BCSE324L – Foundations of Blockchain Technology

AUTOMATED INSURANCE CLAIMS WITH SMART CONTRACTS

21BCE2405 TORAN V ATHANI

Under the Supervision of

NIHA K
Assistant Professor Sr. Grade 2
School of Computer Science and Engineering (SCOPE)

B.Tech.
in
Computer Science and Engineering

School of Computer Science and Engineering

November 2024

ABSTRACT

In recent years, the insurance industry has faced challenges such as inefficient claims processing, susceptibility to fraud, high operational costs, and lack of transparency. Automated insurance claims processing using smart contracts offers a promising solution by leveraging blockchain technology to streamline workflows, minimize human intervention, and improve data security. This document explores the application of smart contracts in the insurance domain, focusing on a specific use case: automated claim handling for [chosen claim type, Health Insurance].

Smart contracts, self-executing code stored on a blockchain, enable automatic validation, approval, and settlement of insurance claims without the need for intermediaries. Through the use of Solidity programming and Ethereum's blockchain, this report demonstrates how conditions specified in an insurance policy can be coded directly into a smart contract. Upon fulfillment of these conditions, such as proof of delay in the case of flight insurance or accident verification in auto claims, the smart contract executes the claim autonomously. This automation leads to reduced processing times, minimized human intervention, enhanced transparency, and improved customer satisfaction.

The report provides a detailed explanation of the application domain, highlights the benefits of adopting smart contracts for claim processing, and outlines the end-to-end implementation of the chosen use case. With Solidity code examples, deployment steps, and screenshots, it demonstrates how smart contracts can automatically verify claim conditions, authorize payouts, and reduce processing times, ultimately enhancing customer satisfaction and operational efficiency. This approach underscores the potential of blockchain-based automation to transform traditional insurance processes, offering insights into practical deployment in real-world insurance applications.

Keywords - Blockchain technology, Smart contracts, Automated claims processing, Insurance automation, Ethereum, Solidity, Decentralized insurance, Fraud prevention, Claims verification, Flight delay insurance, Auto insurance claims, Transparency in insurance, Decentralized applications (DApps), Self-executing contracts, Insurance industry transformation, Digital insurance policies, Customer satisfaction in insurance, Real-time claims settlement, Smart contract deployment, Blockchain-based automation.

TABLE OF CONTENTS

1. APPLICATION DOMAIN: HEALTH INSURANCE

1.1 Introduction to Health Insurance Domain

Health insurance plays a crucial role in providing individuals with financial protection against high medical costs. It is a key component of the broader insurance sector and is designed to cover medical expenses incurred due to illnesses, injuries, or preventive care. Health insurance companies typically operate as intermediaries, collecting premiums from policyholders and reimbursing medical providers based on the terms of a health insurance plan.

One of the primary functions of health insurance is claims processing, where insured individuals submit claims for medical services received. The efficiency of claims processing is paramount to both insurers and policyholders. However, current methods of claims submission and approval are often manual and involve multiple parties, including healthcare providers, insurers, and claims adjusters. These intermediaries introduce delays, increase the risk of errors, and add to operational costs.

The implementation of smart contracts within the health insurance domain promises to streamline these processes by automating decision-making, reducing manual intervention, and enhancing overall system transparency. Smart contracts, based on blockchain technology, offer a decentralized, secure platform to handle claims in an efficient and transparent manner. By automating processes such as claims verification and payment approval, these technologies can provide faster, more reliable, and less error-prone claims handling, benefiting both insurers and policyholders.

1.2 Current Challenges in Health Insurance Claims

Despite advancements in the health insurance industry, several challenges persist in the claims process:

1. **Fraud Detection**: One of the most significant concerns in the insurance sector is fraud. Fraudulent claims can lead to significant financial losses for insurance companies. Detecting fraud often requires extensive manual checks, which can be time-consuming and error-prone.
2. **Delays in Processing**: Health insurance claims are often subject to long delays due to the need for verification of medical records, billing details, and policy coverage. These delays can cause frustration for policyholders, particularly in urgent medical situations.
3. **Administrative Overheads**: Traditional claims processes require a large number of administrative resources to handle paperwork, verify claims, and ensure that all parties are in agreement. This creates overhead costs for insurance companies, which can be passed on to consumers in the form of higher premiums.
4. **Complexity of Terms and Conditions**: Health insurance policies are often complex, with numerous clauses and exclusions that can be difficult for both policyholders and providers to understand. Misinterpretation of these terms can lead to incorrect claim denials or approvals.
5. **Data Privacy and Security**: The healthcare industry handles sensitive personal information, making it a target for cyberattacks. Securing this data while ensuring compliance with privacy

regulations (such as HIPAA in the United States) is a significant challenge.

1.3 Benefits of Using Smart Contracts in Health Insurance

Smart contracts offer several advantages when applied to health insurance claims processing, addressing many of the challenges outlined above:

1. **Automation of Claims Verification:** Smart contracts can automatically verify claims based on predefined rules. For example, once a claim is submitted, the contract can check whether the treatment is covered under the policy, whether the medical service provider is eligible for reimbursement, and whether the claim amount is accurate. This automation reduces the need for manual review and accelerates the claims process.

2. **Enhanced Transparency:** Blockchain's inherent transparency allows all participants in the claims process to view and verify the transactions in real-time. Once a claim is processed, the details (such as the reimbursement amount and claim approval status) are recorded on the blockchain, ensuring that no one party can alter the terms without the knowledge of others. This reduces disputes and builds trust among policyholders, healthcare providers, and insurers.

3. **Cost Reduction:** By eliminating the need for intermediaries such as claims adjusters and manual review processes, smart contracts can significantly reduce administrative costs. This results in faster claim resolution, reducing operational overheads for insurers and improving the overall customer experience.

4. **Real-time Claims Processing:** With smart contracts, health insurance claims can be processed almost instantly. Once the conditions of the contract are met, payment can be automatically triggered and executed. This removes the delays commonly seen with traditional claims processing.

5. **Error Reduction:** Human error is a major factor in the delays and inaccuracies in traditional claims processing. Smart contracts are programmed with predefined rules that eliminate the chance of mistakes that may occur in manual decision-making.

6. **Increased Security:** Blockchain's decentralized nature ensures that data is stored securely, with encryption methods that protect sensitive medical information. This enhances data privacy and reduces the risk of cyberattacks or unauthorized access to health-related data.

By integrating blockchain and smart contracts, the health insurance sector can experience significant improvements in claims management, from faster claim approvals to reduced fraud and operational costs.

2. USE CASE: AUTOMATED HEALTH INSURANCE CLAIMS PROCESSING

2.1 Introduction to the Use Case

Automated health insurance claims processing refers to the application of blockchain technology and smart contracts to handle the submission, validation, approval, and payment of health insurance claims without the need for extensive human intervention. In this use case, policyholders, healthcare providers, and insurance companies interact through a secure, transparent, and efficient system powered by smart contracts.

The claims process typically follows these steps:

1. **Claim Submission**: When a policyholder receives medical treatment, the healthcare provider submits a claim to the insurer via a secure digital platform. This claim contains information such as the type of treatment provided, costs, and the patient's insurance details.
2. **Eligibility Verification**: The smart contract automatically checks whether the patient's policy covers the treatment received and verifies the details against the policy's terms. If any information is missing or incorrect, the contract rejects the claim.
3. **Claim Approval/Denial**: If the claim meets the policy conditions, the smart contract approves it. If there are issues, such as the treatment not being covered or exceeding policy limits, the claim is automatically denied.
4. **Payment Processing**: Once a claim is approved, the smart contract triggers the release of the payment to the healthcare provider, or in some cases, to the insured individual, based on the policy terms.
5. **Claim Completion**: After payment is made, the transaction is recorded on the blockchain, ensuring transparency and immutability.

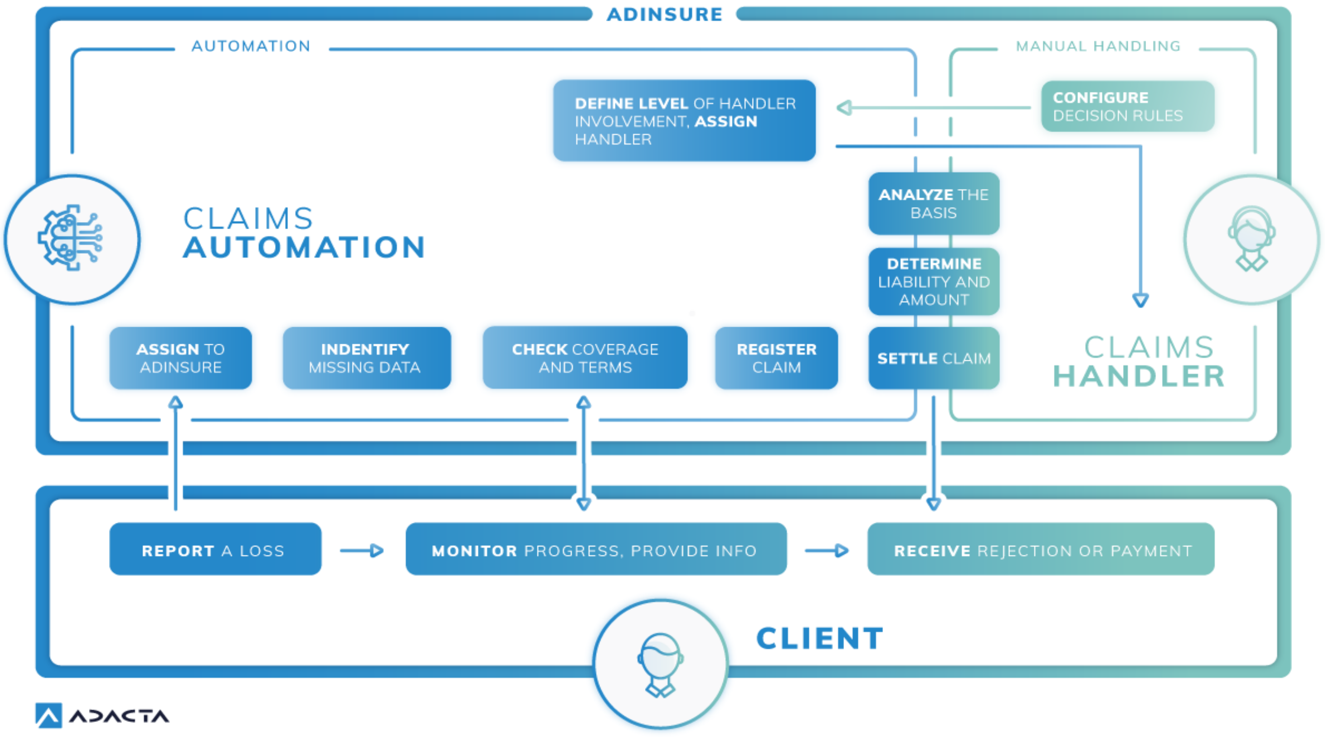
This automated process ensures quicker turnaround times, reduces administrative overhead, and minimizes human error.

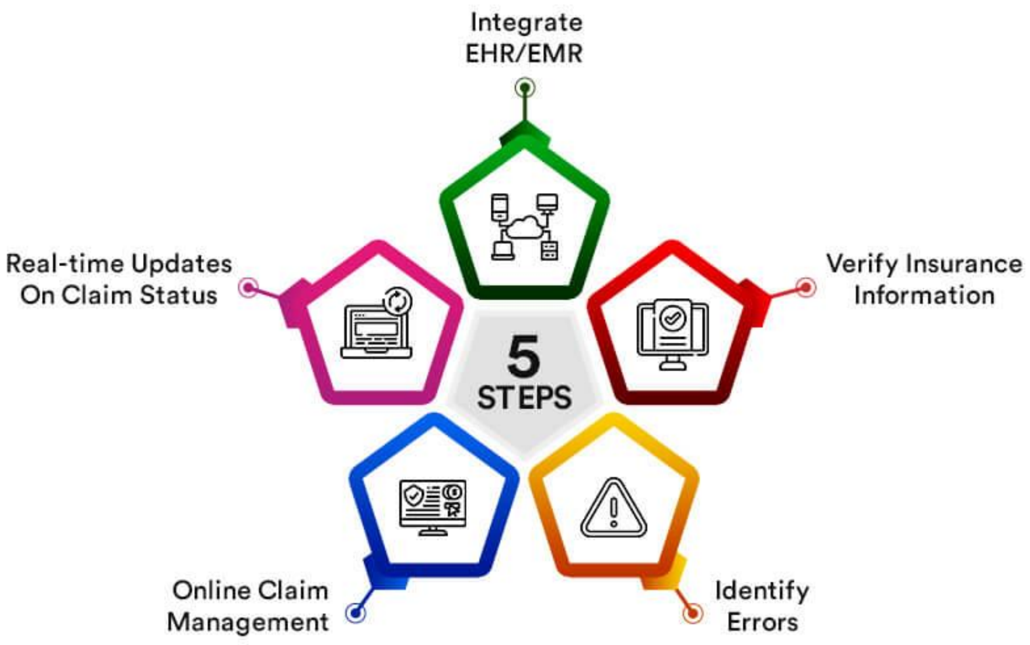
2.2 Key Stakeholders and Workflow

The key stakeholders in the automated claims processing system include:

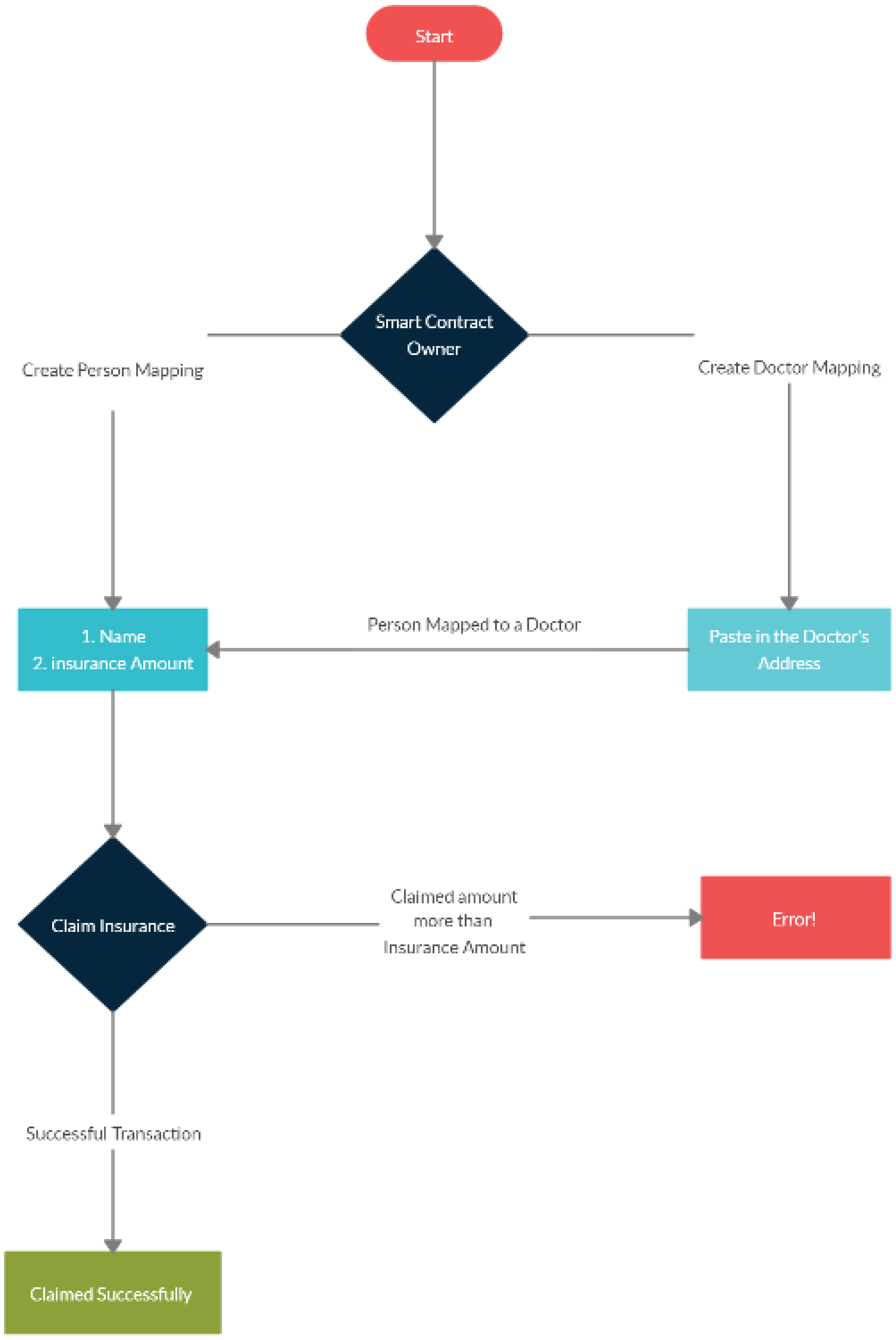
1. **Insurer**: The insurance company is responsible for validating claims and making payments according to the policy terms. Smart contracts allow the insurer to automatically approve or deny claims without the need for extensive manual intervention.
2. **Insured Individual**: The policyholder submits the claim for the medical treatment received. They benefit from quicker claim approval and payment processing, improving their experience with the insurer.
3. **Healthcare Providers**: These include hospitals, clinics, and doctors who provide the medical services and submit claims for reimbursement. Healthcare providers are assured of timely payments through the automation of the claims process.

4. **Regulators**: Regulators ensure that both insurers and healthcare providers comply with relevant laws and regulations, particularly in terms of data privacy (e.g., HIPAA in the U.S.). Blockchain and smart contracts can help maintain audit trails and ensure compliance.





Workflow:


Start
Smart Contract Owner
Create Person Mapping
Create Doctor Mapping
1. Name
2. insurance Amount
Person Mapped to a Doctor
Paste in the Doctor's Address
Claim Insurance
Claimed amount more than Insurance Amount
Error!
Successful Transaction
Claimed Successfully

- When a patient receives medical treatment, the healthcare provider submits a digital claim.
- The smart contract verifies the patient's eligibility and whether the treatment is covered.
- If the claim is valid, the contract approves it and triggers payment to the provider.
- If the claim is denied, the system provides a reason (e.g., exceeding coverage limits) and alerts the insured party.
- The transaction details are recorded on the blockchain, ensuring transparency for both the insurer and insured.

Smart contracts can trigger events automatically based on conditions, such as hospitalization dates, treatment completion, or claim submission, to ensure smooth and real-time processing.

2.3 Smart Contract Functionality Requirements

For the smart contract to function effectively in this use case, the following core functionalities are essential:

1. **Eligibility Verification**:
   The smart contract must check if the patient's policy covers the treatment, considering factors like the type of treatment, treatment cost, and whether it falls under the patient's coverage. This can be done by querying external data sources, such as the insurer's database, which may require oracles to feed real-time data into the smart contract.

   Pros: Automated eligibility checks reduce the need for manual intervention, speeding up the process.
   Cons: Reliance on external oracles can introduce risks if the data source is unreliable or compromised.

2. **Claims Approval and Denial Conditions**:
   The smart contract will automatically approve or deny claims based on predefined criteria, such as the amount covered by the policy, exclusions, or treatment limitations. The contract will evaluate the claim against these conditions, ensuring a consistent and transparent decision-making process.

   Pros: Increases transparency and reduces human error. Claims are processed faster, with no delays from manual review.
   Cons: Overly rigid rules might reject claims that could be valid under special circumstances, requiring some flexibility in contract design.

3. **Payment Release**:
   Once the claim is approved, the smart contract can release payment to the healthcare provider or reimburse the patient, as per the policy agreement. Payment can be made automatically using cryptocurrencies or fiat through a blockchain-powered payment system.

   Pros: Instant payments reduce wait times for both healthcare providers and patients.
   Cons: Regulatory concerns about the use of cryptocurrency for payments, and potential integration complexities with existing banking systems.

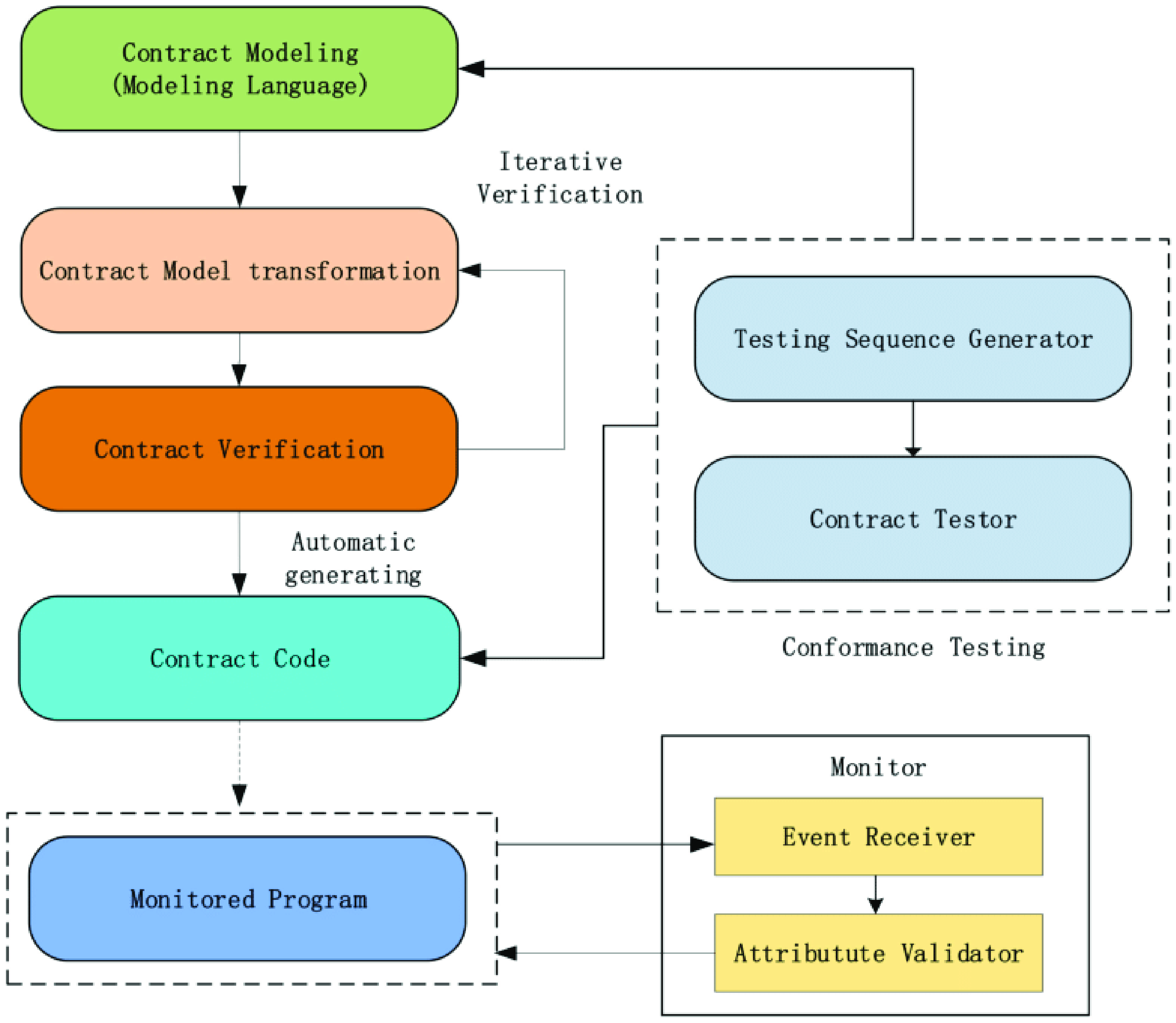
4. **Data Sources and Integration**:
   The smart contract must integrate with various data sources for eligibility verification, claims validation, and medical records (e.g., electronic health records or EHR systems). This ensures the contract makes accurate decisions based on the most up-to-date and accurate information.

   Pros: Real-time integration with healthcare systems ensures accurate claims processing and reduces errors.
   Cons: Data integration and interoperability with existing healthcare systems can be challenging and expensive.

5. **Security Considerations**:
   Smart contracts must ensure that sensitive patient data is protected throughout the process. The use of encryption for sensitive information, alongside compliance with data privacy regulations like HIPAA, is essential. Blockchain provides transparency, but sensitive data should not be stored directly on the blockchain.

Pros: Blockchain's decentralized and encrypted nature enhances security and transparency.
Cons: Protecting personal health data within a smart contract may require additional privacy layers, such as zero-knowledge proofs, which can complicate implementation.

By integrating these functionalities into the smart contract, health insurance claims processing becomes more efficient, transparent, and secure, with minimal human intervention.



3. IMPLEMENTATION STEPS FOR AUTOMATED HEALTH INSURANCE CLAIMS PROCESSING

3.1 Setting Up the Blockchain Environment

For this use case, we will use Ethereum, a popular blockchain platform known for its smart contract capabilities, and Solidity for writing the smart contract. Ethereum is ideal because it supports decentralized applications and allows for the development and deployment of smart contracts.

Instructions for Setting Up Ethereum Environment:

1. **Install Node.js and npm**
   Ethereum development tools like Truffle and Ganache require Node.js and npm (Node Package Manager).

   ```
   node -v

   npm -v
   ```

2. **Install Truffle Framework**
   Truffle is a popular framework for Ethereum development. It helps in compiling, deploying, and testing smart contracts.

   ```
   npm install -g truffle
   ```

3. **Install Ganache**
   Ganache is a local Ethereum blockchain that allows developers to test their contracts in a private environment.

4. **Create a New Truffle Project**
   Once Truffle is installed, create a new project directory:

   ```
   mkdir health-insurance

   cd health-insurance

   truffle init
   ```

5. **Install Web3.js**
   Web3.js is a library used to interact with the Ethereum blockchain. Install it using npm:

   ```
   npm install web3
   ```

6. **Start Ganache**
   Launch Ganache and set up a local Ethereum blockchain instance.

7. **Configure Truffle**
   Open the truffle-config.js file and configure it to connect to Ganache:

```
module.exports = {
  networks: {
    development: {
      host: "127.0.0.1",
      port: 7545,
      network_id: "*"
    }
  },
  compilers: {
    solc: {
      version: "0.8.0" // Specify the Solidity version
    }
  }
};
```

3.2 Creating the Smart Contract

Contract Code:

Here's the Solidity code that implements the health insurance claim process:

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

pragma solidity ^0.8.0;

contract HealthInsurance{

    address public SmartContractOwner; // The owner of the smart contract

    struct person{
        bool authorised;
        string name;
    uint insuranceAmount;
}


mapping(address => person) public personmapping; // Mapping of user addresses to their details
mapping(address => bool) public doctormapping; // Mapping doctor address to boolean for
  validation


constructor() {
    SmartContractOwner = msg.sender; // Assign the contract deployer as the owner
}


modifier onlySmartContractOwner(){
    require(SmartContractOwner == msg.sender, "Only the contract owner can call this
  function");
    _;
}


// Function to register a doctor address
function setDoctor(address _address) public onlySmartContractOwner{
    require(!doctormapping[_address], "Doctor is already registered");
    doctormapping[_address] = true;
}


// Function to register a person (policyholder)
function setPerson(string memory _name, uint _insuranceAmount) public
  onlySmartContractOwner returns (address){
    address uid = address(bytes20(sha256(abi.encodePacked(msg.sender, block.timestamp)))); //
  Unique ID for the user
    require(!personmapping[uid].authorised, "Person already registered");
    personmapping[uid].authorised = true;
    personmapping[uid].name = _name;
    personmapping[uid].insuranceAmount = _insuranceAmount;


    return uid; // Return the unique user ID
    }

    // Function to process an insurance claim
    function claimInsurance(address _uid, uint _insuranceAmountUsed) public returns (string
      memory){
        require(!doctormapping[msg.sender], "Doctor cannot claim insurance");
        if(personmapping[_uid].insuranceAmount < _insuranceAmountUsed){
            revert("Insufficient insurance balance");
        }

        // Debit the insurance amount from the user's account
        personmapping[_uid].insuranceAmount -= _insuranceAmountUsed;
        return "Amount debited from your insurance";
    }
}
```

Explanation of Code:

- **SmartContractOwner:** The contract deployer is designated as the owner of the smart contract.
- **person Struct:** This structure stores information for each user (policyholder), including whether they are authorized, their name, and their available insurance amount.
- **personmapping Mapping:** Maps each user's address to their personal information.
- **doctormapping Mapping:** Maps a doctor's address to a boolean to validate authorized doctors.
- **setDoctor():** Registers a doctor by adding their address to the doctormapping.
- **setPerson():** Registers a person (policyholder) by generating a unique address ID based on the sender’s address and timestamp, then mapping the user's information to it.
- **claimInsurance():** Processes insurance claims by verifying if the policyholder has enough insurance balance. If valid, it debits the claimed amount from the policyholder's balance.

Smart Contract Functions:

1. **setDoctor():** Registers an authorized doctor.
2. **setPerson():** Registers a new person (policyholder).
3. **claimInsurance():** Processes an insurance claim by debiting the specified amount from the policyholder's balance.

3.3 Deploying and Testing the Smart Contract

Deploy the Contract:

1. **Create a Migration Script:** In the migrations folder, create a new deployment script *2_deploy_contracts.js*:

```
const HealthInsurance = artifacts.require("HealthInsurance");

module.exports = function (deployer) {
  deployer.deploy(HealthInsurance);
};
```

2. **Deploy the Contract to Local Network:** Run the following command to deploy the contract:

```
truffle migrate --network development
```

Testing the Contract:

Write tests in the test/healthInsurance.js file to ensure functionality:

```
const HealthInsurance = artifacts.require("HealthInsurance");

contract("HealthInsurance", accounts => {
  it("should register a person and process a claim", async () => {
    let instance = await HealthInsurance.deployed();

    // Register a person
    const personAddress = await instance.setPerson("John Doe", 1000);

    // Register a doctor
    await instance.setDoctor(accounts[1]);

    // Attempt a valid claim
    const result = await instance.claimInsurance(personAddress, 500, { from: accounts[1] });
    assert.equal(result, "Amount debited from your insurance");
    // Check remaining insurance amount
    const person = await instance.personmapping(personAddress);
    assert.equal(person.insuranceAmount, 500);
  });
});
```

Run the test:

```
truffle test
```

3.4 Integrating with Front-End or Application Interface

Connecting Smart Contract with Web Interface:

To connect the smart contract to a front-end, we will use Web3.js. Here's how you can integrate the contract with a simple front-end interface.

1. **Install Web3.js:**

```
npm install web3
```

2. **Web3.js Integration:** Use Web3.js to interact with the deployed contract from a front-end application. Below is an example of how you can use Web3.js to interact with the contract in a front-end interface:

```
const Web3 = require('web3');
const web3 = new Web3(window.ethereum);
await window.ethereum.enable();

const contractAddress = "0xYourDeployedContractAddress"; // Replace with your contract address
const abi = [...] // ABI of the contract

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

// Example of submitting a claim
await contract.methods.claimInsurance(personAddress, 500).send({ from: accounts[1] });
```

User Interaction Example:

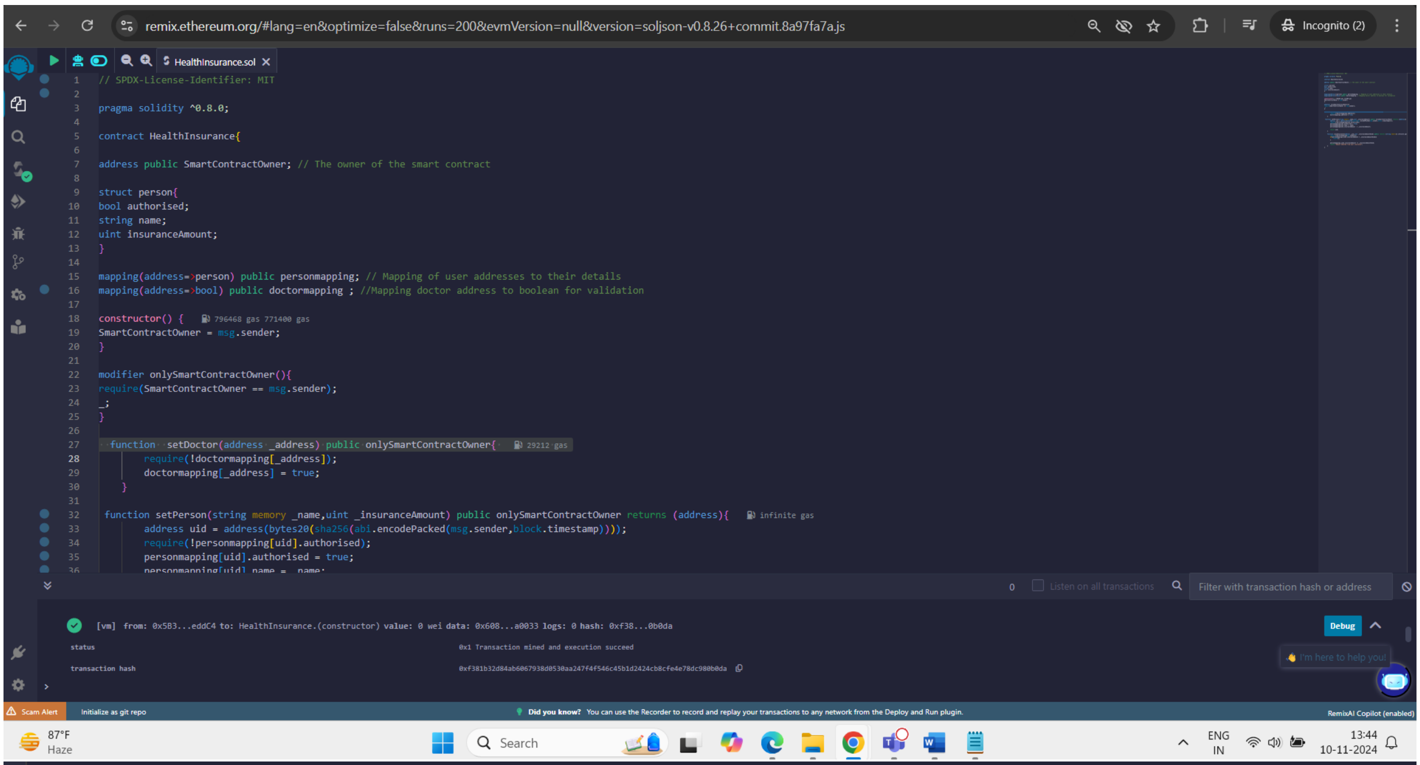
- A user submits a claim via the front-end interface.
- After submission, the smart contract processes the claim, debiting the required amount from the policyholder's balance.

3.5 Simulating Claim Scenarios

1. **Claim Approval:** If the insurance balance is sufficient, the claim is approved, and the amount is debited from the user's account.
2. **Claim Rejection:** If the claim amount exceeds the available insurance balance, the claim is rejected.
3. **Payment Release:** After successful claim processing, the remaining balance is updated.

4. IMPLEMENTATION SCREENSHOTS

Health Insurance :



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

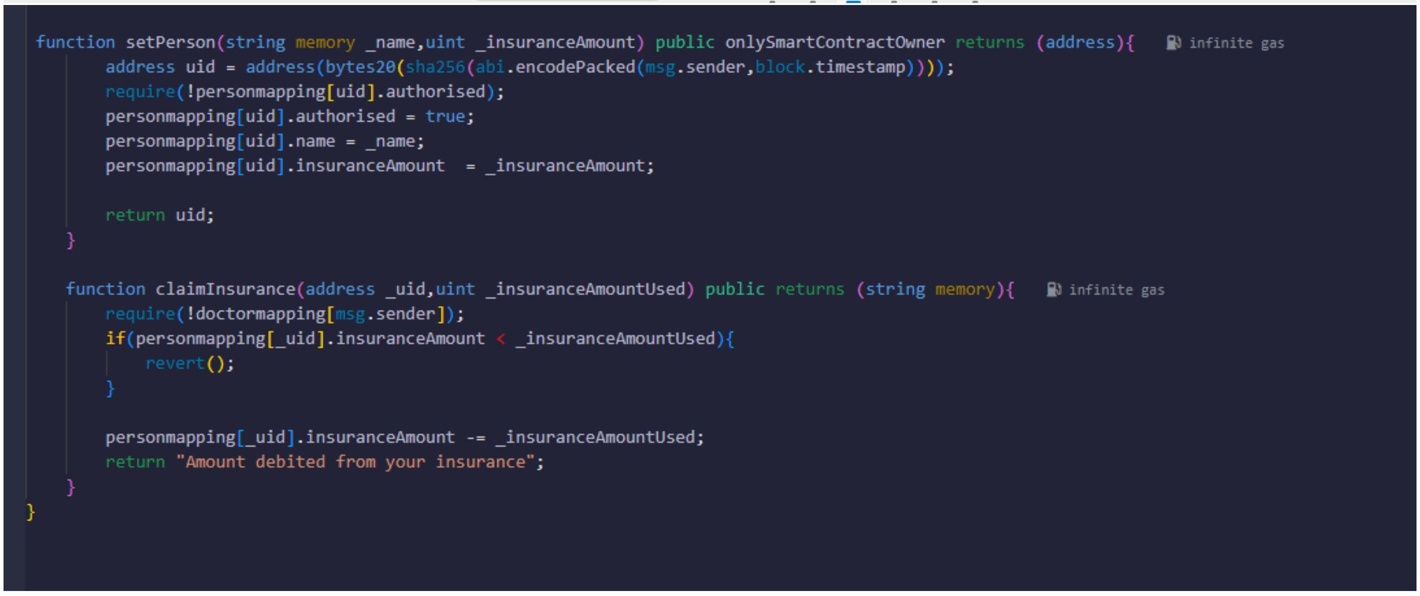
HealthInsurance.sol

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

pragma solidity ^0.8.0;

contract HealthInsurance{

address public SmartContractOwner; // The owner of the smart contract

struct person{
bool authorised;
string name;
uint insuranceAmount;
}

mapping(address=>person) public personmapping; // Mapping of user addresses to their details
mapping(address=>bool) public doctormapping ; //Mapping doctor address to boolean for validation

constructor() {    796468 gas 771400 gas
SmartContractOwner = msg.sender;
}

modifier onlySmartContractOwner(){
require(SmartContractOwner == msg.sender);
_;
}

  function  setDoctor(address _address) public onlySmartContractOwner{    29212 gas
        require(!doctormapping[_address]);
        doctormapping[_address] = true;
    }

 function setPerson(string memory _name,uint _insuranceAmount) public onlySmartContractOwner returns (address){    infinite gas
        address uid = address(bytes20(sha256(abi.encodePacked(msg.sender,block.timestamp))));
        require(!personmapping[uid].authorised);
        personmapping[uid].authorised = true;
        personmapping[uid].name = _name;
```

```
0    Listen on all transactions    Filter with transaction hash or address

[vm] from: 0x5B3...eddC4 to: HealthInsurance.(constructor) value: 0 wei data: 0x608...a0033 logs: 0 hash: 0xf38...0b0da
status              0x1 Transaction mined and execution succeed
transaction hash    0xf381b32d84ab6067938d0530aa247f4f546c45b1d2424cb8cfe4e78dc980b0da
```

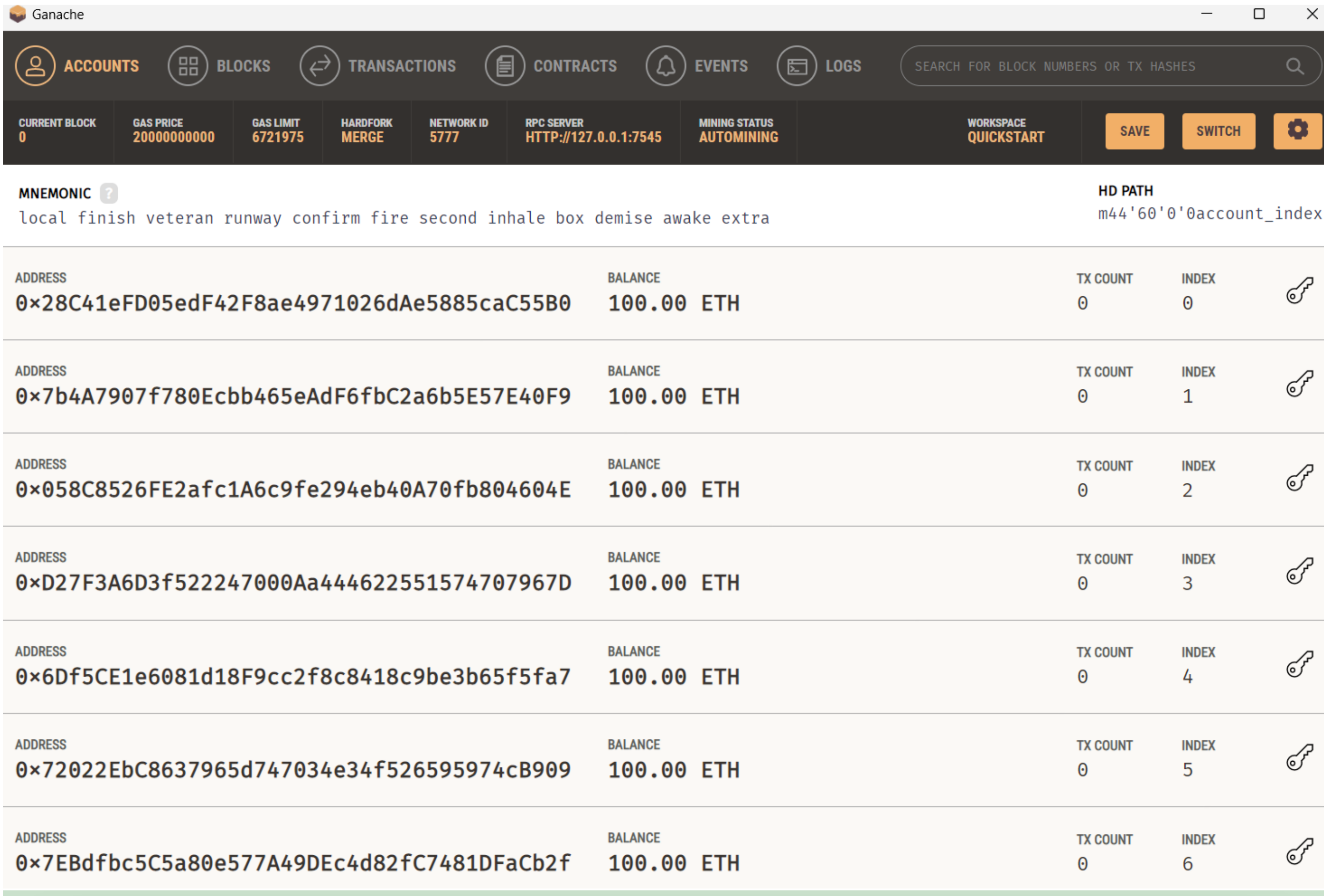
Scam Alert   Initialize as git repo   Did you know? You can use the Recorder to record and replay your transactions to any network from the Deploy and Run plugin.   RemixAI Copilot (enabled)

87°F Haze   Search   ENG IN   13:44 10-11-2024



```
function setPerson(string memory _name,uint _insuranceAmount) public onlySmartContractOwner returns (address){    infinite gas
        address uid = address(bytes20(sha256(abi.encodePacked(msg.sender,block.timestamp))));
        require(!personmapping[uid].authorised);
        personmapping[uid].authorised = true;
        personmapping[uid].name = _name;
        personmapping[uid].insuranceAmount  = _insuranceAmount;

        return uid;
    }

    function claimInsurance(address _uid,uint _insuranceAmountUsed) public returns (string memory){    infinite gas
        require(!doctormapping[msg.sender]);
        if(personmapping[_uid].insuranceAmount < _insuranceAmountUsed){
            revert();
        }

        personmapping[_uid].insuranceAmount -= _insuranceAmountUsed;
        return "Amount debited from your insurance";
    }
}
```

Output :

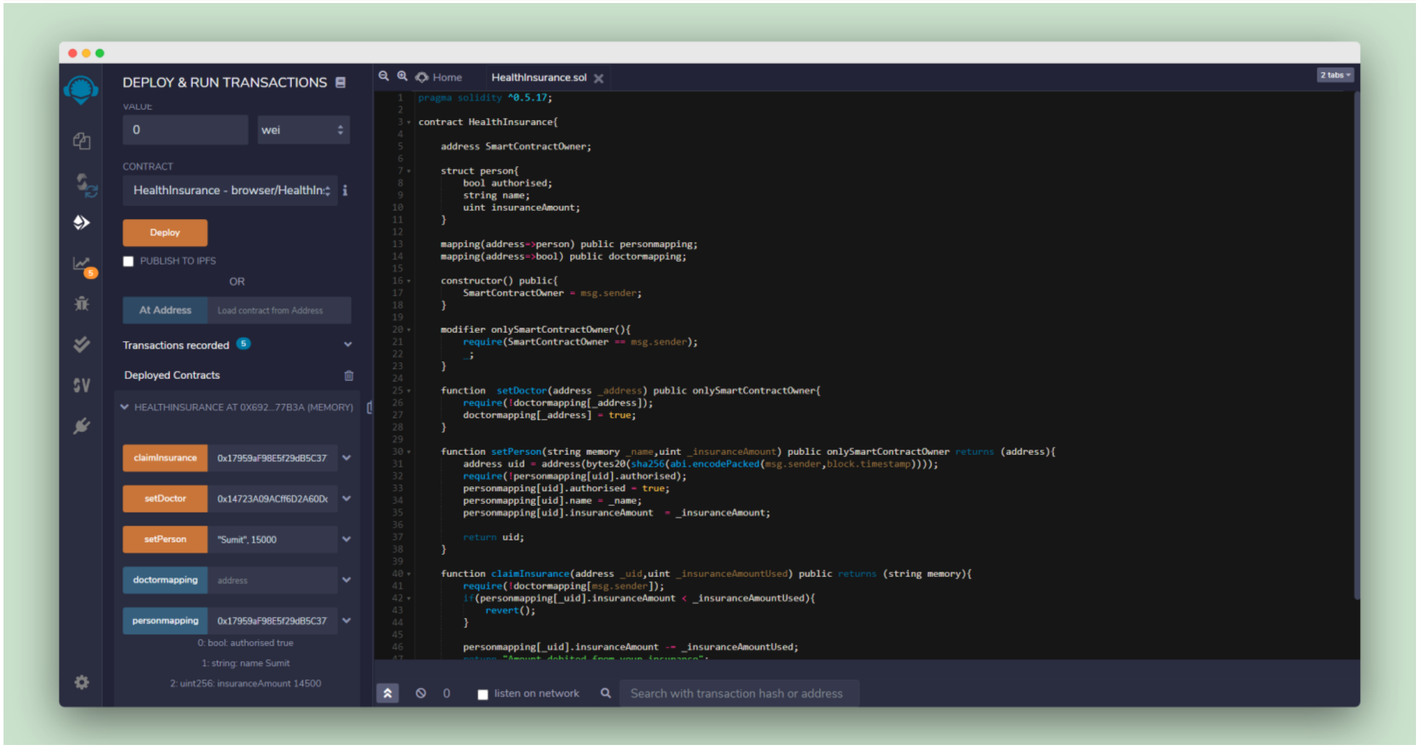


CURRENT BLOCK 0 | GAS PRICE 20000000000 | GAS LIMIT 6721975 | HARDFORK MERGE | NETWORK ID 5777 | RPC SERVER HTTP://127.0.0.1:7545 | MINING STATUS AUTOMINING | WORKSPACE QUICKSTART

MNEMONIC
local finish veteran runway confirm fire second inhale box demise awake extra

HD PATH
m44'60'0'0account_index

ADDRESS	BALANCE	TX COUNT	INDEX
0×28C41eFD05edF42F8ae4971026dAe5885caC55B0	100.00 ETH	0	0
0×7b4A7907f780Ecbb465eAdF6fbC2a6b5E57E40F9	100.00 ETH	0	1
0×058C8526FE2afc1A6c9fe294eb40A70fb804604E	100.00 ETH	0	2
0×D27F3A6D3f522247000Aa444622551574707967D	100.00 ETH	0	3
0×6Df5CE1e6081d18F9cc2f8c8418c9be3b65f5fa7	100.00 ETH	0	4
0×72022EbC8637965d747034e34f526595974cB909	100.00 ETH	0	5
0×7EBdfbc5C5a80e577A49DEc4d82fC7481DFaCb2f	100.00 ETH	0	6


DEPLOY & RUN TRANSACTIONS
Deploy
PUBLISH TO IPFS
OR
At Address
Load contract from Address
Home
HealthInsurance.sol
listen on network
Search with transaction hash or address

VALUE 0 wei

CONTRACT HealthInsurance - browser/HealthIn

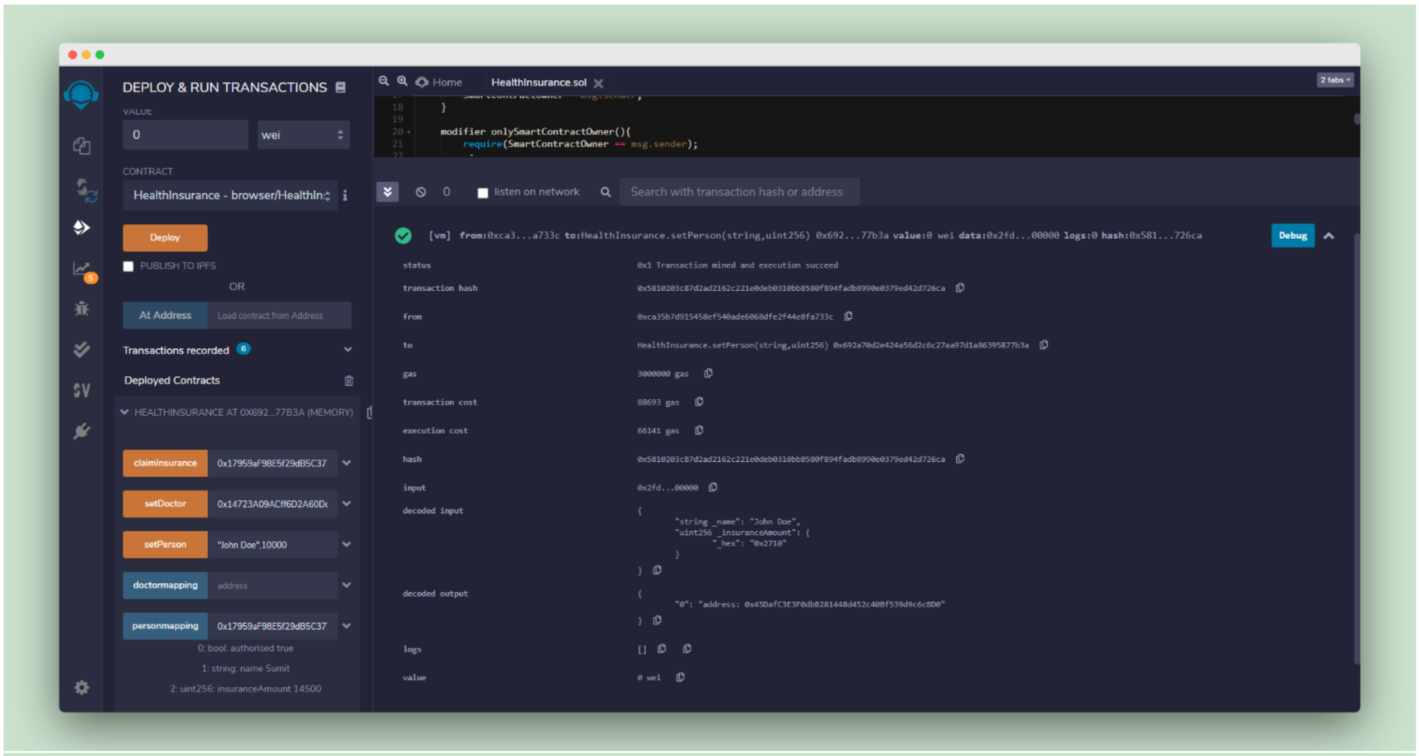
Transactions recorded 5

Deployed Contracts

HEALTHINSURANCE AT 0X692...77B3A (MEMORY)

```
claimInsurance    0x17959aF98E5f29dB5C37
setDoctor         0x14723A09ACff6D2A60Dc
setPerson         "Sumit", 15000
doctormapping     address
personmapping     0x17959aF98E5f29dB5C37
0: bool: authorised true
1: string: name Sumit
2: uint256: insuranceAmount 14500
```

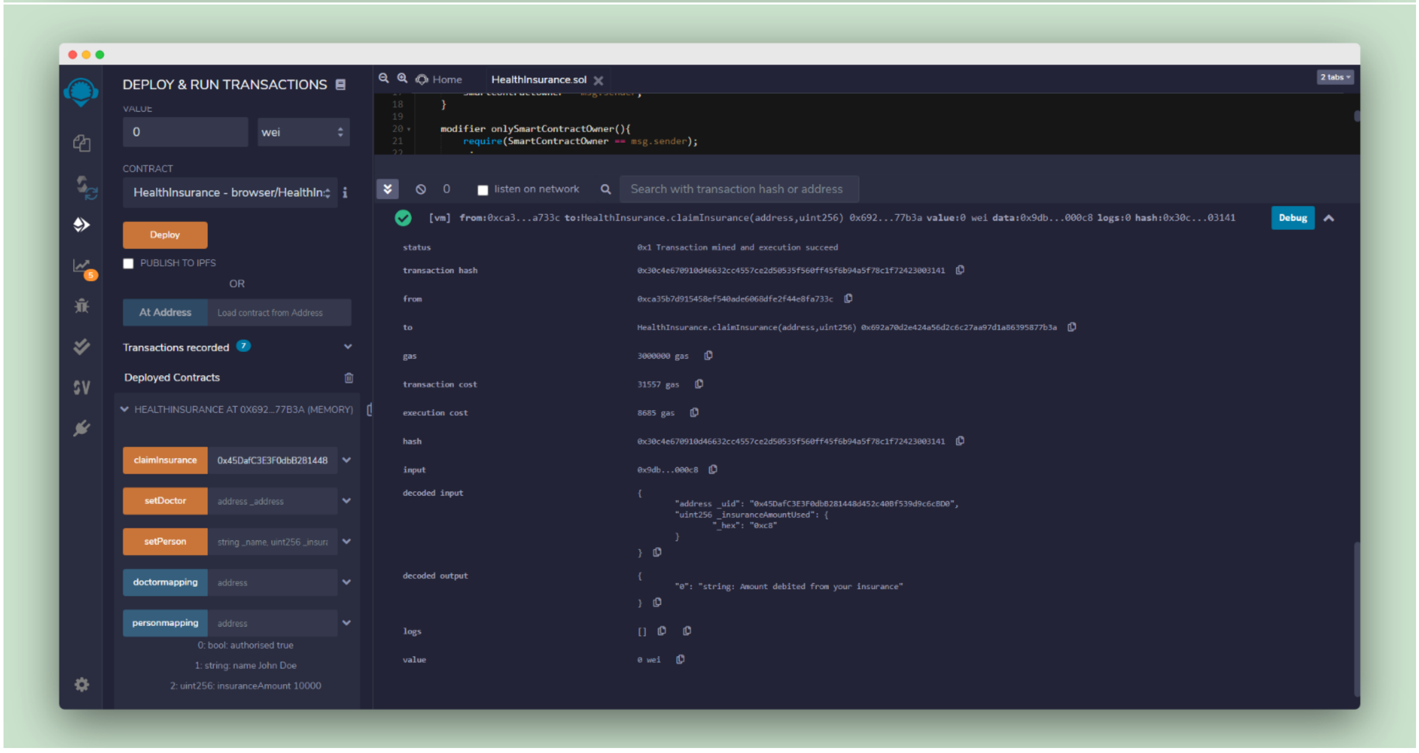
```
pragma solidity ^0.5.17;

contract HealthInsurance{

    address SmartContractOwner;

    struct person{
        bool authorised;
        string name;
        uint insuranceAmount;
    }

    mapping(address=>person) public personmapping;
    mapping(address=>bool) public doctormapping;

    constructor() public{
        SmartContractOwner = msg.sender;
    }

    modifier onlySmartContractOwner(){
        require(SmartContractOwner == msg.sender);
        _;
    }

    function  setDoctor(address _address) public onlySmartContractOwner{
        require(!doctormapping[_address]);
        doctormapping[_address] = true;
    }

    function setPerson(string memory _name,uint _insuranceAmount) public onlySmartContractOwner returns (address){
        address uid = address(bytes20(sha256(abi.encodePacked(msg.sender,block.timestamp))));
        require(!personmapping[uid].authorised);
        personmapping[uid].authorised = true;
        personmapping[uid].name = _name;
        personmapping[uid].insuranceAmount  = _insuranceAmount;

        return uid;
    }

    function claimInsurance(address _uid,uint _insuranceAmountUsed) public returns (string memory){
        require(!doctormapping[msg.sender]);
        if(personmapping[_uid].insuranceAmount < _insuranceAmountUsed){
            revert();
        }

        personmapping[_uid].insuranceAmount -= _insuranceAmountUsed;
```


DEPLOY & RUN TRANSACTIONS
VALUE
0
wei
CONTRACT
HealthInsurance - browser/HealthIns
Deploy
PUBLISH TO IPFS
OR
At Address
Load contract from Address
Transactions recorded 6
Deployed Contracts
HEALTHINSURANCE AT 0X692...77B3A (MEMORY)
Home
HealthInsurance.sol
listen on network
Search with transaction hash or address
Debug

```
claimInsurance    0x17959aF98E5f29dB5C37
setDoctor         0x14723A09ACff6D2A60Dc
setPerson         "John Doe",10000
doctormapping     address
personmapping     0x17959aF98E5f29dB5C37
    0: bool: authorised true
    1: string: name Sumit
    2: uint256: insuranceAmount 14500
```

```
    }

    modifier onlySmartContractOwner(){
        require(SmartContractOwner == msg.sender);
```

```
[vm] from:0xca3...a733c to:HealthInsurance.setPerson(string,uint256) 0x692...77b3a value:0 wei data:0x2fd...00000 logs:0 hash:0x581...726ca

status              0x1 Transaction mined and execution succeed
transaction hash    0x5810203c87d2ad2162c221e0deb0310bb8580f894fadb8990e0379ed42d726ca
from                0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to                  HealthInsurance.setPerson(string,uint256) 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
gas                 3000000 gas
transaction cost    88693 gas
execution cost      66141 gas
hash                0x5810203c87d2ad2162c221e0deb0310bb8580f894fadb8990e0379ed42d726ca
input               0x2fd...00000
decoded input       {
                        "string _name": "John Doe",
                        "uint256 _insuranceAmount": {
                            "_hex": "0x2710"
                        }
                    }
decoded output      {
                        "0": "address: 0x45DafC3E3F0dbB281448d452c40Bf539d9c6cBD0"
                    }
logs                []
value               0 wei
```

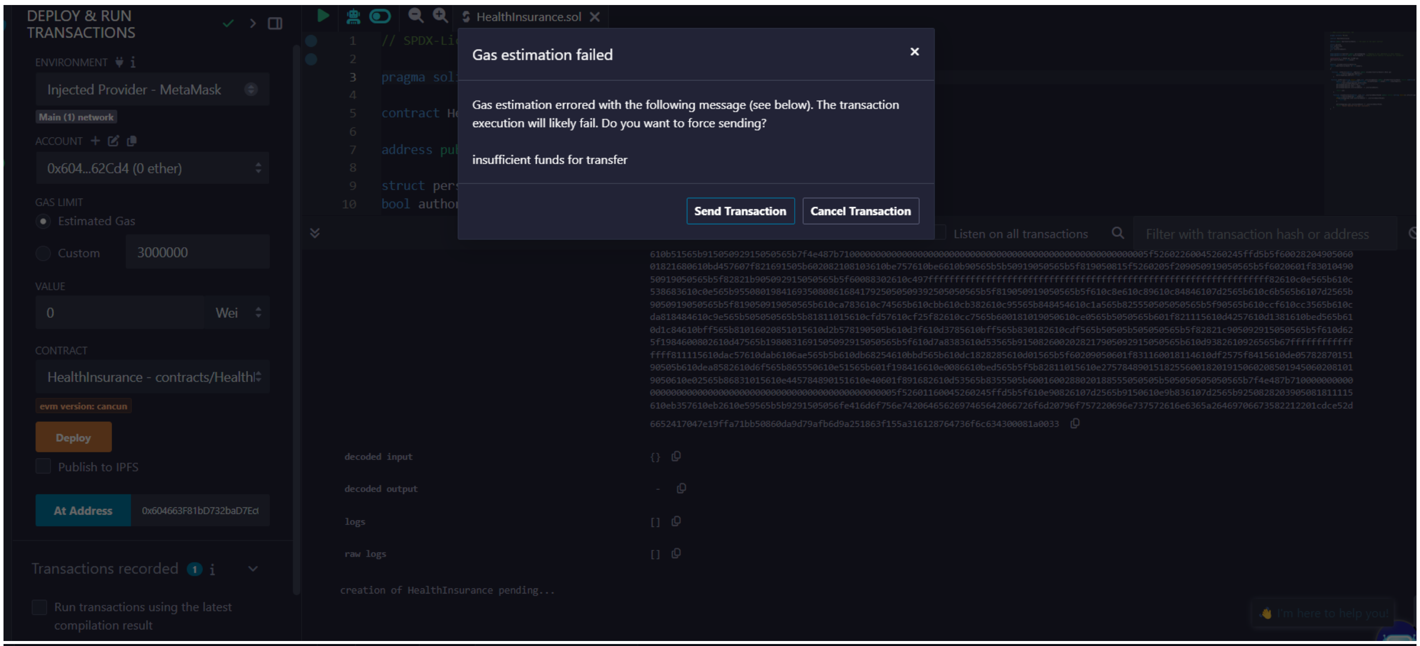

DEPLOY & RUN TRANSACTIONS
VALUE
0
wei
CONTRACT
HealthInsurance - browser/HealthIns
Deploy
PUBLISH TO IPFS
OR
At Address
Load contract from Address
Transactions recorded 7
Deployed Contracts
HEALTHINSURANCE AT 0X692...77B3A (MEMORY)
Home
HealthInsurance.sol
listen on network
Search with transaction hash or address
Debug

```
claimInsurance    0x45DafC3E3F0dbB281448
setDoctor         address _address
setPerson         string _name, uint256 _insur
doctormapping     address
personmapping     address
    0: bool: authorised true
    1: string: name John Doe
    2: uint256: insuranceAmount 10000
```

```
    }

    modifier onlySmartContractOwner(){
        require(SmartContractOwner == msg.sender);
```

```
[vm] from:0xca3...a733c to:HealthInsurance.claimInsurance(address,uint256) 0x692...77b3a value:0 wei data:0x9db...000c8 logs:0 hash:0x30c...03141

status              0x1 Transaction mined and execution succeed
transaction hash    0x30c4e670910d46632cc4557ce2d50535f560ff45f6b94a5f78c1f72423003141
from                0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to                  HealthInsurance.claimInsurance(address,uint256) 0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
gas                 3000000 gas
transaction cost    31557 gas
execution cost      8685 gas
hash                0x30c4e670910d46632cc4557ce2d50535f560ff45f6b94a5f78c1f72423003141
input               0x9db...000c8
decoded input       {
                        "address _uid": "0x45DafC3E3F0dbB281448d452c40Bf539d9c6cBD0",
                        "uint256 _insuranceAmountUsed": {
                            "_hex": "0xc8"
                        }
                    }
decoded output      {
                        "0": "string: Amount debited from your insurance"
                    }
logs                []
value               0 wei
```

Happens when insufficient funds exist :



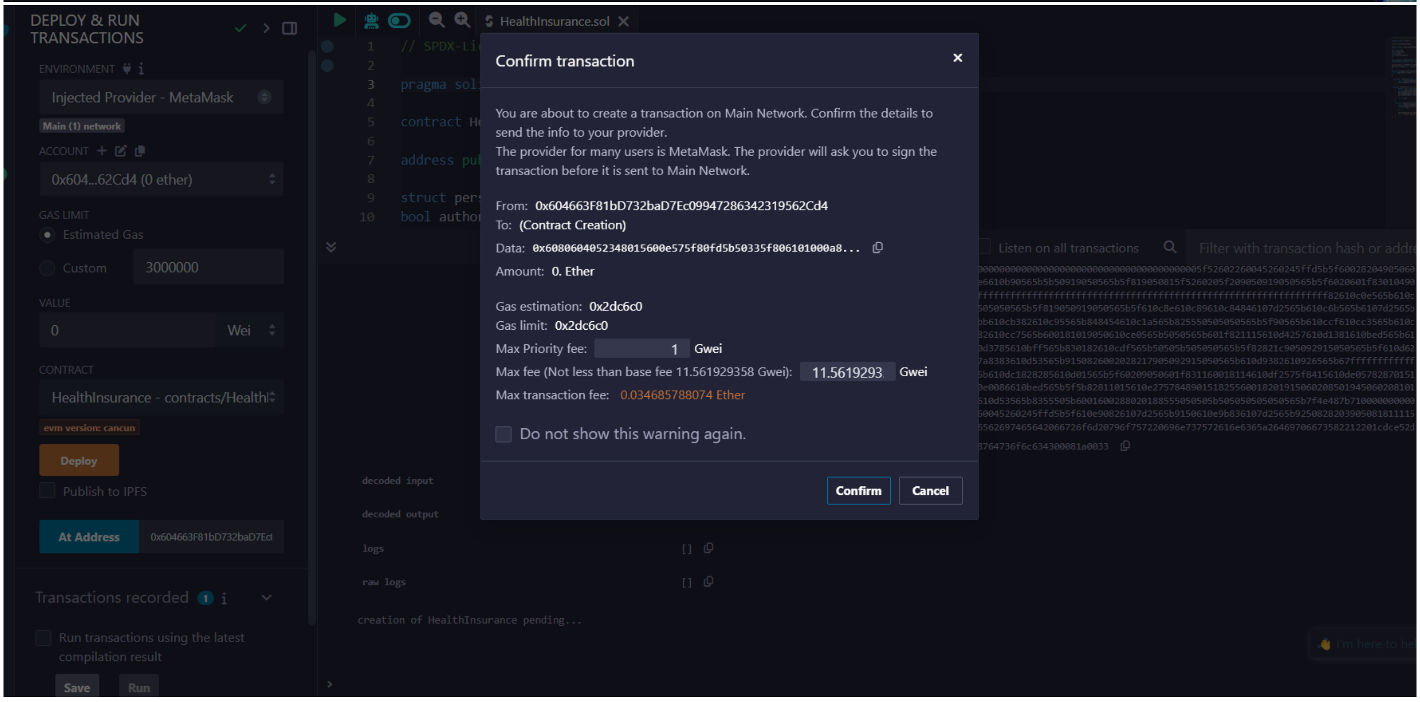
Gas estimation failed

Gas estimation errored with the following message (see below). The transaction execution will likely fail. Do you want to force sending?

insufficient funds for transfer

ENVIRONMENT: Injected Provider - MetaMask (Main (1) network)
ACCOUNT: 0x604...62Cd4 (0 ether)
GAS LIMIT: Estimated Gas; Custom 3000000
VALUE: 0 Wei
CONTRACT: HealthInsurance - contracts/HealthI
evm version: cancun

```
creation of HealthInsurance pending...
```



Confirm transaction

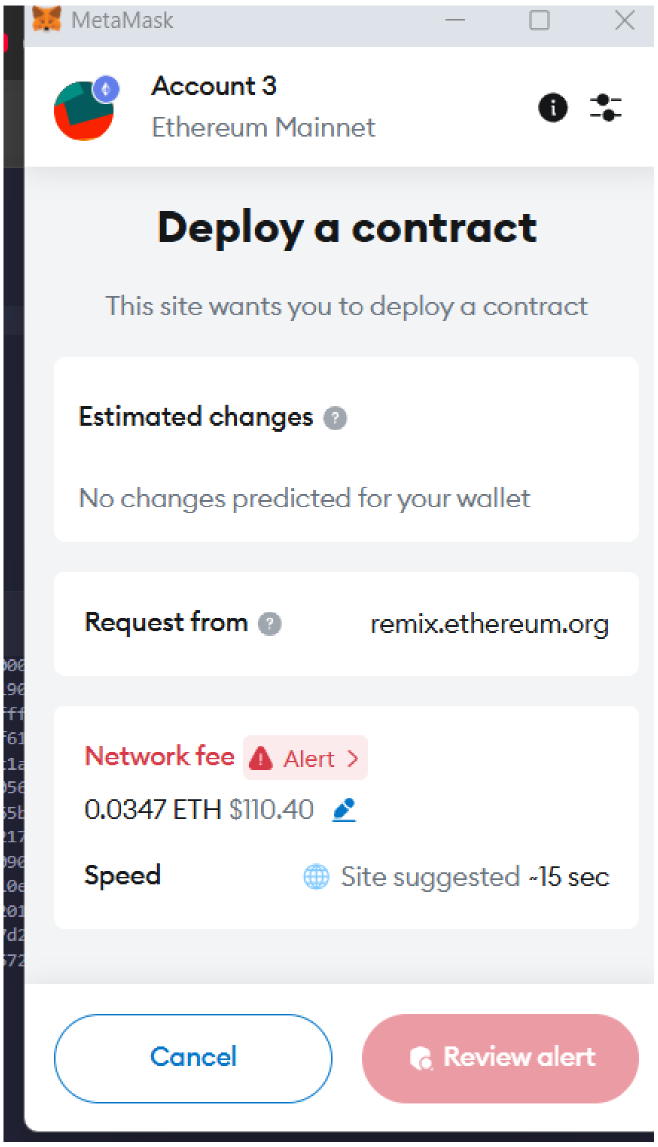
You are about to create a transaction on Main Network. Confirm the details to send the info to your provider.
The provider for many users is MetaMask. The provider will ask you to sign the transaction before it is sent to Main Network.

From: 0x604663F81bD732baD7Ec09947286342319562Cd4
To: (Contract Creation)
Data: 0x6080604052348015600e575f80fd5b50335f806101000a8...
Amount: 0. Ether

Gas estimation: 0x2dc6c0
Gas limit: 0x2dc6c0
Max Priority fee: 1 Gwei
Max fee (Not less than base fee 11.561929358 Gwei): 11.5619293 Gwei
Max transaction fee: 0.034685788074 Ether

Do not show this warning again.

```
creation of HealthInsurance pending...
```



MetaMask

Account 3
Ethereum Mainnet

Deploy a contract

This site wants you to deploy a contract

Estimated changes

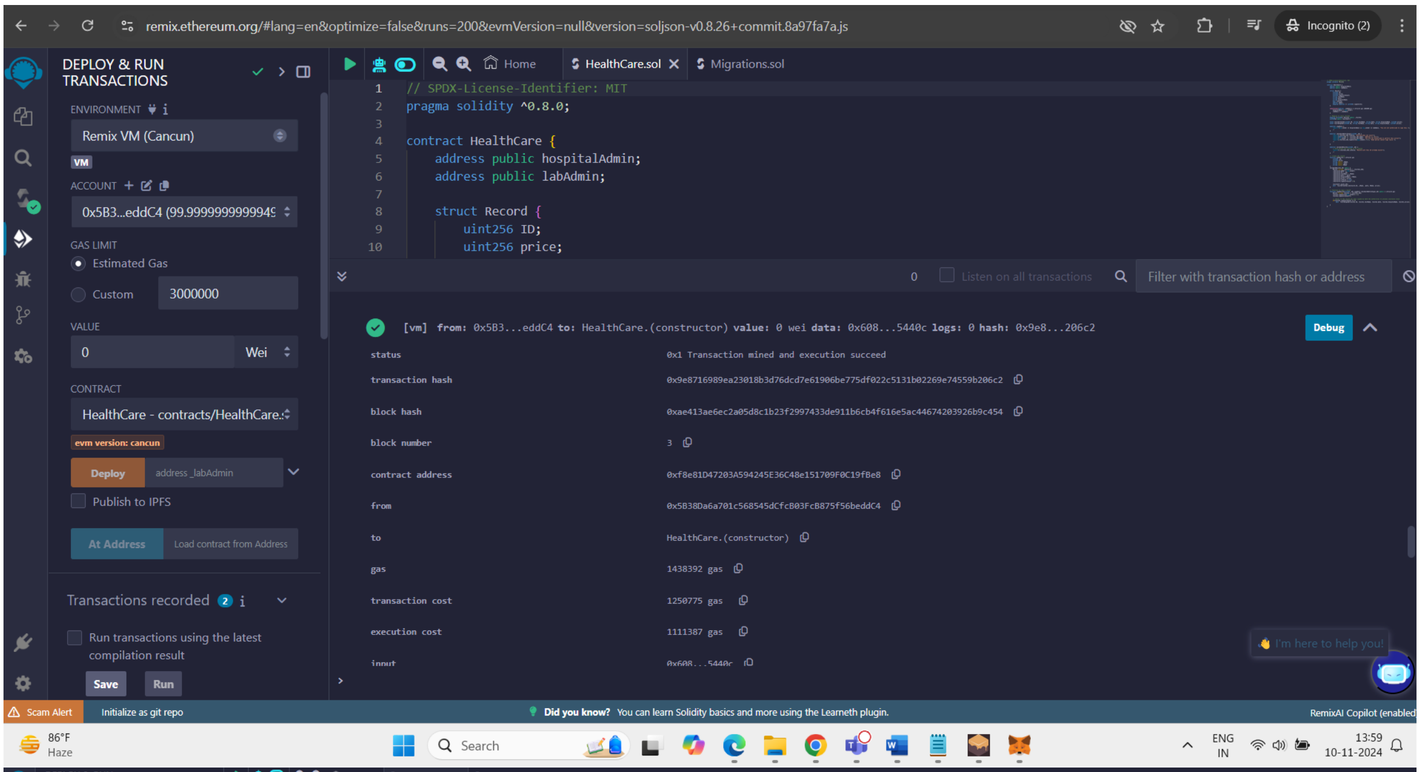
No changes predicted for your wallet

Request from remix.ethereum.org

Network fee Alert >
0.0347 ETH $110.40

Speed Site suggested ~15 sec

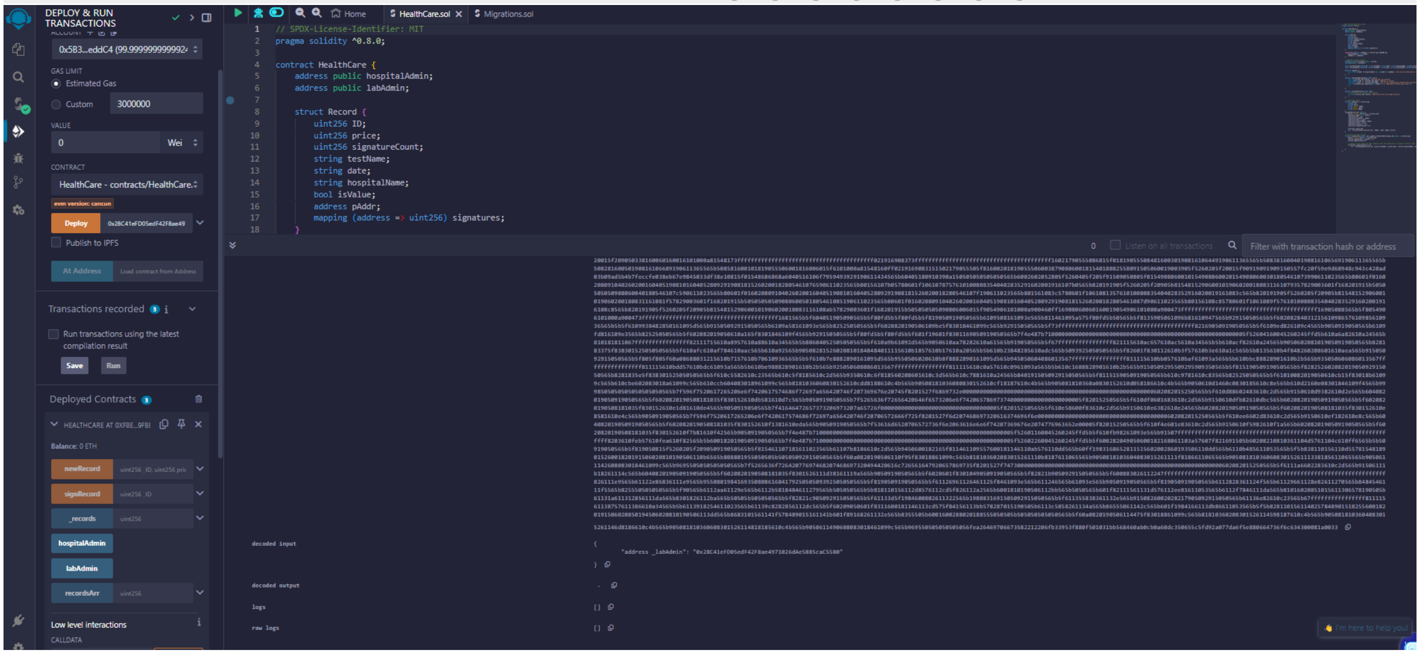
Health Insurance :



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

contract HealthCare {
    address public hospitalAdmin;
    address public labAdmin;

    struct Record {
        uint256 ID;
        uint256 price;
```

```
[vm] from: 0x5B3...eddC4 to: HealthCare.(constructor) value: 0 wei data: 0x608...5440c logs: 0 hash: 0x9e8...206c2
status                 0x1 Transaction mined and execution succeed
transaction hash       0x9e8716989ea23018b3d76dcd7e61906be775df022c5131b02269e74559b206c2
block hash             0xae413ae6ec2a05d8c1b23f2997433de911b6cb4f616e5ac44674203926b9c454
block number           3
contract address       0xf8e81D47203A594245E36C48e151709F0C19fBe8
from                   0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                     HealthCare.(constructor)
gas                    1438392 gas
transaction cost       1250775 gas
execution cost         1111387 gas
input                  0x608...5440c
```

remix.ethereum.org/#lang=en&optimize=false&runs=200&evmVersion=null&version=soljson-v0.8.26+commit.8a97fa7a.js
Incognito (2)
DEPLOY & RUN TRANSACTIONS
ENVIRONMENT
Remix VM (Cancun)
ACCOUNT
0x5B3...eddC4 (99.9999999999949
GAS LIMIT
Estimated Gas
Custom 3000000
VALUE
0 Wei
CONTRACT
HealthCare - contracts/HealthCare.sol
evm version: cancun
Deploy
address _labAdmin
Publish to IPFS
At Address
Load contract from Address
Transactions recorded 2
Run transactions using the latest compilation result
Save
Run
Home
HealthCare.sol
Migrations.sol
Listen on all transactions
Filter with transaction hash or address
Debug
I'm here to help you!
Scam Alert
Initialize as git repo
Did you know? You can learn Solidity basics and more using the Learneth plugin.
RemixAI Copilot (enabled
86°F Haze
Search
ENG IN
13:59
10-11-2024

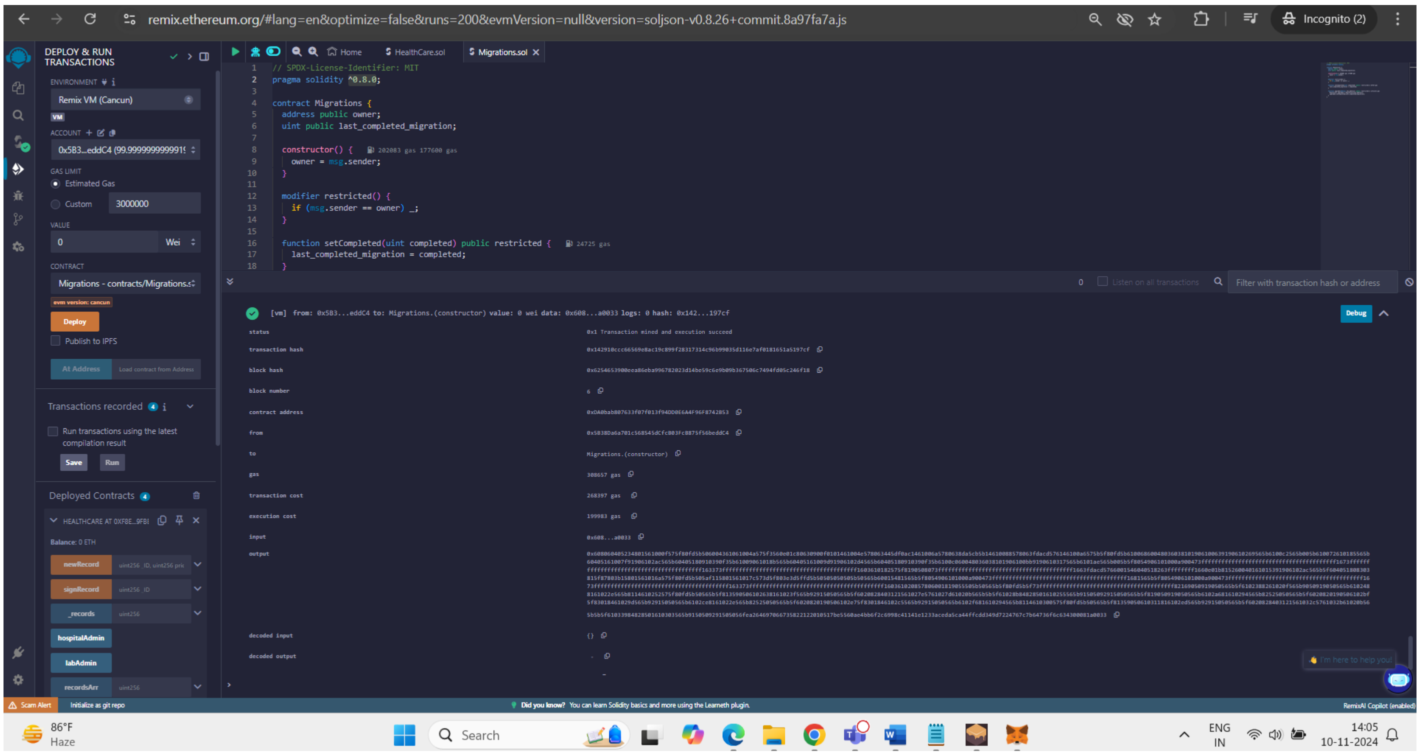


```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract HealthCare {
    address public hospitalAdmin;
    address public labAdmin;

    struct Record {
        uint256 ID;
        uint256 price;
        uint256 signatureCount;
        string testName;
        string date;
        string hospitalName;
        bool isValue;
        address pAddr;
        mapping (address => uint256) signatures;
    }
```

[illegible]

```
decoded input          {
                               "address _labAdmin": "0x28C41eFD05edF42F8ae4971026dAe5885caC55B0"
                       }
decoded output         -
logs                   []
raw logs               []
```

Migrations :



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

DEPLOY & RUN TRANSACTIONS

ENVIRONMENT: Remix VM (Cancun)

ACCOUNT: 0x5B3...eddC4 (99.9999999999999

GAS LIMIT: Estimated Gas / Custom 3000000

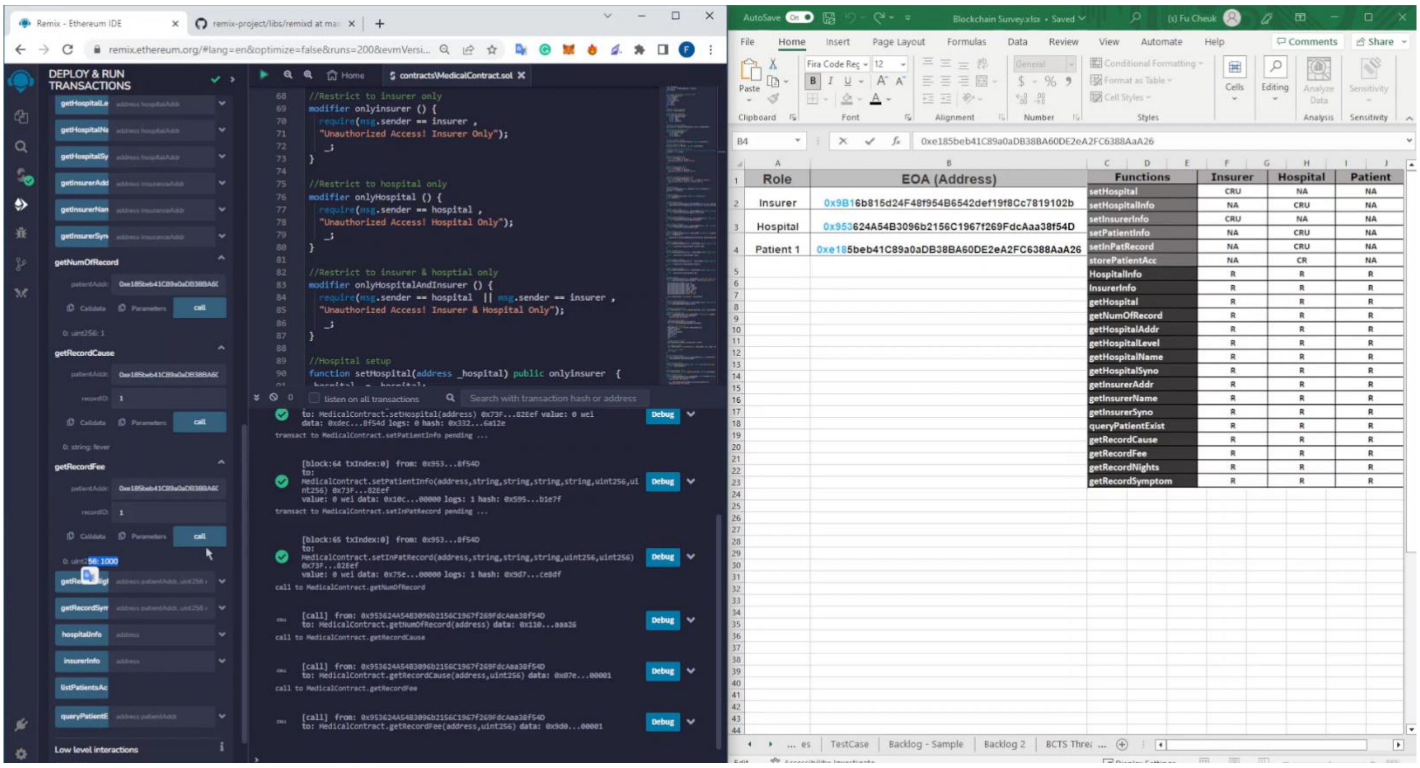
VALUE: 0 Wei

CONTRACT: Migrations - contracts/Migrations.sol

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

contract Migrations {
  address public owner;
  uint public last_completed_migration;

  constructor() {   202083 gas 177600 gas
    owner = msg.sender;
  }

  modifier restricted() {
    if (msg.sender == owner) _;
  }

  function setCompleted(uint completed) public restricted {   24725 gas
    last_completed_migration = completed;
  }
```

```
[vm] from: 0x5B3...eddC4 to: Migrations.(constructor) value: 0 wei data: 0x608...a0033 logs: 0 hash: 0x142...197cf
status              0x1 Transaction mined and execution succeed
transaction hash    0x142910ccc66569e8ac19c899f28317314c96b99035d116e7af0181651a5197cf
block hash          0x6254653900eea86eba996782023d14be59c6e9b09b367506c7494fd05c246f18
block number        6
contract address    0xDA0bab807633f07f013f94DD0E6A4F96F8742B53
from                0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                  Migrations.(constructor)
gas                 308657 gas
transaction cost    268397 gas
execution cost      199983 gas
input               0x608...a0033
output              [illegible]
decoded input       {}
decoded output      -
```

Deployment :



remix.ethereum.org/#lang=en&optimize=false&runs=200&evmVersi...

contracts\MedicalContract.sol

```
//Restrict to insurer only
modifier onlyinsurer () {
  require(msg.sender == insurer ,
  "Unauthorized Access! Insurer Only");
  _;
}

//Restrict to hospital only
modifier onlyHospital () {
  require(msg.sender == hospital ,
  "Unauthorized Access! Hospital Only");
  _;
}

//Restrict to insurer & hosptial only
modifier onlyHospitalAndInsurer () {
  require(msg.sender == hospital  || msg.sender == insurer ,
  "Unauthorized Access! Insurer & Hospital Only");
  _;
}

//Hospital setup
function setHospital(address _hospital) public onlyinsurer  {
```

```
to: MedicalContract.setHospital(address) 0x73F...82Eef value: 0 wei
data: 0xdec...8f54d logs: 0 hash: 0x332...6a12e
transact to MedicalContract.setPatientInfo pending ...

[block:64 txIndex:0] from: 0x953...8f54D
to:
MedicalContract.setPatientInfo(address,string,string,string,string,uint256,uint256) 0x73F...82Eef
value: 0 wei data: 0x10c...00000 logs: 1 hash: 0x595...b1e7f
transact to MedicalContract.setInPatRecord pending ...

[block:65 txIndex:0] from: 0x953...8f54D
to:
MedicalContract.setInPatRecord(address,string,string,string,uint256,uint256)
0x73F...82Eef
value: 0 wei data: 0x75e...00000 logs: 1 hash: 0x9d7...ce8df
call to MedicalContract.getNumOfRecord

[call] from: 0x953624A5483096b2156C1967f269FdcAaa38f54D
to: MedicalContract.getNumOfRecord(address) data: 0x110...aaa26
call to MedicalContract.getRecordCause

[call] from: 0x953624A5483096b2156C1967f269FdcAaa38f54D
to: MedicalContract.getRecordCause(address,uint256) data: 0x07e...00001
call to MedicalContract.getRecordFee

[call] from: 0x953624A5483096b2156C1967f269FdcAaa38f54D
to: MedicalContract.getRecordFee(address,uint256) data: 0x9d0...00001
```

Blockchain Survey.xlsx

B4: 0xe185beb41C89a0aDB38BA60DE2eA2FC6388AaA26

Role	EOA (Address)
Insurer	0x9B16b815d24F48f954B6542def19f8Cc7819102b
Hospital	0x953624A54B3096b2156C1967f269FdcAaa38f54D
Patient 1	0xe185beb41C89a0aDB38BA60DE2eA2FC6388AaA26

Functions	Insurer	Hospital	Patient
setHospital	CRU	NA	NA
setHospitalInfo	NA	CRU	NA
setInsurerInfo	CRU	NA	NA
setPatientInfo	NA	CRU	NA
setInPatRecord	NA	CRU	NA
storePatientAcc	NA	CR	NA
HospitalInfo	R	R	R
InsurerInfo	R	R	R
getHospital	R	R	R
getNumOfRecord	R	R	R
getHospitalAddr	R	R	R
getHospitalLevel	R	R	R
getHospitalName	R	R	R
getHospitalSyno	R	R	R
getInsurerAddr	R	R	R
getInsurerName	R	R	R
getInsurerSyno	R	R	R
queryPatientExist	R	R	R
getRecordCause	R	R	R
getRecordFee	R	R	R
getRecordNights	R	R	R
getRecordSymptom	R	R	R

5. PROJECT REPOSITORY AND DEMO LINKS

5.1 Project Repository Link :

GitHub Repository: https://github.com/toranvathani/ProjectonAutomatedInsuranceClaims

5.2 Video Demo Link :

Video Demo: https://www.loom.com/share/ce9a01d273654879baadb1e5b8c48db0

6. BIBLIOGRAPHY

1. "Mastering Blockchain: Unlocking the Power of Cryptocurrencies, Smart Contracts, and Decentralized Applications" by Imran Bashir (2017).
   - This book provides an in-depth exploration of blockchain technology, including its applications in various domains such as finance and healthcare, and how smart contracts can be used for automating processes like insurance claims.

2. "Blockchain Basics: A Non-Technical Introduction in 25 Steps" by Daniel Drescher (2017).
   - A comprehensive guide to understanding the basics of blockchain, which is crucial for understanding how smart contracts work in decentralized systems, particularly in sectors like health insurance.

3. Ethereum Documentation (2024).
   - Ethereum is the most popular platform for developing decentralized applications and smart contracts. Its official documentation provides detailed guidance on how to write, deploy, and manage smart contracts on the Ethereum blockchain.
   - Available at: https://ethereum.org

4. "Solidity Documentation" (2024).
   - Solidity is the most widely used programming language for writing smart contracts on Ethereum. The official documentation covers the syntax, best practices, and examples for developing secure and efficient smart contracts.
   - Available at: https://soliditylang.org

5. "Blockchain for Healthcare: An Overview of the Benefits and Challenges" by Isabel P. Tewes and Thomas D. Lutz (2019).
   - This article discusses the potential applications of blockchain in healthcare, focusing on the use of smart contracts to automate tasks such as claims processing, data management, and privacy protection.

6. "Smart Contracts and Their Applications in the Health Insurance Industry" by Zhang Wei and Rui Zhu (2020).
   - This paper explores how smart contracts can be used to improve efficiency, transparency, and fraud prevention in the health insurance industry. It includes case studies and examples of successful implementations.

7. "The Basics of Bitcoins and Blockchains" by Antony Lewis (2018).
   - This book explains blockchain and cryptocurrency technologies in a simple and approachable way, with a specific section on the potential uses of smart contracts in industries such as healthcare and insurance.

8. "Web3.js Documentation" (2024).
   - Web3.js is a library for interacting with Ethereum and other blockchains from JavaScript applications. The official documentation provides guidance on how to integrate smart contracts with web-based interfaces for decentralized applications.
   - Available at: https://web3js.readthedocs.io

9. "Introduction to Blockchain and Blockchain for Health: Use Cases" by J. Samuel and R. Raj (2021).
   - This research paper covers the integration of blockchain technology in health systems, with a

particular focus on how it can enhance transparency, security, and automation in health insurance.

10. Truffle Suite Documentation (2024).
   - Truffle Suite is one of the leading frameworks for Ethereum development, providing tools for smart contract development, testing, and deployment. The documentation provides comprehensive tutorials and guides.
   - Available at: https://www.trufflesuite.com/docs

11. "Blockchain in Healthcare Today" (2021).
   - This journal focuses on the applications of blockchain technology in healthcare, including the use of smart contracts for automating insurance claims processing, patient record management, and more.