Tsybus Nikita
SE-2421

PART 1
1.

Web3.js and Ethers.js are two widely used JavaScript libraries for interacting with the Ethereum blockchain, but they differ significantly in design philosophy and internal architecture. Web3.js was created early in Ethereum's development and closely mirrors the structure of Ethereum clients and the JSON-RPC interface. Ethers.js, on the other hand, was designed later with a focus on security, modularity, and developer experience. These differences influence how applications are built, maintained, and secured.
One of the main architectural differences lies in how the libraries are structured. Web3.js follows a largely monolithic design, where most features are bundled into a single package. This approach increases bundle size and makes selective imports difficult. Ethers.js, especially version 6, adopts a modular architecture, separating providers, wallets, utilities, and contract logic into distinct components. This modularity allows developers to include only the necessary functionality, improving performance and reducing complexity in frontend applications.
Example: Sending a transaction in Web3.js:

```
const Web3 = require("web3");
const web3 = new Web3("http://localhost:8545");

const accounts = await web3.eth.getAccounts();

await web3.eth.sendTransaction({
  from: accounts[0],
  to: accounts[1],
  value: web3.utils.toWei("1", "ether")
});
```

In Web3.js, the transaction sender is inferred from the 'from' field, and signing is typically handled implicitly by the connected provider. While this approach simplifies usage, it hides critical details about how and where the private key is stored. Error messages returned from failed transactions are often generic, making debugging more difficult.

Example: Sending a transaction in Ethers.js v6:

```
import { JsonRpcProvider, Wallet, parseEther } from "ethers";

const provider = new JsonRpcProvider("http://localhost:8545");
const wallet = new Wallet(PRIVATE_KEY, provider);

const tx = await wallet.sendTransaction({
  to: RECEIVER_ADDRESS,
  value: parseEther("1")
});

await tx.wait();
```

Ethers.js separates concerns by explicitly defining the wallet and provider. The developer must consciously manage private keys and signing logic, which improves security and transparency. The returned transaction object contains structured metadata, and errors include detailed context such as revert reasons and gas usage.

Provider abstraction is another important distinction. In Web3.js, providers are often injected globally, such as through window.ethereum, which can lead to tight coupling between the application and the execution environment. Ethers.js abstracts providers more cleanly by offering explicit provider classes, such as JsonRpcProvider and BrowserProvider. This makes it easier to switch between local networks, testnets, and mainnet without changing application logic, improving portability and maintainability.

Handling of signing and private keys also differs significantly. Web3.js typically delegates signing to the provider, which may hide important security details from developers. Ethers.js makes signing an explicit operation by separating wallets from providers. Developers must consciously create a wallet instance using a private key or external signer. This explicit design reduces the risk of accidental private key exposure and improves auditability of signing logic.

ABI parsing and contract interaction are more strictly handled in Ethers.js. Web3.js performs ABI decoding dynamically and allows loosely typed method calls, which can result in runtime errors that are difficult to trace. Ethers.js enforces stricter ABI validation and normalizes data types, causing errors to surface earlier. This leads to safer contract interactions and clearer feedback during development.

Gas estimation and error propagation also reflect the philosophical differences between the libraries. Web3.js often returns generic error messages, such as revert or invalid opcode, which provide limited debugging information. Ethers.js propagates structured errors that include revert reasons, error codes, and transaction context. This makes diagnosing failed transactions significantly easier and improves developer productivity.

From a debugging perspective, Ethers.js offers a better overall user experience. Its consistent handling of BigInt values, readable transaction objects, and detailed error messages simplify tracing issues across the transaction lifecycle. Web3.js, while functional, often requires additional tooling or manual decoding to achieve the same level of insight.

Security considerations further explain why modern dApps increasingly migrate from Web3.js to Ethers.js. By avoiding global mutable state and enforcing explicit wallet usage, Ethers.js reduces common security mistakes. Its modular design also limits the attack surface of applications. As Ethereum development standards evolve, Ethers.js better aligns with modern frontend frameworks and security best practices, making it the preferred choice for new decentralized applications.

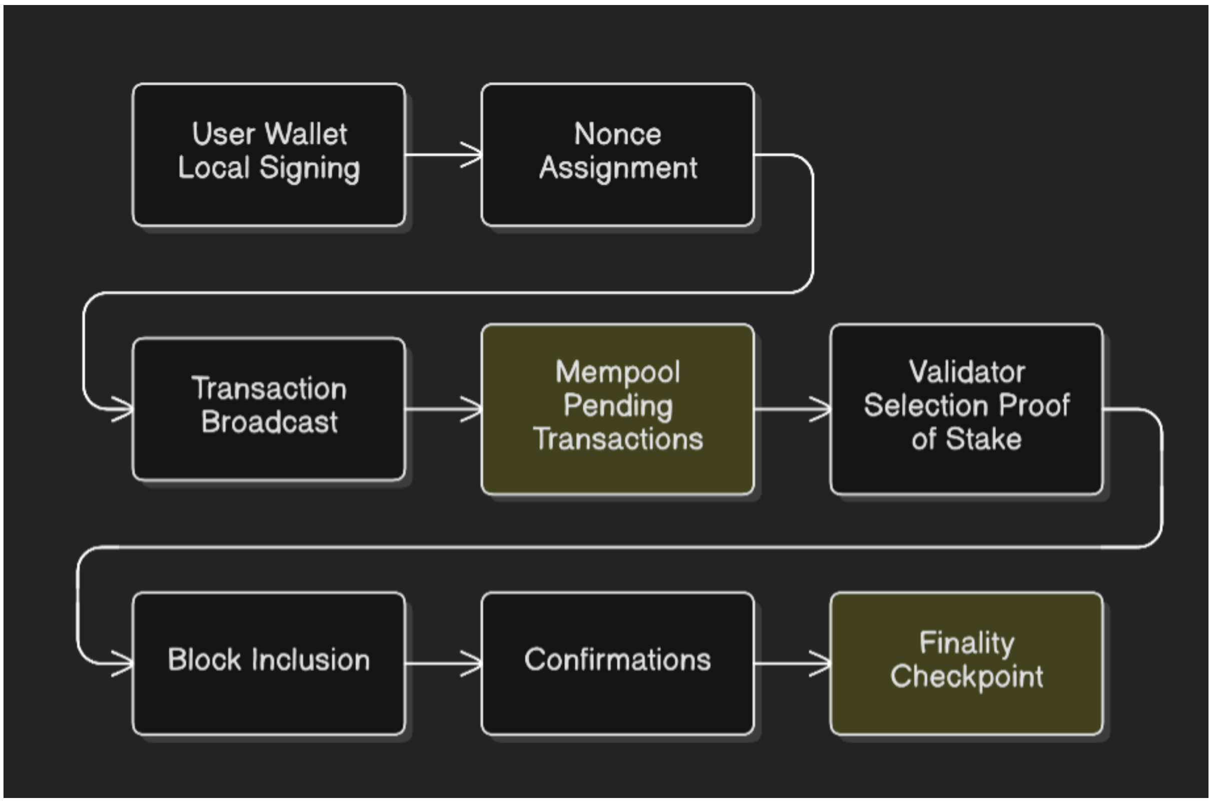
2.

Ethereum clients expose blockchain functionality through a JSON-RPC interface.

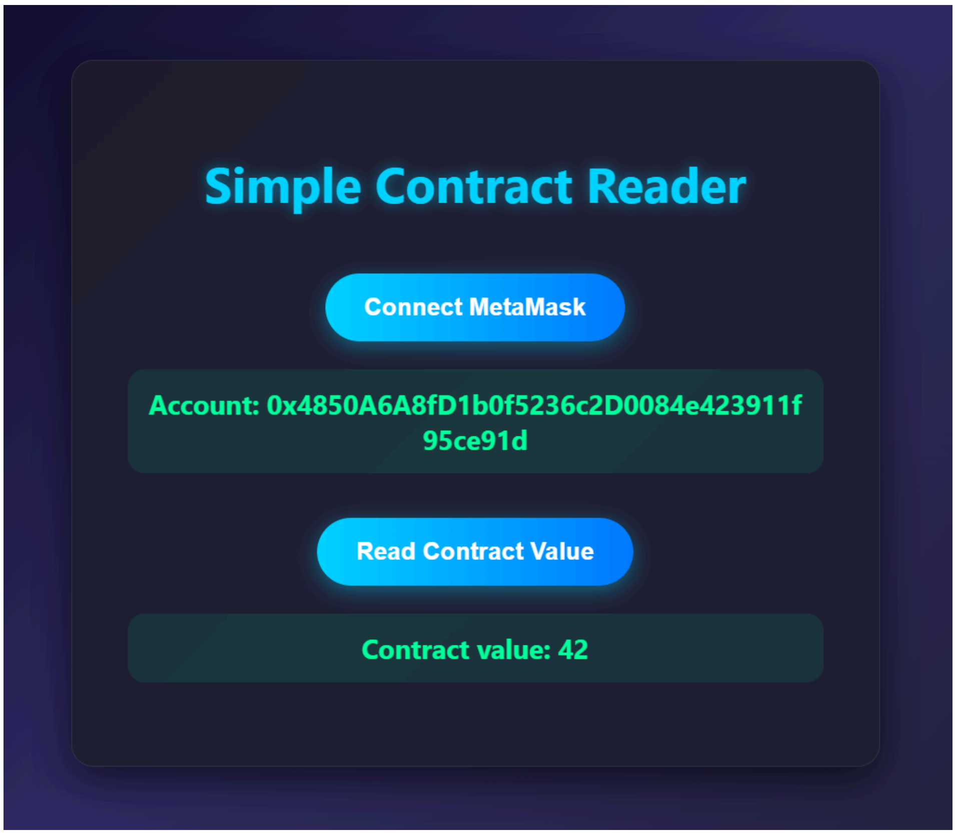
JSON-RPC is a stateless protocol that allows applications to request blockchain data or submit transactions using standardized method calls. Each request includes a method

name, parameters, and an identifier, enabling communication between dApps and Ethereum nodes.
Different network environments significantly affect how applications behave. Local networks such as Ganache provide instant block confirmation and zero economic risk, making them ideal for development. Testnets simulate real network conditions using valueless ETH, allowing developers to test deployment and interaction logic. Mainnet interactions involve real assets and irreversible transactions, requiring strict security and gas optimization.
Ethereum nodes can operate in several modes, including full nodes, archive nodes, and light nodes. Full nodes store recent blockchain state and validate transactions, while archive nodes maintain the entire historical state, enabling advanced queries at the cost of high storage requirements. Light nodes rely on full nodes for data, trading independence for efficiency. Most frontend applications do not run nodes directly.
Because running and maintaining nodes is expensive, most dApps rely on RPC providers such as Infura, Alchemy, or QuickNode. These services act as RPC proxies, abstracting infrastructure complexity and providing scalable access to Ethereum networks. This model allows developers to focus on application logic rather than node maintenance.
Rate limiting imposed by RPC providers strongly influences frontend design. Applications must minimize unnecessary requests, batch calls where possible, and cache frequently accessed data. Poor RPC usage can lead to degraded user experience or service denial, even if smart contracts are correctly implemented.

3.
The transaction lifecycle begins with local signing. A transaction is constructed on the client side and signed using the sender’s private key. This cryptographic signature proves ownership of the account and ensures transaction integrity before it is broadcast to the network.
Each transaction includes a nonce, which represents the number of transactions previously sent from the account. Nonce management ensures transaction ordering and prevents replay attacks. Incorrect nonce values result in failed or stuck transactions.
After signing, the transaction enters the mempool, where it is propagated across the peer-to-peer network. Nodes temporarily store pending transactions and prioritize them based on gas fees and network conditions.
Validators select transactions from the mempool and include them in new blocks. Since Ethereum’s transition to Proof of Stake, validators replace miners, but transaction selection still depends heavily on fee incentives and block capacity.
Even after inclusion in a block, transactions are subject to reorganization risk. Short chain reorganizations may remove recently included transactions. Applications typically wait for multiple confirmations to reduce this risk.
Finality differs between Proof of Work and Proof of Stake. Proof of Stake introduces checkpoint-based finality, providing stronger guarantees that finalized blocks cannot be reverted. This improves reliability for applications handling high-value transactions.
Gas fees are governed by EIP-1559, which introduced a base fee that is burned and a priority fee paid to validators. This mechanism stabilizes gas prices and improves predictability for users, while maintaining incentives for block producers.


User Wallet Local Signing
Nonce Assignment
Transaction Broadcast
Mempool Pending Transactions
Validator Selection Proof of Stake
Block Inclusion
Confirmations
Finality Checkpoint

PART 2
Task A



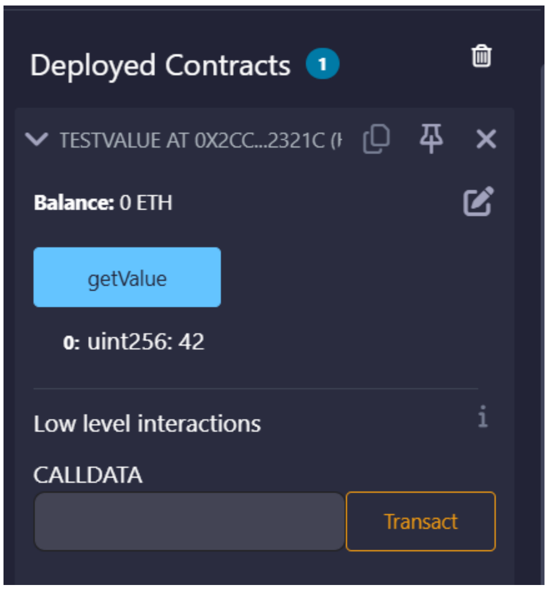
Simple Contract Reader

Account: 0x4850A6A8fD1b0f5236c2D0084e423911f95ce91d

Contract value: 42

contract on remix

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract TestValue {
    uint256 private value = 42;

    function getValue() public view returns (uint256) {
        return value;
    }
}
```



Deployed Contracts 1

TESTVALUE AT 0X2CC...2321C (I

Balance: 0 ETH

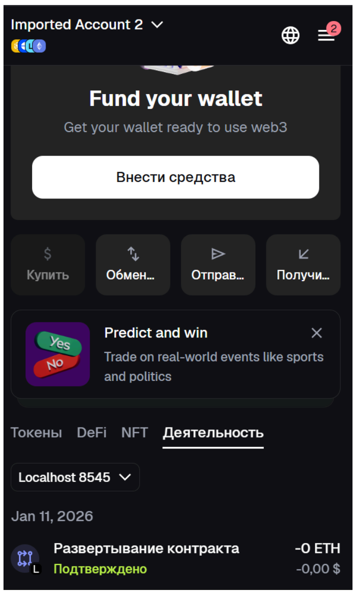
getValue

0: uint256: 42

Low level interactions

CALLDATA

Transact



Imported Account 2

Fund your wallet

Get your wallet ready to use web3

Внести средства

Купить Обмен... Отправ... Получи...

Predict and win

Trade on real-world events like sports and politics

Токены DeFi NFT Деятельность

Localhost 8545

Jan 11, 2026

Развертывание контракта -0 ETH

Подтверждено -0,00 $

Files will be in the zip file

Task B

Security Risks and Mitigations in Decentralized Applications (dApps)

Introduction

Decentralized applications (dApps) allow users to interact directly with blockchain smart contracts through web interfaces. This direct interaction between browser and wallet introduces unique security challenges. Unlike traditional web applications, dApps deal with real assets (cryptocurrencies, NFTs, tokens), making them attractive targets for attackers. This section discusses major risks and how modern wallet providers mitigate them.

1. Why direct RPC exposure is dangerous

Directly exposing an RPC endpoint (e.g., Infura or Alchemy key) in the frontend code allows anyone visiting the site to send requests through your node key. This can lead to:

Rate-limiting abuse or DDoS on your RPC provider
Increased costs (many providers charge per request)
Potential man-in-the-middle attacks if the key is leaked

Mitigation: Modern dApps use the injected provider (window.ethereum) provided by wallets like MetaMask. The user's own wallet handles all RPC communication. The frontend only requests signatures or read operations through the secure bridge — no private RPC keys are exposed in the client code.
2. XSS risks in dApps
Cross-Site Scripting (XSS) allows attackers to inject malicious JavaScript into the dApp page. Once injected, the script can:
Intercept window.ethereum calls
Steal signed messages/transactions
Redirect users to phishing sites
Modify the UI to trick users into approving malicious transactions

Mitigation:
Use strong Content Security Policy (CSP) headers
Sanitize all user inputs (never use innerHTML with untrusted data)
Frameworks like React/Vue automatically escape content by default
MetaMask requires user confirmation in a separate secure popup, making blind signing almost impossible

3. Signature replay risks
A signature replay attack occurs when a valid signature created for one purpose is reused for another (e.g., signing a message on a fake site, then replaying it on the real site).
Classic example: user signs personal_sign message → attacker uses the same signature to drain funds elsewhere.
Mitigation
Use EIP-712 structured data signing with domain separator (includes chainId, verifyingContract, nonce)
Modern wallets (MetaMask, Rabby) enforce domain separation — signatures are bound to specific contract + chain
Add nonces or timestamps in messages
Never sign raw strings without domain

4. Private key extraction attacks using malicious Web3 injection
Attackers can replace window.ethereum with a malicious proxy that intercepts all calls, logs signatures, or even tries to extract private keys (though MetaMask never exposes the private key directly).
Common vectors: fake MetaMask extensions, XSS injection, malicious browser plugins.
Mitigation by modern wallets:

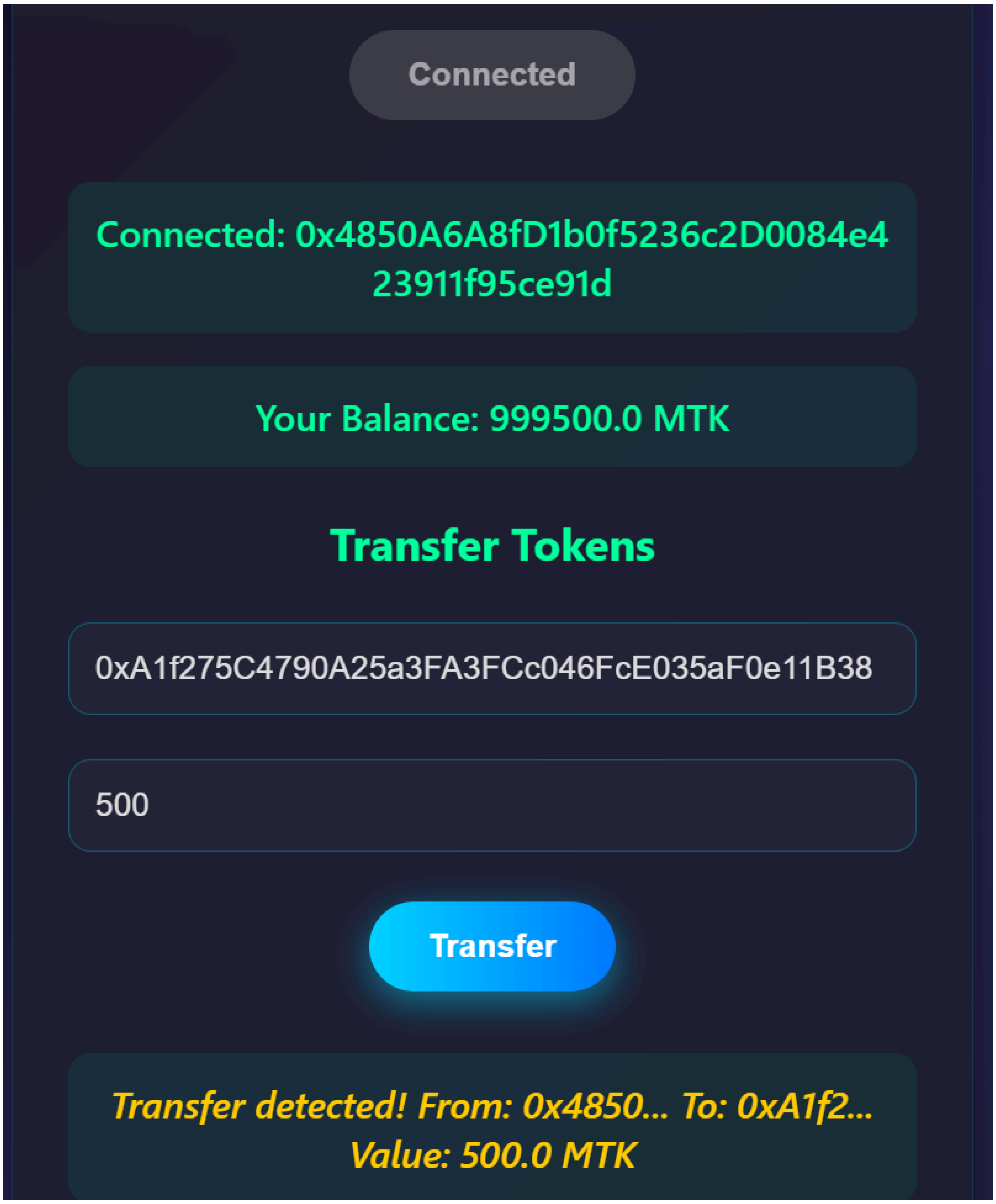
MetaMask runs in a secure isolated environment (separate process/extension sandbox)
All sensitive actions (signing, sending) require user confirmation in a popup
window.ethereum.isMetaMask flag helps detect legitimate provider
Users are strongly advised to install only official extensions from trusted sources

WalletConnect and hardware wallets add extra isolation layers

Conclusion

While dApps introduce powerful new interaction models, they also create novel attack surfaces. Direct RPC exposure, XSS, signature replay, and malicious provider injection are among the most dangerous threats. Modern wallet providers like MetaMask, Rabby, and WalletConnect significantly reduce these risks through secure popups, EIP-712 domain separation, permission models, and sandboxing. However, developers must follow best practices: use injected providers only, sanitize inputs, apply CSP, and educate users about phishing. Security in Web3 is a shared responsibility between wallet providers, dApp developers, and users.

PART 3



Connected: 0x4850A6A8fD1b0f5236c2D0084e423911f95ce91d

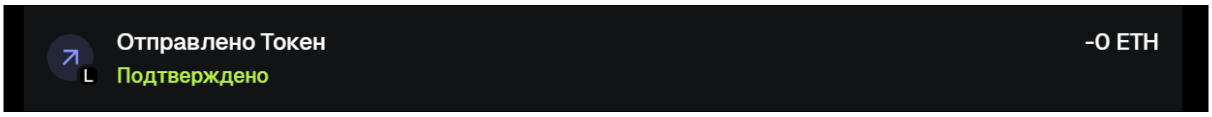
Your Balance: 999500.0 MTK

Transfer Tokens

0xA1f275C4790A25a3FA3FCc046FcE035aF0e11B38

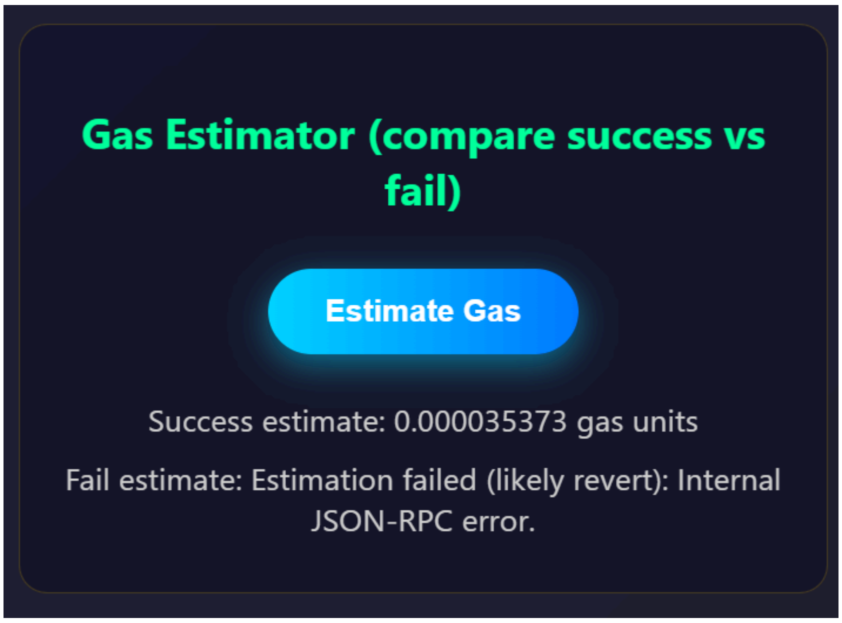
500

Transfer detected! From: 0x4850... To: 0xA1f2... Value: 500.0 MTK


Отправлено Токен

Подтверждено

-0 ETH


Gas Estimator (compare success vs fail)

Success estimate: 0.000035373 gas units

Fail estimate: Estimation failed (likely revert): Internal JSON-RPC error.

The primary goal of this dApp is to provide users with an intuitive, secure, and educational interface for interacting with the MyToken ERC-20 contract. The design balances simplicity for beginners with useful advanced features for learning.

1. Intuitive and Minimalistic Layout

The interface uses a single-column layout with clear sections: wallet connection, balance display, transfer form, and gas estimator. A dark gradient background with neon accents creates a modern Web3 look. Large buttons, high contrast, and readable fonts ensure good accessibility on desktop and mobile.

2. Progressive Disclosure & Safety

- "Connect Wallet" is the only active button at start - prevents invalid actions.
- After connection, the button disables and changes text to "Connected" - clear feedback.
- Transfer form validates inputs (address format, positive amount) and shows alerts for errors.
- All operations are wrapped in try/catch with user-friendly messages ("Processing...", "Success!", "Failed: ...").

3. Real-time Feedback via Events

The dApp listens to Transfer events and auto-updates balance when the user is involved. Status messages inform about detected transfers - users see changes instantly without refresh.

4. Educational Gas Estimator

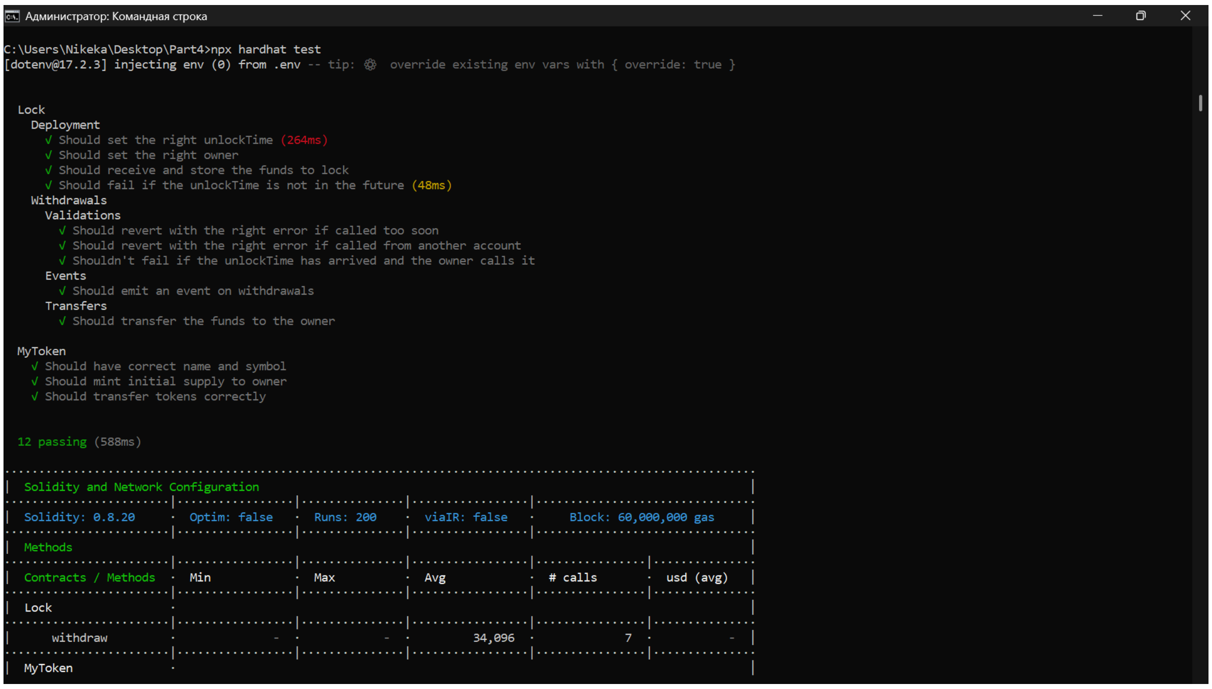
The gas comparison tool (successful vs intentional failing transfer) teaches users about gas costs and reverts. It shows real numbers (e.g., ~50k gas success vs revert) and helps understand blockchain mechanics.

5. Modular & Maintainable Code

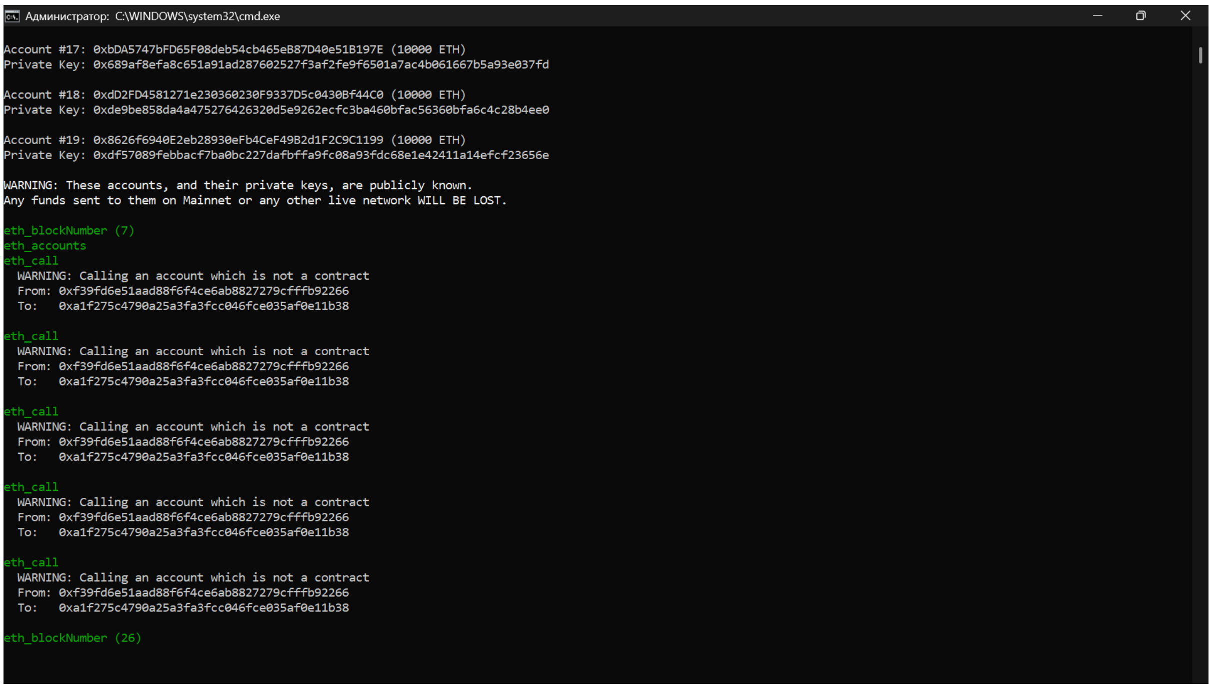
TokenManager class separates UI from blockchain logic - easy to extend (add mint/burn later). Error handling is centralized for robustness.

Trade-offs

- Minimal loading indicators to avoid visual noise.
- Focus on core requirements (no transaction history).
- Future improvements: transaction links, loading spinners, multi-token support.



```
Администратор: Командная строка

C:\Users\Nikeka\Desktop\Part4>npx hardhat test
[dotenv@17.2.3] injecting env (0) from .env -- tip: ⚙  override existing env vars with { override: true }


  Lock
    Deployment
      √ Should set the right unlockTime (264ms)
      √ Should set the right owner
      √ Should receive and store the funds to lock
      √ Should fail if the unlockTime is not in the future (48ms)
    Withdrawals
      Validations
        √ Should revert with the right error if called too soon
        √ Should revert with the right error if called from another account
        √ Shouldn't fail if the unlockTime has arrived and the owner calls it
      Events
        √ Should emit an event on withdrawals
      Transfers
        √ Should transfer the funds to the owner

  MyToken
    √ Should have correct name and symbol
    √ Should mint initial supply to owner
    √ Should transfer tokens correctly


  12 passing (588ms)

·························································································
|  Solidity and Network Configuration                                                   |
························|·················|···············|·················|·····························
|  Solidity: 0.8.20     ·  Optim: false   ·  Runs: 200    ·  viaIR: false   ·     Block: 60,000,000 gas  |
························|·················|···············|·················|·····························
|  Methods                                                                              |
························|·················|···············|·················|················|·············
|  Contracts / Methods  ·  Min            ·  Max          ·  Avg            ·  # calls       ·  usd (avg)  |
························|·················|···············|·················|················|·············
|  Lock                 ·                                                                               |
························|·················|···············|·················|················|·············
|      withdraw         ·              -  ·            -  ·         34,096  ·             7  ·          -  |
························|·················|···············|·················|················|·············
|  MyToken              ·                                                                               |
```



```
Администратор:  C:\WINDOWS\system32\cmd.exe

Account #17: 0xbDA5747bFD65F08deb54cb465eB87D40e51B197E (10000 ETH)
Private Key: 0x689af8efa8c651a91ad287602527f3af2fe9f6501a7ac4b061667b5a93e037fd

Account #18: 0xdD2FD4581271e230360230F9337D5c0430Bf44C0 (10000 ETH)
Private Key: 0xde9be858da4a475276426320d5e9262ecfc3ba460bfac56360bfa6c4c28b4ee0

Account #19: 0x8626f6940E2eb28930eFb4CeF49B2d1F2C9C1199 (10000 ETH)
Private Key: 0xdf57089febbacf7ba0bc227dafbffa9fc08a93fdc68e1e42411a14efcf23656e

WARNING: These accounts, and their private keys, are publicly known.
Any funds sent to them on Mainnet or any other live network WILL BE LOST.

eth_blockNumber (7)
eth_accounts
eth_call
  WARNING: Calling an account which is not a contract
  From: 0xf39fd6e51aad88f6f4ce6ab8827279cfffb92266
  To:   0xa1f275c4790a25a3fa3fcc046fce035af0e11b38

eth_call
  WARNING: Calling an account which is not a contract
  From: 0xf39fd6e51aad88f6f4ce6ab8827279cfffb92266
  To:   0xa1f275c4790a25a3fa3fcc046fce035af0e11b38

eth_call
  WARNING: Calling an account which is not a contract
  From: 0xf39fd6e51aad88f6f4ce6ab8827279cfffb92266
  To:   0xa1f275c4790a25a3fa3fcc046fce035af0e11b38

eth_call
  WARNING: Calling an account which is not a contract
  From: 0xf39fd6e51aad88f6f4ce6ab8827279cfffb92266
  To:   0xa1f275c4790a25a3fa3fcc046fce035af0e11b38

eth_call
  WARNING: Calling an account which is not a contract
  From: 0xf39fd6e51aad88f6f4ce6ab8827279cfffb92266
  To:   0xa1f275c4790a25a3fa3fcc046fce035af0e11b38

eth_blockNumber (26)
```

Part B

Section B - ERC Standards Analysis (5–8 paragraphs)

ERC-20 is the foundational standard for fungible tokens on Ethereum, conceptually modeling regular currency or shares where all tokens are identical, divisible, and interchangeable. The core model relies on simple state structures: a mapping of addresses to balances (balanceOf), a totalSupply variable for the overall token count, and an allowance mapping for delegated spending (approve/transferFrom). Functions such as transfer and transferFrom enable token movement, approve allows granting permission to spend tokens, and events

Transfer and Approval provide logging for external indexing, wallets, and exchanges. In my project, MyToken.sol implements a basic ERC-20 with manual balance management, mint, and burn functions, demonstrating the pure conceptual model without relying on external libraries.

ERC-721, in contrast, is the standard for non-fungible tokens (NFTs), where each token is unique and identified by a tokenId (uint256). The architecture centers around individual ownership tracking: a mapping from tokenId to owner, optional enumeration mappings (owner to tokenIds), and per-token or operator-level approvals (setApprovalForAll). Functions like safeTransferFrom include safeguards such as onERC721Received callbacks to prevent tokens from being locked in incompatible contracts. This design enables true uniqueness, making ERC-721 ideal for digital art, collectibles, in-game items, and virtual land. The metadata extension (tokenURI) returns a URI pointing to JSON containing name, description, image, and attributes — typically stored off-chain on IPFS for decentralization and cost efficiency.

Interfaces (IERC20 and IERC721) play a critical role by defining expected behavior through function signatures and events without any implementation. They ensure interoperability: any contract implementing IERC20 can seamlessly integrate with decentralized exchanges, wallets, and aggregators. The OpenZeppelin library provides audited, battle-tested implementations of these interfaces, reducing common vulnerabilities. While my MyToken.sol is a manual implementation for educational purposes, real-world projects should inherit from OpenZeppelin's ERC20 or ERC721 contracts to ensure security and compliance.

State management differs significantly between the two standards. ERC-20 uses lightweight mappings (address -> uint256), enabling fast updates and batch transfers with low storage costs. Gas for a typical transfer is around 21–30k. ERC-721, however, requires per-token tracking (tokenId -> owner), counters for minting, and additional events per token, leading to higher storage costs and gas usage (typically 50-70k+ for transfers). This makes ERC-721 less suitable for high-volume applications but perfect for scenarios where uniqueness and provenance matter.

Security assumptions vary accordingly. Both standards rely on the checks-effects-interactions pattern to prevent reentrancy, built-in overflow protection (Solidity ≥0.8), and proper approval handling. ERC-20 is vulnerable to front-running on approvals (race conditions), while ERC-721 introduces risks in on ERC721Received callbacks and metadata centralization (if URI points to a centralized server, it can be censored or become unavailable). Both require trust in the contract owner for minting/burning and should undergo audits.

Gas usage differences are substantial: ERC-20 is highly optimized for efficiency and liquidity (ideal for DeFi, utility tokens, and payments), while ERC-721 incurs higher costs due to individual token tracking (better suited for low-volume, high-value assets). Metadata design patterns in ERC-721 include fully on-chain storage (expensive but immutable), off-chain IPFS/Arweave (decentralized but requires pinning services), and dynamic URIs (generated on-the-fly for evolving NFTs). ERC-20 metadata remains minimal (name/symbol/decimals), though extensions like ERC-20 Permit enable gasless approvals via off-chain signatures.

In summary, ERC-20 prioritizes fungibility, liquidity, and low-cost operations, while ERC-721 emphasizes uniqueness, ownership, and collectibility. The interface-based design ensures ecosystem-wide compatibility, and careful implementation (preferably via audited libraries) is essential for both standards to mitigate common vulnerabilities.

PART 5

Part 6

Contents:

Abstract

This document presents a synthesis of peer-reviewed academic research in four key areas: decentralized systems, Ethereum architecture design, JavaScript-based blockchain interaction, and modern full-stack blockchain development. Based on this literature review, a methodological design for the final course project is proposed, including system architecture, data flow descriptions, smart contract structure, selected libraries and frameworks, security methodology, threat model, and testing methodology.

Introduction

Decentralized systems and smart contract platforms, particularly Ethereum, change how we store data, execute transactions, and build trustless applications. This literature review summarizes key findings from peer-reviewed research and proposes a practical plan to build a full-stack decentralized application (dApp) using a JavaScript stack.

Literature Review

The review synthesizes findings from academic sources about four topics: decentralized systems, Ethereum design, JavaScript-based blockchain interaction, and modern full-stack blockchain development.

1. Decentralized Systems

Decentralized systems distribute control and trust among many participants and rely on cryptography and consensus protocols to reach agreement. Key points from the literature:

- Design goals: decentralization, censorship resistance, fault tolerance, and predictable incentives for participants.
- Performance and scalability: transaction throughput and latency remain primary technical limits. Layer-2 solutions (rollups, sidechains) and sharding are commonly suggested approaches to scale systems.
- Tradeoffs: many papers discuss the trilemma between decentralization, security, and scalability, and propose different balances for different use cases.

2. Ethereum Architecture Design

Ethereum provides a programmable blockchain with the Ethereum Virtual Machine (EVM) as a general computation layer. Important architectural aspects:

- EVM and gas model: gas limits the cost of computation and prevents infinite loops. Developers must optimize contracts to reduce gas costs.
- Consensus evolution: Ethereum's move from Proof-of-Work to Proof-of-Stake changes economics, finality properties, and energy usage; research explores its effects on security and decentralization.
- Node types and storage: full, archive, and light nodes have different storage and validation responsibilities. Research explores optimizations for archival storage and indexing.

Practical implication: dApp design must account for gas costs, confirmation times, and efficient on-chain/off-chain data partitioning.

3. JavaScript-Based Blockchain Interaction

JavaScript libraries connect web and backend applications to Ethereum nodes and smart contracts. Key points:

- Libraries: Web3.js and Ethers.js are widely used. Ethers.js is often preferred for a simple API and TypeScript support.
- Developer tools: Hardhat and Truffle help compile, test, and deploy contracts. MetaMask and WalletConnect enable user wallet integration.
- Patterns: common patterns include using the backend to index events and provide aggregated views, while the frontend performs direct contract calls for user actions.

4. Modern Full-Stack Blockchain Development

Building a reliable dApp requires careful design across smart contracts, backend services, and frontend UX.

- Architectural patterns: separation of concerns — keep business logic that must be decentralized on-chain and auxiliary logic off-chain.
- Testing and CI: unit tests for contracts, integration tests combining JS and contracts, and automated security scans are essential for safe releases.
- Security practices: use vetted libraries (OpenZeppelin), static analysis (Slither, Mythril), fuzzing (Echidna), and manual code review.

Synthesis and Gaps in Research

From the literature:

- Engineering-focused research emphasizes scalability, storage, and consensus, while fewer studies rigorously treat user experience and front-end integration.
- Testing and benchmarking practices vary; the community lacks a single standard for reproducible performance and security testing in dApp development.

These gaps suggest opportunities to contribute practical workflows and reproducible test suites for full-stack blockchain projects.

Conclusion

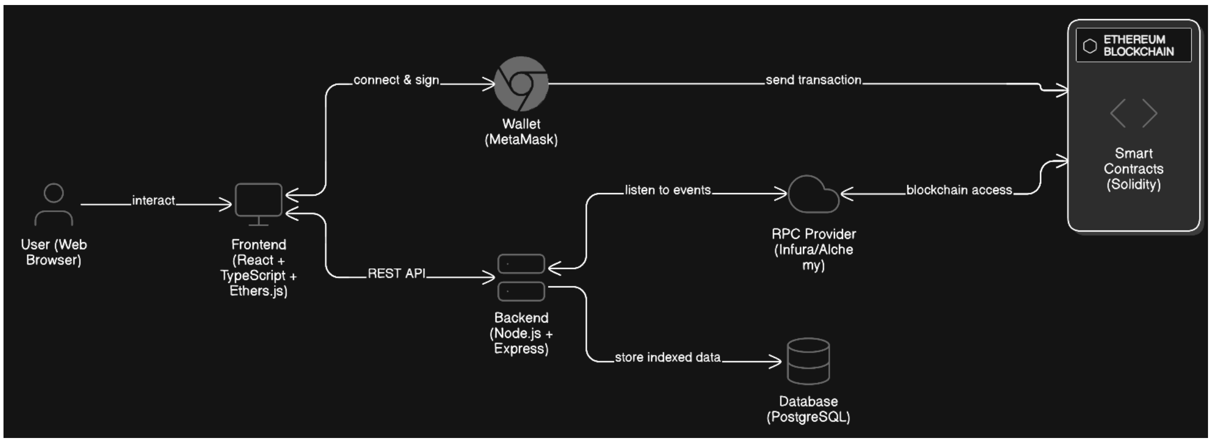
A secure and scalable dApp requires combining knowledge of blockchain internals, smart contract best practices, and modern JavaScript tooling. The methodological design below translates literature findings into a practical plan for the final project.

Methodological Designt

1. System Architecture

Components:

- Frontend: React + TypeScript + Ethers.js. Handles user interface and wallet interactions (MetaMask).
- Backend: Node.js + Express. Indexes blockchain events and stores aggregated data in PostgreSQL. Provides REST API for frontend.
- Smart Contracts: Solidity contracts using OpenZeppelin where appropriate.
- Ethereum network: local Hardhat for development, testnet (e.g., Goerli) for staging, Mainnet for production deployment.



2. Data Flow Descriptions (DFD)

High-level flow:

1. User interacts with frontend and requests an action (e.g., deposit).
2. Frontend prepares a transaction and asks the user to sign it via MetaMask.
3. The transaction is broadcast to the Ethereum network and included in a block.
4. The smart contract executes and emits events.
5. The backend listens for events (via WebSocket or node provider) and writes relevant data to PostgreSQL.
6. Frontend queries backend APIs for historical or aggregated data; it also listens to contract events for real-time updates.

(You can include graphical DFD diagrams in the final PDF.)

3. Smart Contract Structure

A simple contract example (Solidity):

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;


import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

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


contract MyProject is Ownable, ReentrancyGuard {

    mapping(address => uint256) public balances;
    event Deposit(address indexed user, uint256 amount);

    event Withdraw(address indexed user, uint256 amount);


    constructor() {}


    function deposit() external payable {

        require(msg.value > 0, "Zero amount");

        balances[msg.sender] += msg.value;

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

    }


    function withdraw(uint256 amount) external nonReentrant {

        require(balances[msg.sender] >= amount, "Insufficient balance");

        balances[msg.sender] -= amount;

        payable(msg.sender).transfer(amount);

        emit Withdraw(msg.sender, amount);

    }


    function emergencyWithdraw(address to) external onlyOwner {

        // Admin-only emergency logic

    }

}
```

Security notes:

- Use checks-effects-interactions pattern.
- Apply ReentrancyGuard to state-changing functions with external calls.
- Keep admin functions minimal and protected (consider multisig on production).

4. Chosen Libraries & Frameworks

- Solidity for contracts.
- OpenZeppelin for token standards and security helpers.
- Hardhat for local development, testing, and deployment.
- Ethers.js for JavaScript interaction (frontend and backend).
- React + TypeScript for the frontend.
- Node.js + Express for backend services.
- PostgreSQL for indexing and caching on-chain data.
- Slither, Mythril, Echidna for security analysis and fuzzing.

5. Security Methodology

Layered approach: prevention, automatic scanning, and manual review.

Steps:

1. Rely on audited libraries (OpenZeppelin).
2. Run static analysis tools (Slither, Mythril) in CI.
3. Create comprehensive unit tests for contracts (Hardhat tests).
4. Use fuzzing (Echidna) and property-based tests for invariants.
5. Conduct manual code reviews and arrange external audits for release versions.

6. Threat Model

Potential attackers and capabilities:

- External attacker: can call public contract methods and send arbitrary transactions.
- Malicious node/RPC provider: could censor transactions or provide incorrect chain data to a backend that relies on a single provider.
- Compromised admin keys: could misuse privileged functions.

Attack surfaces:

- Smart contract logic (reentrancy, arithmetic bugs, incorrect access control).
- Frontend (phishing, malicious JS, wallet spoofing).
- Backend (exposed APIs, compromised DB, insecure RPC keys).

Mitigations:

- Harden contracts (use ReentrancyGuard, limit admin powers, role-based access).
- Use HTTPS and Content Security Policy on frontend; educate users to verify wallets.
- Run multiple RPC providers and verify event data consistency; rotate keys and store secrets securely.

7. Testing Methodology

Unit tests (contracts):

- Write tests for every function, including edge cases and failure modes.
- Use Hardhat test environment with Waffle/Chai.

Integration tests (JS + contracts):

- Simulate full user flows: frontend signs transaction, contract state changes, backend captures events.

Dynamic testing / fuzzing:

- Use Echidna to test properties and find unexpected edge cases.
- Use Slither and Mythril for automated vulnerability detection.

Load and E2E tests:

- Use a local forked mainnet or a local cluster to simulate many transactions and measure gas and confirmation behavior.

CI/CD:

- GitHub Actions runs tests and security scanners on each push. Deploy to testnet via tagged releases.

Appendix: Test Plan (brief)

1. Unit tests: aim for at least 90% function coverage for contract logic.
2. Integration: 15-25 scenarios covering main user journeys.
3. Fuzzing: 5-10 properties checked by Echidna.
4. Security scans: run Slither and Mythril in CI on every commit.

References (peer-reviewed and academic sources)

1. Dinh, T. N., et al., "Untangling Blockchain: A Data Processing View of Blockchain Systems," 2017. (arXiv/Conference paper)

   https://arxiv.org/abs/1708.05665?utm_source=chatgpt.com

2. Tabatabaei, M., et al., "Understanding Blockchain: Definitions, Architecture, Design and Research Directions," 2022. (arXiv/Review)

   https://arxiv.org/abs/2207.02264?utm_source=chatgpt.com

3. Faqir-Rhazoui, Y., Arroyo, D., "A Comparative Analysis of Platforms for DAOs on Ethereum," Journal of Internet Services and Applications, 2021.

   https://jisajournal.springeropen.com/articles/10.1186/s13174-021-00139-6?utm_source=chatgpt.com

4. Feng Hang, et al., "SLIMARCHIVE: Lightweight Ethereum Archive Nodes," USENIX ATC, 2024.

   https://www.usenix.org/system/files/atc24-feng-hang.pdf?utm_source=chatgpt.com

5. (Systematic Review) "Blockchain Software Patterns for the Design of Decentralized Applications," 2022.

   https://www.sciencedirect.com/science/article/pii/S209672092200001X?utm_source=chatgpt.com

6. Lahami, M., et al., "A Systematic Literature Review on Dynamic Testing of Blockchain Oriented Software," 2024.

   https://www.sciencedirect.com/science/article/pii/S0167642324001345?utm_source=chatgpt.com

7. Iuliano, G., Di Nucci, D., "Smart Contract Vulnerabilities, Tools, and Benchmarks," 2024.

   https://arxiv.org/abs/2412.01719?utm_source=chatgpt.com

8. Rek, P., et al., "Complexity Analysis of Decentralized Application Development," CEUR Workshop Proceedings, 2022.

   https://ceur-ws.org/Vol-2508/paper-rek.pdf?utm_source=chatgpt.com

PART 7

Section A

Block Structure (Headers, Metadata, Transaction Trie)

A typical blockchain block consists of two main parts: block header and block body (transactions list).

Block header (fixed-size, ~80-200 bytes in Bitcoin/Ethereum):

- Version — protocol version (4 bytes)
- Previous block hash - SHA-256(SHA-256(prev_header)) in Bitcoin, Keccak-256 in Ethereum (32 bytes)
- Merkle root - root hash of the Merkle tree of transactions (32 bytes)
- Timestamp - Unix time (4–8 bytes)
- Difficulty target / bits - compact representation of current PoW difficulty (4 bytes in Bitcoin)
- Nonce - 32-bit value for PoW mining (4 bytes in Bitcoin)
- Extra data (Ethereum) - arbitrary data field (up to 32 bytes in post-merge)

Metadata includes uncle hashes (Ethereum pre-merge), state root, receipts root, and mixHash/ommers.

Transaction trie (Merkle Patricia Trie in Ethereum):

- All transactions in a block are organized into a Merkle Patricia Trie (MPT)
- MPT combines Merkle tree (for proof) with Patricia trie (for efficient key-value lookup)
- Key = RLP-encoded transaction index
- Value = RLP-encoded transaction
- Merkle root stored in header ensures integrity

Merkle Trees (Proof Generation, Verification)

Merkle tree is a binary tree where:

- Leaf nodes = hashes of individual data (transactions)
- Internal nodes = hash(left child || right child)

Proof generation:

- To prove a transaction is included: collect sibling hashes along the path to root
- Proof size = O(log n), where n = number of transactions

Verification:

- Recompute root from leaf + proof siblings
- Compare with stored Merkle root in header
- If match -> transaction is included and untampered

In Ethereum, Merkle proofs are used for light clients and stateless clients (EIP-4444, Verkle trees future).

Link Structures Forming the Chain

Chain is formed by cryptographic linking:

- Each block header contains hash of previous block header
- Tampering any block invalidates all subsequent blocks (due to hash chain)
- Longest (most work) chain rule in Nakamoto consensus
- In Ethereum post-merge: finalized chain via Gasper (LMD-GHOST + Casper FFG)

Networking & Node Discovery

Ethereum uses devp2p (RLPx protocol) over TCP/UDP:

- Node discovery via UDP-based Kademlia DHT (discv4/discv5)
- Nodes share ENR (Ethereum Node Record) with IP, port, pubkey
- Peer selection: random + reputation-based
- Gossip protocol for transaction & block propagation

Transaction Propagation Model

1. User signs tx -> broadcasts to connected peers
2. Peers validate (gas, nonce, signature) -> add to mempool -> gossip to others
3. Miners/validators select from mempool for inclusion
4. Propagation delay: ~3-12 seconds globally (Ethereum mainnet)
5. MEV (Maximal Extractable Value) bots re-order transactions in mempool

Mining Process (PoW)

1. Assemble candidate block (header + txs)
2. Compute hash(header + nonce)
3. Find nonce such that hash < target (difficulty)
4. Target adjusted every 2016 blocks (Bitcoin) or dynamically (Ethereum pre-merge)
5. First miner broadcasts valid block -> chain tip updates

Validation & Finality in PoS (Ethereum post-merge)

- Validators stake 32 ETH
- Chosen via RANDAO + stake-weighted randomness
- Propose/attest blocks in slots (12s) and epochs (32 slots)
- Casper FFG provides economic finality: 2/3 validators attest → checkpoint finalized
- LMD-GHOST selects fork with most recent attestations

Fork Choice Rules

- Nakamoto (Bitcoin): longest chain with most cumulative PoW
- Gasper (Ethereum post-merge): combination of LMD-GHOST (latest message-driven GHOST) for chain selection + Casper FFG for finality
    - LMD-GHOST: fork with highest justified checkpoint + most attestations
    - Finality after ~12–13 minutes (2 epochs)

Section B

Comparison of PoW, PoS, and DPoS

1. Security Guarantees

- PoW (Bitcoin, Ethereum pre-merge): Security via computational work. 51% attack requires majority hash power. Very high cost barrier (ASICs, electricity). Nakamoto consensus provides probabilistic finality (after 6 confirmations ~1 hour).
- PoS (Ethereum post-merge, Cardano Ouroboros): Economic security. Attack costs 33%–51% stake slashing. Economic finality (~12 min in Ethereum). Ouroboros Praos provides stronger guarantees against adaptive adversaries via private leader election.
- DPoS (EOS, Tron): Delegated security. 21–27 super representatives elected. Fast finality (~3s). Vulnerable to collusion among delegates.

2. Attack Surfaces

- PoW: 51% attack, selfish mining, block withholding
- PoS: Long-range attack (historical), nothing-at-stake, grinding attacks (mitigated in Casper/Gasper)
- DPoS: Cartelization, bribery of delegates, centralization of power

3. Finality

- PoW: Probabilistic (longer chain wins)
- PoS (Ethereum): Economic + probabilistic (finalized after 2/3 attestations)
- DPoS: Instant/optimistic finality (BFT-style)

4. Decentralization Tradeoffs

- PoW: High decentralization (thousands of miners), but mining pools centralize power
- PoS: Medium-high (hundreds of thousands validators), but stake concentration risk
- DPoS: Low (21=27 delegates), high performance but centralization risk

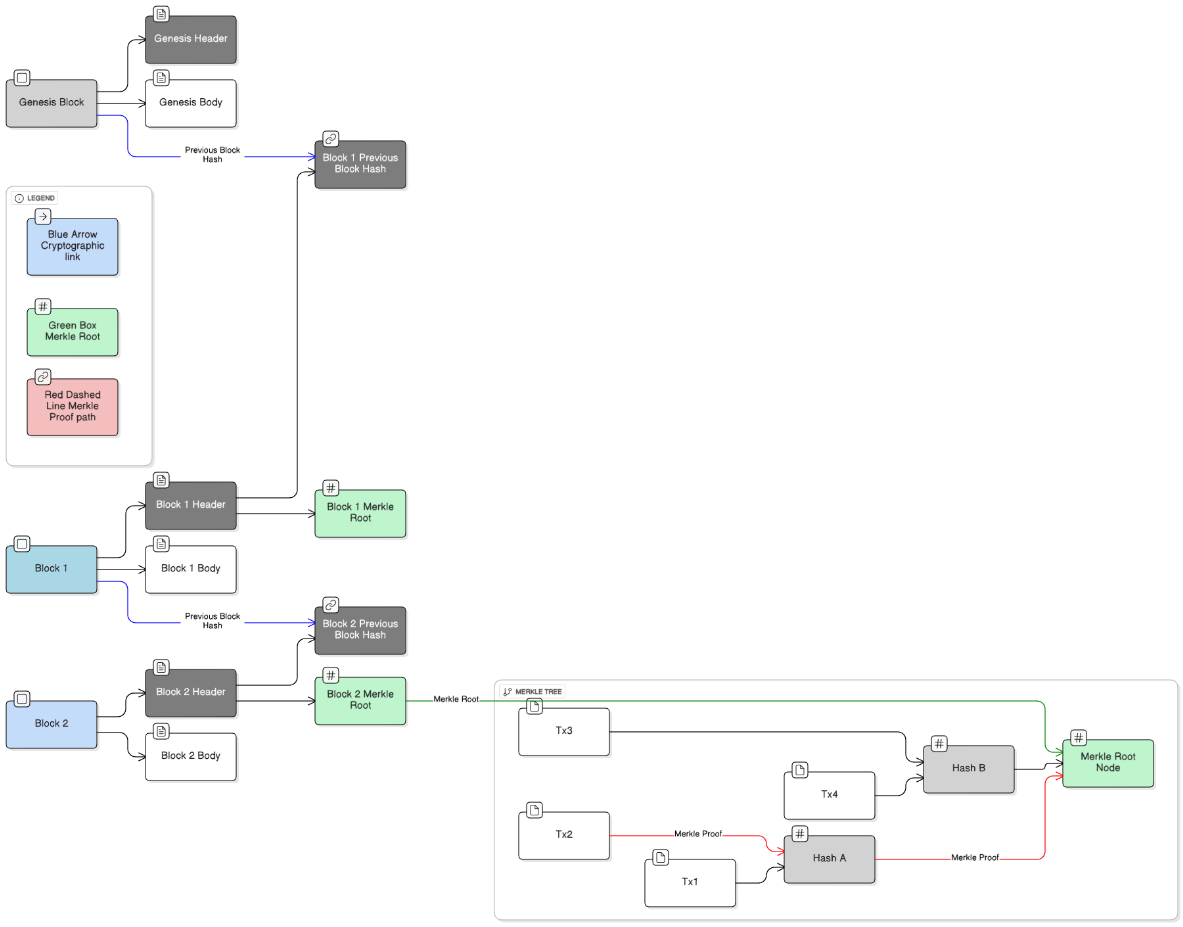
5. Economic Incentives

- PoW: Block reward + fees -> energy-intensive
- PoS: Staking rewards + fees -> sustainable, slashing for misbehavior
- DPoS: Block producer rewards + fees -> voting incentives

6. Hardware Requirements

- PoW: ASIC/GPU farms, high electricity
- PoS: Ordinary servers/laptops (32 ETH minimum)
- DPoS: High-performance servers for delegates

Summary: PoW offers strongest censorship resistance but high energy cost. PoS balances security, sustainability and decentralization. DPoS maximizes performance at the cost of centralization.


Genesis Header
Genesis Block
Genesis Body
Previous Block Hash
Block 1 Previous Block Hash
LEGEND
Blue Arrow Cryptographic link
Green Box Merkle Root
Red Dashed Line Merkle Proof path
Block 1 Header
Block 1 Merkle Root
Block 1
Block 1 Body
Previous Block Hash
Block 2 Previous Block Hash
Block 2 Header
Block 2 Merkle Root
Merkle Root
MERKLE TREE
Block 2
Block 2 Body
Tx3
Hash B
Merkle Root Node
Tx4
Tx2
Merkle Proof
Hash A
Merkle Proof
Tx1