CHAPTER 1

INTRODUCTION

1.1 Introduction

Modern electoral systems must balance security, transparency, and accessibility while handling largescale participation in a trustworthy manner. Traditional paper-based voting and conventional Electronic Voting Machines (EVMs) have improved logistics and reduced manual errors but still face concerns related to tampering, lack of transparency, centralized control, and limited support for remote voters.

The iBallot project, “Decentralized Voting System Using Blockchain Technology”, addresses these limitations by leveraging blockchain as a secure, tamper-resistant, and auditable ledger for recording votes. Each vote is modeled as a cryptographically protected, immutable transaction on a distributed network, eliminating single points of failure and significantly reducing the potential for fraud or unauthorized modification. Smart contracts automate vote validation and tallying, improving accuracy and efficiency while preserving voter confidentiality.

Unlike traditional systems that depend on physical polling stations, iBallot enables remote voting from any internet-enabled device, thereby improving participation for overseas citizens and individuals with mobility or location constraints. Real-time tallying and immediate result availability reduce delays associated with manual counting, while the transparent audit trail offered by blockchain enhances public trust through end-to-end verifiability.

This technical report presents the architectural design, workflow, implementation, and evaluation of iBallot, demonstrating how blockchain can be integrated with a modern web stack to build a secure, transparent, efficient, and scalable digital voting platform for contemporary democratic processes.

1.2 Overview

Blockchain technology is a decentralized and distributed digital ledger system that records transactions across multiple nodes in a peer-to-peer network. Unlike traditional centralized databases, blockchain operates without a single controlling authority, ensuring transparency, security, and resilience against tampering. Each transaction recorded on the blockchain is verified by network participants and stored in a block, which is cryptographically linked to the previous block, forming an immutable chain of records.

A key feature of blockchain is its immutability. Once data is recorded and confirmed, it cannot be altered or deleted without consensus from the network, making blockchain highly resistant to fraud and unauthorized modifications. Cryptographic hashing techniques and digital signatures are used to ensure data integrity and authenticity. This property is particularly important for sensitive applications such as electronic voting, where trust and accuracy are critical. Blockchain systems can be classified as public, private, or consortium blockchains. Public blockchains allow open participation and verification, while private and consortium blockchains restrict access to authorized entities. Smart contracts, another core component of blockchain technology, are self-executing programs stored on the blockchain that automatically enforce predefined rules and conditions. These contracts eliminate the need for intermediaries and ensure transparent and automated execution of processes.

1.3 Motivation

The core motivation for developing a decentralized voting system using blockchain stems from persistent shortcomings in conventional paper-based and electronic voting, including vulnerability to manipulation, limited transparency, lack of trust, and significant barriers to accessibility.

- **Security and Integrity**: Traditional voting systems both paper ballots and Electronic Voting Machines remain susceptible to fraud, tampering, and unauthorized modification of centralized databases. Blockchain provides a decentralized, tamper-proof ledger that ensures all votes are immutable, permanent, and independently verifiable, substantially strengthening public trust in electoral integrity.
- **Transparency**: Centralized approaches restrict public oversight, fueling skepticism about result validity. Blockchain records every vote on a distributed ledger visible to authorized nodes, enabling real-time auditing and ensuring public accountability for the end-to-end process.
- **Voter Privacy**: Blockchain-based systems can protect ballot confidentiality through encryption and anonymized voter identifiers. iBallot implements this balance by hashing voter identity on-chain and abstracting wallet interactions from end-users, ensuring privacy without sacrificing verifiability.
- **Accessibility**: Geographic isolation, physical disabilities, and situational barriers prevent millions of eligible citizens from participating in physical polling stations. A blockchain-enabled web platform allows secure remote voting from any internet-connected device, expanding participation for overseas voters, military personnel, and individuals with mobility constraints.

- **Efficiency and Cost**: Automating registration, validation, vote casting, and result computation via smart contracts significantly reduces operational overhead, eliminates manual counting delays, minimizes human error, and lowers the logistical burden of traditional elections. iBallot directly addresses these motivations by combining blockchain immutability with user-friendly design, gasless meta-transactions, and mock DigiLocker-based identity verification, creating a secure, accessible, and efficient digital voting platform.

1.4 Problem Statement

Traditional voting systems face multiple critical limitations: paper-based voting incurs high logistical costs and manual counting delays, Electronic Voting Machines lack transparency and remain vulnerable to tampering and centralized manipulation, and both approaches exclude voters in remote regions or with mobility constraints. Furthermore, existing blockchain-based voting solutions either require users to manage cryptocurrency wallets, lack national identity integration, or depend on private test networks unsuitable for long-term deployment.

"To design and implement a decentralized, user-friendly, blockchain-based voting platform that ensures secure and transparent ballot recording, eliminates single points of failure, enables gasless remote participation through a simplified web interface, and integrates with government identity systems for verifiable yet accessible digital elections."

1.5 Scope of Project

The scope of this project encompasses the comprehensive design, development, and deployment of a decentralized, blockchain-powered electronic voting system. It includes:

- Architecting a secure blockchain network using consensus protocols and smart contracts for transparent and tamper-resistant recording of votes and election events.
- Developing intuitive web and mobile interfaces facilitate voter registration, secure authentication, vote casting, and real-time results viewing, ensuring broad accessibility and usability.
- Implementing cryptographic techniques such as hashing and encryption to maintain voter privacy, verify voter eligibility, and guarantee data integrity throughout the process.
- Enabling remote participation, full auditability, and real-time verification for both voters and election authorities, supported by immutable blockchain records and automated smart contract logic.
- Ensuring a modular architecture adaptable to evolving regulatory frameworks,

scalable for diverse election types, and updatable as technological standards advance within the digital election domain.

- This scope establishes a foundation for secure, transparent and efficient digital elections, with potential extensions for broader civic engagement and institutional decision-making in the future.

1.6 Objectives of the Project

The main objectives of the project are:

- To ensure secure and transparent digital voting.
- To utilize blockchain for decentralized vote storage and immutability.
- To increase voter accessibility via web interfaces.
- To eliminate fraud and tampering risks.
- To enable Real-Time and Transparent Results Tracking.

1.7 Review of Literature

"DigiVoter: Blockchain Secured Digital Voting Platform with Aadhaar ID Verification" [1]

Description: The paper presents a novel voting system named 'DigiVoter' that combines biometric fingerprint verification with blockchain technology to ensure secure, tamper-proof, and transparent digital elections. The voter's fingerprint is matched against a secure Aadhaar database to verify identity. A unique Aadhaar hash is generated using SHA-512, which is then combined with the vote choice (Hex ID of the political party) to create a Vote Hash. This Vote Hash becomes the data in a blockchain block. Each node in a peer-to-peer network validates the transaction before adding it to the blockchain. The decentralized structure ensures resistance to manipulation or fraud. A radix tree is used to prevent double voting. The entire process is transparent, and only encrypted data is stored to preserve voter privacy.

Remark: The research paper forms a robust base for developing blockchain-based voting systems, especially in the Indian context. It introduces an effective fusion of Aadhaar-based identity validation and blockchain immutability. The implementation, through NodeJS and Python, makes it technically replicable for academic and real-world projects. The DigiVoter platform is highly scalable, secure, and eliminates the possibility of fake or multiple votes, making it ideal for transparent electoral processes at national and regional levels.

Year of Publication: 2022

“DigiVote: Voting System Using Blockchain” [2]

Description: The paper proposes DigiVote, a decentralized e-voting system built on the Ethereum blockchain to ensure secure, transparent, and tamper-proof elections.

Instead of a centralized server, the system uses smart contracts to store immutable voting records. It includes an Election Commission module for managing elections and a Voter module for registration, verification, and vote casting through MetaMask on a private Ethereum network (Ganache). Smart contracts are written in Solidity, and the system is developed using HTML, CSS, JavaScript, PHP, Python, and Truffle Suite following an Iterative/Agile SDLC. DigiVote prevents double voting, maintains voter anonymity, and provides real-time transparency.

Remark: The study demonstrates the potential of blockchain to improve election security and transparency through decentralization. Its modular design and Ethereum-based implementation make it technically strong and scalable. However, relying on a private blockchain and the absence of Aadhaar/NID-based authentication limit real-world applicability. Despite these gaps, DigiVote offers a solid foundation for future improvements such as public blockchain deployment, stronger identity verification, and mobile integration.

Year of Publication: 2022

“Design and development of a decentralized voting system using blockchain” [3]

Description: The paper presents a decentralized e-voting system built on the Ethereum blockchain to improve security, transparency, and voter trust. Using tools such as Truffle, Ganache, MetaMask, and Solidity, the system replaces centralized voting with smart-contract- based, immutable recordkeeping. The voting workflow includes voter verification, vote casting through MetaMask, real-time result updates, and transparent, tamper-proof result display. The development follows the Waterfall SDLC model, supported by system diagrams and interface screenshots that illustrate the complete lifecycle of the application.

Remark: The work effectively demonstrates how blockchain can address traditional voting issues such as tampering and lack of transparency. Its use of Ethereum smart contracts and a clear Waterfall development process strengthens the technical implementation. However, the absence of national ID verification limits real-world applicability, and large-scale performance testing is not explored. Security analysis remains brief, especially regarding client-side risks. Despite these limitations, the system provides a solid baseline for decentralized e-voting, with future potential for oracle-based ID validation and enhanced authentication mechanisms.

Year of Publication: 2023

"Voting and Election System Using Blockchain Technology" [4]

Description: This paper proposes a blockchain-based online voting system that utilizes DigiLocker for authentication and live webcam verification. It addresses vulnerabilities in the traditional Indian voting system, such as vote tampering, booth capturing, and low accessibility. Voters use their DigiLocker credentials for identity verification and cast their votes via a secure website. The system ensures data immutability by encrypting and storing vote data on a blockchain network, accessible only to the Election Commission. The authors detail the methodology including hardware/software requirements, step-by-step voting procedures, security protocols, and fail-safe mechanisms. A pseudocode and system algorithm were also proposed to demonstrate the technical flow.

Remark: The paper clearly establishes blockchain as a secure and scalable solution for modernizing elections in India. The use of real-time verification and encrypted vote storage improves trust, accessibility, and transparency.

Year of Publication: 2023

1.8 Organization of Report

The report is organized into the following chapters to present the complete development of the iBallot system in a structured manner:

Chapter 1: Introduction This chapter describes the background of electronic voting systems, motivation behind the project, problem statement, scope, and objectives of implementing blockchain technology for secure and transparent voting.

Chapter 2: System Requirements Specification This chapter outlines the hardware, software, functional and non-functional requirements necessary for the development and execution of the iBallot system.

Chapter 3: Detailed Design This chapter explains the architecture of the proposed system along with design considerations, workflow diagrams, use case modeling, module specifications, flowcharts, and dataflow descriptions that represent the internal working of iBallot.

Chapter 4: Implementation This chapter details the technologies used (ReactJS, Node.js, Express.js, Solidity, PostgreSQL), backend API development, smart contract deployment, user interface implementation, and overall integration of the system components.

Chapter 5: System Testing This chapter describes the testing strategies, unit and integration test cases, test results, and verification of the system functionalities to ensure iBallot performs securely and efficiently.

Chapter 6: Results and Discussions This chapter includes the experimental outcomes with screenshots of system modules such as voter dashboard, admin panel, and blockchain-based vote confirmation, along with the discussion on performance and transparency improvements.

Chapter 7: Conclusion and Future Enhancements This chapter summarizes the achievements of the project and highlights potential enhancements.

1.9 Summary

This chapter introduces the iBallot project and explains the need for a secure and transparent online voting system using blockchain technology. It highlights the drawbacks of traditional and centralized electronic voting systems and emphasizes how decentralization improves trust, accessibility, and tamper-resistance. The chapter also discusses the motivation behind the system, the core problem addressed, the scope of the proposed solution, and the key objectives that guide the design and implementation of iBallot.

CHAPTER 2

SYSTEM REQUIREMENT SPECIFICATION

The **System Requirement Specification (SRS)** defines the detailed description of the proposed system. It provides a clear and precise understanding of the system's requirements to ensure that the developed system meets the intended goals. It specifies the hardware and software prerequisites, as well as the functional and non-functional requirements necessary for implementation.

2.1 Hardware Requirements

Table 2.1 : Hardware Requirements

Component	Specification
Processor	Quad Core i5 (7th Generation) or above
RAM	8GB or higher
Speed	2.5GHz or higher
Storage	10GB or higher
Internet Connectivity	Required for blockchain node synchronization

2.2 Software Requirements

Table 2.2 : Software Requirements

Category	Specification
Programming Language	JavaScript (ES6+), Solidity
Front-End Framework	React.js + Tailwind CSS + Vite
Back-End Framework	Node.js with Express.js
Database	Neon PostgreSQL (Cloud)
Blockchain Layer	Ethereum Smart Contracts (Solidity)
Blockchain Network	Polygon Amoy Testnet
Authentication	JWT Authentication + Encrypted Custom Auth
Storage	Multer (Image/File upload handling)
Smart Contract Wallet	MetaMask Browser Wallet
Development Tools	VS Code, Hardhat, MetaMask

Real-Time Communication	WebSockets (WS)
Deployment & Orchestration	Docker & Docker Compose
Configuration Management	Doppler (Environment & Secrets Management)
Operating System Support	Windows 10 / Ubuntu 20.04 or above
Version Control	Git + GitHub

2.3 Prerequisites

Table 2.3 : Prerequirements

Requirement	**Version / Details**
Node.js	v16 or higher
PostgreSQL / Neon DB	Latest cloud-hosted or local instance
MetaMask	Latest browser extension
Local Blockchain	Hardhat
IDE	Visual Studio Code

2.4 Functional Requirements

- The user registers with the phone number and the username.
- Voter identity is verified using the Mock DigiLocker system.
- Once verified, a unique voter hash is generated and stored on the blockchain.
- Admin creates and manages elections, defining candidates, symbols, and voting duration.
- Voters can log in, view active elections, and cast their vote securely through smart contracts.
- Votes are recorded immutably on the blockchain and System will prevent double voting.
- After the election ends, results are automatically computed and displayed to the admin.

2.5 Non-Functional Requirements

- **Security Requirements:** The system must ensure data integrity and immutability using blockchain encryption and hash-based verification.

- **Scalability Requirements:** The platform should support multiple concurrent elections and voter participation without system lag.
- **Extensibility Requirements:** The system should be flexible enough to incorporate new modules such as biometric authentication or decentralized identity verification.
- **Reliability Requirements:** The blockchain network must ensure fault tolerance and recovery from node failures without data loss.

2.6 Summary

This chapter outlines the essential hardware and software requirements needed to implement the iBallot system. It also specifies the key functional operations and non-functional needs such as security, reliability, scalability, ensuring that the system supports a secure and efficient blockchain-based voting process.

CHAPTER 3

DETAILED DESIGN

This chapter introduces the detailed design of the iBallot system, a decentralized voting platform based on blockchain technology. It explains how the proposed system is designed to meet the functional and non-functional requirements identified earlier, with a focus on security, transparency, and scalability. The detailed design provides a clear understanding of the internal structure of the system and serves as a foundation for its implementation.

The chapter describes the overall architectural approach adopted for iBallot and explains how various system components interact with each other. It discusses the design methodology, system architecture, and module organization used to support secure voter authentication, vote casting, result computation, and admin- istrative control. Design representations such as architecture diagrams, data flow diagrams, and state charts are included to illustrate the flow of data and control within the system.

3.1 Design Methodology

iBallot employs a hybrid, layered architecture combining on-chain and off-chain components to achieve security, scalability, and user accessibility. The design follows the following key principles:

- **Modularity:** The system is separated into distinct, independently deployable modules (voter registra- tion, vote casting, result computation, admin management) that communicate via well-defined APIs and interfaces.
- **Layered Architecture:** The design consists of four logical layers—presentation (React frontend), ap- plication (Node.js/Express APIs), data storage (PostgreSQL for metadata, blockchain for immutable votes), and blockchain (Solidity smart contracts on Polygon Amoy testnet).
- **Hybrid On-Chain / Off-Chain Design:** Critical voting data (vote hashes, voter eligibility checks, double-voting prevention) lives on then blockchain for immutability and auditability, while supplemen- tary data (candidate profiles, election metadata, audit logs) resides in PostgreSQL for efficient querying and administration.
- **Abstraction and User-Friendliness:** The system abstracts blockchain wallet complexity behind a web interface, hiding MetaMask and gas fees from end-users through a backend relayer model.

This approach balances the immutability and transparency guarantees of blockchain with the performance, flexibility, and regulatory manageability of centralized databases.

3.2 System Architecture

The iBallot system follows a four-layer hybrid architecture that separates user interaction, application logic, data storage, and blockchain execution. The architecture of system is shown in Figure 3.1.

Layer 1 – Presentation (Frontend) A React-based single-page application provides a mobile-responsive interface for voters and administrators, communicating only with the backend API so users never handle wallets or cryptocurrency directly.

Layer 2 – Application Logic (API Layer) A Node.js & Express backend manages authentication with JWT, integrates with the mock DigiLocker service for identity verification, checks voter eligibility, processes vote-casting requests, manages elections and candidates, and exposes WebSocket endpoints for real-time result updates.

Layer 3 – Data Storage PostgreSQL stores off-chain data such as users, elections, candidates, hashed identifiers, and audit logs, while the blockchain records each vote as an immutable transaction with one-vote-per-voter enforced using voter hashes and nonces.

Layer 4 – Smart Contracts Solidity contracts on the Polygon Amoy testnet implement castVoteMeta for gasless, relayer-submitted voting, getResults for on-chain tally retrieval, and emit VoteCast events that drive real-time transparency dashboards.

3.2.1 Use Case Diagram

The Use Case Diagram is as depicted in Figure 3.2 visualizes the interactions between the Voter, Admin, and the automated System.

- **Voter**: The voter's primary actions include Login and Register Vote. A critical step is Verify Voter Identity, where the system authenticates the user via the Mock DigiLocker system to ensure eligibility. Once verified, voters can View Candidate Information and securely Cast Vote.
- **System:** This actor represents the backend logic that enforces rules. It handles Verify Voter Identity to generate unique voter hashes and prevent double voting. During voting, it performs Confirm Vote Submission to record the transaction on the blockchain and triggers Notify User of Success Vote.
- **Admin:** The admin manages the platform through the Administer Voting Process, which involves creating elections and defining candidates. They also utilize the Generate Vote Report use case to compute and display final results from the immutable ledger.

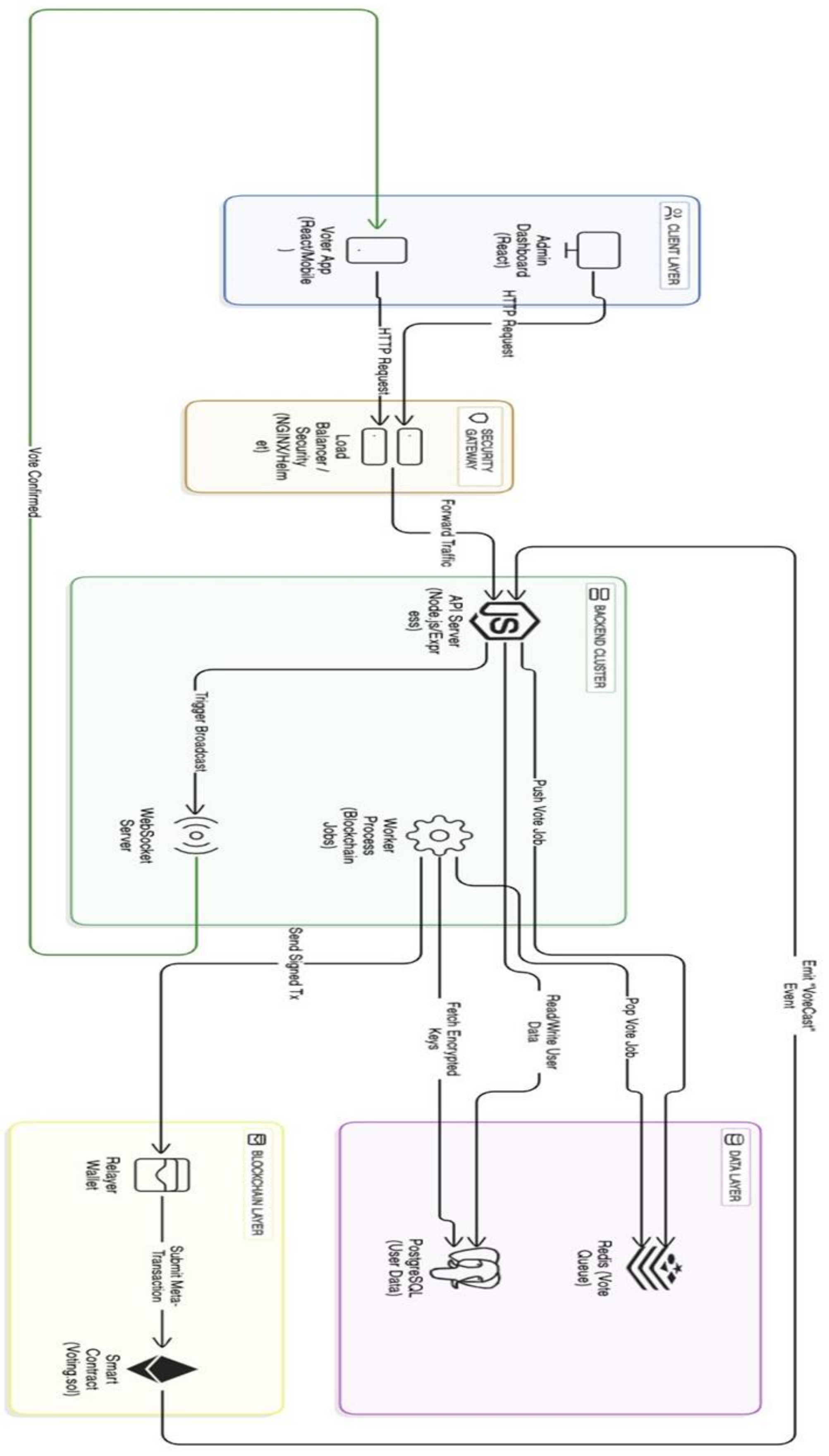


Figure 3.1: Detailed Architecture Diagram

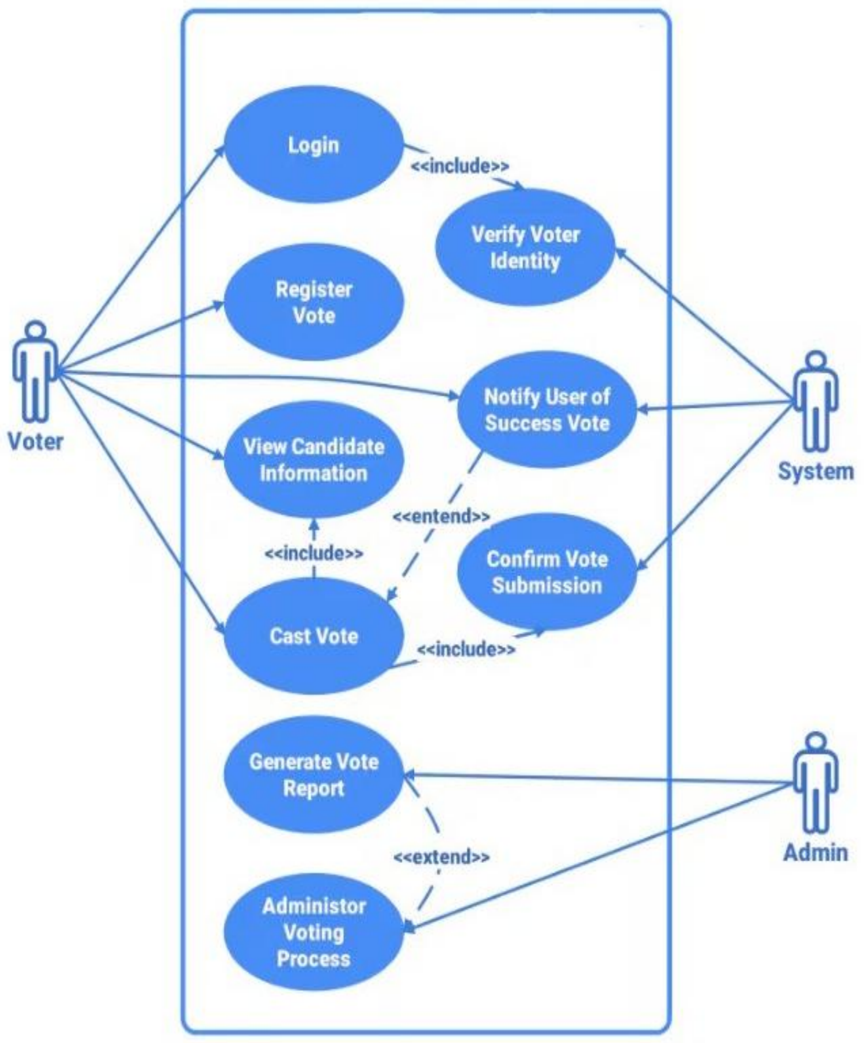


Figure 3.2: Use Case Diagram

3.3 Module Specification

This subsection specifies the system modules, their responsibilities, and their interfaces.

3.3.1 Voter Registration and Authentication Module

- **Responsibilities:** Collect voter details, perform identity verification using the DigiLocker mock, generate and store hashed voter identifiers, and provide two-factor authentication.
- **Interfaces:** Exposes registration APIs and verification endpoints, and integrates with PostgreSQL for persisting voter records.

3.3.2 Blockchain Recording Module

- **Responsibilities:** Interact with smart contracts to record votes as immutable transactions, handle transaction confirmations, and emit events off-chain listeners.

- **Interfaces:** Connects to Solidity smart contracts, the backend blockchain adapter, and node providers or a local blockchain network.

3.3.3 Admin Validation and Election Management Module

- **Responsibilities:** Allow admins to create and manage elections, authorize candidates, publish ballot metadata, and review audit logs.
- **Interfaces:** Provides an admin API secured with token-based access, a management UI, and integrations with smart-contract functions for the election lifecycle.

3.3.4 Result Computation Module

- **Responsibilities:** Query the blockchain ledger, aggregate votes, apply business rules such as thresholds and tie-breakers, and publish election results.
- **Interfaces:** Exposes a Result Service API and uses a caching layer for fast result retrieval

3.4 Data Flow Diagrams

3.4.1 Data Flow Diagram – Voter Authentication Module

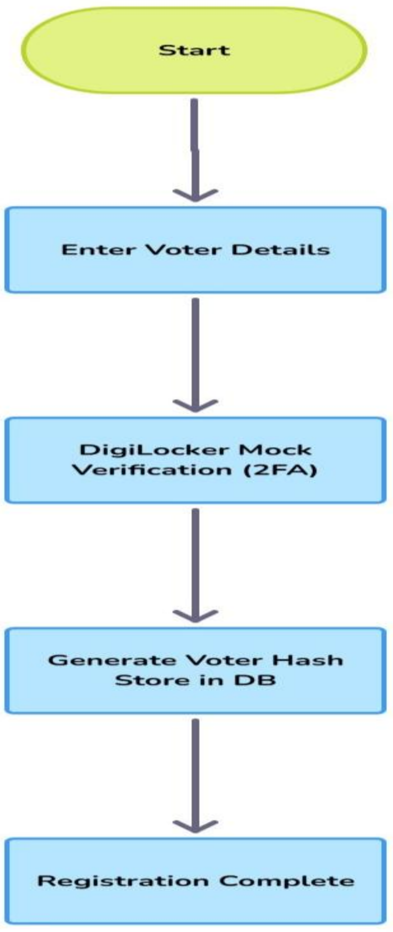

Start
Enter Voter Details
DigiLocker Mock Verification (2FA)
Generate Voter Hash Store in DB
Registration Complete

Figure 3.3: Flow Chart for Voter Authentication Module

The Figure 3.3 shows how a voter's login/registration request flows from the React frontend to the API, which validates credentials and identity using the users and digilocker_mock_data tables before issuing a JWT token and voter hash back to the frontend.

3.4.2 Data Flow Diagram – Voter Registration Module

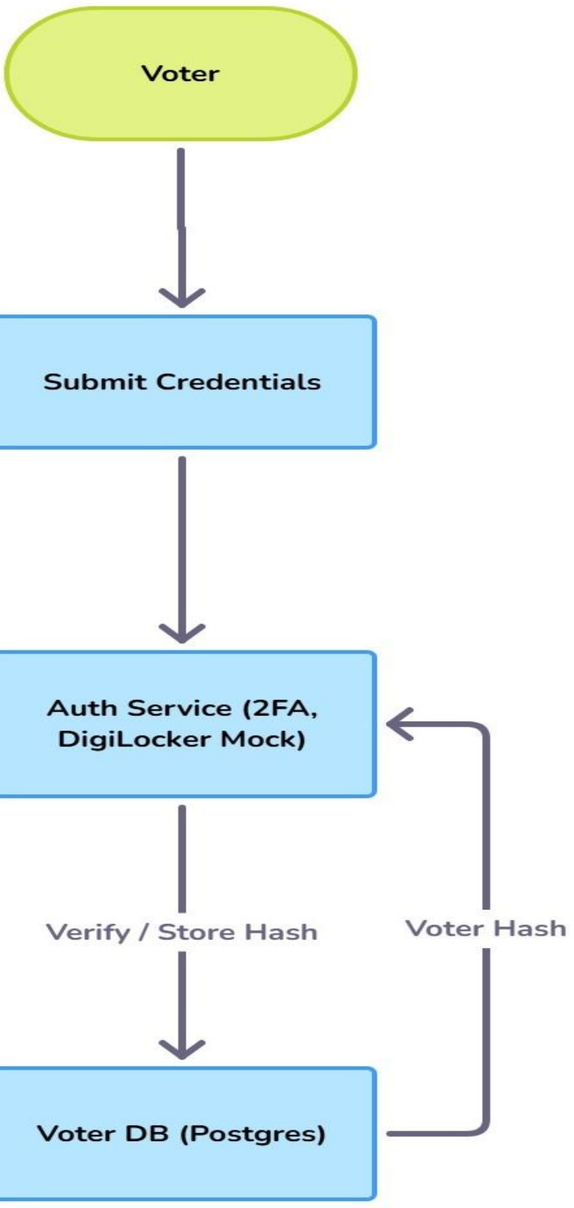


Figure 3.4: Flow Chart for Voter Registration Module

The Figure 3.4 depicts the voter registration flow, where the frontend sends registration data to the API, which stores the user in the users table, fetches UID details from digilocker_mock_data, hashes the UID into eci_admin_data, and confirms successful registration to the frontend.

3.4.3 Data Flow Diagram – Result Computation Module

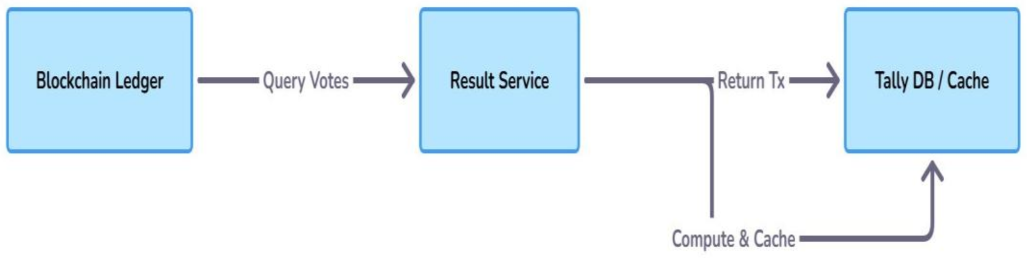


Figure 3.5 : Flow Chart for Result Computation Module

The Figure3.5 illustrates how the admin's result request is sent from the frontend to the API, which calls the smart contract's get Results function on the blockchain, optionally caches tallies in PostgreSQL, and returns final candidate-wise results to the admin dashboard.

3.5 Summary

The high-level design ensures security, transparency, and scalability by combining on-chain immutability with off-chain storage for performance. Modules are separated for maintainability and allow independent scaling.

CHAPTER 4

IMPLEMENTATION

Implementation is the phase of the project where the theoretical designs and system architectures are translated into functional code. This chapter details the technical implementation of the iBallot system, a secure, blockchain-based e-voting platform.

The primary goal of iBallot is to provide a transparent, immutable, and verifiable voting process. To achieve this, the system adopts a multi-tier architecture consisting of a modern React frontend, a Node.js/Express backend, a PostgreSQL database for user and election management, and Solidity smart contracts on an Ethereum-compatible blockchain for core voting and tallying.

This chapter covers:

- The system's overall architecture.
- The technology stack chosen for each component.
- The detailed implementation of the backend server and its REST APIs.
- The structure and role of the blockchain smart contract.
- The development of the frontend user interface.
- The containerization and configuration strategy for deployment using Docker and Doppler.

4.1 System Architecture

The iBallot system is designed as a decoupled client–server application to ensure scalability, maintainability, and clear separation of concerns. The architecture consists of four main layers.

4.1.1 Frontend (Client Side)

A React-based single-page application (SPA) that provides the user interface for both voters and administrators, including ballot views, result dashboards, and admin management screens.

4.1.2 Backend (Server Side)

A Node.js and Express.js server that exposes RESTful APIs, implements business logic, handles authentication and authorization, manages elections, and orchestrates communication with the blockchain layer.

4.1.3 Data and Storage Layer

Relational Database (PostgreSQL): Stores non-immutable data such as voter profiles, election con- figurations, candidate information, and eligibility records.

File and Image Storage: Uses Multer in the Node.js backend for handling file uploads such as candidate images or election documents, which are stored off-chain.

4.1.4 Blockchain Layer

Solidity smart contracts deployed on an Ethereum-compatible network (such as Ganache during devel- opment) are responsible for critical operations including casting votes, enforcing one-vote-per-voter, and tallying results in an immutable ledger.

Multi-factor or document-based verification is implemented through a mock DigiLocker-style authen- tication flow and token-based access control for admins, while browser-based wallets such as MetaMask are used for interacting with the blockchain where appropriate.

4.2 Technology Stack

The technology stack is selected to satisfy security, performance, and developer productivity requirements while enabling containerized deployment and centralized secrets management.

4.2.1 Backend Technologies

- **Runtime:** Node.js (JavaScript ES6+).
- **Framework:** Express.js for building RESTful APIs.
- **Database:** PostgreSQL as the primary relational database.
- **Blockchain Interface:** Web3.js or Ethers.js (via the blockchain adapter, e.g., contract.js) to communicate with the Solidity smart contracts on Ethereum or Ganache.
- **Authentication and Security:** JSON Web Tokens (JWT) for session management, bcrypt for password hashing, and a mock DigiLocker verification API for identity checks; additional utilities are used for encrypting sensitive payloads where required.
- **File Handling:** Multer for secure file and image upload handling on the server.
- **Configuration and Secrets:** Doppler is used to manage environment variables and secrets across de- velopment and deployment environments.
- **Testing:** Jest and Supertest for unit and integration testing of backend services.

4.2.2 Frontend Technologies

- **Library:** React.js for building a responsive single-page interface.

- **Routing:** React Router DOM for client-side navigation.
- **State Management:** React Context API for managing authentication, verification, and election state.
- **Styling:** Tailwind CSS for utility-first, responsive styling.
- **API Communication:** Axios (or the native Fetch API) for HTTP communication with the backend.
- **Wallet and Blockchain Integration:** MetaMask for managing user wallets and signing blockchain transactions in the browser.

4.2.3 Blockchain

- **Smart Contract Language:** Solidity.
- **Execution Platform:** Ethereum-compatible blockchain (e.g., Ganache for local development and testing).
- **Artifacts:** Compiled contract bytecode and ABI (e.g., Voting.json) used by the backend and frontend to interact with the contract.

4.2.4 Deployment and Tooling

- **Containerization:** Docker for packaging the frontend, backend, and supporting services, with Docker Compose orchestrating multi-container setups in development and test environments.
- **Configuration Management:** Doppler CLI integrated into Docker and CI workflows for injecting secrets and environment variables at runtime without exposing them in source control.
- **Operating Systems:** Development and deployment are targeted for Windows 10 and Ubuntu 20.04 or later.
- **Development Tools:** Visual Studio Code as the primary IDE, along with Truffle or similar tools for compiling and migrating smart contracts, Ganache for local blockchain simulation, and MetaMask for browser-based wallet integration.

4.3 Backend Implementation

The backend server, built with Node.js and Express, is the system's operational core. It handles all business logic, from user registration to interfacing with the smart contract.

4.3.1 Server and API Setup

The main backend entry point (server.js) loads the Express application from app.js, creates an HTTP server, initializes the WebSocket server for real-time updates, and starts listening for smart-contract vote events to broadcast them to connected clients.

4.3.2 Database Schema

The PostgreSQL database stores relational data required for application and election management. The schema is defined in backend/dump.sql. Key tables include:

- **users**: Stores login information for registered platform users, including id, username, uid_hash, password, created_at, and last_login timestamps.
- **elections:** Stores election metadata such as election_id, name, type, start_time, end_time, enabled_constituencies, winner_party_name, and created_at.
- **candidates:** Stores candidate information linked to a specific election and constituency, including election_id, candidate_id, candidate_name, party_name, symbol, and constituency_id.
- **eci_admin_data:** Stores hashed identifiers and metadata derived from official ECI voter rolls, including uid_hash, constituency details, encrypted admin private keys, and associated wallet addresses for blockchain operations.
- **digilocker_mock_data:** Stores mock e-KYC records used by the authentication service, including phone number, UID, date of birth, and full name, which are used to simulate DigiLocker-style identity verification.

4.3.3 API Routes and Modules

The backend API is modularized, with separate routes for user-facing flows and admin operations. All sensitive endpoints are protected using JSON Web Tokens (JWT) and role-based checks.

4.3.4 User Authentication Module

- **Registration:** The user submits basic details and credentials. The backend hashes the password using bcrypt (via hashUtils.js) and creates a record in the users table. A mock DigiLocker/OTP or similar verification step can be triggered to validate the user's phone and identity before treating the account as fully verified.
- **Login:** The user submits credentials. The backend compares the stored password hash using bcrypt. On success, a signed JWT is issued that contains the user identifier and role claims, which the client attaches to subsequent requests.

4.3.5 Security and Middleware

- **userAuth.js**: This middleware intercepts requests to protected user routes.
- It reads the JWT from the request headers (e.g., Authorization or x-auth-token), verifies it using process.env.JWT_SECRET, and attaches the decoded payload to req.user before forwarding the request.

- **hashUtils.js:** A utility module that encapsulates password hashing and comparison using bcrypt, ensuring consistent security practices across registration and login flows.

4.3.6 Admin Authentication Module

The admin login flow now uses a two-step mechanism based on a static bootstrap token and a short-lived JWT:

- The admin submits a one-time or configuration token (ADMIN_TOKEN) from the login form.
- The backend compares the submitted token with process.env.ADMIN_TOKEN. If it matches, the server issues a new JWT with an admin role claim (for example, { role: "admin" }) signed with process.env.JWT_SECRET and an expiration time (e.g., one hour).
- The response returns this JWT to the frontend, which then attaches it to subsequent admin requests in the Authorization header.

Admin routes under routes/admin/ are protected using a dedicated middleware (for example, adminAuth.js) that first validates the JWT and then checks that req.user.role === 'admin' before allowing access.

4.3.7 Voter Module: User Route Handlers

The following user-facing route handlers are implemented under routes/user/ and are protected using the userAuth middleware.

- dashboard.js: Returns a list of active elections that the authenticated voter is eligible for, using constituency-based filtering from eci_admin_data.
- candidateList.js: Given an election_id, this route retrieves all associated candidates from the
- candidates table and returns them to the frontend.
- vote.js: Implements a secure meta transaction based voting workflow:
  - Authenticates the user against the users table and retrieves their hashed identifier (uid_hash).
  - Obtains encrypted voter keys and constituency mapping (ac_id / pc_id) from eci_admin_data to validate election eligibility.
  - Decrypts the user's private key locally and creates a temporary ECDSA wallet instance using Ethers.js.
  - Calls contract.getNonce() to ensure replay protection on-chain.
  - Generates a signed payload using solidityPackedKeccak256 with:

  - election_id
  - candidate_id
  - constituency ID
  - user voter hash
  - current nonce
  - signature deadline
- The backend relayer wallet (configured in contract.js) submits the transaction by calling: castVoteMeta (electionId, voterHash, candidateId, constituencyId, deadline, signature)
- Upon transaction confirmation, a JSON success response is returned containing the txHash for blockchain auditability.

4.3.8 Admin Module: Route Handlers

The following administrator-specific route handlers are implemented under routes/admin/, and all are se- cured using the adminAuth middleware, which verifies the admin JWT before granting access:

- elections.js: Enables administrators to create, update, list, and disable elections by operating on the elections table, including linking each election to a deployed blockchain contract address and speci- fying enabled constituency configuration.
- candidates.js: Manages candidate records associated with each election by inserting and updating entries in the candidates table, including party affiliations and constituency assignments.
- results.js: Retrieves election results by invoking the getResults function (or equivalent) in contract.js to read tallies directly from the blockchain. When applicable, the winning party name is stored in elections.winner_party_name for faster lookup.
- eciData.js: Manages administrator-controlled records in eci_admin_data, including importing hashed voter identifiers and constituency mappings derived from official ECI datasets. These records are used to determine voter eligibility and link blockchain wallet details for secure participation.

4.3.9 Blockchain Interface (blockchain/contract.js)

The blockchain/contract.js module acts as the bridge between the Node.js backend and the Ethereum- compatible blockchain. It uses the Ethers.js library (specifically JsonRpcProvider, Wallet and Contract) to connect to a JSON-RPC node, load the deployed contract and subscribe to vote events.

Configuration is provided via environment variables:

- RPC_URL: URL of the blockchain JSON-RPC endpoint.
- CONTRACT_ADDRESS: Deployed address of the voting smart contract.
- RELAYER_PRIVATE_KEY: Private key used by the backend “relayer” wallet to sign transactions and read on-chain data.

At startup, the module:

1. Loads the ABI from Voting.json.
2. Creates a JsonRpcProvider using RPC_URL.
3. Validates and normalizes CONTRACT_ADDRESS.
4. Instantiates a Wallet using RELAYER_PRIVATE_KEY and attaches it to the provider.
5. Constructs an Ethers.js Contract instance that is exported for other modules to use.

The key exported function is startVoteListener(broadcast), which registers a listener on the smart contract’s VoteCast event. Whenever a vote is cast on-chain, the listener constructs a JSON payload containing the election, constituency and
candidate identifiers, along with a timestamp, and forwards it to the supplied broadcast callback. The WebSocket layer uses this callback to push real-time VOTE_UPDATE messages to connected admin dashboards.

4.3.10 Deployment and Containerization

To ensure consistent and reproducible deployments, the iBallot system is fully containerized using Docker. The docker-compose.yml file in the project root defines services for the complete application stack.
Services defined in docker-compose.yml:

- frontend: Builds the React application using its Dockerfile and serves the static assets.
- backend: Builds and runs the Node.js API server using its Dockerfile.
- db: Runs a PostgreSQL instance and initializes the schema using the backend/dump.sql file.

This setup simplifies local development, testing and deployment, as the complete stack can be started with a single docker-compose up command

4.4 Frontend Implementation

The frontend of the iBallot system is built as a Single Page Application (SPA) using React.js. It serves as the secure interface for voters to verify their identity, view eligible elections, and cast ballots.

4.4.1 Application Structure and Routing

The application entry point (main.jsx) initializes the React DOM and wraps the entire component tree with the VerificationProvider to manage global authentication state.

Routing is handled by react-router-dom in App.jsx. To ensure faster load times, the application utilizes Lazy Loading and React.Suspense. This ensures that heavy components (like the Dashboard or Voting Interface) are only loaded when the user navigates to them, reducing the initial bundle size.

4.4.2 Global State Management

Instead of passing props through multiple layers, the system uses the Context API (VerificationContext.jsx) to manage the multi-step registration flow.

- **Role:** It temporarily holds sensitive data such as the verificationData returned from the DigiLocker mock and the isVerified status.
- **Workflow:** This allows the data to persist seamlessly as the user navigates from the "DigiLocker Verification" page to the "Final Registration" form without losing their verified status.

4.4.3 Voter Registration Module

` The registration module (Register.jsx) enforces strict validation rules before data is sent to the backend.

- **Identity Verification:** The system checks the isVerified flag from the context. If false, it blocks registration and redirects the user to the DigiLocker verification page.
- **Input Validation:** It implements real-time regex-based validation for passwords (requiring uppercase, lowercase, numbers, and special characters) and ensures the username is alphanumeric.
- **Submission:** On successful validation, it sends the username, password, and the DigiLocker-verified UID to the backend to generate the voter's blockchain wallet.

4.4.4 Authentication and Session Management

The login module (Login.jsx) handles secure user authentication and session initiation.

- **Credential Check:** It sends the username and password to the API.
- **Secure Storage**: Upon a successful response, it stores the JWT Token and critical

voting metadata in the browser's sessionStorage. This includes the uid_hash (User ID Hash), voterHash, and walletAddress.

- **Constituency Mapping:** The user's constituency data (ac_id and pc_id) is also cached in the session to automatically filter relevant elections in the dashboard.

4.4.5 Dashboard and Election Navigation

The dashboard (Dashboard.jsx) serves as the central hub for the voter.

- **Dynamic Filtering:** It fetches the list of active elections and filters them based on the user's eligibility (Constituency ID).
- **Status Indicators:** It displays real-time status tags (Live, Upcoming, Completed) and prevents users from accessing elections they have already voted in by checking the hasVoted flag.
- **Responsive UI:** It features a collapsible sidebar and a mobile-friendly menu (isMobileMenuOpen state) to ensure accessibility on smaller screens.

4.4.6 Secure Voting Interface

The voting module (CandidateList.jsx) is the most critical component, handling the interaction between the user and the blockchain.

- **Candidate Retrieval:** It fetches the official candidate list for the specific electionId and assemblyId, sorting them to ensure "NOTA" (None of the Above) always appears at the bottom.
- **Vote Casting:** When a user selects a candidate and enters their password, the frontend sends a secure request to the backend. The backend signs the transaction and queues it.
- **Concurrency Control:** A local isVoting state locks the interface immediately after submission to prevent multiple clicks or double-voting attempts while the transaction is processing**.**

4.4.7 Real-Time Blockchain Feedback (WebSocket)

To provide transparency and immediate confirmation, the frontend maintains a persistent WebSocket connection (ws://) to the backend.

- **Event Listening:** The CandidateList component listens for the VOTE_CONFIRMED event broadcasted by the blockchain listener (Section 4.3.9).

- **Hash Verification:** When a confirmation arrives, the client performs a cryptographic check:
    1. It retrieves the local uid_hash from sessionStorage.
    2. It compares it against the voterHash received in the WebSocket payload.
    3. If the hashes match, it confirms that *this specific user's* vote was included in a block.
- **Success Feedback:** Upon verification, the system displays a success modal containing the Transaction Hash (TxHash), allowing the user to independently verify their vote on the blockchain explorer.

4.5 Pseudocode

4.5.1 Frontend Implementation

This section outlines the algorithmic logic for the key frontend modules: User Registration, Authentication, and the Voting Interface.

(a) User Registration (Register.jsx)

```
Begin
    Import React, useState, axios, useNavigate
    Import useVerification from Context

    Initialize State:
        confirmPassword, loading, error
        validation flags (touched)

    Retrieve Verification Context:
        username, password, phoneNumber, verificationData, isVerified

    Function handleRegister(event)
        Prevent default form submission behavior

        If isVerified is False Then
            Set Error "Please verify identity via Digilocker first"
            Return
        End If
  If password is not equal to confirmPassword Then
```

```
   Set Error "Password is required"
            Return
        End If
          Set isVoting to True

        Try
            Send POST request to "/api/vote" with:
                username, password, electionId, candidateId

            Log "Vote Queued. Waiting for Blockchain confirmation..."

        Catch Error
            Set isVoting to False
            Set Error message from response
    End Function

    Lifecycle on Mount
        Call fetchCandidates()
        Call connectWebSocket()

    Render UI
        If Loading Return Spinner
        Display Election Header

        List Candidates
            For each candidate
                Render Symbol and Name
                On Click -> Set selectedCandidate

        Input Password for Confirmation
        Button "Submit Final Vote" (Disabled if processing)

        If showSuccessModal is True Then
            Render VoteSuccessModal with txHash
End
```

(b) Login.jsx (Authentication Module)

```
Begin
  Import React, useState, axios, useNavigate, toast
  Import Context variables (username, setUsername, password, setPassword)

  Initialize State: loading, error

  Function handleSubmit(event)
    Prevent default form submission
    Set Loading to True
    Clear previous errors

    Try
      Send POST request to "/api/login" with { username, password }

      Receive Response data

      Store in SessionStorage:
        "token" <- response.token
        "user" <- JSON string (username, hasVoted, uid_hash)
        "voterHash" <- response.voterHash
        "walletAddress" <- response.walletAddress
        "constituency" <- JSON string (response.constituency)

      Display Success Toast "Login successful! Redirecting..."
      Wait 1500ms
      Navigate to "/dashboard"

    Catch Error
      Log error details
      Set Error "Login failed. Please check your credentials."
      Display Error Toast
      Set Loading to False
  End Function
```

```
   Render Login Form
      Input Username
   Input Password
         Link "Forgot password?"
         Link "Register"
         Button Sign In (Disabled if Loading)
End
```

(c) CandidateList.jsx (Voting & Blockchain Confirmation)

```
Begin
   Import React, useState, useEffect, useRef, axios, WebSocket
   Get electionId, assemblyId from URL Parameters

   Initialize State:
      candidates list, selectedCandidate, isVoting, password
      showSuccessModal, txHash, error

   Function user()
      Return parsed "user" object from SessionStorage

   Function connectWebSocket()
      Determine WebSocket URL based on environment (Development/Production)
      Open WebSocket connection to Backend

      On Message Received (event)
         Parse JSON data from event

         If data.type equals "VOTE_CONFIRMED" Then
            Get localHash from SessionStorage (user.uid_hash)
            Get blockchainHash from data.payload.voterHash

            Normalize hashes (lowercase, remove '0x')

            If localHash equals blockchainHash Then
               Set isVoting to False
Set txHash to data.payload.txHash
                  Set showSuccessModal to True
              End If
          End If
      End Function

      Function fetchCandidates()
          If token or user is missing Then
              Set Error "Missing required information"
              Return
          End If

          Try
              Send GET request to "/api/candidates/{electionId}/{assemblyId}"
              Format candidate symbols (append image paths)
              Sort candidates (Place NOTA at bottom)
              Set candidates state
          Catch Error
              If status is 401 (Unauthorized) Then
                  Clear Session
                  Navigate to "/login"
              Else
                  Set Error "Failed to load official candidate list"
      End Function

      Function handleVoteSubmit(event)
          If selectedCandidate is Null Then
              Set Error "Please select a candidate"
              Return
          End If

          If password is Empty Then
              Set Error "Password is required"
              Return
          End If
```

```
Set isVoting to True
  Try
        Send POST request to "/api/vote" with:
          username, password, electionId, candidateId

        Log "Vote Queued. Waiting for Blockchain confirmation..."

      Catch Error
        Set isVoting to False
        Set Error message from response
    End Function

    Lifecycle on Mount
      Call fetchCandidates()
      Call connectWebSocket()

    Render UI
      If Loading Return Spinner
      Display Election Header

      List Candidates
        For each candidate
          Render Symbol and Name
          On Click -> Set selectedCandidate

      Input Password for Confirmation
      Button "Submit Final Vote" (Disabled if processing)

      If showSuccessModal is True Then
        Render VoteSuccessModal with txHash
End
```

4.4.2 Backend Implementation

This section outlines the server-side algorithms for secure identity verification, cryptographic vote processing, and real-time blockchain monitoring.

(a)User Registration (register.js)

```
Begin
  Import express, crypto, bcrypt, ethers, pool, contract
  Import AES encryption utilities

  Route POST "/"
    Read username, password, phoneNumber from request body

    # Step 1: Verification against DigiLocker
    Query database "digilocker_mock_data" for phoneNumber
    If no record found Then
      Return Error "User not verified via Digilocker"
    End If

    Calculate Age from Date of Birth (DOB)
    If Age < 18 Then
      Return Error "Voter must be at least 18 years old"
    End If

    # Step 2: Identity & Security Setup
    Generate SHA256 hash of UID (Aadhaar) -> uidHash
    Check if username or uidHash already exists in "users" table
    If exists Then
      Return Error "User already registered"
    End If
    Hash Password using Bcrypt
    Create new Random Ethereum Wallet (Public/Private Key pair)
    Encrypt Private Key using AES and Secret Salt

    Begin Database Transaction
      Insert into "users" (username, uid_hash, password_hash)
      Update "eci_admin_data" set (enc_private_key, wallet_address) for this uid_hash

    # Step 3: Blockchain Authorization
    Call Smart Contract function authorizeVoter(voterHash, walletAddress)
    Wait for Transaction Confirmation
```

```
     Commit Database Transaction
     Return Success JSON (walletAddress, userDetails)

   Catch Error
     Rollback Database Transaction
     Return HTTP 500 Error
 End
```

(a)Vote Processing (vote.js)

```
 Begin
   Import express, crypto, bcrypt, ethers, pool, Queue (BullMQ)
   Initialize Redis Queue "vote-processing"

   Route POST "/" (Protected by UserAuth Middleware)
     Read password, electionId, candidateId from request body
     Get username from Authenticated Session

     Connect to Database

     # Step 1: Authentication & Data Retrieval
     Fetch User record (password_hash, uid_hash) from "users" table
     Verify Password using Bcrypt
     If Invalid Then Return Error "Invalid Password"

     Fetch Encrypted Private Key & Constituency ID from "eci_admin_data"

     # Step 2: Cryptographic Signing
     Decrypt Private Key using AES
     Initialize Wallet instance with Decrypted Key

     Fetch Nonce from Smart Contract for (electionId, voterHash)
     Set Deadline = Current Time + 1 Hour

     Generate Message Hash (Keccak256) containing:
       [electionId, voterHash, candidateId, constituencyId, nonce, deadline]
```

```
      Sign Message Hash using Wallet -> Generate Signature
    # Step 3: Queueing
      Insert Log into "voter_logs" with status "QUEUED"

      Add Job to Redis Queue "vote-processing" with data:
        (electionId, candidateId, signature, deadline, voterHash)

      Return HTTP 202 "Vote Queued Successfully"

    Catch Error
      Log Error
      Return HTTP 500 Error
    Finally
      Release Database Connection
  End
```

(a)Blockchain Event Listener (contract.js)

```
Begin
  Import ethers, JsonRpcProvider, Contract
  Load Contract ABI and Address from Config

  Function startVoteListener(broadcastCallback)
    Initialize Provider and Connect to Blockchain
    Initialize Contract instance

    # Event Listener
    Subscribe to Contract Event "VoteCast"

    On Event "VoteCast" (electionId, voterHash, candidateId, eventObj):
      Extract Transaction Hash (txHash) from event logs

      # Broadcast 1: Admin Dashboard Update
      Create Payload "VOTE_UPDATE":
        electionId, candidateId, timestamp
```

```
Call broadcastCallback(Payload)

      # Broadcast 2: Voter Confirmation
      Create Payload "VOTE_CONFIRMED":
        electionId, voterHash, txHash

      Call broadcastCallback(Payload)

      Log "Vote Confirmed on Blockchain for Voter: " + voterHash

  End Function
End
```

4.6 Summary

The implementation of iBallot demonstrates a practical and secure integration of a modern web stack with blockchain technology. A React-based interface, a modular Node.js backend and a PostgreSQL data layer work together to manage identity and election information efficiently, while the blockchain smart contract ensures that votes remain tamper-proof and publicly verifiable. Real-time VoteCast event updates and Docker- based deployment further enhance transparency, scalability and maintainability. Overall, this hybrid architecture effectively supports secure, reliable, and auditable electronic voting.

CHAPTER 5

SYSTEM TESTING

System testing is a critical phase that validates the complete and fully integrated iBallot application. Since iBallot deals with secure electronic voting, the testing strategy must be comprehensive, covering everything from small utility functions to full user journeys. The testing approach for iBallot is organized into three levels:

- **Unit Testing:** Verifies the smallest testable units (utility functions, middleware, and isolated modules) in isolation.
- **Integration Testing:** Verifies the interaction between modules, primarily focusing on API endpoints and their interaction with the database and blockchain layer.
- **End-to-End (E2E) Testing:** Validates complete user flows from the voter's and admin's perspectives, from the React frontend through the backend APIs to the database a blockchain.

This chapter explains the tools, strategies, and representative test cases used to ensure that the iBallot system is robust, secure, and behaves according to the specified requirements.

5.1 Test Plan Overview

The testing strategy for iBallot focuses on validating correctness, security, and reliability across the critical modules of the system. The following areas are covered:

- **Unit Testing:** Utility functions (hashing, JWT), middleware (authentication/authorization), and blockchain adapter functions are tested in isolation to verify correct behavior and error handling.
- **Integration Testing:** Endpoints for registration, login, DigiLocker verification, vote casting, and result retrieval are tested against a running PostgreSQL instance and the deployed smart contract to ensure correct interaction between components.

5.2 Testing Frameworks and Tools

- **Backend Test Runner - Jest :** Jest is used as the primary testing framework for the Node.js/Express backend. It provides a test runner, assertion library, mocking capabilities, and support for code coverage.
- **API Integration Testing – Supertest:** Supertest is used together with Jest to perform integration tests on the backend API endpoints. It allows us to send HTTP requests to the Express application and assert on status codes and response bodies.

- **Frontend Testing – Vitest + React Testing Library + Jest-DOM:** For the React-based voter and admin frontends, we use:
  - Vitest as the test runner.
  - React Testing Library (RTL) to render components and interact with them in a way that mimics real users.
  - @testing-library/jest-dom to provide convenient DOM matchers such as toBeInTheDocument, toBeDisabled, etc. These tests focus on verifying page rendering, form inputs, navigation, and state updates (e.g., context, sessionStorage).

5.3 Unit Testing

Unit tests were written for critical security utilities and isolated backend logic. These tests are located under:

backend/__tests__/unit/

They ensure that low-level functionality behaves correctly before being combined in larger flows.

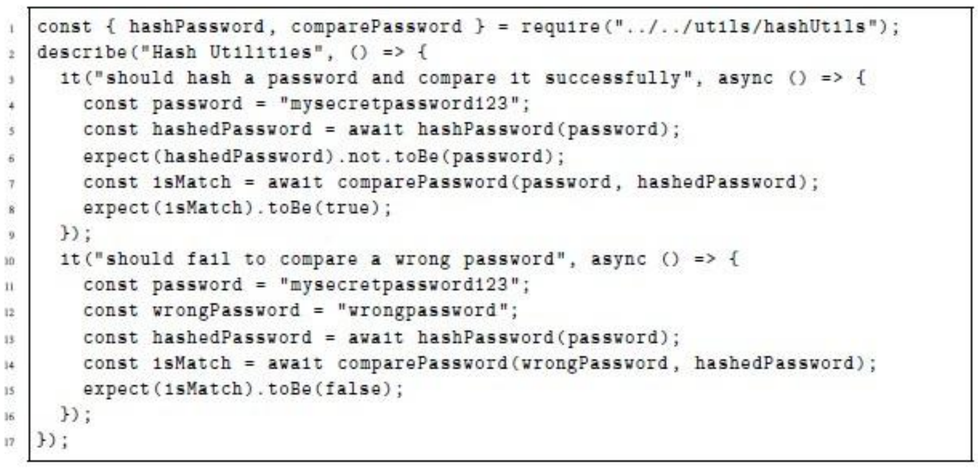
5.3.1 Security Utilities (hashUtils.test.js, aesUtils.test.js)

hashUtils.test.js: These tests verify the correctness of password hashing and comparison using bcrypt-based helpers.

Typical test cases include:

- Test Case 1: Should hash a plain-text password and not return the original password.
- Test Case 2: Should successfully compare a valid password with its stored hash.
- Test Case 3: Should reject an incorrect password when compared to the hash.

  Example (illustrative) unit test snippet:



```
const { hashPassword, comparePassword } = require("../../utils/hashUtils");
describe("Hash Utilities", () => {
  it("should hash a password and compare it successfully", async () => {
    const password = "mysecretpassword123";
    const hashedPassword = await hashPassword(password);
    expect(hashedPassword).not.toBe(password);
    const isMatch = await comparePassword(password, hashedPassword);
    expect(isMatch).toBe(true);
  });
  it("should fail to compare a wrong password", async () => {
    const password = "mysecretpassword123";
    const wrongPassword = "wrongpassword";
    const hashedPassword = await hashPassword(password);
    const isMatch = await comparePassword(wrongPassword, hashedPassword);
    expect(isMatch).toBe(false);
  });
});
```

aesUtils.test.js: These tests validate the AES-based encryption and decryption logic used to protect sensitive data such as private keys.

Typical test cases include:

- Test Case 1: Should encrypt a string and decrypt it back to the original value.
- Test Case 2: Should produce different cipher text (due to IV) even for the same input and key.

5.3.2 Middleware and Access Control (adminAuth.test.js)

adminAuth.test.js: Admin routes in iBallot are protected by an adminAuth middleware that checks a shared secret or token before allowing access. Unit tests for this middleware verify that:

- Test Case 1: A request with a valid admin token is allowed to proceed (next() is called, and res.status is not triggered).
- Test Case 2: A request with an invalid token is blocked and returns HTTP 401 Unauthorized.
- Test Case 3: A request without any Authorization header returns HTTP 401 Unauthorized.

These unit tests help ensure that security-sensitive logic works correctly before it is used n higher-level routes.

5.4 Integration Testing

Integration tests form the most important part of the backend testing strategy. They verify that Express routes, database access, authentication, and blockchain interaction all work together as expected.

All integration tests are located in:

backend/__tests__/integration/

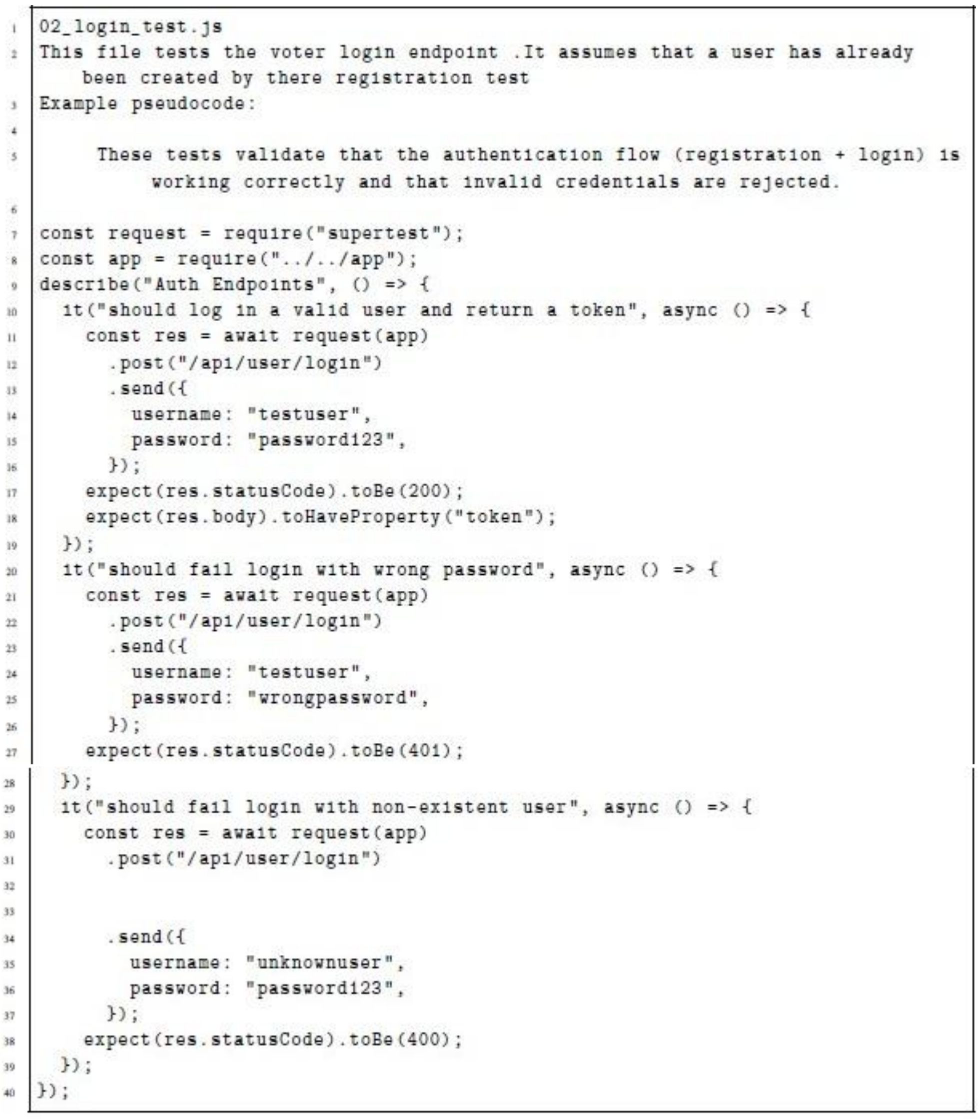
They are implemented using Jest + Supertest, invoking the real Express application (app.js) and mocking external dependencies such as the database connection and blockchain contract where needed.

5.4.1 Authentication Endpoints (01_register.test.js, 02_login.test.js)

01_register.test.js This file contains integration tests for the voter registration endpoint. Example test cases:

- **Test Case:** POST /api/user/register – should register a new voter with valid data
  - Expected Result: HTTP 201 Created, with a success message in the response.

- **Test Case:** POST /api/user/register – should fail to register a voter with duplicate credentials (e.g., same phone/username)
  - Expected Result: HTTP 400 Bad Request, with an error message indicating that the user already exists.
- **Test Case:** POST /api/user/register – should fail when required fields are missing.
  - Expected Result: HTTP 400 Bad Request.



```
02_login_test.js
This file tests the voter login endpoint .It assumes that a user has already
    been created by there registration test
Example pseudocode:

     These tests validate that the authentication flow (registration + login) is
          working correctly and that invalid credentials are rejected.

const request = require("supertest");
const app = require("../../app");
describe("Auth Endpoints", () => {
  it("should log in a valid user and return a token", async () => {
    const res = await request(app)
      .post("/api/user/login")
      .send({
        username: "testuser",
        password: "password123",
      });
    expect(res.statusCode).toBe(200);
    expect(res.body).toHaveProperty("token");
  });
  it("should fail login with wrong password", async () => {
    const res = await request(app)
      .post("/api/user/login")
      .send({
        username: "testuser",
        password: "wrongpassword",
      });
    expect(res.statusCode).toBe(401);
  });
  it("should fail login with non-existent user", async () => {
    const res = await request(app)
      .post("/api/user/login")


      .send({
        username: "unknownuser",
        password: "password123",
      });
    expect(res.statusCode).toBe(400);
  });
});
```

These tests validate that the authentication flow (registration + login) is working correctly and that invalid credentials are rejected.

5.4.2 Voting Endpoint (vote.test.js)

The voting endpoint is the core of iBallot and has been thoroughly tested using integration tests in vote.test.js. The blockchain layer and database queries are mocked so that different scenarios can be simulated without hitting the real network.

Representative test cases:

- **Test Case:** POST /vote – should successfully cast a vote with valid credentials and data.

  MOCKS:

  - Database returns a valid user and corresponding ECI record.

  - Password comparison succeeds.

  - Decryption of the stored private key succeeds.

  - Blockchain castVote call resolves successfully and returns a mock transaction hash.

  Expected Result:

  - HTTP 200 OK

  - response.body.success === true

  - response.body.message === "Vote cast successfully!"

  - response.body.txHash contains the mock transaction hash.

  - **Test Case 1:** POST /vote – should return 401 for an invalid password.
    - Mocks password comparison to fail.
    - Expected Result: HTTP 401 Unauthorized, with an error field

      such as " Invalid credentials.
  - **Test Case 2:** POST /vote – should return 404 if ECI data is not found for the user
    - Mocks the user query to succeed but the ECI query to return no records.
    - Expected Result: HTTP 404 Not Found, with error === "User

      data not found in ECI records.
  - **Test Case 3:** POST /vote – should return 500 if the blockchain transaction fails

- Mocks the blockchain interaction (e.g., castVoteMeta) to throw an error.
- Expected Result: HTTP 500 Internal Server Error, with error

  === "Failed to cast vote due to an internal server error.

These scenarios ensure that the vote-casting process correctly handles success, authentication failure, missing data, and blockchain-level failures.

5.4.3 Admin Endpoints (adminAuth.test.js, elections.test.js, results.test.js)

Admin functionality is protected and must only be accessible to authorized administrative users. Tests for these endpoints verify both security and correctness.

adminAuth.test.js (unit, but closely related to integration)

- **Test Case 1:** Should allow a request with a valid admin token to reach the protected route.
- **Test Case 2:** Should reject a request with an invalid or missing token with HTTP 401.

elections.test.js (integration) These tests validate the endpoints that allow the admin to manage elections. Typical test cases include:

- **Test Case 1:** POST /api/admin/elections – should allow an admin to create a new election
  - **Expected Result:** HTTP 201 Created, with the new election returned in the response.
- **Test Case 2:** DELETE /api/admin/elections/:id – should allow an admin to delete an existing election
  - **Expected Result:** HTTP 200 OK and a confirmation message.

results.test.js (integration) These tests check that election results can only be fetched by admins and that the data returned matches what was recorded through votes.

- **Test Case:** GET /api/admin/results/:id – should allow an admin to fetch election results
  - **Expected Result:** HTTP 200 OK, with aggregate vote counts per candidate.

5.5 Integration Test Case Summary

Table 5.1 summarizes the main integration test cases that were designed and executed for the iBallot backend.

Table 5.1: Integration Test Case Summary

Module	Method	Endpoint	Test Case Description	Expected Result	Status
Auth	POST	/api/user/register	Register with valid data	HTTP 201 Created	Pass
Auth	POST	/api/user/register	Register with duplicate user data	HTTP 400 Bad Request	Pass
Auth	POST	/api/user/login	Login with valid credentials	HTTP 200, token in response body	Pass
Auth	POST	/api/user/login	Login with invalid credentials	HTTP 401 Unauthorized	Pass
Vote	POST	/vote	Cast a vote with valid authentication and data	HTTP 200, success message + txHash	Pass
Vote	POST	/vote	Invalid password during vote	HTTP 401 Unauthorized	Pass
Vote	POST	/vote	Missing ECI data for user	HTTP 404 Not Found	Pass
Vote	POST	/vote	Blockchain failure when casting vote	HTTP 500 Internal Server Error	Pass
Admin	POST	/api/admin/elections	Create election as admin	HTTP 201 Created	Pass
Admin	GET	/api/admin/results/:id	Get results as admin	HTTP 200 OK	Pass

5.6 Test Results and Conclusion

The testing phase of iBallot was effective in identifying and resolving issues in both the backend and frontend layers, with particular focus on authentication, authorization, and vote-casting logic.

- **Unit Tests:** All unit tests for hashing utilities, encryption utilities, and admin middleware passed. This gives confidence that the core security primitives (password hashing, AES encryption, and token-based access control) function correctly.

- **Integration Tests:** Backend API integration tests, implemented with Jest and Supertest, cover the critical endpoints for registration, login, vote casting, and admin election management. During testing, several issues were discovered and fixed, including:
  - Inconsistent error handling in vote routes, now replaced with a standard 500 error message for unexpected internal failures.
  - Missing checks for user data in ECI records before casting a vote.
  - Handling of invalid credentials and non-existent users in the login flow.
- **Frontend Tests:** Vitest and React Testing Library were used to validate the key frontend flows for both admin and voter interfaces. Tests confirm correct rendering of pages, form behavior (e.g., enabling/disabling buttons), context updates, and navigation after login.

Overall, the combination of backend unit tests, backend integration tests, and frontend component tests provides high confidence in the reliability, security, and correctness of the iBallot system. The application is considered ready for controlled pilot deployment.

5.7 Summary

System testing of the iBallot system verified the correct functioning of backend services, blockchain interactions, and frontend interfaces. Unit and integration tests validated authentication, vote casting, and election management workflows. Identified issues were resolved, ensuring reliable, secure, and accurate system operation.

CHAPTER 6

RESULTS AND DISCUSSIONS

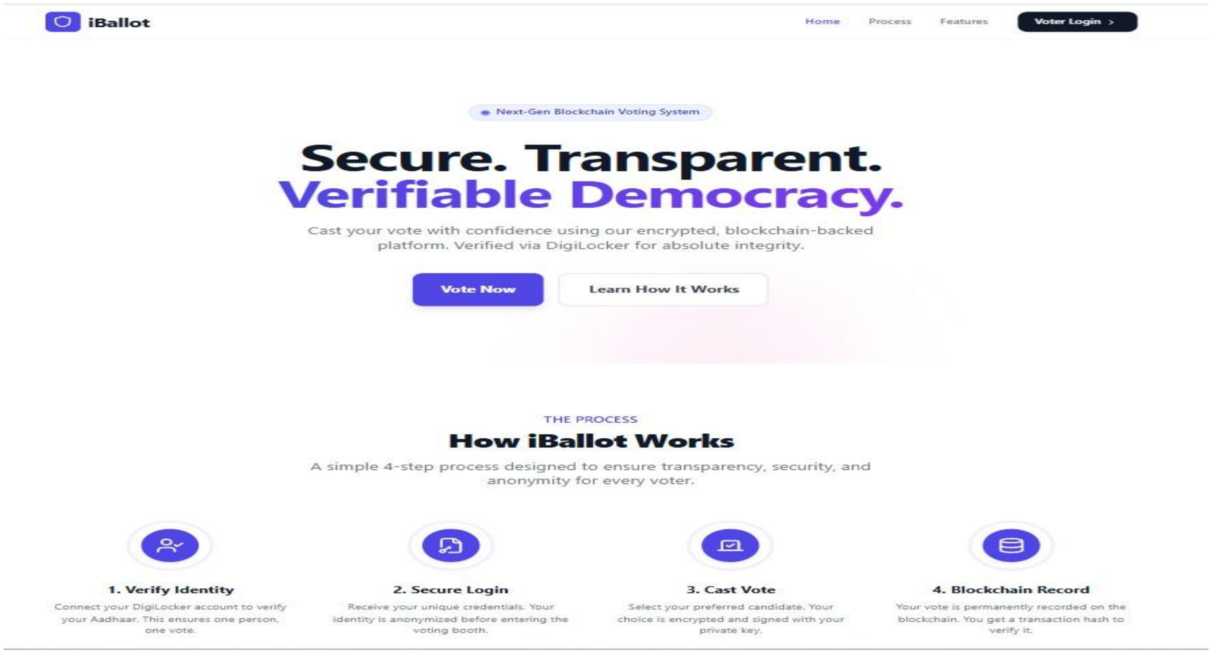
6.1 Experimental Results

The **iBallot** system was successfully designed and implemented as a secure **blockchain-based voting solution**. The functional workflow starting from **voter registration**, **authentication**, **blockchain authorization**, **election access**, **candidate selection**, and final vote casting was tested end-to-end using structured system testing methods.

The integration of the Admin Dashboard enables authorized officials to create elections, manage candidates, and activate/deactivate voting windows. Once activated, the election becomes visible to authenticated voters. After a vote is cast, the system marks the voter status on-chain to prevent duplicate or fraudulent voting, ensuring full transparency and integrity of the election process.

6.2 Snapshots of the e-voting system

The Snapshot 6.1 shows the front page of the iBallot system provides users with a secure and intuitive entry point to participate in blockchain-based voting. It highlights the platform's key features and guides voters towards authentication and the voting process



Secure. Transparent.
Verifiable Democracy.

Cast your vote with confidence using our encrypted, blockchain-backed platform. Verified via DigiLocker for absolute integrity.

How iBallot Works

A simple 4-step process designed to ensure transparency, security, and anonymity for every voter.

Connect your DigiLocker account to verify your Aadhaar. This ensures one person, one vote.

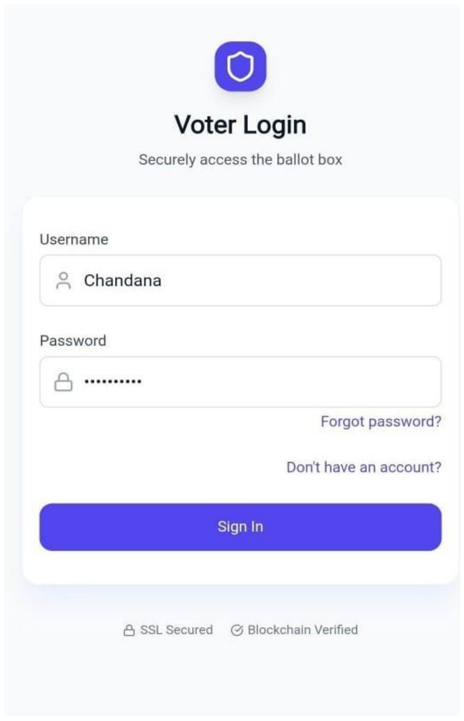
Receive your unique credentials. Your identity is anonymized before entering the voting booth.

Select your preferred candidate. Your choice is encrypted and signed with your private key.

Your vote is permanently recorded on the blockchain. You get a transaction hash to verify it.

Snapshot 6.1: iBallot User Landing Interface

The Snapshot 6.2 shows interface allows authenticated users to securely access the ballot system using their registered credentials. It ensures protected access through encryption and identity-verified.


Voter Login

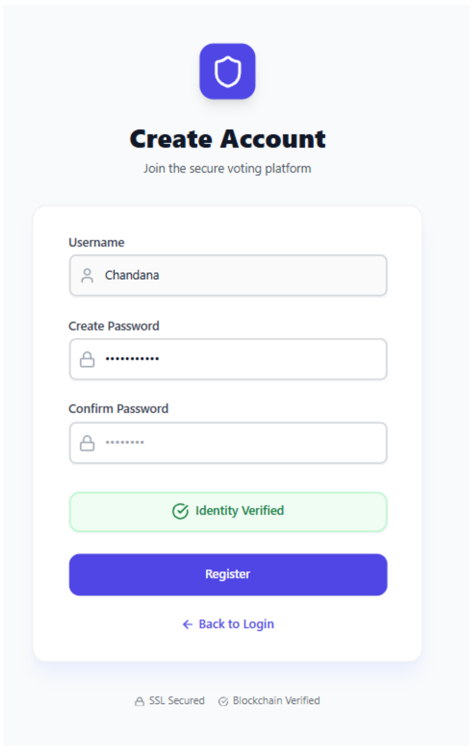
Securely access the ballot box

Username: Chandana

Password: ••••••••••

Snapshot 6.2: Voter Login Page

The Snapshot 6.3 shows the register page that enables new voters to create an account by setting up secure login credentials. Once registered, their identity is verified to ensure only legitimate voters can participate.


Create Account

Join the secure voting platform

Username: Chandana

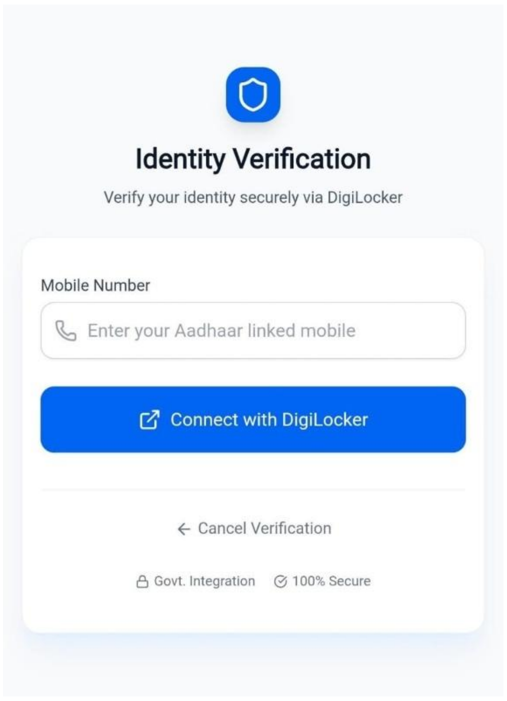
Create Password: ••••••••••

Confirm Password: ••••••••

Identity Verified

Snapshot 6.3: User Registration Page

The Snapshot 6.4 shows the interface that integrates with DigiLocker to authenticate the voter's Aadhaar-linked mobile identity. It guarantees government-verified and secure access before granting voting permissions.



Identity Verification

Verify your identity securely via DigiLocker

Mobile Number

Enter your Aadhaar linked mobile

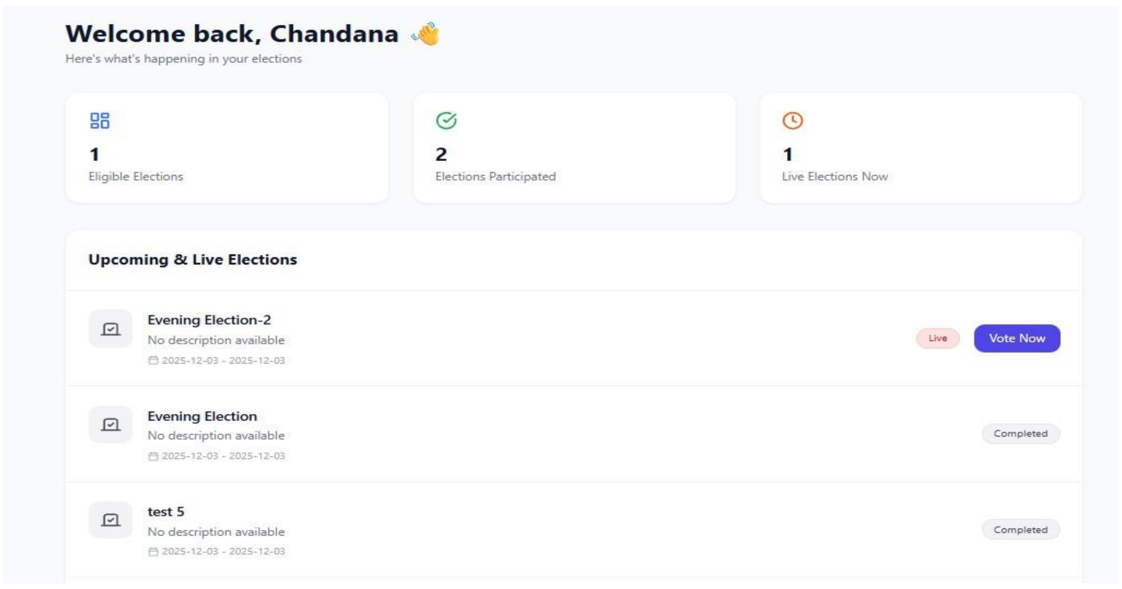
Connect with DigiLocker

Cancel Verification

Govt. Integration 100% Secure

Snapshot 6.4: DigiLocker Identity Verification Page

The Snapshot 6.5 shows the dashboard that provides voters with a personalized overview of their eligible, ongoing, and past elections. It enables users to easily track election status and participate in active polls securely through a unified interface. The voter dashboard displays the election as Live with a "Vote Now" button enabled, allowing the user to participate in the ongoing election.



Welcome back, Chandana 👋

Here's what's happening in your elections

1 Eligible Elections

2 Elections Participated

1 Live Elections Now

Upcoming & Live Elections

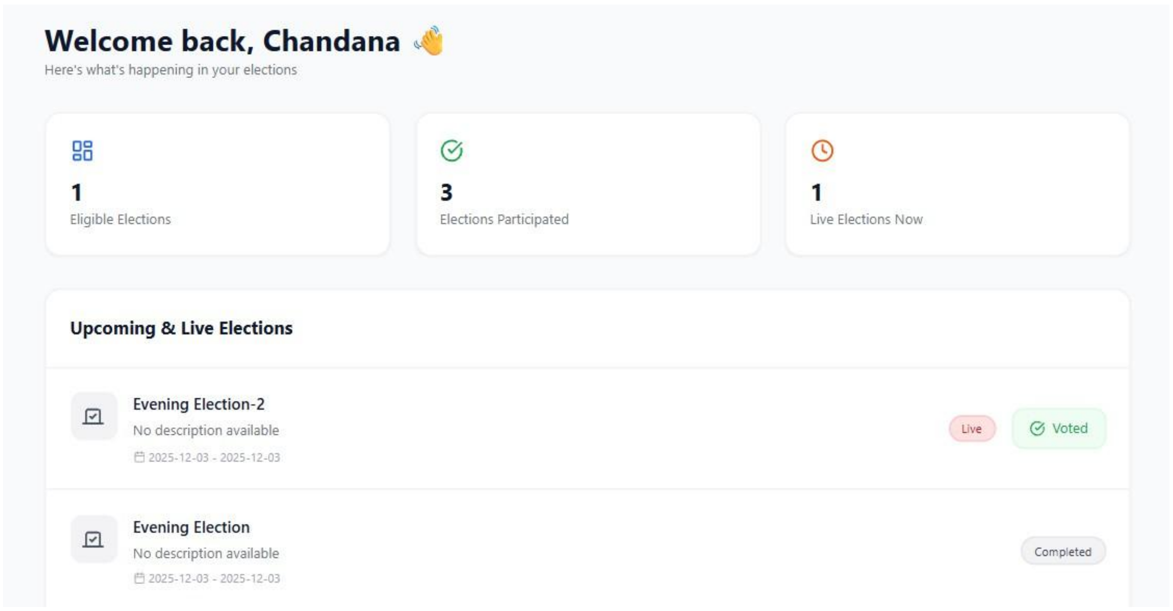
Evening Election-2
No description available
2025-12-03 - 2025-12-03
Live Vote Now

Evening Election
No description available
2025-12-03 - 2025-12-03
Completed

test 5
No description available
2025-12-03 - 2025-12-03
Completed

Snapshot 6.5: Voter Dashboard Interface

The Snapshot 6.6 shows the interface that after voting once the vote is successfully cast, the system disables further voting and marks the election status as “Voted”, ensuring one person, one vote while still showing the election as live if ongoing.


Welcome back, Chandana 👋

Here's what's happening in your elections

1 Eligible Elections

3 Elections Participated

1 Live Elections Now

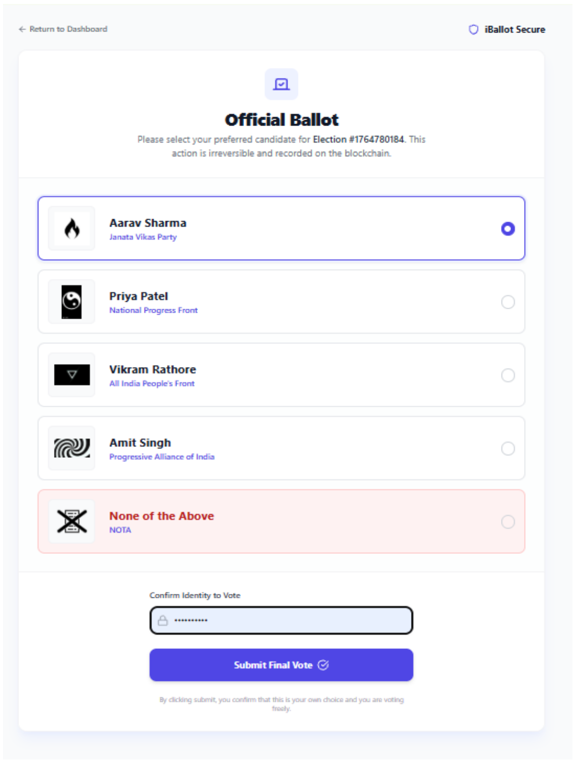
Upcoming & Live Elections

Evening Election-2
No description available
2025-12-03 - 2025-12-03
Live | Voted

Evening Election
No description available
2025-12-03 - 2025-12-03
Completed

Snapshot 6.6: After Voting

The Snapshot 6.7 shows the interface allows the voter to securely select their preferred candidate and confirm their identity before casting the final vote. Once submitted, the vote becomes irreversible and permanently recorded on the blockchain.


← Return to Dashboard | iBallot Secure

Official Ballot

Please select your preferred candidate for **Election #1764780184**. This action is irreversible and recorded on the blockchain.

- [✓ Aarav Sharma — Janata Vikas Party]
- Priya Patel — National Progress Front
- Vikram Rathore — All India People's Front
- Amit Singh — Progressive Alliance of India
- None of the Above — NOTA

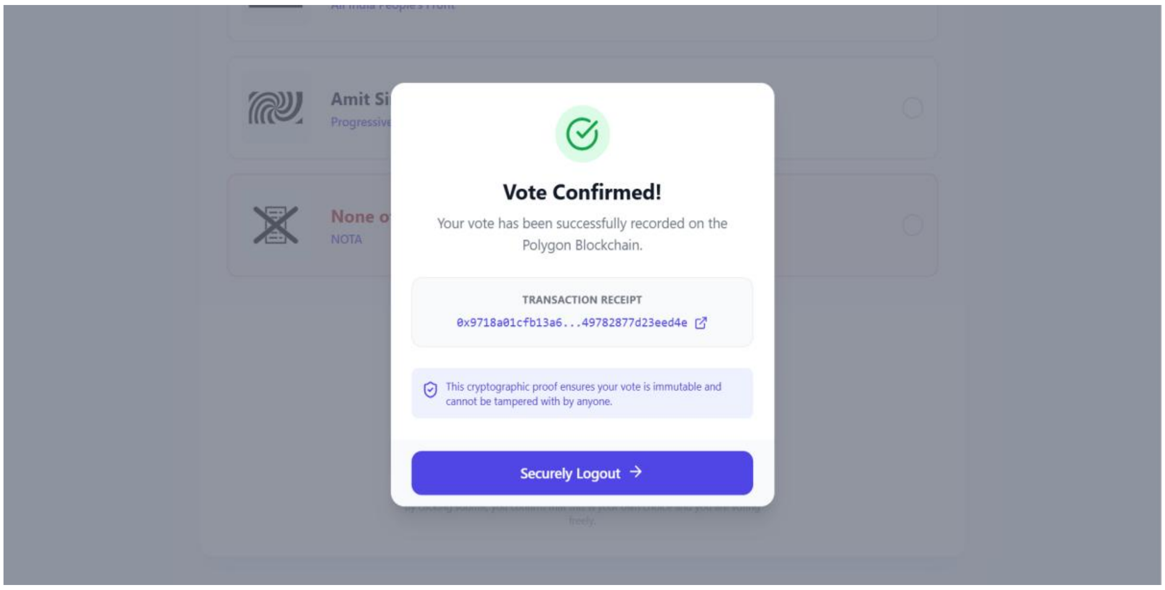
Confirm Identity to Vote: •••••••••••

Submit Final Vote

By clicking submit, you confirm that this is your own choice and you are voting freely.

Snapshot 6.7: Official Ballot Voting Screen

The Snapshot 6.8 shows the prompt after voting, the system displays a confirmation message with the blockchain transaction hash. This ensures transparency and verifiability, proving that the vote has been securely recorded in the ledger.


Vote Confirmed!

Your vote has been successfully recorded on the Polygon Blockchain.

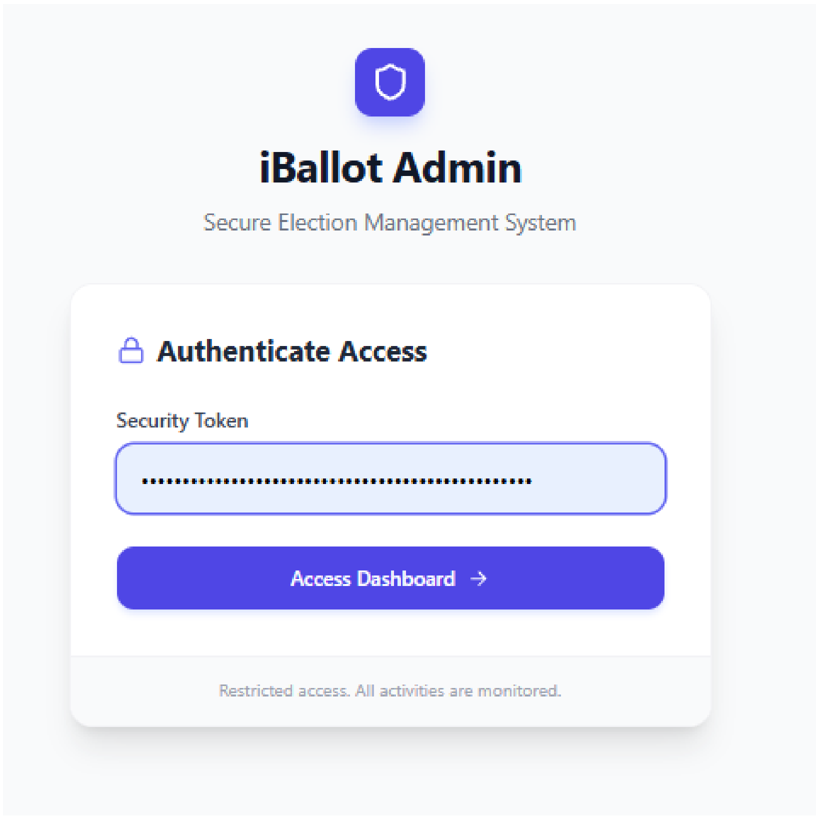
TRANSACTION RECEIPT

0x9718a01cfb13a6...49782877d23eed4e

This cryptographic proof ensures your vote is immutable and cannot be tampered with by anyone.

Snapshot 6.8: Successful Vote Confirmation

The Snapshot 6.9 shows the authentication page that ensures that only authorized election administrators can access the management dashboard using a secure token. It adds a strong security layer protecting sensitive election configurations.


iBallot Admin

Secure Election Management System

Authenticate Access

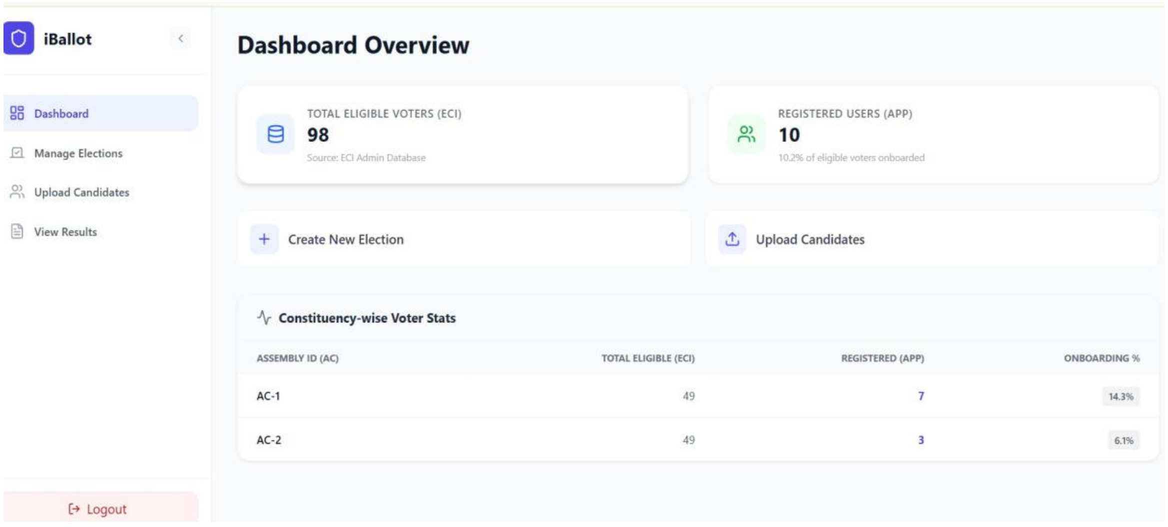
Security Token

••

Restricted access. All activities are monitored.

Snapshot 6.9: Admin Authentication Page

The Snapshot 6.10 shows the dashboard that provides key statistics such as total eligible voters and registered users, along with options to create elections and upload candidate data. It serves as the control center for monitoring voter onboarding and constituency-level stats.



Dashboard Overview

TOTAL ELIGIBLE VOTERS (ECI)
98
Source: ECI Admin Database

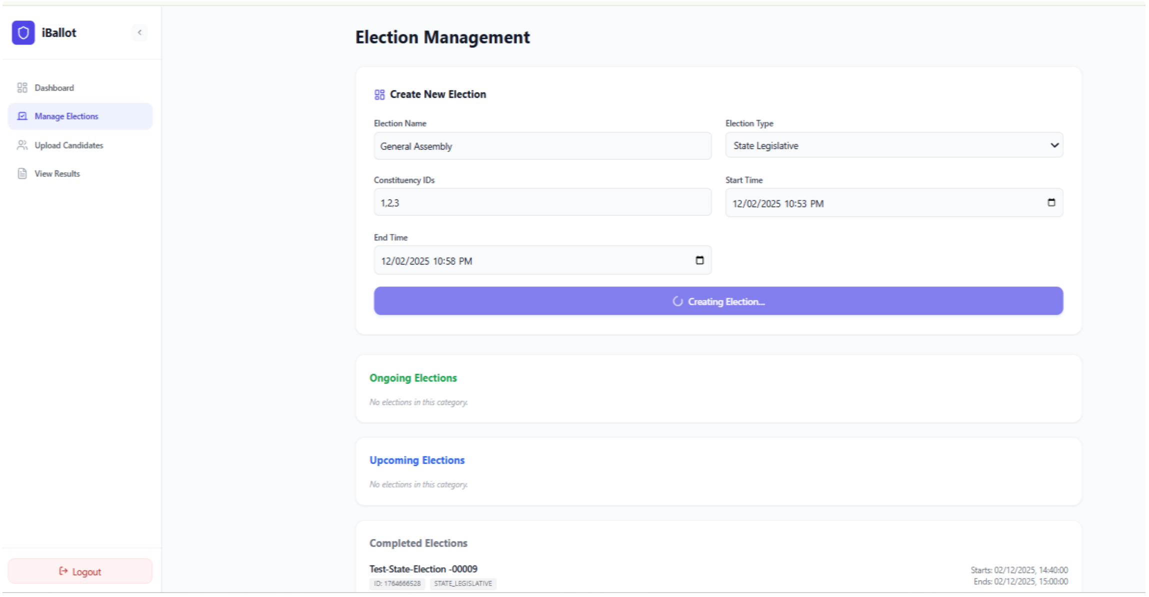
REGISTERED USERS (APP)
10
10.2% of eligible voters onboarded

Constituency-wise Voter Stats

ASSEMBLY ID (AC)	TOTAL ELIGIBLE (ECI)	REGISTERED (APP)	ONBOARDING %
AC-1	49	7	14.3%
AC-2	49	3	6.1%

Snapshot 6.10: Admin Dashboard Overview

The Snapshot 6.11 shows the interface that enables administrators to set up and schedule new elections by defining election type, timings, and constituency details. It also categorizes elections as ongoing, upcoming, or completed for simplified management.



Election Management

Create New Election

Election Name: General Assembly
Election Type: State Legislative
Constituency IDs: 1,2,3
Start Time: 12/02/2025 10:53 PM
End Time: 12/02/2025 10:58 PM

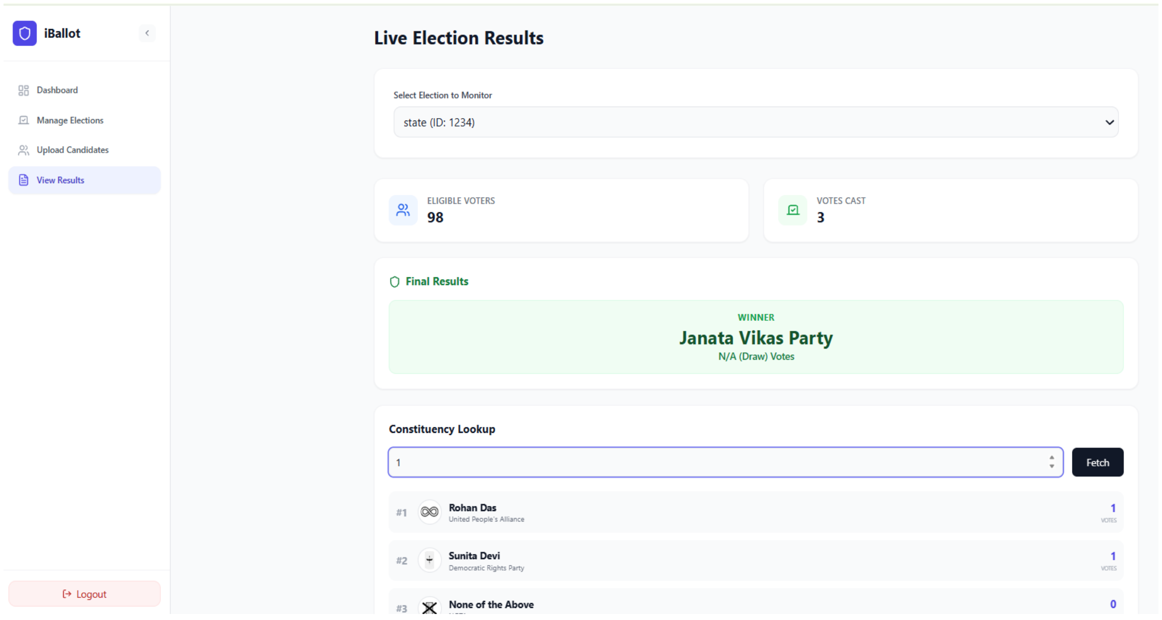
Ongoing Elections
No elections in this category.

Upcoming Elections
No elections in this category.

Completed Elections
Test-State-Election -00009
ID: 1764666528 STATE_LEGISLATIVE
Starts: 02/12/2025, 14:40:00
Ends: 02/12/2025, 15:00:00

Snapshot 6.11: Election Management Panel

The Snapshot 6.12 shows the interface that displays real-time election outcomes, including total eligible voters, votes cast, and the final winning party. It also allows constituency-wise results lookup for transparent and detailed monitoring.



Live Election Results

Select Election to Monitor: state (ID: 1234)

ELIGIBLE VOTERS 98

VOTES CAST 3

Final Results

WINNER
Janata Vikas Party
N/A (Draw) Votes

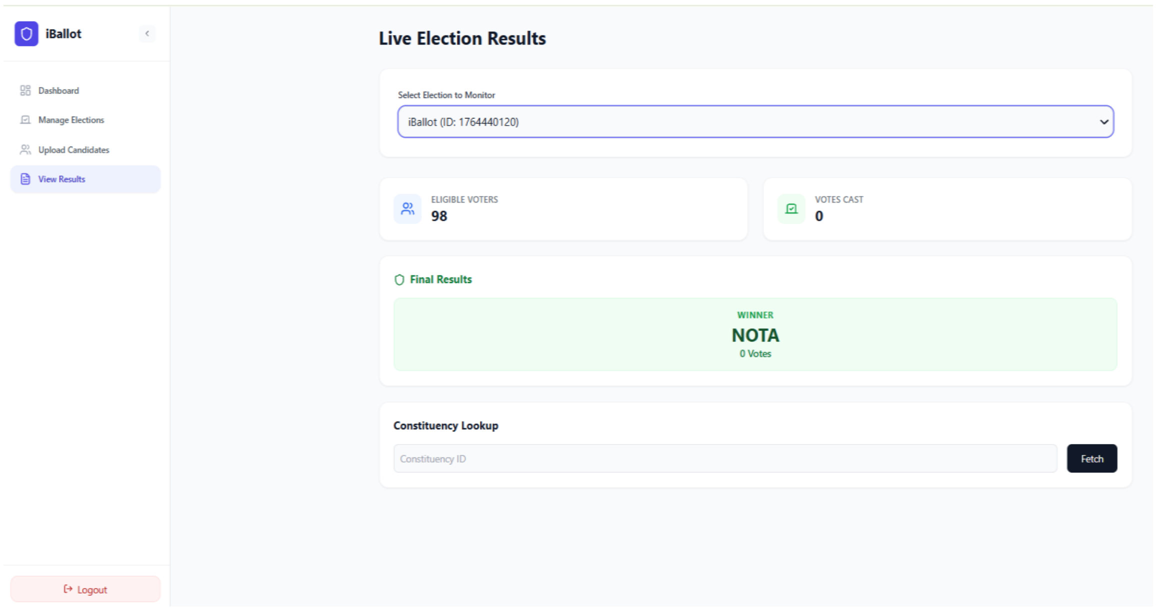
Constituency Lookup

1 — Fetch

#1	Rohan Das United People's Alliance	1 VOTES
#2	Sunita Devi Democratic Rights Party	1 VOTES
#3	None of the Above	0

Snapshot 6.12: Live Election Results Page

The Snapshot 6.13 represents If no votes are cast in an election, the system automatically marks NOTA (None of the Above) as the default winner. This ensures result transparency while clearly indicating that no voter has participated in the election.



Live Election Results

Select Election to Monitor: iBallot (ID: 1764440120)

ELIGIBLE VOTERS 98

VOTES CAST 0

Final Results

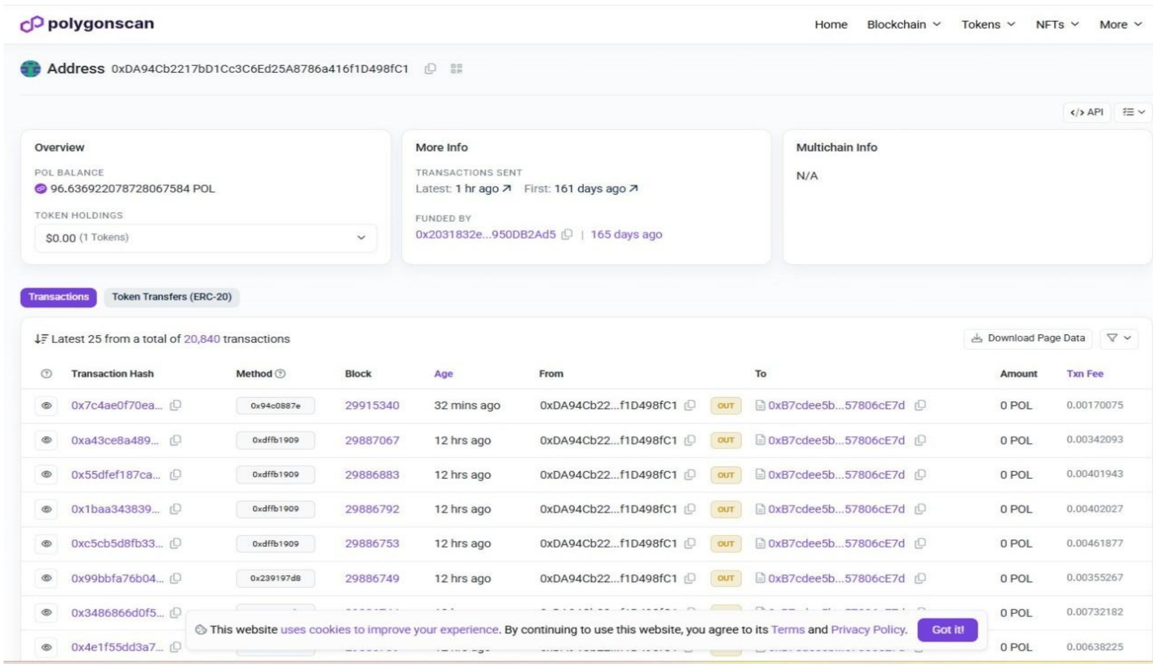
WINNER
NOTA
0 Votes

Constituency Lookup

Constituency ID — Fetch

Snapshot 6.13: NOTA(None of the Above)

The Snapshot 6.14 shows the screenshot displaying the on-chain vote transactions recorded on the Polygon blockchain, where each cast vote is assigned a unique transaction hash for verification. These records ensure transparency, immutability, and secure auditing of the entire voting process



Address 0xDA94Cb2217bD1Cc3C6Ed25A8786a416f1D498fC1

Overview
POL BALANCE
96.636922078728067584 POL
TOKEN HOLDINGS
$0.00 (1 Tokens)

More Info
TRANSACTIONS SENT
Latest: 1 hr ago ↗ First: 161 days ago ↗
FUNDED BY
0x2031832e...950DB2Ad5 | 165 days ago

Multichain Info
N/A

Latest 25 from a total of 20,840 transactions

Transaction Hash	Method	Block	Age	From		To	Amount	Txn Fee
0x7c4ae0f70ea...	0x94c0887e	29915340	32 mins ago	0xDA94Cb22...f1D498fC1	OUT	0xB7cdee5b...57806cE7d	0 POL	0.00170075
0xa43ce8a489...	0xdffb1909	29887067	12 hrs ago	0xDA94Cb22...f1D498fC1	OUT	0xB7cdee5b...57806cE7d	0 POL	0.00342093
0x55dfef187ca...	0xdffb1909	29886883	12 hrs ago	0xDA94Cb22...f1D498fC1	OUT	0xB7cdee5b...57806cE7d	0 POL	0.00401943
0x1baa343839...	0xdffb1909	29886792	12 hrs ago	0xDA94Cb22...f1D498fC1	OUT	0xB7cdee5b...57806cE7d	0 POL	0.00402027
0xc5cb5d8fb33...	0xdffb1909	29886753	12 hrs ago	0xDA94Cb22...f1D498fC1	OUT	0xB7cdee5b...57806cE7d	0 POL	0.00461877
0x99bbfa76b04...	0x239197d8	29886749	12 hrs ago	0xDA94Cb22...f1D498fC1	OUT	0xB7cdee5b...57806cE7d	0 POL	0.00355267
0x3486866d0f5...	[illegible]	[illegible]	[illegible]	[illegible]		[illegible]	0 POL	0.00732182
0x4e1f55dd3a7...	[illegible]	[illegible]	[illegible]	[illegible]		[illegible]	0 POL	0.00638225

This website uses cookies to improve your experience. By continuing to use this website, you agree to its Terms and Privacy Policy.

Snapshot 6.14: Blockchain Vote Transaction Records on PolygonScan

6.3 Comparision Of Exisiting System Vs Proposed System

Table 6.1:Electronic Voting machine vs iBallot

Parameter	**Existing Systems (EVM)**	**Proposed System (iBallot)**
Architecture	Standalone Hardware: Isolated chips; physical access required for counting.	Decentralized: Distributed ledger; data exists across the entire network.
Data Integrity	Hardware Dependent: Vulnerable to chip replacement or physical tampering.	Immutable: Cryptographically sealed; mathematically impossible to alter history.
Transparency	"Black Box": Counting logic is hidden inside the machine's circuit.	Open Source: Smart contract logic is public and verifiable by anyone.
Audit Trail	Paper VVPAT: Physical slips can be lost, damaged, or miscounted.	Digital: Instant, permanent transaction hash (0x…) for every vote.
Cost	High: Expensive manufacturing, storage, transport, and security.	Low: Software-based; uses "Gasless" relayer for minimal fees.
Accessibility	Restricted: Voters must travel to specific physical booths.	Universal: Remote voting from anywhere via secure web portal.

Table 6.2: Apps vs iBallot

Parameter	Existing Systems (Apps)	Proposed System (iBallot)
Trust Architecture	Centralized Server: Relies on a central authority (server admin) to host and count votes honestly.	Decentralized Ledger: Relies on Ethereum smart contracts; no single entity controls the counting logic.
Voter Identity	Admin-Managed: Uses email lists or simple passwords created by the election admin (prone to fake accounts).	Government-Verified: Integrates DigiLocker API to strictly authenticate legitimate Indian citizens via Aadhaar.
Data Storage	Private Database: Votes are stored on a cloud database which can be modified or deleted by a super-admin.	Immutable Blockchain: Votes are written to a public block; once mined, they are mathematically impossible to delete or alter.
Security Model	Encryption-Based: Uses cryptography to hide votes, but the server is a single point of failure (DDoS risk).	Distribution-Based: Data exists on thousands of nodes simultaneously, making the system resilient to crashes or hacks.
Auditability	Complex Proofs: Verification requires understanding complex zero-knowledge proofs or mix-nets.	Instant Hash: Any user can copy their unique transaction hash (0x…) and verify their vote on a public explorer immediately.

6.3 Summary

The results demonstrate the successful implementation of the iBallot system, validating its secure and transparent blockchain-based voting workflow. Experimental outcomes confirm the correct functioning of key modules, including voter registration, authentication, vote casting, and real-time result generation. The admin dashboard effectively supports election creation, candidate management, and result monitoring. These visual results confirm usability, functional accuracy, and alignment with the proposed system design, demonstrating that iBallot meets its intended objectives.

CHAPTER 7

CONCLUSION AND FUTURE ENHANCEMENTS

7.1 Conclusion

The **iBallot** system demonstrates a secure, transparent, and practical **blockchain-based digital voting platform** designed with real-world usability in mind. Unlike earlier systems that rely on **conventional Ethereum test networks** or **private environments**, **iBallot** is deployed on the Polygon Amoy testnet, offering better scalability and a production-aligned development environment.

Each vote in iBallot is permanently recorded on the blockchain in hashed form, ensuring immutability, tamper resistance, and a verifiable audit trail. Identity verification is handled through a **mock DigiLocker dataset**, where sensitive user information is abstracted and a hashed UID is used as the on-chain voter identifier, balancing privacy with verifiability.

The system adopts a pure web-based model that hides blockchain complexity from users. Voters are not required to manage wallets or pay transaction fees, as **gasless meta-transactions** are executed through an **admin-controlled relayer wallet**. Security is further strengthened by preventing double voting using both on-chain cryptographic checks and off-chain database validations.

Overall, **iBallot** highlights how a **hybrid blockchain architecture** combined with user-centric design can deliver a **scalable, secure, and accessible digital voting solution** suitable for future electoral systems.

7.2 Future Enhancement

While the current version of **iBallot** demonstrates the feasibility of a hybrid blockchain-based voting system, several enhancements are planned to move from an academic prototype to a production-ready electoral platform.

7.2.1 Infrastructure and Network Upgrade

Migration from Testnet to Mainnet: The system is currently deployed on the Polygon Amoy testnet for safe experimentation and development. A key roadmap item is migrating the smart contracts to a production-grade **mainnet** (such as Polygon PoS Mainnet) to benefit from stronger security guarantees, higher decentralization, and long-term persistence of election data.

7.2.2 Identity and Authentication Integration

- **Official DigiLocker API Integration:** The current implementation uses a mock DigiLocker database for identity verification. A planned enhancement is to integrate with the official DigiLocker APIs, allowing direct verification of voter records against government-maintained identity repositories. This would enable real-time eligibility checks, dynamic retrieval of constituency information, and removal of manually seeded mock data.

7.2.3 Advanced Privacy and Cryptography (Research Direction)

- **Zero-Knowledge Proofs (ZKP):** At present, the system uses hashed identifiers and a trusted backend relayer model, which still allows administrators to theoretically correlate a voter with their vote. As a research direction, integrating zero-knowledge proof systems (e.g., ZK-SNARK–based protocols) would allow voters to prove eligibility and one-time participation without revealing their actual identity or linking it to a specific ballot. This would move the platform closer to information-theoretic anonymity while preserving auditability.
- Enhance wallet and transaction transparency within the iBallot UI by showing transaction status, confirmations, and blockchain explorer links directly in the application, while retaining a relayer-based gasless experience.

7.2.4 Regulatory and Legal Integration

- Compliance with Election Commission Norms For real-world adoption, iBallot must align with the legal and procedural frameworks defined by the Election Commission and related authorities. This includes supporting officially prescribed workflows for voter roll updates, polling schedules, and contesting results.

7.3 Summary

iBallot establishes a secure, transparent voting platform on the Polygon Amoy testnet, using gasless meta-transactions to abstract blockchain complexity from voters. The system ensures data immutability and prevents double voting through a hybrid architecture that combines hashed on-chain records with mock DigiLocker identity verification. Future enhancements target a migration to the Polygon Mainnet for greater security and the integration of official DigiLocker APIs to replace mock data with real-time government verification.

REFERENCES

[1] R. Nandimath and S. Mandape, “Voting and Election System: using Blockchain Technology,” *International Journal of Engineering Research and Technology (IJERT)*, vol. 12, no. 4, Apr. 2023. [Online].
Available: https://www.ijert.org/research/voting-and-election-system-using-blockchain-technology-IJERTV12IS040084.pdf

[2] T. M. Navamani, T. S. Sondhi, and S. Ghildiyal, “DigiVoter: Blockchain Secured Digital Voting Platform with Aadhaar ID Verification,” *ECS Transactions*, vol. 107, no. 1, pp. 7427–7440, Apr. 2022.

[3] Md. Ibrahim, “Design and Development of a Decentralized Voting System Using Blockchain,” *International Journal of Communication and Information Technology (IJCIT)*, vol. 4, no. 1, pp. 1–11, Jan. 2023.

[4] S. B. Thapa, K. Vijendran, and V. K. V., “DigiVote: Voting System Using Blockchain,” Project Report, Dept. of CSE, AMC Engineering College, VTU, July 2021.

[5] OpenAI, “GPT-4 Technical Report,” *arXiv preprint arXiv:2303.08774*, 2023. [Online]. Available: https://arxiv.org/abs/2303.08774 (Used for code optimization and security analysis).

[6] Google DeepMind, “Gemini: A Family of Highly Capable Multimodal Models,” *arXiv preprint arXiv:2312.11805*, 2023. (Used for documentation assistance and architectural planning).

[7] V. Buterin, “A Next-Generation Smart Contract and Decentralized Application Platform,” *Ethereum White Paper*, 2014. [Online].
Available: https://ethereum.org/en/whitepaper/

[8] Polygon Labs, “Polygon PoS: Architecture and Security,” *Polygon Documentation*, 2024. [Online].
Available: https://wiki.polygon.technology/

[9] Github Link .

Available : https://www.github.com/sidd1224/iBallot.git