Introduction to

Smart contracts, Web 3.0

& DApps development (2024) v1.0

Dr. Enis KARAARSLAN
MSKU Department of Computer Engineering
Digital Technologies and Cyber Security Lab
https://linktr.ee/eniskaraarslan

INDEX

Dr. Enis KARAARSLAN

I used ChatGPT to enhance my previous slides. Napkin AI is used to draw some of the diagrams

These slides are made for “Decentralized systems & applications” class.

Dr. Enis Karaarslan made the formatting of the content, screenshots and code implementations

Free to distribute the content

Smart Contracts

ONE

Smart contracts …

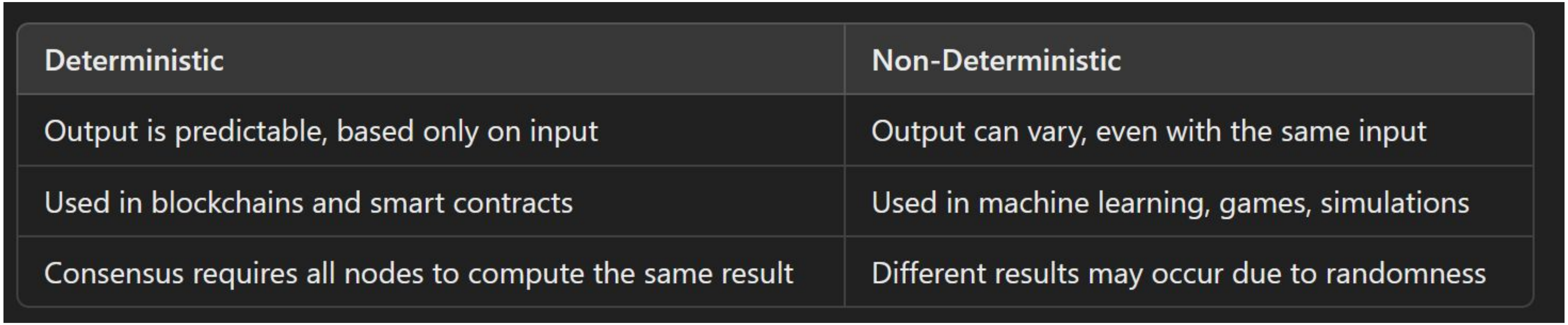
Deterministic Programming: Key Concept for Blockchain & Smart Contracts

- **Definition:** Deterministic programming is a programming paradigm where the output of a function or process is entirely determined by its inputs, with no randomness or hidden states.
- **Importance in Blockchain:** In decentralized networks like Ethereum, nodes must reach consensus on the state of the blockchain.

  This is only possible if every transaction and computation leads to the same result on every node.

  A function that adds two numbers, f(x, y) = x + y, will always return the same result given the same inputs. This is deterministic.

Deterministic vs. Non-Deterministic Programming



Deterministic	Non-Deterministic
Output is predictable, based only on input	Output can vary, even with the same input
Used in blockchains and smart contracts	Used in machine learning, games, simulations
Consensus requires all nodes to compute the same result	Different results may occur due to randomness

Why Deterministic Matters:

In smart contracts, every node in the network runs the same contract code and must arrive at the same result to ensure consistency across the decentralized ledger.

How Deterministic Programming Relates to Smart Contracts

- **Smart Contracts:** Programs that run on the blockchain, where the outcome of the contract must be consistent across all nodes.
- **Deterministic Requirement:** Smart contracts cannot have random elements or external states that could differ across nodes.
- **Real-World Example:** If a smart contract for a decentralized lottery uses randomness from a local machine, the result would differ on each node, breaking consensus.

```
function calculateSum(uint256 x, uint256 y) public pure returns (uint256) {
    return x + y;  // Deterministic behavior: same input -> same output
}
```

Avoiding Non-Determinism in Smart Contracts

- **Time-Based Functions:** Using block.timestamp for randomness can be problematic.
- **Accessing External APIs:** Data from outside the blockchain (via oracles) can differ between nodes.
- **Machine-Specific Variables:** Variables that depend on the local environment, such as msg.sender or msg.value, should be used cautiously.

Solution: Always use on-chain or deterministic sources of data and avoid any code that could introduce inconsistencies across nodes.

Ensuring Determinism in Solidity

- **Pure Functions:** Functions that have no side effects and whose output depends only on the input.
- **No Randomness:** Use deterministic mechanisms, such as block hashes, but ensure they don’t compromise security.
- **State Changes and Consensus:** Ensure that all state changes are deterministic and do not rely on external data that can change between nodes.

```
function getBlockHash(uint256 blockNumber) public view returns (bytes32) {
    return blockhash(blockNumber);  // Deterministic: all nodes agree on
the same blockhash
}
```

Trust and Security through Determinism

- **Verifiability:** Anyone can check the correctness of a smart contract by knowing that its behavior will be the same on all nodes.
- **Security:** By ensuring that smart contracts are deterministic, developers avoid vulnerabilities caused by different outcomes.
- **Finality:** Deterministic smart contracts provide predictable results, ensuring finality in blockchain transactions.

Key Takeaway: Deterministic behavior is at the core of blockchain's promise of trustlessness and decentralization.

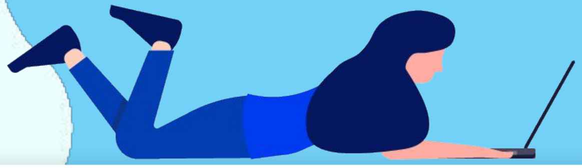
Determinism in Smart Contract Best Practices

- Follow Solidity’s Best Practices:
  - Use pure and view functions where applicable.
  - Avoid relying on block timestamps for critical logic.
  - Ensure contract logic is consistent across all nodes by using on-chain data sources.
- Gas Efficiency and Determinism:
  - Deterministic functions tend to be more gas-efficient as they avoid external data calls and complicated logic.

Summary: The Role of Deterministic Programming in Smart Contracts

- **Ensures Consensus:** All nodes in the blockchain must arrive at the same result.
- **Prevents Bugs:** Avoids issues caused by unpredictable behavior.
- **Enhances Security:** No room for different outcomes, ensuring the integrity of smart contracts.

Final Point: Without deterministic programming, blockchain and smart contracts could not function as a reliable, trustless system.



Smart Contracts

Smart Contracts

TWO

Introduction to Smart Contracts

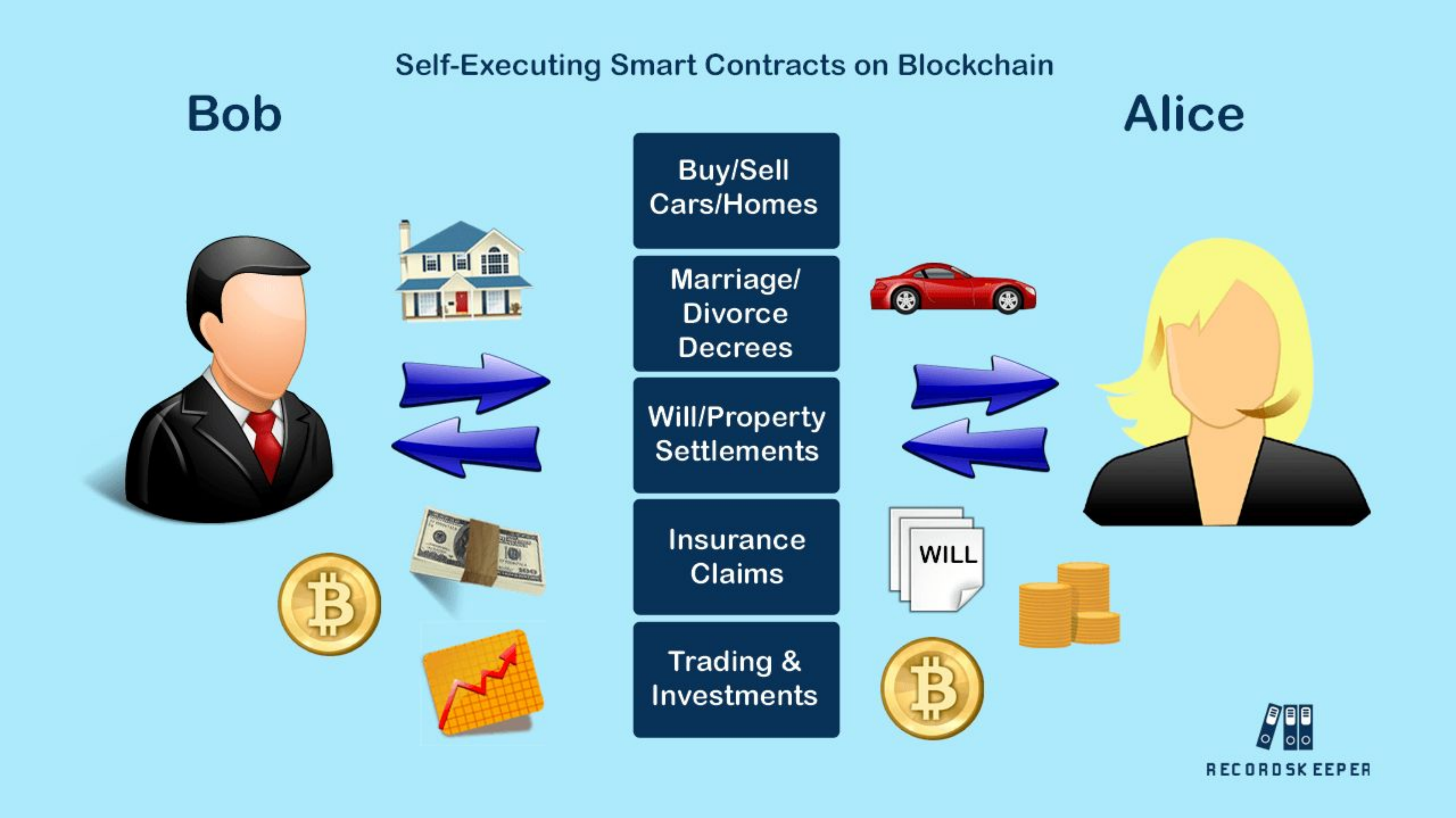
What are Smart Contracts?

- Smart contracts are self-executing contracts with the terms of the agreement directly written into code.
- Operate on decentralized networks, typically blockchain (e.g., Ethereum).

Smart Contract to Automate …

Assume
Bob & Alice
Decides
to Divorce

and there is
“prenuptial
agreement”



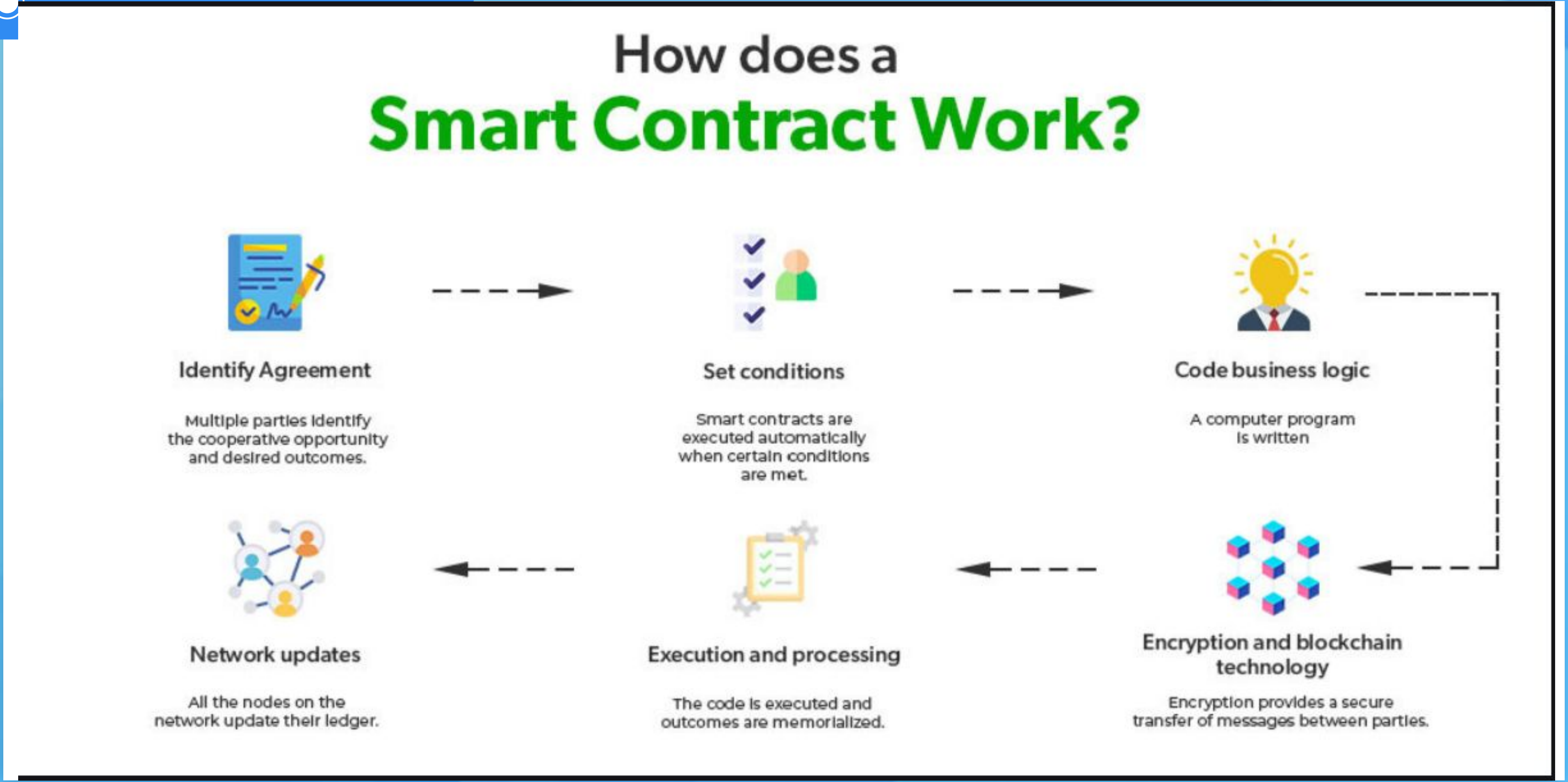
How Smart Contracts Work

Key Features:

- Autonomous: No need for intermediaries
- Immutable: Once deployed, they cannot be altered.
- Transparent: Anyone can verify the contract’s code.

Execution Example:

"If X happens, then execute Y automatically."



How does a Smart Contract Work?

Identify Agreement

Multiple parties identify the cooperative opportunity and desired outcomes.

Set conditions

Smart contracts are executed automatically when certain conditions are met.

Code business logic

A computer program is written

Encryption and blockchain technology

Encryption provides a secure transfer of messages between parties.

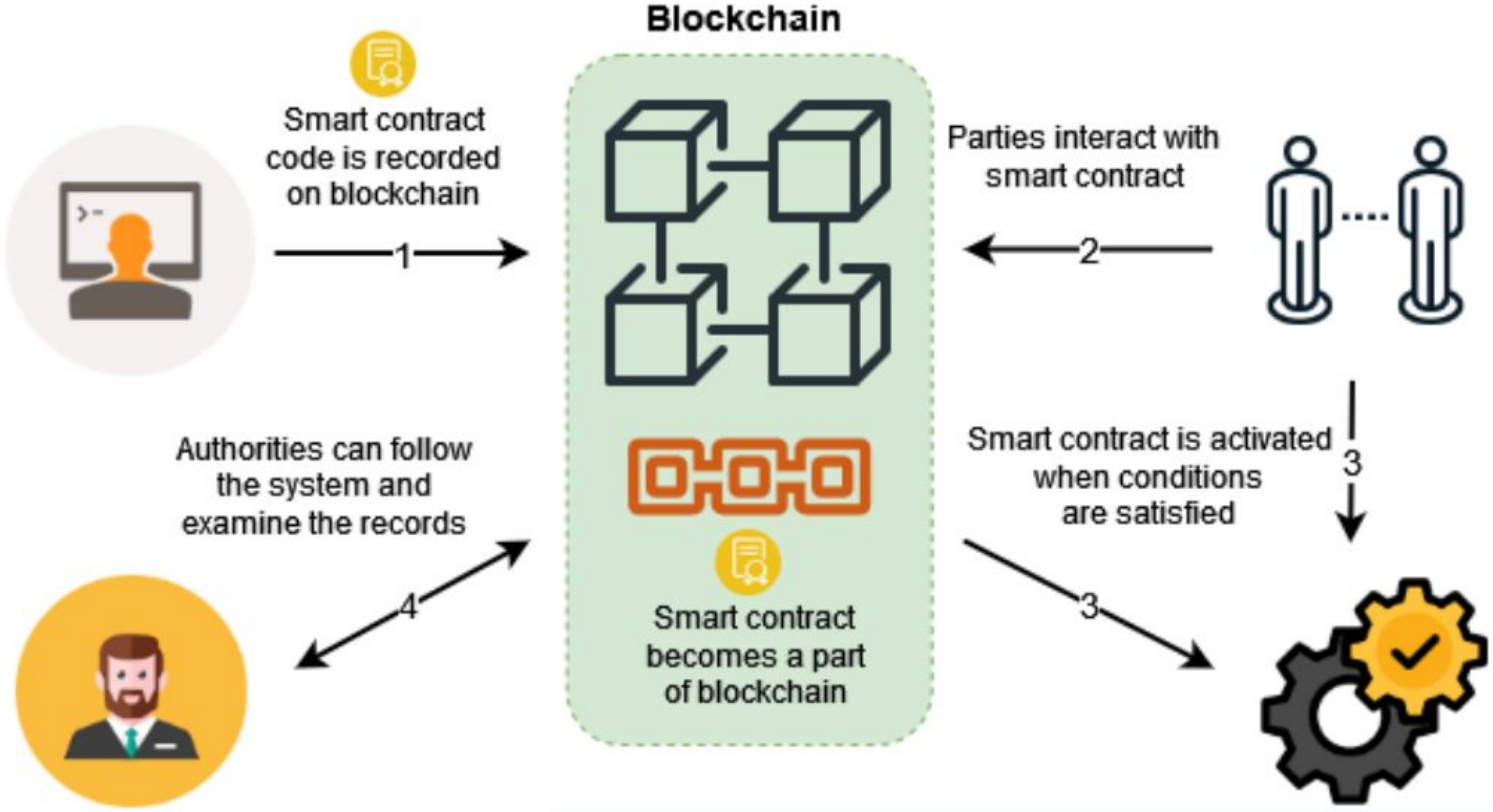
Execution and processing

The code is executed and outcomes are memorialized.

Network updates

All the nodes on the network update their ledger.

Operation of Smart Contracts [2]



Basic Structure of a Smart Contract

- Components:
    - ○ Functions
    - ○ Events
    - ○ State variables
    - ○ (store data)

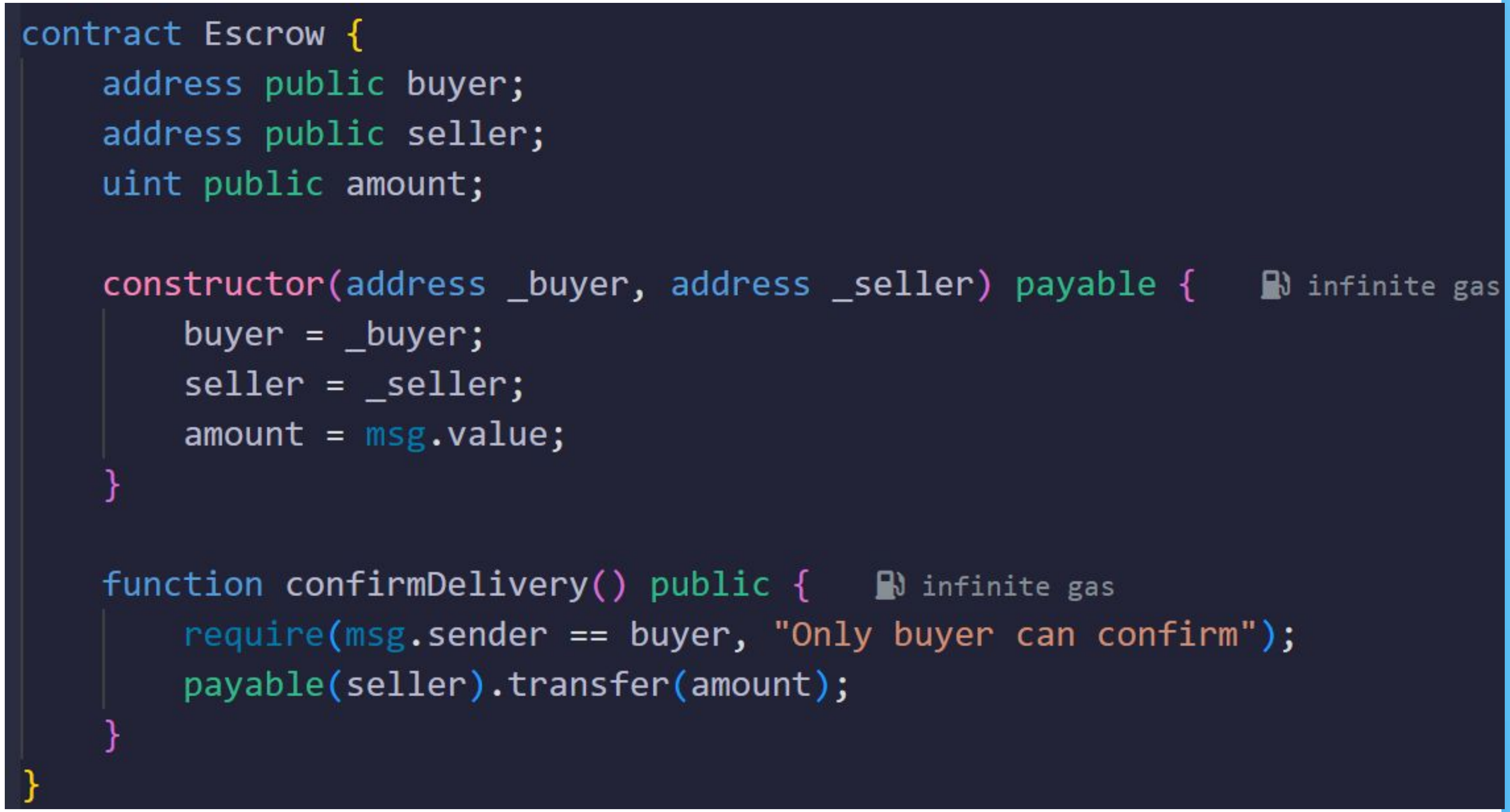
```
pragma solidity ^0.8.0;

contract SimpleContract {
    uint public balance;

    function deposit(uint amount) public {
        balance += amount;
    }

    function withdraw(uint amount) public {
        require(balance >= amount, "Insufficient balance");
        balance -= amount;
    }
}
```

Example Use Case: Escrow Service

- Problem:

  Buyer and seller don’t trust each other.
- Smart Contract Solution:

  Funds are locked in the contract until the buyer confirms receipt of goods.



```
contract Escrow {
    address public buyer;
    address public seller;
    uint public amount;

    constructor(address _buyer, address _seller) payable {    infinite gas
        buyer = _buyer;
        seller = _seller;
        amount = msg.value;
    }

    function confirmDelivery() public {    infinite gas
        require(msg.sender == buyer, "Only buyer can confirm");
        payable(seller).transfer(amount);
    }
}
```

Benefits of Smart Contracts

- Efficiency: Automated execution reduces delays.
- Security: Blockchain ensures data integrity.
- Cost-effective: No need for third-party intermediaries.

Challenges of Smart Contracts

- **Coding Bugs:** If a contract has a bug, it can lead to significant losses.
- **Legal Uncertainty:** Lack of regulation in some jurisdictions.
- **Immutability Issues:** Mistakes can’t be corrected after deployment.

Section conclusion

- Smart contracts are revolutionizing industries by increasing transparency, security, and efficiency. Such as:
    - Decentralized Finance (DeFi): Smart contracts are used for lending, borrowing, and trading without banks.
    - Supply Chain Management: Track goods from production to delivery.
- However, they must be carefully written to avoid bugs and exploitations. For a reference, see [1]

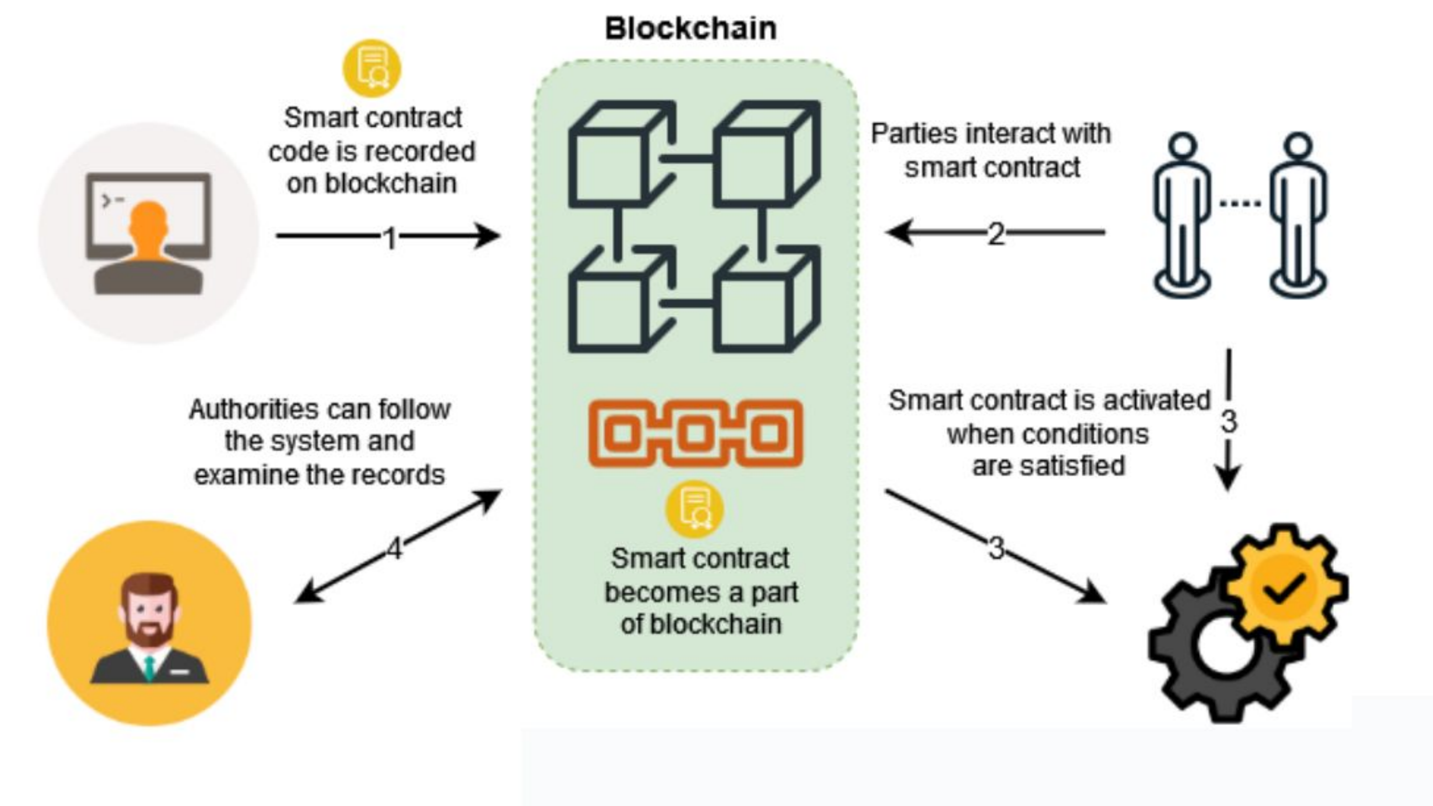
Development

THREE

Development

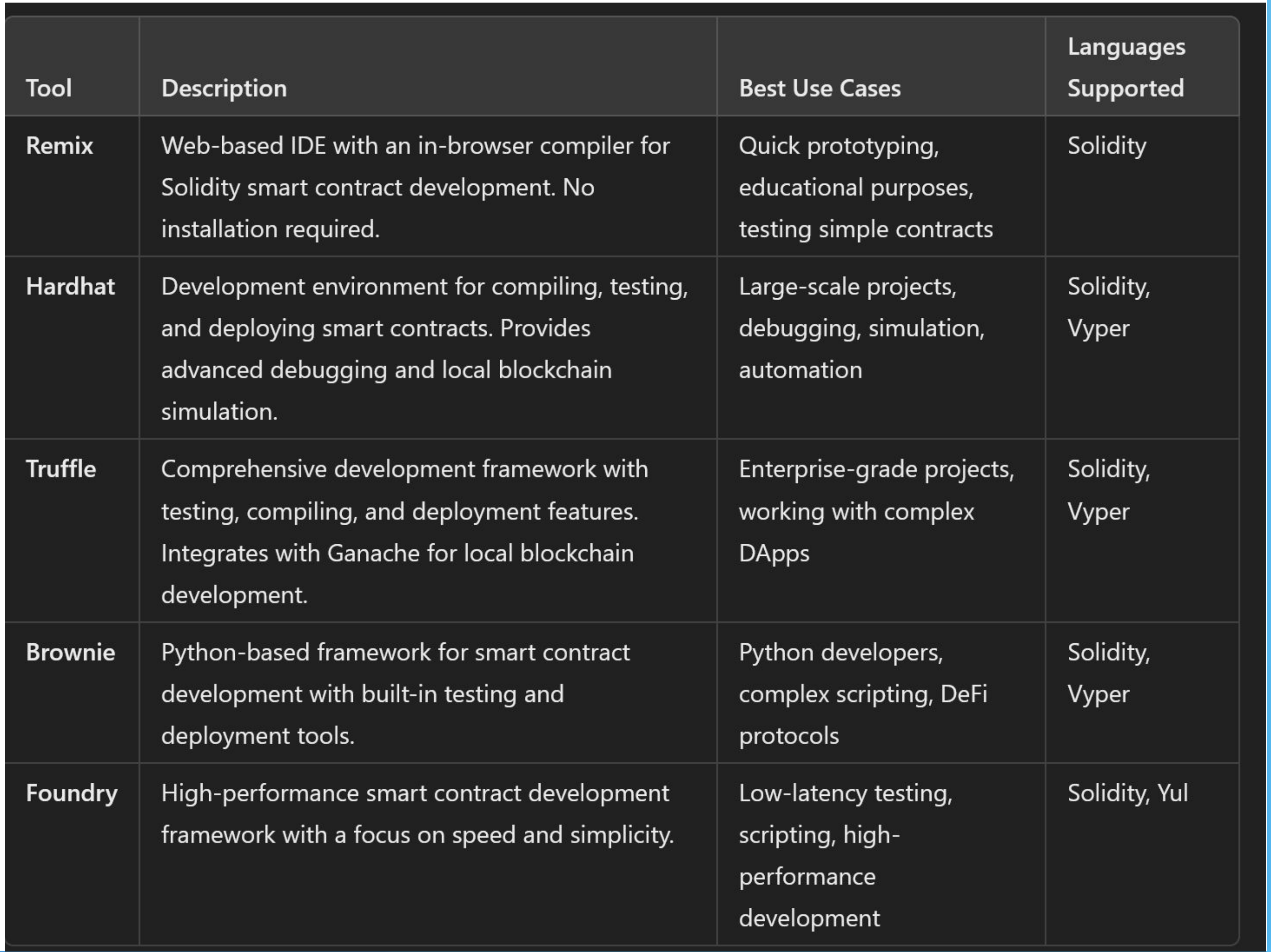
Development

- Write the code
- Deploy it on a test network
- Test and debug



Development Environments

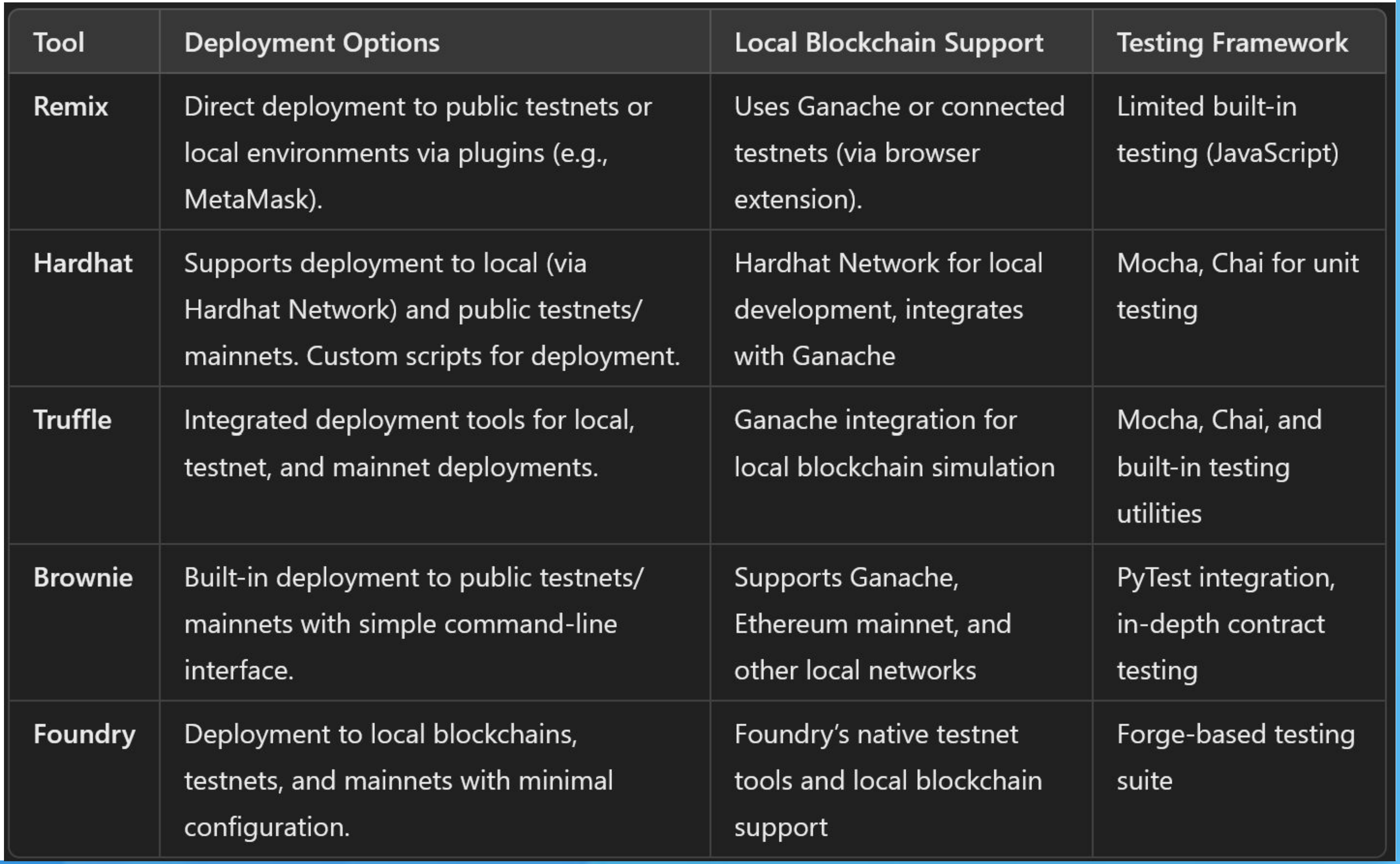
Development Environments for Smart Contracts - Overview



Tool	Description	Best Use Cases	Languages Supported
Remix	Web-based IDE with an in-browser compiler for Solidity smart contract development. No installation required.	Quick prototyping, educational purposes, testing simple contracts	Solidity
Hardhat	Development environment for compiling, testing, and deploying smart contracts. Provides advanced debugging and local blockchain simulation.	Large-scale projects, debugging, simulation, automation	Solidity, Vyper
Truffle	Comprehensive development framework with testing, compiling, and deployment features. Integrates with Ganache for local blockchain development.	Enterprise-grade projects, working with complex DApps	Solidity, Vyper
Brownie	Python-based framework for smart contract development with built-in testing and deployment tools.	Python developers, complex scripting, DeFi protocols	Solidity, Vyper
Foundry	High-performance smart contract development framework with a focus on speed and simplicity.	Low-latency testing, scripting, high-performance development	Solidity, Yul

Features Comparison

Development Environments for Smart Contracts Features

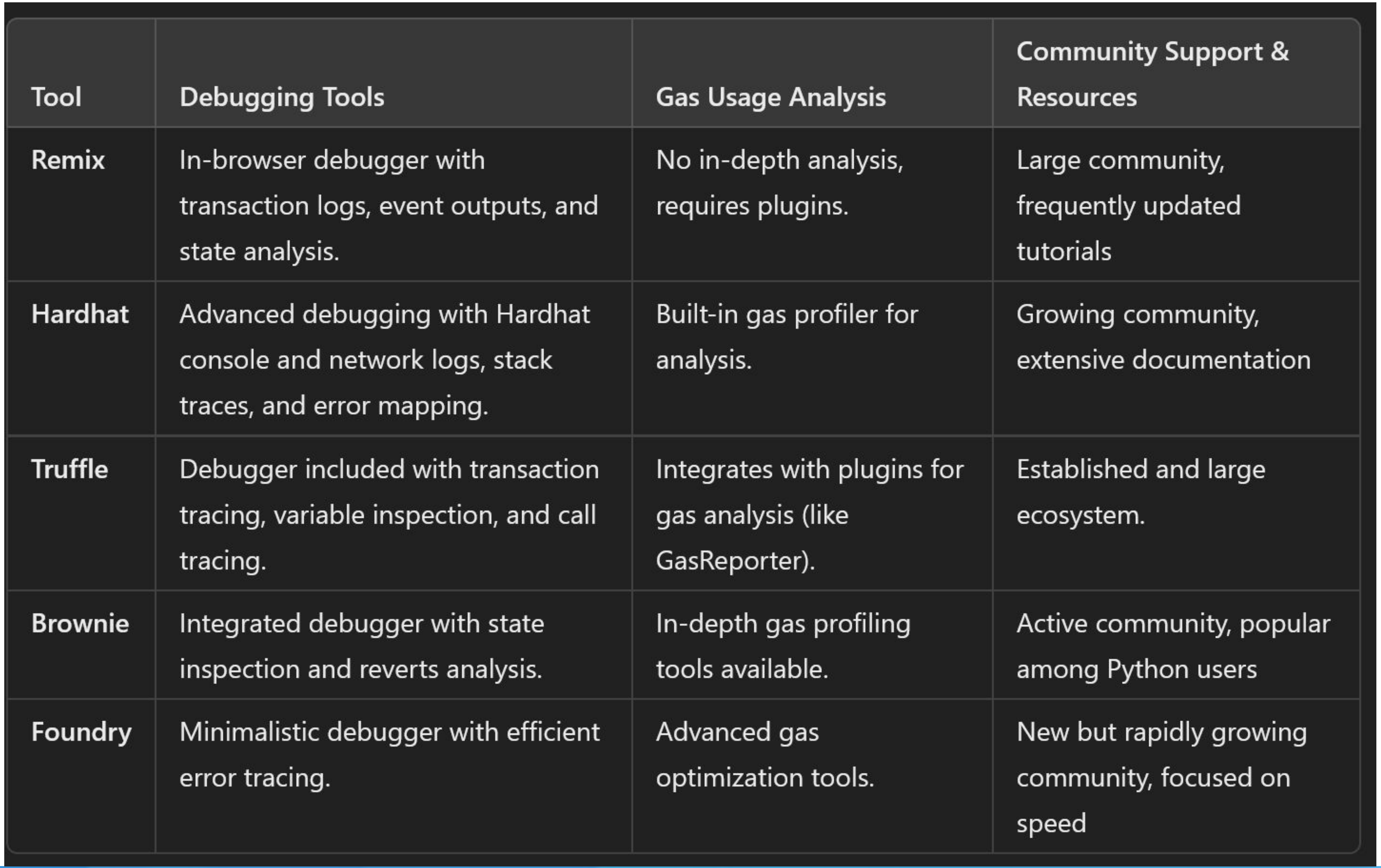


Tool	Deployment Options	Local Blockchain Support	Testing Framework
Remix	Direct deployment to public testnets or local environments via plugins (e.g., MetaMask).	Uses Ganache or connected testnets (via browser extension).	Limited built-in testing (JavaScript)
Hardhat	Supports deployment to local (via Hardhat Network) and public testnets/ mainnets. Custom scripts for deployment.	Hardhat Network for local development, integrates with Ganache	Mocha, Chai for unit testing
Truffle	Integrated deployment tools for local, testnet, and mainnet deployments.	Ganache integration for local blockchain simulation	Mocha, Chai, and built-in testing utilities
Brownie	Built-in deployment to public testnets/ mainnets with simple command-line interface.	Supports Ganache, Ethereum mainnet, and other local networks	PyTest integration, in-depth contract testing
Foundry	Deployment to local blockchains, testnets, and mainnets with minimal configuration.	Foundry's native testnet tools and local blockchain support	Forge-based testing suite

Features Comparison

Development Environments for Smart Contracts

- Features



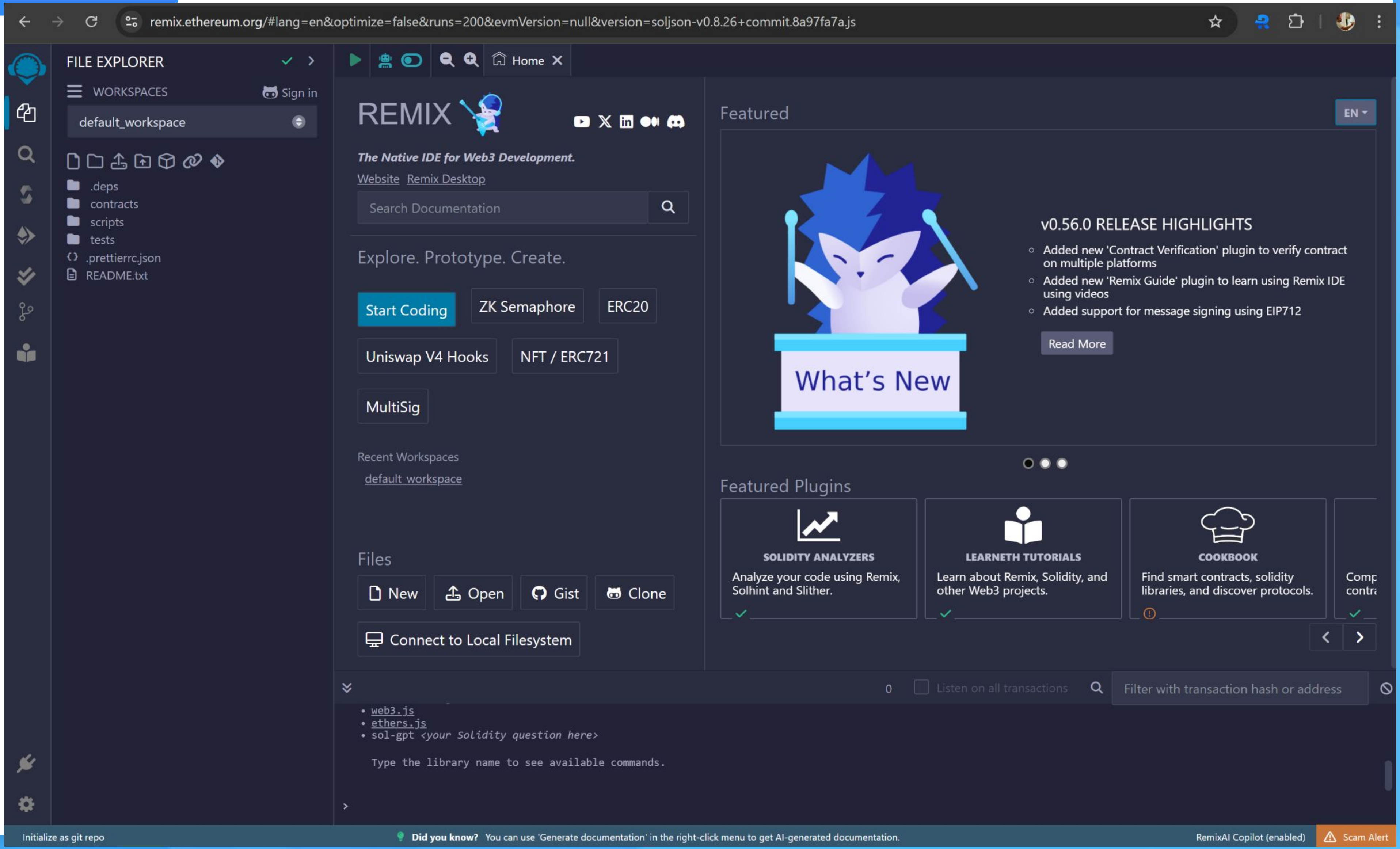
Tool	Debugging Tools	Gas Usage Analysis	Community Support & Resources
Remix	In-browser debugger with transaction logs, event outputs, and state analysis.	No in-depth analysis, requires plugins.	Large community, frequently updated tutorials
Hardhat	Advanced debugging with Hardhat console and network logs, stack traces, and error mapping.	Built-in gas profiler for analysis.	Growing community, extensive documentation
Truffle	Debugger included with transaction tracing, variable inspection, and call tracing.	Integrates with plugins for gas analysis (like GasReporter).	Established and large ecosystem.
Brownie	Integrated debugger with state inspection and reverts analysis.	In-depth gas profiling tools available.	Active community, popular among Python users
Foundry	Minimalistic debugger with efficient error tracing.	Advanced gas optimization tools.	New but rapidly growing community, focused on speed

Remix

remix.

ethereum.

org



remix.ethereum.org/#lang=en&optimize=false&runs=200&evmVersion=null&version=soljson-v0.8.26+commit.8a97fa7a.js

The Native IDE for Web3 Development.

Explore. Prototype. Create.

v0.56.0 RELEASE HIGHLIGHTS

- Added new 'Contract Verification' plugin to verify contract on multiple platforms
- Added new 'Remix Guide' plugin to learn using Remix IDE using videos
- Added support for message signing using EIP712

Analyze your code using Remix, Solhint and Slither.

Learn about Remix, Solidity, and other Web3 projects.

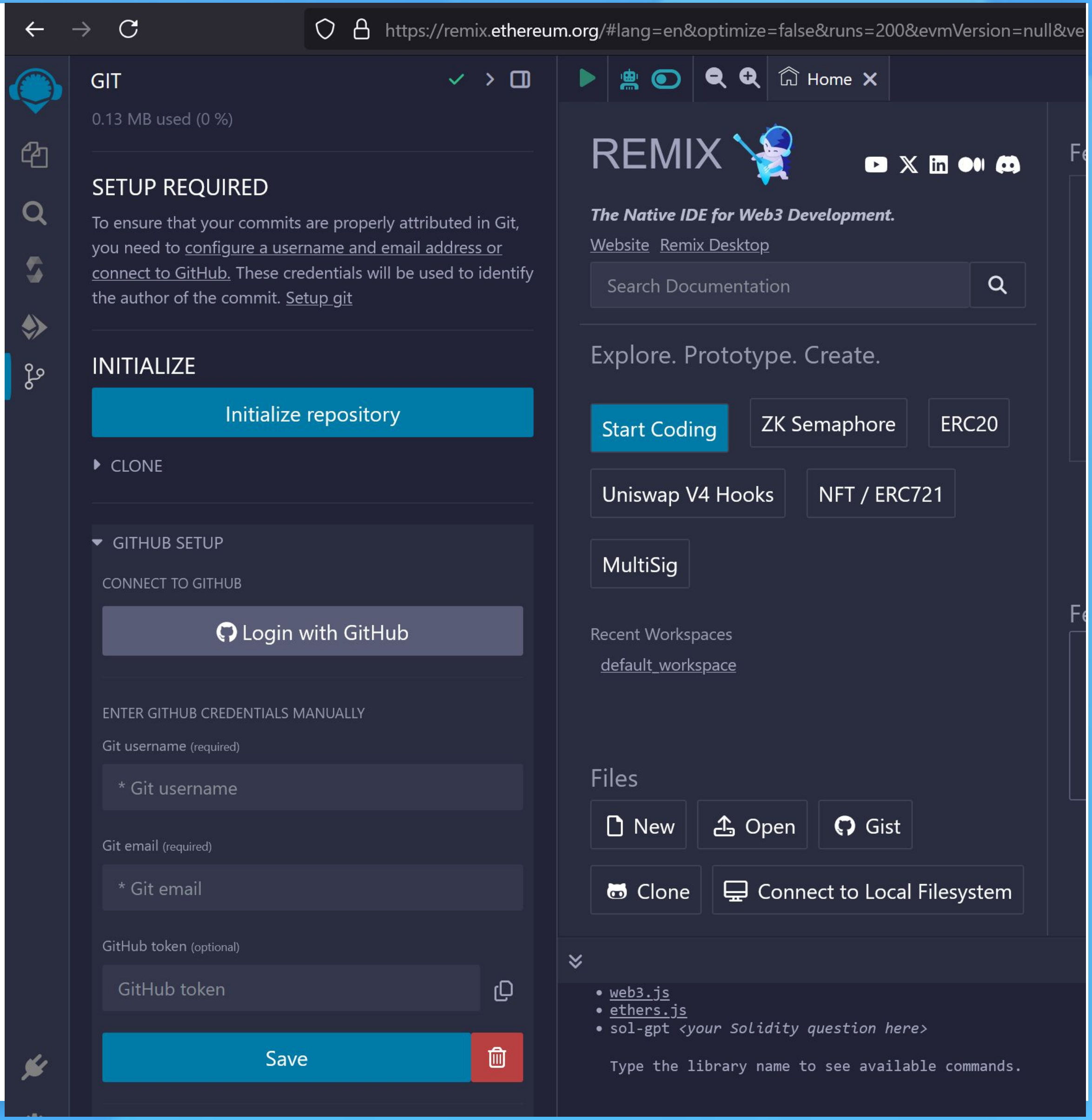
Find smart contracts, solidity libraries, and discover protocols.

```
• web3.js
• ethers.js
• sol-gpt <your Solidity question here>

 Type the library name to see available commands.
```

Did you know? You can use 'Generate documentation' in the right-click menu to get AI-generated documentation.

Remix + GitHub

Connect to GitHub Account



https://remix.ethereum.org/#lang=en&optimize=false&runs=200&evmVersion=null&ve

GIT

0.13 MB used (0 %)

SETUP REQUIRED

To ensure that your commits are properly attributed in Git, you need to configure a username and email address or connect to GitHub. These credentials will be used to identify the author of the commit. Setup git

INITIALIZE

GITHUB SETUP

CONNECT TO GITHUB

ENTER GITHUB CREDENTIALS MANUALLY

Git username (required)

Git email (required)

GitHub token (optional)

REMIX

The Native IDE for Web3 Development.

Explore. Prototype. Create.

Recent Workspaces

default_workspace

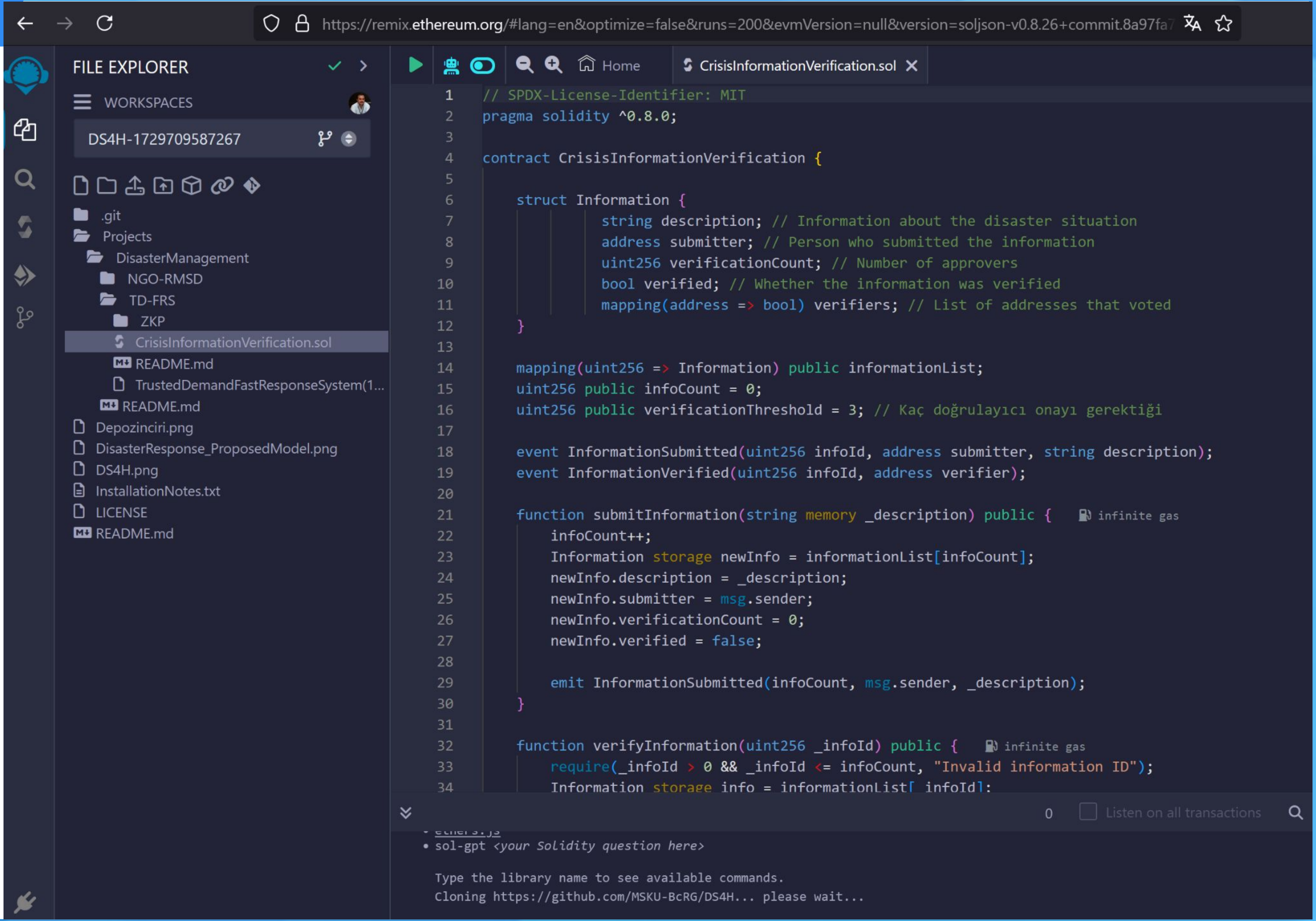
Files

- web3.js
- ethers.js
- sol-gpt *<your Solidity question here>*

```
Type the library name to see available commands.
```

Remix

Clone Repo from Github



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

contract CrisisInformationVerification {

    struct Information {
            string description; // Information about the disaster situation
            address submitter; // Person who submitted the information
            uint256 verificationCount; // Number of approvers
            bool verified; // Whether the information was verified
            mapping(address => bool) verifiers; // List of addresses that voted
    }

    mapping(uint256 => Information) public informationList;
    uint256 public infoCount = 0;
    uint256 public verificationThreshold = 3; // Kaç doğrulayıcı onayı gerektiği

    event InformationSubmitted(uint256 infoId, address submitter, string description);
    event InformationVerified(uint256 infoId, address verifier);

    function submitInformation(string memory _description) public {    infinite gas
        infoCount++;
        Information storage newInfo = informationList[infoCount];
        newInfo.description = _description;
        newInfo.submitter = msg.sender;
        newInfo.verificationCount = 0;
        newInfo.verified = false;

        emit InformationSubmitted(infoCount, msg.sender, _description);
    }

    function verifyInformation(uint256 _infoId) public {    infinite gas
        require(_infoId > 0 && _infoId <= infoCount, "Invalid information ID");
        Information storage info = informationList[ infoId];
```

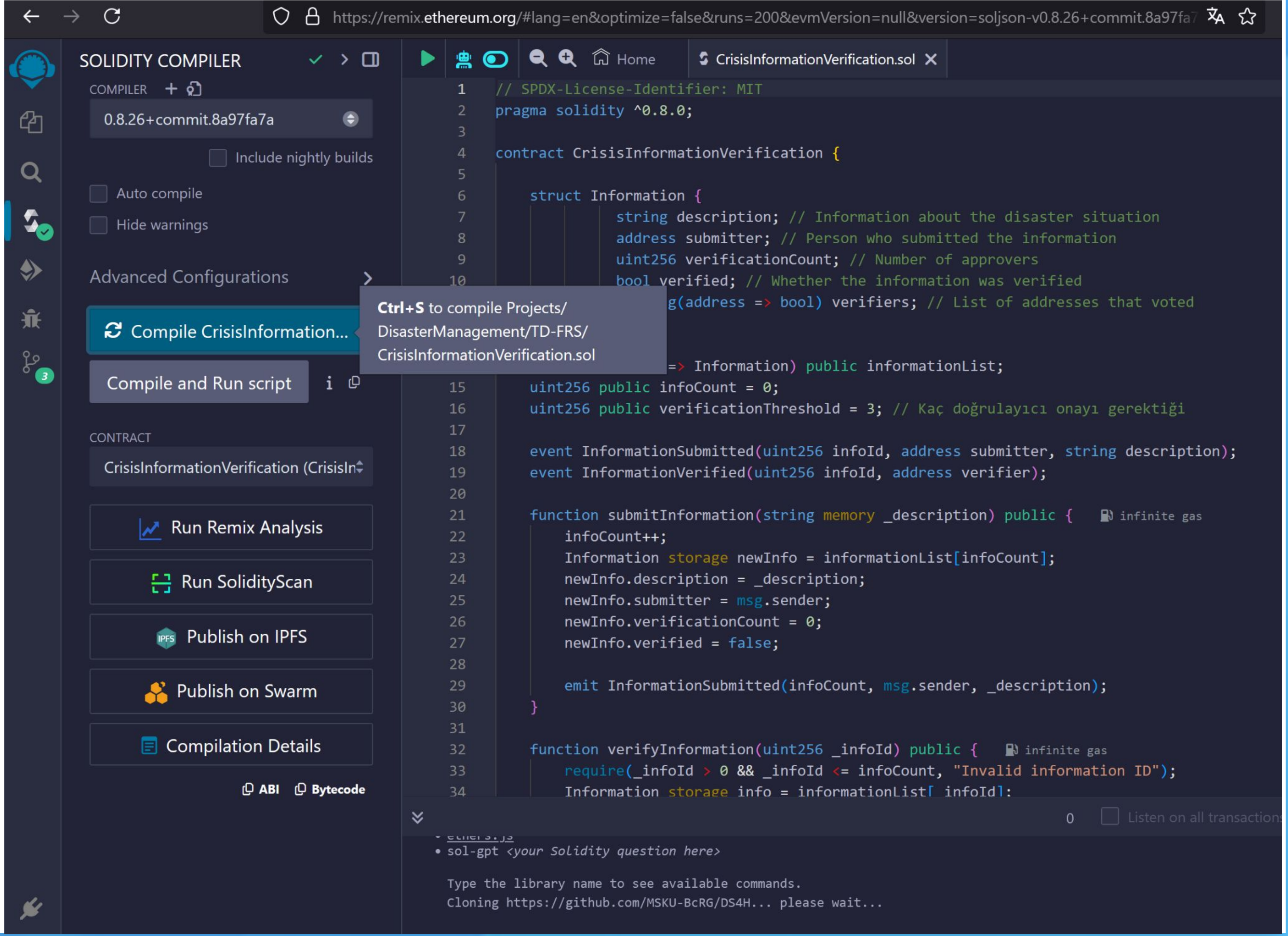
```
• ethers.js
• sol-gpt <your Solidity question here>

Type the library name to see available commands.
Cloning https://github.com/MSKU-BcRG/DS4H... please wait...
```

Remix - compile

remix.

ethereum.

org



https://remix.ethereum.org/#lang=en&optimize=false&runs=200&evmVersion=null&version=soljson-v0.8.26+commit.8a97fa7

SOLIDITY COMPILER

COMPILER

0.8.26+commit.8a97fa7a

☐ Include nightly builds

☐ Auto compile

☐ Hide warnings

CONTRACT

CrisisInformationVerification (CrisisIn

Ctrl+S to compile Projects/DisasterManagement/TD-FRS/CrisisInformationVerification.sol

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

contract CrisisInformationVerification {

    struct Information {
        string description; // Information about the disaster situation
        address submitter; // Person who submitted the information
        uint256 verificationCount; // Number of approvers
        bool verified; // Whether the information was verified
        g(address => bool) verifiers; // List of addresses that voted

        => Information) public informationList;
    uint256 public infoCount = 0;
    uint256 public verificationThreshold = 3; // Kaç doğrulayıcı onayı gerektiği

    event InformationSubmitted(uint256 infoId, address submitter, string description);
    event InformationVerified(uint256 infoId, address verifier);

    function submitInformation(string memory _description) public {    infinite gas
        infoCount++;
        Information storage newInfo = informationList[infoCount];
        newInfo.description = _description;
        newInfo.submitter = msg.sender;
        newInfo.verificationCount = 0;
        newInfo.verified = false;

        emit InformationSubmitted(infoCount, msg.sender, _description);
    }

    function verifyInformation(uint256 _infoId) public {    infinite gas
        require(_infoId > 0 && _infoId <= infoCount, "Invalid information ID");
        Information storage info = informationList[_infoId];
```

```
• sol-gpt <your Solidity question here>

  Type the library name to see available commands.
  Cloning https://github.com/MSKU-BcRG/DS4H... please wait...
```

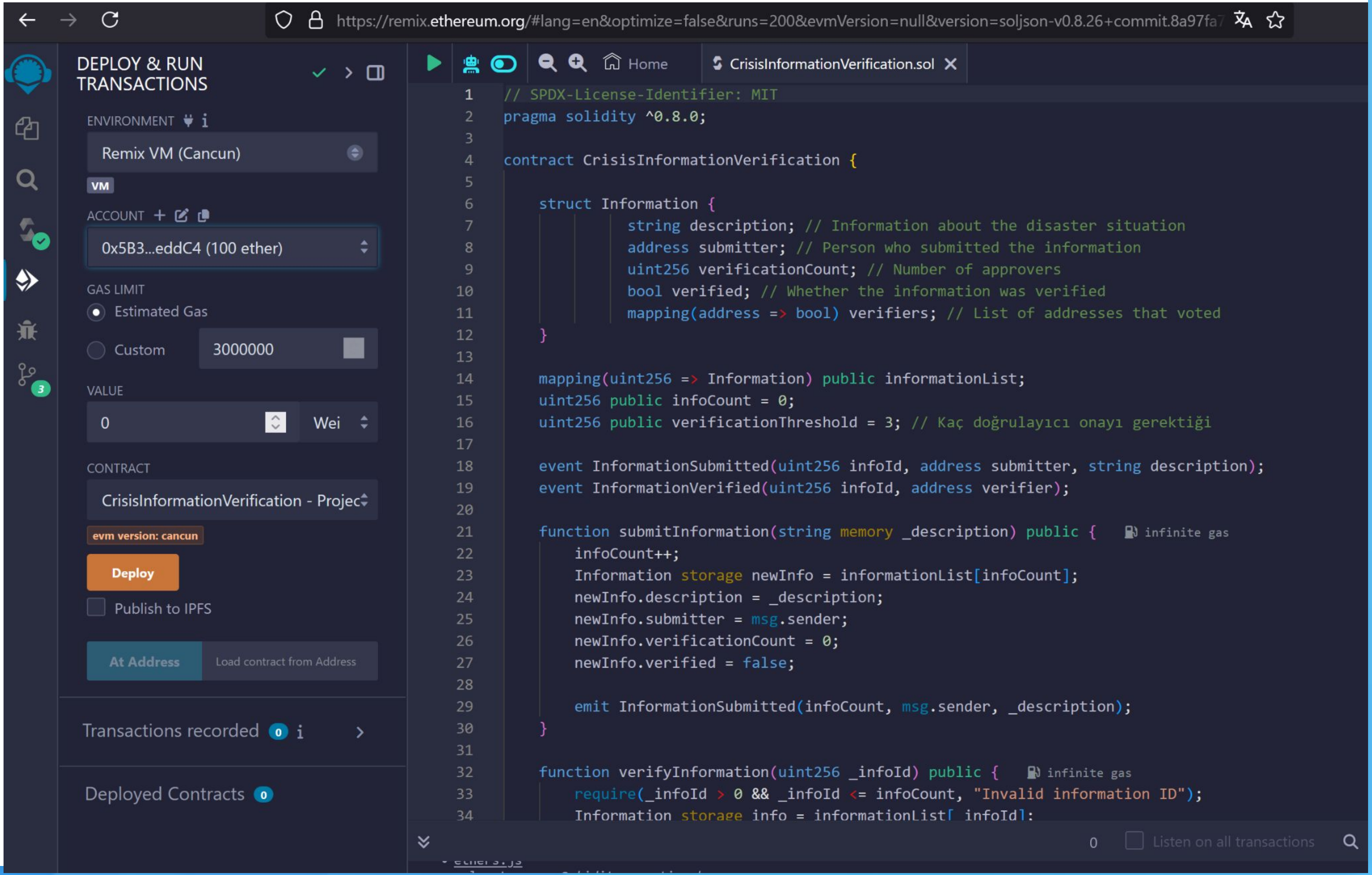
Remix - deploy and run

Deploy & Run

on

Remix vms

or …



https://remix.ethereum.org/#lang=en&optimize=false&runs=200&evmVersion=null&version=soljson-v0.8.26+commit.8a97fa

DEPLOY & RUN TRANSACTIONS

ENVIRONMENT: Remix VM (Cancun)

ACCOUNT: 0x5B3...eddC4 (100 ether)

GAS LIMIT

● Estimated Gas

○ Custom 3000000

VALUE: 0 Wei

CONTRACT: CrisisInformationVerification - Projec

evm version: cancun

☐ Publish to IPFS

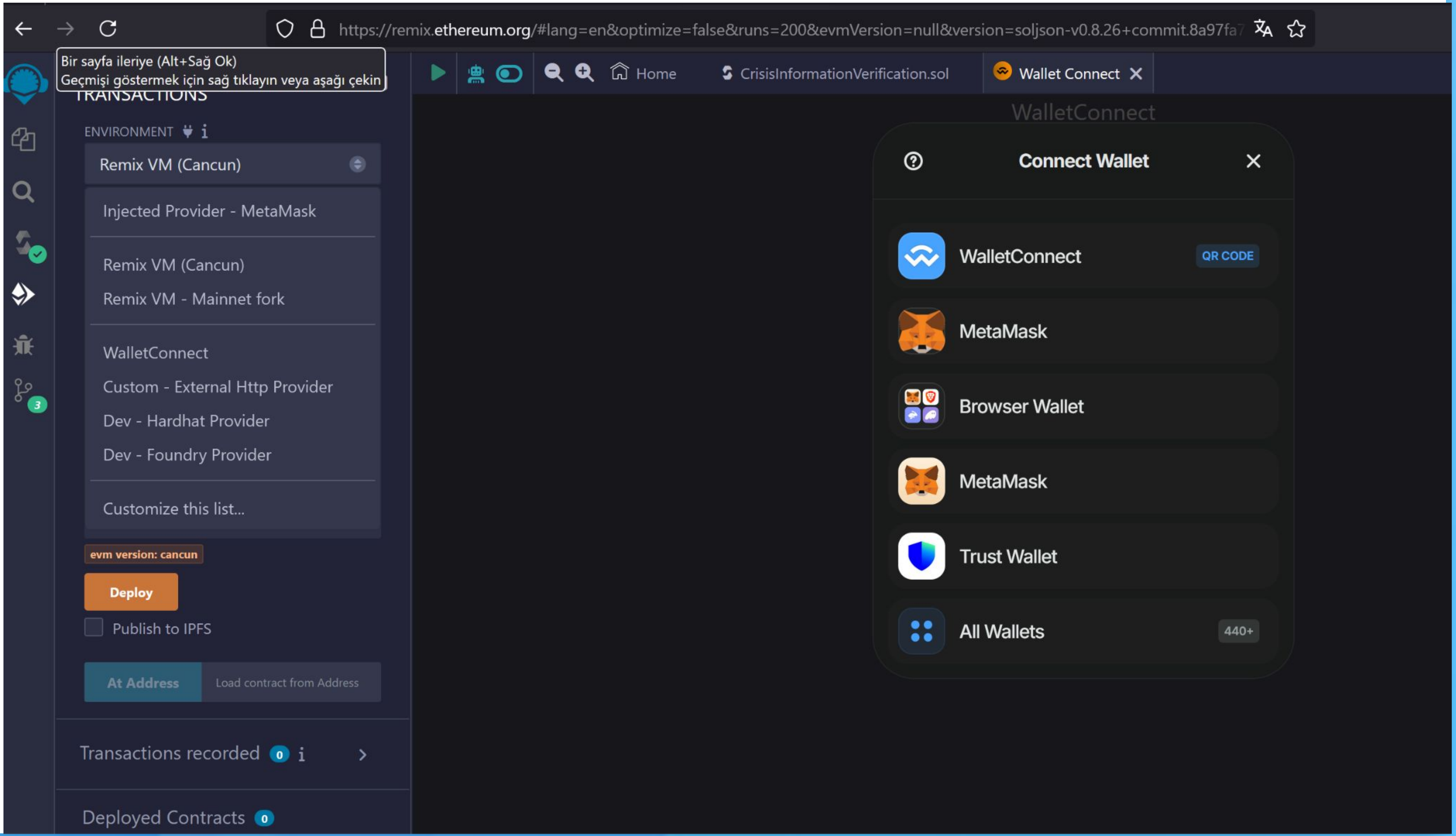
Transactions recorded 0

Deployed Contracts 0

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

contract CrisisInformationVerification {

    struct Information {
            string description; // Information about the disaster situation
            address submitter; // Person who submitted the information
            uint256 verificationCount; // Number of approvers
            bool verified; // Whether the information was verified
            mapping(address => bool) verifiers; // List of addresses that voted
    }

    mapping(uint256 => Information) public informationList;
    uint256 public infoCount = 0;
    uint256 public verificationThreshold = 3; // Kaç doğrulayıcı onayı gerektiği

    event InformationSubmitted(uint256 infoId, address submitter, string description);
    event InformationVerified(uint256 infoId, address verifier);

    function submitInformation(string memory _description) public {
        infoCount++;
        Information storage newInfo = informationList[infoCount];
        newInfo.description = _description;
        newInfo.submitter = msg.sender;
        newInfo.verificationCount = 0;
        newInfo.verified = false;

        emit InformationSubmitted(infoCount, msg.sender, _description);
    }

    function verifyInformation(uint256 _infoId) public {
        require(_infoId > 0 && _infoId <= infoCount, "Invalid information ID");
        Information storage info = informationList[_infoId];
```

Remix - deploy and run

Deploy & Run
on
Remix vms
or other
- such as
- “connect
wallets”



https://remix.ethereum.org/#lang=en&optimize=false&runs=200&evmVersion=null&version=soljson-v0.8.26+commit.8a97fa7

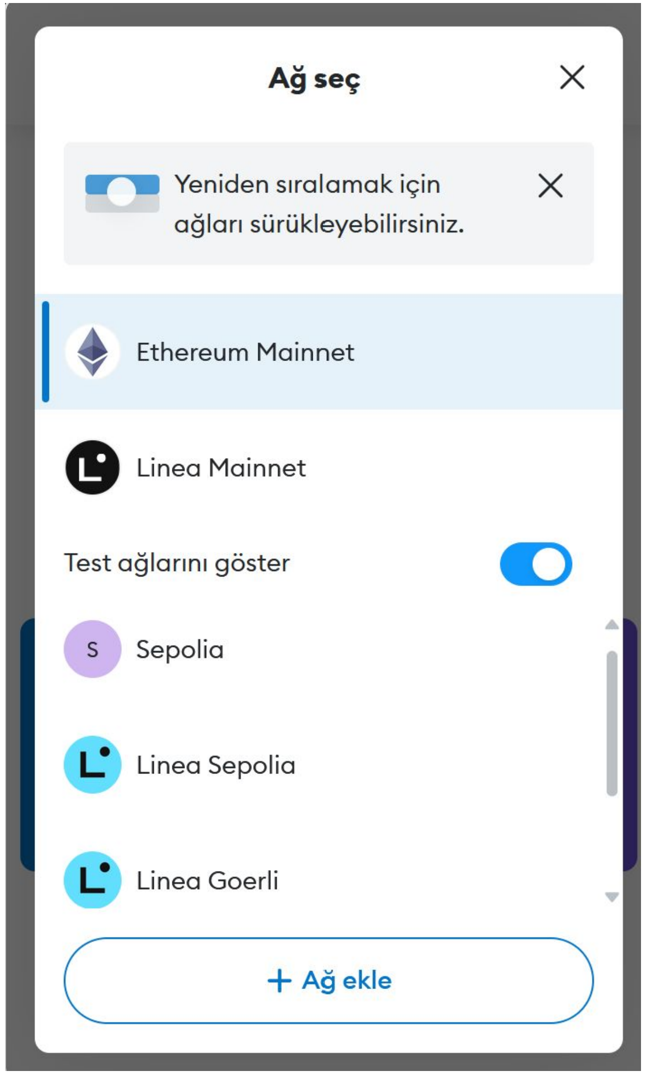
Bir sayfa ileriye (Alt+Sağ Ok)
Geçmişi göstermek için sağ tıklayın veya aşağı çekin

ENVIRONMENT: Remix VM (Cancun)

evm version: cancun

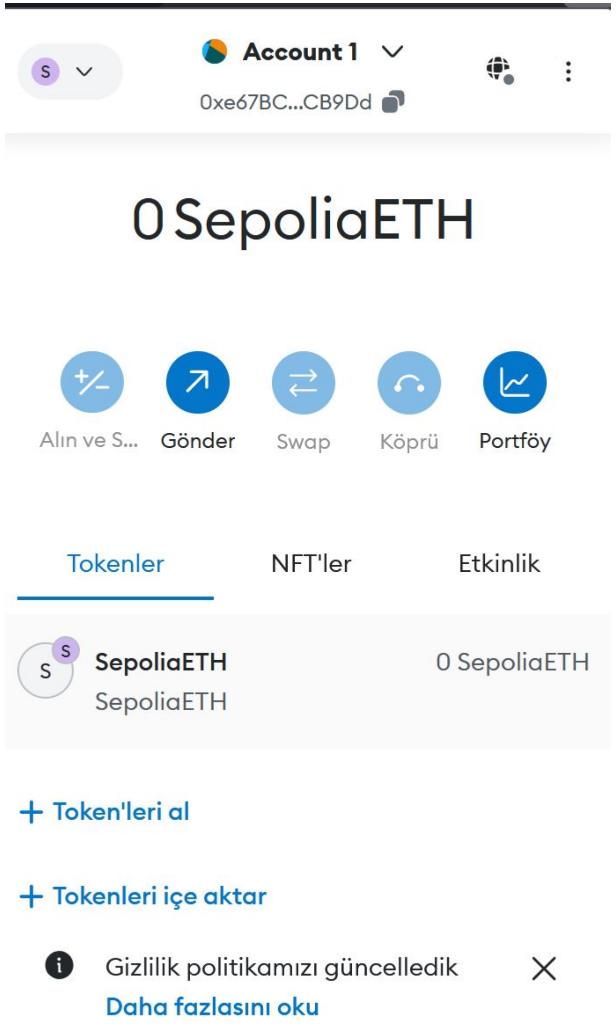
Metamask - wallet

Add an account from that test network



Ağ seç

Yeniden sıralamak için ağları sürükleyebilirsiniz.



Account 1

0xe67BC...CB9Dd

0 SepoliaETH

SepoliaETH 0 SepoliaETH

SepoliaETH

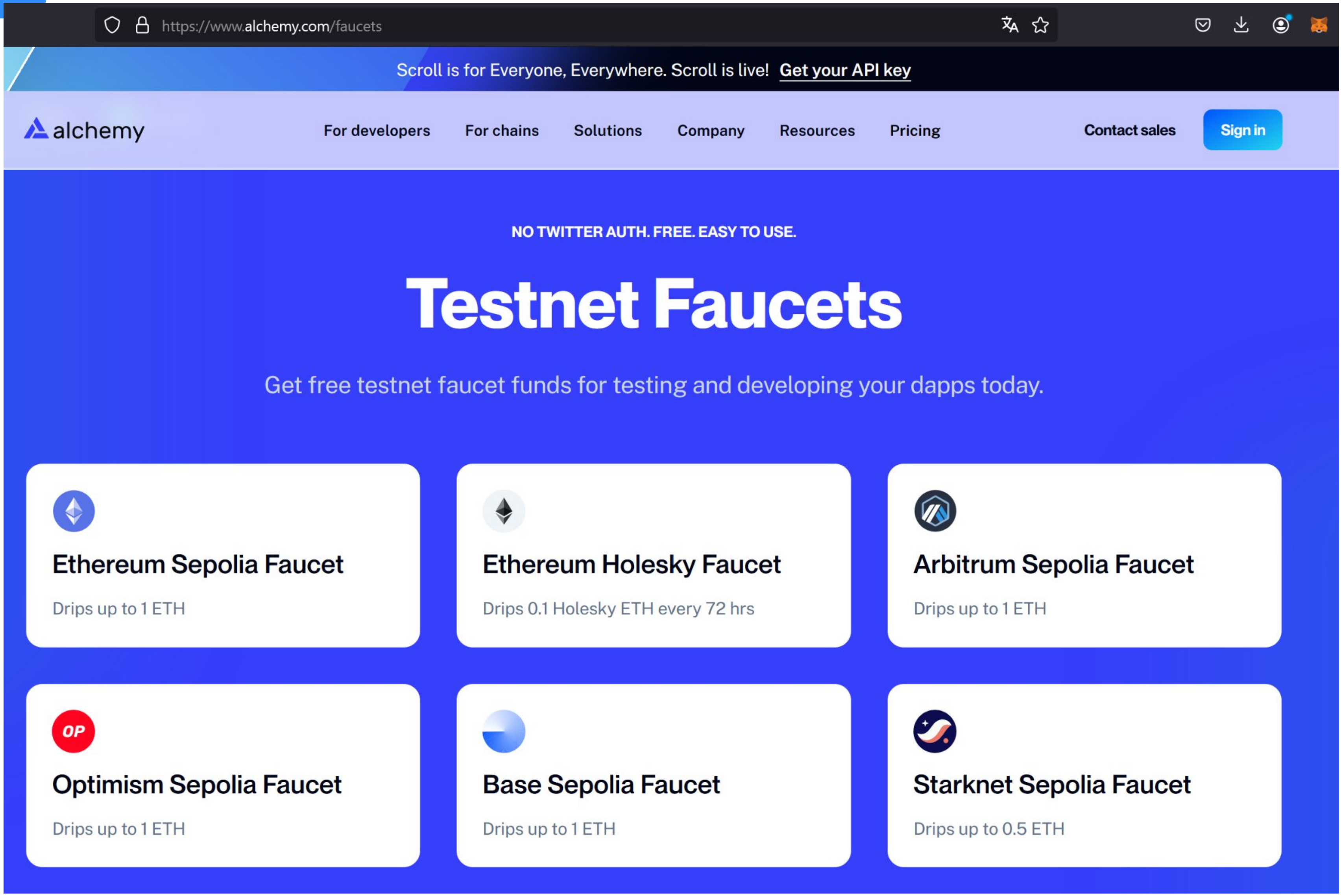
Gizlilik politikamızı güncelledik

Testnet Faucets

Faucets: Alchemy, Chainlink, or QuickNode)

Paste your MetaMask address, and request test ETH.

Each may have its own conditions such as verifying through social media or the presence of ETH in your wallet.



https://www.alchemy.com/faucets

Scroll is for Everyone, Everywhere. Scroll is live! Get your API key

NO TWITTER AUTH. FREE. EASY TO USE.

Testnet Faucets

Get free testnet faucet funds for testing and developing your dapps today.

Ethereum Sepolia Faucet
Drips up to 1 ETH

Ethereum Holesky Faucet
Drips 0.1 Holesky ETH every 72 hrs

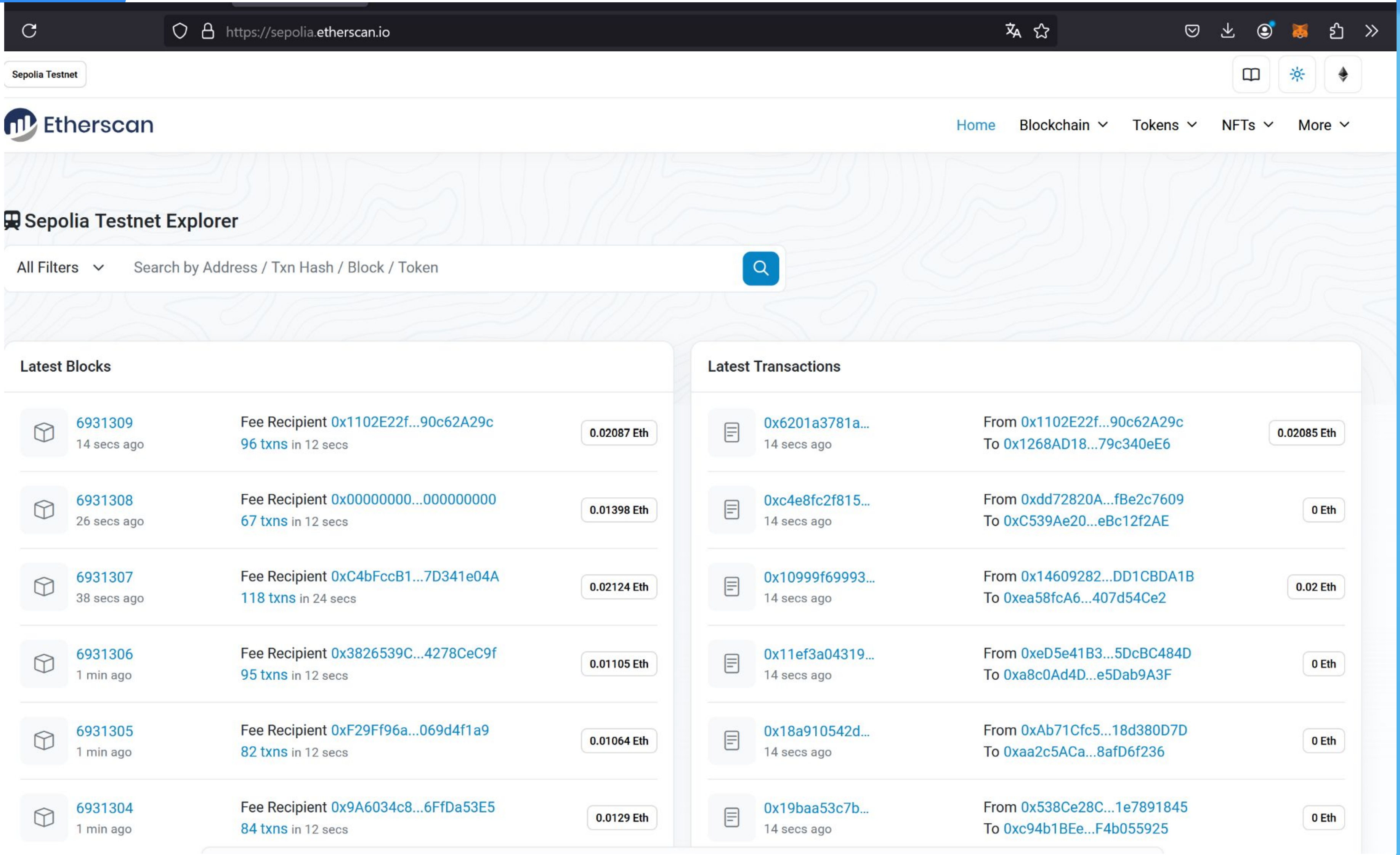
Arbitrum Sepolia Faucet
Drips up to 1 ETH

Optimism Sepolia Faucet
Drips up to 1 ETH

Base Sepolia Faucet
Drips up to 1 ETH

Starknet Sepolia Faucet
Drips up to 0.5 ETH

Testnet Explorers



https://sepolia.etherscan.io

Sepolia Testnet

Etherscan

Home | Blockchain | Tokens | NFTs | More

Sepolia Testnet Explorer

All Filters | Search by Address / Txn Hash / Block / Token

Latest Blocks

6931309 14 secs ago	Fee Recipient 0x1102E22f...90c62A29c 96 txns in 12 secs	0.02087 Eth
6931308 26 secs ago	Fee Recipient 0x00000000...000000000 67 txns in 12 secs	0.01398 Eth
6931307 38 secs ago	Fee Recipient 0xC4bFccB1...7D341e04A 118 txns in 24 secs	0.02124 Eth
6931306 1 min ago	Fee Recipient 0x3826539C...4278CeC9f 95 txns in 12 secs	0.01105 Eth
6931305 1 min ago	Fee Recipient 0xF29Ff96a...069d4f1a9 82 txns in 12 secs	0.01064 Eth
6931304 1 min ago	Fee Recipient 0x9A6034c8...6FfDa53E5 84 txns in 12 secs	0.0129 Eth

Latest Transactions

0x6201a3781a... 14 secs ago	From 0x1102E22f...90c62A29c To 0x1268AD18...79c340eE6	0.02085 Eth
0xc4e8fc2f815... 14 secs ago	From 0xdd72820A...fBe2c7609 To 0xC539Ae20...eBc12f2AE	0 Eth
0x10999f69993... 14 secs ago	From 0x14609282...DD1CBDA1B To 0xea58fcA6...407d54Ce2	0.02 Eth
0x11ef3a04319... 14 secs ago	From 0xeD5e41B3...5DcBC484D To 0xa8c0Ad4D...e5Dab9A3F	0 Eth
0x18a910542d... 14 secs ago	From 0xAb71Cfc5...18d380D7D To 0xaa2c5ACa...8afD6f236	0 Eth
0x19baa53c7b... 14 secs ago	From 0x538Ce28C...1e7891845 To 0xc94b1BEe...F4b055925	0 Eth

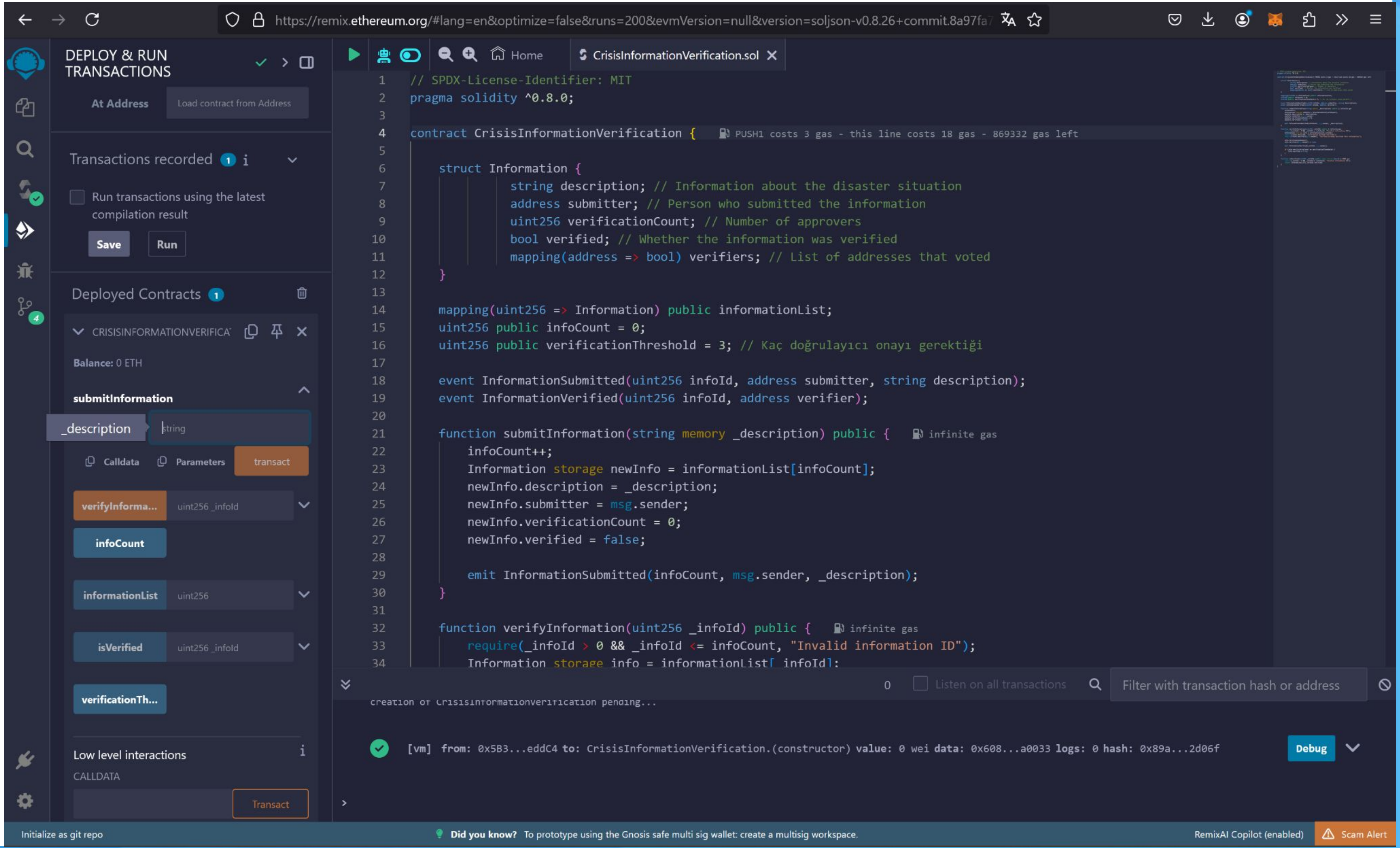
Remix - deploy and run

Deploy & Run

on

Remix vms

or …

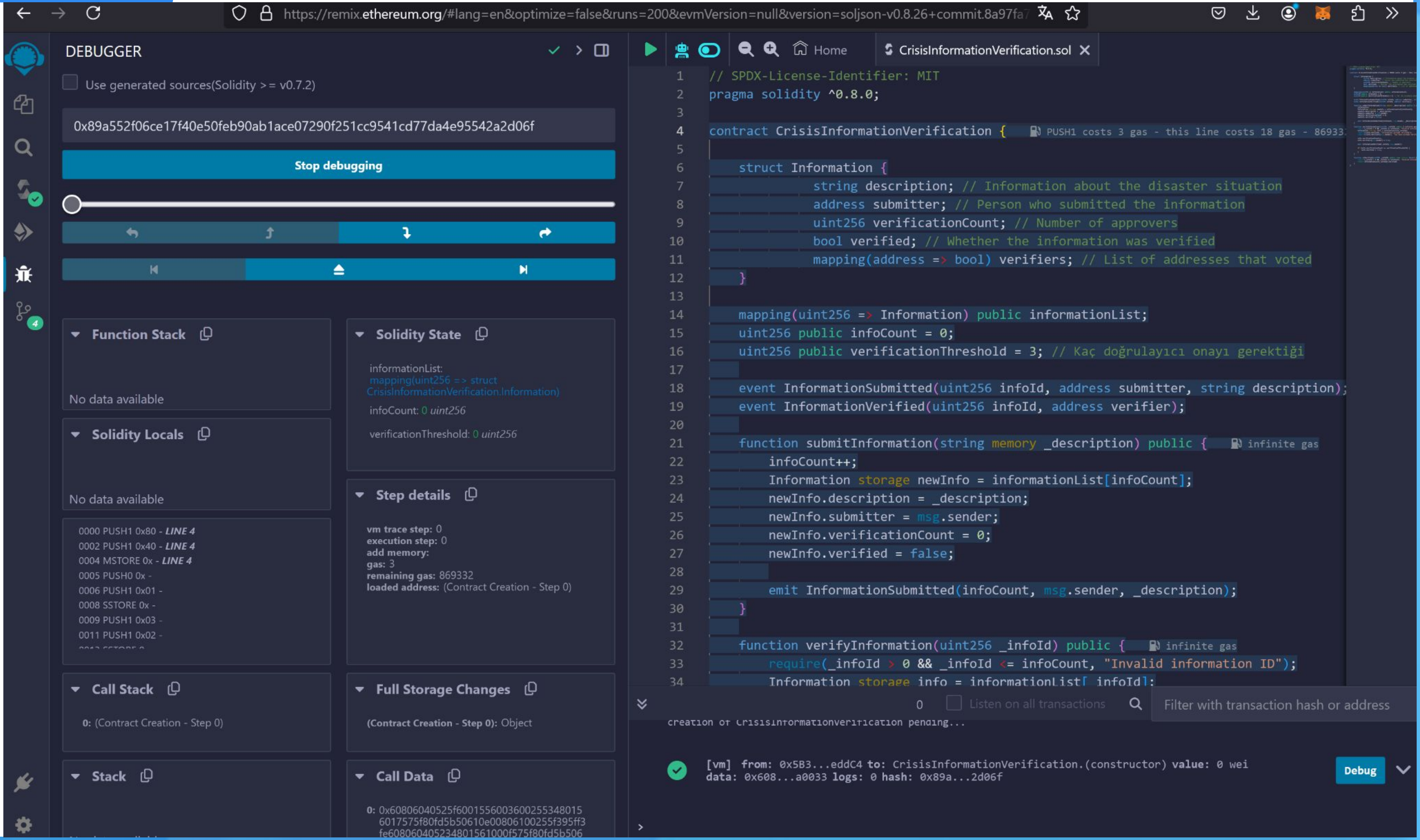


```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract CrisisInformationVerification {

    struct Information {
            string description; // Information about the disaster situation
            address submitter; // Person who submitted the information
            uint256 verificationCount; // Number of approvers
            bool verified; // Whether the information was verified
            mapping(address => bool) verifiers; // List of addresses that voted
    }

    mapping(uint256 => Information) public informationList;
    uint256 public infoCount = 0;
    uint256 public verificationThreshold = 3; // Kaç doğrulayıcı onayı gerektiği

    event InformationSubmitted(uint256 infoId, address submitter, string description);
    event InformationVerified(uint256 infoId, address verifier);

    function submitInformation(string memory _description) public {
        infoCount++;
        Information storage newInfo = informationList[infoCount];
        newInfo.description = _description;
        newInfo.submitter = msg.sender;
        newInfo.verificationCount = 0;
        newInfo.verified = false;

        emit InformationSubmitted(infoCount, msg.sender, _description);
    }

    function verifyInformation(uint256 _infoId) public {
        require(_infoId > 0 && _infoId <= infoCount, "Invalid information ID");
        Information storage info = informationList[_infoId];
```

```
creation of CrisisInformationVerification pending...

[vm] from: 0x5B3...eddC4 to: CrisisInformationVerification.(constructor) value: 0 wei data: 0x608...a0033 logs: 0 hash: 0x89a...2d06f
```

Remix - debug

Debug



☐ Use generated sources(Solidity >= v0.7.2)

0x89a552f06ce17f40e50feb90ab1ace07290f251cc9541cd77da4e95542a2d06f

Function Stack

No data available

Solidity Locals

No data available

```
0000 PUSH1 0x80 - LINE 4
0002 PUSH1 0x40 - LINE 4
0004 MSTORE 0x - LINE 4
0005 PUSH0 0x -
0006 PUSH1 0x01 -
0008 SSTORE 0x -
0009 PUSH1 0x03 -
0011 PUSH1 0x02 -
```

Call Stack

0: (Contract Creation - Step 0)

Stack

Solidity State

informationList: mapping(uint256 => struct CrisisInformationVerification.Information)

infoCount: 0 uint256

verificationThreshold: 0 uint256

Step details

vm trace step: 0
execution step: 0
add memory:
gas: 3
remaining gas: 869332
loaded address: (Contract Creation - Step 0)

Full Storage Changes

(Contract Creation - Step 0): Object

Call Data

0: 0x60806040525f600155600360025534801560
17575f80fd5b50610e00806100255f395ff3
fe608060405234801561000f575f80fd5b506

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

contract CrisisInformationVerification {    PUSH1 costs 3 gas - this line costs 18 gas - 86933

    struct Information {
            string description; // Information about the disaster situation
            address submitter; // Person who submitted the information
            uint256 verificationCount; // Number of approvers
            bool verified; // Whether the information was verified
            mapping(address => bool) verifiers; // List of addresses that voted
    }

    mapping(uint256 => Information) public informationList;
    uint256 public infoCount = 0;
    uint256 public verificationThreshold = 3; // Kaç doğrulayıcı onayı gerektiği

    event InformationSubmitted(uint256 infoId, address submitter, string description);
    event InformationVerified(uint256 infoId, address verifier);

    function submitInformation(string memory _description) public {    infinite gas
        infoCount++;
        Information storage newInfo = informationList[infoCount];
        newInfo.description = _description;
        newInfo.submitter = msg.sender;
        newInfo.verificationCount = 0;
        newInfo.verified = false;

        emit InformationSubmitted(infoCount, msg.sender, _description);
    }

    function verifyInformation(uint256 _infoId) public {    infinite gas
        require(_infoId > 0 && _infoId <= infoCount, "Invalid information ID");
        Information storage info = informationList[ infoId];
```

```
creation of CrisisInformationVerification pending...

[vm]  from: 0x5B3...eddC4 to: CrisisInformationVerification.(constructor) value: 0 wei
data: 0x608...a0033 logs: 0 hash: 0x89a...2d06f
```

Test Networks Compared

FOUR

Test Networks Compared

Test environments

- Remix VM
- Testnets
- Local Setups
  - Ganache
  - Hardhat
- DS4H

General Features and Network Simulation

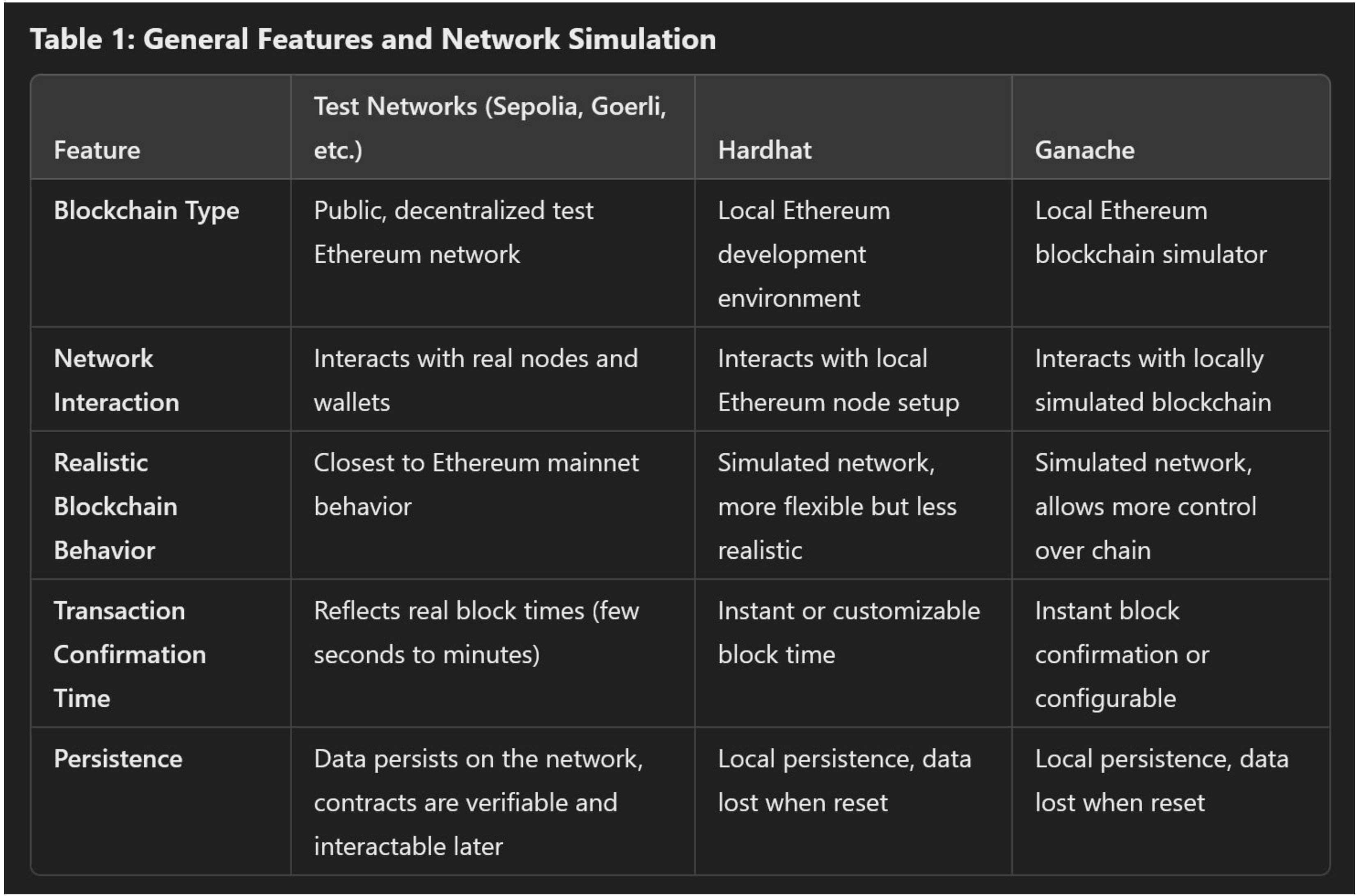


Table 1: General Features and Network Simulation

Feature	Test Networks (Sepolia, Goerli, etc.)	Hardhat	Ganache
Blockchain Type	Public, decentralized test Ethereum network	Local Ethereum development environment	Local Ethereum blockchain simulator
Network Interaction	Interacts with real nodes and wallets	Interacts with local Ethereum node setup	Interacts with locally simulated blockchain
Realistic Blockchain Behavior	Closest to Ethereum mainnet behavior	Simulated network, more flexible but less realistic	Simulated network, allows more control over chain
Transaction Confirmation Time	Reflects real block times (few seconds to minutes)	Instant or customizable block time	Instant block confirmation or configurable
Persistence	Data persists on the network, contracts are verifiable and interactable later	Local persistence, data lost when reset	Local persistence, data lost when reset

Development and Testing Tools

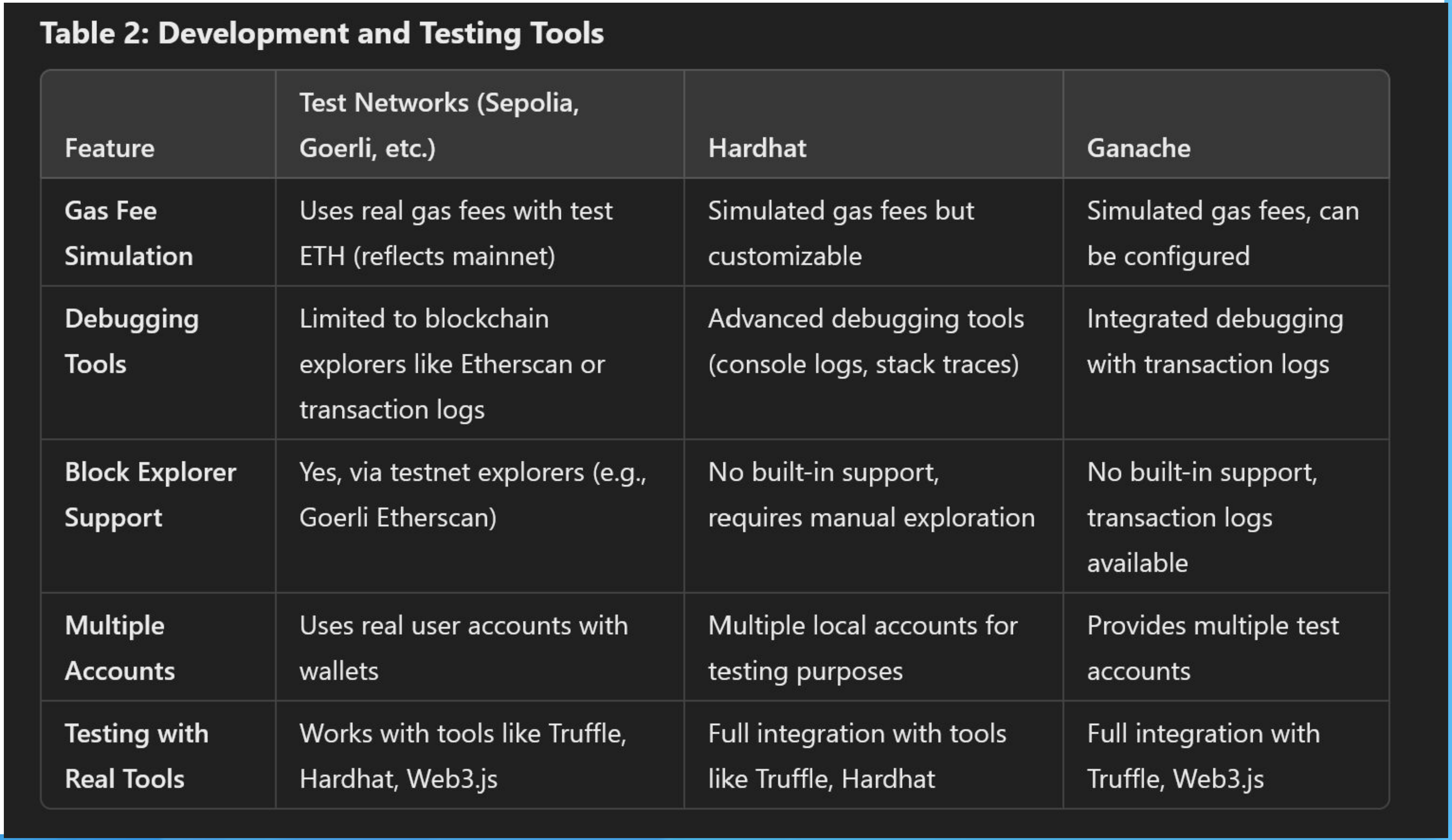


Table 2: Development and Testing Tools

Feature	Test Networks (Sepolia, Goerli, etc.)	Hardhat	Ganache
Gas Fee Simulation	Uses real gas fees with test ETH (reflects mainnet)	Simulated gas fees but customizable	Simulated gas fees, can be configured
Debugging Tools	Limited to blockchain explorers like Etherscan or transaction logs	Advanced debugging tools (console logs, stack traces)	Integrated debugging with transaction logs
Block Explorer Support	Yes, via testnet explorers (e.g., Goerli Etherscan)	No built-in support, requires manual exploration	No built-in support, transaction logs available
Multiple Accounts	Uses real user accounts with wallets	Multiple local accounts for testing purposes	Provides multiple test accounts
Testing with Real Tools	Works with tools like Truffle, Hardhat, Web3.js	Full integration with tools like Truffle, Hardhat	Full integration with Truffle, Web3.js

Performance and Realism

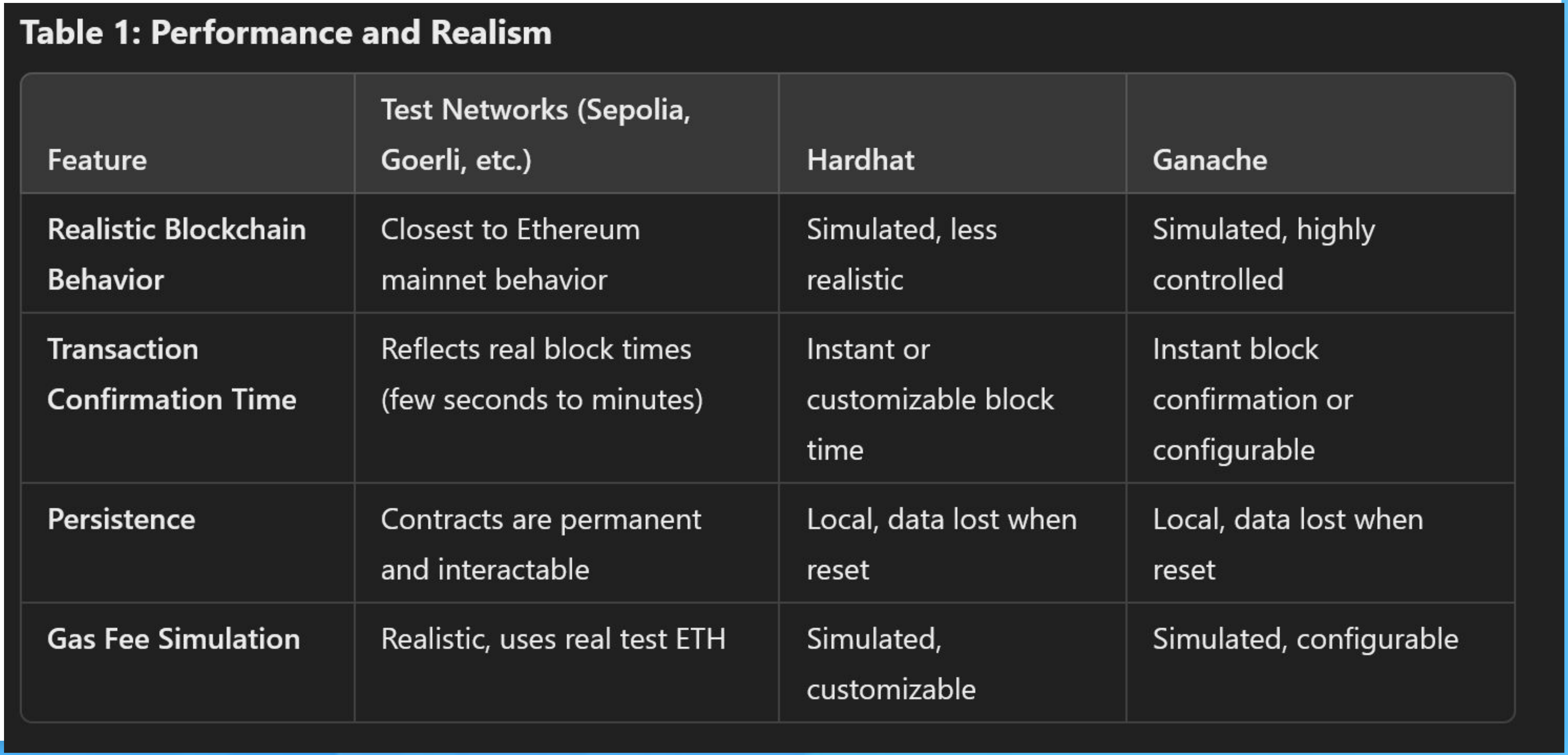


Table 1: Performance and Realism

Feature	Test Networks (Sepolia, Goerli, etc.)	Hardhat	Ganache
Realistic Blockchain Behavior	Closest to Ethereum mainnet behavior	Simulated, less realistic	Simulated, highly controlled
Transaction Confirmation Time	Reflects real block times (few seconds to minutes)	Instant or customizable block time	Instant block confirmation or configurable
Persistence	Contracts are permanent and interactable	Local, data lost when reset	Local, data lost when reset
Gas Fee Simulation	Realistic, uses real test ETH	Simulated, customizable	Simulated, configurable

User Interaction and Flexibility

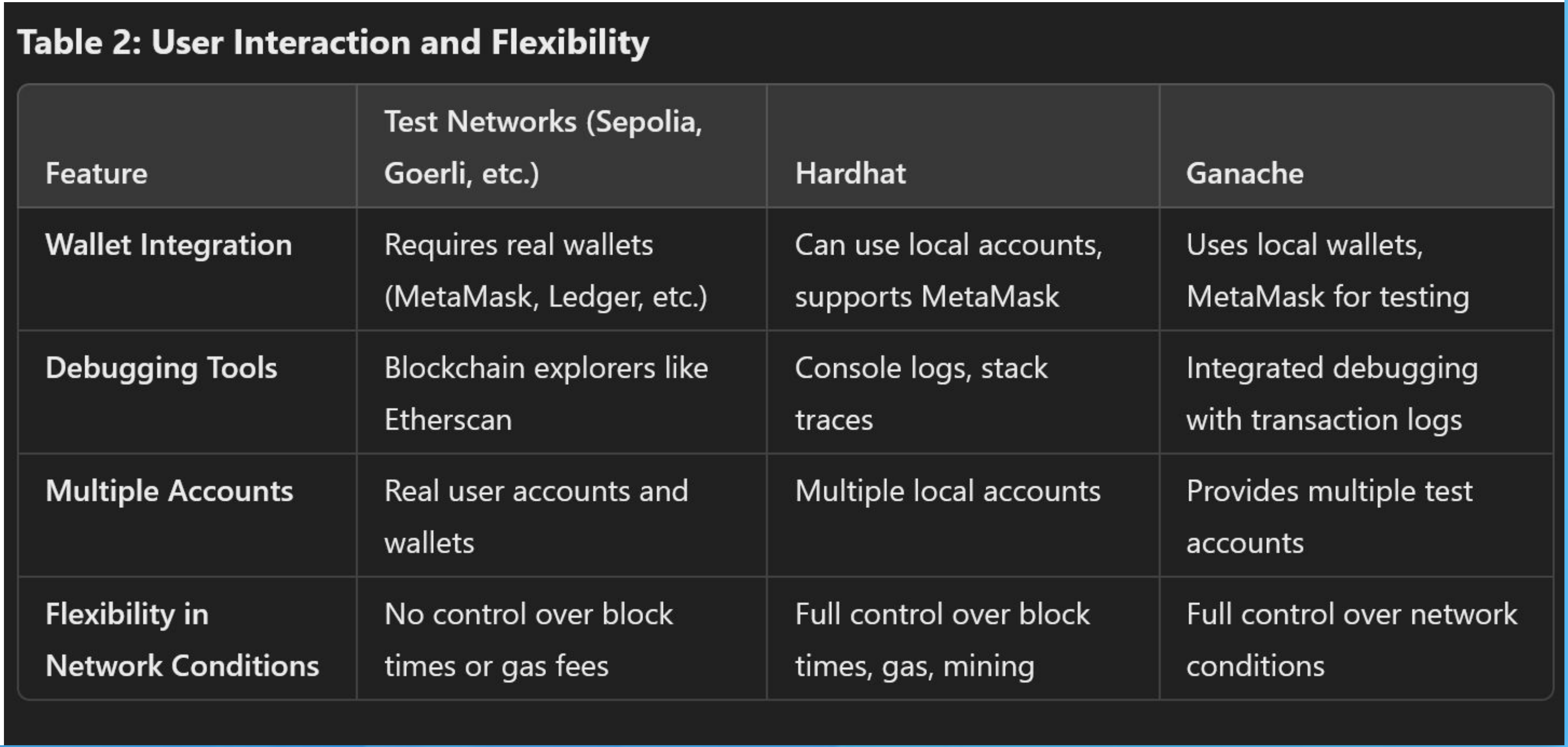
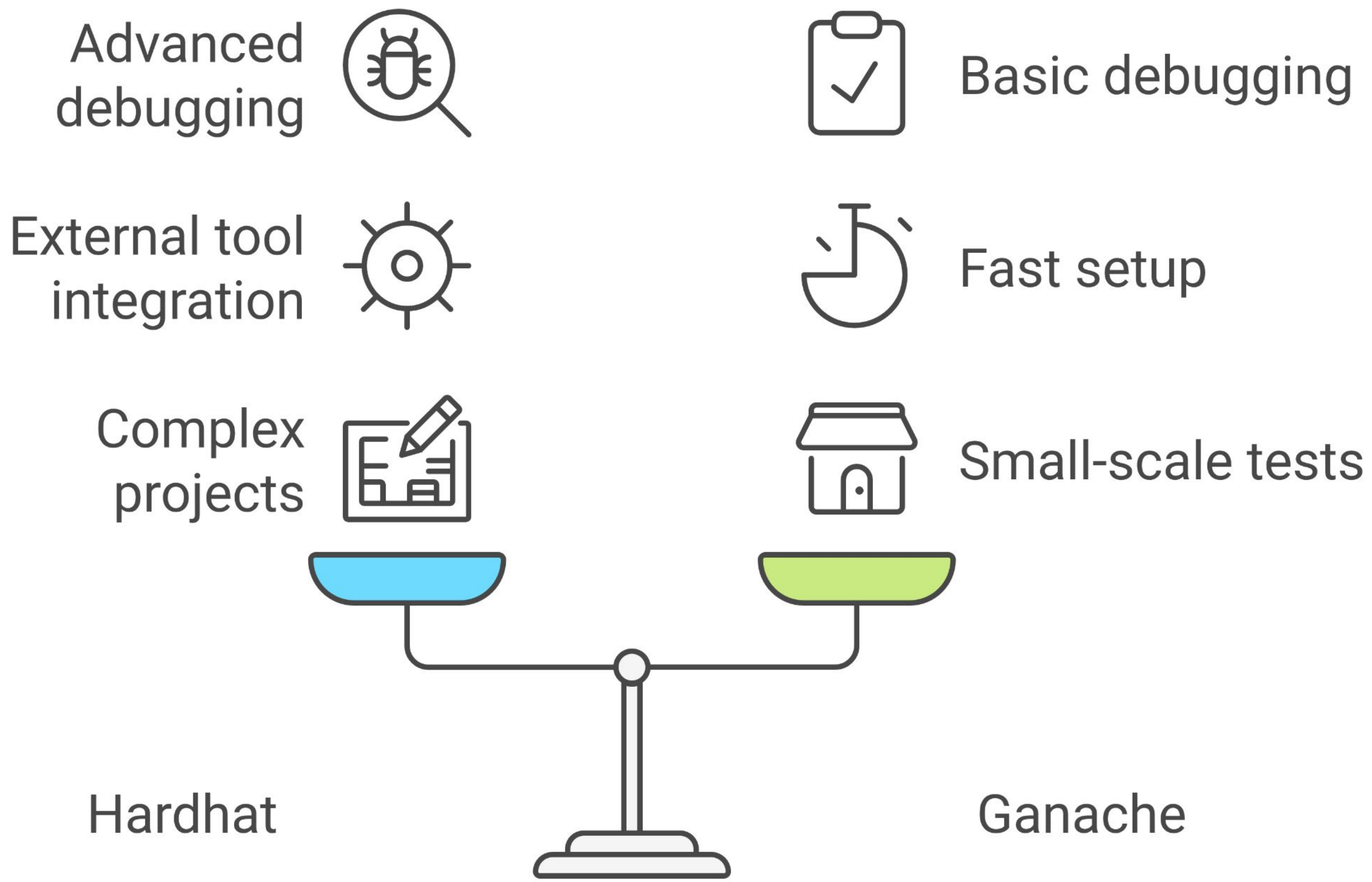


Table 2: User Interaction and Flexibility

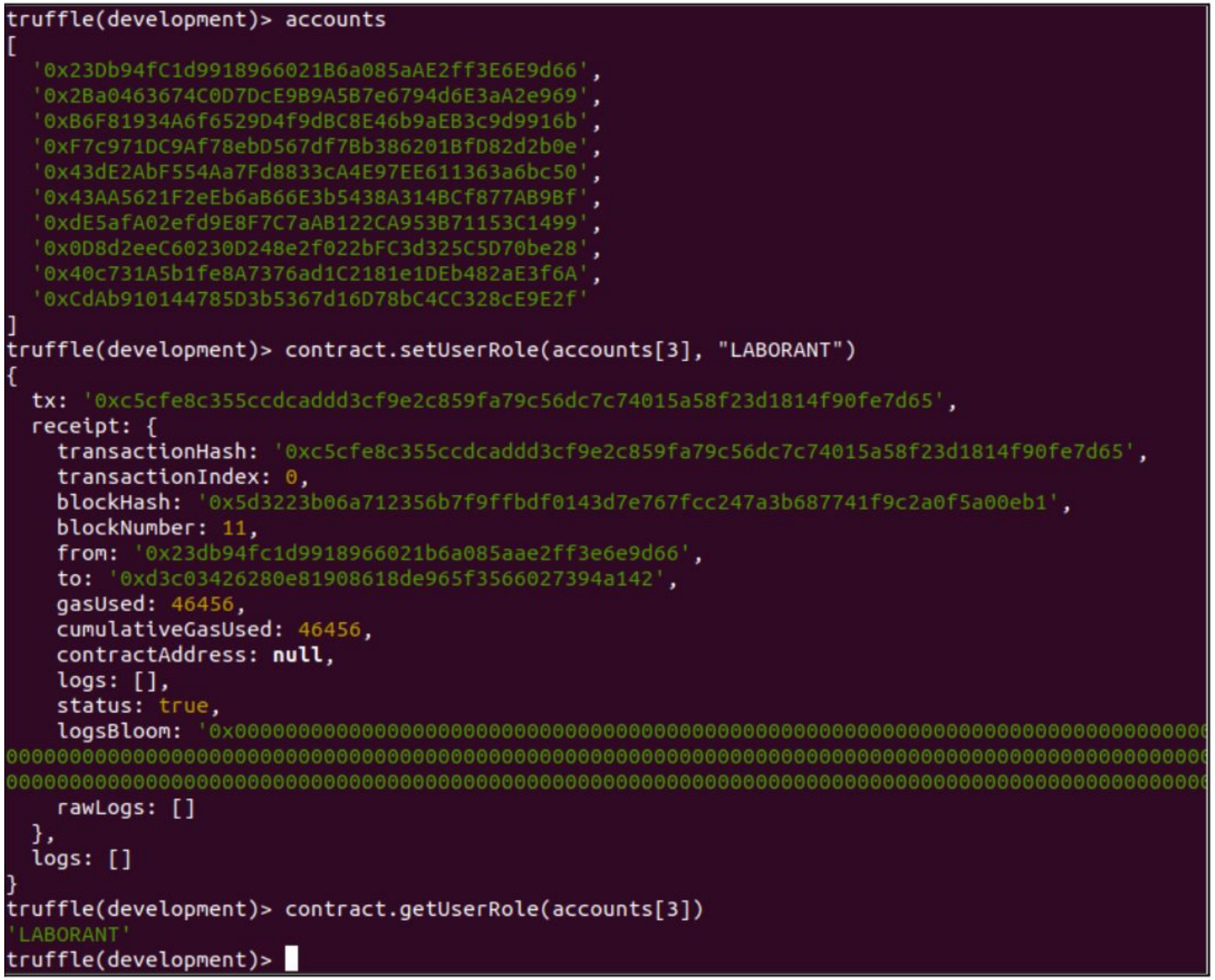
Feature	Test Networks (Sepolia, Goerli, etc.)	Hardhat	Ganache
Wallet Integration	Requires real wallets (MetaMask, Ledger, etc.)	Can use local accounts, supports MetaMask	Uses local wallets, MetaMask for testing
Debugging Tools	Blockchain explorers like Etherscan	Console logs, stack traces	Integrated debugging with transaction logs
Multiple Accounts	Real user accounts and wallets	Multiple local accounts	Provides multiple test accounts
Flexibility in Network Conditions	No control over block times or gas fees	Full control over block times, gas, mining	Full control over network conditions

Hardhat or Ganache



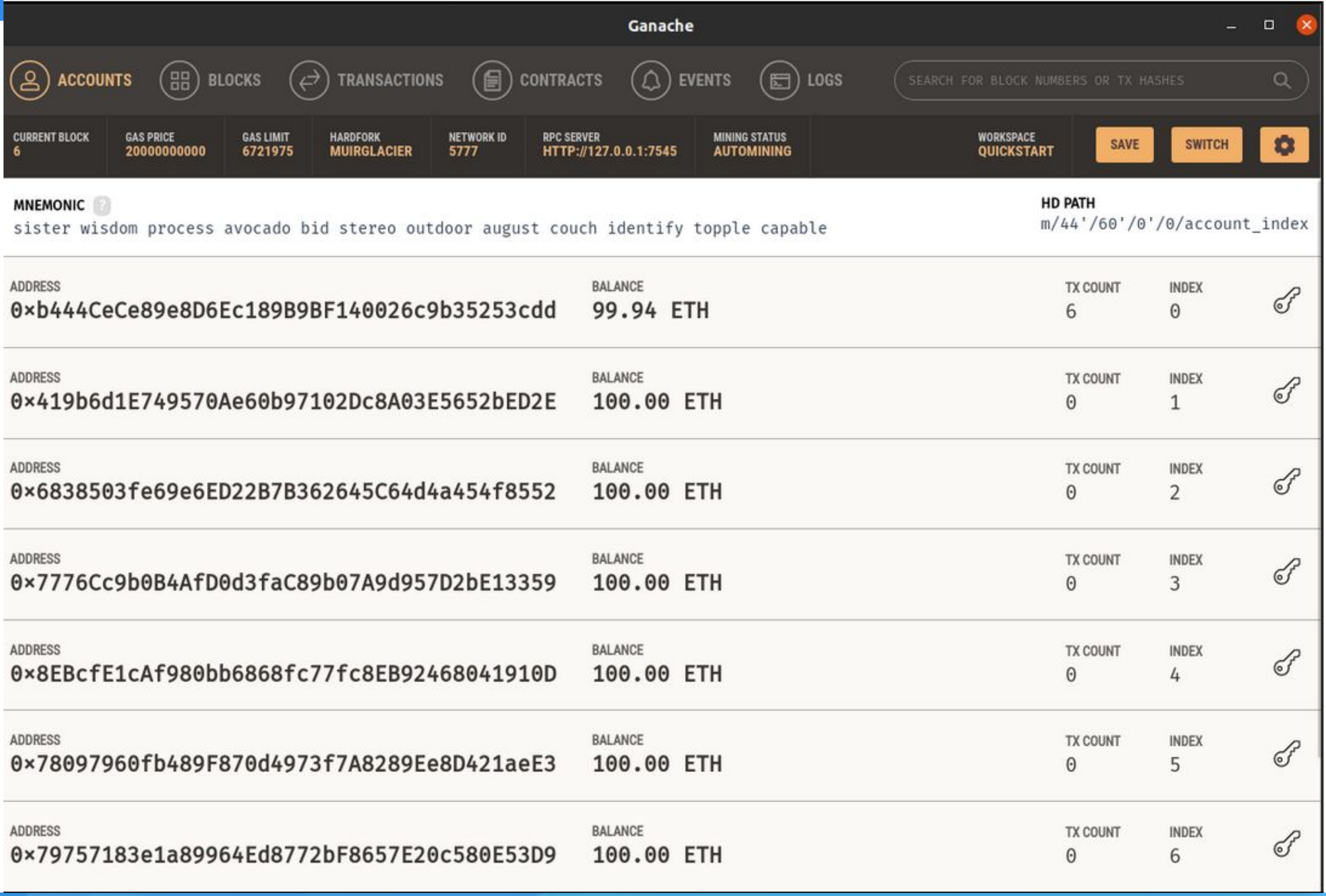
Choosing the right tool for blockchain development.

Truffle and Ganache



```
truffle(development)> accounts
[
  '0x23Db94fC1d9918966021B6a085aAE2ff3E6E9d66',
  '0x2Ba0463674C0D7DcE9B9A5B7e6794d6E3aA2e969',
  '0xB6F81934A6f6529D4f9dBC8E46b9aEB3c9d9916b',
  '0xF7c971DC9Af78ebD567df7Bb386201BfD82d2b0e',
  '0x43dE2AbF554Aa7Fd8833cA4E97EE611363a6bc50',
  '0x43AA5621F2eEb6aB66E3b5438A314BCf877AB9Bf',
  '0xdE5afA02efd9E8F7C7aAB122CA953B71153C1499',
  '0x0D8d2eeC60230D248e2f022bFC3d325C5D70be28',
  '0x40c731A5b1fe8A7376ad1C2181e1DEb482aE3f6A',
  '0xCdAb910144785D3b5367d16D78bC4CC328cE9E2f'
]
truffle(development)> contract.setUserRole(accounts[3], "LABORANT")
{
  tx: '0xc5cfe8c355ccdcaddd3cf9e2c859fa79c56dc7c74015a58f23d1814f90fe7d65',
  receipt: {
    transactionHash: '0xc5cfe8c355ccdcaddd3cf9e2c859fa79c56dc7c74015a58f23d1814f90fe7d65',
    transactionIndex: 0,
    blockHash: '0x5d3223b06a712356b7f9ffbdf0143d7e767fcc247a3b687741f9c2a0f5a00eb1',
    blockNumber: 11,
    from: '0x23db94fc1d9918966021b6a085aae2ff3e6e9d66',
    to: '0xd3c03426280e81908618de965f3566027394a142',
    gasUsed: 46456,
    cumulativeGasUsed: 46456,
    contractAddress: null,
    logs: [],
    status: true,
    logsBloom: '0x00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    rawLogs: []
  },
  logs: []
}
truffle(development)> contract.getUserRole(accounts[3])
'LABORANT'
truffle(development)> 
```

Truffle and Ganache



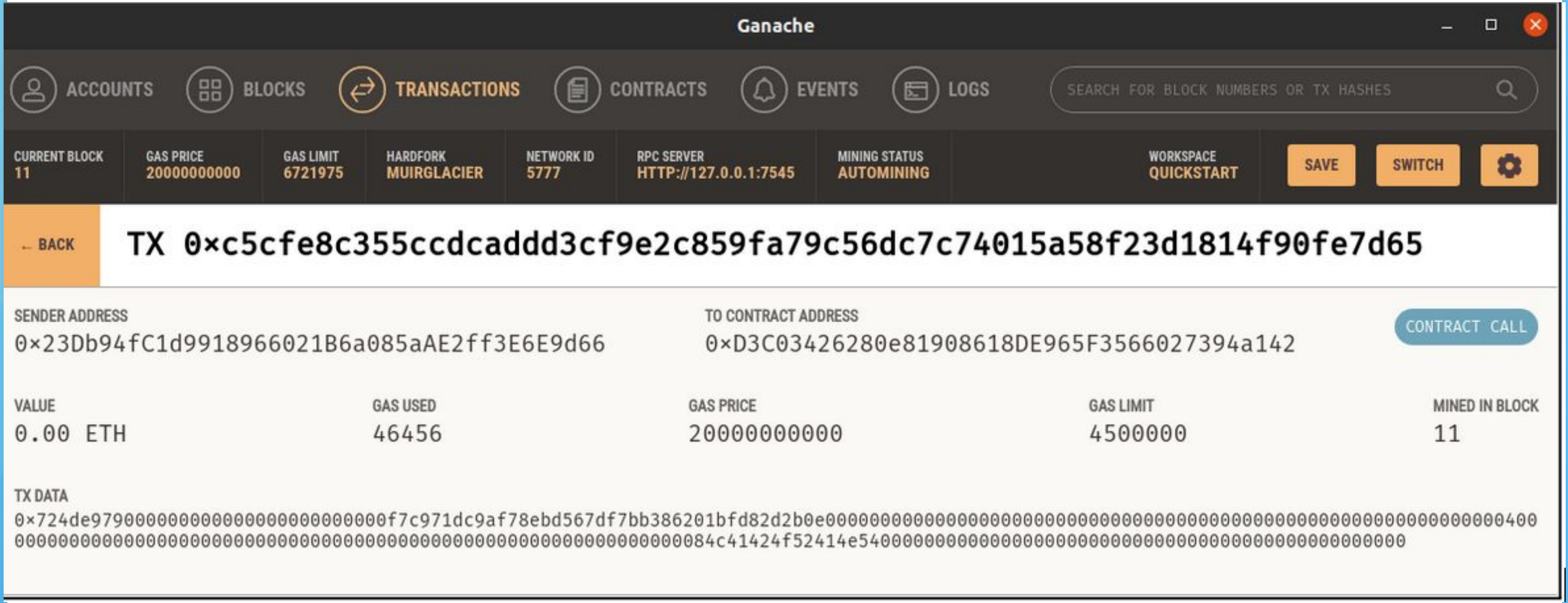
CURRENT BLOCK 6
GAS PRICE 20000000000
GAS LIMIT 6721975
HARDFORK MUIRGLACIER
NETWORK ID 5777
RPC SERVER HTTP://127.0.0.1:7545
MINING STATUS AUTOMINING
WORKSPACE QUICKSTART

MNEMONIC
sister wisdom process avocado bid stereo outdoor august couch identify topple capable

HD PATH
m/44'/60'/0'/0/account_index

ADDRESS	BALANCE	TX COUNT	INDEX
0×b444CeCe89e8D6Ec189B9BF140026c9b35253cdd	99.94 ETH	6	0
0×419b6d1E749570Ae60b97102Dc8A03E5652bED2E	100.00 ETH	0	1
0×6838503fe69e6ED22B7B362645C64d4a454f8552	100.00 ETH	0	2
0×7776Cc9b0B4AfD0d3faC89b07A9d957D2bE13359	100.00 ETH	0	3
0×8EBcfE1cAf980bb6868fc77fc8EB92468041910D	100.00 ETH	0	4
0×78097960fb489F870d4973f7A8289Ee8D421aeE3	100.00 ETH	0	5
0×79757183e1a89964Ed8772bF8657E20c580E53D9	100.00 ETH	0	6

Truffle and Ganache



CURRENT BLOCK 11
GAS PRICE 20000000000
GAS LIMIT 6721975
HARDFORK MUIRGLACIER
NETWORK ID 5777
RPC SERVER HTTP://127.0.0.1:7545
MINING STATUS AUTOMINING
WORKSPACE QUICKSTART

TX 0xc5cfe8c355ccdcaddd3cf9e2c859fa79c56dc7c74015a58f23d1814f90fe7d65

SENDER ADDRESS
0x23Db94fC1d9918966021B6a085aAE2ff3E6E9d66

TO CONTRACT ADDRESS
0xD3C03426280e81908618DE965F3566027394a142

VALUE
0.00 ETH

GAS USED
46456

GAS PRICE
20000000000

GAS LIMIT
4500000

MINED IN BLOCK
11

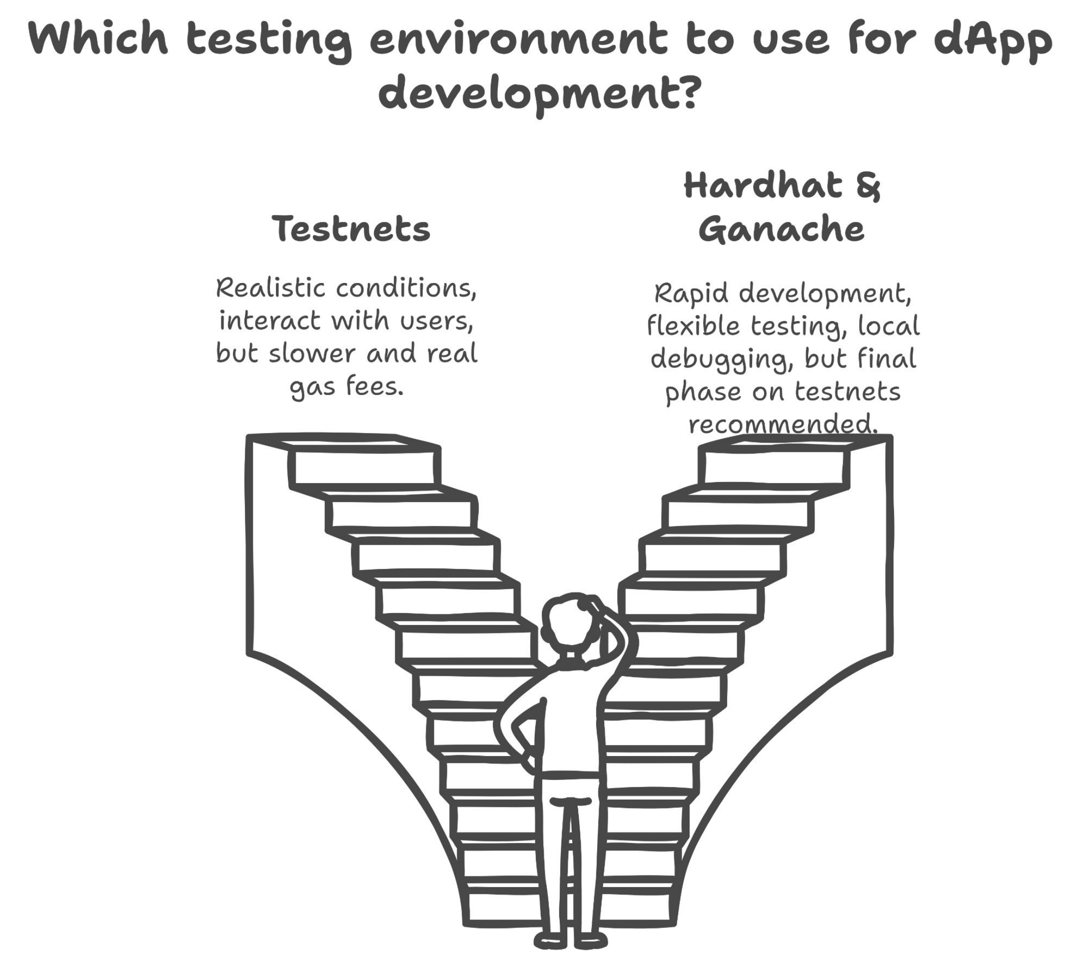
TX DATA

```
0x724de979000000000000000000000000f7c971dc9af78ebd567df7bb386201bfd82d2b0e00000000000000000000000000000000000000000000000000000000000000400
000000000000000000000000000000000000000000000000000000000000000084c41424f52414e54000000000000000000000000000000000000000000000000
```

Section conclusion

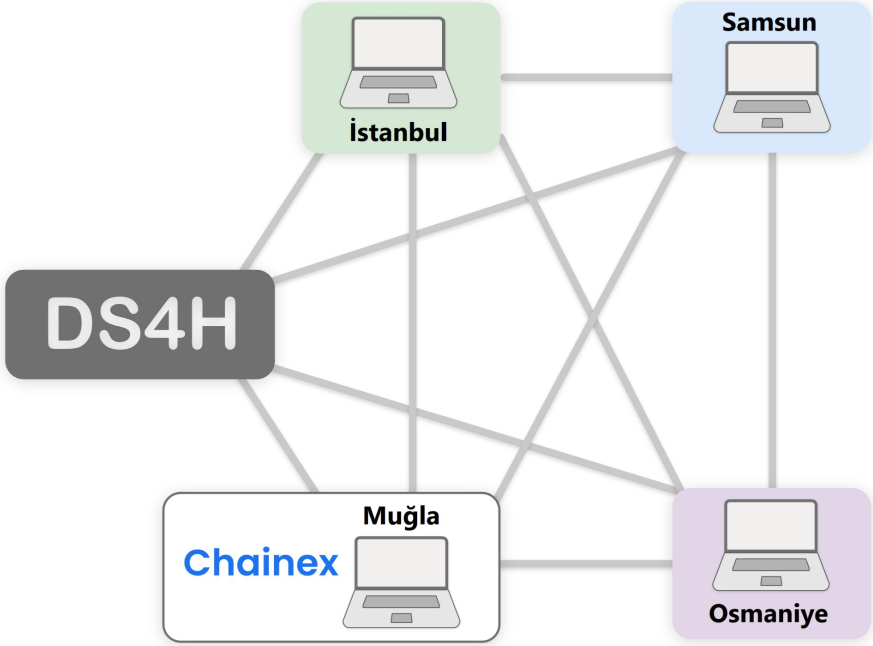
We need to use each, but in different times:

- Testnets: If you want to test your DApps in a mainnet-like environment and interact with other users or developers, testnets are ideal. They offer realistic conditions, but can be slower and gas fees are real.
- Hardhat & Ganache: Best for rapid development, flexible testing, and local debugging. However, it’s usually a better strategy to do a final phase on testnets before moving to the mainnet environment.



DS4H Test Network

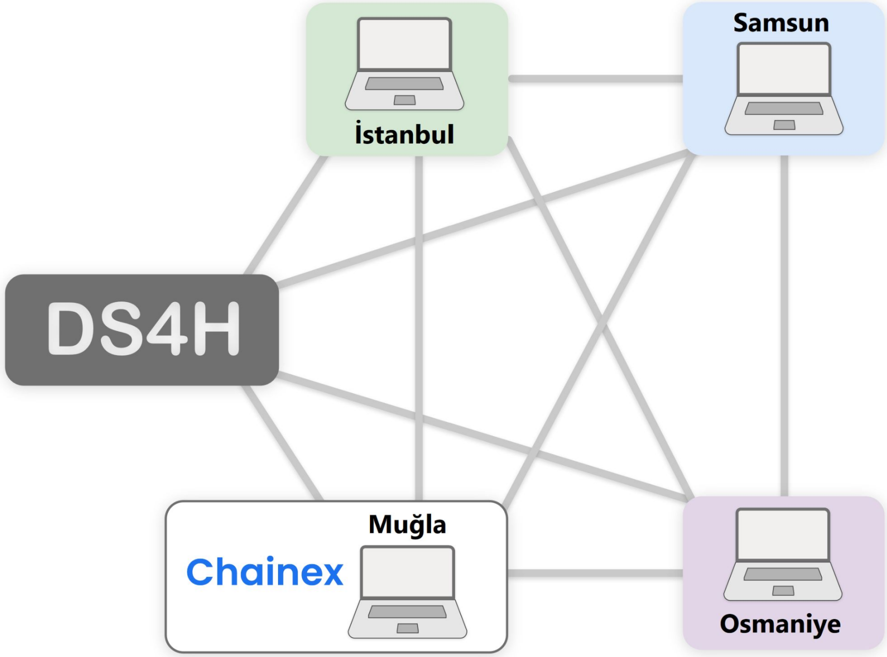
- DS4H blockchain research network [3]
  - Quorum framework is selected as the main blockchain platform,
  - a private/permissioned blockchain with low energy consumption.
  - Docker container technology
  - ordinary virtual machines (single CPU, 8 GB RAM, 256 GB disk) serve as nodes.
  - QBFT was chosen as the consensus protocol
  - Block time interval was adjusted to 1 second (default is 5 seconds), and the empty block interval was increased to 600 seconds (default is 60 seconds). Log verbosity was reduced from level 5 to 3, which was sufficient for detecting errors.



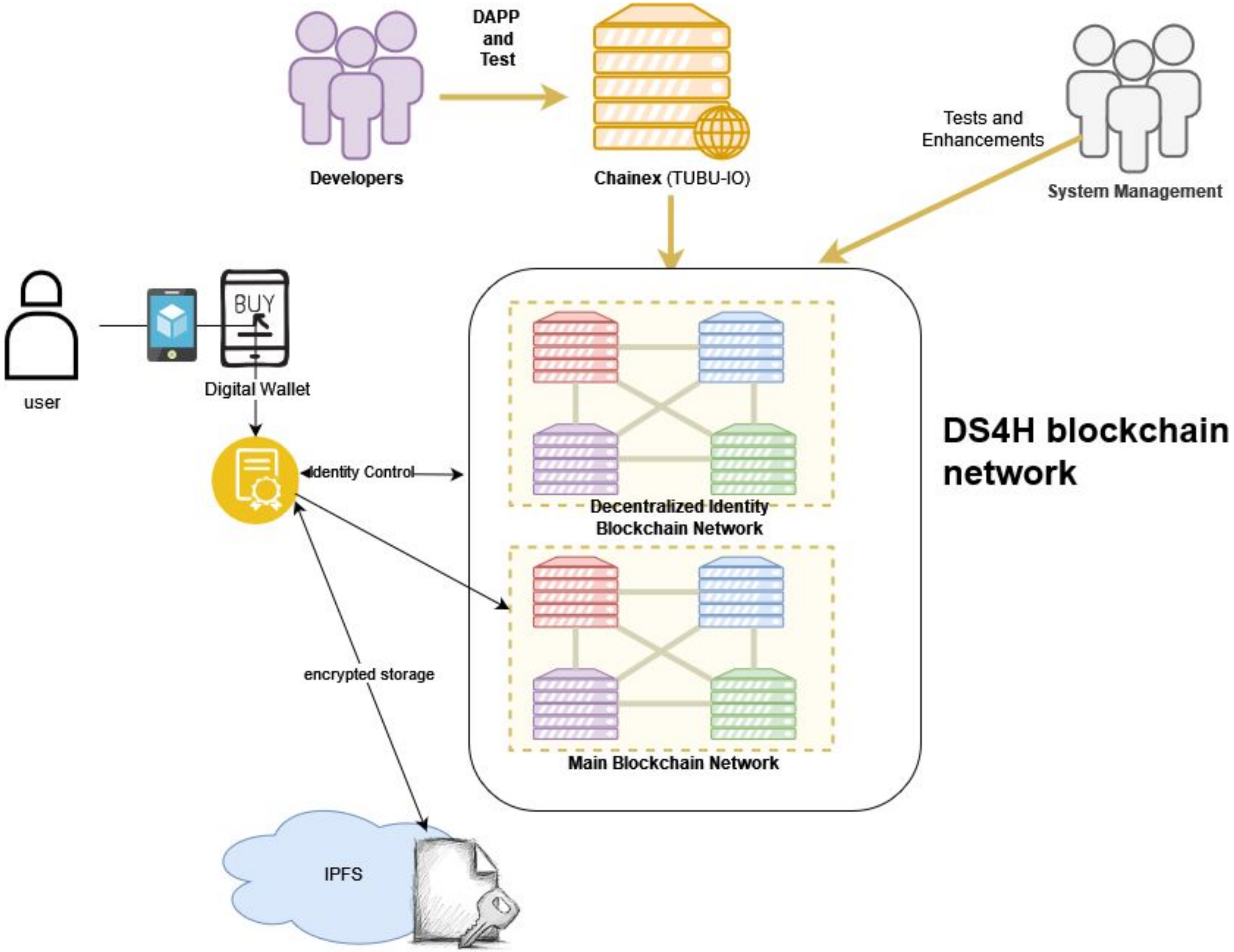
DS4H Test Network

New Installations are on the way

- Reinstalling the deployment (On Process)
- Add new services (On Process)
  - Hyperledger Indy - Decentrlized identity
  - Hyperledger Aries - wallet integration
  - IPFS - Distributed File Storage



Our Design - DS4H NEW



◉ DApps and Web 3.0…

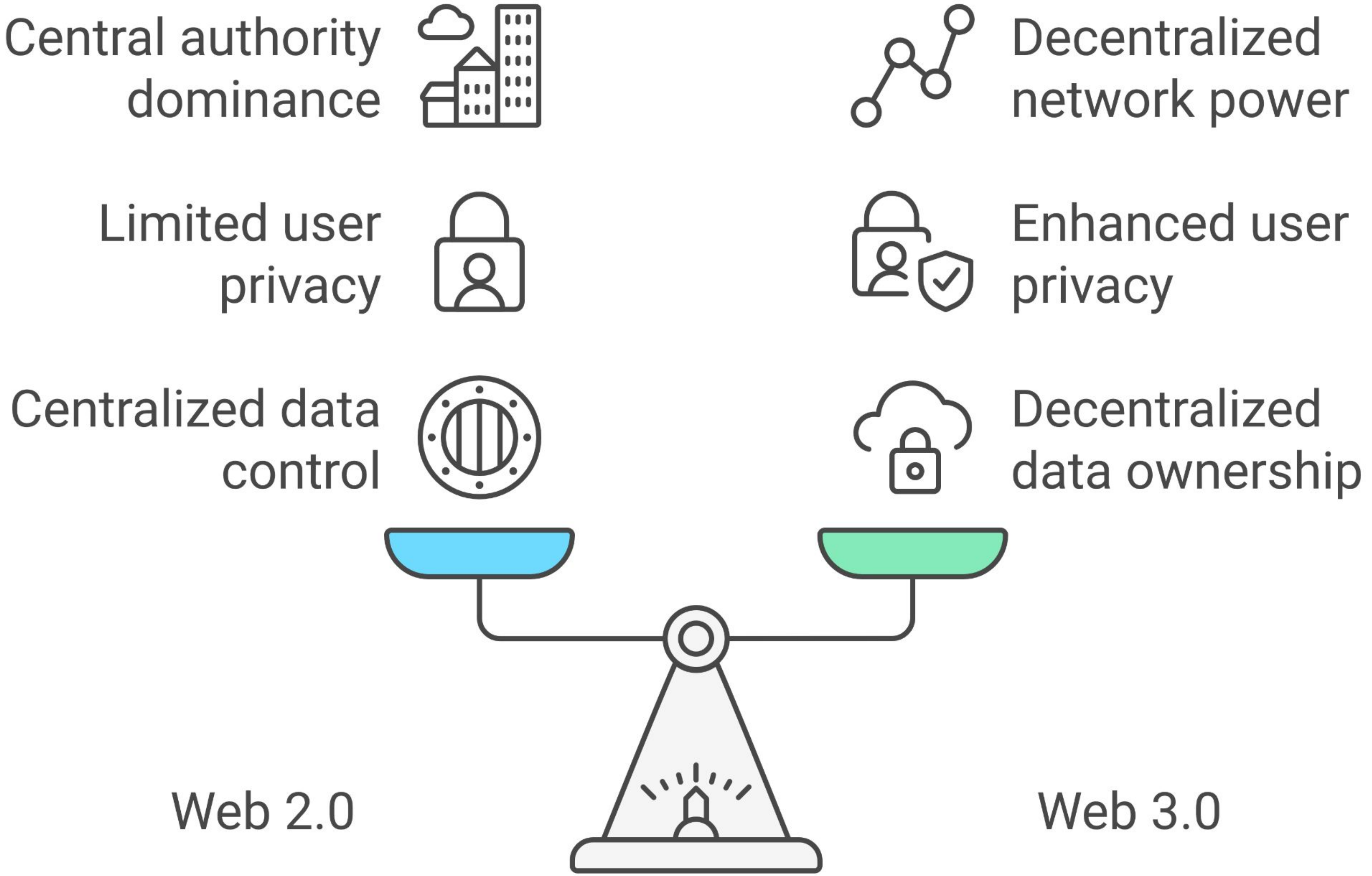
FIVE

DApps and Web 3.0 development …

● ● ● ●

Introduction to Web 3.0

- Web 3.0 refers to the next generation of the internet, where data and services are decentralized, users have control over their own data, and transparency is ensured through blockchain technology.
- Core Concepts:
  - Decentralization: No central authority controls the data or services.
  - User Ownership: Users retain ownership of their data and digital assets.
  - Transparency: Actions and data are publicly verifiable on blockchains.



Central authority dominance

Limited user privacy

Centralized data control

Web 2.0

Decentralized network power

Enhanced user privacy

Decentralized data ownership

Web 3.0

Comparing data control and privacy in Web 2.0 and Web 3.0.

Core Principles of Web 3.0

- Decentralization:
    - Data is stored across a distributed network, not controlled by a single entity.
    - Blockchain ensures that all participants have access to the same information.
- User Ownership:
    - Users control their own data, digital identity, and assets using cryptographic wallets (e.g., MetaMask).
- Transparency:
    - Actions on decentralized applications (DApps) are verifiable by anyone.
    - Trust is built into the system through smart contracts.

Decentralized Applications (DApps)

- DApps are applications that run on a decentralized network, using smart contracts to operate without a central server or intermediary.
- How DApps Work:
    - DApps are powered by blockchain smart contracts that execute transactions and logic in a trustless manner.
    - Users interact with DApps through wallets that provide cryptographic identity and control.

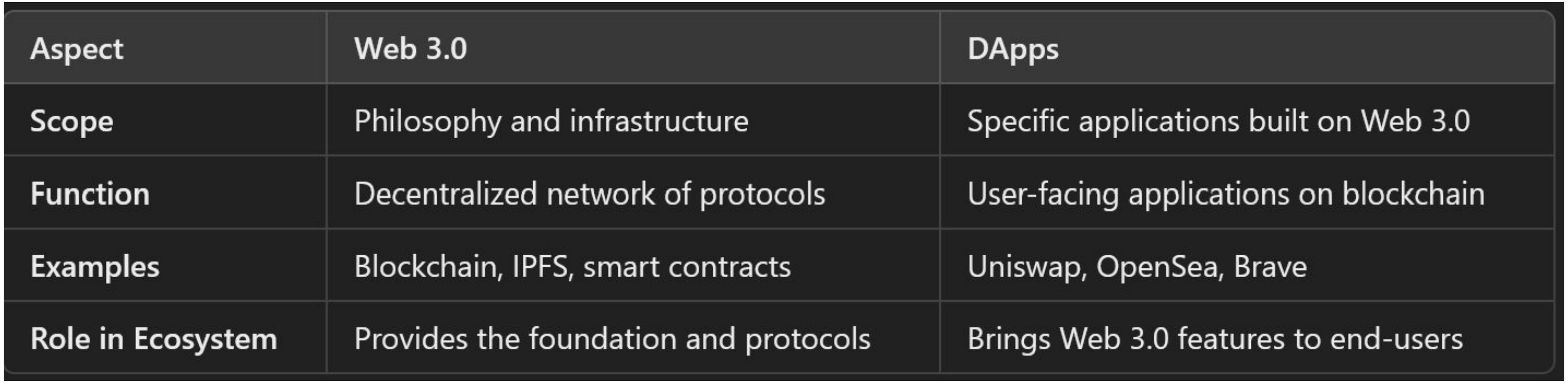
Decentralized Applications (DApps)

Key Feature of DApps:

- **Autonomy:** DApps operate independently, once deployed they cannot be controlled by a single entity.
- **Transparency:** Code is open-source and verifiable, ensuring trust.
- **User Control:** Users hold control over their data and interactions with the app.

Web 3.0 and DApps

Web 3.0 is the decentralized platform, while DApps are applications on that platform designed to make Web 3.0 functionality accessible and valuable to users.



Aspect	Web 3.0	DApps
Scope	Philosophy and infrastructure	Specific applications built on Web 3.0
Function	Decentralized network of protocols	User-facing applications on blockchain
Examples	Blockchain, IPFS, smart contracts	Uniswap, OpenSea, Brave
Role in Ecosystem	Provides the foundation and protocols	Brings Web 3.0 features to end-users

How DApps Leverage Web 3.0 Principles

DApps heavily rely on smart contracts to function, ensuring that the core principles of decentralization, transparency, and ownership are maintained.

- **Decentralization:** No single point of failure. DApps run on decentralized blockchain networks (e.g., Ethereum).
- **User Ownership:** Users own their digital assets and control their data via private keys.
- **Transparency:** Smart contract execution is public and verifiable, creating a transparent operating environment.

The Impact of Web 3.0 on Industries

- **Finance:** Decentralized finance (DeFi) is transforming traditional banking and investment through DApps that remove intermediaries.
- **Gaming:** Ownership of in-game assets through NFTs allows users to truly own and trade digital goods.
- **Social Media:** Web 3.0 enables user-owned social platforms where content and privacy are controlled by the user, not a corporation.
- **Healthcare:** Blockchain technology is enhancing data privacy and access control, giving patients ownership over their health records.

Key Takeaway: Web 3.0 is reshaping industries by decentralizing control and empowering users with ownership and transparency.

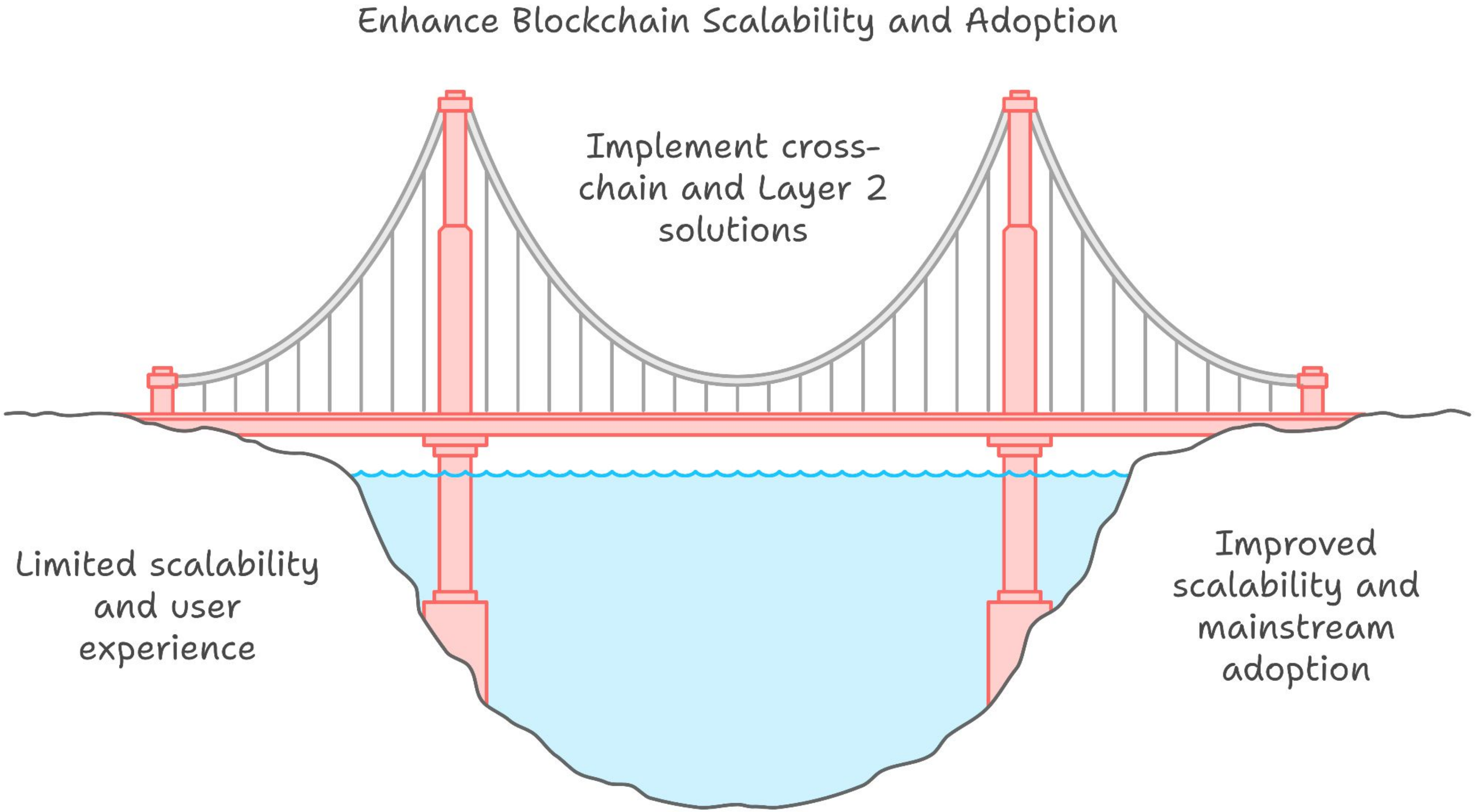
Real-World Examples of Web 3.0 DApps

- Finance (DeFi):
  - Uniswap: A decentralized exchange where users can trade tokens without an intermediary.
  - Aave: A decentralized lending and borrowing platform that eliminates the need for banks.
- Gaming:
  - Axie Infinity: A play-to-earn game where users own and trade in-game assets (NFTs).
- Social Networks:
  - Minds: A decentralized social media platform where users control their content and earn rewards for engagement.
- Supply Chain:
  - VeChain: Uses blockchain to enhance supply chain transparency and traceability.

Challenges and the Future of Web 3.0

- *Challenges:*
  - ***Scalability:*** *Current blockchain networks can struggle with high transaction volumes.*
  - ***User Experience:*** *DApps often have complex interfaces, limiting mainstream adoption.*
  - ***Regulation:*** *Governments are still catching up with how to regulate decentralized systems.*
- *Future Prospects:*
  - ***Interoperability:*** *Cross-chain solutions will allow different blockchain ecosystems to communicate and work together.*
  - ***Layer 2 Scaling Solutions:*** *Technologies like Optimism and Polygon are improving the scalability of blockchains.*
  - ***Increased Adoption:*** *As the user experience improves, more industries will adopt Web 3.0 technologies.*

Challenges and the Future of Web 3.0



The Power of Web 3.0 and DApps

- **Web 3.0:** Represents a shift towards decentralized, user-controlled, and transparent internet services.
- **DApps:** Leverage these principles to create decentralized applications that operate without a central authority.
- **Smart Contracts:** Ensure deterministic and predictable behavior, key for building trustless and transparent systems.

Closing Thought:

Web 3.0 and DApps together form the foundation of a more open, transparent, and user-empowered internet.

○ DApps implementation

SIX

web3.js and node.js

Basic Web3 Architecture

- **Smart Contract:** Deployed on the Ethereum blockchain (Ganache/HardHat for local testing).
- **Web3.js or Ethers.js:** JavaScript libraries for interacting with Ethereum.
- **Front-End:** Built with HTML, CSS, JavaScript, and integrated with Web3.js or Ethers.js.
- **MetaMask:** A browser extension used to sign and send transactions to Ethereum networks (mainnet, testnets, or Ganache/HardHat).

Node.js Web3.js

- **Web3.js:** is a JavaScript library that interacts with the Ethereum blockchain, enabling the creation of decentralized applications (DApps) and smart contracts.
- **Node.js:** Node.js is a runtime that enables JavaScript to be used server-side, allowing developers to build fast, scalable network applications.
- **Node.js Advantages:** Non-blocking I/O, scalability, and ecosystem of libraries (e.g., Express, Axios).
- **Web3.js Role:** Adds blockchain functionality, allowing you to perform transactions, check balances, and interact with smart contracts.

What Node.js Offers for Web3 DApps Development

- Server-Side Automation:
  - Automate smart contract interactions (e.g., recurring tasks, automated payouts).
- Backend-Blockchain Integration:
  - Build a middle layer between the blockchain and your front-end or external systems (like databases).
- Security:
  - Manage private keys and sensitive data on the back-end, keeping them away from the client-side.
- Handling Complex Logic:
  - Execute complex logic, interact with multiple blockchains, and aggregate data from the blockchain.

Writing a Basic Node.js Script for Web3 Interactions

Connecting to Ethereum with Web3.js:

```
const Web3 = require('web3');
const web3 = new Web3('http://127.0.0.1:7545'); // Ganache URL

const contractAddress = 'YOUR_CONTRACT_ADDRESS';
const contractABI = [/* ABI from compiled contract */];

const simpleStorage = new web3.eth.Contract(contractABI, contractAddress);
```

Setting and Getting Values (Node.js Script):

```
async function interactWithContract() {
    const accounts = await web3.eth.getAccounts();

    // Set a value
    await simpleStorage.methods.set(100).send({ from: accounts[0] });

    // Get the value
    const value = await simpleStorage.methods.get().call();
    console.log('Stored Value:', value);
}

interactWithContract();
```

What Node.js Offers for Web3 DApps Development

- Server-Side Automation:
  - Automate smart contract interactions (e.g., recurring tasks, automated payouts).
- Backend-Blockchain Integration:
  - Build a middle layer between the blockchain and your front-end or external systems (like databases).
- Security:
  - Manage private keys and sensitive data on the back-end, keeping them away from the client-side.
- Handling Complex Logic:
  - Execute complex logic, interact with multiple blockchains, and aggregate data from the blockchain.

Difference Between Browser and Node.js Approach

- Browser-Based Approach:
  - Users interact with the smart contract directly from the client-side (via MetaMask).
  - Private keys are handled by MetaMask, and interaction is limited to user-triggered events.
- Node.js Server-Based Approach:
  - You can handle contract interactions automatically on the server (e.g., scheduled tasks, automated payments).
  - Private keys can be securely stored and managed on the server instead of the client.
  - Easier to aggregate data from multiple sources (smart contracts, databases) before passing it to the front-end.

Building an API with Node.js and Express.js

```
const express = require('express');
const Web3 = require('web3');
const app = express();
const port = 3000;

const web3 = new Web3('http://127.0.0.1:7545');
const contractAddress = 'YOUR_CONTRACT_ADDRESS';
const contractABI = [/* ABI */];
const simpleStorage = new web3.eth.Contract(contractABI, contractAddress);

app.get('/value', async (req, res) => {
    const value = await simpleStorage.methods.get().call();
    res.send({ storedValue: value });
});

app.listen(port, () => {
    console.log(`Server is running on http://localhost:${port}`);
});
```

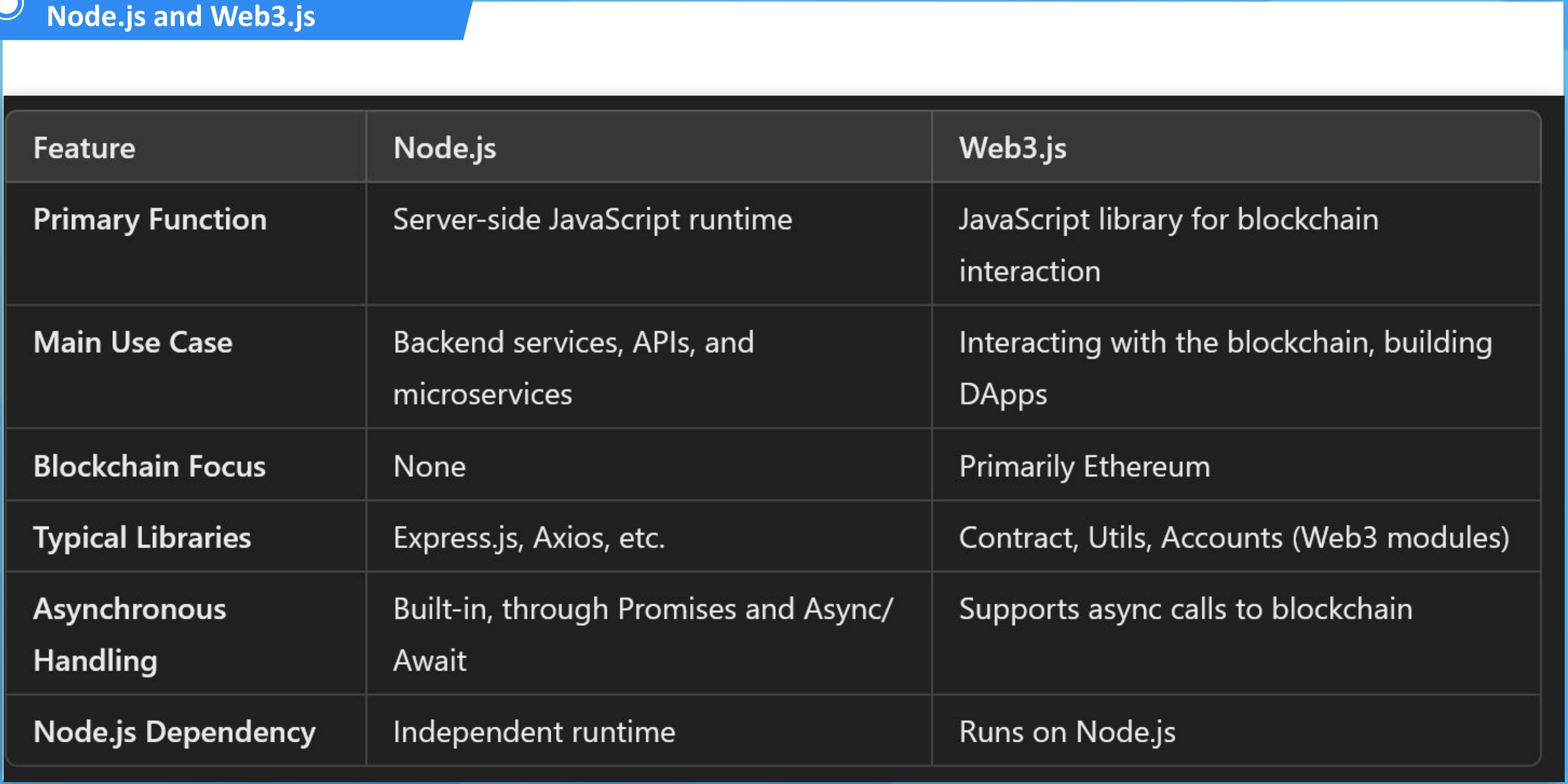
Build a Simple API
by which users can get the value stored in the contract
by making a GET request to the API.

Managing Security with Node.js

```
const privateKey = process.env.PRIVATE_KEY;  // Store in environment variables
const account = web3.eth.accounts.privateKeyToAccount(privateKey);

async function signTransaction() {
     const tx = {
     to: contractAddress,
     data: simpleStorage.methods.set(123).encodeABI(),
     gas: 2000000,
     };

     const signedTx = await account.signTransaction(tx);
     const receipt = await web3.eth.sendSignedTransaction(signedTx.rawTransaction);
     console.log('Transaction receipt:', receipt);
}

signTransaction();
```

Using a Private Key to Sign Transactions
Store private keys securely on the server, using Node.js's file system (with appropriate encryption) or environment variables.

Key Differences Between Node.js and Web3.js



Feature	Node.js	Web3.js
Primary Function	Server-side JavaScript runtime	JavaScript library for blockchain interaction
Main Use Case	Backend services, APIs, and microservices	Interacting with the blockchain, building DApps
Blockchain Focus	None	Primarily Ethereum
Typical Libraries	Express.js, Axios, etc.	Contract, Utils, Accounts (Web3 modules)
Asynchronous Handling	Built-in, through Promises and Async/Await	Supports async calls to blockchain
Node.js Dependency	Independent runtime	Runs on Node.js

Node.js + Web3.js

- Node.js provides the necessary backend infrastructure while Web3.js acts as the bridge between the application and the blockchain.
- Node.js is used for handling server logic, while Web3.js performs blockchain transactions.
- Node.js is better for
  - server-side operations like managing private keys, automating contract interactions, and building DApps infrastructure.
  - Enables asynchronous execution, which is useful for handling multiple blockchain requests.
- ***Combination Use Case:*** Create a secure, decentralized backend that performs blockchain operations.

Security Measures in Node.js + Web3.js Integration

Security is Essential. When working with blockchain back-ends, securing private keys and transaction data is crucial to prevent unauthorized access and maintain data integrity. Node.js can add an extra security layer by handling sensitive operations server-side, away from client access. Core Security Practices:

- Private Key Management:
  - Avoid hardcoding private keys in code. Instead, use environment variables or secure storage solutions (e.g., AWS Secrets Manager, HashiCorp Vault).
  - Use Node.js's ***dotenv*** package to manage private keys safely in ***.env*** files.
- Transaction Signing:
  - Use server-side transaction signing to keep private keys secure and prevent exposure to the front-end.
  - Web3.js provides signing methods like ***signTransaction*** to securely sign and send transactions on behalf of the user.

Practical Sec. Ex.: Environment Variables & Transaction Signing

Set up .env file:

```
PRIVATE_KEY=your_private_key_here
```

Access the key securely in code:

```
require('dotenv').config();
const privateKey = process.env.PRIVATE_KEY;
const account = web3.eth.accounts.privateKeyToAccount(privateKey);
```

Signing Transactions Server-Side:

```
async function signAndSendTransaction(toAddress, amount) {
    const tx = {
    to: toAddress,
    value: web3.utils.toWei(amount, 'ether'),
    gas: 2000000,
    };
    const signedTx = await account.signTransaction(tx);
    const receipt = await web3.eth.sendSignedTransaction(signedTx.rawTransaction);
    console.log("Transaction receipt:", receipt);
}
```

This ensures private keys are never exposed to the client and minimizes risk

VS code versus remix

Feature	VS Code	Remix
Purpose	General-purpose code editor with Web 3.0 plugins	Browser-based IDE for Ethereum smart contracts
Best For	Full-stack DApp and backend development	Smart contract creation, testing, and deployment
Environment	Desktop application	Web-based

Node.js + Web3.js

- You install both on your development environment
  - Node.js - https://nodejs.org/en/download
- For full stack Web 3.0 development any environment which you are familiar such as:
  - VSCode (https://code.visualstudio.com/download)

Recommendation	VS Code	Remix
Ideal Use Case	Full-stack Web 3.0 development	Dedicated smart contract development
Best For	Advanced developers looking for flexibility	Beginners or developers focused on Ethereum contracts

Node.js + Web3.js

- You install both on your development environment
  - Node.js - https://nodejs.org/en/download
- For full stack Web 3.0 development any environment which you are familiar such as:
  - Eclipse is best for large, enterprise-level projects, especially if Web 3.0 is only a small component.
  - VSCode is ideal for full-stack Web 3.0 and DApps development due to its flexibility, plugin ecosystem, and strong community support. (https://code.visualstudio.com/download)

Recommendation	VS Code	Remix
Ideal Use Case	Full-stack Web 3.0 development	Dedicated smart contract development
Best For	Advanced developers looking for flexibility	Beginners or developers focused on Ethereum contracts

Connecting to the Blockchain Using Web3.js

- Connecting Locally (Ganache): Use Ganache to set up a local Ethereum blockchain for testing.

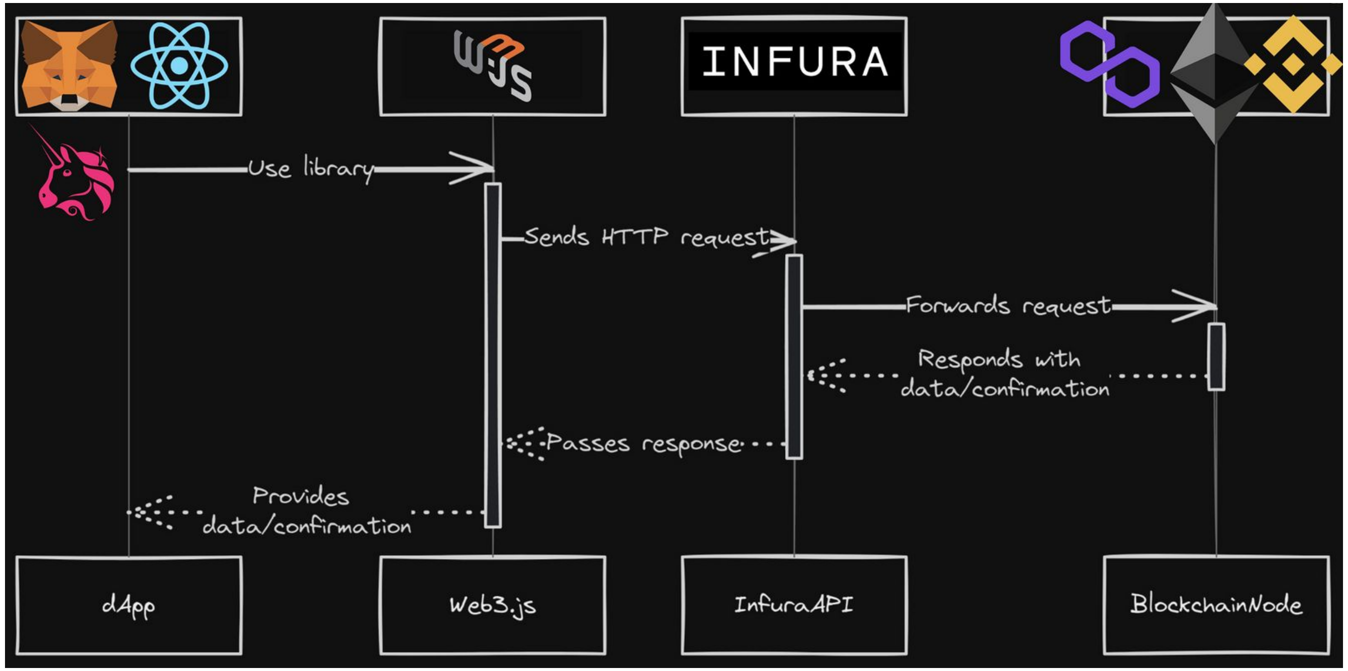
  `Code snippet: const Web3 = require('web3'); const web3 = new Web3('http://localhost:7545');`

- Connecting Remotely (Infura): Use Infura (or alike) to connect to the Ethereum mainnet or testnet.

  `Code snippet: const web3 = new Web3(new Web3.providers.HttpProvider('<INFURA_URL>'));`

Web3.js and INFURA

- Platforms that can be used providing instant access to Ethereum (or alike EVMs) and IPFS network:
  - INFURA, Alchemy, QuickNode, Moralis
- INFURA is recommended as it enables easy blockchain connectivity for Node.js applications with Web3.js.
- The system with such a platform will work as follows [4]:



Setup and implementation details can be reached at reference [4].

Connecting the DApps to Ethereum

In the app.js file, initialize Web3 and connect to the smart contract.

```
// Connect to MetaMask's provider
if (typeof window.ethereum !== 'undefined') {
    const web3 = new Web3(window.ethereum);
    await window.ethereum.request({ method: 'eth_requestAccounts'
});
} else {
    console.error("Please install MetaMask!");
}

// Get the contract ABI and address
const contractAddress = 'YOUR_CONTRACT_ADDRESS';
const contractABI = [/* ABI from compiled contract */];
const simpleStorage = new web3.eth.Contract(contractABI,
contractAddress);
```

HTML Structure for DApps Interface

Basic HTML file:

```
<!DOCTYPE html>
<html>
<head>
     <title>Web3.0 DApps</title>
</head>
<body>
     <h1>Simple Storage DApps</h1>

     <label for="inputValue">Set Value: </label>
     <input type="number" id="inputValue">
     <button onclick="setValue()">Set Value</button>

     <h3>Stored Value: <span id="storedValue"></span></h3>
     <button onclick="getValue()">Get Value</button>

     <script src="https://cdn.jsdelivr.net/npm/web3/dist/web3.min.js"></script>
     <script src="app.js"></script>
</body>
</html>
```

Interacting with Smart Contracts via Web Interface

Add JavaScript functions to interact with the smart contract.

```
async function setValue() {
    const value = document.getElementById('inputValue').value;
    const accounts = await web3.eth.getAccounts();
    await simpleStorage.methods.set(value).send({ from:
accounts[0] });
}


async function getValue() {
    const value = await simpleStorage.methods.get().call();
    document.getElementById('storedValue').innerText = value;
}
```

This allows users to set and retrieve values from the blockchain via the DApps.

Sending a Transaction Using Web3.js

Code for sending Ether through Infura’s Ethereum endpoint.

Transaction details are processed via Infura, reducing node management overhead.

```
const sendTransaction = async () => {
  const accounts = await web3.eth.getAccounts();
  await web3.eth.sendTransaction({
    from: accounts[0],
    to: 'recipient_address',
    value: web3.utils.toWei('0.1', 'ether')
  });
};
```

Web3.js Functions

web3.js provides functions that you can easily use to interact with the blockchain. Such as:

- web3.eth.getBalance()
- web3. eth.getChainId()
- web3.eth.getGasPrice()
- web3.eth.getTransactionCount(),

Key Functions of web3-utils

- **Data Conversion:** Converts between common data formats, such as Ether to Wei (smallest unit of Ether) and vice versa, making it easier to handle currency values.

  Functions: toWei, fromWei, toHex, hexToUtf8.
- **Hashing:** Provides secure hashing functions, such as sha3 and keccak256, to generate cryptographic hashes of strings or values, commonly used in transactions and smart contract verifications.
- **Hexadecimal and Big Number Utilities:**
  - hexToNumber, numberToHex, and other utilities to work with Ethereum's hexadecimal data.
  - Big number support to handle blockchain integers and floating points accurately.

Why Use web3-utils?

- Efficiency: Simplifies complex data conversions and cryptographic operations, reducing the need for additional libraries.
- Reliability: Ensures consistency when working with Ethereum’s native formats and data structures.

Practical Application Examples:

- Smart Contract Interactions: Converting user inputs to the correct format for contract calls.
- Frontend DApps Development: Ensuring data matches blockchain requirements in the UI, e.g., converting values for display.
- Security & Validation: Using hashing to validate and secure transactions.

Common problems & Debugging

- **Common Problems:**
    - MetaMask not connected: Ensure MetaMask is properly connected to the local blockchain network (such as Ganache)
    - Contract Address Mismatch: Verify that the correct contract address is used in the front-end.
    - Gas Errors: Ensure you have sufficient gas in the accounts on the llocal blockchain network
- **Debugging:**
    - Use browser’s Developer Tools (F12) to inspect errors in the JavaScript console.
    - Use Ganache’s transaction log to track contract interactions.

INFURA and other EVMs

- You can interact with other EVM-compatible blockchains such as Polygon, Binance Smart Chain, Avalanche, etc.
- Usage is straightforward [4] as:
  1. Go to Infura (or alike platform)
  2. Get the API endpoint for that specific network
  3. Initiate a Web3 provider with that endpoint in your code.

Additional Security Best Practices

- Limit Access to Sensitive Data:
  - Use server-side authentication and role-based access control (RBAC) to restrict access to blockchain operations.
- Use HTTPS and Secure RPC Providers:
  - Always connect to Ethereum nodes (e.g., Infura, Alchemy) over HTTPS.
  - Consider using services like Infura or Alchemy that handle node security and rate limiting.
- Monitoring & Auditing:
  - Set up logging and monitoring for transaction activity.
  - Regularly audit your smart contract interactions and API endpoints for security vulnerabilities.

Security (To be continued)

Security soon (As a new presentation)

- Common vulnerabilities found in smart contracts, such as reentrancy attacks, integer overflows, and denial-of-service exploits.
- Best practices for writing secure smart contracts, including thorough testing, code audits, and the use of established security patterns.
- The importance of considering real-world legal and regulatory implications.

Conclusion …

SEVEN

Conclusion

Conclusions …

- Try to understand the decentralized philosophy [2] behind
- **Smart contract development environments:** Remix, Ganache + Truffle (Also consider Hardhat) is enough as a start
- **Web 3.0 environments:** Node.js, Web3.js, VS Code
- However keep updated such as the sunset of some projects such as Ganache and Truffle https://consensys.io/blog/consensys-announces-the-sunset-of-truffle-and-ganache-and-new-hardhat
- Security section will be given as another slide.
- There is much to talk about, so keep on following our studies at aperta and zenodo https://aperta.ulakbim.gov.tr/search?page=1&size=20&q=blockchain&authors=Karaarslan,%20Enis
- Internet is full of free courses such as:
  - Learn Blockchain, Solidity, and Full Stack Web3 Development with JavaScript – 32-Hour Course https://www.youtube.com/watch?v=gyMwXuJrbJQ

Conclusions …

- Next Steps:
  - Build a full-stack DApps using a Node.js backend and a Web3.js front-end.
  - Implement security features, such as private key management and server-side signing.
- There is much to talk about, so keep on following our studies at aperta and zenodo https://aperta.ulakbim.gov.tr/search?page=1&size=20&q=blockchain&authors=Karaarslan,%20Enis
- The most recent version of the slides will be available at aperta and zenodo. https://zenodo.org/records/13996877

REFERENCES

[1] Karaarslan, E. ve Birim, M. (2021). Blokzincirde Güvenli ve Güvenilir Uygulama Geliştirme Temelleri. Siber Güvenlik ve Savunma Blokzinciri ve Kriptografi içinde (ss. 1–48). Nobel Yayınevi. https://aperta.ulakbim.gov.tr/record/273870

[2] Karaarslan, E., & Yazici Yilmaz, S. (2023). Metaverse and Decentralization. In Metaverse: Technologies, Opportunities and Threats (pp. 31-44). Singapore: Springer Nature Singapore. https://www.researchgate.net/publication/374687011_Metaverse_and_Decentralization

[3] Karaarslan, E., Birim, M., & Ari, H. E. (2022). Forming a decentralized research network: DS4H. Turkish Journal of Electrical Engineering and Computer Sciences, 30(2), 436-450. https://aperta.ulakbim.gov.tr/record/273886

[4] A Beginner's Guide to Web3.js, https://blog.chainsafe.io/beginners-guide-web3js/

Thanks for listening

Dr. Enis KARAARSLAN

enis.karaarslan@mu.edu.tr

https://linktr.ee/eniskaraarslan

https://www.linkedin.com/in/enis-karaarslan-1b195617/