World Journal of Advanced Research and Reviews
eISSN: 2581-9615 CODEN (USA): WJARAI
Cross Ref DOI: 10.30574/wjarr
Journal homepage: https://wjarr.com/

(REVIEW ARTICLE)

Using blockchain platforms to build decentralized applications

Vitalii Pankin *

Senior Fullstack Engineer Antalya, Turkey.

World Journal of Advanced Research and Reviews, 2025, 27(03), 1102-1116

Publication history: Received on 09 August 2025; revised on 14 September 2025; accepted on 17 September 2025

Article DOI: https://doi.org/10.30574/wjarr.2025.27.3.3245

Abstract

This study aims to analyze the development and implementation of decentralized applications (DApps) on blockchain platforms. The research employs a comprehensive approach, examining architectural features of popular blockchain platforms, security aspects, and risk management strategies. A practical implementation of a supply chain management DApp on Ethereum is presented, detailing key development stages such as smart contract design, data storage integration, and user interface creation, alongside an analysis of associated technical, economic, and regulatory challenges. The study reveals that while DApps offer significant advantages in transparency, censorship resistance, and immutability of data, they encounter significant barriers in scalability due to limited transaction throughput, challenges in delivering user-friendly interfaces, and uncertainty in regulatory compliance across multiple jurisdictions. The research contributes to the field by providing an in-depth analysis of current DApp development practices, innovative approaches to identity and access management, and strategies for mitigating operational, financial, and reputational risks. The findings suggest that further interdisciplinary research and industry-academia collaboration are crucial for realizing the full potential of DApps in transforming various sectors of the digital economy.

Keywords: Blockchain; Decentralized Applications; Smart Contracts; Ethereum; Security; Scalability; Identity Management; Risk Mitigation; Supply Chain; Interdisciplinary Research.

1. Introduction

In the era of digital transformation, blockchain technology emerges as a revolutionary tool capable of fundamentally changing approaches to application development and functionality. Decentralized applications (DApps), based on blockchain platforms, represent a new paradigm in the software field, promising enhanced security, transparency, and resistance to manipulation.

The relevance of DApp research is driven by the growing interest in decentralized systems from various economic sectors. According to DappRadar, the DApp industry experienced significant growth in 2023, with unique active wallets (UAW) increasing by 124%, reaching an average of 4.2 million daily UAW by the end of the year [1]. This trend indicates an increasing need for a deep understanding of the technical aspects and potential of blockchain platforms for DApp creation.

* Corresponding author: Vitalii Pankin

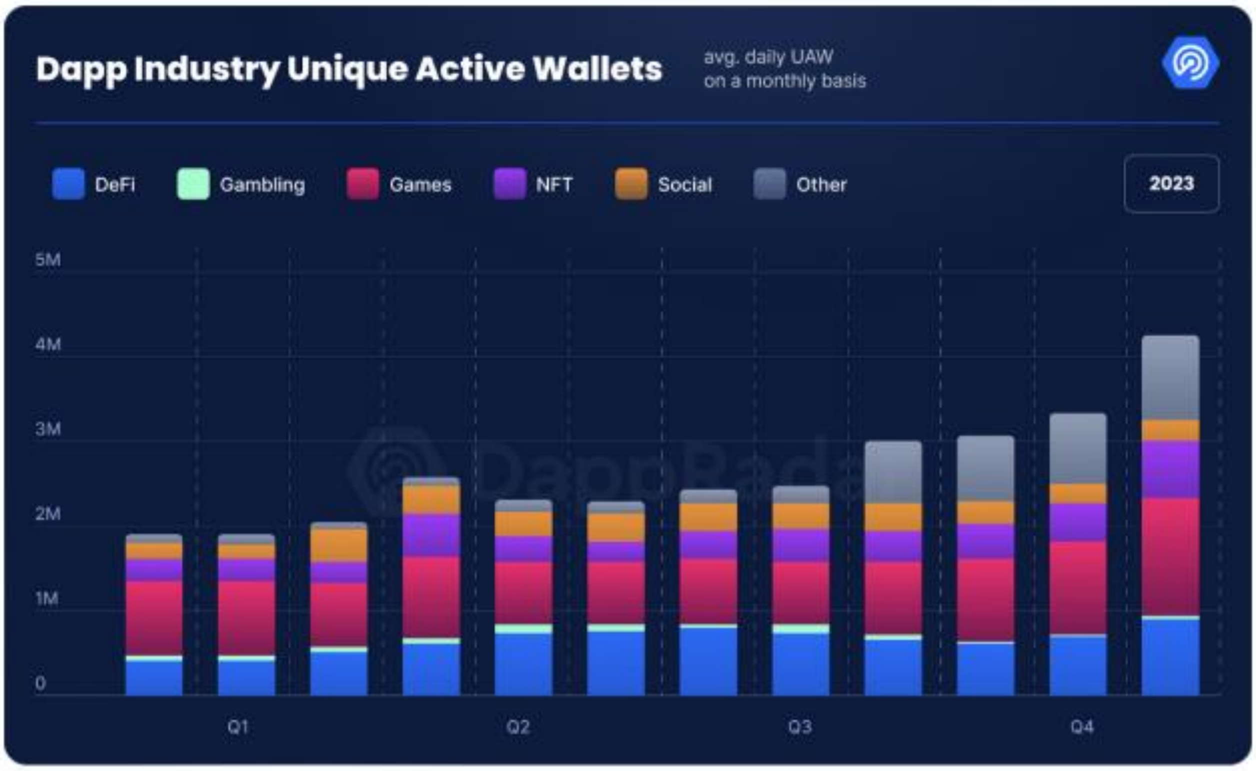

Figure 1 Growth of DApps in 2023 [1]

Additionally, according to a recent Deloitte study, 53% of organizations are exploring blockchain technology for supply chain integration, highlighting the growing interest in using blockchain to improve supply chain management. Another Gartner report forecasts that by 2023, 30% of manufacturing companies with revenues exceeding $5 billion will implement blockchain technology in their supply chain processes [2].

Despite evident progress, the DApp industry faces several significant challenges. Critical challenges persist in scalability, with many blockchain platforms unable to efficiently handle a large volume of transactions, energy efficiency issues particularly pronounced in networks employing Proof-of-Work mechanisms, and user experience hurdles stemming from complexity, slow transaction processing times, and high transaction fees, all of which significantly constrain widespread adoption [3].

In the context of these challenges and opportunities, the aim of this paper is to conduct a comprehensive analysis of the use of blockchain platforms for the creation of decentralized applications. The research focuses on:

- Studying the architectural features and technical aspects of DApp development on various blockchain platforms.
- Analyzing security issues and risk management strategies in the context of DApps.
- Demonstrating the practical implementation of a DApp on a selected blockchain platform with a detailed breakdown of the key development stages.

This approach will not only deepen the theoretical understanding of the technology but also provide practical insights for developers and researchers working on the creation and implementation of decentralized applications.

2. Architectural Features and Technical Aspects of DApp Development

Blockchain technology, initially developed as the foundation for the Bitcoin cryptocurrency, has evolved into a powerful platform for creating decentralized applications (DApps). These applications represent a new paradigm in software development, offering unique advantages in terms of security, transparency, and censorship resistance.

Blockchain platforms for DApps are characterized by a distributed architecture (Fig. 2), where data and computations are spread across a network of nodes rather than centralized on a single server. Key characteristics of such platforms include the use of cryptographic algorithms to ensure data integrity, consensus mechanisms to validate transactions, and smart contracts to automate business logic [4].

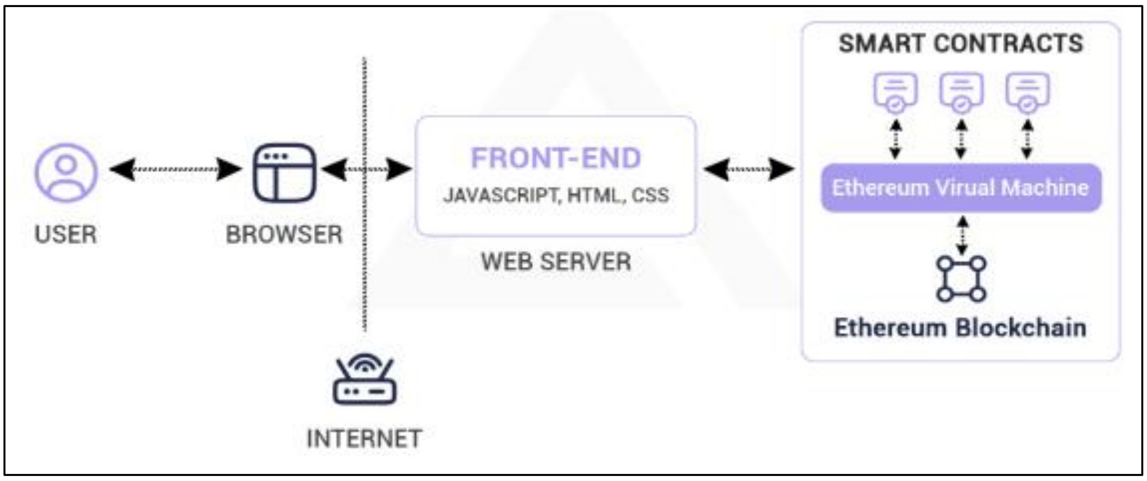

Figure 2 DApp Architecture

Developing DApps is a complex process that requires a deep understanding of the architectural features and technical aspects of blockchain platforms. Key blockchain platforms for DApps—Ethereum, Hyperledger Fabric, and EOS—offer different architectural solutions that define the approaches to creating and operating applications (Table 1).

Table 1 Key Blockchain Platforms for DApps

Characteristic	**Ethereum**	**Hyperledger Fabric**	**EOS**
Consensus Model	Proof-of-Stake (PoS)	Pluggable consensus	Delegated Proof-of-Stake (DPoS)
Smart Contracts	Solidity, Vyper	Go, Java, JavaScript	C++
Performance (TPS)	~15-30	>3000	>4000
Finality	~6 minutes	Instantaneous	0.5 seconds
Transaction Privacy	Public	Configurable	Public
Governance Model	On-chain governance	Modular, consortium	On-chain governance

The first architecture, Ethereum, is based on the Ethereum Virtual Machine (EVM), a Turing-complete computing environment functioning as a distributed computer with millions of executable nodes. The EVM executes smart contracts primarily written in Solidity, a high-level, statically-typed programming language specifically designed for the EVM, and Vyper, a Python-like language with enhanced security properties. Both languages compile into low-level EVM bytecode [4,7].

The process of executing a smart contract in the EVM includes the following key stages:

- Compilation: Solidity code is transformed into EVM bytecode using a compiler such as solc.
- Deployment: The contract bytecode is sent to the Ethereum network as a special transaction.
- Initialization: Upon deployment, the contract's constructor is executed, setting the initial state.
- Execution: Function calls to the contract trigger the execution of the corresponding bytecode in the EVM.

The EVM employs a stack-based architecture with a maximum depth of 1024 elements, supporting around 140 operations (opcodes), categorized into arithmetic, logical, cryptographic, environmental, memory management, and control-flow groups. This opcode design provides deterministic execution and predictable computational resource consumption for transactions. Each operation has a fixed gas cost, ensuring predictable computational expenses and preventing denial-of-service attacks.

Smart contracts in this architecture are self-executing, deterministic programs immutably stored on the blockchain. Their deterministic nature ensures identical execution outcomes across all nodes, enabling consensus on application state and making them ideal for implementing complex, verifiable business logic in decentralized applications. They define rules and automatically perform actions when predefined conditions are met, making them ideal for implementing complex business logic in decentralized applications (DApps). The deterministic execution of smart contracts in the EVM ensures that all network nodes reach the same state after processing transactions, which is critical for maintaining the integrity and consistency of the Ethereum blockchain.

Hyperledger Fabric, unlike Ethereum, offers a modular architecture with customizable components. A key feature of Fabric is the concept of channels, which allow the creation of isolated subnetworks within the main network. This solution is crucial for enterprise applications requiring a high degree of confidentiality. Fabric uses "chaincode" instead of smart contracts, which can be written in Go, Java, or JavaScript, simplifying integration with existing enterprise systems [4].

The EOS architecture, on the other hand, is based on WebAssembly (WASM), providing high performance and the ability to use multiple programming languages for smart contracts. EOS employs a Delegated Proof-of-Stake (DPoS) consensus mechanism, utilizing a voting system where token holders delegate validation authority to 21 block producers. This limited validator set significantly enhances transaction speed and throughput, offering sub-second finality but introduces concerns regarding decentralization, as validation authority is concentrated among fewer entities. This significantly enhances system performance but may compromise the level of decentralization [4].

Developing DApps on any of these platforms requires considering the specifics of distributed data storage. Traditional databases are unsuitable for fully decentralized applications. Instead, distributed storage systems like InterPlanetary File System (IPFS) or Swarm are used. These decentralized storage systems integrate seamlessly with blockchain platforms, ensuring data immutability, resilience to censorship, redundancy, and high availability. For instance, Ethereum often uses IPFS Content Identifiers (CIDs) stored within smart contracts to link immutable on-chain logic with off-chain stored data, thus effectively balancing storage costs and data persistence. For example, in Ethereum, an IPFS Content Identifier (CID) can be stored in a smart contract, creating a link between on-chain logic and off-chain data.

A major challenge in DApp development remains scalability. Ethereum, despite wide adoption, has limited throughput (~15-30 TPS), causing network congestion and higher transaction costs during peak usage. This significantly constrains the potential for creating high-load applications. To address Ethereum's scalability limitations, second-layer solutions, including Optimistic Rollups, zk-Rollups, and Validium, are actively being developed and adopted. These approaches significantly enhance throughput—potentially achieving thousands of TPS—by moving intensive computations and data storage off-chain, periodically settling transactions and ensuring security through cryptographic proofs verified on the main chain.

Transaction finality time is also a critical factor. In Ethereum, it is about 6 minutes, which is unacceptable for many applications requiring quick transaction confirmation. EOS, on the other hand, provides finality in 0.5 seconds, making it attractive for certain types of DApps.

Transaction costs, notably high Ethereum gas fees during congestion, can render many small or frequent transactions economically unviable, limiting DApp practicality and adoption. Therefore, Layer 2 solutions like Polygon and Arbitrum aim to reduce transaction costs and increase overall network efficiency.

Developing efficient DApps requires a comprehensive approach to architecture, taking into account the limitations and capabilities of the chosen blockchain platform. Data structures and update logic within smart contracts must carefully account for blockchain immutability, employing careful design, thorough testing, and strategies for future maintainability and scalability. Asynchronous operations and the need to work with events require special attention when developing the user interface and business logic of the application.

Optimizing smart contract code to minimize computational overhead and gas costs is critical on platforms with high transaction fees, requiring efficient coding patterns, minimized storage, and leveraging off-chain computations. Developers must balance functionality, security, and efficiency, considering gas and computational resource constraints [4,5].

3. Security and Risk Management in DApps

The security of DApps is a critical aspect of their development and operation. The unique architecture based on blockchain technologies presents new challenges in the field of cybersecurity and risk management.

Smart contracts, as the key component of DApps, are the main attack vector. The most common vulnerabilities include:

- Integer overflow and underflow: These vulnerabilities occur when numeric operations surpass predefined limits, due to fixed-size integer representation in the Ethereum Virtual Machine (EVM).
- Reentrancy: This allows an attacker to repeatedly call a function before its initial execution is complete.
- Uncontrolled external calls: Improper handling of external calls without proper validation can lead to unexpected contract behavior and expose the DApp to external exploitation.
- Denial of service due to gas exhaustion: Malicious depletion of the gas limit can block the execution of critical operations [6].

To prevent vulnerabilities in smart contracts, a comprehensive approach is applied, including various methods and tools. Static code analysis using specialized tools such as Mythril and Slither allows for the automatic detection of potential vulnerabilities at early development stages. Formal verification methods, using mathematical proofs to rigorously verify smart contract correctness, significantly enhance trustworthiness beyond automated analysis, especially for critical smart contracts handling sensitive assets. Adopting secure coding patterns like 'Checks-Effects-Interactions,' which enforces state updates before external calls, effectively mitigates vulnerabilities such as reentrancy attacks. The final step in ensuring security is conducting an independent code audit by blockchain security experts, which can identify complex vulnerabilities and issues that automated tools might miss.

Table 2 Comparison of Smart Contract Security Analysis Tools

Tool	Type of Analysis	Supported Platforms	Detection Effectiveness
Mythril	Symbolic execution	Ethereum, EOS	High for complex vulnerabilities
Slither	Static analysis	Ethereum	Fast, low false-positive rate
Oyente	Symbolic execution	Ethereum	Medium, focus on known vulnerabilities
Securify	Static analysis	Ethereum	High for pattern-based vulnerabilities

Cryptographic protocols play a fundamental role in ensuring the security of DApps. Key aspects include:

- Asymmetric cryptography: Used for creating digital signatures and identity management. Elliptic curves (e.g., secp256k1 in Bitcoin and Ethereum) provide high security with shorter key lengths compared to RSA.
- Hash functions: SHA-256 and Keccak (SHA-3) are used for creating unique identifiers for transactions and blocks, as well as in the mining process.
- Consensus protocols, including Proof-of-Work (PoW), Proof-of-Stake (PoS), and Byzantine Fault Tolerance (BFT), provide network security by preventing double-spending attacks and ensuring blockchain integrity through distributed consensus.
- Zero-Knowledge Proofs (ZKP): These allow for transaction verification without disclosing confidential information, which is critical for privacy-preserving DApps.

Innovative cryptographic solutions such as homomorphic encryption and quantum-resistant algorithms are currently under research and have the potential to significantly enhance the security of DApps in the future.

Alongside these cryptographic innovations, the field of identity and access management (IAM) is also evolving in the context of decentralized applications. The concept of Self-Sovereign Identity (SSI) redefines the approach to digital identity, providing users with full control over their identifiers without reliance on centralized authorities. Decentralized Identifiers (DID), developed under the W3C standard, provide a universal mechanism for creating and managing identifiers in distributed systems. Verifiable Credentials serve as digital equivalents of physical documents, enabling reliable verification of identity attributes in the digital space. Blockchain-based multi-factor authentication (MFA) leverages the potential of smart contracts to implement complex and secure authentication schemes adapted to the decentralized nature of DApps. Together, these innovations form a new paradigm of security and identity management in the world of decentralized applications, contributing to the creation of a more robust and trustworthy digital ecosystem.

Implementing an effective IAM strategy in DApps is a complex task that requires a careful balance between security, privacy, and usability. Integration with existing identification systems, such as OAuth, can facilitate the adoption of DApps; however, this solution requires careful analysis of potential risks and vulnerabilities that may arise from the interaction between decentralized and centralized systems.

Risk management in the context of DApps covers a wide range of aspects, each requiring special attention. Operational risks associated with possible infrastructure failures, smart contract code errors, and scalability issues can significantly impact the functionality and reliability of DApps. Financial risks, driven by the volatility of cryptocurrencies and potential economic attacks, pose additional challenges to the stability and resilience of decentralized ecosystems.

Regulatory risks, caused by the uncertainty of the legal status of DApps in various jurisdictions, require continuous monitoring and adaptation to changing regulatory requirements. Reputational risks, related to potential security incidents, can seriously undermine the trust of users and investors, which is critical for the long-term success of any DApp project.

To effectively manage these risks, a comprehensive approach is necessary. Routine independent security audits, penetration testing, and comprehensive stress-testing procedures facilitate early detection and mitigation of potential vulnerabilities, enhancing overall resilience. Implementing smart contract upgrade management mechanisms, such as proxy patterns, ensures flexibility in maintaining and improving the codebase without compromising the integrity of data and application logic.

The use of multi-signature wallets for managing critical assets enhances security levels and reduces risks associated with a single point of failure. Developing and regularly testing incident response plans enables prompt and effective responses to potential threats, minimizing their negative impact on DApp functionality and user trust [8].

Thus, security and risk management in DApps represent a complex task that requires the integration of advanced cryptographic methods, innovative identity management approaches, and thorough analysis of potential vulnerabilities. As the DApp ecosystem evolves, so do the methods for ensuring their security, opening new directions for research and development in the field of cybersecurity for decentralized systems.

4. Practical Implementation of a DApp on a Selected Blockchain Platform

To demonstrate the practical implementation of a decentralized application (DApp), we will choose the Ethereum platform, considering its wide adoption and developed ecosystem of development tools. This example will represent a decentralized supply chain management system, ensuring transparency and traceability of goods from the manufacturer to the end consumer.

Ethereum is chosen for the following reasons:

- Mature ecosystem and wide developer community support.
- Availability of advanced development and testing tools (Truffle, Hardhat, Remix).
- Support for Turing-complete smart contracts using the Solidity programming language.
- Integration with decentralized data storage (IPFS).
- Scalability through Layer 2 solutions [7].

The practical implementation is formed as a step-by-step development of the DApp:

4.1. Architecture Design

This foundational stage involves defining the application's high-level structure, including smart contracts, decentralized storage solutions (IPFS), the web-based user interface (React), and middleware layers (Web3.js) to facilitate blockchain interactions. This is the initial and very important step that lays the foundation for the entire project.

In this case, the components of the DApp include:

- Smart contracts (Ethereum): Implement business logic and store critical data.
- Decentralized storage (IPFS): Provides storage for large volumes of data off-chain.
- Web interface (React): Provides a user interface for interacting with the DApp.
- Connecting layer (Web3.js): Ensures communication between the web interface and the blockchain.

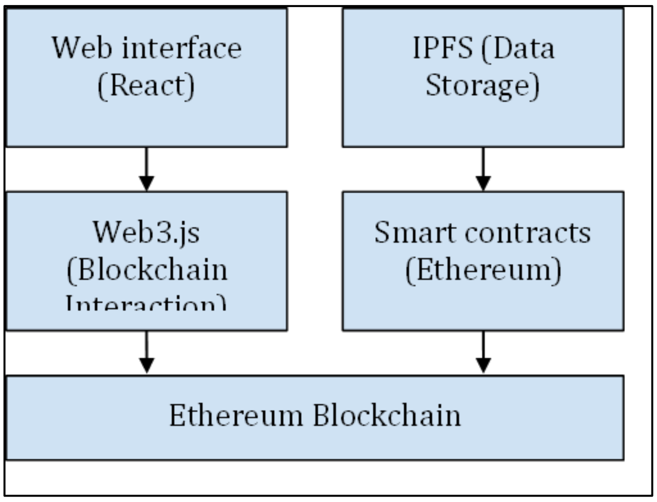

Figure 3 Component Interaction Diagram

4.2. Smart Contract Development

Smart Contract Development: We develop the primary smart contract (SupplyChain.sol) using Solidity (version 0.8.0), explicitly defining data structures, state transitions, events for product lifecycle tracking, and ensuring compliance with security best practices and standards. This contract defines the product structure, states, and main operations in the supply chain. We use Solidity version 0.8.0 to ensure security and leverage the latest language features.

```
pragma solidity ^0.8.0;

contract SupplyChain {

  struct Product {

    uint256 id;

    string ipfsHash;

    address currentOwner;

    ProductState state;

  }

  enum ProductState { Created, InTransit, Delivered }

  mapping(uint256 => Product) public products;

  uint256 public productCount;

  event ProductCreated(uint256 indexed id, string ipfsHash, address owner);

  event ProductShipped(uint256 indexed id, address from, address to);

  event ProductDelivered(uint256 indexed id, address recipient);
  function createProduct(string memory _ipfsHash) public {
    productCount++;
    products[productCount] = Product(productCount, _ipfsHash, msg.sender, ProductState.Created);
    emit ProductCreated(productCount, _ipfsHash, msg.sender);
  }
  function shipProduct(uint256 _id, address _to) public {
    require(products[_id].currentOwner == msg.sender, "Not the current owner");
    require(products[_id].state == ProductState.Created, "Product already shipped");
    products[_id].currentOwner = _to;
    products[_id].state = ProductState.InTransit;
    emit ProductShipped(_id, msg.sender, _to);
  }
  function receiveProduct(uint256 _id) public {
    require(products[_id].currentOwner == msg.sender, "Not the intended recipient");
    require(products[_id].state == ProductState.InTransit, "Product not in transit");
    products[_id].state = ProductState.Delivered;
    emit ProductDelivered(_id, msg.sender);
  }
  function getProduct(uint256 _id) public view returns (uint256, string memory, address, ProductState) {
    Product memory product = products[_id];
    return (product.id, product.ipfsHash, product.currentOwner, product.state);
  }
}
```

4.3. Smart Contract Deployment

Smart Contract Deployment: Leveraging the Hardhat development framework, we automate smart contract compilation, testing, and deployment processes, facilitating efficient development workflows and reliable contract deployments. We create a script deploy.js to automate the deployment process.

```
// scripts/deploy.js
const hre = require("hardhat");
async function main() {
  const SupplyChain = await hre.ethers.getContractFactory("SupplyChain");
  const supplyChain = await SupplyChain.deploy();

  await supplyChain.deployed();

  console.log("SupplyChain deployed to:", supplyChain.address);

}

main()

  .then(() => process.exit(0))

  .catch((error) => {

    console.error(error);

    process.exit(1);

  });
```

To deploy, run the command:

```
npx hardhat run scripts/deploy.js --network rinkeby
```

4.4. Integration with IPFS

Integration with IPFS: Utilizing the IPFS, we implement secure and reliable interactions with IPFS for off-chain storage, enabling large-scale decentralized data management while minimizing on-chain storage costs.

```
// ipfsUtils.js

import { create } from 'ipfs-http-client';

const ipfs = create({ host: 'ipfs.infura.io', port: 5001, protocol: 'https' });

export async function uploadToIPFS(data) {

  const result = await ipfs.add(JSON.stringify(data));

  return result.path;

}

export async function getFromIPFS(hash) {

  const stream = ipfs.cat(hash);

  let data = '';

  for await (const chunk of stream) {

    data += chunk.toString();

  }

  return JSON.parse(data);

}
```

4.5. Development of Web Interface

We create React components for interacting with the smart contract and IPFS, implementing functionalities for product creation, shipping, and receiving.

```
// ProductCreation.js

import React, { useState } from 'react';

import { ethers } from 'ethers';

import { uploadToIPFS } from './ipfsUtils';

function ProductCreation({ contract }) {

  const [productData, setProductData] = useState('');

  const createProduct = async () => {

    try {

      const ipfsHash = await uploadToIPFS(productData);

      await contract.createProduct(ipfsHash);

    } catch (error) {

      console.error('Error creating product:', error);

    }

  };

  return (

    <div>

      <textarea onChange={(e) => setProductData(e.target.value)} />

      <button onClick={createProduct}>Create Product</button>

    </div>

  );

}

// ProductTracking.js

function ProductTracking({ contract }) {

  const [productId, setProductId] = useState('');

  const [productInfo, setProductInfo] = useState(null);

  const trackProduct = async () => {

    try {
```

```
    const product = await contract.getProduct(productId);
    setProductInfo(product);
  } catch (error) {
    console.error('Error tracking product:', error);
  }
 };
 return (
  <div>
    <input onChange={(e) => setProductId(e.target.value)} />
    <button onClick={trackProduct}>Track Product</button>
    {productInfo && (
      <div>
        <p>ID: {productInfo[0].toString()}</p>
        <p>IPFS Hash: {productInfo[1]}</p>
        <p>Current Owner: {productInfo[2]}</p>
        <p>State: {['Created', 'InTransit', 'Delivered'][productInfo[3]]}</p>
      </div>
    )}
  </div>
 );
}
```

4.6. Implementation of Business Logic

We implement the main supply chain management functions in a separate module to centralize business logic and simplify its testing and maintenance.

```
// supplyChainLogic.js
import { ethers } from 'ethers';
import { uploadToIPFS, getFromIPFS } from './ipfsUtils';
export async function createProduct(contract, productData) {
 const ipfsHash = await uploadToIPFS(productData);
 const tx = await contract.createProduct(ipfsHash);
```

```
  await tx.wait();
  return tx.hash;
}
export async function shipProduct(contract, productId, toAddress) {
  const tx = await contract.shipProduct(productId, toAddress);
  await tx.wait();
  return tx.hash;
}
export async function receiveProduct(contract, productId) {
  const tx = await contract.receiveProduct(productId);
  await tx.wait();
  return tx.hash;
}
export async function trackProduct(contract, productId) {
  const productInfo = await contract.getProduct(productId);
  const detailedInfo = await getFromIPFS(productInfo[1]);
  return {
    id: productInfo[0].toString(),
    ipfsHash: productInfo[1],
    currentOwner: productInfo[2],
    state: ['Created', 'InTransit', 'Delivered'][productInfo[3]],
    details: detailedInfo
  };
}
```

4.7. Performance Optimization

We implement mechanisms to enhance DApp efficiency by caching data on the client side and using events to track changes.

```
// optimizationUtils.js
import { ethers } from 'ethers';
```

```
const eventCache = new Map();

export function setupEventListeners(contract, provider) {

  contract.on("ProductStateChanged", (id, newState) => {

    eventCache.set(id.toString(), { state: newState, timestamp: Date.now() });

  });

}

export async function getProductState(contract, productId) {

  const cachedEvent = eventCache.get(productId.toString());

  if (cachedEvent && Date.now() - cachedEvent.timestamp < 60000) {

    return cachedEvent.state;

  }

  const productInfo = await contract.getProduct(productId);

  return productInfo[3];

}
```

5. Testing and Debugging

We create comprehensive tests for all DApp components using Mocha and Chai for smart contract testing and Jest for React component testing.

```
// test/SupplyChain.test.js

const { expect } = require("chai");

const { ethers } = require("hardhat");

describe("SupplyChain", function() {

  let SupplyChain;

  let supplyChain;

  let owner;

  let addr1;

  let addr2;

  beforeEach(async function() {

    SupplyChain = await ethers.getContractFactory("SupplyChain");

    [owner, addr1, addr2] = await ethers.getSigners();

    supplyChain = await SupplyChain.deploy();
  });

  it("Should create a new product", async function() {

    await supplyChain.createProduct("QmTest");

    const product = await supplyChain.getProduct(1);

    expect(product[1]).to.equal("QmTest");

  });

  it("Should ship a product", async function() {

    await supplyChain.createProduct("QmTest");

    await supplyChain.shipProduct(1, addr1.address);

    const product = await supplyChain.getProduct(1);

    expect(product[2]).to.equal(addr1.address);

    expect(product[3]).to.equal(1); // InTransit state

  });

  it("Should receive a product", async function() {

    await supplyChain.createProduct("QmTest");

    await supplyChain.shipProduct(1, addr1.address);

    await supplyChain.connect(addr1).receiveProduct(1);

    const product = await supplyChain.getProduct(1);

    expect(product[3]).to.equal(2); // Delivered state

  });

});
```

This implementation provides a comprehensive overview of developing a DApp for supply chain management. It includes all necessary components: smart contracts, IPFS integration, web interface, business logic, optimization, and testing. Using this code along with appropriate Hardhat and React configurations, you can create a functional decentralized application.

6. Conclusion

Decentralized applications (DApps) built upon blockchain platforms represent a transformative paradigm in software development, enabling unprecedented transparency, data immutability, censorship resistance, and reliability compared to traditional centralized applications. This article has analyzed key aspects of the development and operation of DApps, from architectural features to practical implementation and risk management.

The detailed study of popular blockchain platforms, including Ethereum, Hyperledger Fabric, and EOS, highlighted distinct architectural characteristics, each with unique strengths in scalability, security models, transaction throughput, privacy options, and smart contract functionalities, alongside their inherent limitations influencing DApp development choices. Each platform offers its own approach to solving issues of scalability, security, and performance, which significantly influences the choice of the technology stack when developing DApps.

Comprehensive attention was given to the critical aspects of security and risk management in DApps, including smart contract vulnerabilities, innovative cryptographic approaches, decentralized identity management techniques, and robust strategies for effectively addressing operational, financial, regulatory, and reputational risks. Cryptographic solutions, modern approaches to identity and access management, as well as strategies for minimizing operational, financial, regulatory, and reputational risks were considered. These aspects are critically important for ensuring user trust and the sustainable development of the decentralized applications ecosystem.

The practical demonstration of a supply chain management DApp on Ethereum underscored the complexity inherent in DApp development, encompassing rigorous smart contract development and security verification, decentralized data storage integration using IPFS, performance optimization techniques, and the careful design of a user-friendly web interface to facilitate broad user adoption. This example highlighted the importance of careful architecture design, performance optimization, and security assurance at all levels of the application.

Despite significant progress in the field of DApps, unresolved challenges remain that require further research and innovation. Key challenges include enhancing blockchain network scalability through advanced layer-two and sharding solutions, significantly improving user experience via intuitive interfaces and reduced complexity, enabling robust cross-platform interoperability standards, and effectively navigating the continuously evolving and diverse global regulatory frameworks governing decentralized technologies.

Compliance with ethical standards

Disclosure of conflict of interest

No conflict of interest to be disclosed.

References

[1] Dapp Industry Report 2023. URL: https://dappradar.com/blog/dapp-industry-report-2023-defi-nft-web3-games

[2] Building Next-Gen Blockchain Solutions for Enterprises. URL: https://moldstud.com/articles/p-building-next-gen-blockchain-solutions-for-enterprises

[3] Zheng Z. et al. Blockchain challenges and opportunities: A survey //International journal of web and grid services. – 2018. – T. 14. – No. 4. – pp. 352-375.

[4] Xu X., Weber I., Staples M. Architecture for blockchain applications. – Cham: Springer, 2019. – P. 1-307.

[5] De Angelis S. et al. PBFT vs proof-of-authority: Applying the CAP theorem to permissioned blockchain //CEUR workshop proceedings. – CEUR-WS, 2018. – T. 2058.

[6] Atzei N., Bartoletti M., Cimoli T. A survey of attacks on ethereum smart contracts (sok) //Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6. – Springer Berlin Heidelberg, 2017. – pp. 164-186.

[7] Antonopoulos A. M., Wood G. Mastering ethereum: building smart contracts and dapps. – O'Reilly Media, 2018.

[8] Casino F., Dasaklis T. K., Patsakis C. A systematic literature review of blockchain-based applications: Current status, classification and open issues //Telematics and informatics. – 2019. – T. 36. – P. 55-81.