().loadContractData();

toast({

title: "Results refreshed",

description: "The latest voting results have been loaded",

});

},

checkIfVoted: async () => {

try {

const { contract, contractInfo } = get();

const { account } = useWeb3Store.getState();

if (!contract || !account || !contractInfo.currentCampaignId) { set({ hasVoted: false });

return;

} try {

// hasVoted takes campaignId and address parameters in this contract

const hasVoted = await contract.hasVoted(contractInfo.currentCampaignId, account); set({ hasVoted: Boolean(hasVoted) });

} catch (error) {

// Silently handle this error as it's not critical to the UI

// Only log to console for debugging

console.error("Error checking vote status:", error);

// Default to false to allow users to attempt to vote set({ hasVoted: false });

}

} catch (error) {

console.error("Error in checkIfVoted function:", error); set({ hasVoted: false });

}

},

createCampaign: async (options, startTime, endTime) => { try {

const { contract } = get();

const { account, chainId } = useWeb3Store.getState();

if (!contract) {

throw new Error("Contract not initialized");

}

if (!account) {

throw new Error("Wallet not connected");

}

// Check if we're on a supported network

if (!SUPPORTED\_NETWORKS.some(network => network.id === chainId)) {

throw new Error( Network not supported. Please switch to $

{SUPPORTED\_NETWORKS.map(n => n.name).join(' or ')} );

}

set({ isVoting: true, errorMessage: null });

try {

// Send create campaign transaction

const tx = await contract.createCampaign(options, startTime, endTime); set({ txHash: tx.hash });

// Wait for transaction confirmation

await tx.wait();

// Refresh data after successful campaign creation await get().loadContractData();

toast({

title: "Campaign created successfully!",

description:  Transaction hash: ${tx.hash.substring(0, 10)}... ,

});

} catch (txError: any) { console.error("Transaction error:", txError); let errorMessage = "Failed to create campaign. ";

// Check for common contract revert reasons if (txError.message.includes("execution reverted")) { errorMessage += "The transaction was rejected by the contract. Possible reasons include: invalid parameters or insufficient permissions.";

} else if (txError.message.includes("user rejected")) { errorMessage = "Transaction was rejected in your wallet."; } else if (txError.message.includes("insufficient funds")) {

errorMessage += "You don't have enough ETH to cover the transaction fee.";

} else {

errorMessage += txError.message;

}

set({ errorMessage });

toast({

title: "Campaign Creation Error", description: errorMessage,

variant: "destructive",

});

throw new Error(errorMessage);

} finally {

set({ isVoting: false });

}

} catch (error: any) {

console.error("Campaign creation error:", error);

set({

errorMessage: error.message || "Failed to create campaign",

isVoting: false

});

toast({

title: "Campaign Creation Error",

description: error.message || "Failed to create campaign", variant: "destructive",

});

throw error;

}

},

registerVoter: async (categoryId, voterAddress) => { try {

const { contract } = get();

const { account, chainId } = useWeb3Store.getState();

if (!contract) {

throw new Error("Contract not initialized");

}

if (!account) {

throw new Error("Wallet not connected");

}

// Check if we're on a supported network

if (!SUPPORTED\_NETWORKS.some(network => network.id === chainId)) {

throw new Error( Network not supported. Please switch to $

{SUPPORTED\_NETWORKS.map(n => n.name).join(' or ')} );

}

set({ isVoting: true, errorMessage: null });

try {

// Send register voter transaction

const tx = await contract.registerVoter(categoryId, voterAddress); set({ txHash: tx.hash });

// Wait for transaction confirmation

await tx.wait();

toast({

title: "Voter registered successfully!",

description:  Transaction hash: ${tx.hash.substring(0, 10)}... ,

});

} catch (txError: any) { console.error("Transaction error:", txError); let errorMessage = "Failed to register voter. "; // Check for common contract revert reasons if (txError.message.includes("execution reverted")) { errorMessage += "The transaction was rejected by the contract. Possible reasons include: voter already registered or insufficient permissions."; } else if (txError.message.includes("user rejected")) { errorMessage = "Transaction was rejected in your wallet."; } else if (txError.message.includes("insufficient funds")) {

errorMessage += "You don't have enough ETH to cover the transaction fee.";

} else {

errorMessage += txError.message;

}

set({ errorMessage });

toast({ title: "Voter Registration Error", description: errorMessage,

variant: "destructive",

});

throw new Error(errorMessage);

} finally {

set({ isVoting: false });

}

} catch (error: any) {

console.error("Voter registration error:", error);

set({

errorMessage: error.message || "Failed to register voter", isVoting: false

});

toast({

title: "Voter Registration Error",

description: error.message || "Failed to register voter", variant: "destructive",

});

throw error;

}

},

createCategory: async (categoryId, weight) => { try {

const { contract } = get();

const { account, chainId } = useWeb3Store.getState();

if (!contract) {

throw new Error("Contract not initialized");

}

if (!account) {

throw new Error("Wallet not connected");

}

// Check if we're on a supported network

if (!SUPPORTED\_NETWORKS.some(network => network.id === chainId)) {

throw new Error( Network not supported. Please switch to $

{SUPPORTED\_NETWORKS.map(n => n.name).join(' or ')} );

}

set({ isVoting: true, errorMessage: null });

try {

// Send create category transaction

const tx = await contract.createCategory(categoryId, weight); set({ txHash: tx.hash });

// Wait for transaction confirmation

await tx.wait();

toast({

title: "Category created successfully!",

description:  Transaction hash: ${tx.hash.substring(0, 10)}... ,

});

} catch (txError: any) { console.error("Transaction error:", txError); let errorMessage = "Failed to create category. ";

// Check for common contract revert reasons if (txError.message.includes("execution reverted")) { errorMessage += "The transaction was rejected by the contract. Possible reasons include: category already exists or insufficient permissions."; } else if (txError.message.includes("user rejected")) { errorMessage = "Transaction was rejected in your wallet."; } else if (txError.message.includes("insufficient funds")) {

errorMessage += "You don't have enough ETH to cover the transaction fee.";

} else {

errorMessage += txError.message;

}

set({ errorMessage });

toast({

title: "Category Creation Error", description: errorMessage, variant: "destructive",

});

throw new Error(errorMessage);

} finally {

set({ isVoting: false });

}

} catch (error: any) {

console.error("Category creation error:", error);

set({

errorMessage: error.message || "Failed to create category", isVoting: false

});

toast({

title: "Category Creation Error",

description: error.message || "Failed to create category",

variant: "destructive",

});

throw error;

}

}

})); function App() { return (

<Web3Provider>

<div className="flex flex-col min-h-screen">

<Header />

<main className="flex-grow">

<Router />

</main>

<Footer />

</div>

<Toaster />

</Web3Provider>

);

}

export default App;

4.3 Constraints, Alternatives and Tradeoffs

Constraints include limited scalability on Ethereum due to gas fees and network congestion, the complexity of writing secure smart contracts, and the steep learning curve for Web3 development. Alternatives considered included using other blockchains like Hedera or Polygon for lower fees and higher throughput, but Ethereum was chosen for its popularity and rich developer ecosystem. Tradeoffs were made between decentralization and usability; for example, while fully decentralized storage was considered via IPFS, centralized backend elements were retained for performance and ease of use. Additionally, instead of building a complex authentication system, wallet-based login via MetaMask was chosen for simplicity and alignment with Web3 principles.

#### CHAPTER 5 SCHEDULES, TASKS AND MILESTONES

Gantt Chart

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Task | Start Date |  | End Date | Duration |
| Project research and exploration | 13 Dec 2024 |  | 20 Dec 2024 | 1 week |
| Finalize idea and requirements | 21 Dec 2024 |  | 02 Jan 2025 | 2 weeks |
| Setup development environment | 03 Jan 2025 |  | 10 Jan 2025 | 1 week |
| Smart contract development | 11 Jan 2025 |  | 20 Jan 2025 | 1.5 weeks |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Smart contract testing and  debugging | 21 Jan 2025 |  | 30 Jan 2025 | 1.5 weeks |
| Frontend UI  development | 01 Feb 2025 |  | 10 Feb 2025 | 1 week |
| Smart contract and frontend integration | 11 Feb 2025 |  | 20 Feb 2025 | 1 week |
| Backend setup and schema design | 21 Feb 2025 |  | 28 Feb 2025 | 1 week |
| Backend and  frontend connection | 01 Mar 2025 |  | 10 Mar 2025 | 1 week |
| Full system testing and improvements | 11 Mar 2025 |  | 18 Mar 2025 | 1 week |
| UI polishing and  extra features | 19 Mar 2025 |  | 25 Mar 2025 | 1 week |
| Documentation and report writing | 26 Mar 2025 |  | 31 Mar 2025 | 1 week |
| Final deployment and presentation prep | 01 Apr 2025 |  | 06 Apr 2025 | 6 days |

The project schedule for BlockFund spans from 13th December 2024 to 6th April 2025, outlining a structured and phased development process. It begins with foundational research and requirement gathering, followed by environment setup and smart contract development. The mid-phase focuses on frontend creation, smart contract integration, and backend implementation. Subsequent weeks are dedicated to full system testing, user interface enhancements, and the addition of optional features. The final phase involves documentation, report writing, and preparing for deployment and presentation. This timeline ensures a balanced workflow with dedicated time for both development and quality assurance.

#### CHAPTER 6 SCHEDULES, TASKS AND MILESTONES

6.1 Sample Codes

Index.css

@tailwind base;

@tailwind components;

@tailwind utilities;

@layer base {

\* {

@apply border-border;

}

body {

@apply font-sans antialiased bg-gray-50 text-gray-800;

}

.truncate-2 { display: -webkit-box;

-webkit-line-clamp: 2; -webkit-box-orient: vertical; overflow: hidden;

}

.wallet-address { @apply fontmono;

}

.progress-bar { transition: width 0.5s ease-in-out;

}

}

Main.tsx import { createRoot } from "react-dom/client"; import App from "./App"; import "./index.css"; import { QueryClientProvider } from "@tanstack/react-query"; import { queryClient } from "./lib/queryClient";

createRoot(document.getElementById("root")!).render( <QueryClientProvider client={queryClient}><App />

</QueryClientProvider>

);

Index.ts import express, { type Request, Response, NextFunction } from "express"; import { registerRoutes } from "./routes"; import { setupVite, serveStatic, log } from "./vite";

const app = express(); app.use(express.json());

app.use(express.urlencoded({ extended: false }));

app.use((req, res, next) => { const start = Date.now(); const path = req.path; let capturedJsonResponse: Record<string, any> | undefined = undefined;

const originalResJson = res.json; res.json = function (bodyJson, ...args)

{ capturedJsonResponse = bodyJson;

return originalResJson.apply(res, [bodyJson, ...args]);

};

res.on("finish", () => { const duration = Date.now() - start; if (path.startsWith("/api")) { let logLine = `${req.method} ${path} ${res.statusCode} in ${duration}ms`; if (capturedJsonResponse) {

logLine += ` :: ${JSON.stringify(capturedJsonResponse)}`;

}

if (logLine.length > 80) { logLine = logLine.slice(0, 79) + "…";

}

log(logLine);

}

});

next();

});

(async () => { const server = await registerRoutes(app);

app.use((err: any, \_req: Request, res: Response, \_next: NextFunction) =>

{ const status = err.status || err.statusCode || 500; const message = err.message || "Internal Server Error";

res.status(status).json({ message }); throw err;

});

// importantly only setup vite in development and after

// setting up all the other routes so the catch-all route // doesn't interfere with the other routes if (app.get("env") === "development") { await setupVite(app, server);

} else { serveStatic(app);

}

// ALWAYS serve the app on port 5000 // this serves both the API and the client. // It is the only port that is not firewalled.

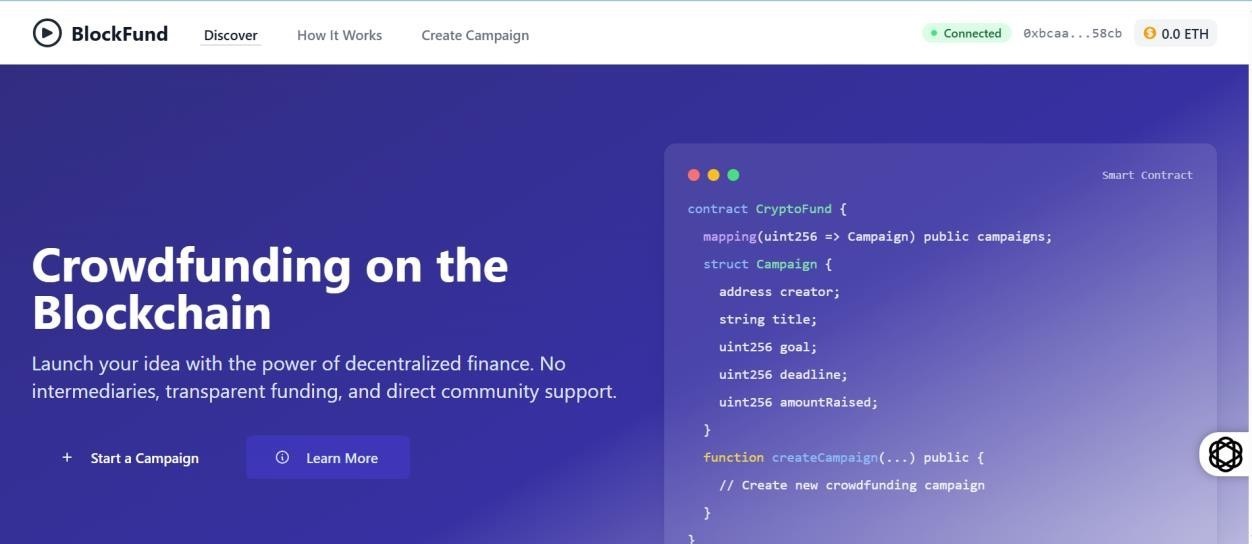
const port = 5000; server.listen({ p ort, host: "0.0.0.0", reusePort: true,

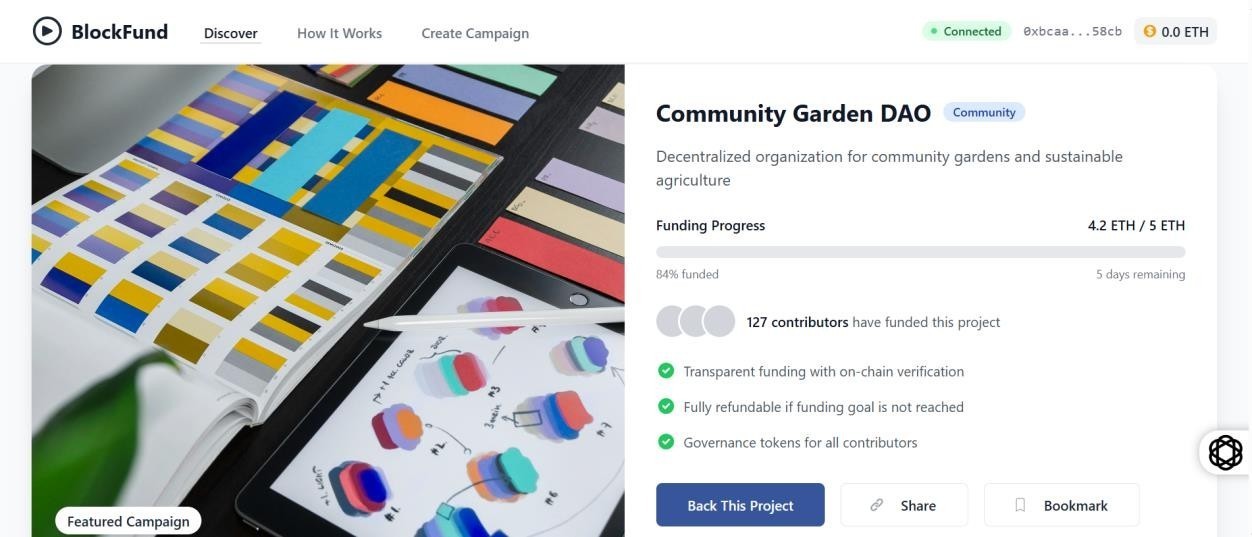
}, () => { log(`serving on port ${port}`);

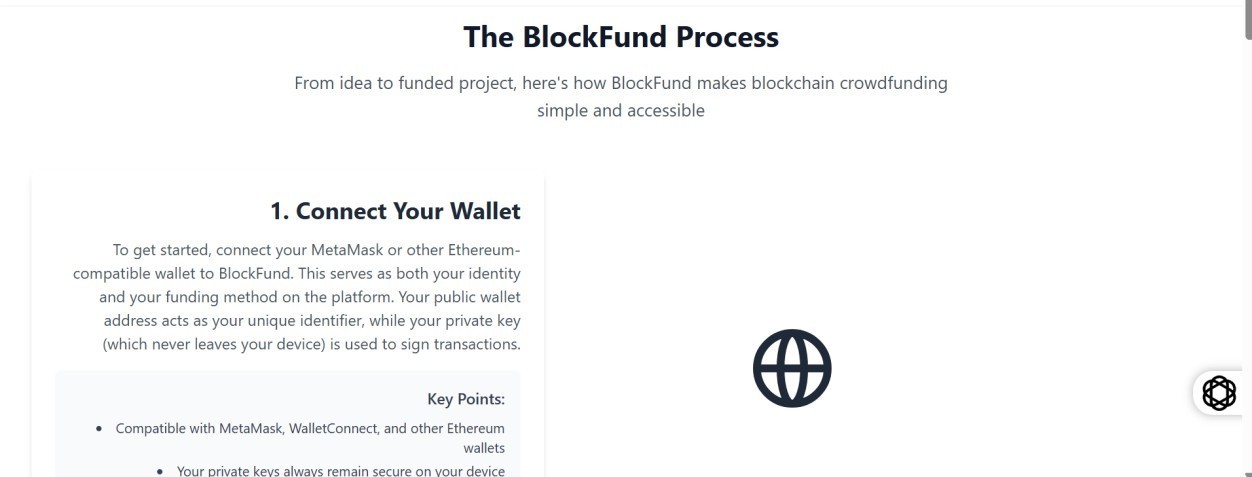
});

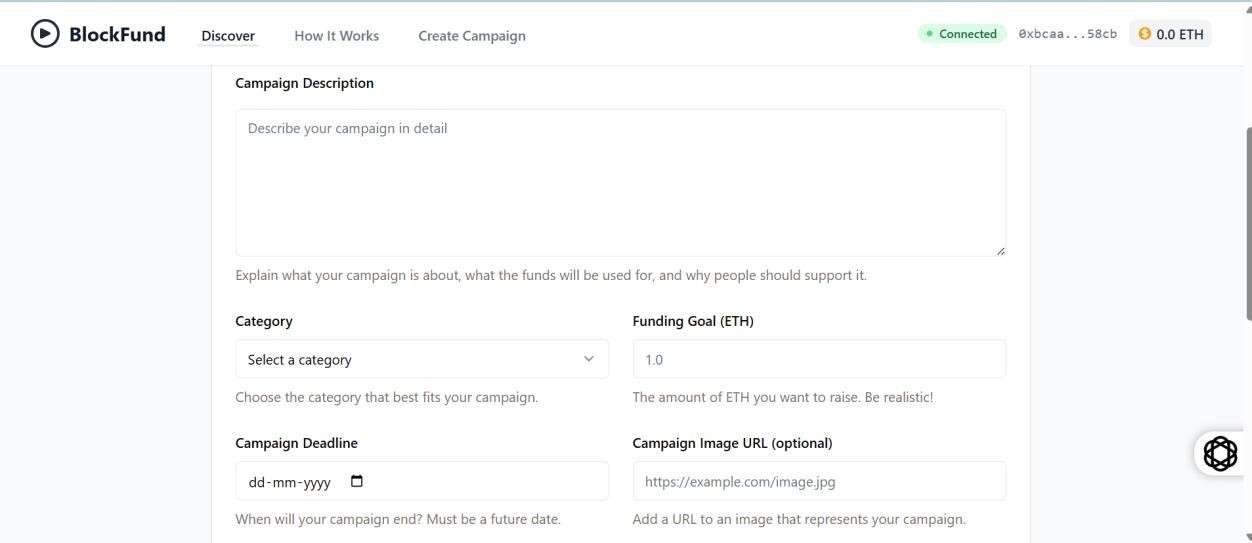
})();

6.2 Sample Screenshots









6.1 Screenshots

#### CHAPTER 7 RESULTS AND DISCUSSION

Platform Functionality

BlockFund is a decentralized crowdfunding platform powered by the Ethereum blockchain, providing an immutable, transparent, and secure environment for creating and supporting campaigns. Campaign organizers can set clear goals, milestones, and deadlines, with funds being released only when specific conditions are met. Smart contracts automate the process, ensuring funds are distributed according to predefined rules, without the need for a central authority. Integration with MetaMask ensures that users can easily manage their funds and authenticate their identities securely. The platform's user interface, built with React.js and styled with Tailwind CSS, offers a seamless experience for both campaign organizers and backers. Blockchain’s transparency ensures all transactions are publicly verifiable, building trust between users.

Smart Contract Performance:

Smart contracts form the backbone of BlockFund, automating the entire crowdfunding process. These self-executing contracts are programmed to release funds only when certain conditions are met, such as campaign success or milestone achievements. The use of Ethereum's Solidity programming language ensures that the smart contracts are both secure and efficient. Their performance is critical to the platform’s functionality, as they handle everything from fund distribution to refunds in case campaigns do not meet their goals. The decentralized nature of smart contracts eliminates the need for intermediaries, reducing overhead costs and increasing operational efficiency. Ethereum's gas fees, while sometimes fluctuating, remain an essential factor to consider, as they impact the cost of executing transactions and contract operations. Ensuring optimized smart contract code is vital for minimizing transaction costs and improving speed.

Transaction and Gas Efficiency:

Transaction and gas efficiency are pivotal for BlockFund's success as a decentralized platform. Ethereum’s blockchain requires gas fees to process transactions, which can sometimes fluctuate depending on network demand. To minimize costs, BlockFund's smart contracts are optimized to ensure that only necessary transactions are executed, reducing redundant operations. Efficient contract code ensures that the required actions are executed with minimal gas consumption, improving overall platform efficiency. Users are notified about the gas costs upfront, allowing them to make informed decisions when participating in campaigns. By optimizing the interaction between the front-end and Ethereum’s blockchain,

BlockFund aims

to provide users with a cost-effective crowdfunding experience.

Scalability and Future Enhancements:

Scalability is an important consideration for BlockFund as the platform aims to handle a large number of campaigns and transactions efficiently. While the Ethereum blockchain is currently one of the most secure and decentralized options, its scalability issues, such as high gas fees and network congestion, may affect performance during peak usage. To address this, BlockFund plans to explore Layer-2 solutions like Optimistic Rollups or zk-Rollups, which offer faster transaction processing at a lower cost. Additionally, integrating with other blockchain networks may provide further scalability and lower costs. Future enhancements also include adding features like multi-currency support, improved user interfaces, and increased customization options for campaign organizers, enabling the platform to cater to a wider range of use cases.

Security Considerations:

Security is a critical aspect of BlockFund, as it deals with financial transactions and sensitive user data. Smart contracts are the primary tool for ensuring security, as they automate processes in a way that makes them tamper-proof and resistant to fraud. However, ensuring the security of smart contracts requires thorough code auditing and testing to avoid vulnerabilities. BlockFund implements best practices for security, including formal verification of smart contract code, regular audits by third-party services, and adherence to the latest security protocols in blockchain technology. Additionally, the platform employs encryption to protect user data and prevent unauthorized access to funds. The decentralized nature of Ethereum ensures that no single entity can alter transaction records, providing an added layer of security for users.

#### CHAPTER 8 SUMMARY

BlockFund represents a bold step into the next generation of decentralized finance applications, merging the power of blockchain with the accessibility of crowdfunding platforms. The project was conceived as a solution to the growing need for transparency, user empowerment, and reduced dependency on centralized intermediaries in the crowdfunding space. This section summarizes the comprehensive journey taken—from ideation and development to testing and proposed future scalability.

1. Overview of BlockFund’s Purpose and Design

BlockFund was created to address specific inefficiencies in traditional crowdfunding platforms such as Kickstarter, Indiegogo, and GoFundMe. These platforms, though widely used, suffer from high operational fees, lack of contributor protection, delayed disbursements, and reliance on central authorities that can unilaterally modify policies or even block access to funds.

By leveraging Ethereum’s decentralized ledger and smart contract capabilities, BlockFund allows contributors and campaign creators to interact transparently without a middleman. The system operates on predefined logic encoded within smart contracts—ensuring that funds are only released upon meeting campaign goals or milestones. The use of MetaMask allows secure walletbased user authentication and simplifies blockchain interactions for non-technical users.

1. Technology Stack and Implementation Strategy

The core architecture of BlockFund is rooted in modularity and cross-layer synchronization:

* + Frontend: Developed using React.js and styled using Tailwind CSS, the interface is minimal, responsive, and tailored for an intuitive user journey. Routing is managed via Wouter for lightweight control, and animation libraries like Framer Motion enhance UX.
  + Smart Contracts: Solidity-based contracts govern campaign logic—contribution limits, deadline enforcement, milestone validation, and refunds. CampaignFactory acts as a manager of individual Campaign contracts.
  + Backend: A Node.js server manages campaign metadata and contributor history, storing data in PostgreSQL using drizzle-orm for type safety. The backend supports RESTful API endpoints and is protected with middleware for security and logging.
  + Testing & Deployment: Local testing is performed using Ganache and Hardhat, allowing quick iterations of contract development. Once finalized, the contracts are deployed on Ethereum testnets before going to mainnet.

BlockFund’s tech stack was carefully chosen for its community support, security maturity, and extensibility.

1. Features and Functional Modules

BlockFund supports a wide range of features that align with real-world crowdfunding needs:

* + Campaign Creation & Customization: Users can define title, description, funding goal, deadline, category, and an optional image. These parameters are stored on-chain or in a centralized metadata repository.
  + Contribution Flow: Contributors interact via MetaMask to transfer ETH to campaigns. Transactions are handled by the Ethereum blockchain and recorded immutably.
  + Escrow and Refund Logic: Funds are held in escrow by the smart contract and are only released when the campaign’s milestones are met. If the campaign fails, refunds are issued automatically to each contributor.
  + Real-Time Tracking: Users can monitor ongoing campaigns through visual progress bars and backend API queries. The UI updates dynamically based on the smart contract state.
  + Security Protocols: Reentrancy protection, transaction limit validation, and access control modifiers are enforced in all contracts. The frontend sanitizes inputs and uses secure libraries to reduce attack vectors.

This modular architecture ensures that each component can evolve independently, enabling continuous enhancement without breaking the core system.

1. Evaluation and Results

After integrating and testing the entire platform, BlockFund demonstrated a reliable and consistent performance across all layers:

* + Smart Contract Accuracy: All business logic defined in the campaign lifecycle—contributions, deadlines, and refunds—performed as expected. Error handling was accurate, and gas usage was optimized to minimize transaction costs.
  + Frontend Responsiveness: The frontend displayed accurate campaign data in real-time. Wallet connection was stable across test environments, and error messages were clear and actionable.
  + Backend Reliability: API endpoints reliably returned contributor and campaign data.

Logging systems captured all user interactions, aiding future audits.

The end-to-end workflow was tested with dummy data, demonstrating that users with no prior blockchain experience could comfortably use the platform after minimal onboarding.

1. User Impact and Real-World Relevance

BlockFund was designed to be user-centric, with both technical robustness and ease of use in mind. It enables users worldwide to participate in financial empowerment through decentralized mechanisms. The system can support a wide array of use cases:

* + Startups and Entrepreneurs: Small businesses looking for seed funding without approaching venture capitalists.
  + Charity and Social Impact: NGOs can transparently show how funds are being utilized.
  + Personal Campaigns: Individuals can fund personal medical or educational needs with direct control.

By ensuring that every transaction is verifiable and tamper-proof, BlockFund builds trust and accountability, critical elements that are often lacking in current systems.

1. Challenges Encountered

During development, several technical and conceptual challenges were encountered:

* + Smart Contract Complexity: Writing secure and gas-efficient contracts required deep knowledge of Solidity and Ethereum architecture. Certain scenarios, such as partial refunds or deadline extensions, had to be carefully modeled.
  + MetaMask Integration Issues: Handling wallet disconnection, switching networks, and user denial of permissions required graceful fallbacks.
  + IPFS and Storage Handling: Initially planned decentralized storage via IPFS introduced delay and data reliability issues, leading to a hybrid model.
  + User Education: Introducing blockchain concepts to non-technical users meant the UI and documentation had to be simplified significantly.

Despite these hurdles, strategic compromises and modular refactoring enabled the team to deliver a stable MVP.

1. Future Enhancements and Expansion

BlockFund is a foundational version, and several enhancements are already planned:

* + Layer-2 Blockchain Support: Migrating to Optimism or zkSync to reduce gas fees and improve scalability.
  + Multi-Wallet Support: Allowing login and transactions from WalletConnect, Coinbase Wallet, etc.
  + Reputation System: Scoring system for campaign creators based on past performance and ratings.
  + Analytics Dashboard: Contributor and admin analytics to track trends, success rates, and campaign categories.
  + Mobile App: A lightweight mobile interface with MetaMask Mobile support for broader accessibility.

These improvements will help transition BlockFund from a working prototype to a fully production-ready decentralized application.

1. Final Thoughts

BlockFund reflects the convergence of emerging technology with impactful real-world use cases. It highlights how blockchain can empower individuals, reduce dependency on centralized control, and offer robust transparency. The project’s modular codebase, secure smart contract logic, and modern frontend showcase not just technical acumen but also a thoughtful approach to product development.

By democratizing access to capital and ensuring ethical fund usage, BlockFund is well-positioned to inspire future innovation in decentralized fundraising.

#### CHAPTER 9 REFERENCES

1. R. NaveenKumaran, S. K. Geetha, K. Selvaraju, C. Kishore and A. Nagha Rathish, "Blockchain Based Crowd Funding", *2023 International Conference on Computer Communication and Informatics (ICCCI)*, pp. 1-8, 2023
2. V. Patil, V. Gupta and R. Sarode, "Blockchainbased crowdfunding application", *2021 Fifth International Conference on I-SMAC (IoT in Social Mobile Analytics and Cloud) (I-SMAC) [Conference paper]*, pp. 1546-1553, 2021, November.
3. H. Khatter, H. Chauhan, I. Trivedi and J. Agarwal, "Secure and transparent crowdfunding using blockchain", *IEEE Xplore*, pp. 76-80, 2021, August.
4. W. Teng, H. Xu, Z. Huang, Y. Bai and Z. Wang, "A Smart Contract-based Service Platform for Trustworthy Crowd Funding and Crowd Innovation", *2022 International Conference on Service Science (ICSS) [Conference paper]*, pp. 263-270, 2022, May.
5. S. Vp, L. Jain and H. Ahmed, "Crowd-Funding Using Blockchain", *2023 2nd International Conference on Advancements in Electrical Electronics Communication Computing and Automation (ICAECA) [Conference paper]*, pp. 1-5, 2023, June.
6. A. K. Dubey, S. C. Shingte, M. S. Siddiqui and S. Patil, "Crowdfunding using Blockchain for Startups and Investors", *2023 7th International Conference on Intelligent Computing and Control Systems (ICICCS) [Conference paper]*, pp. 1400-1405, 2023, May.
7. H. S. Rao, P. Sinha, S. S. BC, V. P. Aniketh and M. Namratha, "Blockchain Based Crowdfunding Platforms-Exploratory Literature Survey", *2023 5th Biennial International Conference on Nascent Technologies in Engineering (ICNTE) [Conference paper]*, pp. 1-4, 2023, January.
8. S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han and F. Y. Wang, "Blockchain-enabled smart contracts: Architecture applications and future trends", *IEEE Transactions on Systems Man and Cybernetics: Systems [Journal]*, vol. 49, no. 11, pp. 2266-2277, 2019.
9. Savant, R. V., Sunder, S. N., Spoorthi, M., Rajagopal, S. M., & Balakrishnan, R. M. (2024, June). Harnessing the Power of web3: A Blockchain Approach to Crowdfunding. In *2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT)* (pp. 1-9). IEEE.
10. Harshini, S. I. S., & Sabitha, R. (2024). Ethereum Powered Web3 Crowdfunding. *Computer, Communication, and Signal Processing. Smart Solutions Towards SDG.*, 339.
11. Vadlana, N., Thirupathi Rao, N., & Bhattacharyya, D. (2024, June). Decentralized Crowdfunding Application Using Blockchain Technologies. In *International Conference on Inventive Communication and Computational Technologies* (pp. 457-468). Singapore: Springer Nature Singapore.
12. Chamola, V. (2020). BitFund: A blockchain-based crowd funding platform for future smart and connected nation.
13. Hongjiang Zhao and Cephas Coffie, "The Applications of Blockchain Technology in Crowdfunding Contract", *SSRN Electronic Journal.*, 2018.

**APPENDIX A**

Schema.ts

**import { pgTable, text, serial, integer, boolean, timestamp, varchar } from "drizzle- orm/pg-core"; import { createInsertSchema } from "drizzle-zod"; import { z } from "zod";**

**export const users = pgTable("users", {**

**id: serial("id").primaryKey(), username: text("username").notNull().unique(), password: text("password").notNull(), walletAddress: text("wallet\_address").notNull().unique(),**

**});**

**export const campaigns = pgTable("campaigns",**

**{ id: serial("id").primaryKey(), title: text("title").notNull(), description: text("description").notNull(), category: text("category").notNull(), goal: text("goal").notNull(), // ETH amount as string to avoid precision issues deadline: timestamp("deadline").notNull(), imageUrl: text("image\_url"), creator: text("creator").notNull(), // Wallet address contractAddress: text("contract\_address"), // The deployed contract address amountRaised: text("amount\_raised").default("0"), // ETH amount as string status: text("status").default("active"), // active, completed, expired createdAt: timestamp("created\_at").defaultNow()**

**});**

**export const contributions = pgTable("contributions",**

**{ id: serial("id").primaryKey(), campaignId: integer("campaign\_id").notNull(), contributor: text("contributor").notNull(), // Wallet address amount: text("amount").notNull(), // ETH amount as string timestamp: timestamp("timestamp").defaultNow()**

**});**

**export const bookmarks = pgTable("bookmarks",**

**{ id: serial("id").primaryKey(), campaignId: integer("campaign\_id").notNull(), userAddress: text("user\_address").notNull(), // Wallet address createdAt: timestamp("created\_at").defaultNow()**

**});**

**// Insert Schemas export const insertUserSchema = createInsertSchema(users).pick({ username: true, password: true,**

**walletAddress: true,**

**});**

**export const insertCampaignSchema = createInsertSchema(campaigns)**

**.pick({ title : true, description: true, category: true, goal: true, deadline: true, imageUrl: true, creator: true, contractAddress: true,**

**})**

**.extend({**

**// Override the deadline field to accept string and convert to Date deadline: z.string().transform(val => new Date(val)),**

**});**

**export const insertContributionSchema = createInsertSchema(contributions).pick({ campaignId: true, contributor: true, amount: true,**

**});**

**export const insertBookmarkSchema = createInsertSchema(bookmarks).pick({ campaignId: true, userAddress: true,**

**});**

**// Types export type InsertUser = z.infer<typeof insertUserSchema>; export type User = typeof users.$inferSelect;**

**export type InsertCampaign = z.infer<typeof insertCampaignSchema>; export type Campaign = typeof campaigns.$inferSelect;**

**export type InsertContribution = z.infer<typeof insertContributionSchema>; export type Contribution = typeof contributions.$inferSelect;**

**export type InsertBookmark = z.infer<typeof insertBookmarkSchema>; export type Bookmark = typeof bookmarks.$inferSelect;**