Assignment 3

Team:

RNT corp.

Members: Ramazan, Nuray, Temirlan

PART 1 - Deep Theoretical Questions

1. Web3.js vs Ethers.js

Architectural differences:

Web3.js has a monolithic architecture, while Ethers.js is modular and lightweight. This makes Ethers.js easier to maintain and more secure for modern dApps.

Provider abstraction models:

Ethers.js uses explicit and well-defined provider classes, while Web3.js relies on a more implicit global provider. Explicit providers reduce the risk of accidental or malicious provider misuse.

Signing, ABI parsing and contract interaction:

Ethers.js performs stricter ABI parsing and safer transaction signing compared to Web3.js. This reduces runtime errors and improves contract interaction reliability.

Advantages of Ethers.js v6 modular design:

Ethers.js v6 separates functionality into modules, reducing bundle size and improving performance. The modular design also limits attack surface and improves security.

Security implications of private key handling:

Ethers.js never directly accesses private keys and delegates signing to external wallets like MetaMask. This prevents private keys from being exposed to the frontend code.

Gas estimation and error propagation:

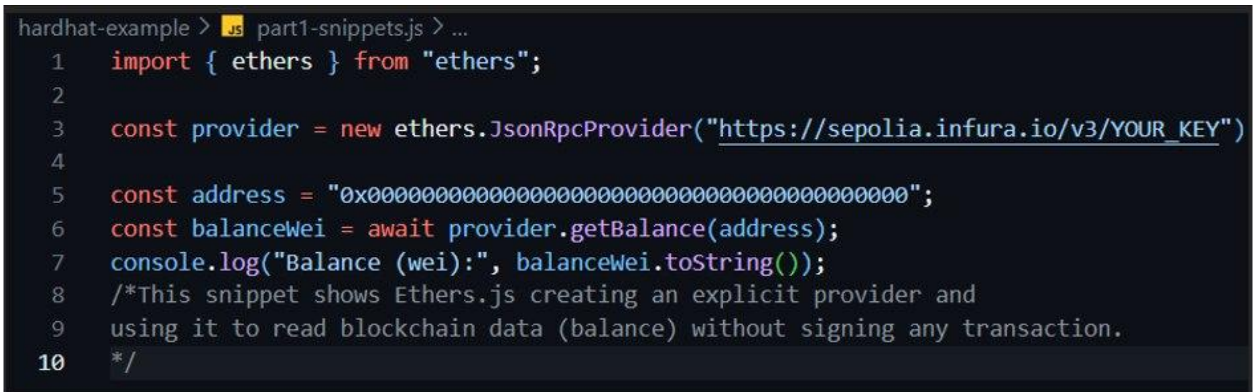
Ethers.js provides more accurate gas estimation and more descriptive error messages than Web3.js. Better error propagation improves debugging and user experience.

Why modern dApps migrate from Web3.js to Ethers.js:

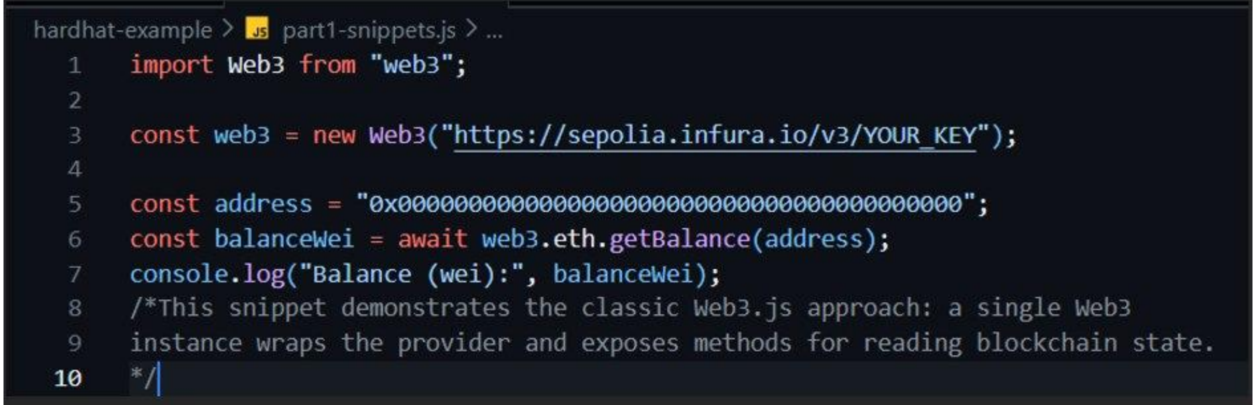
Modern dApps migrate to Ethers.js because it offers better security, clearer APIs, and active maintenance. It is also better suited for modern frontend frameworks.

Debugging user experience:

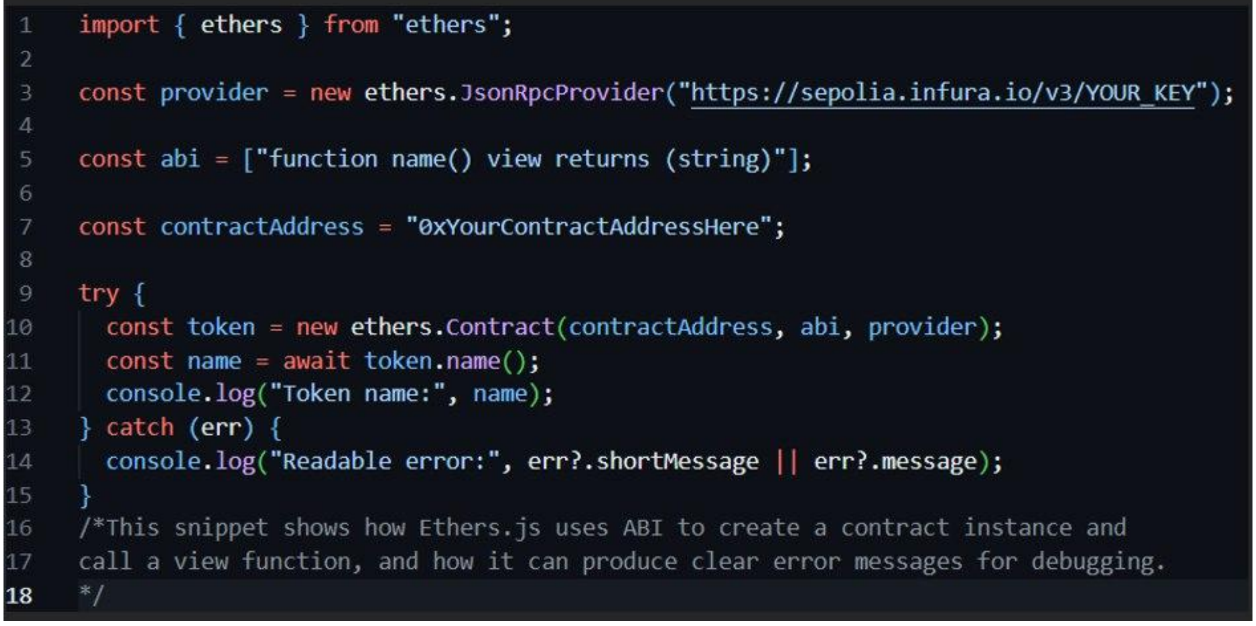
Ethers.js provides clearer stack traces and more readable error messages than Web3.js. This makes debugging faster and more developer-friendly.


hardhat-example > part1-snippets.js > ...

```javascript
import { ethers } from "ethers";

const provider = new ethers.JsonRpcProvider("https://sepolia.infura.io/v3/YOUR_KEY")

const address = "0x0000000000000000000000000000000000000000";
const balanceWei = await provider.getBalance(address);
console.log("Balance (wei):", balanceWei.toString());
/*This snippet shows Ethers.js creating an explicit provider and
using it to read blockchain data (balance) without signing any transaction.
*/
```


hardhat-example > part1-snippets.js > ...

```javascript
import Web3 from "web3";

const web3 = new Web3("https://sepolia.infura.io/v3/YOUR_KEY");

const address = "0x0000000000000000000000000000000000000000";
const balanceWei = await web3.eth.getBalance(address);
console.log("Balance (wei):", balanceWei);
/*This snippet demonstrates the classic Web3.js approach: a single Web3
instance wraps the provider and exposes methods for reading blockchain state.
*/
```



```javascript
import { ethers } from "ethers";

const provider = new ethers.JsonRpcProvider("https://sepolia.infura.io/v3/YOUR_KEY");

const abi = ["function name() view returns (string)"];

const contractAddress = "0xYourContractAddressHere";

try {
  const token = new ethers.Contract(contractAddress, abi, provider);
  const name = await token.name();
  console.log("Token name:", name);
} catch (err) {
  console.log("Readable error:", err?.shortMessage || err?.message);
}
/*This snippet shows how Ethers.js uses ABI to create a contract instance and
call a view function, and how it can produce clear error messages for debugging.
*/
```

2. Connecting to the Ethereum Network

JSON-RPC structure and common methods:

JSON-RPC is a standardized protocol used to communicate with Ethereum nodes using methods like eth_getBalance and eth_sendRawTransaction.

Local vs Testnet vs Mainnet RPC endpoints:

Local networks are used for development, testnets for safe testing, and mainnet for real-value transactions.
Testnets allow realistic blockchain interaction without financial risk.

Node types (full, archival, light, RPC proxy):

Full and archival nodes store complete blockchain data, light nodes store minimal data, and RPC proxy nodes provide remote access.
Most dApps rely on RPC proxy services to avoid running their own infrastructure.

Why rate limiting affects frontend design:

RPC providers limit the number of requests, forcing dApps to minimize unnecessary calls.
Efficient frontend design improves performance and reliability.

RPC virtualization (Infura, Alchemy, QuickNode):

RPC virtualization services abstract node infrastructure and provide scalable blockchain access.
They simplify dApp architecture and speed up development.

3. Transaction Lifecycle

1. Local Signing

When a user initiates a transaction, it is signed locally in the wallet (e.g., MetaMask), ensuring that the private key is never exposed to the application or the network.

2. Account Nonce Logic

Each transaction includes a nonce, which is a sequential number that guarantees correct transaction order and prevents replay attacks.

3. Mempool Propagation

After signing, the transaction is broadcast to the Ethereum network and stored in the mempool, where it waits to be picked up by a miner or validator.

4. Miner / Validator Selection

Miners (PoW) or validators (PoS) select transactions from the mempool based on gas fees and protocol rules to include them in the next block.

5. Block Inclusion

The selected transaction is included in a new block, which is then proposed and added to the blockchain.

6. Reorganization (Reorg) Risk

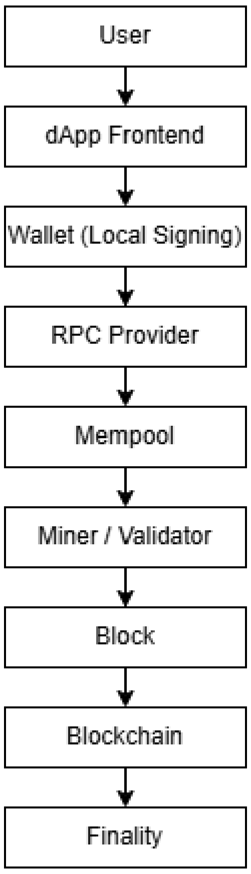
Before finality is reached, blocks can be replaced due to competing chains, temporarily removing transactions from the blockchain.

7. Finality Differences (PoW vs PoS)

In Proof-of-Work, finality is probabilistic and requires multiple confirmations, while in Proof-of-Stake, finality is faster and more deterministic.

8. Gas Fee Markets (EIP-1559 Analysis)

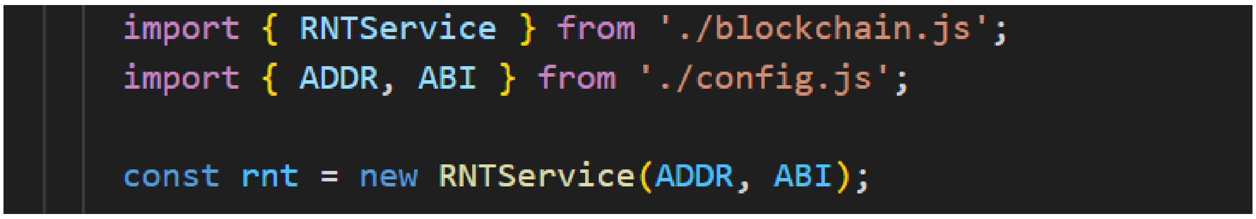
EIP-1559 introduced a base fee that is burned and a priority fee paid to validators, making transaction fees more predictable and reducing fee volatility.



The diagram illustrates how a transaction moves from user signing through mempool propagation to block inclusion and finality.

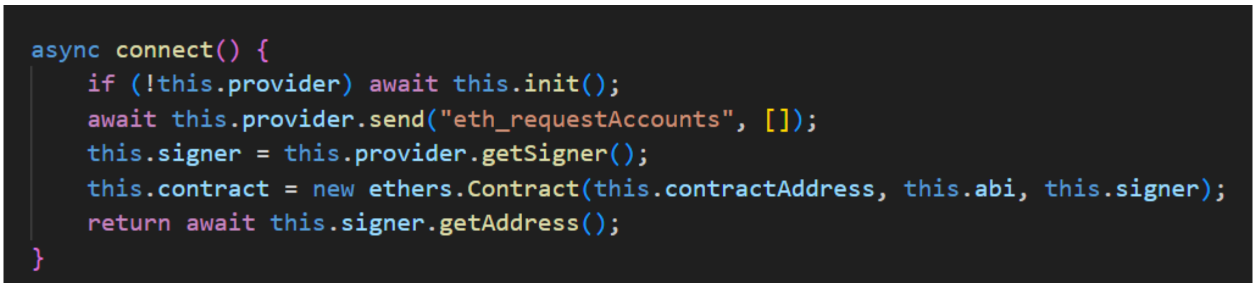
PART 2 - Practice Session: Minimal dApp Frontend

Task A - Implement Basic Frontend Interaction



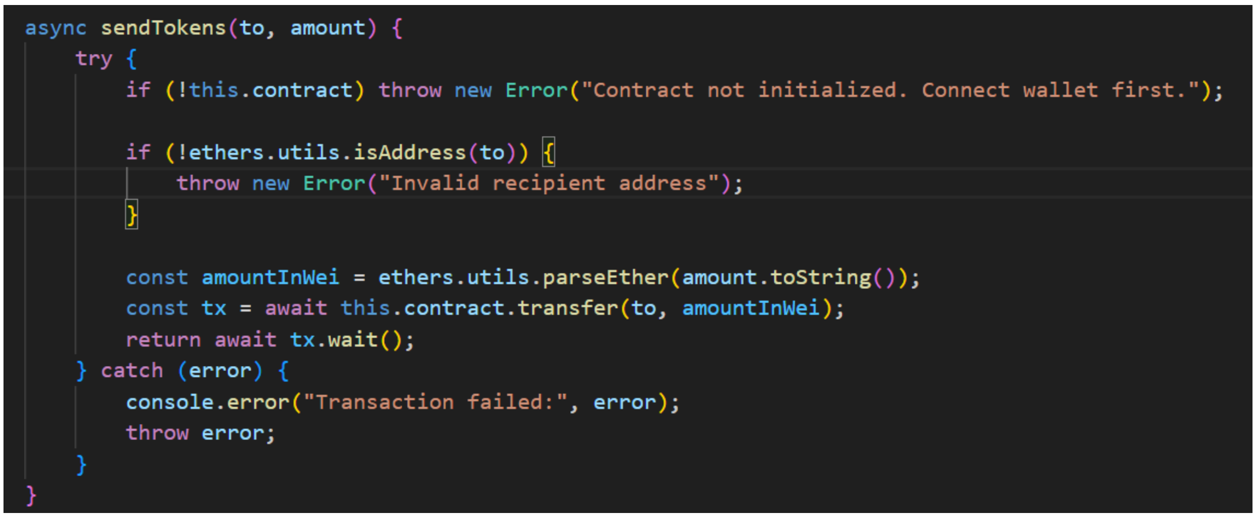
```
import { RNTService } from './blockchain.js';
import { ADDR, ABI } from './config.js';

const rnt = new RNTService(ADDR, ABI);
```

Laoding data from .env



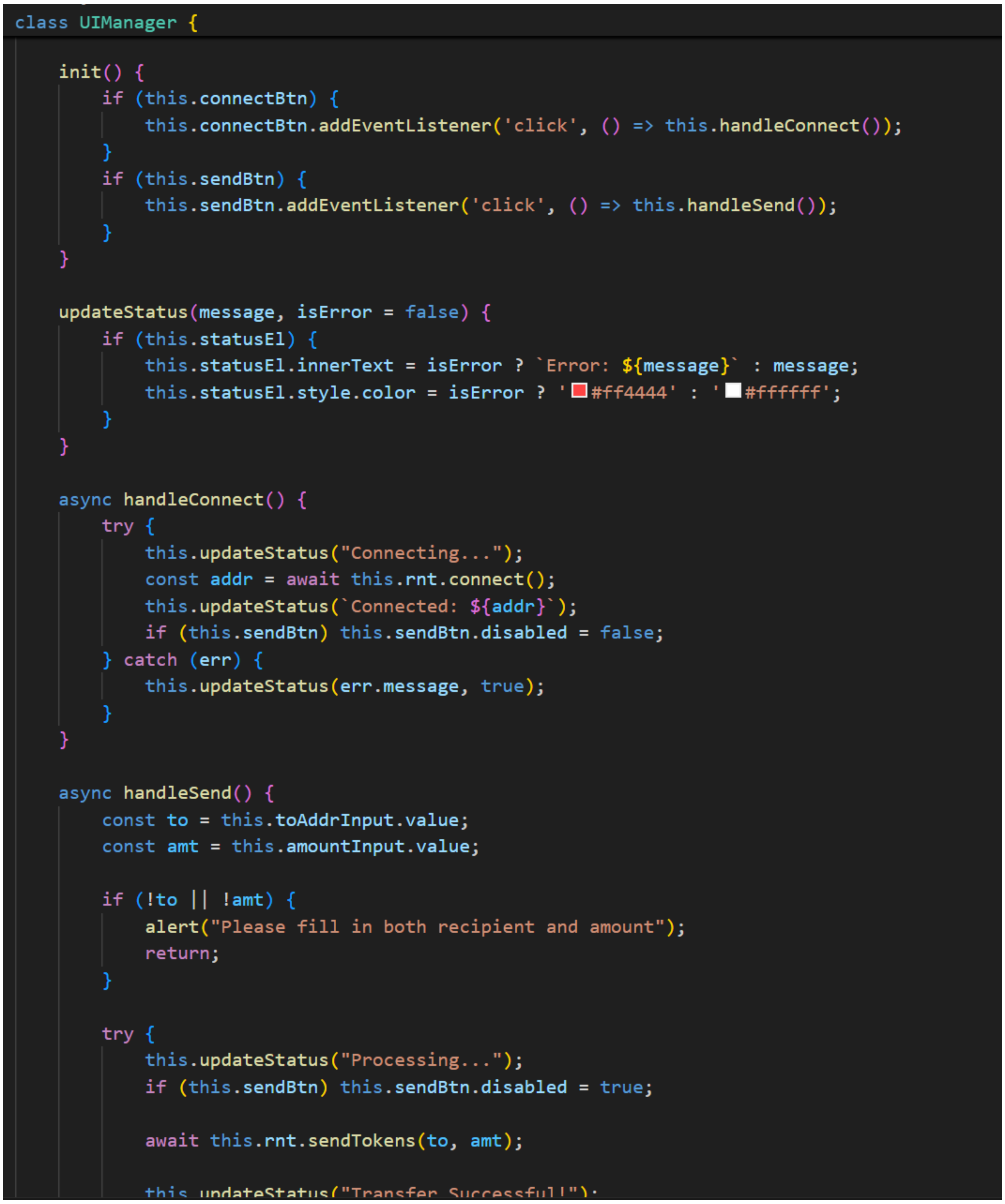
```
async connect() {
    if (!this.provider) await this.init();
    await this.provider.send("eth_requestAccounts", []);
    this.signer = this.provider.getSigner();
    this.contract = new ethers.Contract(this.contractAddress, this.abi, this.signer);
    return await this.signer.getAddress();
}
```

Method to check the connection



```
async sendTokens(to, amount) {
    try {
        if (!this.contract) throw new Error("Contract not initialized. Connect wallet first.");

        if (!ethers.utils.isAddress(to)) {
            throw new Error("Invalid recipient address");
        }

        const amountInWei = ethers.utils.parseEther(amount.toString());
        const tx = await this.contract.transfer(to, amountInWei);
        return await tx.wait();
    } catch (error) {
        console.error("Transaction failed:", error);
        throw error;
    }
}
```

Sending the token and verification of it



```javascript
class UIManager {

    init() {
        if (this.connectBtn) {
            this.connectBtn.addEventListener('click', () => this.handleConnect());
        }
        if (this.sendBtn) {
            this.sendBtn.addEventListener('click', () => this.handleSend());
        }
    }

    updateStatus(message, isError = false) {
        if (this.statusEl) {
            this.statusEl.innerText = isError ? `Error: ${message}` : message;
            this.statusEl.style.color = isError ? '#ff4444' : '#ffffff';
        }
    }

    async handleConnect() {
        try {
            this.updateStatus("Connecting...");
            const addr = await this.rnt.connect();
            this.updateStatus(`Connected: ${addr}`);
            if (this.sendBtn) this.sendBtn.disabled = false;
        } catch (err) {
            this.updateStatus(err.message, true);
        }
    }

    async handleSend() {
        const to = this.toAddrInput.value;
        const amt = this.amountInput.value;

        if (!to || !amt) {
            alert("Please fill in both recipient and amount");
            return;
        }

        try {
            this.updateStatus("Processing...");
            if (this.sendBtn) this.sendBtn.disabled = true;

            await this.rnt.sendTokens(to, amt);

            this.updateStatus("Transfer Successful!");
```

The UI handling class

Task B - Security Discussion

Why direct RPC exposure is dangerous

Direct RPC exposure is dangerous because attackers can abuse the endpoint to overload the service or analyze user activity.

XSS risks in dApps

XSS attacks in dApps can inject malicious scripts that trick users into signing unintended blockchain transactions.

Signature replay risks

Signature replay risks occur when a valid transaction signature is reused on another network or context.

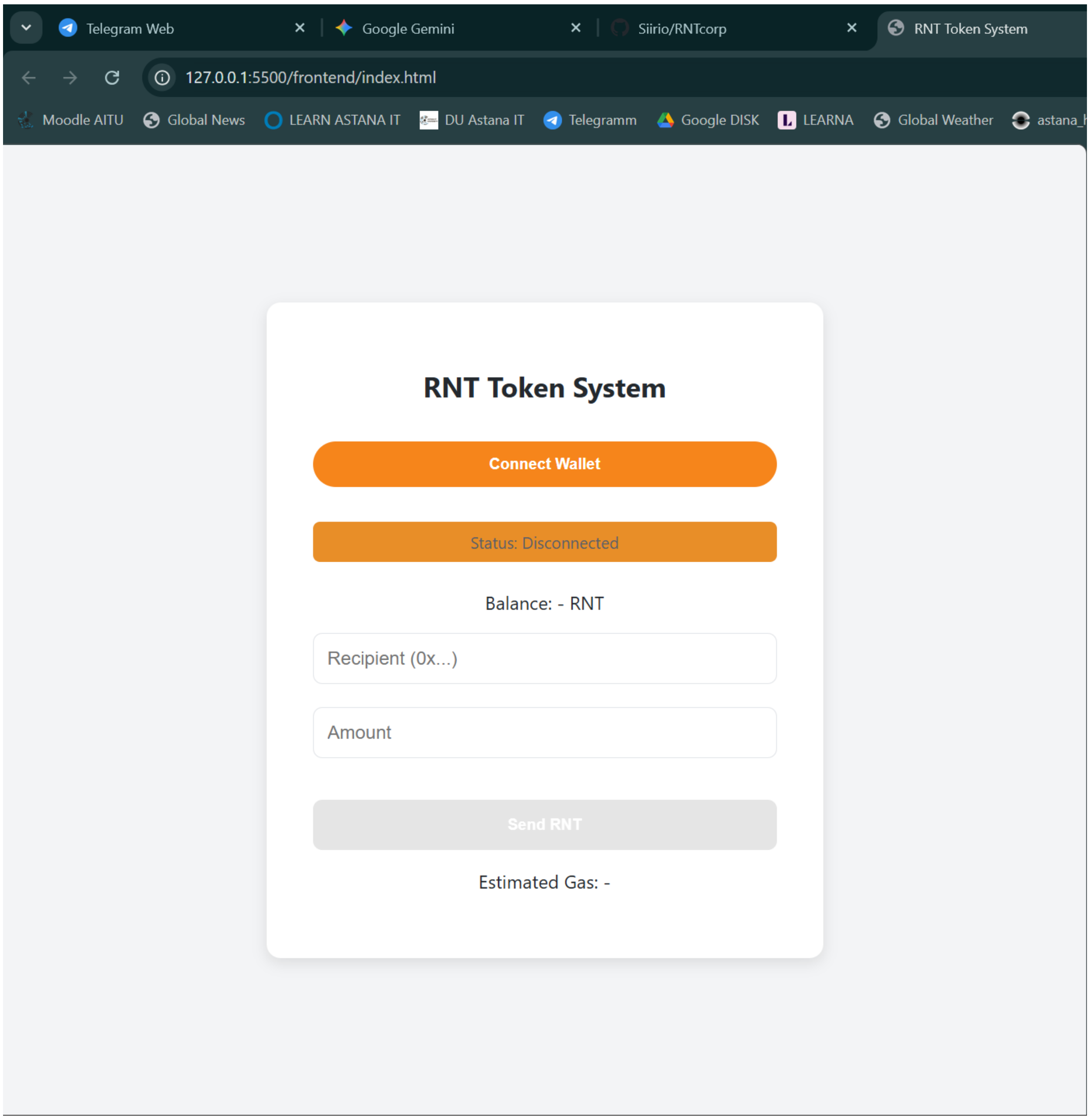
Private key extraction using malicious Web3 injection

Malicious Web3 injection can replace a trusted provider and deceive users into approving transactions that steal assets.

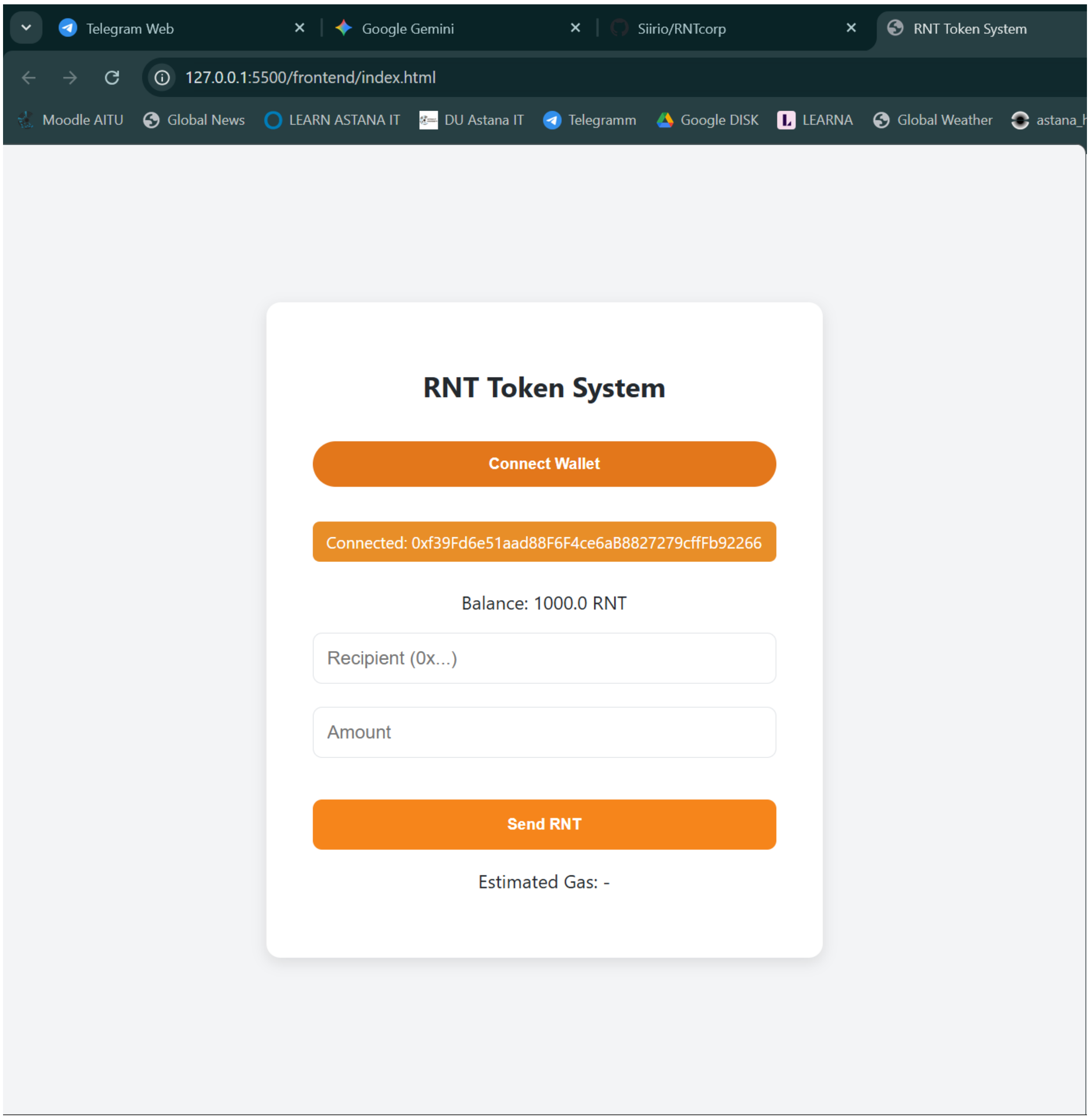
How modern wallet providers mitigate risks

Modern wallet providers protect users by isolating private keys, validating networks, and requiring explicit confirmation for every transaction.

PART 3 - Activity: dApp for Week 4 Token



Not connected

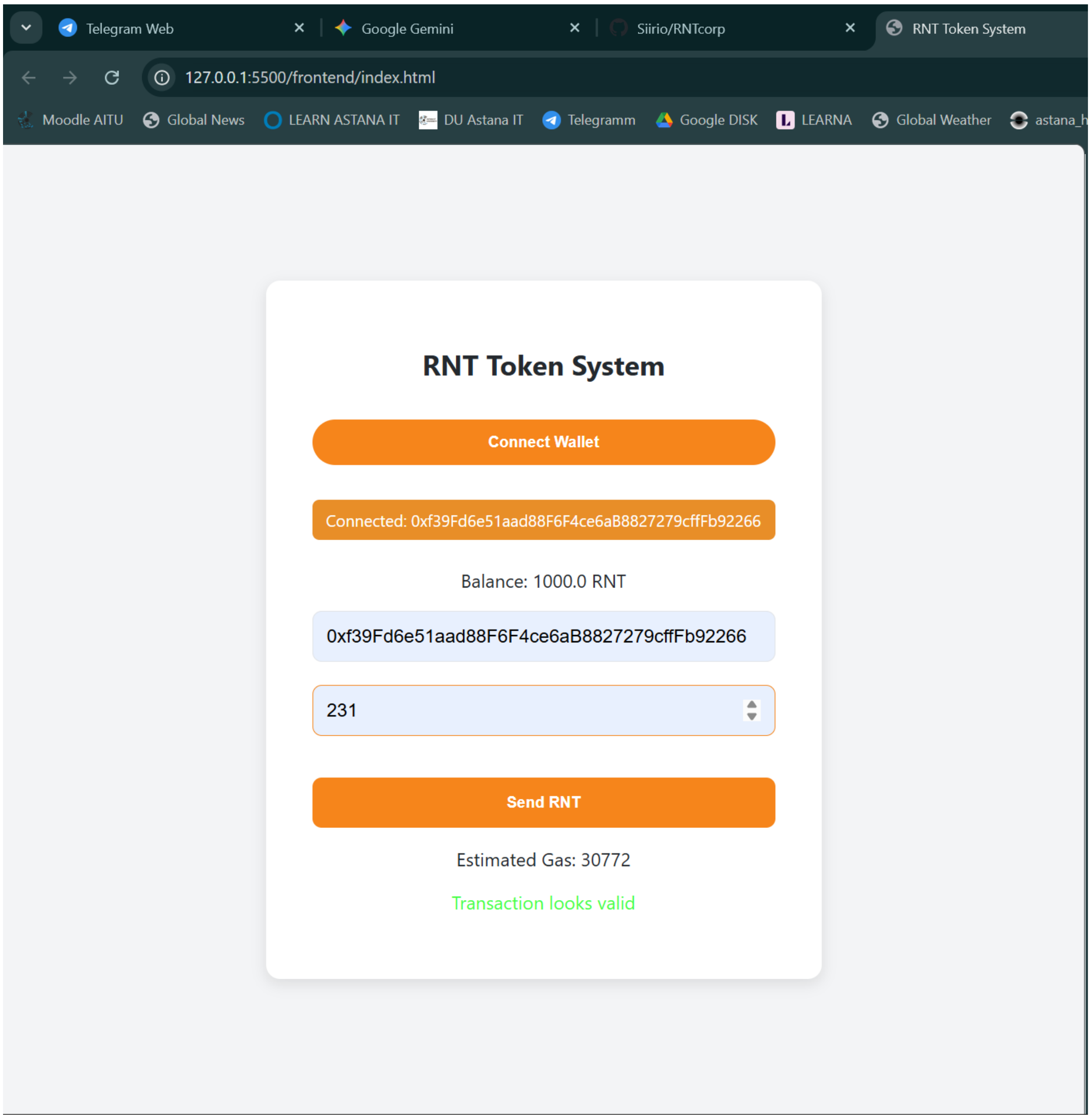

RNT Token System

Connected: 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266

Balance: 1000.0 RNT

Estimated Gas: -

Connected



Telegram Web
Google Gemini
Siirio/RNTcorp
RNT Token System

127.0.0.1:5500/frontend/index.html

Moodle AITU Global News LEARN ASTANA IT DU Astana IT Telegramm Google DISK LEARNA Global Weather astana_h

RNT Token System

Connect Wallet

Connected: 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266

Balance: 1000.0 RNT

0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266

231

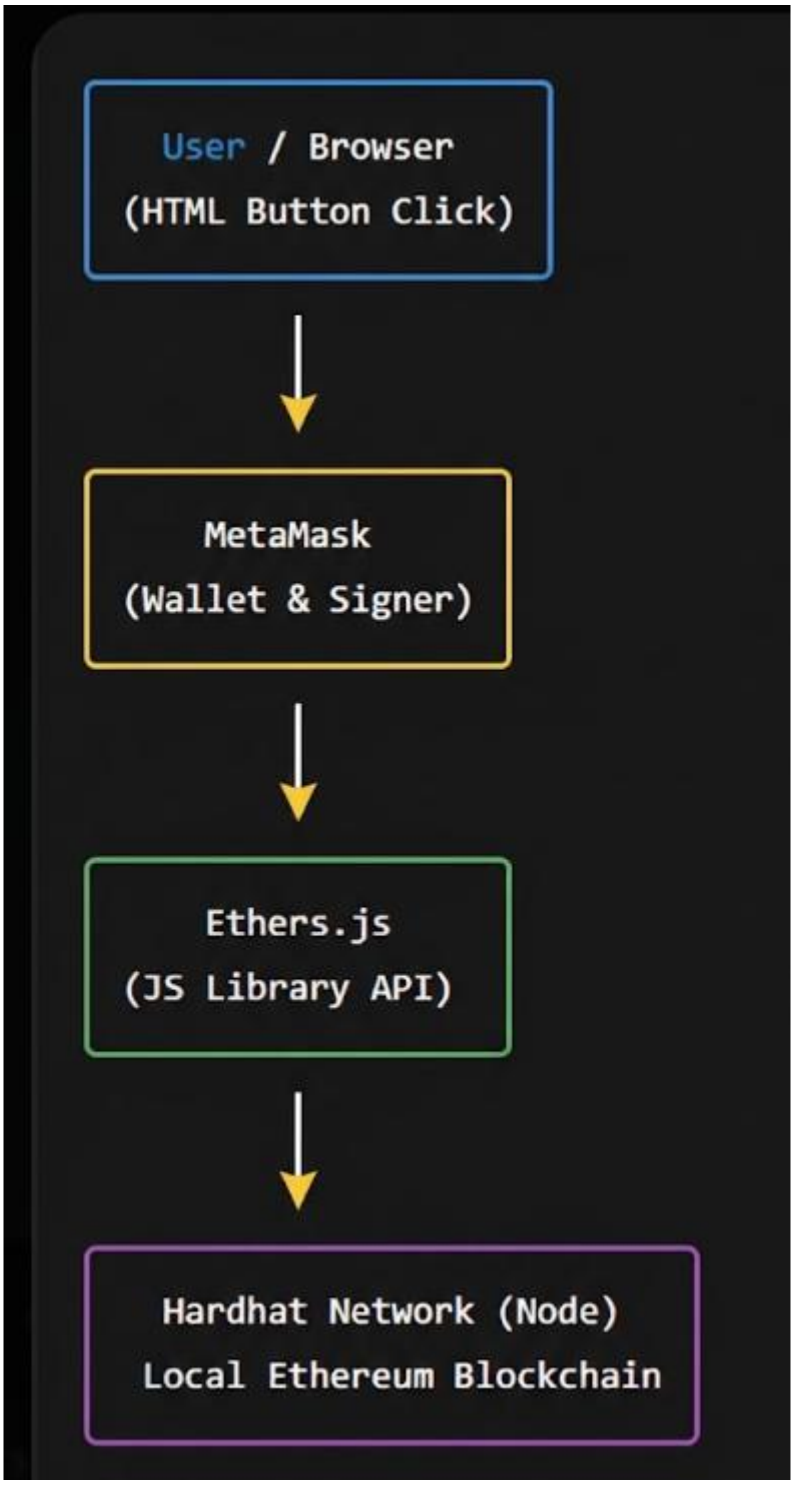
Send RNT

Estimated Gas: 30772

Transaction looks valid

Transaction validation and Gas estimation

UX reasoning provided as MD



This diagram shows the layered architecture of the decentralized application.
The user interacts with the system using a web browser. The user does not connect to the blockchain directly.

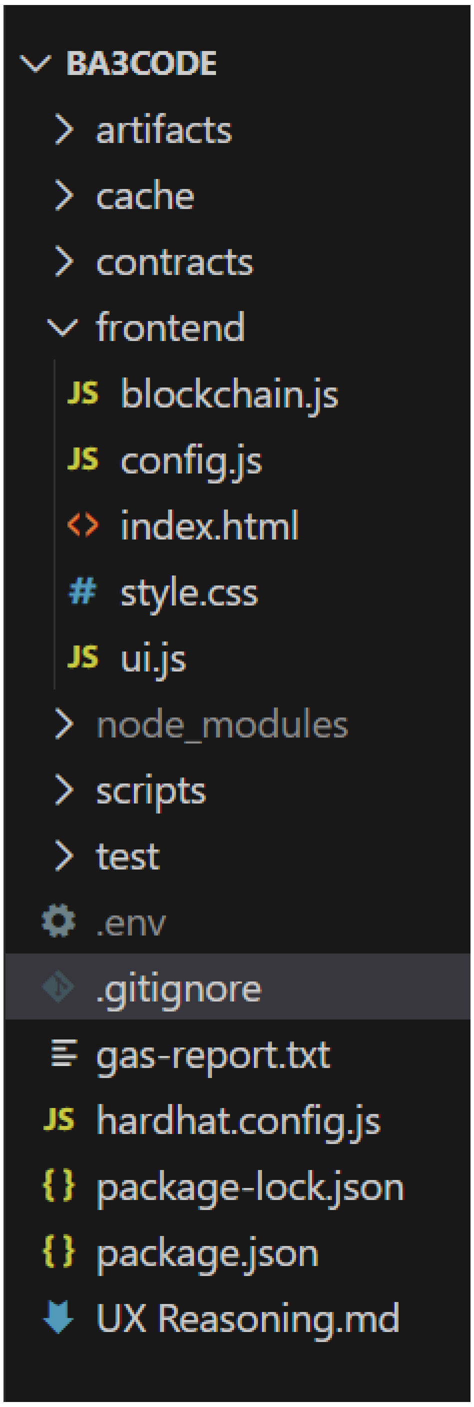
First, the user clicks a button in the browser.
Then MetaMask is used as a wallet to sign the transaction.
After that, Ethers.js sends the signed transaction to the Hardhat Network.
The Hardhat Network works as a local blockchain node and processes the transaction.

This architecture helps to separate the user interface from the blockchain logic.

PART 4 - In-Depth dApp Development & Frameworks

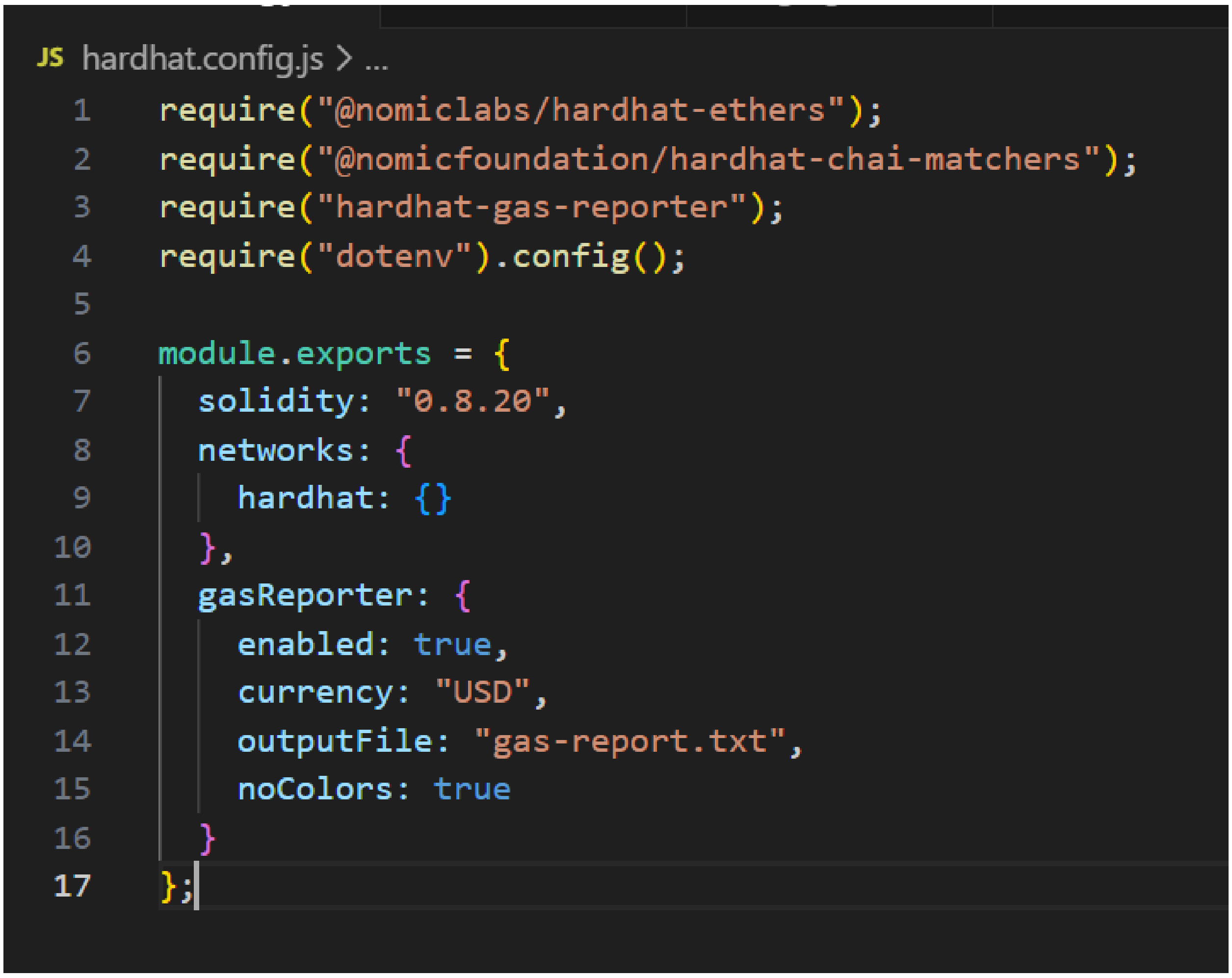
Section A - Hardhat/Truffle Setup

Our architecture:



```
BA3CODE
  artifacts
  cache
  contracts
  frontend
    blockchain.js
    config.js
    index.html
    style.css
    ui.js
  node_modules
  scripts
  test
  .env
  .gitignore
  gas-report.txt
  hardhat.config.js
  package-lock.json
  package.json
  UX Reasoning.md
```

Config:



hardhat.config.js > ...

```javascript
require("@nomiclabs/hardhat-ethers");
require("@nomicfoundation/hardhat-chai-matchers");
require("hardhat-gas-reporter");
require("dotenv").config();

module.exports = {
  solidity: "0.8.20",
  networks: {
    hardhat: {}
  },
  gasReporter: {
    enabled: true,
    currency: "USD",
    outputFile: "gas-report.txt",
    noColors: true
  }
};
```

Has gas-report pplugin

Has Chai mathcers

Dotenv to manage .env

To run the code we do:

Terminal 1(compile): npx hardhat node

Then open new terminal and run:

Terminal 2(deploy): npx hardhat run scripts/deploy.js --network hardhat

To run the tests:

Terminal(tests): npx hardhat test

Section B - ERC Standards Analysis

ERC-20 Conceptual Model

ERC-20 defines a standard interface for fungible tokens, ensuring compatibility across wallets and decentralized applications.

ERC-721 Conceptual Architecture

ERC-721 represents non-fungible tokens, where each token is unique and identified by a specific token ID.

How Interfaces Define Behavior

Interfaces in ERC standards define mandatory functions and events that ensure consistent token behavior across platforms.

Differences in State Management

ERC-20 manages balances using a mapping of addresses to token amounts, while ERC-721 tracks ownership of individual token IDs.

Security Assumptions

ERC standards assume correct implementation of access control and safe handling of approvals to prevent unauthorized transfers.

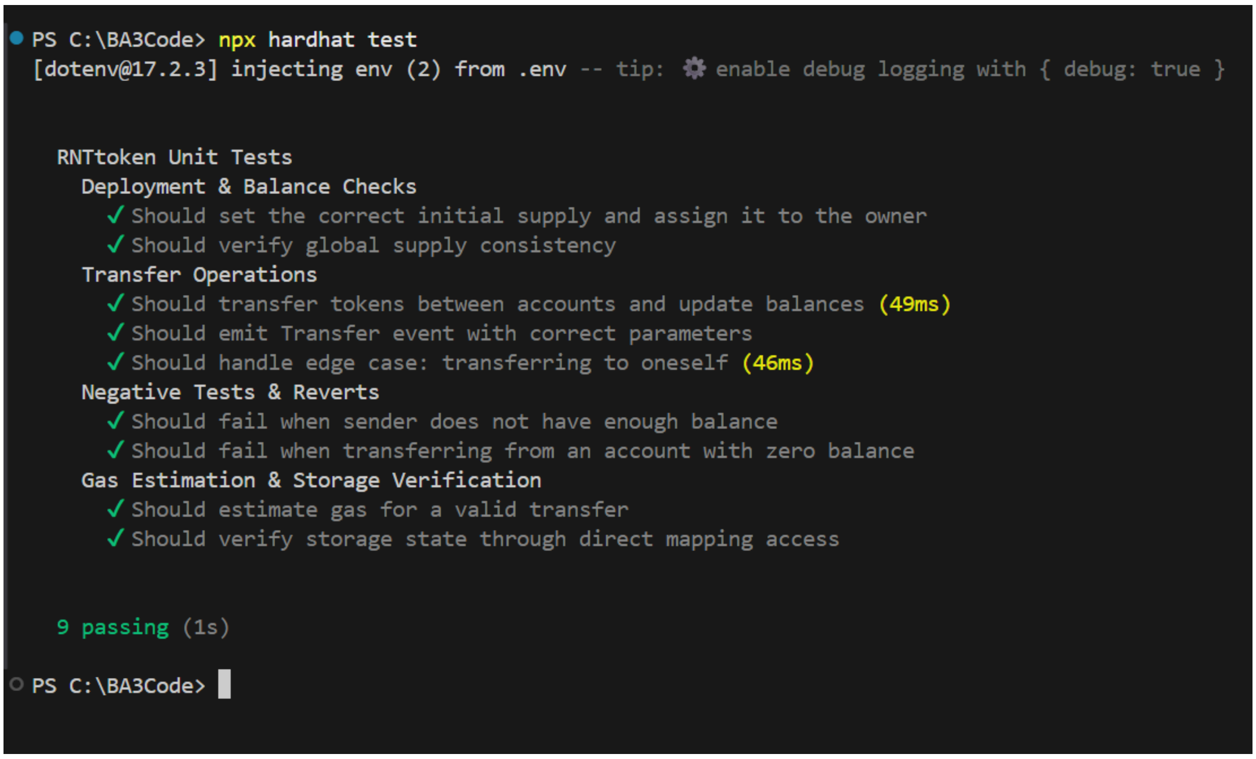
Gas Usage Differences

ERC-721 transactions typically consume more gas than ERC-20 due to additional metadata and ownership tracking.

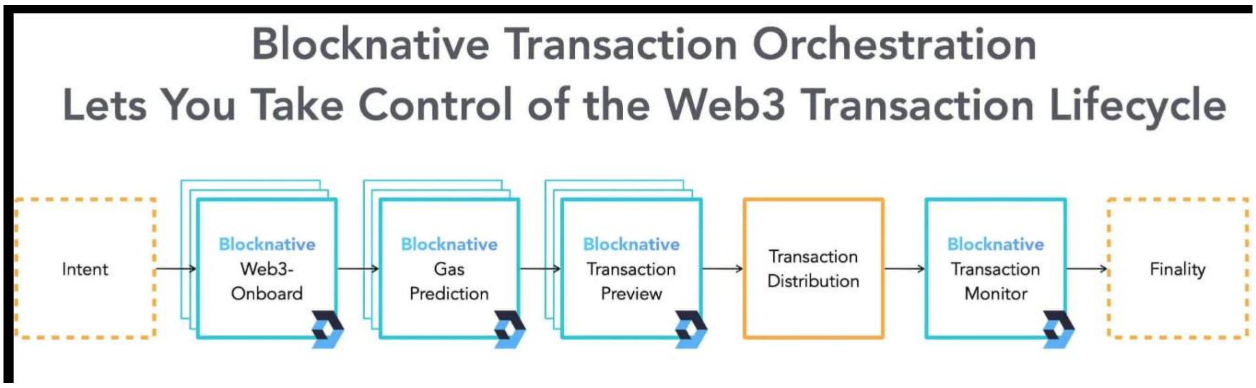
Metadata Extension Design Patterns

Metadata extensions allow tokens to store off-chain or on-chain descriptive data, improving usability and interoperability.

PART 5 - Lab Exercise: Testing the Token Contract



```
● PS C:\BA3Code> npx hardhat test
  [dotenv@17.2.3] injecting env (2) from .env -- tip: ⚙ enable debug logging with { debug: true }


    RNTtoken Unit Tests
      Deployment & Balance Checks
        ✓ Should set the correct initial supply and assign it to the owner
        ✓ Should verify global supply consistency
      Transfer Operations
        ✓ Should transfer tokens between accounts and update balances (49ms)
        ✓ Should emit Transfer event with correct parameters
        ✓ Should handle edge case: transferring to oneself (46ms)
      Negative Tests & Reverts
        ✓ Should fail when sender does not have enough balance
        ✓ Should fail when transferring from an account with zero balance
      Gas Estimation & Storage Verification
        ✓ Should estimate gas for a valid transfer
        ✓ Should verify storage state through direct mapping access


    9 passing (1s)

○ PS C:\BA3Code> █
```


Blocknative Transaction Orchestration
Lets You Take Control of the Web3 Transaction Lifecycle
Intent
Blocknative Web3-Onboard
Blocknative Gas Prediction
Blocknative Transaction Preview
Transaction Distribution
Blocknative Transaction Monitor
Finality

This sequence diagram illustrates the full lifecycle of a transaction from button click to blockchain confirmation.

PART 6 - Project Checkpoint

Literature Review

Decentralized Systems

Decentralized systems distribute control across multiple nodes, reducing single points of failure and increasing system resilience.
Research shows that decentralization improves transparency and fault tolerance but introduces challenges in coordination and scalability [1][2].

Ethereum Architecture Design

Ethereum extends blockchain functionality by introducing smart contracts executed on the Ethereum Virtual Machine (EVM).
Studies highlight Ethereum's account-based model as flexible for dApp development but note performance and gas-cost limitations [3][4].

JavaScript-Based Blockchain Interaction

JavaScript libraries such as Web3.js and Ethers.js abstract low-level JSON-RPC communication and enable browser-based interaction with blockchain networks. Academic and technical research emphasizes the importance of secure wallet integration and proper provider abstraction in frontend applications [5][6].

Modern Full-Stack Blockchain Development

Modern blockchain applications combine smart contracts, frontend frameworks, and wallet providers into a full-stack architecture.
Research indicates that this approach improves usability and adoption but requires careful security and architectural design [7][8].

References:

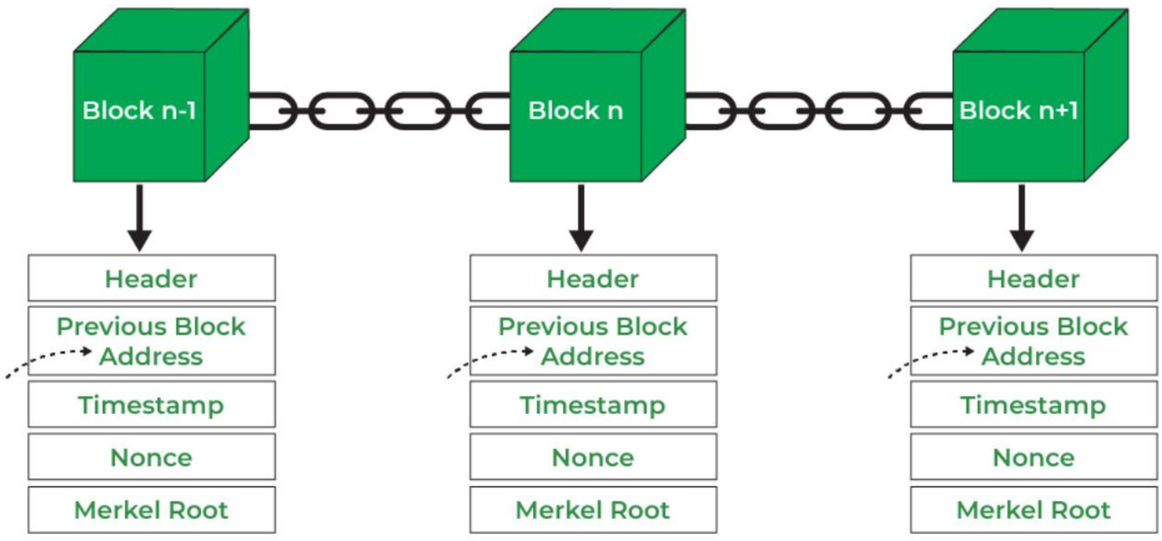
1. S. Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.
2. T. Dinh et al., BLOCKBENCH: A Framework for Analyzing Private Blockchains, IEEE, 2017.
3. V. Buterin, Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform, 2014.
4. G. Wood, Ethereum: A Secure Decentralised Generalised Transaction Ledger (Yellow Paper), 2014.
5. M. Di Angelo, G. Salzer, A Survey of Tools for Analyzing Ethereum Smart Contracts, IEEE, 2019.
6. L. Xu et al., Designing Blockchain-Based Applications, IEEE Internet Computing, 2019.
7. C. Dannen, Introducing Ethereum and Solidity, Apress, 2017.
8. P. Zhang et al., Blockchain-Based Decentralized Applications: A Survey, IEEE Access, 2020.

Methodological Design

PART 7 - Blockchain Architecture & Consensus

Section A - Deep Architecture Exploration

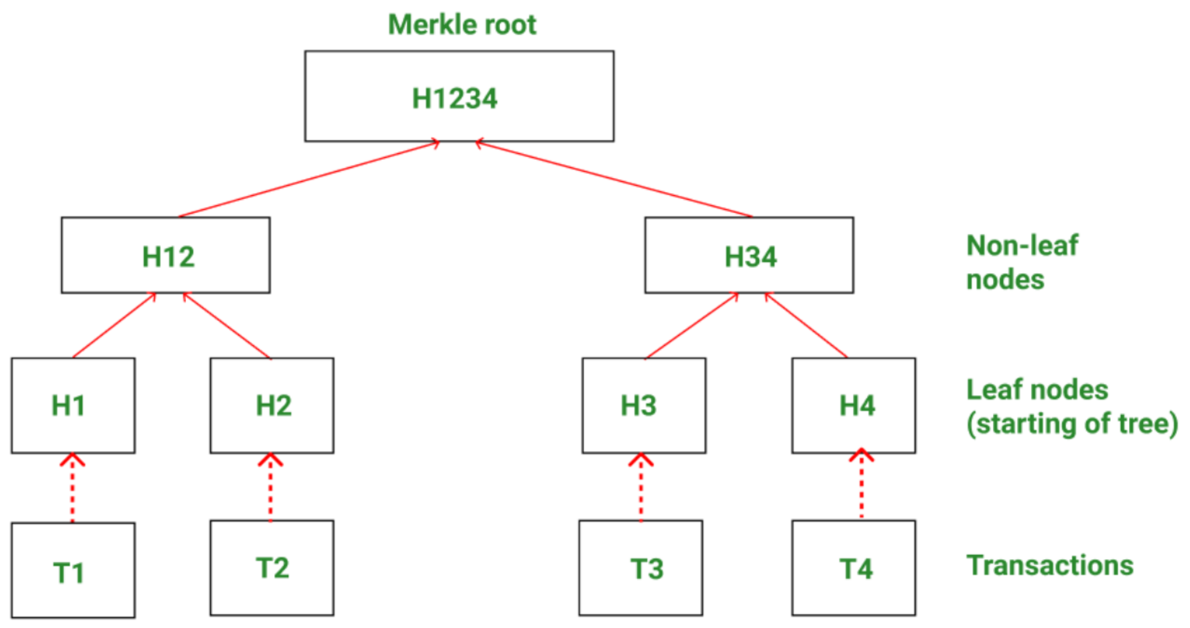
1. **Block Structure and Metadata**



A block is the atomic unit of a blockchain, consisting of a **Block Header** and a **Block Body**.

- **Block Header:** Contains the "fingerprint" of the block. Key fields include:
    - **Parent Hash:** The SHA-256 or Keccak-256 hash of the previous block’s header, creating the "chain."
    - **Nonce:** A 64-bit arbitrary number used in PoW to find a valid hash.
    - **Timestamp:** The Unix time when the block was mined/validated.
    - **Merkle Root:** The final hash representing all transactions in the block.
- **Block Body:** Contains the list of transactions. Each transaction includes sender, receiver, amount, signature, and nonce.

2. Merkle Trees (Proof and Verification)



A **Merkle Tree** (Binary Hash Tree) is used to efficiently summarize all transactions in a block.

- **Proof Generation:** Transactions are hashed in pairs. The hashes of those pairs are then hashed together, repeating until one root hash remains.
- **Verification:** Merkle Trees allow for **Simple Payment Verification (SPV)**. A light node can verify if a transaction is included in a block by only knowing its hash, the Merkle Root, and the "Merkle Path" (the hashes of sibling nodes), without needing the entire block data.

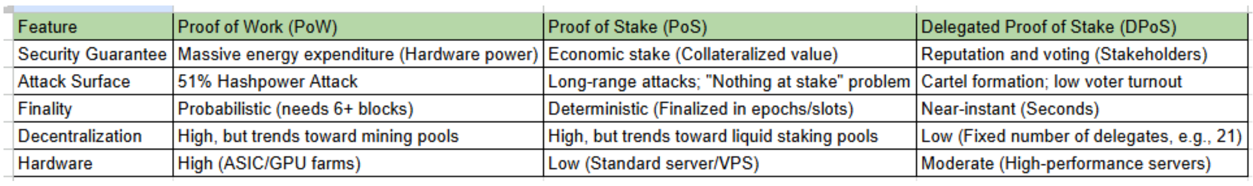
3. Networking and Transaction Propagation

- **Node Discovery:** Nodes use P2P protocols (like **ĐΞVp2p** in Ethereum) and Discovery protocols (e.g., **Kademlia**) to find peers.
- **Propagation Model:** Transactions are broadcasted using a **Gossip Protocol**. When you initiate a transfer in your dApp, it enters a node's **Mempool**. The node verifies the signature and then "gossips" it to its neighbors until the entire network is aware of the pending transaction.

4. Mining and Consensus Rules

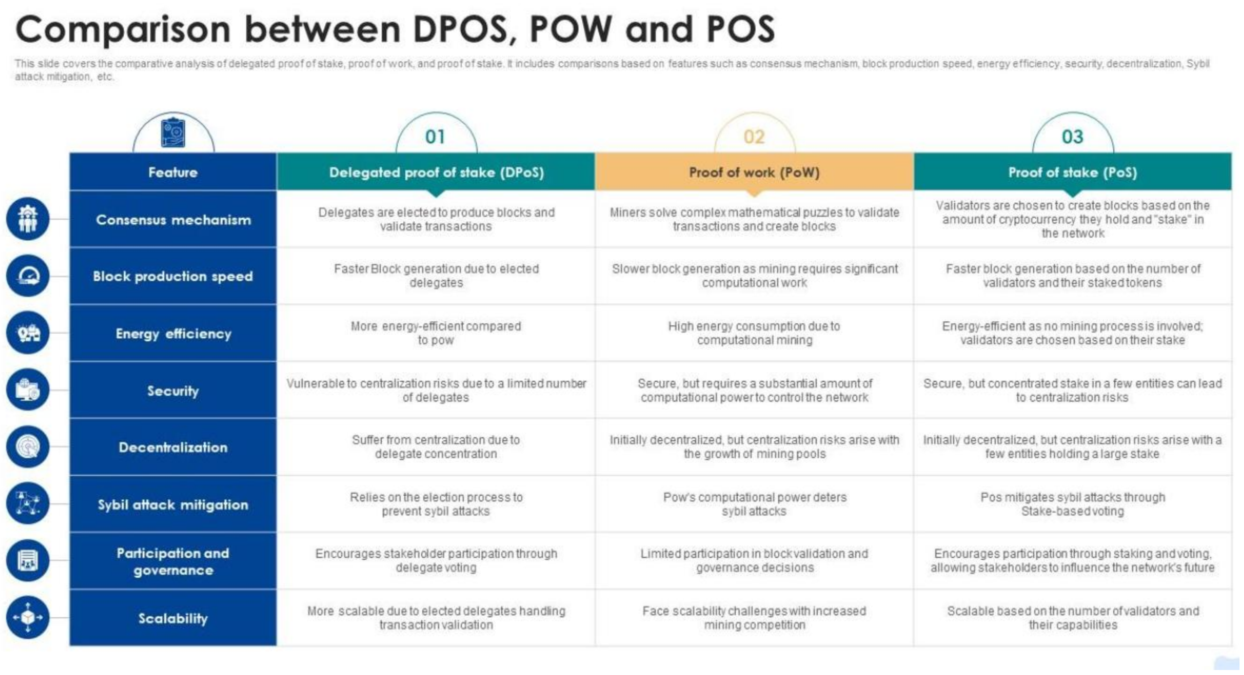
- **Mining (PoW):** Miners compete to solve a cryptographic puzzle ($Hash(Header) < Target$). This provides **probabilistic finality**, where a block becomes more "final" as more blocks are added on top of it.
- **Fork Choice Rules:**
  - **Nakamoto Consensus:** The "Longest Chain Rule."

- **LMD-GHOST (Latest Message Driven Greediest Heaviest Observed SubTree):** Used in Ethereum PoS to determine the head of the chain by looking at the weight of attestations (votes) from validators.
- **Gasper:** The combination of LMD-GHOST and **Casper FFG** (the finality gadget) to ensure blocks reach **economic finality** (cannot be reverted without burning validator stake).



Feature	Proof of Work (PoW)	Proof of Stake (PoS)	Delegated Proof of Stake (DPoS)
Security Guarantee	Massive energy expenditure (Hardware power)	Economic stake (Collateralized value)	Reputation and voting (Stakeholders)
Attack Surface	51% Hashpower Attack	Long-range attacks; "Nothing at stake" problem	Cartel formation; low voter turnout
Finality	Probabilistic (needs 6+ blocks)	Deterministic (Finalized in epochs/slots)	Near-instant (Seconds)
Decentralization	High, but trends toward mining pools	High, but trends toward liquid staking pools	Low (Fixed number of delegates, e.g., 21)
Hardware	High (ASIC/GPU farms)	Low (Standard server/VPS)	Moderate (High-performance servers)

Section B:



Comparison between DPOS, POW and POS

This slide covers the comparative analysis of delegated proof of stake, proof of work, and proof of stake. It includes comparisons based on features such as consensus mechanism, block production speed, energy efficiency, security, decentralization, Sybil attack mitigation, etc.

Feature	01 Delegated proof of stake (DPoS)	02 Proof of work (PoW)	03 Proof of stake (PoS)
Consensus mechanism	Delegates are elected to produce blocks and validate transactions	Miners solve complex mathematical puzzles to validate transactions and create blocks	Validators are chosen to create blocks based on the amount of cryptocurrency they hold and "stake" in the network
Block production speed	Faster Block generation due to elected delegates	Slower block generation as mining requires significant computational work	Faster block generation based on the number of validators and their staked tokens
Energy efficiency	More energy-efficient compared to pow	High energy consumption due to computational mining	Energy-efficient as no mining process is involved; validators are chosen based on their stake
Security	Vulnerable to centralization risks due to a limited number of delegates	Secure, but requires a substantial amount of computational power to control the network	Secure, but concentrated stake in a few entities can lead to centralization risks
Decentralization	Suffer from centralization due to delegate concentration	Initially decentralized, but centralization risks arise with the growth of mining pools	Initially decentralized, but centralization risks arise with a few entities holding a large stake
Sybil attack mitigation	Relies on the election process to prevent sybil attacks	Pow's computational power deters sybil attacks	Pos mitigates sybil attacks through Stake-based voting
Participation and governance	Encourages stakeholder participation through delegate voting	Limited participation in block validation and governance decisions	Encourages participation through staking and voting, allowing stakeholders to influence the network's future
Scalability	More scalable due to elected delegates handling transaction validation	Face scalability challenges with increased mining competition	Scalable based on the number of validators and their capabilities

This section compares three blockchain consensus mechanisms:
Proof of Work (PoW), Proof of Stake (PoS), and Delegated Proof of Stake (DPoS).

The analysis focuses on security guarantees, attack surfaces, finality, decentralization tradeoffs, economic incentives, and hardware requirements.

1. Proof of Work (PoW)

Examples: Bitcoin, Ethereum (pre-merge)

Security Guarantees

PoW is very secure because an attacker needs more than 50% of the total network hash power to control the blockchain.

Attack Surfaces

The main attack is a **51% attack**, but it is very expensive and difficult to perform on large networks.

Finality

PoW has **probabilistic finality**. Transactions become more secure over time but are never instantly final.

Decentralization Tradeoffs

In theory, PoW is decentralized.
In practice, mining pools and expensive hardware reduce decentralization.

Economic Incentives

Miners earn block rewards and transaction fees.
High electricity costs make mining expensive.

Hardware Requirements

PoW requires powerful hardware (ASICs or GPUs) and a lot of energy.

2. Proof of Stake (PoS)

Examples: Ethereum (post-merge), Cardano (Ouroboros)

Security Guarantees

Security is based on locked tokens (stake).
Validators can lose their stake if they behave dishonestly.

Attack Surfaces

Possible attacks include nothing-at-stake and long-range attacks, but modern PoS systems use slashing and checkpoints to reduce these risks.

Finality

PoS provides faster and stronger finality than PoW.
Some blocks become final after validator agreement.

Decentralization Tradeoffs

PoS is more accessible because no special hardware is needed.
However, users with more stake have more influence.

Economic Incentives

Validators earn rewards for honest behavior.
Dishonest actions result in financial penalties.

Hardware Requirements

PoS requires only normal servers or computers with stable internet.

3. Delegated Proof of Stake (DPoS)

Examples: EOS, Tron

Security Guarantees

Security depends on a small number of elected validators.
If validators collude, security can be reduced.

Attack Surfaces

Main risks include validator collusion and governance attacks.

Finality

DPoS has very fast finality, often within seconds.

Decentralization Tradeoffs

DPoS sacrifices decentralization for speed.
Only a limited number of validators produce blocks.

Economic Incentives

Validators are rewarded for block production.
Token holders vote and receive indirect benefits.

Hardware Requirements

Low hardware requirements.
Standard servers are enough.

Section C:

We made external file.