Program Code:

1. MetaMask Wallet View (After Setup)

```markdown
MetaMask Wallet
------------------------------
Ethereum Address: 0xABcD1234... (Your public wallet address)
Balance: 1.0 ETH
```

2. Deposit Confirmation View (Activity Tab)

```yaml
MetaMask Activity
------------------------------
1.0 ETH received from 0x1234abcd...
Date: 2024-11-10 12:34:56 PM
TX Hash: 0x9876abcd...
Status: Success
```

3. Send Transaction View (Transaction Details)

```yaml
MetaMask Send
------------------------------
Recipient Address: 0x5678efgh...
Amount: 0.1 ETH
Gas Fee: 0.001 ETH
Transaction Fee: 0.001 ETH
Total: 0.101 ETH

[Next] [Confirm]
```

4. Transaction Status View (After Sending)

```yaml
MetaMask Activity
------------------------------
0.1 ETH sent to 0x5678efgh...
TX Hash: 0x1234abcd5678...
Block Number: 987654
Gas Used: 21000
Transaction Fee: 0.001 ETH
Status: Success
View on Etherscan
```

Inputs:

- **MetaMask Extension Installation:** ○ Install the MetaMask browser extension (Chrome, Firefox, Brave) or the MetaMask mobile app (iOS/Android).

● **Wallet Creation Process:**
○ Choose a **strong password** to protect the wallet.
○ **Backup the 12-24 word seed phrase**. This phrase is crucial for recovering the wallet if you lose access to it.
● **Account Creation**:
○ By default, MetaMask creates an **Account1** which will generate a unique Ethereum address. This address is your **public key** that you can share to receive ETH or tokens.
Terminal/MetaMask Tool View:
● After installation, MetaMask will open a small pop-up interface in your browser with options to **create a wallet** or **import an existing one**. The wallet creation process will be guided through the UI.
Depositing Cryptocurrency into MetaMask Inputs:
● **Ethereum Address**:
○ Copy your MetaMask wallet's Ethereum address (Account 1) from the MetaMask interface. This is the address you will send ETHorERC-20 tokens to.
● **Source Wallet or Exchange**:
○ Use an external wallet (e.g., another MetaMask account, Trust Wallet, or any Ethereumcompatible wallet) or an exchange (like Binance or Coinbase) to send ETH to your MetaMask address.
● **Amount**:
○ Specify the amount of cryptocurrency (e.g.,0.1ETH)to transfer.

Terminal/MetaMask Tool View:
● The **MetaMaskUI** provides an easy way to view your account balance in real-time. After depositing, you will see the ETH or token balance updated in the interface. Example: ○ Once the deposit is successful, your MetaMask wallet balance will show something like:
■ ETH:0.1
■ OrERC-20Token:50tokens
Transaction Input (Sending ETH from MetaMask) Inputs:
● **Recipient's Ethereum Address**:
○ Enter the Ethereum address of the recipient in MetaMask or paste it from your clipboard.
● **Amount to Send**:
○ Enter the exact amount of ETH you wish to send(e.g.,0.05ETH).
● **Gas Fees**:
○ MetaMask typically suggest sag as fee automatically based on current network conditions. However, you can choose to customize it:
■ **Low gas fees** will lead to slower transactions, while **high gas fees** will speed up the transaction.
○ Gas fees are paid in ETH and are essential for executing the transaction on the Ethereum blockchain.
● **Network Choice**:
○ If you're using **Ethereum's Mainnet**, transactions will be processed on the Ethereum blockchain.
○ Alternatively, if you want to use a**Testnet** (e.g.,Rinkeby,Goerli),you can set this in MetaMask.
Terminal/MetaMaskToolView:
● In the **MetaMaskUI**, when you click "Send", the fields to input:
○ Recipient address
○ Amount to send

○ Gas fees
○ Network selection(Main net, Test net) will appear in the interface.

Output

The outputs in this context will refer to what you get after performing actions such as creating a wallet, depositing crypto, or making a transaction. Let's break down each output:

MetaMask Wallet Creation Output

● **Generated Ethereum Address**:

○ After creating the wallet, you will have a unique Ethereum address for your MetaMask wallet (public address), which can be used to receive ETH and ERC-20 tokens.

○ Example:0x1234abcd5678efgh9101ijklmnop1234567890.

● **SeedPhrase**:

○ A **12-24 word backup phrase** is generated, which is essential to restore the wallet in case of device loss.

Depositing Cryptocurrency (ETH) into MetaMask

Outputs:

● **Updated Balance**:

○ After the deposit transaction is confirmed, your MetaMask wallet balance will show the updated amount of ETH or tokens that you received. It should match the amount you transferred from the external wallet or exchange. ● **Transaction Confirmation**:

○ Once the deposit is successfully processed, you will get a **transaction hash** (TXID) to check the status of the deposit on a block explorer like Etherscan.

● **Example**:

○ Transaction Hash: 0xabcdef1234567890abcdef...

○ You can use this hash to look up details of the transaction (sender, recipient, gas fees, etc.) on a blockchain explorer.

Ethereum Transaction Output (Transaction Execution)

Outputs:

● **Transaction Hash(TXID)**:

○ When the transaction is initiated from MetaMask, you'll receive a **Transaction Hash** after clicking "Confirm". This is a unique identifier that lets you trace the transaction in the Ethereum blockchain.

● **Transaction Details**:

○ After the transaction is mined, it will have the following information:

■ **Block Number**: Which block the transaction was included in.

■ **Timestamp**: The time when the transaction was mined.

■ **GasUsed**: How much gas was consumed during the transaction.

■ **Sender& Recipient**: Both the sender's and recipient's Ethereum addresses.

■ **Transaction Status**: Whether the transaction was successful or failed.

Output on Block Explorer (Etherscan):

● **TransactionHash**:0xabcdef1234567890abcdef... ● **BlockNumber**:12456789

● **GasUsed**:21000

● **From**:0x1234abcd5678efgh9101ijklmnop1234567890

● **To**:0x9876zxyw1234abcd5678efgh9101ijklmnop

● **Amount**:0.05ETH

● **Status**: **Success** ● **TransactionFee**:0.00021ETH

4. Transaction Status View (After Sending)

yaml Copy code

```
MetaMask Activity
------------------------------
0.1 ETH sent to 0x5678efgh...
TX Hash: 0x1234abcd5678...
Block Number: 987654
Gas Used: 21000
Transaction Fee: 0.001 ETH
Status: Success
View on Etherscan
```

a)Working with variable :

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

// Contract to demonstrate working with variables
contract VariablesExample {
  // State Variable: Stored permanently on the blockchain
  uint256 public stateNumber = 42;
  string public stateText = "Hello, Remix!";

  // Function to demonstrate local variables
  function localVariables() public pure returns (uint256) {
    uint256 localNumber = 100; // Local variable
    return localNumber;
  }

  // Function to demonstrate global variables
  function globalVariables() public view returns (address, uint256) {
    address sender = msg.sender; // Address of the caller
    uint256 timestamp = block.timestamp; // Current block timestamp
    return (sender, timestamp);
  }

  // Function to modify state variable
  function setStateNumber(uint256 _newNumber) public {
    stateNumber = _newNumber;
  }
}
```

b)Send Money:

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

// Contract to demonstrate working with variables
contract VariablesExample {
  // State Variable: Stored permanently on the blockchain
  uint256 public stateNumber = 42;
  string public stateText = "Hello, Remix!";

  // Function to demonstrate local variables
  function localVariables() public pure returns (uint256) {
    uint256 localNumber = 100; // Local variable
    return localNumber;
  }

 // Function to demonstrate global variables
```

```
    function globalVariables() public view returns (address, uint256) {
        address sender = msg.sender; // Address of the caller
        uint256 timestamp = block.timestamp; // Current block timestamp
        return (sender, timestamp);
    }

    // Function to modify state variable
    function setStateNumber(uint256 _newNumber) public {
        stateNumber = _newNumber;
    }
}
```

c)Mapping and struct:

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

// Contract demonstrating Mapping and Struct
contract MappingStructExample {
    // Struct to store user information
    struct User {
        string name;
        uint256 balance;
    }

    // Mapping to associate addresses with User structs
    mapping(address => User) public users;

    // Function to add or update a user's information
    function setUser(string memory _name, uint256 _balance) public {
        users[msg.sender] = User(_name, _balance);
    }

    // Function to retrieve user information
    function getUser(address _userAddress) public view returns (string memory, uint256) {
        User memory user = users[_userAddress];
        return (user.name, user.balance);
    }

    // Function to update the balance of a user
    function updateBalance(uint256 _newBalance) public {
        require(bytes(users[msg.sender].name).length != 0, "User does not exist");
        users[msg.sender].balance = _newBalance;
    }
}
```

d)Error handling:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
// Contract demonstrating error handling
contract ErrorHandling {    address
public owner;
  uint256 public balance;

  // Constructor to set the owner of the contract
  constructor() {
     owner = msg.sender;
  }

  // Function to deposit Ether into the contract
  function deposit() public payable {
     require(msg.value > 0, "Deposit amount must be greater than zero");
     balance += msg.value;
  }

  // Function to withdraw Ether from the contract
  function withdraw(uint256 _amount) public {
     require(msg.sender == owner, "Only the owner can withdraw");
     require(_amount <= balance, "Insufficient balance");
     balance -= _amount;
     payable(msg.sender).transfer(_amount);
  }

  // Function to demonstrate the use of revert
  function checkBalance() public view returns (string memory) {
     if (balance == 0) {
        revert("Balance is zero, no funds available");
     }
     return "Funds are available";
  }

  // Function to demonstrate assert
  function assertOwner() public view {
     assert(msg.sender == owner);
  }
}
```

View/Pure, Receive Function and Fallback Function

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

// Contract demonstrating View, Pure, Receive, and Fallback functions
contract ViewReceiveFallback {
```

```
  uint256 public number;

  // Function to set the number (example of a state-changing function)
  function setNumber(uint256 _number) public {
     number = _number;
  }

  // View function: Reads the state without modifying it
  function getNumber() public view returns (uint256) {
     return number;
  }

  // Pure function: Performs calculations without accessing state
  function addNumbers(uint256 a, uint256 b) public pure returns (uint256) {
return a + b;
  }

  // Receive function: Automatically executed when Ether is sent to the contract
receive() external payable {}

  // Fallback function: Triggered when the function called doesn't exist
  fallback() external payable {
     // You can optionally add logic here
  }

  // Function to check the contract balance
  function getBalance() public view returns (uint256) {
     return address(this).balance;
  }
}
```

e)Events and return:

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

// Contract demonstrating Events and Return Variables
contract EventsAndReturns {    // State variable to
store a number
  uint256 public storedNumber;

  // Event to log changes to the stored number
  event NumberUpdated(address indexed updater, uint256 oldNumber, uint256 newNumber);

  // Function to update the stored number
  function updateNumber(uint256 _newNumber) public returns (uint256) {
     uint256 oldNumber = storedNumber;
     storedNumber = _newNumber;
```

```
        // Emit the event
        emit NumberUpdated(msg.sender, oldNumber, _newNumber);

        // Return the new number
        return _newNumber;
    }

    // Function to get the sum of two numbers (pure function with return value)
    function getSum(uint256 a, uint256 b) public pure returns (uint256) {
        return a + b;
    }

    // Function to demonstrate returning multiple variables
    function getDetails() public view returns (address, uint256) {
        return (msg.sender, storedNumber);
    }
}
```

Program Code:

Interacting with a Smart Contract**:**

- Obtain the smart contract's ABI (Application Binary Interface) and address.
- UseWeb3.jstointeractwiththedeployed contract:

```
const contract = new web3.eth.Contract(abi, contractAddress);
contract.methods.methodName().call().then(result => {
  console.log(result);
});
```

Using Web3.js in Chrome (via MetaMask)**:**

- Install the MetaMask extension in your Chrome browser.
- Connect to the Ethereum network using MetaMask and load the Web3.js library directly from your browser's console.
- Interact with the contract using Web3.js directly from the browser, utilizing MetaMask as a wallet provider.

```
if (typeof window.ethereum !== 'undefined') {
  const web3 = new Web3(window.ethereum);
  await window.ethereum.request({ method: 'eth_requestAccounts' });
}
```

Inputs:

1. Ethereum Account Details**:**
   - Sender's and receiver's Ethereum addresses.
   - Sender's private key (used for signing transactions).
2. Ether Amount**:**
   - The amount of Ether to transfer, specified in ether units.
3. Smart Contract Details**:**
   - The ABI (Application Binary Interface) of the contract.
   - The deployed contract's address.
4. Web3.jsSetup**:**
   - The Web3 provider URL(e.g.,InfuraendpointorlocalGanache network).
5. **MetaMask Setup** (for browser interaction):
   - MetaMask wallet extension installed and connected to an Ethereum network.

Outputs:

1. Transaction Status**:**
   - A confirmation or error message indicating whether the Ether transfer was successful or failed.

2. Smart Contract Call Result**:**
3. The result of calling a method on the smart contract, typically returned as a JavaScript object.
   Browser Interaction Results**:**

- If interacting with a smart contract in the browser using MetaMask, the results of contract interactions (e.g., balance check, state change) are displayed in the browser console.

4. Transaction Hash**:**

- The transaction hash (txHash) generated after a successful transaction is broadcast to the network.

Program Code

ToolView(Node.js&Web3.js)

1. SetupNode.jsEnvironment

YouwillneedtoinstallNode.js(ifnotalreadyinstalled)andWeb3.js:
Install Node.js (if not installed)
#Downloadandinstallfromhttps://
nodejs.org/ # Initialize a new Node.js
project
npm init -y

#InstallWeb3.jsforinteractingwiththeblockchai
n npm install web3

2. Web3.jsInteractionExample(Node.jsScript)

Here'sanexamplescriptthatshowshowtointeractwithaHyperledgerFabricnetworkusing Web3.js in Node.js. javascript

```
// Get the account address
const account = '0x...'; // Replace with your account address
asyncfunctioninteractWithContract(){ try {
// Call a function (for example, to query the balance)

        let balance = await contract.methods.getBalance().call();
console.log("Balance:", balance);
        //Invokeafunction(forexample,tosetanew balance)
        awaitcontract.methods.setBalance(100).send({from: account });
console.log("Balance updated!");

} catch (error) {

console.error("Error interacting with contract:",
error);

}

}
interactWithContract();code
//ImportingtheWeb3.jslibrary const Web3 = require('web3');
// Connect to the Fabric test network using
Web3.js//FabricprovidesanEthereumcompatibleAPIusingthe Web3.js interface
constweb3=newWeb3('http://localhost:8545');//Assuming Fabric uses an Ethereum-compatible port
//Definetheaddressofthedeployedcontract(smartcontract address)
constcontractAddress='0x...';//Replacewithyourcontract address
//ABI(ApplicationBinaryInterface)ofthedeployedcontract const abi = [
{
```

```
"constant":true, "inputs": [],
"name": "getBalance",

"outputs":[{"name":"","type":"uint256"}], "payable": false,
"stateMutability":"view", "type": "function"
},

{

"inputs":[{"name":"_amount","type":"uint256"}], "name": "setBalance",
"outputs": [],
"payable": false, "stateMutability":"nonpayable",
                    "type": "function"
}

];

// Create a contract object

const contract = new web3.eth.Contract(abi, contractAddress);
```

Node.js and Web3.js Interaction Example Output

1. Node.js Environment Setup:

Command: npm init -y

Output:

```
{
  "name": "project-name",
  "version": "1.0.0",
  ...
}
```

Command: npm install web3

Output:

```
+ web3@<version>
added <number> packages in <time>
```

2. Running the Script:

Connection Output:

```
Connected to the blockchain network at http://localhost:8545
```

Function Call Outputs:

```
Balance: 1000
```

Program Code:

SolidityCodeforSharedWallet
solidity

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


contract SharedWallet
        { addresspublicowner; mapping(address => bool) public
owners; mapping(address=>uint256)publicbalances; uint256
public totalBalance;

event Deposit(address indexed sender, uint256 amount);
        eventWithdraw(addressindexedreceiver,uint256amount); event
        OwnershipTransferred(address indexed oldOwner,
address indexed newOwner);
modifier onlyOwner() { require(owners[msg.sender],"Notanowner");
_;
}
modifier onlyContractOwner()
{ require(msg.sender==owner,"Onlycontractownercan add owners");
_;
}
constructor()
{ owner=msg.sender;//Setthedeployerasthe contract owner
                owners[owner]=true;//Initially,thedeployeris the only owner
}

        functionaddOwner(address_newOwner)external onlyContractOwner {
require(!owners[_newOwner],"Alreadyanowner"); owners[_newOwner] = true;
emit OwnershipTransferred(msg.sender, _newOwner);

}

        functionremoveOwner(address_owner)external onlyContractOwner {
require(owners[_owner],"Notanowner"); owners[_owner] = false;
emit OwnershipTransferred(msg.sender, _owner);

}

function deposit() external payable {

require(msg.value > 0, "Deposit must be greater than
0");
```

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

totalBalance += msg.value;

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

}

function withdraw(uint256 _amount) external onlyOwner
                { require(_amount<=balances[msg.sender],"Insufficient
balance");

                require(_amount<=totalBalance,"Insufficient contract balance"); balances[msg.sender]
                -= _amount; totalBalance -= _amount;
                payable(msg.sender).transfer(_amount); emit
                Withdraw(msg.sender, _amount);
}

functioncheckBalance()externalviewreturns(uint256){ return address(this).balance;
}

        functionownerBalance(address_owner)externalview returns (uint256) {
returnbalances[_owner];

}
```

ToolView(Node.js,Web3.js)
TointeractwiththedeployedcontractusingNode.jsandWeb3.js,youcanfollowthesteps below:

1. InstallDependencies

Install the required packages:

bash

```
npm install web3
```

2. Web3.jsSetup

CreateanewNode.jsfile(e.g.,index.js)tointeractwiththecontract: javascript

```
const Web3 = require('web3');

//ConnecttolocalEthereumnode(GanacheorInfurafor testnet)
constweb3=newWeb3('http://localhost:8545');//oran Infura URL

//ContractABI(copythisfromyourSolidityContract compilation output)
const abi = [
```

```
// Add the ABI of the SharedWallet contract here

];

//Deployedcontractaddress(afterdeploymentonthe blockchain)
const contractAddress = '0xYourContractAddress';
// Instantiate the contract

const contract = new web3.eth.Contract(abi, contractAddress);

// Your Ethereum account address

const account = '0xYourAccountAddress'; //DepositFundsExample(sendingEthertothecontract)
async function depositFunds() { constamountInEther='0.1';//TheamountofEtherto deposit
// Convert Ether to Wei constamountInWei=web3.utils.toWei(amountInEther,
        'ether');

//Sendatransactiontothedepositfunction await contract.methods.deposit().send({
from: account, value:amountInWei
});

}

// Check balance of the contract
asyncfunctioncheckContractBalance(){ const balance
        = await contract.methods.checkBalance().call(); console.log('Contract
Balance: ', web3.utils.fromWei(balance,'ether'),'ETH');
}
//WithdrawFundsExample(fromanowner) async function withdrawFunds()
  {
const amountInEther = '0.05'; // Amount to withdraw

        constamountInWei=web3.utils.toWei(amountInEther, 'ether');


        awaitcontract.methods.withdraw(amountInWei).send({from: account });
}

//Exampleusage (async ()
 => {
awaitdepositFunds();
```

```
plaintext                                                    Copy code

Transaction Sent: Deposit of 0.1 ETH to contract from 0xYourAccountAddress
Contract Balance: 0.1 ETH
```

Then, the `checkContractBalance()` function is executed to verify the current balance of the contract.

awaitcheckContractBalance(); await withdrawFunds();
})();

3. RunningtheCode

ToruntheNode.jsscript:

bash

nodeindex.js

```plaintext
Contract Balance: 0.1 ETH
```

Next, the `withdrawFunds()` function is executed. The contract successfully sends `0.05 ETH` to the account.

```plaintext
Transaction Sent: Withdrawal of 0.05 ETH from 0xYourAccountAddress
Contract Balance: 0.05 ETH
```

Error Handling (if applicable):

If any issues arise, such as insufficient balance, you may see error messages like:

```plaintext
Error: Insufficient contract balance
```

Expected Final Console Output:

```plaintext
Transaction Sent: Deposit of 0.1 ETH to contract from 0xYourAccountAddress
Contract Balance: 0.1 ETH
Transaction Sent: Withdrawal of 0.05 ETH from 0xYourAccountAddress
Contract Balance: 0.05 ETH
```

Program Code:

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

contract MessageContract {
string public message;

constructor(string memory initialMessage) {
message = initialMessage;
  }

function setMessage(string memory newMessage) public {
message = newMessage;
  }
}
```

Steps to Compile and Deploy:

1. **Compile the Contract in Remix:**
   - Open Remix.
   - Paste the above code into a new file and compile it.
   - Copy the **ABI** and **bytecode** from the Remix compilation output.
2. **Set Up MEW with Ganache:**
   - Open MEW and navigate to the "Network" settings.
   - Add a custom network using Ganache's RPC URL (e.g., http://127.0.0.1:7545).
3. **Deploy Contract via MEW:**
   - Log in to MEW using a private key from Ganache.
   - Go to "Interact with Contract" > "Deploy Contract".
   - Paste the contract bytecode and constructor arguments (e.g., "Hello, Blockchain!").
   - Click "Deploy". **Input & Output**

4. **Input:**

   - Ganache RPC URL and private key.
   - Solidity contract bytecode.
   - Constructor argument: "Hello, Blockchain!".

Output:

- A successful transaction on the Ganache blockchain.
- The deployed contract's address..

Program Coding

Step 1:Install Golang

1. Download and install Golang from https://golang.org/dl/.
2. Verify the installation: bash
   go version

Step 2: Initialize a Go Module

1. Create a project folder: bash
   mkdir GoProject
   cd GoProject

Initialize the module:

bash
go mod init example.com/gopackage

Step 3: Create a Package

1. Create a file reverse/reverse.go:

```
go package
reverse

// ReverseString reverses a given string
funcReverseString(s string)string {
result :="" for _, v
:= range s {
result = string(v) + result
  } return
result }
```

Step 4: Create the Main Command

1. Create a file main.go:

```
go

package main

import (
"fmt"
"example.com/gopackage/reverse"
)

funcmain() { input := "Golang" output
:= reverse.ReverseString(input)
fmt.Println("Original String:", input)
fmt.Println("Reversed String:", output)
}
```

Step 5: Build and Run
1. Run the application:
bash

go run main.go 2.
Build an executable:
bash

go build -o gopackage
./gopackage

Input and Output
Input:

Input string provided in the main.go file:
go

input := “Golang"

Output:

When executed, the program outputs:
arduino

Original String: Golang
Reversed String: gnalog

Programming Coding

Step 1: Import Required Libraries

go

```
package main

import (
"bytes"
"crypto/sha256"
"fmt"
"strconv"
"time"
)
```

Step 2: Define the Block

go

```
type Block struct {
Timestamp     int64
Data          []byte
PrevHash      []byte
Hash          []byte
Nonce         int
}

func(b *Block) SetHash() {
timestamp := []byte(strconv.FormatInt(b.Timestamp, 10)) headers :=
bytes.Join([][]byte{b.PrevHash, b.Data, timestamp}, []byte{}) hash :=
sha256.Sum256(headers) b.Hash = hash[:]
}

funcNewBlock(data string, prevHash []byte) *Block {
block := &Block{time.Now().Unix(), []byte(data), prevHash, []byte{}, 0}
return block }
```

Step 3: Define Proof of Work

go

```
const difficulty = 2

type ProofOfWork struct {
Block  *Block
Target []byte
}
```

```
funcNewProofOfWork(b *Block) *ProofOfWork {
target := bytes.Repeat([]byte{0}, difficulty)
return&ProofOfWork{b, target}
}

func(pow *ProofOfWork) Run() (int, []byte) {
var hash [32]byte var nonce int for { data :=
bytes.Join([][]byte{ pow.Block.PrevHash,
pow.Block.Data,
[]byte(strconv.Itoa(int(pow.Block.Timestamp))),
[]byte(strconv.Itoa(nonce)),
}, []byte{}) hash =
sha256.Sum256(data) if
bytes.HasPrefix(hash[:], pow.Target) {
break }
else {
nonce+
+ }
}
return nonce, hash[:]
}

func(pow *ProofOfWork) Validate() bool {
data := bytes.Join([][]byte{
pow.Block.PrevHash, pow.Block.Data,
[]byte(strconv.Itoa(int(pow.Block.Timestamp))),
[]byte(strconv.Itoa(pow.Block.Nonce)),
}, []byte{}) hash :=
sha256.Sum256(data)
return bytes.HasPrefix(hash[:], pow.Target)
}
```

Step 4: Define the Blockchain

go

```
type Blockchain struct {
Blocks []*Block
}

funcNewBlockchain() *Blockchain { genesisBlock :=
NewBlock("Genesis Block", []byte{}) pow :=
NewProofOfWork(genesisBlock) nonce, hash :=
pow.Run() genesisBlock.Hash = hash
genesisBlock.Nonce = nonce
return&Blockchain{[]*Block{genesisBlock}}
```

```
}

func(bc *Blockchain) AddBlock(data string) {
prevBlock := bc.Blocks[len(bc.Blocks)-1]
newBlock := NewBlock(data, prevBlock.Hash)
pow := NewProofOfWork(newBlock) nonce,
hash := pow.Run() newBlock.Hash = hash
newBlock.Nonce = nonce bc.Blocks =
append(bc.Blocks, newBlock) }
```

Step 5: Main Program

```
go

funcmain() { bc :=
NewBlockchain()
fmt.Println("Mining
block 1...")
bc.AddBlock("Block 1
Data")

fmt.Println("Mining block 2...")
bc.AddBlock("Block 2 Data")

fmt.Println("\nBlockchain:") for
_, block := range bc.Blocks {
fmt.Printf("Timestamp: %d\n", block.Timestamp)
fmt.Printf("Data: %s\n", block.Data)
fmt.Printf("PrevHash: %x\n", block.PrevHash)
fmt.Printf("Hash: %x\n", block.Hash)
fmt.Printf("Nonce: %d\n\n", block.Nonce)
}
}
```

Input & Output

Input:

The program automatically mines blocks using the AddBlock function, taking data for each block (e.g., "Block 1 Data").

Output:

Sample output after running the program:

```
makefile
```

Mining block 1...
Mining block 2...

Blockchain:
Timestamp: 1697590650
Data: Genesis Block
PrevHash:
Hash: 00a9f7c9bdbb5ad487...
Nonce: 245

Timestamp: 1697590651
Data: Block 1 Data
PrevHash: 00a9f7c9bdbb5ad487...
Hash: 00ff46f3c3b3af987...
Nonce: 132

Timestamp: 1697590653
Data: Block 2 Data
PrevHash: 00ff46f3c3b3af987...
Hash: 009d4f2e7bcbbff9aa...
Nonce: 188

Block 1

Block 1:
Timestamp: Sat Nov 16 11:02:43 2024
Data: Genesis Block
PrevHash: 0
Hash: 0069a2cc8642a01de1f5dd12f784c3a6230b4e20de013204af08d5879473af81
Nonce: 113

Block 2

Block 2:
Timestamp: Sat Nov 16 11:02:43 2024
Data: Block 1 Data
PrevHash: 0069a2cc8642a01de1f5dd12f784c3a6230b4e20de013204af08d5879473af81
Hash: 0000f3dab3f99659a75bcabce0eaa9ab5d430a2947e61991f2187c4a1804db2d
Nonce: 315

Block 3

Block 3:
Timestamp: Sat Nov 16 11:02:43 2024
Data: Block 2 Data
PrevHash: 0000f3dab3f99659a75bcabce0eaa9ab5d430a2947e61991f2187c4a1804db2d
Hash: 00ee86c94cda153147a9612017a351e1f78bfd203e3c2734a41ff9abeee82718
Nonce: 26

Program Coding

Step 1: Load Connection Profile

java

```
import org.hyperledger.fabric.gateway.*;
import java.nio.file.Path; import
java.nio.file.Paths;

publicclassBlockchainApp {
publicstaticvoidmain(String[] args)throws Exception {
PathwalletPath=Paths.get("wallet");
Walletwallet=Wallets.newFileSystemWallet(walletPath);

PathnetworkConfigPath=Paths.get("connection-org1.json");
      Gateway.Builderbuilder=Gateway.createBuilder()
            .identity(wallet, "admin")
            .networkConfig(networkConfigPath);

try (Gatewaygateway= builder.connect()) {
Networknetwork=gateway.getNetwork("mychannel");
Contractcontract=network.getContract("mycc");

// Submit a transaction
byte[] result = contract.submitTransaction("createAsset", "asset1", "value1");
System.out.println("Transaction successful: " + newString(result));

// Query the asset byte[] queryResult =
contract.evaluateTransaction("readAsset", "asset1");
System.out.println("Query Result: " + newString(queryResult));
      }
   }
}
```

Step 2: Connection Profile and Wallet Setup

• Use the connection-org1.json file exported during the Fabric network setup.
• Create a wallet directory to store user identities.

Input & Output
Input:

• Input provided via smart contract transactions:
java

```
contract.submitTransaction("createAsset", "asset1", "value1");
```

Output:

Upon successful execution:

sql

```
Transaction successful: Asset created with ID asset1
Query Result: {"id":"asset1","value":"value1"}
```

Program Coding

Example Chaincode (Go):

go

```
package main

import (
"encoding/json"
"fmt"
"github.com/hyperledger/fabric-contract-api-go/contractapi"
)

type SmartContract struct {
  contractapi.Contract
}

type Asset struct {
ID    string`json:"id"`
  Value string`json:"value"`
}

func(s *SmartContract) CreateAsset(ctx contractapi.TransactionContextInterface, id string, value
string) error { asset := Asset{
      ID:    id,
      Value: value,
  }
assetJSON, err := json.Marshal(asset)
if err != nil {
return err
  }
return ctx.GetStub().PutState(id, assetJSON)
}

func(s *SmartContract) ReadAsset(ctx contractapi.TransactionContextInterface, id string) (*Asset,
error) {
assetJSON, err := ctx.GetStub().GetState(id)
if err != nil || assetJSON == nil {
returnnil, fmt.Errorf("asset %s does not exist", id)
  }
var asset Asset
json.Unmarshal(assetJSON, &asset)
return&asset, nil
}

funcmain() {
```

```
chaincode, err := contractapi.NewChaincode(new(SmartContract))
if err != nil {
fmt.Printf("Error starting chaincode: %s", err)
   }
chaincode.Start()
}
```

Input & Output

Input:

Invoke chaincode commands:

bash

peer chaincode invoke -C mychannel -n basic -c '{"function":"CreateAsset","Args":
["asset1","value1"]}' peer chaincode query -C mychannel -n basic -c
'{"Args":["ReadAsset","asset1"]}' **Output:** json

{"id":"asset1","value":"value1"}

Program Code:

python

```
import hashlib
import time

# Define a Block class Block: def __init__(self, index,
previous_hash, timestamp, data, hash):
    self.index = index
    self.previous_hash = previous_hash
self.timestamp = timestamp
self.data = data
    self.hash = hash

# Generate Block Hash def calculate_hash(index,
previous_hash, timestamp, data): value = str(index) +
previous_hash + str(timestamp) + data return
hashlib.sha256(value.encode('utf-8')).hexdigest()

# Create the Blockchain
class Blockchain: def
__init__(self):
self.chain = []
    self.create_genesis_block()

  # Create the first block (genesis block) def create_genesis_block(self):      genesis_block =
Block(0, "0", time.time(), "Genesis Block", calculate_hash(0, "0", time.time(),
"Genesis Block"))
self.chain.append(genesis_block)

  # Add a new block def
add_block(self, data):
    last_block = self.chain[-1]      new_index = last_block.index + 1
new_timestamp = time.time()      new_hash = calculate_hash(new_index,
last_block.hash, new_timestamp, data)      new_block = Block(new_index,
last_block.hash, new_timestamp, data, new_hash) self.chain.append(new_block)

  # Print the Blockchain
def print_blockchain(self):
for block in self.chain:
print(f"Block #{block.index} [Hash: {block.hash}] - Data: {block.data}")

# Create Blockchain and add blocks
my_blockchain = Blockchain()
my_blockchain.add_block("Transaction 1")
my_blockchain.add_block("Transaction 2")
```

```
# Print the blockchain
my_blockchain.print_blockchain()
```

Explanation of the Code:

- **Block Class**: Defines a block with attributes such as index, previous hash, timestamp, data, and hash.
- **Blockchain Class**: Manages the chain of blocks, including adding new blocks and printing the entire blockchain.
- **Hash Calculation**: Uses SHA256 to create a hash that is unique for every block, ensuring blockchain integrity.

Input and Output Example:

- **Input**: "Transaction 1", "Transaction 2"
- **Output**: A printed blockchain with each block's hash and data.

Screenshot of Output:

plaintext

```
Block #0 [Hash: 6b49e6e3806ed057e5e45c18be7792ab3f7b29fe7c3b8de3f1b30930f30d3794] - Data: Genesis Block
Block #1 [Hash: 35a9868d4a51fa7e4b8764bdf67d2d039459cd27c63c99fdd17de46f3f03752f] - Data: Transaction 1
Block #2 [Hash: bfe74f5f73c89c8acb755ad4593b63e1a38a0723a70e810a7b027939963b47b6] - Data: Transaction 2
```

***Coding**:*

python

```
from bitcoinlib.wallets import Wallet from
bitcoinlib.services.services import Service

# Create a new wallet
defcreate_wallet(wallet_name="MyWallet"):
# Create or load a wallet wallet =
Wallet.create(wallet_name)

# Show wallet details
print(f"Wallet Created: {wallet.name}")
print(f"Public Address: {wallet.get_key().address}")
print(f"Private Key: {wallet.get_key().private_hex}")
return wallet

# Save wallet details defsave_wallet_details(wallet): #
Export the private key to a file
withopen(f"{wallet.name}_private_key.txt", "w") as f:
f.write(wallet.get_key().private_hex)

print(f"Private key saved to {wallet.name}_private_key.txt")

# Example: Create a wallet and save details
my_wallet = create_wallet("MyCryptoWallet")
save_wallet_details(my_wallet)
```

***Explanation**:*

1. **Creating a Wallet**: Wallet.create(wallet_name) creates a new wallet with the given name.
2. **Retrieving Keys**: The wallet's public and private keys can be accessed via wallet.get_key().address and wallet.get_key().private_hex.
3. **Saving Private Key**: The private key is saved into a file for later use.

Modules in Blockchain:

1. **Blockchain Package Modules**:
   - bitcoinlib: Wallet and transaction generation, blockchain exploration.
   - pycoin: Tools for handling Bitcoin transactions and addresses.
   - blockcypher: API for querying blockchain data and sending transactions .
2. **Blockchain Block Explorer**:
   - **Function**: Allows users to query a blockchain for block information (e.g., using APIs from blockcypher or blockchain.com).
   - **Example API Call**:

python

```
import requests
```

block_hash = "000000000000000000c4d63b2b9c21b537cfbb31a960f9574c4cf99c5bdb15d" url = f"https://api.blockcypher.com/v1/btc/main/blocks/{block_hash}" response = requests.get(url) block_data = response.json() print(block_data)

3. **Create Wallet Module**:
   - **Function**: Generates and manages wallets and keys.
   - **Example**: Refer to the wallet creation code above.

4. **Exchange Module**: ○ **Function**: Interfaces with cryptocurrency exchanges like Binance, Coinbase, etc., to fetch market data. ○ **Example**: Using ccxt for interaction with exchanges.

python

```
import ccxt
exchange = ccxt.binance()
ticker = exchange.fetch_ticker('BTC/USDT')
print(ticker)
```

5. **Pushtx Module**:
   - **Function**: Allows users to send transactions to the blockchain network.
   - **Example**: Sending a Bitcoin transaction using bitcoinlib.

python

```
from bitcoinlib.transactions import Transaction
tx = Transaction()
# Add inputs and outputs to the transaction
tx.fee_per_kb(1000).send() print(tx.info())
```

6. **V2.Receive Module**: ○ **Function**: Receives transactions on the blockchain, using APIs to get incoming transactions for addresses. ○ **Example**: Can be done using services like blockchain.com or custom node implementations.

7. **Statistics Module**:
   - **Function**: Collects blockchain transaction data and generates statistics, using libraries like pandas and matplotlib.
   - **Example**:

python

```
import pandas as pd import
matplotlib.pyplot as plt

# Example transaction data (replace with real data) data =
{"Block": [1, 2, 3], "Transactions": [100, 200, 300]} df =
pd.DataFrame(data)

# Plotting
df.plot(x="Block", y="Transactions", kind="bar")
plt.show()
```

Input and Output Example:

Input:

- User creates a wallet named "MyCryptoWallet".
- Wallet public and private keys are displayed.
- The private key is saved to a file.

Output:

plaintext

```
Wallet Created: MyCryptoWallet
Public Address: 1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa
Private Key: 5Jv9rM7oszUwFbz5Mno3g4qkPHEk8nSvdG8vWhAbCwMDe6vsXY9
Private key saved to MyCryptoWallet_private_key.txt
```

Screenshot (Example Output):

plaintext

```
Wallet Created: MyCryptoWallet
Public Address: 1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa
Private Key: 5Jv9rM7oszUwFbz5Mno3g4qkPHEk8nSvdG8vWhAbCwMDe6vsXY9
Private key saved to MyCryptoWallet_private_key.txt
```

Program Code

solidity

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

contract Transaction {
address public owner;

constructor() {
owner = msg.sender;
  }

  // Function to transfer Ether
function transferEther(address payable recipient) public payable {
require(msg.value > 0, "Must send some Ether");
recipient.transfer(msg.value);
  }

  // Get balance of contract
function getContractBalance() public view returns (uint) {
return address(this).balance;
  }
}
```

Input

1. Use Ganache's test accounts to:
   - Select an *account* to deploy the contract.
   - Send Ether from one account to another through the contract.
2. Deploy the contract using Remix, specifying the deployed account.
3. Input the recipient address and Ether value in the transfer Ether function.

Output

- Successful deployment of the contract.
- Successful transaction of Ether between accounts.
- Verification of updated account balances on Ganache

Program Code

A simple Solidity smart contract to perform transactions:

solidity

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

contract SimpleWallet {
address public owner;

constructor() {
owner = msg.sender;
  }

  // Deposit Ether to the contract
function deposit() public payable {}

  // Withdraw Ether from the contract function withdraw(address
payable recipient, uint amount) public { require(msg.sender ==
owner, "Only owner can withdraw"); require(amount <=
address(this).balance, "Insufficient balance");
recipient.transfer(amount);
  }

  // Get contract balance
function getBalance() public view returns (uint) {
return address(this).balance;
  }
}
```

Input

1. Ganache Details: ○ RPC URL: http://127.0.0.1:7545 ○ Chain ID: 1337
2. Account Address and Private Key:
○ From Ganache, copy any account's private key.
3. Interaction Inputs:
○ Deploy the contract using Remix and Ganache account. ○ Use the deposit function to send Ether to the contract. ○ Use the withdraw function to transfer Ether from the contract to another address.

Output

1. Successful connection of Remix with Ganache (verified by account synchronization).
2. Deployment of the smart contract on Ganache via Remix.
3. Transaction details visible in the Ganache Transactions tab.
4. MyEtherWallet connected to Ganache and showing account balances.

Program Code:

solidity
Copy code

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

contract HelloWorld {
string public message;

constructor(string memory initMessage) {
message = initMessage;
  }

function updateMessage(string memory newMessage) public {
message = newMessage;
  }
}
```

Input and Output:

- **Input:** Deploy the contract with an initial message (e.g., "Hello, Blockchain!") using MetaMask.
- **Output:** Successfully deployed contract and updated message visible on the Ropsten Test Network.

Use Case 1

Retail - Establishing Unconditional Transparency in the Food Supply Chain Using Blockchain (Hyperledger Fabric)

Introduction

In the retail industry, particularly in the food supply chain, ensuring transparency, traceability, and trust has become a pressing need. Traditional supply chains often suffer from inefficiencies, fraud, and lack of visibility. Blockchain, powered by **Hyperledger Fabric**, addresses these issues by creating an immutable, decentralized ledger to track the lifecycle of food products from farm to fork. This enhances food safety, boosts consumer trust, and minimizes risks.

Challenges in the Food Supply Chain

1. **Lack of Transparency**: Limited visibility into product origin, quality, and handling.
2. **Inefficiency**: Manual and siloed data recording leads to delays and errors.
3. **Fraud**: Counterfeit products and tampering affect brand integrity.
4. **Food Safety Issues**: Difficulty in tracking and recalling contaminated products.

Blockchain Solution with Hyperledger Fabric

Hyperledger Fabric is an enterprise-grade, permissioned blockchain framework. It enables a consortium of stakeholders (farmers, transporters, manufacturers, retailers) to securely share data, providing the following benefits:

1. **Traceability**: Track every step of the product lifecycle with timestamps.
2. **Transparency**: Immutable records visible to all stakeholders.
3. **Automation**: Smart contracts enforce compliance and automate key processes.
4. **Efficiency**: Eliminate intermediaries and manual data reconciliations.

Implementation Workflow

1. **Network Design**:
   - Establish a Hyperledger Fabric network with key stakeholders (farmers, distributors, retailers).
   - Define roles and permissions.
2. **Data Recording**:
   - Farmers record crop details (e.g., type, harvest date, organic status) on the blockchain.
   - Transporters log shipping conditions (e.g., temperature, time).
3. **Smart Contracts**:
   - Automate verification processes (e.g., temperature thresholds during transport).
   - Trigger alerts for discrepancies or non-compliance.
4. **Consumer Access**: ○ End-users scan QR codes on product packaging to view the product's origin, journey, and certifications.

Technical Architecture

1. **Consensus Mechanism**: PBFT (Practical Byzantine Fault Tolerance) for validation.
2. **Key Components**:
   - **Peers**: Nodes representing stakeholders.

- **Chaincode (Smart Contracts)**: Define supply chain rules. - **Ledger**: Immutable, shared database.
3. **Integration**:
- IoT sensors for real-time data (temperature, humidity).
- APIs for data input/output and QR code generation.

Benefits

1. **For Consumers**:
- Confidence in product authenticity and quality.
- Access to detailed product history.
2. **For Retailers**:
- Strengthened brand reputation. -
Faster recalls and reduced losses.
3. **For Regulators**:
- Simplified audits and compliance verification.

Example: Mango Supply Chain

A mango farm records harvest details on the blockchain. As the mangoes move through transport, processing, and retail, each stakeholder updates the blockchain with details like shipping conditions, processing times, and shelf placement. At the store, consumers scan a QR code to confirm that the mangoes were organically grown and shipped under optimal conditions.

Conclusion

By integrating **Hyperledger Fabric** into the food supply chain, retailers can establish unconditional transparency, improve operational efficiency, and build consumer trust. This blockchain-based solution sets a benchmark for reliability, fostering innovation and sustainability in the retail sector.

Use Case 2

Banking and Financial Services – Increasing transaction volume and network resilience while maintaining confidentiality requirements for real time gross settlement among different banks of a central banking consortium

Aim

The aim of this use case is to enhance the **Real-Time Gross Settlement (RTGS)** system by utilizing blockchain technology to increase transaction volume, improve network resilience, and maintain confidentiality across multiple banks within a central banking consortium. Blockchain will enable real-time, secure, and transparent settlements between banks while ensuring compliance with regulatory confidentiality requirements.

The key objectives are:

- **Increase transaction through put** to support high-frequency trading and large transaction volumes.
- **Enhance network resilience** to ensure continuous, uninterrupted operations even during system failures or attacks.
- **Ensure confidentiality** for sensitive financial transactions while enabling necessary transparency.

Algorithm

1. **NetworkSetup:**

o Establish a **permissioned blockchain** network using **Hyperledger Fabric** or **Quorum** (Ethereum-based blockchain platform) to allow only authorized banks and financial institutions to participate.

o Setup nodes for each participating bank, the central bank, and trusted third-party auditors (if required).

o Implement **smart contracts (chaincode)** to define the rules for RTGS transactions ,including validation, settlement instructions, and compliance checks.

2. Transaction Initiation:

o A bank(Bank A) sends a **transaction request** to transfer funds to another bank(Bank B).The transaction includes the amount, recipient details, and timestamp.

o This transaction is encrypted using cryptographic techniques to maintain confidentiality,ensuringthatonlyauthorizedpartiescanaccessthedetails.

3. Transaction Validation:

o The participating banks involved in the transaction (via consensus mechanism) validate that the transaction adheres to the network rules and liquidity constraints.

o The **central bank** or an appointed trusted authority performs additional checks to ensure regulatory compliance (e.g., Anti-Money Laundering, fraud prevention).

o If valid ,the smart contract triggers the transaction for settlement, updating the respective ledgers of Bank A and Bank B. 4. Real-Time Settlement:

o The blockchain system updates both banks' ledgers simultaneously in real-time, ensuring that Bank A' s balance is debited and Bank B's balance is credited. o **Finality**: Transactions are **finalized immediately**, preventing double-spending or rollback issues typically seen in traditional systems.

5. Confidentiality Management:

- To meet confidentiality requirements, the blockchain implements **zero-knowledge proofs (ZKPs)** or **private channels** between transacting banks, ensuring that sensitive transaction data(e.g., customer information)is only visible to the involved parties.
- Additionally, banks can use **encrypted messaging** within the blockchain, ensuring that only authorized entities can decrypt and view transaction details.

6. Network Resilience:
- Blockchain's decentralized nature ensures that the network remains resilient even in the event of node failures or attacks.
- **Fault tolerance**: The system is designed to automatically recover from node failures without disrupting the transaction flow. This is achieved by maintaining copies of the ledger across multiple nodes in the network.
- **Redundancy**: In case of a breakdown or attack on any part of the network, otherbanksornodescontinueprocessingtransactions,ensuringuninterrupted operations.

7. Audit and Compliance:
- The blockchain ledger provides an immutable, time-stamped, and transparent record of all transactions.
- The **central bank** and regulatory bodies can access this data (within confidentiality limits)to audit the transactions and verify compliance with financial regulations.
- **Regulatory oversight** is simplified by providing an easily accessible, tamper-proof audit trail.

Inputs

1. **TransactionData:**
- Amount of money to be transferred.
- Sender and receiver bank details.
- Timestampofthetransaction.
- Any associated reference or transaction ID.

2. Banking Regulations:
- Compliance criteria, such as Anti-Money Laundering(AML)rules, fraud detection criteria, and transaction limits.
- Confidentialityrequirements(e.g.,customerprivacy,financialinstitution privacy).

3. Blockchain Configuration:
- Consensus mechanism(e.g., **Practical Byzantine Fault Tolerance(PBFT)**, **Raft**).
- Cryptographic keys for transaction validation and encryption.
- Smart contract templates for RTGS rules and validation.

4. Node Information:
- Details about the participating nodes (banks, central bank, regulators, etc.).
- Node permissions and access controls (ensuring confidentiality and security).

Outputs

1. **Blockchain Ledger:**
- A fully updated, immutable ledger reflecting the real-time settlement of the transaction across the banks.

2. Transaction Status:

o Confirmation of successful transaction settlement or an error message if the transaction fails (e.g., insufficient liquidity).
o Real-time notifications sent to both sending and receiving banks about the transaction status.
3. Audit Trail:
o An immutable audit trail of the transaction with timestamps, involved parties, and settlement details. This provides transparency to regulatory bodies and the central bank.
4. Smart Contract Execution Result:
o The execution result of smart contracts, including validation, transaction approval, and regulatory checks.
5. Transaction Settlement Report:
o A report generated after the transaction is completed, detailing the transaction amount, involved entities, and settlement confirmation.

1. Increased Transaction Volume:

o By utilizing blockchain's decentralized structure, the system is capable of processing a high volume of transactions in real-time, thus enabling large-scale settlement operations without compromising efficiency.
2. Enhanced Network Resilience:
o Blockchain's fault-tolerant design ensures that the RTGS system is more resilient to node failures, down time, or network attacks. Even if some banks' nodes go offline, other nodes can continue operations without disruption. 3. Confidentiality and Compliance:
o The use of cryptographic techniques, zero-knowledge proofs, and private channels ensures that transaction data remains confidential, thus meeting the strict confidentiality requirements for financial institutions.
o Regulatory compliance is strengthened through trans parent audit trails and automated compliance checks via smart contracts. 4. Reduced Transaction Costs and Time:
o The blockchain eliminates intermediaries, automates the settlement process, and reduces the risk of fraud, leading to cost savings and faster transaction processing.
o Transaction finality is immediate, reducing the need for traditional reconciliation processes.
5. Transparency and Trust:
o Blockchain's transparent ledger provides real-time, immutable records of all transactions. This increases trust among consortium banks and regulators as they can independently verify the integrity and correctness of the transactions.
6. Improved Liquidity Management:
o By providing real-time updates of account balances, banks can better manage liquidity and ensure that sufficient funds are available for transactions in real time.
7. Scalability:
o The use of blockchain ensures that the system is scalable to handle growing transactionvolumesasthenetworkofparticipatingbanksexpands,making it future-proof for large financial ecosystems.

Conclusion

In this use case, the implementation of blockchain technology for **Real-Time Gross Settlement(RTGS)** with in a central banking consortium enables significant

improvements in transaction volume, network resilience, and confidentiality. By leveraging a **permission blockchain** like **Hyperledger Fabric** or **Quorum**, the system ensures fast, secure, and transparent transactions between multiple banks while adhering to strict confidentiality and regulatory requirements. The decentralized nature of blockchain provides enhanced fault tolerance, reducing the risk of downtime and system failures. Additionally, real-time settlement and automation through **smart contracts** streamline operations, reduce costs, and improve liquidity management. Overall, blockchain offers a scalable, efficient, and trustworthy solution for modernizing banking and financial services, ensuring greater operational efficiency and regulatory compliance.

USE CASE3

HEALTH CARE-ELECTRONICHEAL THRECORDS AND MEDICAL RESEARCH DATA DIGITIZATION

Aim

The primary aim of this use case is to leverage blockchain technology for the secure, transparent, and efficient management of **Electronic Health Records (EHRs)** and **medical research data**. The goal is to ensure the integrity, privacy, and accessibility of patient records and research data, while also enabling faster and more accurate medical insights.

- **Enhance Data Security**: Protect sensitive medical data from unauthorized access or tampering.
- **Improve Data Interoperability**: Allow different health care providers, researchers, and institutions to access and share medical data in a standardized and trust less environment.
- **Enable Transparent Research**: Provide a transparent system for managing medical research data, ensuring proper attribution, and fostering collaboration.
-

Algorithm

1. **Data Collection**:

o Health care data(EHR, medical history, diagnosis, treatment records, etc.)and research data (clinical trials, studies, medical papers) are collected from various sources such as hospitals, clinics, and research institutions.

2. Data Digitization:

o Data is digitized and standardized into compatible formats(e.g.,HL7,FHIR) for easy blockchain integration. 3. Hashing:

o Each piece of data (patient record or research result) is hashed using a cryptographic hashing algorithm (e.g., SHA-256). This ensures that the data can be uniquely identified and its integrity verified without revealing sensitive information.

4. Smart Contract Implementation:

o Smart contracts are created to automate and enforce privacy policies, permissions,andconsentmanagement(e.g.,whocanaccessspecificpatient data or research findings). o Smart contracts can also ensure that researchers or healthcare professionals follow the required protocols before accessing or publishing medical research data.

5. Data Storing:

o The hashed data, along with associated meta data(time stamp, owner's identity, etc.), is stored on the blockchain.

o Only critical metadata and hashes are stored on the blockchain to ensure scalability, while detailed data remains off-chain or stored in decentralized file storage systems (e.g., IPFS).

6. Access Control& Permissioning :

o Permissions for accessing data are set based on user roles(doctor, patient, researcher, etc.) via blockchain-based access control systems.

Patients can grant or revoke access to their medical data through the blockchain using smart contracts.

7. Data Verification and Audit:

o Anyone can verify the authenticity of medical records or research data on the blockchain by checking the hash against the stored data.

o A transparent audit trail allows tracking who accessed or modified data and when.

8. Data Sharing &Research Collaboration:

o Researchers can share anonymized or aggregated medical data through the blockchain, ensuring data integrity and providing animmutable record of data usage.

○ Incentive mechanisms(e.g., tokenization)can encourage data sharing and collaboration.

2. Inputs

- **Patient Health Records**: Medical histories, diagnoses, prescriptions, test results, imaging data, etc.
- **Medical Research Data**: Clinical trials, pharmaceutical research, medical studies, experimental results, patient data anonymized for research.
- **Access Permissions**: Patient consent, researcher access, and healthcare professional roles. ● **Blockchain Network**: Blockchain platform(e.g., Ethereum, Hyperledger, or a custom healthcarefocused blockchain).

3. Outputs

- **Immutable Blockchain Records**: The finalized EHRs and research data are stored on the blockchain, accessible and verifiable.
- **Secure Medical Data Access**: Only authorized users (e.g., doctors, researchers with consent) can access or update patient records or medical research data.
- **Transparent Audit Logs**: A full, transparent record of all actions(access, modifications) taken on the data, visible on the blockchain.
- **Real-time Data Sharing**: Research data can be securely and efficiently shared between institutions, leading to quicker scientific advancements.

- **Data Integrity**: All healthcare data stored on the blockchain is immutable and cannot be altered or deleted without detection.
- **Enhanced Privacy**: Patients maintain control over their health records through private keys and smart contracts, ensuring only authorized parties access their information.
- **Interoperability**: Blockchain facilitates seamless sharing of medical records across different healthcare systems, reducing administrative burden and improving patient care.
- **Trust in Research**: Medical research data on the blockchain is transparent and verifiable, which helps establish trust in the research process and findings.
- **Collaboration and Innovation**: The decentralized nature of blockchain encourages collaboration among hospitals, research labs, pharmaceutical companies, and other stakeholders, accelerating

- medical innovation and improving patient outcomes.

-

Conclusion

By using blockchain for EHR and medical research data, the healthcare industry can address challenges such as data security, privacy, interoperability, and fraud, all while enabling more efficient research collaboration and better patient care.

Usecase 4

EnergyandUtilities – Renewable energy trading which are locally owned deliver benefits that are environmentally relevant for the communities involved

Aim

The aim of this use case is to enable a decentralized and transparent platform for renewable energy trading in local communities using blockchain technology. By leveraging blockchain, communities can trade locally produced renewable energy(such as solar or wind)in a peer-to-peer manner, ensuring fair and transparent transactions, reducing energy waste, and promoting sustainability. The platform aims to empower local communities to manage their energy resources efficiently and benefit economically while reducing their carbon footprint.

Algorithm

1. Energy Generation Tracking

- Install IoT-based smart meters in homes or renewable energy installations (e.g., solar panels or wind turbines)that continuously measure the amount of energy generated.
- Each smart meter records energy production data(in kilowatt-hours)and automatically uploads it to the blockchain ledger at regular intervals.

2. Energy Demand and Trading Platform

- A decentralized blockchain-based platform(e.g., a smart contract system)is used where local producers and consumers can interact.
- Consumers(households or businesses)place orders for renewable energy in the platform's marketplace.

3. Matching Energy Supply with Demand

- The platform uses an automated matching algorithm to match excess renewableenergysupply(fromlocalproducers)withconsumerdemand,based on geographic proximity and energy preferences.
- Smart contracts are created to facilitate real-time and transparent energy transactions.

4. Transaction Validation

- The blockchain verifies each energy transaction by validating the data from the smart meters and ensuring that the energy supplied matches the energy consumed.
- A consensus mechanism(e.g., Proof of Authority or Proof of Stake)ensures the integrity of the transactions.

5. Energy Transfer and Settlement

- Once energy demand is met, the smart contract triggers the transfer of energy credits or tokens from the buyer to the seller.
- Blockchain handles the financial transaction, which may be in the form of digital tokens (e.g., Energy Tokens or local cryptocurrency).

6. Incentive and Reward Distribution

The platform provides an incentive mechanism, rewarding consumers who consume renewable energy and producers who generate it. These incentives could be in the form of renewable energy credits, discounts on energy purchases, or local currency.

- Rewards are tracked and distributed transparently via smart contracts.

o

7. Environmental Impact Tracking

o A third-party service or smart contract is used to track and record the environmental impact of the renewable energy traded(e.g., reduction in CO2 emissions).

o Communities receive transparent reports on the environmental benefits(e.g., tons of CO2saved, energy efficiency gains) through block chain's immutable ledger.

Inputs

1. **Energy Generation Data**: From IoT -enabled smart meters, which continuously send data about the amount of energy generated by local renewable sources.
2. **Energy Demand Data**: Consumer demand data for energy within the local community, including preferences for renewable energy sources.
3. **Energy Credits/ Token System**: Tokens or energy credits used to represent energy units within the blockchain network.
4. **Smart Contract Parameters**: Parameters to govern the matching process(e.g., transaction fees, energy pricing, geographical constraints).
5. **Environmental Impact Metrics**: Data that helps calculate the reduction inCO2 emissions or other sustainability indicators.
6. **Local Regulation Rules**: Legal or regulatory inputs related to local energy trading laws or tariffs.

Outputs

1. **Real-time Energy Transactions**: A record of each energy trade(buyer, seller, energy units traded, time, price) stored on the blockchain.
2. **Energy Consumption and Production Data**: Graphical or tabular data showing energy consumption by consumers and energy generation by local producers.
3. **Energy Credits/Token Transaction History**: Transparent history of energy token transfers on the blockchain.
4. **Environmental Impact Report**: Regular reports showing the environmental benefits of local renewable energy trading, such as reductions in CO2 emissions.
5. **Incentive Distribution**: A report of their wards or tokens distributed to community members as incentives for participating in the renewable energy market.

Results

1. **Increased Local Energy Autonomy**: Communities gain more control over their energy production and consumption, reducing reliance on external energy providers and enhancing resilience.
2. **Transparency and Trust**: Blockchain ensures that all energy trades are transparent, tamperproof, and verifiable by all participants, reducing fraud and disputes.
3. **Economic Benefits**: Local producers benefit from the ability to monetize excess energy, while consumers may receive renewable energy at allowed cost compared to traditional grid prices.
4. **Environmental Impact**: Reduction in carbon emissions as more communities transition to locally produced, renewable energy.
5. **Community Empowerment**: Blockchain-based energy trading fosters community engagement and sustainability by allowing individuals to become active participants in energy management and local environmental protection.

6. **Efficient Energy Market**: A peer-to-peer market with automated smart contracts ensuresenergyisefficientlytradedbasedonreal-timedemandandsupply,reducing waste and optimizing energy distribution.

Use Case 5

Blockchain in Real-estate – The real estate market currently faces several issues related to housing affordability, rising rate and economy, many of which cannot be addressed with blockchain

Aim

The aim of this use case is to explore how blockchain technology can be applied to the real estate sector to improve transparency, reduce fraud, enhance transaction efficiency, and provide solutions to some of the challenges related to housing affordability, rising costs, and market in efficiencies. However, it also addresses the limitations of blockchain in fully solving these challenges, particularly in the context of economic factors and external market forces.

Algorithm

1. Property Ownership Registration

o Each property is assigned a unique identifier and its details (e.g., location, owner, transaction history) are recorded on the blockchain in an immutable ledger.

o A digital token or smart contract is used to represent property ownership, ensuring that the transfer of ownership can be tracked and verified securely.

2. Smart Contract for Real Estate Transactions

o When a property is bought or sold, a smart contract is created that includes all necessary transaction details (price, payment terms, buyer/seller info, etc.).

o The smart contract ensures that once predefined conditions(e.g., payment confirmation, property inspection) are met, the transaction is executed automatically, transferring ownership to the buyer and funds to the seller. 3. Decentralized Marketplace

o A blockchain- powered market place for real estate listings allows buyers and sellers to connect directly, reducing the need for intermediaries (e.g., real estate agents, brokers).

o Listings include verified property data, owner history, and pricing, making it easier for buyers to assess properties.

4. Tokenization of Property Assets

o Real estate assets can be tokenized, allowing fractional ownership of properties.Investorscanbuytokensrepresentingsharesofaproperty,making it more accessible to smaller investors.

o These tokens are stored on the blockchain and represent a claim to a portion of the property's income (e.g., rental income) or future appreciation.

5. Property Lease and Rental Agreements

o Blockchain-based smart contracts can automate rental agreements, ensuring timelypayments,compliancewithterms(e.g.,maintenance),andeasydispute resolution.

Tenants and landlords can interact in a transparent and secure manner, with rent payments and terms recorded on the blockchain.

○

6. Verification and Authentication

○ Blockchain ensures that all property documents, including titles, deeds, and certificates, are authentic and immutable. This reduces the risk of fraud or disputes over property ownership. ○ Third-party verifiers(e.g., notaries or legal professionals)can be integrated into the blockchain to certify transactions and documents. 7. MarketTransparency and Data Sharing

○ Blockchain provides a transparent ledger for historical

○ real estate transactions, making it easier for potential buyers and investors to evaluate trends, pricing, and market behaviour.

○ Open, accessible data can help regulators monitor the market and take necessary actions to prevent price manipulation or market bubbles.

Inputs

1. **Property Data**: Basic details about the property(e.g., address, square footage, ownership history, legal documentation).
2. **Transaction Data**: Data about the sale, purchase, or rental of properties(e.g., buyer/seller details, price, payment terms).
3. **Smart Contract Parameters** :Terms and conditions for transactions (e.g., timelines, conditions for payment, contingencies).
4. **Tokenized Property Data** :Information regarding the tokenization of properties for fractional ownership (e.g., number of tokens, share price).
5. **Legal and Regulatory Data**: Information about local property laws ,taxes, or zoning regulations that may affect property transactions.
6. **User Data**: Buyer, seller, and tenant profiles(e.g., credit worthiness, identity verification, payment history).

Outputs

1. **Property Ownership Records**: Immutable records of ownership and transaction history, stored on the blockchain.
2. **Smart Contract Execution Logs**: Logs showing when certain conditions in a smart contract are met (e.g., payment received, ownership transferred).
3. **Transaction Verification and Audit Trail**: Transparent and verifiable transaction histories, accessible to relevant parties (e.g., regulators, buyers, sellers).
4. **Tokenized Property Shares**: Blockchain tokens representing fractional ownership in real estate properties.
5. **Lease and Rental Contracts**: Smart contracts that automatically enforce the terms of lease agreements (e.g., monthly rent payment).
6. **Market Data and Insights**: Data analytics showing trends in property prices, transaction volumes, and buyer/seller activity.

Results

1. **Increased Transparency**: Blockchain provides an immutable and transparent record of real estate transactions, reducing the risk of fraud, double-selling, and other issues.

2. **Faster Transactions**: Smart contracts and automated processes reduce the time required for property transactions and rental agreements, eliminating the need for intermediaries.
3. **Lower Transaction Costs**: With fewer intermediaries (e.g., agents, notaries), transaction fees are reduced, making real estate more affordable for buyers and sellers.
4. **Improved Security**: Property titles and transactions stored on the blockchain are secure and verifiable, minimizing the risk of identity theft or fraudulent property sales.
5. **Increased Accessibility**: The tokenization of real estate allows smaller investors to participate in the market by purchasing fractional ownership in high-value properties.
6. **Better Market Efficiency**: A decentralized market place for real estate an enhance market liquidity by enabling direct buyer-seller interactions, potentially reducing housing market inefficiencies.

Conclusion

While blockchain has the potential to address several issues in the real estate market—such as increasing transparency, reducing transaction costs, and improving market efficiency—there are significant challenges it cannot fully overcome. For example:

- **Housing Affordability**: Blockchain cannot directly address the economic factors contributing to rising housing prices, such as inflation, wage stagnation, or government policies. The root causes of affordability issues are primarily economic and regulatory, and blockchain, by itself, cannot solve these problems.

- **Economic and Market Forces**: Blockchain can streamline transactions and improve trust in the market, but it cannot control market fluctuations or prevent issues like property bubbles, speculative investments, or economic recessions that affect real estate prices.
- **Regulatory Hurdles**: Real estate markets are highly regulated, and blockchain adoption may face resistance due to legal complexities and the need for regulatory updates to accommodate blockchain-based transactions.