

Votereum: Blockchain-Based Election Lifecycle Management System

Lakshmeesha

*Department of Information Science and Engineering
CMR Institute of Technology, Bengaluru, India
1CR23IS089*

Jahanara Shaik

*Assistant Professor, Department of ISE
CMR Institute of Technology, Bengaluru, India
jahanara.s@cmrit.ac.in*

Pandappa Ramappa Vasanad

*Department of Information Science and Engineering
CMR Institute of Technology, Bengaluru, India
1CR23IS115*

Mohammedtaha M Walikar

*Department of Information Science and Engineering
CMR Institute of Technology, Bengaluru, India
1CR23IS102*

Abstract:

Votereum is a blockchain-based election lifecycle management system designed for structured institutional elections. The system combines a web-based React interface, Firebase authentication and data services, and an Ethereum smart contract to manage the election process from configuration and nomination through candidate approval, voting, finalization, and result presentation. The platform is intended for controlled college environments where eligibility can be defined using admission-batch information and where election activities require a transparent and auditable record. The smart contract maintains election identifiers, posts, candidate records, approval status, vote counts, election status, and per-post voting restrictions. Firebase is used for identity and application-level data synchronization, while the blockchain provides the authoritative vote-related state used for result synchronization. The implementation supports local development using Ganache and deployment to the Ethereum Sepolia test network through Hardhat. The application provides separate user and administrative workflows, protected routes, nomination management, election status control, and a finalized-results interface. The demonstrated implementation shows that blockchain can be integrated with conventional web application services without requiring the entire application state to reside on-chain. The resulting architecture provides a practical model for institution-level elections while preserving a clear separation between identity management, application data, and tamper-resistant vote records.

Keywords — Blockchain, E-Voting, Ethereum, Smart Contract, Solidity, React, Firebase, Firestore, Election Lifecycle, Sepolia, Hardhat, Digital Voting.

I. INTRODUCTION

Electronic voting systems are increasingly used to reduce administrative effort and provide faster election processing. However, an election platform must do more than present a voting form. It must control who can participate, define candidate eligibility, manage election phases, prevent repeated voting, preserve trustworthy records, and

present results consistently. Conventional centralized applications normally keep these responsibilities inside a database and application server. This creates a strong dependency on the administrative backend and makes the integrity of critical vote records dependent on the security and governance of that central infrastructure.

Blockchain provides a complementary mechanism for critical election records. A

blockchain stores transactions in a distributed ledger and uses cryptographic verification to make unauthorized alteration of committed records difficult [1], [11]. Ethereum extends this model with programmable smart contracts, allowing application rules to be encoded as deterministic functions [12], [13]. Earlier blockchain voting research has demonstrated the feasibility of recording election operations through Ethereum-based systems [2], while other studies have examined blockchain voting from broader system-design and automation perspectives [1], [3].

Votereum applies these principles to a college-election context. Rather than attempting to place every user profile, interface element, and administrative record on-chain, the system adopts a hybrid architecture. Firebase manages authentication and application data, while the ElectionSystem smart contract manages election identifiers, candidates, candidate approval, election status, vote counts, and voting restrictions. This division reduces unnecessary blockchain storage while preserving a verifiable ledger for the core election operations.

The project therefore addresses four practical requirements: controlled participation, structured election lifecycle management, blockchain-backed vote recording, and synchronized result presentation. The frontend provides distinct interfaces for voters and administrators. Eligibility is derived from college identity information and election-specific batch configuration. Administrative functions are protected by role-based routing and smart-contract access control. Voting is permitted only during the ongoing phase and the contract prevents an address from voting more than once for the same post. Finalized results are read back from the blockchain and synchronized with application-level candidate data.

The remainder of this paper describes the related work, proposed architecture, smart-contract design, election workflow, security controls, implementation, and functional results demonstrated by the Votereum application.

II. RELATED WORK

Kshetri and Voas [1] discuss blockchain-enabled electronic voting as a mechanism for improving

transparency and resistance to tampering. Their work provides a foundation for considering distributed ledgers in election systems. Hjálmarsson et al. [2] present an Ethereum-oriented blockchain voting implementation and illustrate how smart contracts can be incorporated into an end-to-end election workflow. Pawlak et al. [3] investigate intelligent-agent concepts in blockchain e-voting, highlighting opportunities for automation around election activities.

Nakamoto's original blockchain architecture [11] established the distributed-ledger model that underlies transaction chaining and consensus. Ethereum's technical specification [12] and Buterin's Ethereum proposal [13] extended the concept to general-purpose smart contracts and decentralized applications. These foundations are directly relevant to Votereum because the election rules are implemented as Solidity functions and state transitions rather than being limited to application-server logic.

The implementation also relies on established software frameworks. Solidity documentation [4] supports smart-contract development, Hardhat [5] supports compilation and deployment, and Ganache [14] provides a local blockchain environment. Firebase documentation [6] supports authentication, Firestore synchronization, and security rules. React [7], ethers.js [8], MetaMask [9], Tailwind CSS [10], React Router [16], OpenZeppelin guidance [17], Vite [18], TypeScript [19], and Lucide [20] provide the application-development foundation.

The main design distinction of Votereum is its lifecycle-oriented organization. Instead of treating voting as one isolated transaction, the system models election creation, nomination, approval, ongoing voting, finalization, and results as connected stages. This approach is suitable for institutional elections in which eligibility and administrative approval must be established before voting becomes available.

TABLE I
MAJOR COMPONENTS OF VOTEREUM

Component	Technology	Primary Role	Data/Operation
Frontend	React, TypeScript, Vite	User interface	Login, dashboard, elections, admin, results
Identity	Firebase Authentication	User authentication	Google sign-in and session state
Application DB	Cloud Firestore	Application synchronization	Users, elections, nominations
Blockchain	Ethereum Sepolia	Ledger and vote state	Election/candidate state and votes
Smart Contract	Solidity 0.8.19	Election rules	Create, nominate, approve, status, vote
SDK	ethers.js	Contract interaction	Read/write calls and events
Development	Hardhat, Ganache	Build/test/deploy	Compilation and local testing

III. PROPOSED VOTEREUM FRAMEWORK

Votereum is organized as a hybrid web and blockchain system. The frontend is responsible for presentation and user interaction. Firebase Authentication establishes the user session, while Firestore stores application-level records and supports real-time synchronization. The blockchain layer is responsible for the election state that must be independently verifiable, including candidate registration, approval state, election status, and vote counts. This separation is deliberate: storing all interface and profile data on-chain would increase complexity and transaction cost without providing a corresponding benefit.

The application defines two principal operational roles. The administrator creates elections, specifies posts, selects eligible voter and candidate batches, controls election status, and reviews nominations. A voter can inspect eligible elections, submit a nomination when eligible, and cast votes during the ongoing phase. Protected frontend routes prevent ordinary users from accessing the administrative interface, while the smart contract independently protects administrator-only state-changing functions through the `onlyAdmin` modifier.

A typical election contains one or more posts such as President, Vice President, General Secretary, and Technical Head. The administrator can configure a nomination deadline and specify admission batches that are allowed to vote or nominate. The frontend derives eligibility from the authenticated user's college information, while Firestore security rules also enforce election-specific access constraints for application data.

The lifecycle is represented by the statuses *nomination*, *ongoing*, and *ended*. During nomination, eligible users may submit candidate information. Approved candidates become available for voting after the administrator moves the election into the ongoing phase. The vote function verifies that the election is active, that voting is live, that the candidate has been approved, and that the sender has not already voted for the candidate's post. When the election is ended, the results interface reads candidate vote counts and presents the finalized outcome.

A. System Architecture

Fig. 1 shows the logical hybrid architecture. Identity and application services remain in Firebase, while election state and vote counts are maintained by the Ethereum smart contract.

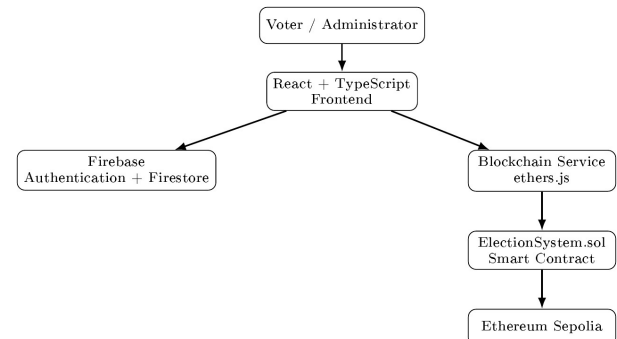


Fig. 1 Hybrid architecture of the Votereum system.

IV. SMART CONTRACT DESIGN

The ElectionSystem contract is implemented in Solidity 0.8.19. The contract defines Candidate and Election structures. A Candidate contains a name, statement, post, voteCount, wallet address, and approval flag. An Election contains its title, description, active state, lifecycle status, nomination deadline, voter batches, candidate batches, post list, candidate count, candidate mapping, and a nested `hasVoted` mapping. This

structure allows multiple elections to coexist within a single deployed contract.

The administrator is stored as a contract-level address. The `onlyAdmin` modifier restricts `createElection`, `approveCandidate`, and `updateStatus` to that address. The `createElection` function increments `electionsCount` and initializes the election with the configured posts, batch lists, nomination deadline, and nomination status. The contract emits `ElectionCreated`, allowing the application to identify the newly created election.

Candidate registration is performed through `nominate`. The function verifies that the election is active and currently in the nomination phase before creating a candidate entry. The candidate is initially unapproved. This prevents a submitted nomination from becoming automatically eligible to vote. The administrator must explicitly approve the candidate through `approveCandidate`.

The `vote` function provides the core blockchain voting rule. It requires the election to be active and in the ongoing state. It then checks candidate approval and consults `hasVoted[msg.sender][candidate.post]`. If the address has already voted for that post, the transaction is rejected. Otherwise, the candidate `voteCount` is incremented and the address/post combination is recorded. The contract emits `VoteCast` after the state change.

The contract exposes read functions including `getElectionBasics`, `getCandidate`, and `getElectionPosts`. These functions are used by the application to retrieve blockchain state without creating a transaction. Event declarations such as `ElectionCreated`, `Nominated`, `CandidateApproved`, `VoteCast`, and `StatusUpdated` provide an auditable record of major state transitions.

TABLE II
SMART-CONTRACT FUNCTIONS

Function	Access	Purpose
<code>createElection</code>	Admin	Creates and configures an election
<code>nominate</code>	Caller	Registers a candidate during nomination
<code>approveCandidate</code>	Admin	Approves a nominated candidate

Function	Access	Purpose
<code>updateStatus</code>	Admin	Moves an election between lifecycle states
<code>vote</code>	Caller	Records one vote per post
<code>getElectionBasics</code>	Public	Returns election metadata
<code>getCandidate</code>	Public	Returns candidate and vote information
<code>getElectionPosts</code>	Public	Returns configured posts

V. AUTHENTICATION, ELIGIBILITY AND SECURITY

Firestore Authentication is used for Google-based sign-in. After authentication, the application extracts college information from the authenticated email identifier and stores user information in Firestore. The application maintains a role value for administrative and voter workflows. The administrative role is restricted through a whitelist in the current implementation and is also checked by Firestore rules and protected frontend routes.

Election eligibility is batch-oriented. An election can define `voterBatches` and `candidateBatches`, allowing the administrator to restrict participation to selected admission-year groups. The frontend uses this information to determine whether a user can vote or nominate, and Firestore security rules contain corresponding eligibility checks. This provides two layers of enforcement: interface-level eligibility and database-level access control.

Firestore rules deny access globally by default and then grant narrowly defined permissions for users, elections, nominations, and vote records. Election updates are restricted to administrators except for a controlled candidate-array update path during an ongoing election. The votes subcollection also binds creation and update permissions to the authenticated user's UID and the election's ongoing state. These rules complement, rather than replace, the smart-contract checks.

The blockchain contract provides the strongest protection for the vote count itself. Since candidate `voteCount` and `hasVoted` state are maintained by the contract, a normal client-side database edit cannot directly change the on-chain vote count. This distinction is central to the hybrid design: Firestore is used for application coordination, while blockchain state is used for the core vote record and result synchronization.

A security limitation must nevertheless be acknowledged. The supplied development source currently contains a system-relayer private key in the frontend blockchain service. A private signing key must not be embedded in browser-delivered application code or committed to a repository. For production deployment, this key should be rotated immediately and transaction signing should be moved to a protected backend or secure key-management service. The current implementation should therefore be treated as a project/testnet implementation rather than production-ready election infrastructure.

Another limitation is that the smart contract itself does not encode the full college eligibility policy. Batch eligibility is partly enforced in Firebase/application logic, while the blockchain contract focuses on election state and voting constraints. This is a deliberate hybrid choice for the project, but a production system would need stronger cryptographic identity-to-eligibility binding if resistance to a compromised application layer were required.

VI. ELECTION LIFECYCLE AND APPLICATION WORKFLOW

The Votereum workflow begins when an administrator initializes an election. The management interface accepts the election title and description, allows posts to be added, specifies the nomination deadline, and selects voter and candidate admission batches. The administrator then launches the nomination phase. The application creates the blockchain election and stores the corresponding application metadata, including the blockchain election identifier.

During nomination, an eligible candidate selects a post and submits a statement. The nomination is recorded in the application layer and the blockchain nomination function registers the candidate. The administrator reviews the nomination queue and can approve or reject candidates. Approval is also reflected on-chain, so a candidate cannot simply become vote-eligible by changing a client-side record.

After candidate approval, the administrator transitions the election to the ongoing state. The voter interface displays approved candidates

grouped by post. When a vote is submitted, the blockchain service invokes the `vote` function. The application also records a user-level voting marker in Firestore so that the interface can immediately show the relevant post as already voted. The authoritative vote count is later obtained from the blockchain.

At the end of the election, the administrator changes the status to ended. The results page retrieves ended elections from Firestore and then reads candidate vote counts from the smart contract. This synchronization step is important because it avoids treating an application-level cached number as the final authority. The interface groups candidates by post and displays the candidate with the highest recorded count as the elected result for that post.

The supplied screenshots demonstrate this lifecycle visually. The login screen provides Google authentication; the dashboard presents active proposals and navigation; the election-management screen configures posts, nomination deadlines, and batches; the election screen displays candidates by post; and the finalized-results screen presents election outcomes and vote counts.

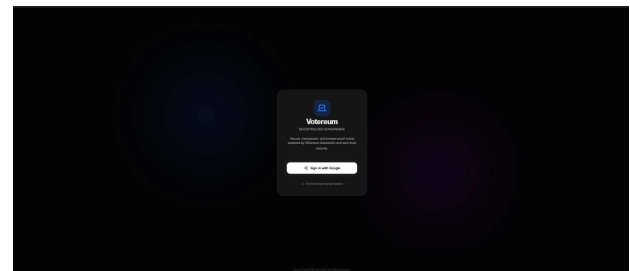


Fig. 2 Votereum Google authentication interface.

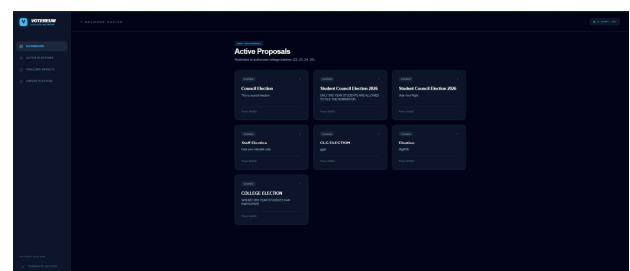


Fig. 3 Votereum dashboard showing active election proposals.

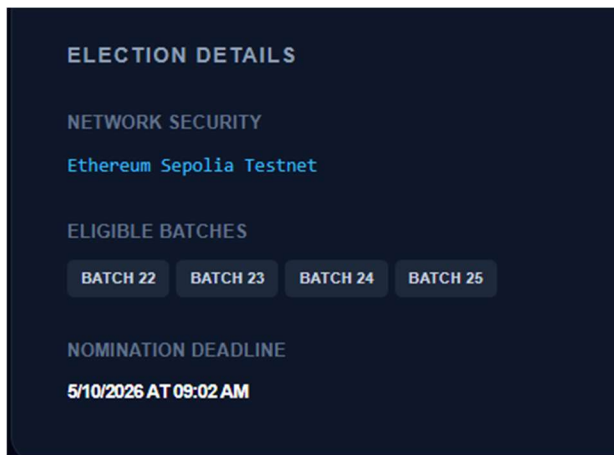


Fig. 4 Election Management interface for posts, deadlines and eligible batches.

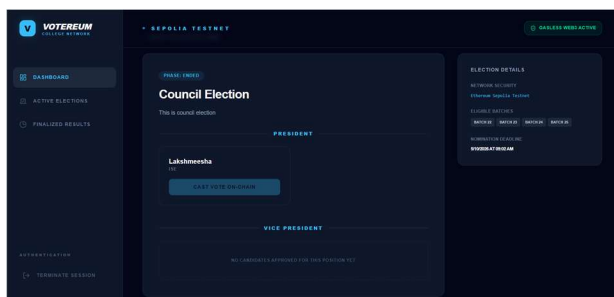


Fig. 5 Election interface showing candidates organized by post.

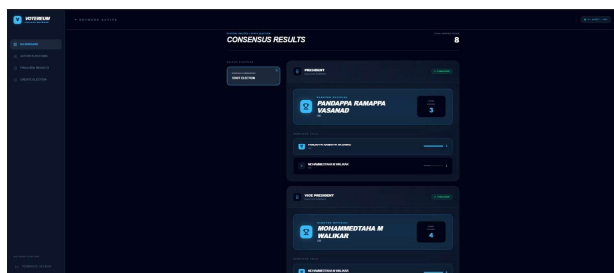


Fig. 6 Finalized results interface showing post-wise vote counts.

VII. IMPLEMENTATION AND DEPLOYMENT

The frontend is built with React and TypeScript and uses Vite for development and build tooling. React Router provides routes for login, dashboard, elections, results, and the administrator panel. The ProtectedRoute component checks authentication and, for administrative paths, verifies the administrator role before rendering the requested page. Tailwind CSS provides responsive visual styling, while Motion and Lucide support interface animation and icons.

The Firebase layer is initialized through the project configuration and is used by the authentication hook and page components. Firestore

listeners use real-time snapshot updates so that election and nomination information can update in the interface without manual refresh. This behavior is useful during administration because the nomination queue and election state can change while the management interface remains open.

The blockchain service uses ethers.js v6 and a contract ABI that exposes the ElectionSystem functions and events. A read-only Sepolia provider is used for blockchain queries. The deployment scripts use Hardhat and support Ganache, Sepolia, and Polygon Amoy network configurations. For this project, Ganache is used for local-host development and testing, while the demonstrated deployed environment is Ethereum Sepolia.

The deployment script compiles the Solidity contract, obtains a deployer signer, deploys ElectionSystem, waits for deployment, reports the contract address and transaction hash, and updates the frontend constants file with the deployed address. This reduces the chance of manually copying an outdated contract address into the application after redeployment.

The application uses a hybrid synchronization process. Firestore maintains user, election, nomination, and interface-oriented state, while blockchain reads are used to refresh candidate vote counts. In the results page, each candidate's application record is associated with a blockchain candidate identifier and the current on-chain vote count is read using `getCandidate`. This design gives the UI the convenience of a conventional database while retaining an on-chain source for the core voting result.

VIII. FUNCTIONAL RESULTS AND PERFORMANCE ANALYSIS

The supplied implementation and screenshots support a functional evaluation rather than a statistically significant performance benchmark. The demonstrated system includes interfaces for authentication, election browsing, election configuration, candidate nomination and approval, voting, and finalized result presentation. The source code also contains explicit validation rules for election status, candidate approval, administrator access, and duplicate voting.

TABLE III
FUNCTIONAL VERIFICATION

Test Area	Observed / Implemented Behavior	Evidence
Authentication	Google sign-in establishes application session.	Login + Firebase
Election creation	Admin defines title, description, posts, deadline and batches.	Management + contract
Nomination	Eligible users submit candidate information during nomination.	Workflow + contract
Approval	Admin approval changes candidate eligibility on-chain.	Admin workflow
Voting	Votes accepted only when election is ongoing and candidate approved.	Vote function
Double-vote prevention	Same address cannot vote twice for the same post.	hasVoted mapping
Finalization	Admin ends election and results page lists ended elections.	Results page
Synchronization	Candidate vote counts are read from deployed contract.	Results code

The project does not provide sufficient controlled measurements to claim a numerical throughput, latency, or accuracy improvement. Therefore, this paper does not invent such values. The appropriate result for the current implementation is functional correctness across the election lifecycle. A future evaluation can add transaction confirmation time, gas consumption, concurrent-user throughput, Firestore synchronization latency, and contract-level test coverage as measurable performance indicators.

The finalized-results screenshots demonstrate that the system can present post-wise outcomes after an election has ended. The displayed counts are treated as demonstration data from the project environment and not as a benchmark population. The results interface reads blockchain vote counts and organizes them by election post, supporting the intended separation between application presentation and the ledger-backed voting state.

IX. DISCUSSION

The principal benefit of the Votereum architecture is the separation of responsibilities. Firebase provides a practical mechanism for identity, session management, real-time application state, and eligibility information. Ethereum

provides a persistent state machine for election operations and vote counts. This is more practical for a college project than putting every record on-chain, because blockchain transactions are comparatively expensive and less convenient for frequently changing interface data.

The lifecycle model also provides a stronger structure than a single voting endpoint. Candidate approval is separated from nomination, voting is separated from nomination, and results are separated from active voting. Each phase has a corresponding status check. This reduces accidental cross-phase operations and makes the workflow easier to explain and audit.

However, blockchain does not automatically solve every security or election-integrity problem. The system still depends on correct identity management, eligibility configuration, administrator governance, secure deployment, and protection of signing keys. A blockchain can make an incorrect transaction difficult to alter, but it cannot determine whether the original identity or eligibility decision was correct. Votereum should therefore be understood as a blockchain-backed institutional election prototype rather than a complete solution for public-government elections.

Privacy is another consideration. The current contract records candidate information and wallet addresses associated with nominations and emits voter addresses in VoteCast events. Although a blockchain provides transparency, public transaction data can create privacy concerns when wallet identities can be linked to real people. A production election system would require a stronger privacy-preserving voting protocol, such as cryptographic voter credentials, unlinkable ballots, or zero-knowledge mechanisms, depending on the election requirements.

The current relayer-based write architecture also creates a central operational dependency. Although it can simplify user interaction by avoiding direct gas management, the relayer becomes a critical signing component. Its key must therefore be isolated from client code, rotated when exposed, and protected using secure server-side infrastructure. This limitation is especially important before any deployment beyond a controlled test environment.

X. CONCLUSION

Votereum demonstrates a hybrid approach to institutional election management in which conventional web technologies and blockchain are combined according to their strengths. React and TypeScript provide the user interface, Firebase provides authentication and application synchronization, and Ethereum smart contracts provide the core election and vote state. The ElectionSystem contract supports multiple elections, configurable posts and eligibility batches, candidate nomination and approval, controlled status transitions, and one-vote-per-post enforcement for each blockchain address.

The implemented workflow covers the complete election lifecycle from configuration to finalized results. Functional evidence from the application demonstrates login, election management, nomination and voting interfaces, and post-wise result presentation. The use of Ethereum Sepolia provides a public testnet environment for the deployed contract, while Ganache remains useful for local development and testing.

The system also illustrates the limits of blockchain-based voting. Ledger immutability does not by itself guarantee correct identity, eligibility, privacy, or administrative governance. The current implementation therefore represents a practical academic prototype and a foundation for further work. Future development can strengthen identity binding, privacy, secure key management, automated smart-contract testing, formal verification, scalability evaluation, and independent security auditing.

Overall, Votereum shows how blockchain can be integrated into an election lifecycle without forcing every part of the application onto the ledger. The combination of Firebase-based application services, role-aware workflows, smart-contract validation, and blockchain-backed result synchronization provides a clear technical demonstration of decentralized vote management within a college-election setting.

REFERENCES

- [1] N. Kshetri and J. Voas, "Blockchain-Enabled E-Voting," *IEEE Software*, vol. 35, no. 4, pp. 95–99, 2018.
- [2] F. Þ. Hjálmarsson, G. K. Hreiðarsson, M. Hamdaqa, and G. Hjálmtýsson, "Blockchain-Based E-Voting," in *11th IEEE International Conference on Cloud Computing*, pp. 983–986, 2018.
- [3] M. Pawlak, A. Poniszewska-Marañda, and N. Kryvinska, "Towards the Intelligent Agents for Blockchain E-Voting System," *Procedia Computer Science*, vol. 141, pp. 239–246, 2018.
- [4] Ethereum Foundation, "Solidity Documentation v0.8.x," 2024. [Online]. Available: <https://docs.soliditylang.org>
- [5] Hardhat, "Hardhat Documentation," 2024. [Online]. Available: <https://hardhat.org/docs>
- [6] Google Firebase, "Firebase Documentation," 2024. [Online]. Available: <https://firebase.google.com/docs>
- [7] Meta Open Source, "React Documentation," 2024. [Online]. Available: <https://react.dev>
- [8] Ethers.js Contributors, "Ethers.js Documentation v6," 2024. [Online]. Available: <https://docs.ethers.org>
- [9] MetaMask, "MetaMask Developer Documentation," 2024. [Online]. Available: <https://docs.metamask.io>
- [10] Tailwind CSS, "Tailwind CSS v4 Documentation," 2024. [Online]. Available: <https://tailwindcss.com/docs>
- [11] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.
- [12] G. Wood, "Ethereum: A Secure Decentralised Generalised Transaction Ledger," Ethereum Project Yellow Paper, 2014.
- [13] V. Buterin, "Ethereum White Paper: A Next-Generation Smart Contract and Decentralized Application Platform," 2013.
- [14] Truffle Suite, "Ganache Documentation," 2024. [Online]. Available: <https://trufflesuite.com/ganache>
- [15] Framer Motion Contributors, "Motion/React Documentation," 2024. [Online]. Available: <https://motion.dev>
- [16] React Router, "React Router v7 Documentation," 2024. [Online]. Available: <https://reactrouter.com/docs>
- [17] OpenZeppelin, "Smart Contract Security Best Practices," 2024. [Online]. Available: <https://docs.openzeppelin.com>
- [18] Vite, "Vite Documentation," 2024. [Online]. Available: <https://vitejs.dev/guide>
- [19] TypeScript, "TypeScript Documentation," 2024. [Online]. Available: <https://www.typescriptlang.org/docs>
- [20] Lucide Icons, "Lucide React Documentation," 2024. [Online]. Available: <https://lucide.dev>