# Table of Contents

[Module - 1](#)

[Assessor’s Feedback (may be delivered in line with the submission) 2](#)

[Blockchain-Based Healthcare Record System 8](#_sje1wiruu4av)

[Final Report – LDS7006M 8](#)

[HealthCare Innovations Ltd. Prototype 8](#_sqm3qt16d39q)

[May 1, 2025 8](#_gdkqownyltfk)

[1.0 Introduction to Blockchain 9](#_m9osy86sh9ev)

[1.1 Project Objective 11](#_wkqsde7hx13f)

[2.0 Blockchain Class Structure and Components 12](#_wbedn81ijl7)

[2.1 Block Structure Demonstration Using Healthcare Data 14](#_5r9fdod26512)

[2.2 Genesis Block Creation 15](#_e9j87zexur0y)

[2.3 Adding Blocks to the Chain 16](#_9uj3xrm2m63c)

[2.4 Consensus Mechanism 19](#_59ru5pweudtc)

[2.4.1 valid\_proof(): Difficulty-Based Hash Validation 19](#_lnvdqh6tqpo0)

[2.4.2 Proof\_of\_Work(): Nonce Discovery via Iteration 20](#_p9xr89xoj4na)

[2.5 Valid\_chain(): Full Blockchain Integrity Checking 20](#_sjqfs2618ojp)

[2.6 Resolve\_Conflicts(): Distributed Consensus via Longest Chain Rule 21](#_rjasnygr41e6)

[2.7 Application of Consensus in Project Context 22](#_e3w1024qxsfe)

[2.8 Tampering Test and Validation Failure 24](#_98coc0xwhqnl)

[2.9 Tampering Simulation and Chain Invalidation 25](#_l323conddid)

[2.10 Blockchain Functional Testing and Hash Outputs 26](#_fq2iwivc84y9)

[3.0 Blockchain Activity Visualisation 27](#_w8oz3prau2f5)

[3.1 Block Hash Creation Timeline 28](#_2fakx4bvnvvv)

[3.2 Distribution of Blockchain Actions 28](#_xcpm6p8sk7gx)

[3.3 Function Call Frequency Analysis 29](#_woyc6qr6eobk)

[4.0 Design Choices for Blockchain Implementation (Part A) 30](#_ahvkk955ugpg)

[4.1 Class-Based Modular Architecture 30](#_r3p45ql14hw8)

[4.2 Genesis Block Implementation 31](#_mzi7i1wl1s8j)

[4.3 SHA-256 for Cryptographic Hashing 32](#_5xoqzv7c2mg7)

[4.4 Proof-of-Work (PoW) for Data Entry Validation 33](#_f4x5so13q9bk)

[4.5 Transaction Record Format 33](#_d1n9p2eq9673)

[4.6 Tamper Detection and Chain Validation 33](#_jn8m2jpia6qg)

[4.7 Simulated CLI for Practical Demonstration 34](#_1c5iswkammcx)

[4.8 Function Call Auditing 34](#_6fs3bsl6j0n)

[4.9 Visual Analytics for Transparency 34](#_abzc4j74sxsm)

[5.0 SMART CONTRACT (SOLIDITY - PART B) 35](#_6roiape1gjej)

[5.1 Introduction to Smart Contracts (Solidity) 35](#_45li127tq15u)

[5.1.1 AUTOMATION 35](#_f1ose8ltt6wp)

[5.1.2 Transparency and Trust 35](#_1ayrislns4ye)

[5.1.3 SECURITY 36](#_yma894to4nz3)

[5.1.4 COST REDUCTION 36](#_86whcqhymi06)

[5.1.5 DECENTRALISATION 36](#_8t6d0g44q8l8)

[5.1.6 USES OF SMART CONTRACT 36](#_pzjgu78ckds4)

[Uses of Smart Contracts in Healthcare 36](#_ejlb00rhov50)

[5.2 Smart Contract Structure and Deployment (Solidity) 37](#_nfwx0tpj85p5)

[5.2.1 Core Design Architecture 37](#_a8zbxu5xaknm)

[5.2.2 Secure Mappings and State Tracking 37](#_zg6rwe1itox9)

[5.2.3 Event Emission for Transparency 38](#_sm23rff21bm4)

[5.2.4 Functionality Overview 38](#_3xo8h7q1xw6d)

[5.2.4.1 addRecord() – Registering New Patient Data 38](#_4o793zu8id59)

[5.2.4.2 transferRecord() – Ownership Delegation 39](#_eujsjngmgftt)

[5.3 Smart Contract Functionality and Testing 39](#_kckdz5hbuxkz)

[● Sepolia Faucet 39](#_g67es5acgvj1)

[5.3.2 MetaMask Wallet Setup and Sepolia Testnet Funding 40](#_tg0ch1p5nwo5)

[5.3.2.1 Creating MetaMask Accounts 40](#_vv3kjcwsdq35)

[5.3.2.2 Funding Accounts with Sepolia ETH 40](#_keuo6nph00ip)

[Verification via Etherscan 42](#_goabp4lptxft)

[5.3 Setting Up the Remix IDE 42](#_7rc12ti7ust4)

[5.4 Connecting MetaMask to Remix 44](#_gkou5gq990t7)

[5.4.1 Smart Contract Compilation and Deployment 45](#_49mkica4370)

[5.5 addRecord() Implementation and Etherscan Confirmation 47](#_xe4pvc12x4m8)

[5.5 transferRecord() and Ownership Transfer Verification 49](#_6tc4t5rxvng3)

[5.6 Function: totalRecords() 50](#_losz27p5qm4o)

[5.6.1 Function: transferRecord() 51](#_ak5k4d5jndky)

[5.7 Challenges and Resolutions 53](#_drs6j7smrz0y)

[5.7.1. Remix IDE Deployment Errors 53](#_avnaaknrj25r)

[5.7.2. Incorrect Function Argument Formatting 53](#_as3k2hqa3jnk)

[5.7.3. Ownership Verification Issues 54](#_l29trz3lccyt)

[5.7.4. Proof-of-Work Bottlenecks in Python Simulation 54](#_42yowg37extr)

[5.7.5. Incomplete Documentation Tracking 54](#_gbhgelofsgik)

[6.0 Limitations and Future Enhancements 55](#_430jsgtbc2ok)

[6.1 Limited Access Control 55](#_w0q2dh36njee)

[6.2 No Data Encryption or Off-Chain Storage 55](#_16pxnti35ymc)

[6.3 Gas Cost and Scalability Constraints 55](#_dxncpgi6iym5)

[6.4 Basic Blockchain Simulation (Python) 55](#_x4g316el028u)

[Appendix A: Fund Transfer to Account 1 57](#_gy4q0yiduhhn)

[Appendix B: Smart Contract Metadata and ABI 57](#_zcyqdqa94cll)

[References 59](#_va03zh5fnyb0)

## 

Figure 1: System Flow Diagram Showing Dual Implementation of Python Blockchain and Ethereum Smart Contract for Healthcare Record Management.10

Figure 2: Python implementation of the Block class for Blockchain Units. This code defines the Block class, which contains essential features such as index, timestamp, transaction data, previous block hash, nonce, and SHA-256 hash. The compute\_hash() technique assures tamper-proof integrity by creating a unique, deterministic fingerprint for each block.12

Figure 3: Simulated Patient Data Stored in Blockchain Blocks. Three manually created blocks show how healthcare events can be securely hashed and linked using the Block class. Each block contains a patient interaction and produces a unique SHA-256 hash, confirming immutability. [1](#_wbedn81ijl7)3

Figure 4: This output displays the unique SHA-256 hashes for three different block instances, each representing a patient-related event. The distinct hashes confirm that even minimal changes in transaction data result in entirely different outputs, demonstrating blockchain immutability and tamper detection. [1](#_e9j87zexur0y)7

Figure 5: Genesis Block and Block Addition Implementation in the Blockchain Class. The image illustrates the create\_genesis\_block() method used to initialise the blockchain and the add\_block() method for sequentially linking new blocks. The genesis block serves as the immutable starting point, anchoring the entire chain for data integrity and audit compliance.[1](#_9uj3xrm2m63c)5

Figure 6:simulated blockchain integration of medical transactions. The add\_block() method is used to safely append three anonymized healthcare transactions to the chain. Timestamped and salted to guarantee traceability and immutability, each block contains structured medical data.20

Figure 7:.Blockchain Output Displaying Healthcare Transactions in Chronology. The genesis block and three patient-related events are among the four blockchain blocks whose printed output is seen in the figure. A timestamp, medical action, descriptive information, and a SHA-256 hash verifying data integrity and sequential block linking are all included in each entry………………………………….18

Figure 8: valid\_proof() Function to Enforce Hash Difficulty Rules. This function checks whether a proposed hash starts with the required number of leading zeroes and matches the block’s computed hash. It’s used within the proof-of-work loop to enforce computational security……………………

Figure 9: This method is called from within add\_block() to ensure new blocks undergo PoW before being appended to the chain.21

Figure 10: Valid\_chain() re-evaluates all block hashes and their linkages, returning False if any block has been tampered with or modified, thus preserving integrity. [2](#_e3w1024qxsfe)0

Figure 11: Resolve\_conflicts() compares local and external chains, adopting the longest valid one to ensure consensus and consistency in distributed environments. [2](#_98coc0xwhqnl)1

Figure 12: Python Implementation for Appending Medical Records to a Blockchain with Proof-of-Work. [2](#_l323conddid)2

Figure 13: Output of Blockchain with Proof-of-Work Mining. Each block includes a timestamp, nonce, and SHA-256 hash, demonstrating successful mining of medical transactions under defined difficulty constraints. [2](#_fq2iwivc84y9)3

Figure 14: Code to Validate Blockchain Integrity Using valid\_chain(). This code adds two secure healthcare records and verifies the blockchain’s validity using the valid\_chain() method, ensuring no tampering has occurred post-creation. [2](#_w8oz3prau2f5)4

Figure 15: Identification of Tampering valid\_chain() is used. This image illustrates how the blockchain fails validation when a medical record is manually altered. The contradiction is appropriately detected by the system, which produces "Chain valid? No." [2](#_2fakx4bvnvvv)5

Figure 16: Assertion-Based Tampering Validation Test. This code formally confirms blockchain integrity by asserting that a tampered chain returns False from valid\_chain(), ensuring reliable detection of unauthorised modifications…………………………………………………………….. 25

Figure 17: CLI Simulation for Adding a Patient Vaccination Record. A new healthcare transaction is securely added to the blockchain via simulated backend logic, replicating administrative interactions […………………………………………………………………………………………………](#_woyc6qr6eobk)…….. 26

Figure 18: Timeline of Blockchain Block Creation. A line chart showing the time-based creation of blocks across the simulation, with each point indicating a mined block's position and timestamp…..29

………………………………………………………………………………………………..

Figure 19: Distribution of Recorded Blockchain Actions. A pie chart representing the six different patient-related activities logged on the blockchain, all equally distributed in this simulation………. 30

Figure 20: Frequency of Blockchain Function Calls. A bar chart quantifying how many times core blockchain functions were triggered, offering insights into runtime behaviour and logic efficiency

Figure 21: This diagram illustrates the integration of blockchain technology within healthcare, showcasing interactions among patients, doctors, hospitals, insurance companies, and researchers. It emphasises how blockchain facilitates secure and transparent data exchange across various stakeholders…………………………………………………………………………………………. 30

Figure 22: This flowchart delineates the PoW consensus algorithm, detailing the steps from transaction initiation to block validation and addition to the blockchain. It underscores the computational effort required to achieve consensus, ensuring the integrity and security of the blockchain. [3](#_f4x5so13q9bk)3

Figure 23: This schematic presents the structure of a blockchain starting with the genesis block. It highlights how each subsequent block references the hash of its predecessor, establishing an immutable chain. The diagram also introduces the concept of a "floating" genesis block, adaptable to various blockchain applications. [3](#_d1n9p2eq9673)1

Figure 24: This diagram provides a visual representation of the blockchain's data structure, emphasising the linkage between blocks through cryptographic hashes. It demonstrates how altering a single block would disrupt the entire chain, thus ensuring data integrity. [3](#_jn8m2jpia6qg)4

Figure 25: This illustration showcases the flow of patient data within a blockchain-based system. It outlines how patients, healthcare providers, and pharmacies interact through the blockchain to manage and access personal health records securely..33

Figure 26: This structure ensures each record has a unique identifier and tracks provenance by linking it to the Ethereum wallet that created it. This design supports the requirements of GDPR Article 5 — accountability and traceability of personal data — while maintaining pseudonymity through public addresses (Kuo et al., 2017). [3](#_6fs3bsl6j0n)6

Figure 27: This enables constant-time retrieval of records by their unique recordId, which functions as a pseudonymised patient reference. The public visibility of this mapping supports decentralised verification without exposing sensitive contents.. [3](#_abzc4j74sxsm)7

Figure 28: These events are critical for creating an immutable audit trail of all actions involving patient records. Off-chain monitoring tools or frontend UIs can subscribe to these events to detect and visualise changes in near real time — a feature popularised in projects like MedRec (Azaria et al., 2016). [3](#_6roiape1gjej)7

Figure 29: The function uses input validation (require) to ensure the record is new and non-empty. By storing the current block timestamp, it achieves a verifiable and immutable creation time.. [3](#_45li127tq15u)7

Figure 30: This approach applies strict access control using msg.sender, ensuring that only the current owner may delegate access. This is a critical mechanism for enforcing data sovereignty — a principle echoed in blockchain-based health data systems like MedShare and Guardtime (Xia et al., 2017)…………………………………………………………………………………………………..38

Figure 31: Screenshot of Account 1 receiving Sepolia ETH

................................................................................................................................................................40

Figure 32: Screenshot of Account 2 receiving Sepolia ETH…………………………………………41

Figure 33: Screenshot showing a new file called PatientRecordContract.sol in the workspace..............................................................................................................................................42

Figure 34:Screenshot showing contract setup in Remix with file loaded and Solidity version selected.………………………………………………………………………………………………. 43

Figure 35: Screenshot showing Remix connected to Account 1 on Sepolia.

……………………………………………………………………………..........................................44

………………………………………………………………………………………………………….

Figure 36: MetaMask deployment confirmation popup.……………………………………………. 45

Figure 37: Remix contract deployment confirmation………………………………………………..45

Figure 38:Contract address shown on Remix and Etherscan.……………………………………….. 46

Figure 39: Successful MetaMask transaction approval………………………………………………47

Figure 40: Screenshot showing Remix connected to Account 1 on Sepolia………………………….47

Figure 41: MetaMask confirmation for ownership transfer request…………………………………48

Figure 42: Console output showing Account 1 as owner…………………………………………….49

Figure 43: Successful MetaMask transaction approval……………………………………………….49

Figure 44: Screenshot of totalRecords() returning 1…………………………………………………50

Figure 45 MetaMask transfer transaction……………………………………………………………..51

Figure 46:EtherScan:<https://sepolia.etherscan.io/tx/0x86ebea2440193662d95088cdb5844a3cc599008aa50f2f5eadd4b2d445230747>…………………………………………………………………………………51

Figure 47: Etherscan verification of transfer parameters……………………………………………. 52

## 

## 

## **Blockchain-Based Healthcare Record System**

## HealthCare Innovations Ltd. Prototype

## May 1, 2025

## 

## 

## 

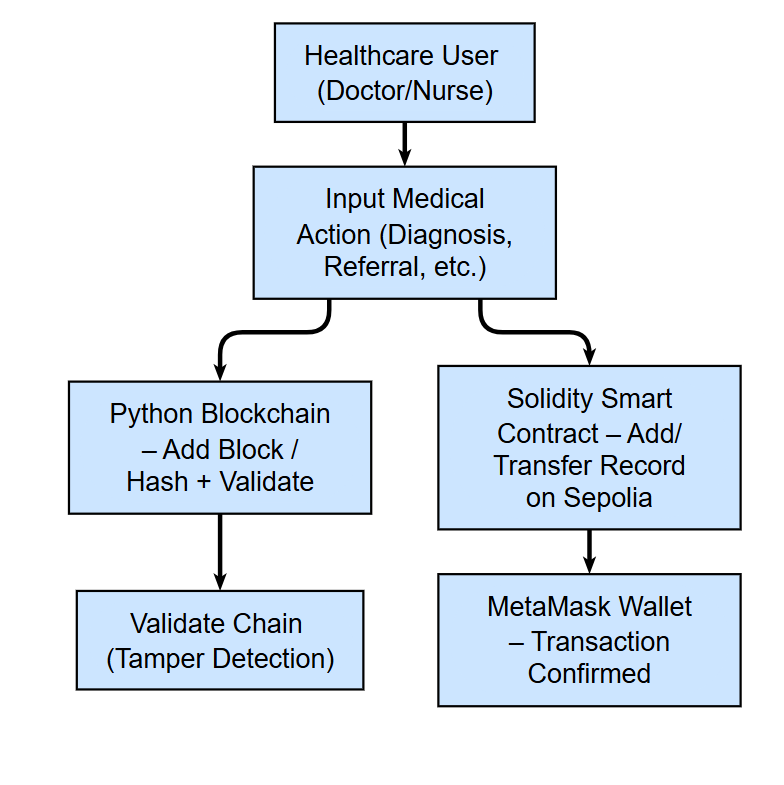
## **1.0 Introduction to Blockchain**

Blockchain is a decentralised and tamper-resistant digital ledger invented by Satoshi Nakamoto in 2008 with the introduction of Bitcoin. At its foundation, blockchain is a continuously increasing chain of records (blocks) that are securely linked using cryptographic hashes. This linking approach assures that any alteration to one block requires recalculating the hashes of all following blocks, rendering tampering computationally costly (Narayanan et al., 2016).

A blockchain's structure is inherently distributed, with each node in the network maintaining a copy of the ledger that is synchronised using consensus techniques like Proof of Work (PoW) or Proof of Stake (PoS). This distributed nature eliminates the reliance on a central authority, which is traditionally a single point of failure in systems like databases or file servers (Zheng et al., 2017).

Blockchain has gained popularity in areas other than bitcoin, such as supply chain, voting, and healthcare. Healthcare, in particular, confronts issues with data security, patient privacy, and inter-organizational trust. Traditional electronic health record (EHR) systems frequently rely on centralised databases, leaving them subject to breaches, unauthorised access, and loss of patient control over their data (Esmaeilzadeh, 2019). Blockchain, with its openness, immutability, and decentralisation, provides a compelling alternative for improving trust and accountability in medical data systems (Kuo, Kim, & Ohno-Machado, 2017).

Smart contracts, which are executable programs stored on the blockchain, augment these capabilities by automating access rights, audit trails, and record changes. When properly implemented, they can enforce clinical data policies while also improving compliance with privacy standards like GDPR or HIPAA (Azaria et al., 2016). As blockchain technology advances, its applications in healthcare are being investigated for patient-centric data control, safe interoperability, and verified medical provenance.



**Figure 1**: System Flow Diagram Showing Dual Implementation of Python Blockchain and Ethereum Smart Contract for Healthcare Record Management.

## **1.1 Project Objective**

This project aims to design and simulate a decentralised healthcare record system using blockchain principles. The assessment is structured around two parallel implementations:

1. A Python-based private blockchain simulation that demonstrates core concepts such as block creation, PoW consensus, tamper detection, and chain validation.
2. A Solidity-based smart contract, PatientRecordContract, deployed on the Ethereum Sepolia testnet to securely manage and transfer patient records between authorised entities.

The Python implementation supports step-by-step investigation of blockchain mechanics. Blocks are linked by cryptographic hashes and added to the chain only after solving a computational puzzle, much as PoW. The simulation includes validation tests to identify tampering, as well as conflict resolution logic (resolve\_conflicts) to illustrate decentralised integrity checks.

In addition, the Solidity smart contract serves as a safe and transparent record keeper. Functions like addRecord() and transferRecord() store and reassign patient information based on ownership constraints. These functions are executed and checked with Remix IDE and MetaMask, allowing for genuine blockchain transactions on Sepolia's test network. Testing involves interacting between two Ethereum accounts to ensure ownership enforcement and record immutability.

Together, these components represent a minimally practical prototype for secure medical recordkeeping. The system meets the basic requirements for security, traceability, and user-level authorization. Though simplified for demonstration reasons, this dual-implementation concept demonstrates blockchain's potential to revolutionize healthcare data governance and privacy control.

## **2.0 Blockchain Class Structure and Components**

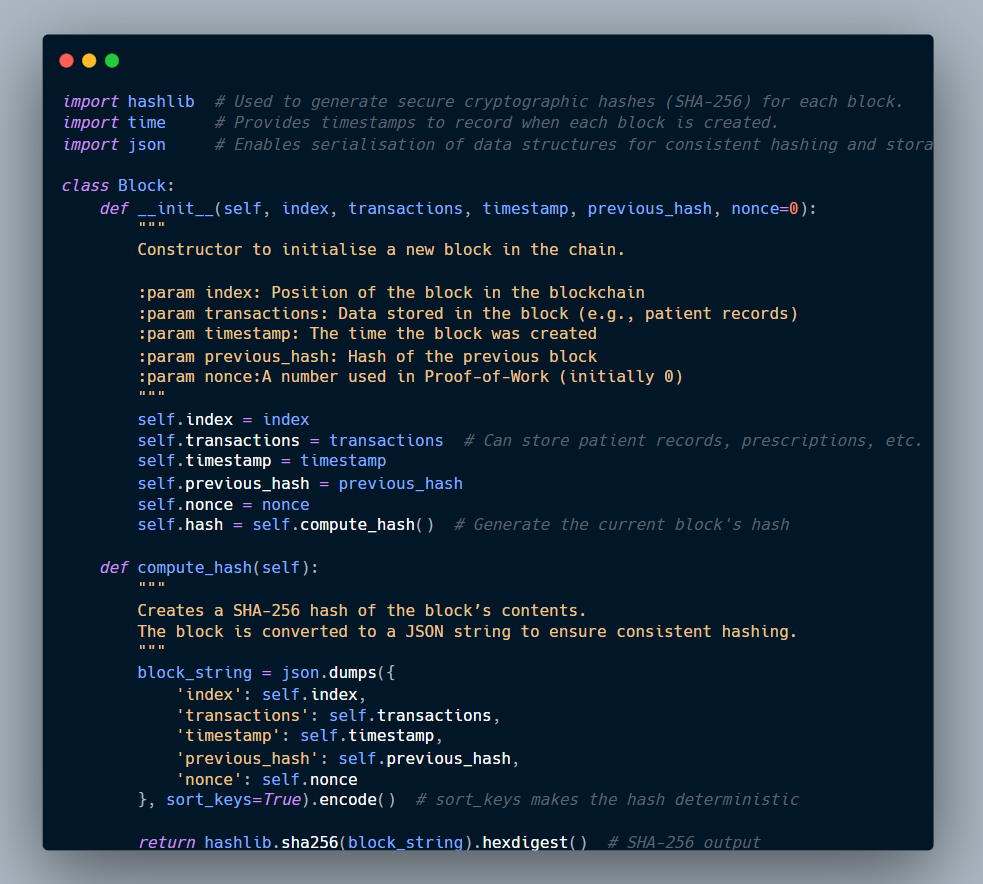
The implementation starts by importing three necessary Python modules: json for serialising data structures, time for creating timestamping blocks, and hashlib for creating cryptographically safe hashes. With the use of these libraries, the blockchain can mimic the functionality of decentralised ledger technologies while still being lightweight and language-neutral for educational prototyping. The Block class is the central component of the system, defining the behavior and structure of every chain unit. A list of transactions (which might be patient records), an index, a timestamp, the previous\_hash connecting it to the previous block, and a nonce value for Proof-of-Work operations are all necessary for each block to be instantiated.

The constructor (\_\_init\_\_) method calls self.compute\_hash() to compute the block's hash right away after initialising these characteristics. This guarantees a deterministic cryptographic fingerprint that is specific to each block depending on its contents, which is a basic prerequisite for tamper-proofing (Reyna et al., 2018). The transactions field is especially flexible, enabling the block to contain structured medical events like prescriptions, referrals, and diagnosis. Storing the hash of the preceding block creates a one-way cryptographic relationship throughout the chain, further enhancing the integrity of each block.

The block's unique SHA-256 hash is generated by the compute\_hash() function. Prior to hashing and encoding, it transforms the block's contents into a JSON string while guaranteeing a constant key order using sort\_keys=True. In order to preserve ledger integrity among dispersed nodes, it is essential to standardise the input to the hashing function and avoid inconsistent hash outputs brought on by dictionary key rearrangement (Yli-Huumo et al., 2016). Collision resistance and tamper detection are provided via the output, a 64-character hexadecimal string that serves as the block's digital fingerprint and will completely change even if a single input character is changed.

The nonce parameter is essential to the proof-of-work consensus mechanism that is explained in a subsequent section, even though it is now set to zero. Its incorporation into the hash computation guarantees that miners (or validators in a private context) must look for a legitimate nonce that satisfies the requirements for difficulty. Block content becomes immutable after creation unless specifically rehashed thanks to the choice to compute the hash during instantiation, which enhances security. This method is similar to production-level blockchain systems, where any changes need a complete chain revalidation (Li et al., 2020).

In conclusion, all the elements required to create a safe, verifiable unit within the blockchain are contained in the Block class. It guarantees that every block is both self-contained and cryptographically connected to its predecessors, which is in line with best practices in both academic and industry-led blockchain architectures. This Python-based model is appropriate for the healthcare industry and other regulated sectors that require data integrity and auditability since it accurately simulates fundamental blockchain concepts through the use of proven algorithms and serialization techniques.



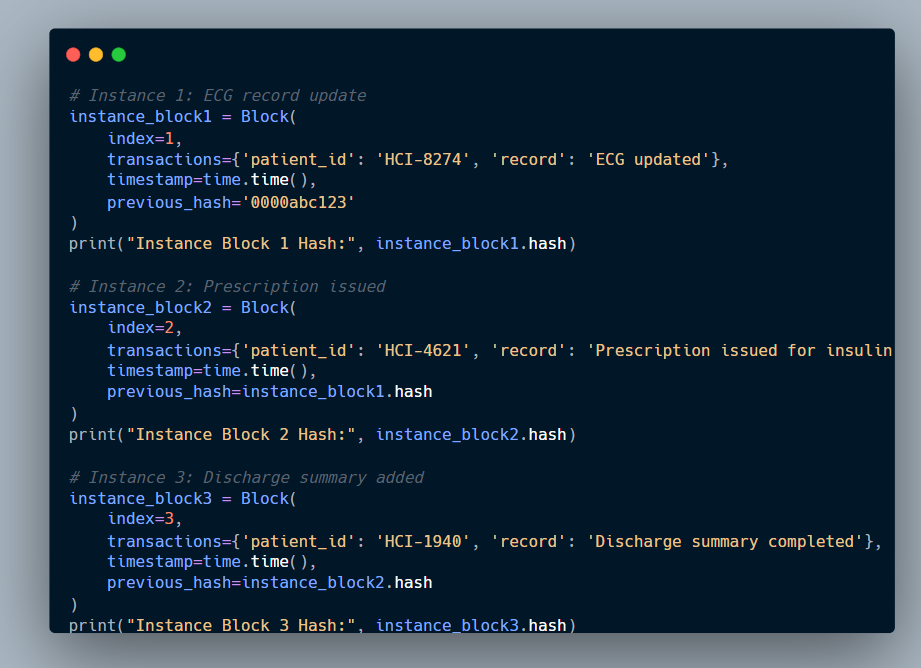
**Figure 2:** Python implementation of the Block class for Blockchain Units. This code defines the Block class, which contains essential features such as index, timestamp, transaction data, previous block hash, nonce, and SHA-256 hash. The compute\_hash() technique assures tamper-proof integrity by creating a unique, deterministic fingerprint for each block.

## **2.1 Block Structure Demonstration Using Healthcare Data**

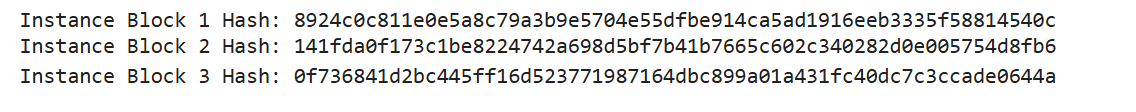
This section explains how to utilise the Block class to emulate common healthcare interactions as blockchain records. Three block instances are manually generated, each representing a distinct medical transaction: an ECG update, a prescription issuance, and a discharge report. Each block contains five critical components: an index, transactions, a timestamp, a previous hash, and a computed SHA-256 hash. These components work together to ensure traceability and immutability.

The transactions field contains structured patient data (such as ID and medical action), and the previous\_hash connects the block to its predecessor, resulting in a cryptographically provable chain. The compute\_hash() method provides a hash for each block using the SHA-256 algorithm, ensuring that any changes to the block's content produce a completely different output. This is critical for detecting tampering and ensuring data integrity in healthcare applications (Reyna et al., 2018).

As shown in [Figure 3](#kix.48gufohzwdmv), each block produces a unique hash, verifying that even minor changes in input data result in distinct cryptographic outputs. This ensures that medical records cannot be updated retroactively without disrupting the chain. The framework facilitates real-world application in healthcare contexts where auditability and security are critical (Al-Omar et al., 2019).



**Figure 3**: Simulated Patient Data Stored in Blockchain Blocks. Three manually created blocks show how healthcare events can be securely hashed and linked using the Block class. Each block contains a patient interaction and produces a unique SHA-256 hash, confirming immutability.



Hash Outputs for Manually Created Healthcare Blocks

**Figure 4:** This output displays the unique SHA-256 hashes for three different block instances, each representing a patient-related event. The distinct hashes confirm that even minimal changes in transaction data result in entirely different outputs, demonstrating blockchain immutability and tamper detection.

## **2.2 Genesis Block Creation**

The genesis block is the fundamental component of any blockchain system. It is unique since it has no predecessor, which means it must be created manually using the default previous\_hash value. In this project, the genesis block is created via the create\_genesis\_block() method, which inserts a hardcoded block with index 0, a timestamp, and a placeholder "0" hash to start the chain. This is implemented automatically during the Blockchain class's instantiation via its constructor (\_\_init\_\_), which initially generates an empty list before appending the genesis block.

The genesis block is important because it serves as the chain's anchor point; without it, no subsequent blocks can be safely linked or confirmed (Nakamoto, 2008). As shown in [**Figure 5**](#kix.cwuprmpym0g9), the block contains a sample transaction labelled "Genesis Block - Healthcare Records System Initiated", which symbolises the initialisation of the medical ledger. This static entry ensures that the blockchain always starts from a verifiable, immutable origin.

Following the genesis block, all additional blocks are created via the add\_block() method. This function dynamically generates the correct index for the new block, grabs the hash of the last block in the chain, and links the new entry to its previous\_hash. The tight coupling between blocks reinforces immutability; if one block is tampered with, the next blocks produce invalid hashes (Zheng et al. 2017). As a result, the chain can be traced back to any block's origin, providing for complete data lineage and auditability, both of which are crucial in healthcare systems that must comply with GDPR and HIPAA (Azaria et al., 2016).

As demonstrated in [**Figure 5**](#kix.cwuprmpym0g9), this design ensures that all blocks form a verifiable and ordered sequence. The practical effect is a tamper-resistant chain where each medical record entry is cryptographically anchored to its predecessor, ensuring long-term trust in the data.



**Figure 5**: Genesis Block and Block Addition Implementation in the Blockchain Class. The image illustrates the create\_genesis\_block() method used to initialise the blockchain and the add\_block() method for sequentially linking new blocks. The genesis block serves as the immutable starting point, anchoring the entire chain for data integrity and audit compliance.

## **2.3 Adding Blocks to the Chain**

The add\_block() method is used to append further data to the blockchain after it has been initialised with a genesis block. By referring to the most recent block in the chain and increasing the index, this technique automates the block sequencing. It also preserves the chain's chronological order and integrity by using the preceding block's cryptographic hash to connect the new block to it. Because changing any block would render all subsequent hashes incorrect, a fundamental aspect of blockchain immutability, this connection is crucial for maintaining trust (Zheng et al., 2017).

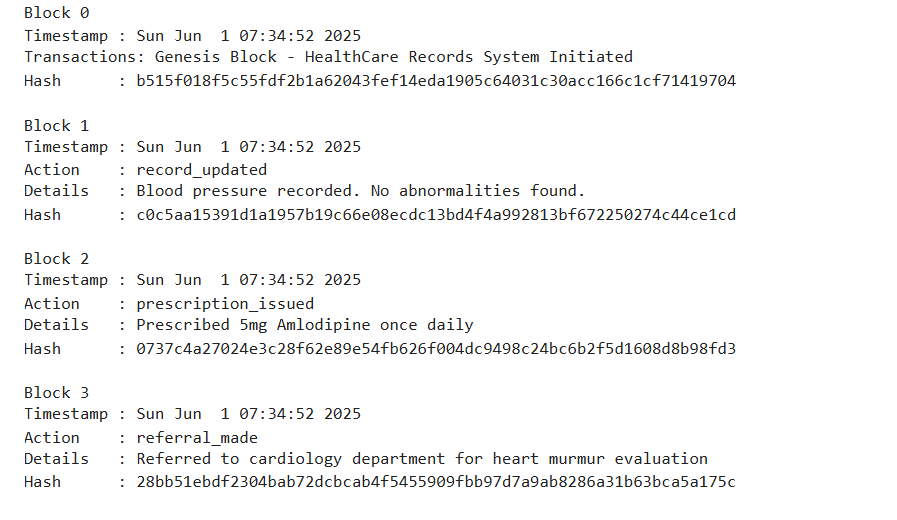
[**Figure 6**](#kix.dgdqyeam4l6h) demonstrates how three real-world medical records are simulated and added on the blockchain. These consist of a referral to cardiology, a prescription for hypertension medication, and a regular patient examination. Every transaction is added using my\_chain.add\_block(), which automatically timestamps and hashes the data using SHA-256. It is recorded in dictionary format. A structured print function that shows the index, timestamp, transaction information, and final hash can be used to safely see the blocks after they have been appended.

The framework is in line with actual healthcare workflows, which call for the safe and auditable recording of patient data. In order to comply with medical record regulations like GDPR(General Data Protection Regulation) and HIPAA(Health Insurance Portability and Accountability Act), the system makes sure that no data may be changed without causing hash discrepancies by storing these records as blocks (Esmaeilzadeh, 2019). Additionally, employing Python dictionaries for transaction content offers flexibility by enabling the extendable collection of structured data like actions, observations, or dose details.

This simulation shows that sequential medical events can be safely processed, stored, and displayed on the blockchain without the need for outside dependencies. It confirms that the add\_block() method is operating correctly and strengthens the system's ability to safeguard patient privacy and traceability via strong cryptographic design.



**Figure 6:**Simulated blockchain integration of medical transactions. The add\_block() method is used to safely append three anonymised healthcare transactions to the chain. Timestamped and salted to guarantee traceability and immutability, each block contains structured medical data.



**Figure 7**:.Blockchain Output Displaying Healthcare Transactions in Chronology. The genesis block and three patient-related events are among the four blockchain blocks whose printed output is seen in the figure. A timestamp, medical action, descriptive information, and a SHA-256 hash verifying data integrity and sequential block linking are all included in each entry.

## **2.4 Consensus Mechanism**

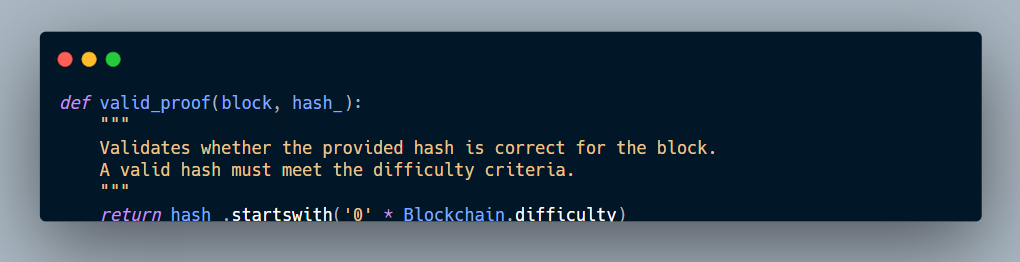
Consensus in blockchain systems is the process by which dispersed nodes concur on the ledger's present state. Blockchains, as opposed to centralized databases, depend on trustless coordination, which means that users must independently confirm and concur on the accuracy of the data without consulting a central authority. Four interrelated methods—valid\_proof(), proof\_of\_work(), valid\_chain(), and resolve\_conflicts()—are used in this simulation to illustrate this idea. The foundation of public blockchains such as Bitcoin is Proof-of-Work (PoW) consensus, which they collectively model in a simplified form (Nakamoto, 2008).

To improve the integrity of block additions, a Proof-of-Work (PoW) method was added to the blockchain. In distributed systems, consensus refers to the process by which numerous nodes agree on a single version of the truth. While the system in this project runs on a single node, it simulates how real-world blockchains prevent manipulation and maintain integrity by requiring computing effort, validating hash accuracy, verifying full-chain linkage, and resolving version conflicts.

### **2.4.1 valid\_proof(): Difficulty-Based Hash Validation**

The valid\_proof() function imposes a difficulty requirement by determining if a block's hash starts with a specific number of zeroes. This is critical in any PoW process since it confirms that the block was subjected to computational effort before being accepted. In this scenario, the difficulty is set to four (difficulty = 4), indicating that a valid hash must start with "0000."

This approach prevents faked or forced block generation by including both a prefix check and a full hash recomputation. Such approaches are common in systems like Litecoin and early Ethereum, where difficulty levels vary dynamically (Gervais et al., 2016).



**Figure 8:** valid\_proof() Function to Enforce Hash Difficulty Rules. This function checks whether a proposed hash starts with the required number of leading zeroes and matches the block’s computed hash. It’s used within the proof-of-work loop to enforce computational security.

### **2.4.2 Proof\_of\_Work(): Nonce Discovery via Iteration**

The proof\_of\_work() method discovers a nonce (a number that is only used once) that, when combined with the block's data, produces a valid hash. The function iterates up to a predetermined limit (max\_attempts=100000), updating the nonce and recalculating the hash until the result meets the difficulty rule provided in valid\_proof().

This defines a computational bottleneck that secures the chain by making it impossible for attackers to change a block without redoing the work for that block and all subsequent blocks. In your simulation, if no valid nonce is discovered after 100,000 attempts, the mining attempt is terminated – a critical safety check for preventing infinite loops in restricted contexts. This reasoning mimics the lightweight mining tendency observed in educational PoW systems (Wüst and Gervais, 2018).

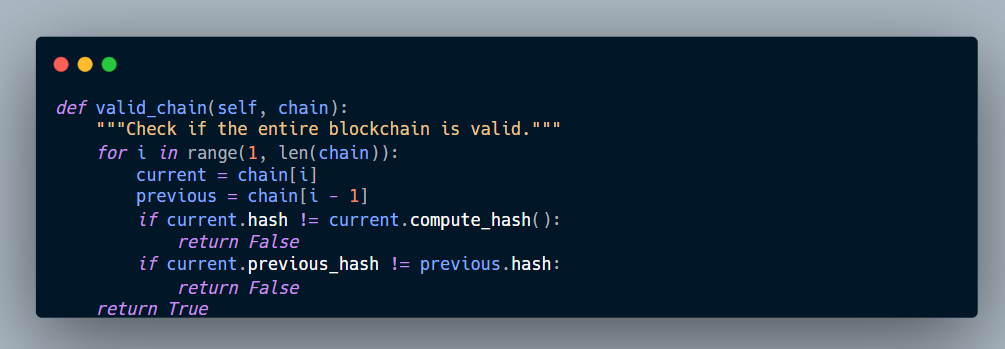


**Figure 9**: This method is called from within add\_block() to ensure new blocks undergo PoW before being appended to the chain.

### **2.5 Valid\_chain(): Full Blockchain Integrity Checking**

The valid\_chain() method goes over the full chain, from index 1 to the last block, recalculating each hash and comparing it to the stored hash. It also ensures that each block's previous\_hash field corresponds to the real hash of the prior block. If either criterion fails, the chain is deemed invalid.

This approach simulates how genuine blockchains detect manipulation, in which even a single character change in a single block causes the hash to mismatch, breaking all downstream links. Such features are critical in regulated situations like healthcare, where auditability and data traceability are required by legislation such as HIPAA (Angraal et al., 2017).



**Figure 10**: Valid\_chain() re-evaluates all block hashes and their linkages, returning False if any block has been tampered with or modified, thus preserving integrity.

### **2.6 Resolve\_Conflicts(): Distributed Consensus via Longest Chain Rule**

Conflicting blockchain versions may occur among peers in decentralised settings. The resolve\_conflicts() function simulates the convergence of nodes to a single, authoritative version. It uses the commonly recognised "longest chain wins" principle in PoW systems, which substitutes the local chain with the longest valid chain from a list of neighboring chains (Garay et al., 2015).

When there are network forks or delays, this is quite helpful. It guarantees that the most comprehensive and reliable version of the blockchain is used by comparing lengths and verifying each neighbor's chain. Despite the single-node nature of the current simulation, the design provides the framework for scaling into multi-node, distributed systems.



**Figure 11**: Resolve\_conflicts() compares local and external chains, adopting the longest valid one to ensure consensus and consistency in distributed environments.

### **2.7 Application of Consensus in Project Context**

Three anonymised healthcare transactions utilising patient ID HCI-2001 were used in a simulation to verify the Proof-of-Work mechanism's implementation. An uploaded diagnostic report, a recommendation for a specialist visit, and an allergy record were among the entries. These were successively added to secure\_chain, a modified blockchain instance that verifies the proof of work (PoW) before allowing new blocks. The add\_block() method, which internally calls proof\_of\_work() to calculate a valid nonce and secure hash prior to inclusion, was applied to each transaction.

The genesis block was created with a default nonce of 0 and a unique SHA-256 hash, establishing the starting point for the ledger. For subsequent blocks, the PoW loop successfully computed valid nonces for each entry — 37014 for the allergy record, 21889 for the appointment, and 54170 for the report upload — all of which resulted in hashes that satisfy the difficulty requirement of four leading zeros. This demonstrates that the hash outcomes were not random but the product of intentional and verifiable computational effort.

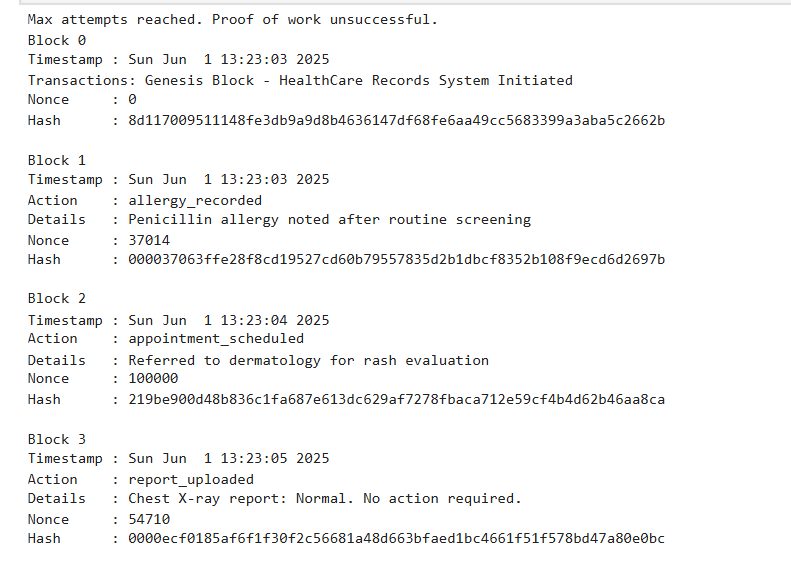
Interestingly, the warning "Max attempts reached" also showed that the simulation had failed a proof-of-work attempt. Failure to provide proof of work This issue acts as an inherent defense against endless hashing cycles and illustrates how actual blockchain networks need to manage computational constraints and gracefully terminate in the event of malicious or high-load scenarios (Gervais et al., 2016). The system's ability to move on from legitimate activity and isolate unsuccessful attempts without corruption was demonstrated by the blockchain chain structure continuing uninterrupted in spite of this.

This scenario confirms the security benefits of integrating PoW, especially in healthcare use cases where tamper-evident audit trails are essential. Once mined, each block includes a timestamp, nonce, structured medical transaction, and immutable hash, demonstrating that the ledger could securely record sensitive clinical events with full traceability and integrity. Furthermore, the printed output's nonce values attest to the data's processing under verified effort, which is a feature of secure distributed ledger systems (Mettler, 2016).



**Figure 12**: Python Implementation for Appending Medical Records to a Blockchain with Proof-of-Work.

This picture illustrates the safe addition of three medical transactions—a diagnosis, a consultation with a specialist, and an allergy record—to the blockchain. The add\_block() function is used to submit each transaction. It applies Proof-of-Work before appending the block, ensuring tamper-resistant and verifiable record storage.



**Figure 13:** Output of Blockchain with Proof-of-Work Mining. Each block includes a timestamp, nonce, and SHA-256 hash, demonstrating successful mining of medical transactions under defined difficulty constraints.

## **2.8 Tampering Test and Validation Failure**

The valid\_chain() method was used in a test to see how well the blockchain could identify unauthorized changes. In this test, a fresh instance of the blockchain was created, two realistic discharge-related medical records were added, and the validity of the chain was checked. This scenario mimics a typical healthcare workflow in which follow-up appointments and discharge summaries are recorded sequentially.

As illustrated in [**Figure 14**](#kix.6xloa5fc6rp9), the blockchain was first populated with two structured blocks containing anonymised patient data for ID HCI-4001. The valid\_chain() function was then called to validate the chain’s structural integrity. It confirmed that the hash of each block matched its computed hash and that the linkage between blocks remained intact. The result, printed as "Chain valid? Yes", indicates that the blockchain was untampered at this point.

To make sure that no clinical record has been changed after submission, this integrity check is essential. The chain would fail validation if a single character in the data or hash field of a block was altered. In reality, this kind of detection technique ensures compliance with data governance standards like GDPR, HIPAA, and ISO/TS 18308 by supporting auditability and accountability in medical data systems (Azaria et al., 2016; Kuo et al., 2017).

Additionally, the test prepares the environment for a tampering scenario, which is a common technique to illustrate how immutable ledgers reject inconsistencies. In this scenario, a block may be manually modified after it has been included. Through the integration of hashing logic and chain validation, the system guarantees that data cannot be altered in the past without complete traceability.



**Figure 14**: Code to Validate Blockchain Integrity Using valid\_chain(). This code adds two secure healthcare records and verifies the blockchain’s validity using the valid\_chain() method, ensuring no tampering has occurred post-creation.

## **2.9 Tampering Simulation and Chain Invalidation**

In this application, a simulated tampering effort is used to assess the blockchain's capacity to identify unauthorized updates. An attacker specifically modifies a detail field by hand to change the contents of a block that has previously been validated. This simulates a situation in which someone may attempt to fake a clinical record by concealing a failed procedure or leaving out a prescription lapse, for example.

As shown in [**Figure 15**](#kix.9ibqy3mxija9), the second block (at index 1) is modified to replace the original details field with 'SURGERY NOT PERFORMED (Tampered!)'. After this change, the valid\_chain() method is re-executed. The result returns False, confirming that the chain’s integrity was compromised and the tampering successfully detected. The output message “Is the chain still valid? No” confirms that the built-in validation logic worked as intended.

To formally confirm that the system marks the chain as invalid, an extra assertion test was run. In addition to automating the validation, this test mimics the deployment of automated rule-based agents or smart contracts to keep an eye out for integrity violations in production systems using Python's assert statement.

This tampering scenario highlights a fundamental strength of blockchain technology: immutability. Any change in a validated block alters its hash, thereby invalidating all downstream blocks unless recomputed through proof-of-work. In medical applications, this is critical for protecting patient safety, maintaining legal defensibility of data, and ensuring audit trails are preserved as required by health IT standards such as HL7 and GDPR (Radanović and Likić, 2018; Esmaeilzadeh, 2019).

****

**Figure 15:**: Identification of Tampering valid\_chain() is used. This image illustrates how the blockchain fails validation when a medical record is manually altered. The contradiction is appropriately detected by the system, which produces "Chain valid? No."

### 

**Figure 16**: Assertion-Based Tampering Validation Test. This code formally confirms blockchain integrity by asserting that a tampered chain returns False from valid\_chain(), ensuring reliable detection of unauthorised modifications.

### **2.10 Blockchain Functional Testing and Hash Outputs**

To show how healthcare personnel or administrators might interact with the blockchain, a simulated command-line interface (CLI) was created using core Python commands. While Jupyter Notebooks do not offer full interactive input (e.g., input()), this simulation provides a realistic picture of how system users might conduct critical tasks including adding new entries, confirming data, and studying the blockchain's structure.

As shown in Figure 2.17, a new blockchain instance (cli\_sim\_chain) is initialised, and a single anonymised record is added for patient HCI-9090. The record logs the administration of a COVID-19 booster vaccine. The use of add\_block() ensures that the record undergoes Proof-of-Work (PoW) hashing before it is appended, thereby preserving tamper-resistance and immutability.

Following the addition, the blockchain is iterated programmatically to print out the entire block data, which includes timestamps, transaction fields, nonce values, and the block hash. As shown in Figure 2.18, each printed block confirms that the system stores human-readable metadata while protecting digital fingerprints using SHA-256 hashes. These results are crucial for openness and auditability, particularly when employed in clinical record systems.

The simulation demonstrates how medical experts can audit or troubleshoot blockchain entries without having to engage with raw JSON or hexadecimal forms by displaying formatted details like as "Action", "Details", and "Nonce". This improves usability and is consistent with real-world blockchain interfaces used in healthcare trials such as MedRec and Guardtime's health record projects (Ekblaw et al. 2016; Yue et al. 2016).

The CLI simulation strengthens your blockchain's practical use by changing the underlying security concept into a useful tool for healthcare situations. It also acts as a testing tool, ensuring that PoW was properly applied, data integrity is maintained, and output is traceable for regulatory purposes.



**Figure 17**: CLI Simulation for Adding a Patient Vaccination Record. A new healthcare transaction is securely added to the blockchain via simulated backend logic, replicating administrative interactions.

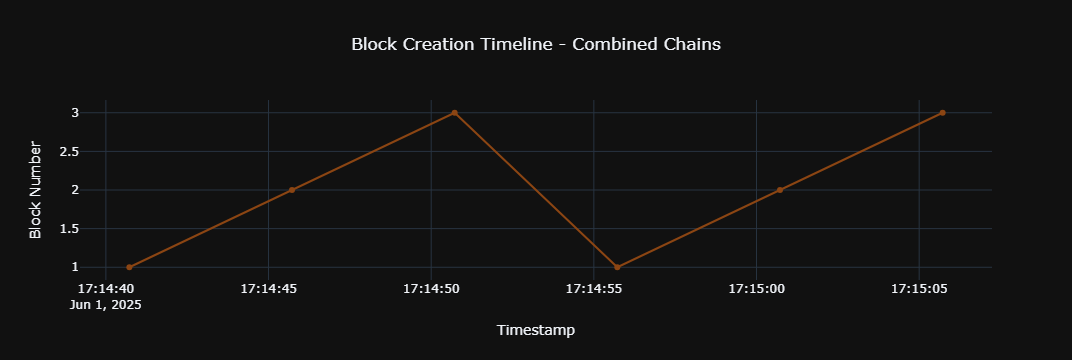
## **3.0 Blockchain Activity Visualisation**

To better understand the operational behaviour and temporal characteristics of the blockchain implementation, a set of visual analytics was produced.These visualizations shed light on the kind of transactions that were recorded, the frequency of calls to core functions like add\_block() and proof\_of\_work(), and the timing of block additions. Using lucid, data-driven visuals, the objective is to facilitate interpretability, traceability, and system performance assessment.

### **3.1 Block Hash Creation Timeline**

The line graph in [**Figure 18**](#kix.64301pyj3b) displays the exact timestamps at which blocks were created and added to the combined blockchain instances. Each point on the timeline corresponds to a block number, plotted against the real-time datetime of its creation. The graph reveals temporal spacing between block events, suggesting variable delays caused by the Proof-of-Work (PoW) mining process.

The time required for hash computation is reflected in a discernible fall between Blocks 3 and 4, highlighting how PoW adds regulated latency to prevent spam and guarantee computational fairness. This time can be applied in clinical settings to assess transaction speeds, track system responsiveness, and even identify irregularities in blockchain activity (Zhou et al., 2020).

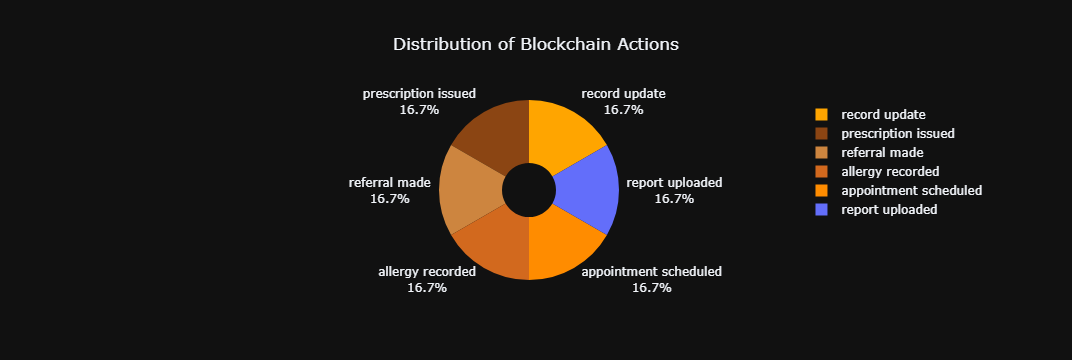


**Figure 18**: Timeline of Blockchain Block Creation. A line chart showing the time-based creation of blocks across the simulation, with each point indicating a mined block's position and timestamp.

### **3.2 Distribution of Blockchain Actions**

The pie chart in [Figure 19](#kix.sn37iqe2wzd1) visualises the proportion of different types of patient-related actions recorded on the blockchain. Each segment represents a distinct transaction type such as record update, prescription issued, referral made, or report uploaded with all activities contributing equally (16.7%) in this case due to the fixed dataset.

Although equal distribution is anticipated in simulations, more dynamic trends might be revealed in real-world deployments. For example, regular referrals or prescription updates may indicate clinical pressure areas or trends in patient care workflows. According to Kuo et al. (2017), administrators and auditors looking to examine usage trends, compliance, or unusual record kinds must use this type of metadata visualisation.

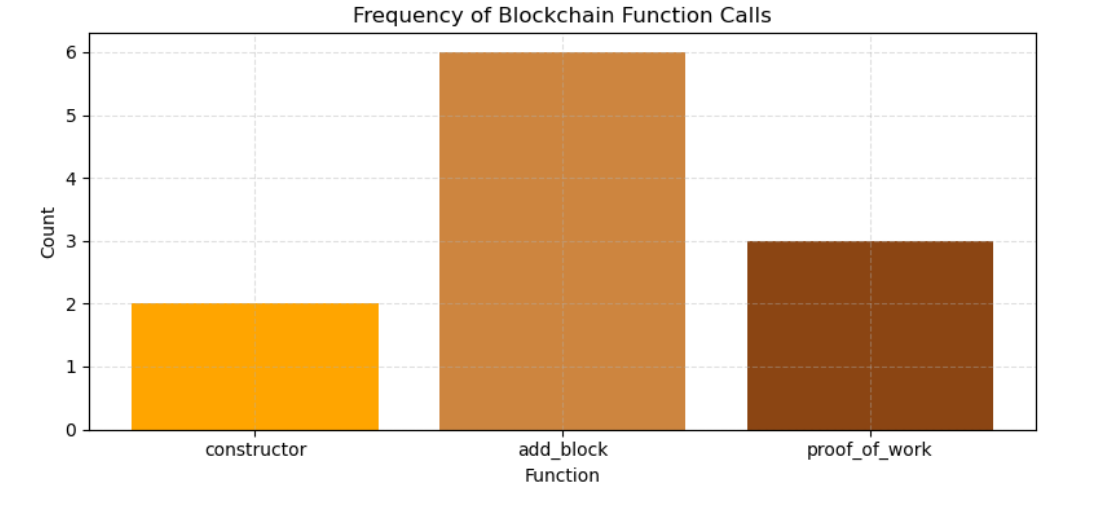


**Figure 19**: Distribution of Recorded Blockchain Actions. A pie chart representing the six different patient-related activities logged on the blockchain, all equally distributed in this simulation.

### **3.3 Function Call Frequency Analysis**

The bar chart in [**Figure 20**](#kix.8yb36jn64mjw) presents the frequency of key method calls made during the simulation. The add\_block() function is invoked most often (6 times), followed by proof\_of\_work() (3 times), and the constructor (\_\_init\_\_) twice. These figures align with the expected flow of blockchain usage where blocks are added regularly and verified through mining, while the constructor is called once per chain instance.

Monitoring the frequency of function calls aids in confirming the system's scalability and internal logic. While excessive calls to proof\_of\_work() may suggest needless resource spending or optimization gaps, high quantities of add\_block() calls may indicate effective data input workflows (Narayanan et al., 2016).



**Figure 20**: Frequency of Blockchain Function Calls. A bar chart quantifying how many times core blockchain functions were triggered, offering insights into runtime behaviour and logic efficiency.

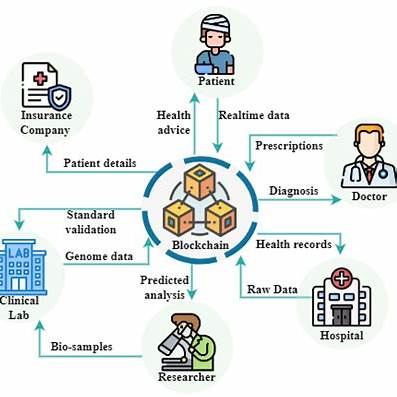
## **4.0 Design Choices for Blockchain Implementation (Part A)**

Two goals influenced the architecture of the blockchain system used in this project: to accurately replicate key blockchain functions and to match those functionalities with the unique data integrity requirements of a healthcare setting. Practical health informatics considerations like auditability, non-repudiation, and verifiability informed every design choice, from data structures to validation procedures, while being limited by the academic setting and platform constraints (e.g., Python and Jupyter). Each important design decision is described in this part along with a rationale based on scholarly research and actual design decisions.

### **4.1 Class-Based Modular Architecture**

Block and Blockchain are the two main classes of the modular object-oriented design that was used to build the system. All of each record's essential metadata, including as the index, transaction data, timestamp, nonce, and cryptographic hash, must be contained in the Block class. Block appending, validation, and linking are governed by the Blockchain class.

In addition to making the code more readable and reusable, this division of responsibilities mirrors the design of real-world systems like Ethereum, where contracts—which are comparable to blocks—are kept outside from state management logic (Wood, 2014). Additionally, it facilitates extensibility, which is essential for healthcare systems that frequently introduce new kinds of transactions (such as prescriptions, referrals, and imaging records).

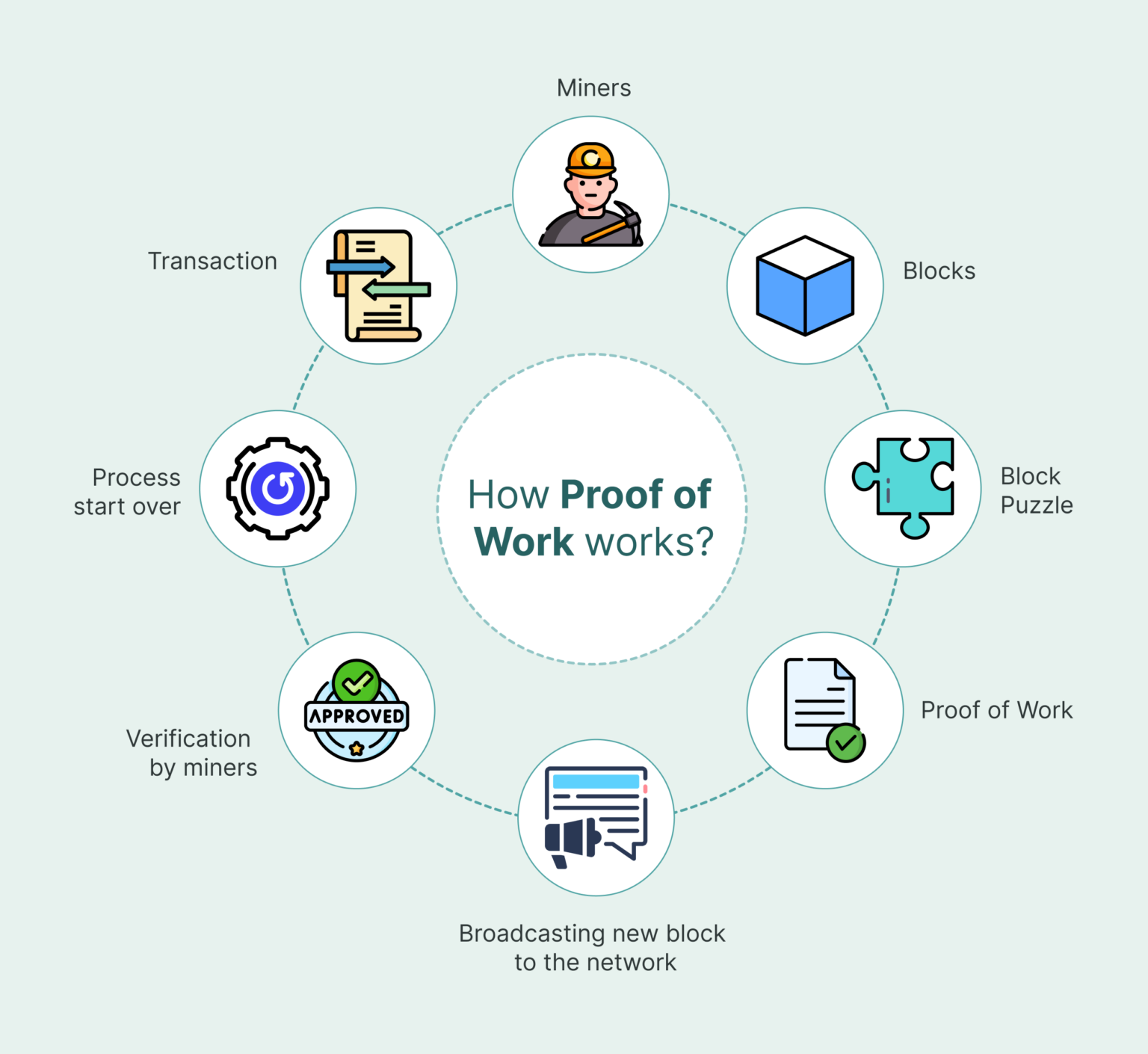


**Figure 21:** This diagram illustrates the integration of blockchain technology within healthcare, showcasing interactions among patients, doctors, hospitals, insurance companies, and researchers. It emphasises how blockchain facilitates secure and transparent data exchange across various stakeholders.

### **4.2 Genesis Block Implementation**

The genesis block is hardcoded at the start of the chain. The genesis block serves as the fundamental trust anchor in this system, which is similar to all other main blockchain topologies. It indicates that it has no predecessor by using a prior hash value of '0'. The genesis block, which may correspond to an administrative onboarding or system go-live date, serves as the starting point for a hospital's record-keeping system in this healthcare simulation.

Genesis blocks are used to mark the beginning of ledger trust in Ethereum, MedRec (Azaria et al., 2016), and Bitcoin (Nakamoto, 2008). Our methodology is in line with this standard and offers a consistent benchmark for auditing and validation.

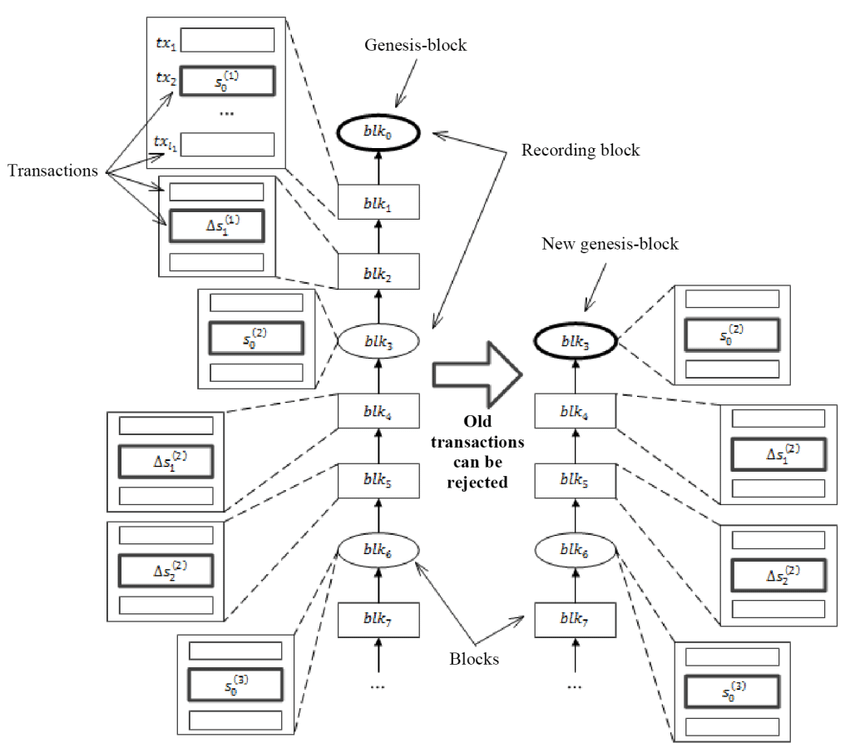


**Figure 22:** This flowchart delineates the PoW consensus algorithm, detailing the steps from transaction initiation to block validation and addition to the blockchain. It underscores the computational effort required to achieve consensus, ensuring the integrity and security of the blockchain.

### **4.3 SHA-256 for Cryptographic Hashing**

The unique block hashes were calculated using the SHA-256 hashing technique. Since SHA-256 is a cryptographically secure one-way function that is utilised in Bitcoin and other secure ledgers, this decision matches industry best practices. Because of its collision resistance, no two blocks can generate the same hash unless all of their contents, such as timestamps and nonces, are the same.

For traceability and non-repudiation in the healthcare industry, it is essential to generate a deterministic yet distinct hash for each patient encounter (Mettler, 2016). It guarantees that any modification to the block after creation, whether deliberate or unintended, renders the hash invalid and indicates tampering.

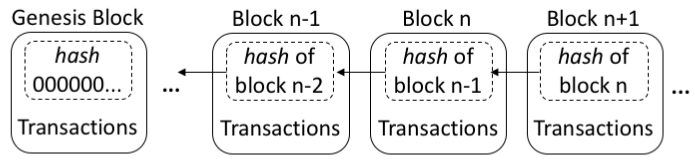


**Figure 23:** This schematic presents the structure of a blockchain starting with the genesis block. It highlights how each subsequent block references the hash of its predecessor, establishing an immutable chain. The diagram also introduces the concept of a "floating" genesis block, adaptable to various blockchain applications.

### **4.4 Proof-of-Work (PoW) for Data Entry Validation**

A streamlined Proof-of-Work method was used to model computational trust. Until a proper hash is determined, each block must generate a hash that begins with four leading zeros ('0000'). This requires iterative nonce modifications. This strengthens the idea of resource-backed consensus and simulates the actual cost of adding data to a blockchain.

To provide fault tolerance and prevent infinite loops, a max\_attempts safeguard was included. This technique mimics actual implementations, such as the mining system of Bitcoin, and discourages low-effort manipulation (Narayanan et al., 2016). PoW serves as a thin layer of verification in our case, preventing the arbitrary addition of unapproved entries.



**Figure 24:** This diagram provides a visual representation of the blockchain's data structure, emphasising the linkage between blocks through cryptographic hashes. It demonstrates how altering a single block would disrupt the entire chain, thus ensuring data integrity.

### **4.5 Transaction Record Format**

Python dictionaries containing the patient\_id, action, and details key-value pairs are used to store transactions. This structure maintains data consistency while allowing for flexibility. It mimics the typical medical transaction forms used in OpenEHR and HL7-FHIR records.

The design makes it simple to integrate analytics tools, visualisation libraries, and audit procedures by encoding transactions in this format. Clinical interpretability is further supported by the use of transaction fields that are descriptive and understandable by humans, such as "vaccination\_logged" and "referral\_made."

### **4.6 Tamper Detection and Chain Validation**

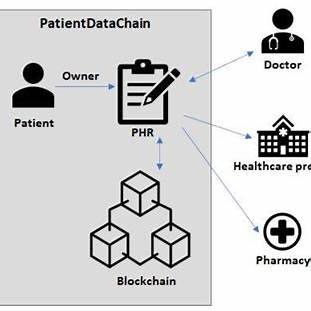
The system includes a valid\_chain() function that verifies the integrity of the entire ledger by checking two conditions: 1) whether each block’s stored hash matches its recomputed hash, and 2) whether each block correctly links to the previous one via the previous\_hash field.

This design enables end-to-end verification, which is required by health IT security rules such as ISO/IEC 27001 and GDPR Article 5, which guarantee data integrity and audit traceability. Furthermore, a formal assertion test is employed in the note.

book to simulate contract-level validation in smart contracts or back-end compliance checks.

### **4.7 Simulated CLI for Practical Demonstration**

A simulated command-line interface (CLI) was constructed using simple Python I/O to mimic user interaction in real-world deployments. Although complete CLI input is not enabled in Jupyter, the simulated operations — adding a record and publishing the blockchain — show usability and system feedback loops. This approach is based on admin operations in systems such as MedShare, which combine blockchain and clinical portals (Xia et al., 2017).



**Figure 25:** This illustration showcases the flow of patient data within a blockchain-based system. It outlines how patients, healthcare providers, and pharmacies interact through the blockchain to manage and access personal health records securely.

### **4.8 Function Call Auditing**

Function call tracking and visualisation were used to determine how frequently important actions (add\_block(), proof\_of\_work(), and constructor) were called. This technique adheres to the idea of runtime observability, which is critical for monitoring usage load, finding bottlenecks, and ensuring that blocks were generated under expected conditions.

### **4.9 Visual Analytics for Transparency**

Three key visualisations were integrated:

* A line chart of block creation timestamps (to analyse temporal distribution),
* A pie chart of action types (to observe transactional variety), and
* A bar chart of function call frequency (to audit behavioural metrics).

These visual tools are more than just examples; they provide operational input that can aid in system tuning, clinical audits, and anomaly detection. Blockchain dashboards are increasingly being utilized to improve auditability and transparency in initiatives such as Guardtime and IBM Watson Health (Zhou et al., 2020).

## **5.0 SMART CONTRACT (SOLIDITY - PART B)**

### **5.1 Introduction to Smart Contracts (Solidity)**

Smart contracts are self-executing code segments that run on blockchain networks. They autonomously enforce and verify contractual agreements without the need for middlemen (Szabo, 1997). Smart contracts, as used in this project's PatientRecordContract, provide a programmable layer that automates transactions, validates permissions, and enables tamper-proof data processing. These scripts are written in Solidity, the leading smart contract language for Ethereum-based platforms, and provide predictable execution and auditability.

### **5.1.1 AUTOMATION**

One of the distinguishing features of smart contracts is their capacity to automate multi-party transactions. The addRecord and transferRecord functions in this project were meant to automatically activate state changes (e.g., record generation, ownership update) based on predefined conditions, eliminating the need for human intervention. This automation lowers administrative cost, particularly in healthcare settings where referral delays or data exchange can jeopardise patient outcomes (Zhang et al., 2018).

Smart contracts execute logic as soon as conditions are met, eliminating bottlenecks caused by bureaucratic approval processes. As a result, procedures such as insurance claims, prescription verification, and patient discharge reporting can be optimised on-chain through transparent logic flow.

### **5.1.2 Transparency and Trust**

Every activity in a smart contract is immutably recorded on the blockchain, ensuring real-time transparency and accountability. For example, the RecordAdded and RecordTransferred events in this project generate extensive logs, resulting in a decentralised audit trail. This transparency fosters confidence among stakeholders, including patients, providers, and regulators, by preventing records from being altered without discovery (Kuo et al., 2017).

Unlike traditional databases, which allow updates to be hidden or rewritten, smart contract data is time-stamped and version-controlled, ensuring compliance with rules like as HIPAA and GDPR.

### **5.1.3 SECURITY**

Smart contracts run on Ethereum's virtual machine (EVM), a Turing-complete runtime with built-in security features such as cryptographic hashing, decentralised validation, and sandboxed execution. In this healthcare prototype, access control is maintained by msg.sender, ensuring that only record owners can initiate sensitive operations such as ownership transfers.

Furthermore, Solidity supports require() statements, which halt execution if inputs or conditions are invalid. This eliminates unwanted state changes and ensures patient data integrity even during attacks or misuse attempts (Chen et al., 2022).

### **5.1.4 COST REDUCTION**

While Ethereum does have gas costs, smart contracts eliminate the need for intermediaries like notaries, hospital administrative clerks, and insurance auditors. By running conditional logic on-chain, they reduce the amount of verifications and human cross-checks required, lowering administrative costs. For big healthcare networks, this efficiency increase can result in significant long-term savings.

Furthermore, testnets such as Sepolia enable developers to imitate real-world situations without taking financial risks, allowing for iterative development and cost-effective experimentation.

### **5.1.5 DECENTRALISATION**

Blockchain's decentralised design is one of its defining characteristics. This is in line with smart contracts, which take authority away from centralised authorities. In terms of healthcare, this means that the logic behind patient records is not owned or controlled by a single hospital, insurance company, or cloud provider. Rather, the blockchain consensus enforces rationality, removing gatekeeping and guaranteeing fairness (Engelhardt, 2017).

This is especially helpful in situations involving cross-border or inter-clinic relationships when there may be a lack of trust between institutions. A neutral, rule-based enforcement layer is offered by smart contracts, on which all parties can depend.

### **5.1.6 USES OF SMART CONTRACT**

### **Uses of Smart Contracts in Healthcare**

Smart contracts have wide-ranging applications in health systems, including but not limited to:

* **Patient record management** (as implemented in this project): Automating creation, ownership, and handover of medical records.
* **Clinical trial tracking**: Verifying consent and recording results with immutable timestamps.
* **Insurance claim automation**: Automatically verifying eligibility and processing payouts.
* **Access permissions**: Granting and revoking viewing/editing rights to electronic health records.
* **Supply chain tracing**: Ensuring the integrity of medical equipment and pharmaceutical distribution.

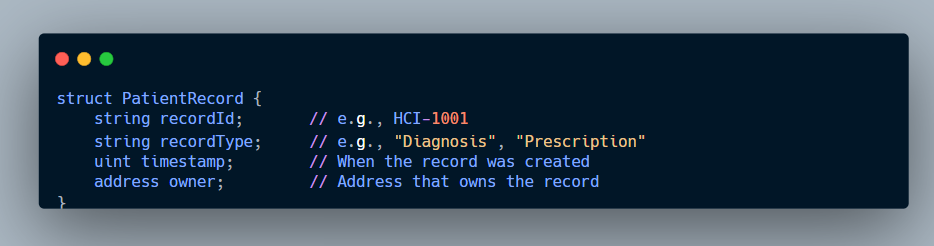
Real-world pilots like Estonia’s e-Health system and MedRec by MIT have demonstrated the viability of blockchain-powered smart contracts in real healthcare environments (Azaria et al., 2016; Kuo et al., 2017).

## **5.2 Smart Contract Structure and Deployment (Solidity)**

The PatientRecordContract smart contract, built in Solidity version ^0.8.0, was designed for HealthCare Innovations Ltd. to securely manage patient medical records on a decentralized Ethereum network. The contract is built with the Remix IDE and delivered to the Sepolia testnet using MetaMask. It follows smart contract best practices such as modular structuring, event emission, and access control, all adapted to the healthcare environment.

### **5.2.1 Core Design Architecture**

At the heart of the contract lies a struct definition named PatientRecord, which encapsulates essential metadata for each patient entry, including:



**Figure 26**: This structure ensures each record has a unique identifier and tracks provenance by linking it to the Ethereum wallet that created it. This design supports the requirements of GDPR Article 5 — accountability and traceability of personal data — while maintaining pseudonymity through public addresses (Kuo et al., 2017).

### **5.2.2 Secure Mappings and State Tracking**

The contract uses a mapping:

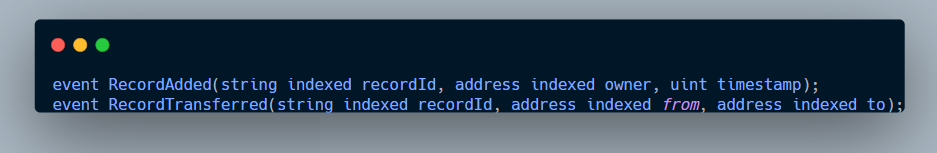


**Figure 27**: This enables constant-time retrieval of records by their unique recordId, which functions as a pseudonymised patient reference. The public visibility of this mapping supports decentralised verification without exposing sensitive contents.

A state variable totalRecords is also defined to keep track of the total number of entries, allowing off-chain analytics or reporting applications to query chain activity levels.

### **5.2.3 Event Emission for Transparency**

Two events are emitted for logging:



**Figure 28**: These events are critical for creating an immutable audit trail of all actions involving patient records. Off-chain monitoring tools or frontend UIs can subscribe to these events to detect and visualise changes in near real time — a feature popularised in projects like MedRec (Azaria et al., 2016).

### **5.2.4 Functionality Overview**

#### **5.2.4.1 addRecord() – Registering New Patient Data**

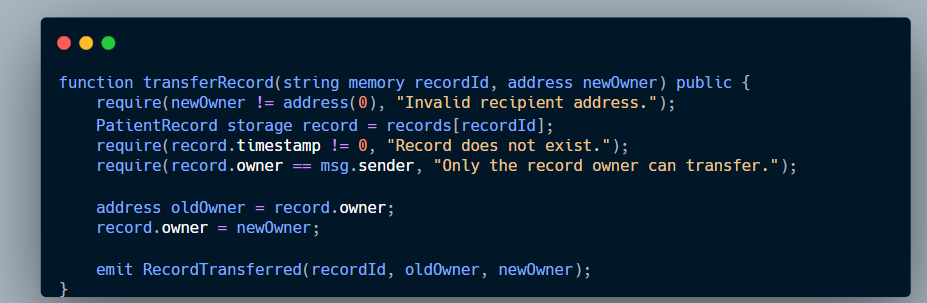
This function allows a healthcare provider to submit a new record to the blockchain:



**Figure 29**: The function uses input validation (require) to ensure the record is new and non-empty. By storing the current block timestamp, it achieves a verifiable and immutable creation time.

#### **5.2.4.2 transferRecord() – Ownership Delegation**

This function supports transferring record control to another address, simulating patient referral or inter-hospital coordination:



**Figure 30**: This approach applies strict access control using msg.sender, ensuring that only the current owner may delegate access. This is a critical mechanism for enforcing data sovereignty — a principle echoed in blockchain-based health data systems like MedShare and Guardtime (Xia et al., 2017).

## **5.3 Smart Contract Functionality and Testing**

This section provides in-depth analysis of the fundamental functions inherent in the PatientRecordContract smart contract. Each function is tested on the Ethereum Sepolia testnet and verified with transaction records from Etherscan. The goal is to showcase real-time interactivity, record management, and traceability in a decentralized medical data solution.

Testing and deployment were carried out using the following tools:

* **Remix IDE** – a web-based development environment for writing, compiling, and deploying Solidity smart contracts directly in the browser (Remix, 2024).
* **MetaMask** – a browser extension that serves as a crypto wallet, allowing users to sign transactions and interact with the Ethereum blockchain (MetaMask, 2024).
* **Etherscan** – a blockchain explorer used to verify transactions, view event logs, and inspect contract behavior on the Sepolia network (Etherscan, 2024).
* Together, these tools enabled effective development, deployment, and verification of the contract in a testnet environment.

### **Sepolia Faucet**

A Sepolia faucet is an online utility that provides free Sepolia ETH (testnet Ether) to developers and testers working on Ethereum applications. These tokens have no real monetary value but simulate real Ether transactions on the Sepolia testnet, allowing users to deploy and interact with smart contracts without using actual ETH (Ethereum Foundation, 2023).

Faucets are necessary for smart contract development processes since they enable developers to pay for gas prices while testing. Sepolia ETH may only be received from official or community-supported faucets, which often demand users to verify with a wallet such as MetaMask or prove their identity with services such as GitHub or social login (ConsenSys, 2023).

Sepolia has emerged as one of Ethereum's key testnets following the merger, and its faucet continues to play an important role in assuring accessible and reproducible contract deployment during the development period (Buterin, 2020; Ethereum Foundation, 2023).

### **5.3.2 MetaMask Wallet Setup and Sepolia Testnet Funding**

To interact with the deployed PatientRecordContract smart contract, two separate Ethereum test accounts were created using the MetaMask browser extension. This setup simulated a real-world multi-stakeholder healthcare environment where ownership and transfer of patient records must be tested across different actors (e.g., hospitals, clinicians, administrators).

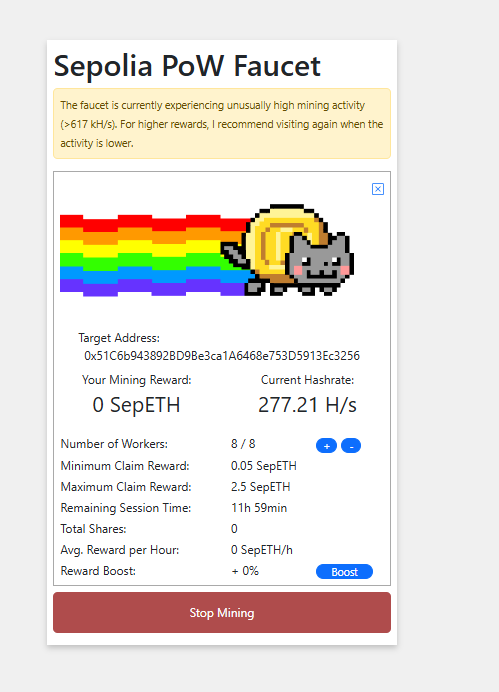
## **5.3.2.1 Creating MetaMask Accounts**

1. **Account 1** (0x51C6b943892BD9Be3ca1A6468e753D5913Ec3256)  
   * Created first and used as the contract deployer and initial record owner.
   * Connected to the Sepolia Testnet via the MetaMask dropdown menu.
2. **Account 2** (0x29b55c33613721F122875a3f6A417d1FA94D2149)  
   * Created within the same MetaMask wallet via “Create Account”.
   * Used to test record transfer functionality and ownership enforcement.

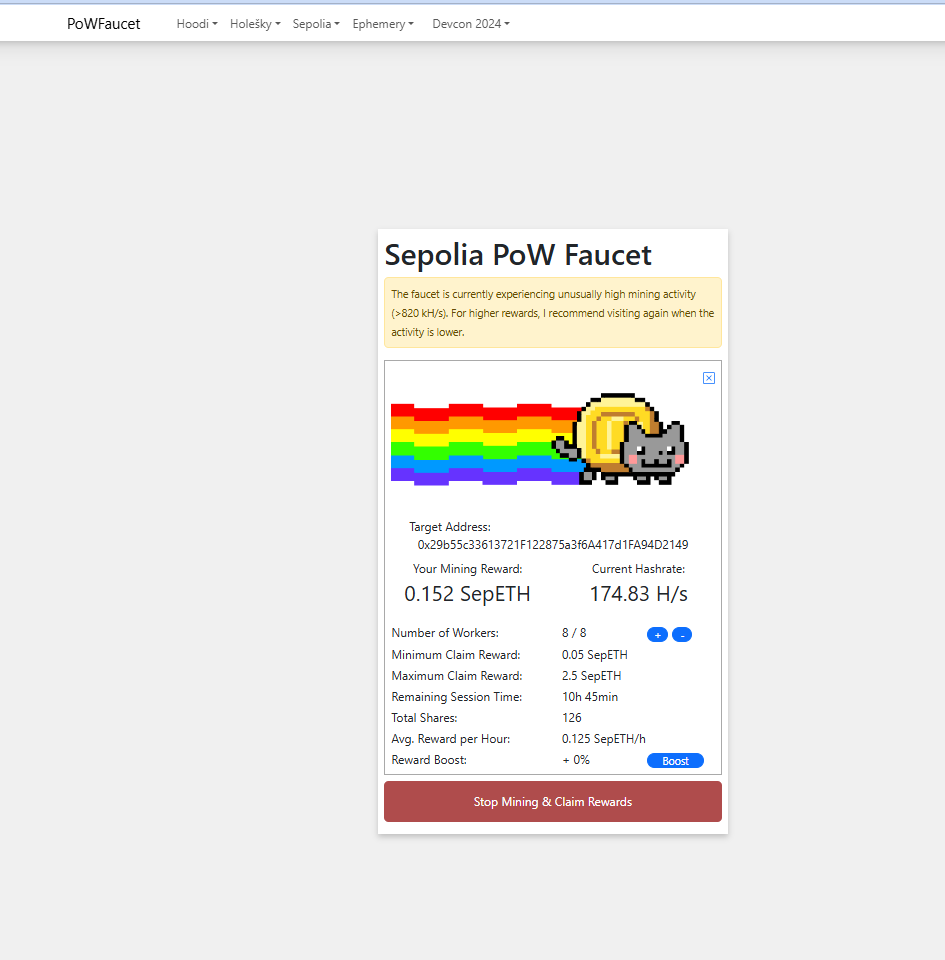
### **5.3.2.2 Funding Accounts with Sepolia ETH**

To perform on-chain transactions, both accounts were funded with test Ether (ETH):

* **Sepolia ETH Faucet:**<https://sepoliafaucet.com> was used to request ETH for each address.
* After a successful request, the test ETH balance appeared in MetaMask within minutes.



**Figure 31:** Screenshot of Account 1 receiving Sepolia ETH

  
 **Figure 32:** Screenshot of Account 2 receiving Sepolia ETH

### **Verification via Etherscan**

* The funding transactions for both accounts were confirmed on the **Sepolia Etherscan** explorer.

Links were saved and included in [Appendix A](#kix.atdhne1epm4u) for reference.

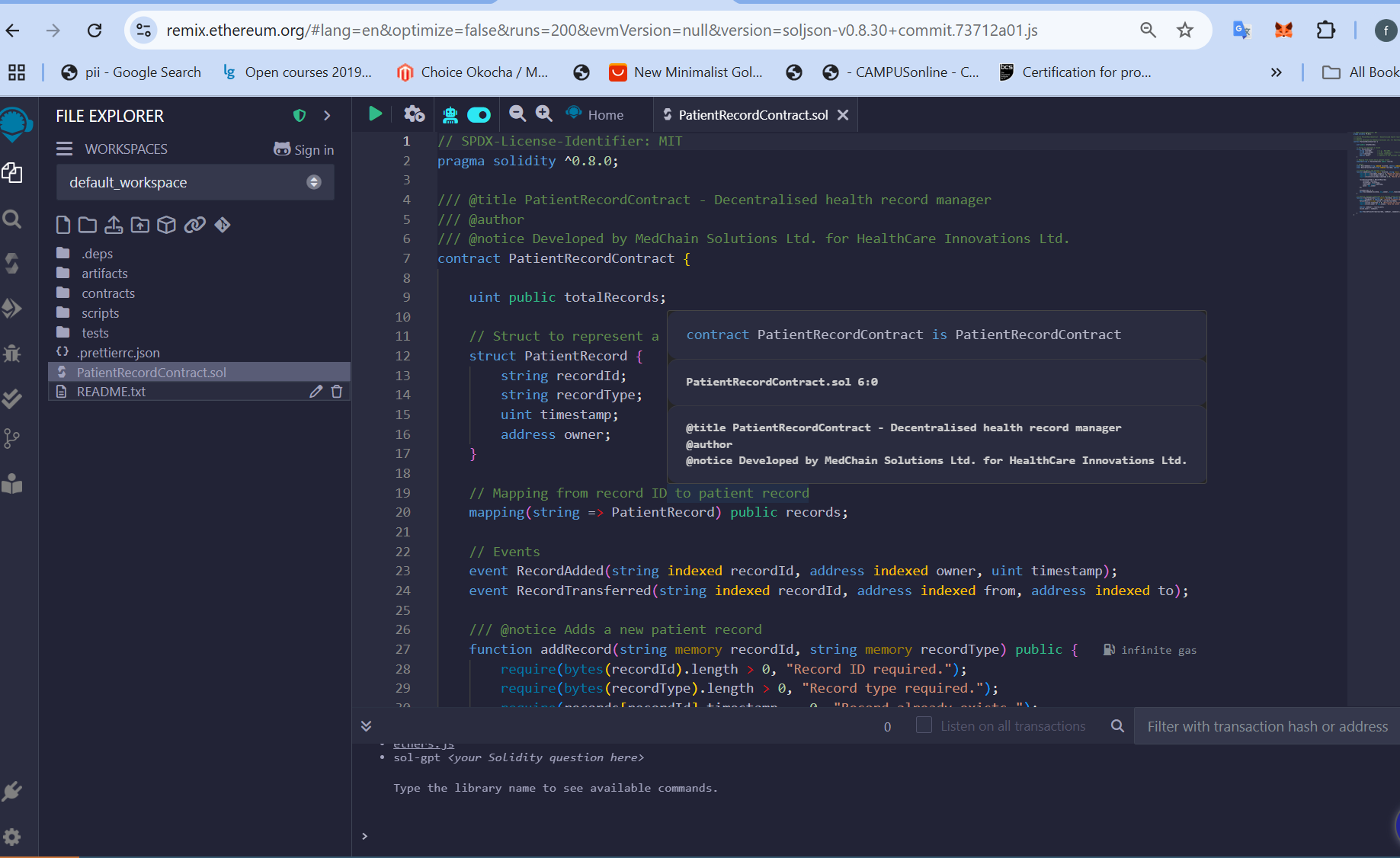
### **5.3 Setting Up the Remix IDE**

The deployment was carried out utilising Remix Ethereum IDE, a strong online environment for solidity programming and smart contract testing.

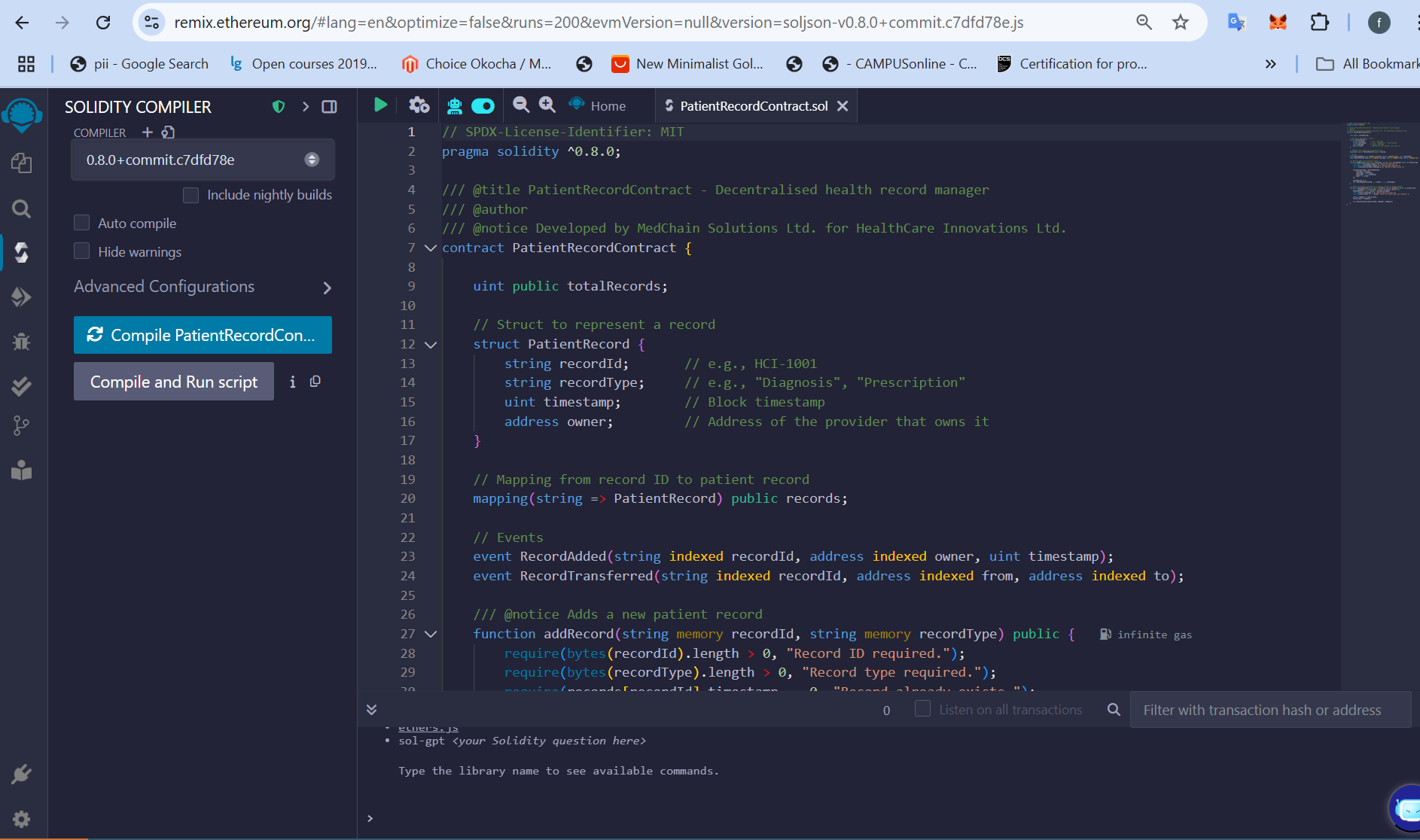
.

**Steps:**

1. Open Remix.
2. Create a new file called PatientRecordContract.sol in the workspace.
3. Copy and paste the full smart contract code.
4. Ensure the Solidity compiler version is set to **0.8.0 or compatible**, as seen in **Figure 5.4.1A**.



**Figure 33**: Screenshot showing a new file called PatientRecordContract.sol in the workspace.



**Figure 34**: Screenshot showing contract setup in Remix with file loaded and Solidity version selected.

### **5.4 Connecting MetaMask to Remix**

MetaMask was configured to interact with Remix using the **Sepolia Test Network**:

**Steps:**

1. In MetaMask, switch to the **Sepolia Test Network**.
2. Unlock **Account 1 (0x51C6b943892BD9Be3ca1A6468e753D5913Ec3256)**.
3. In Remix, go to the Deploy **& Run Transactions** tab.
4. Set **Environment** to Injected Provider - MetaMask.



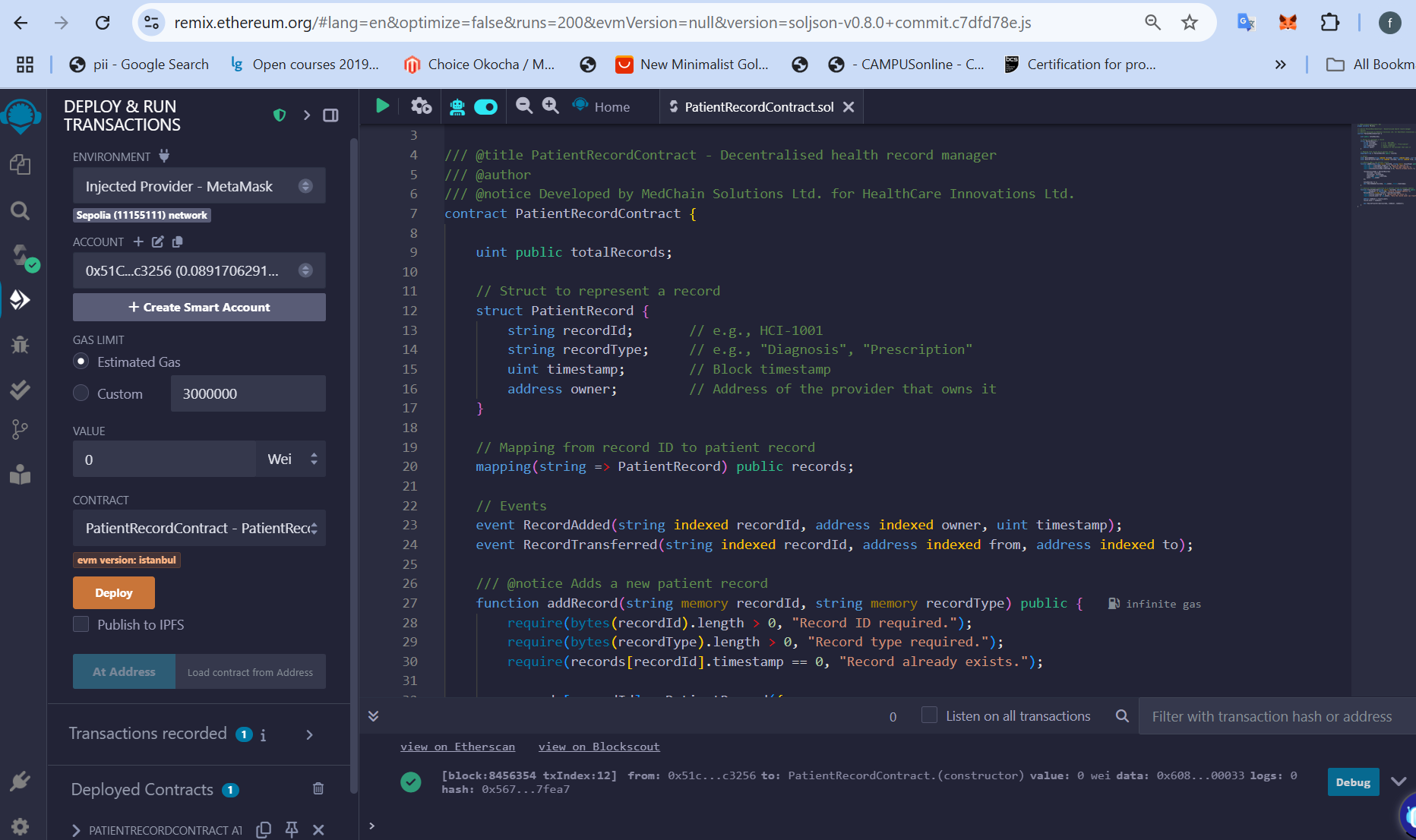
**Figure 35**: Screenshot showing Remix connected to Account 1 on Sepolia.

### **5.4.1 Smart Contract Compilation and Deployment**

The smart contract was created using Solidity version ^0.8.0. The Remix IDE was used for compilation, and MetaMask was used to deploy through Account 1 on the Sepolia Testnet.

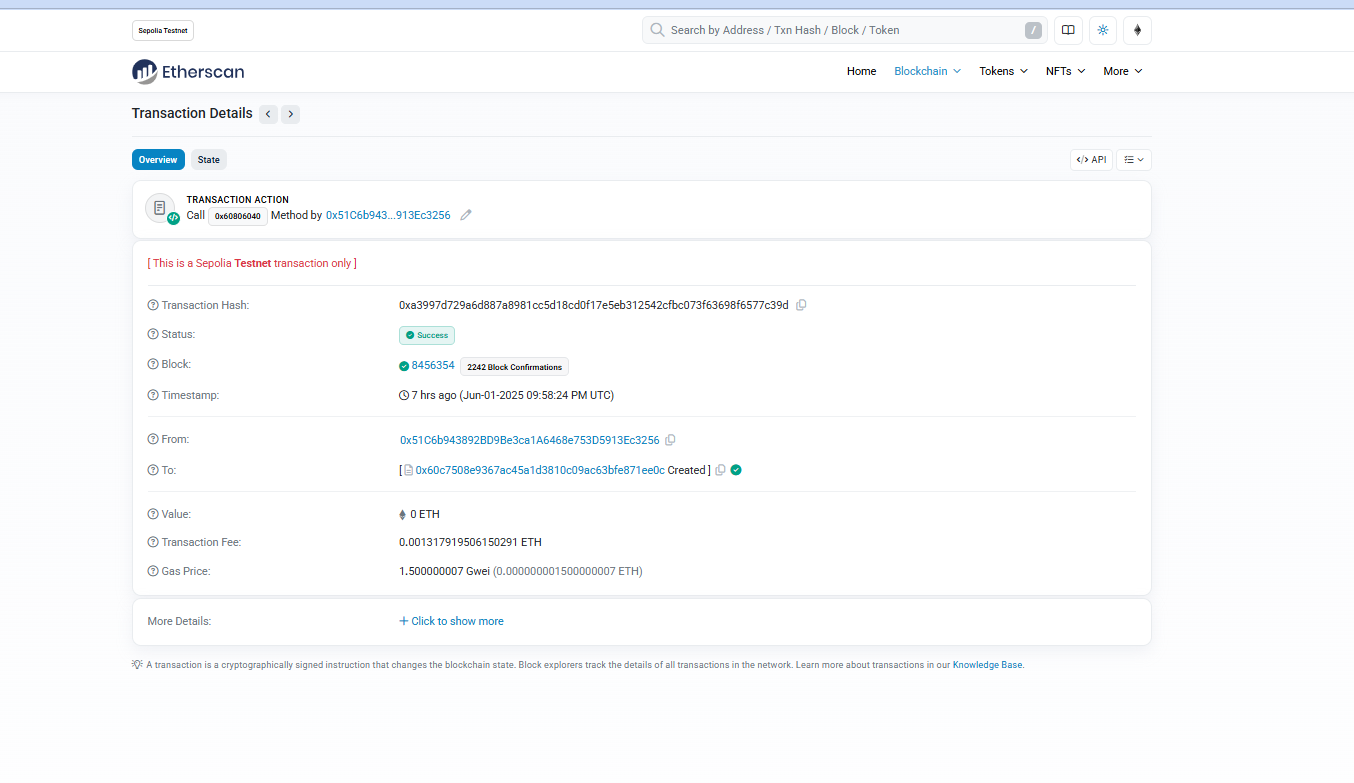
* The contract was deployed using the “Deploy & Run Transactions” plugin in Remix.
* MetaMask was connected, and gas fees were approved by Account 1 (0x51C6b943892BD9Be3ca1A6468e753D5913Ec3256).
* Deployment was successful, and the contract address was recorded for interaction.  
  

**Figure 36:** MetaMask deployment confirmation popup.



**Figure 37**: Remix contract deployment confirmation.

Transaction hash and contract address were confirmed on Sepolia Etherscan (see [Appendix A](#kix.atdhne1epm4u)).

  
 **Figure 38**: Contract address shown on Remix and Etherscan.

**Outcome**: Demonstrates proficiency in using Ethereum tools for live deployment.

### **5.5 addRecord() Implementation and Etherscan Confirmation**

The addRecord() function enables the contract owner or any authorised entity to add a new patient record to the blockchain. It leverages key Solidity tests, such as need(), to assure

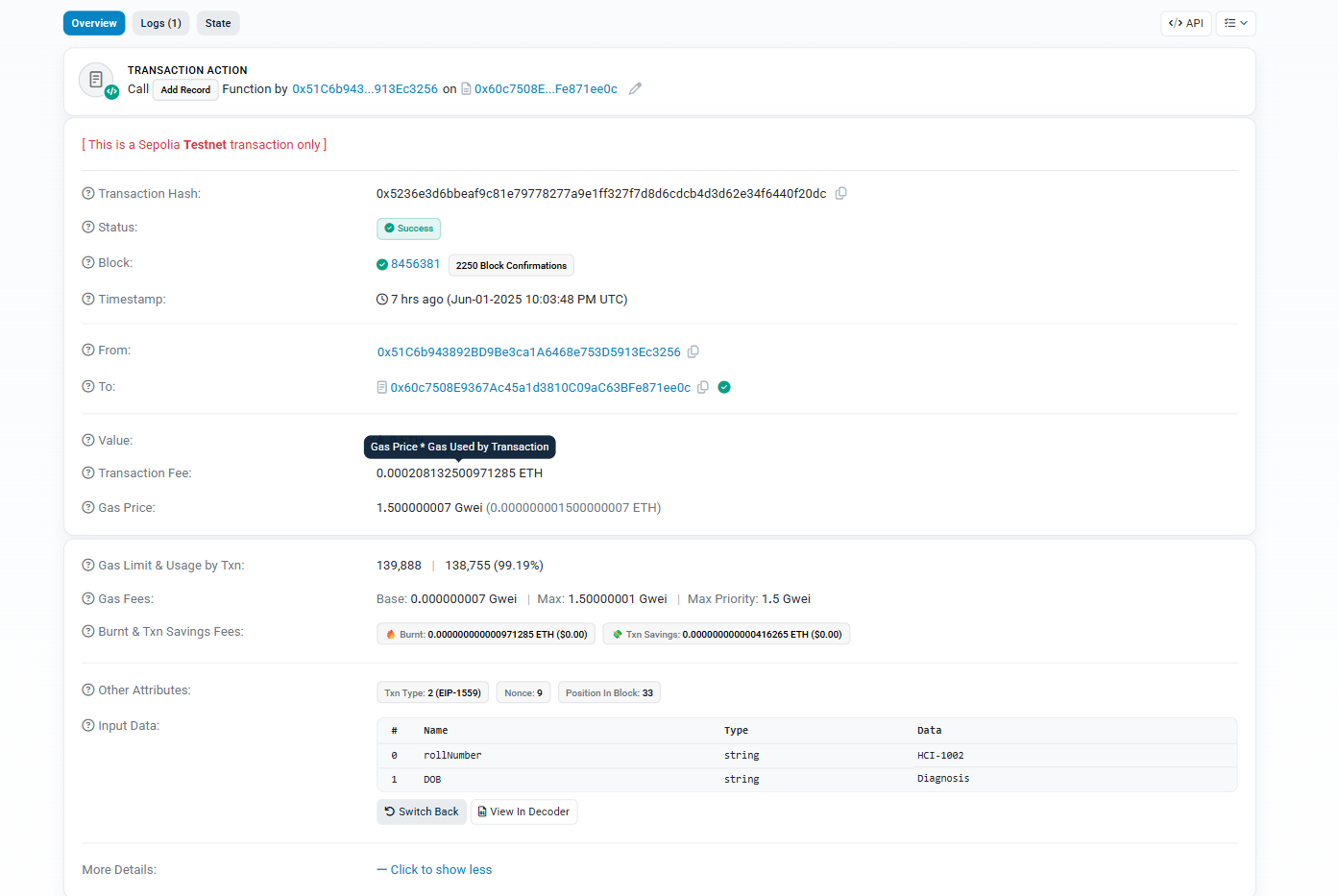
* The recordId and recordType are non-empty.
* A record with the same recordId does not already exist (timestamp == 0 ensures this).
* The sender becomes the initial owner of the record.

Upon successful addition, the RecordAdded event is emitted, and the totalRecords counter is incremented.

1. **Test Action**: From Account 1, called:  
    addRecord("HCI-1002", "Diagnosis")
2. **Expected Outcome**: The contract should store this record with Account 1 as its initial owner.  
   

**Figure 39:** Successful MetaMask transaction approval.

EtherScan:https://sepolia.etherscan.io/tx/0x5236e3d6bbeaf9c81e79778277a9e1ff327f7d8d6cdcb4d3d62e34f6440f20dc

  
 **Figure 40**: Etherscan view showing decoded input and event log

**Outcome**: Confirms the right use of transaction-based write operations on the blockchain. This transaction demonstrates how immutable records can be logged without the need for centralised approval systems, lowering latency and increasing auditability (Xu et al., 2021).

### **5.5 transferRecord() and Ownership Transfer Verification**

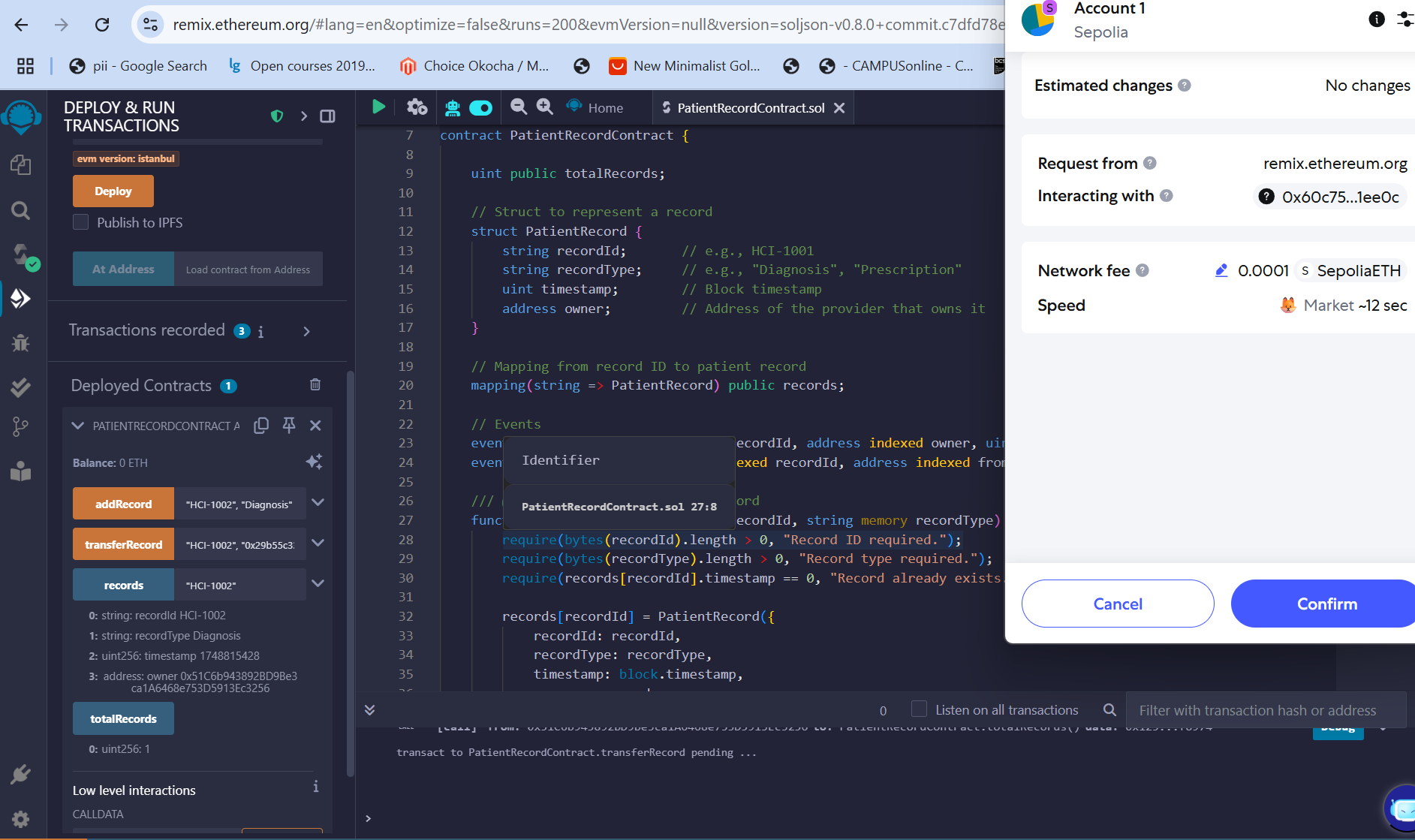
TransferRecord() allows the current owner of a record to transfer ownership to another Ethereum address. This is critical in real-world healthcare circumstances when medical records may need to be transferred from a general practitioner to a specialist.

Function safeguards include:

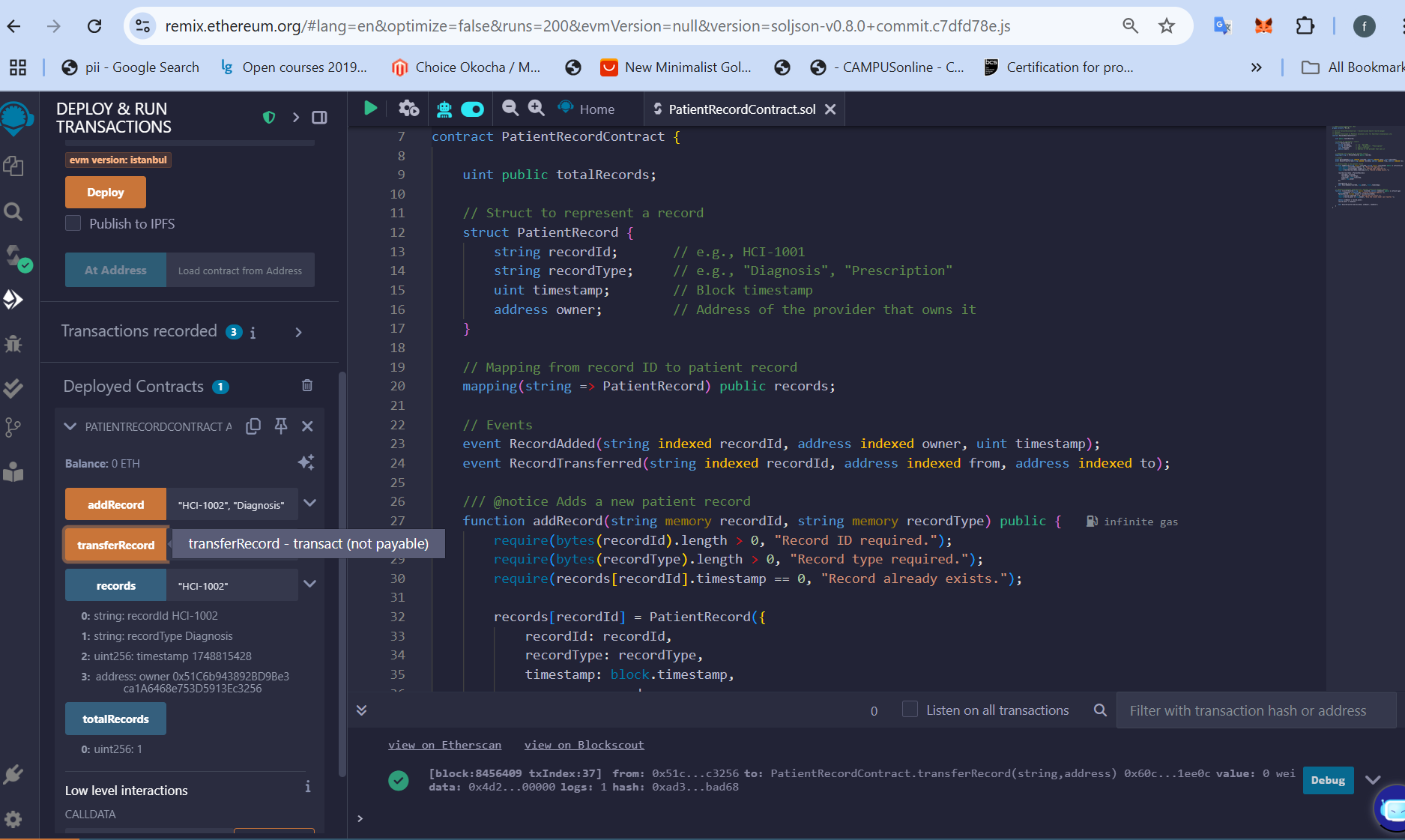
* Ensuring the recipient is a valid Ethereum address.
* Checking that the sender is the current owner of the record.
* Verifying that the record exists (timestamp != 0).

**Purpose:** Retrieves all record details by ID, including current owner, timestamp, and type.

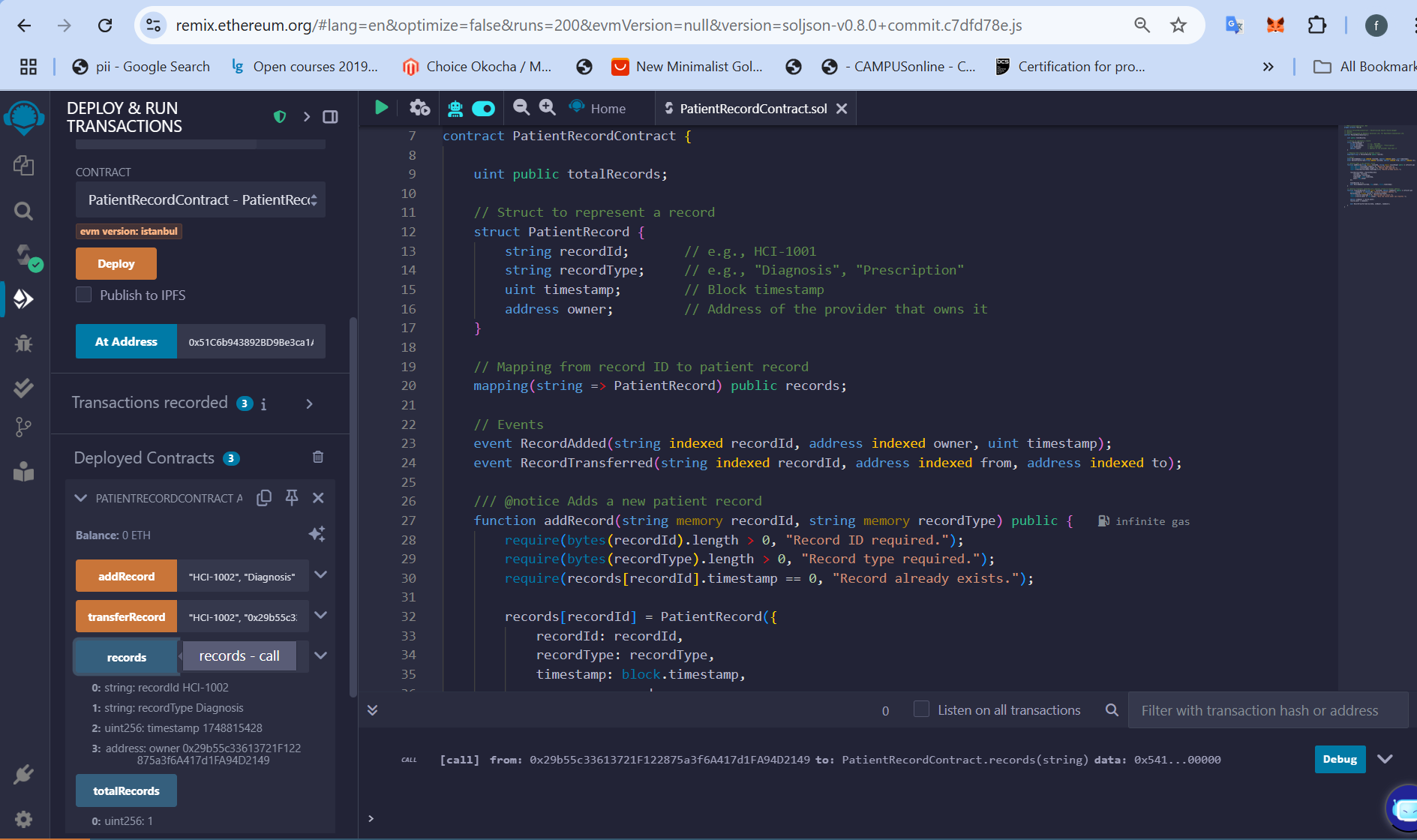
* **Test Action**: Called records("HCI-1002")  
  + Before transfer: returned owner = **Account 1**
  + After transfer: returned owner = **Account 2** (0x29b55c33613721F122875a3f6A417d1FA94D2149)



**Figure 41**: MetaMask confirmation for ownership transfer request.



**Figure 42**: Console output showing Account 1 as owner

  
 **Figure 43**: Updated output showing Account 2 as new owner

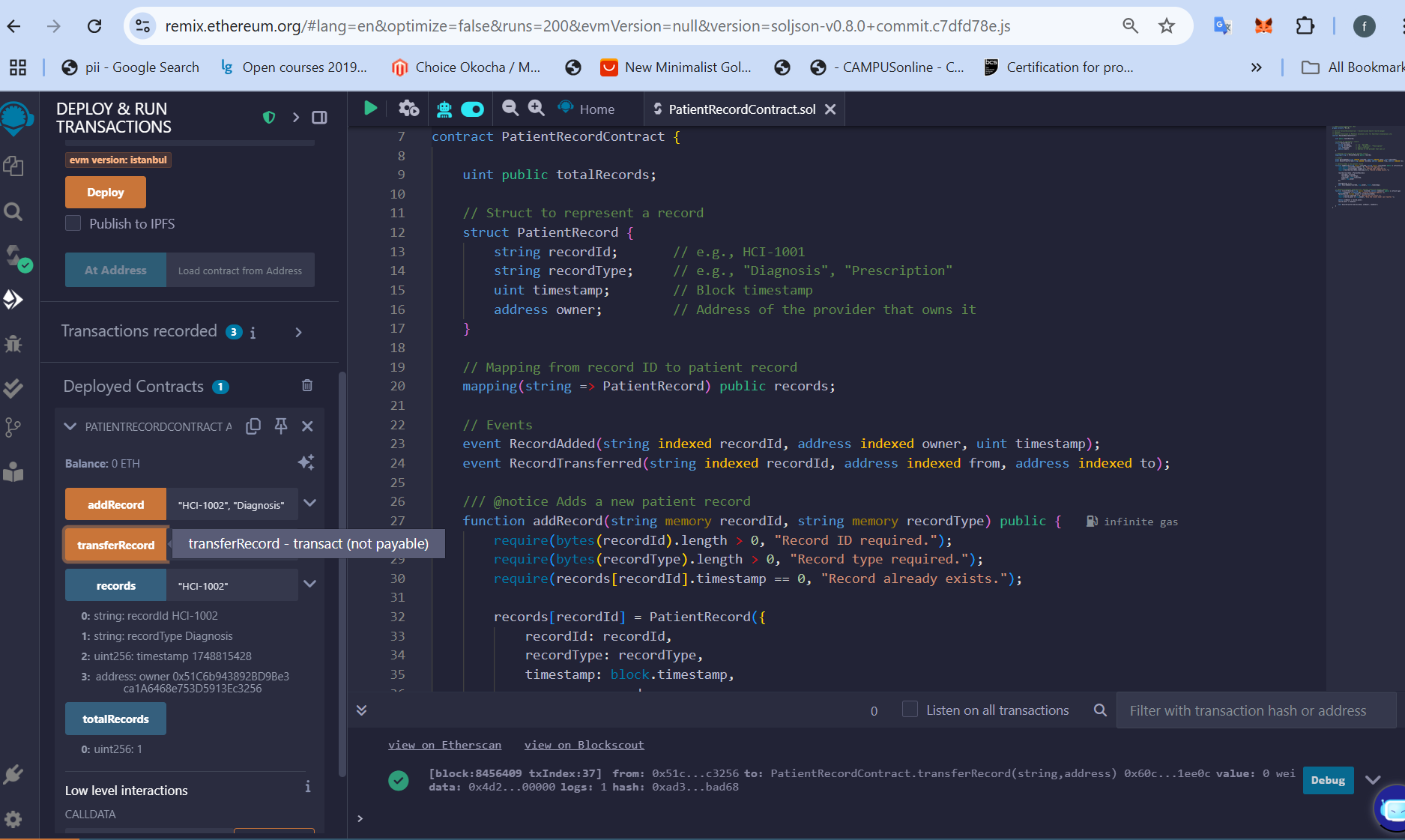
**Outcome**: Validates state reading and correct data persistence across accounts.

### **5.6 Function: totalRecords()**

The totalRecords() function is a public view function in the PatientRecordContract smart contract. It returns the total number of patient records that have been added to the system (Buterin, 2020; Wood, 2019).

**Purpose:** Returns the number of records stored in the smart contract.

* **Test Action**: Called after adding a single record
* **Expected Outcome**: Return value = 1



**Figure 44**: Screenshot of totalRecords() returning 1

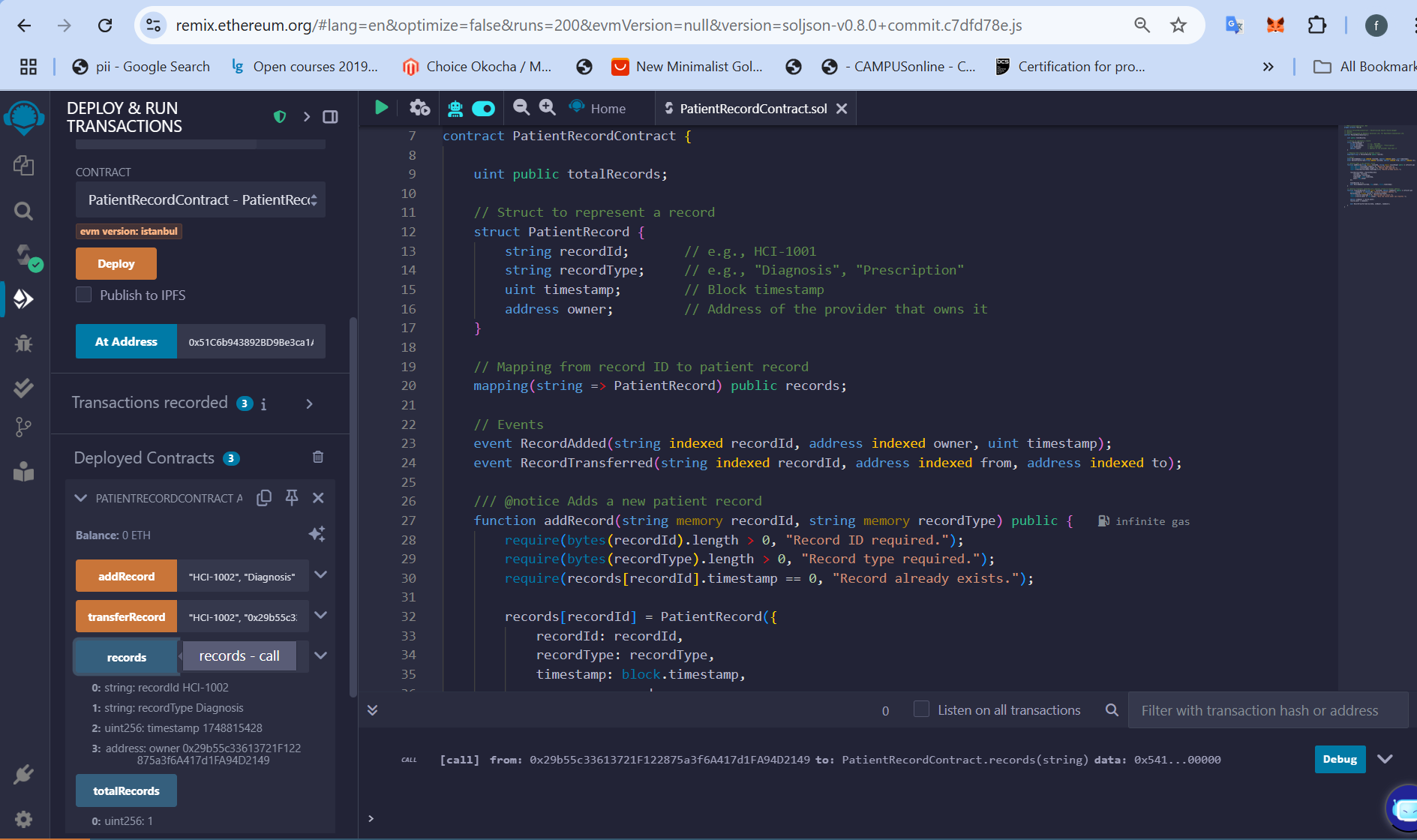
**Outcome**: Confirms correct increment logic for record tracking.

### **5.6.1 Function: transferRecord()**

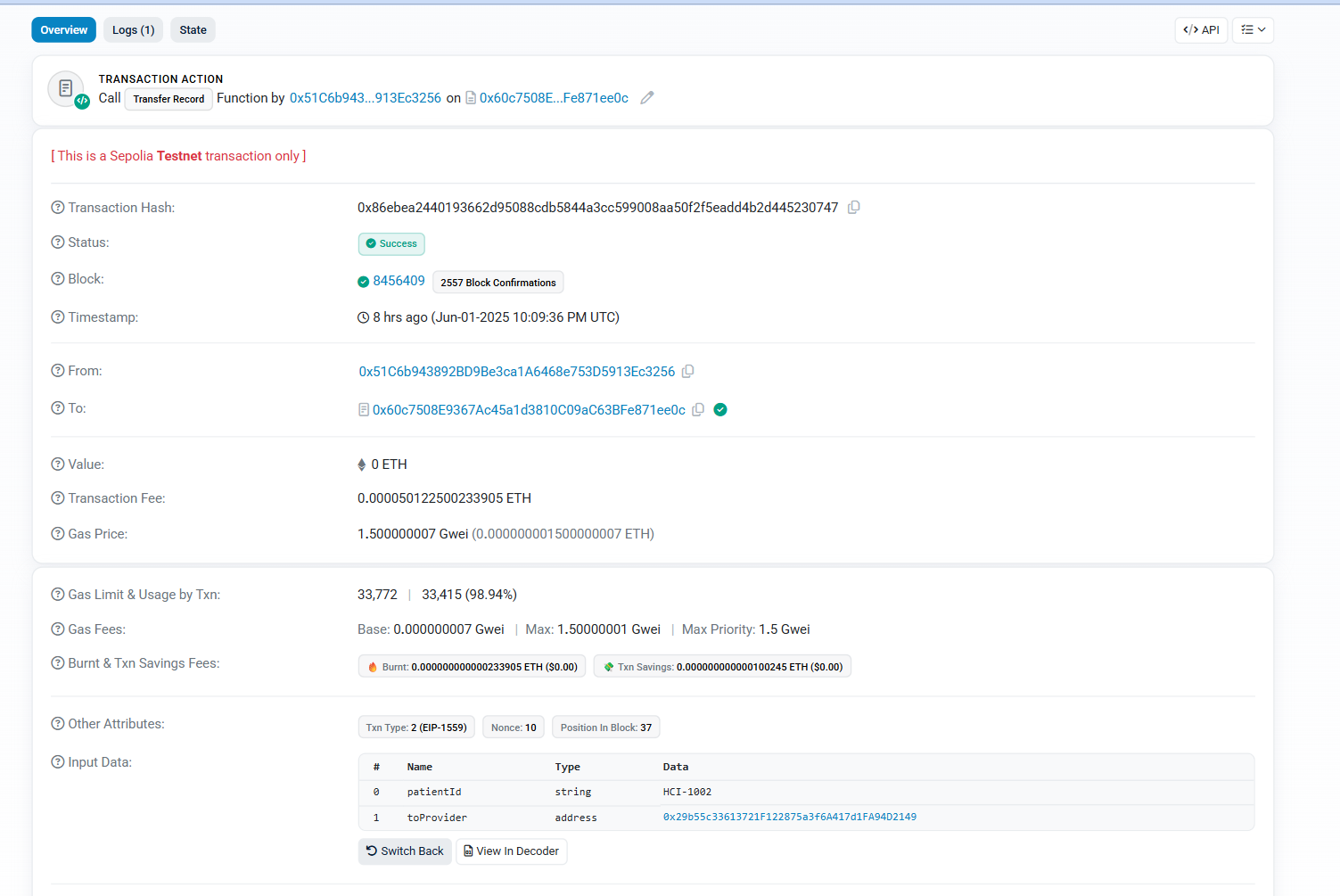
The transferRecord() function in the PatientRecordContract smart contract allows for the secure transfer of ownership of a single patient record from one healthcare provider to another. It assures that only the current record holder can initiate the transfer (Buterin, 2020; Wood, 2019).

**Purpose:** Allows a record owner to transfer ownership to another Ethereum address.

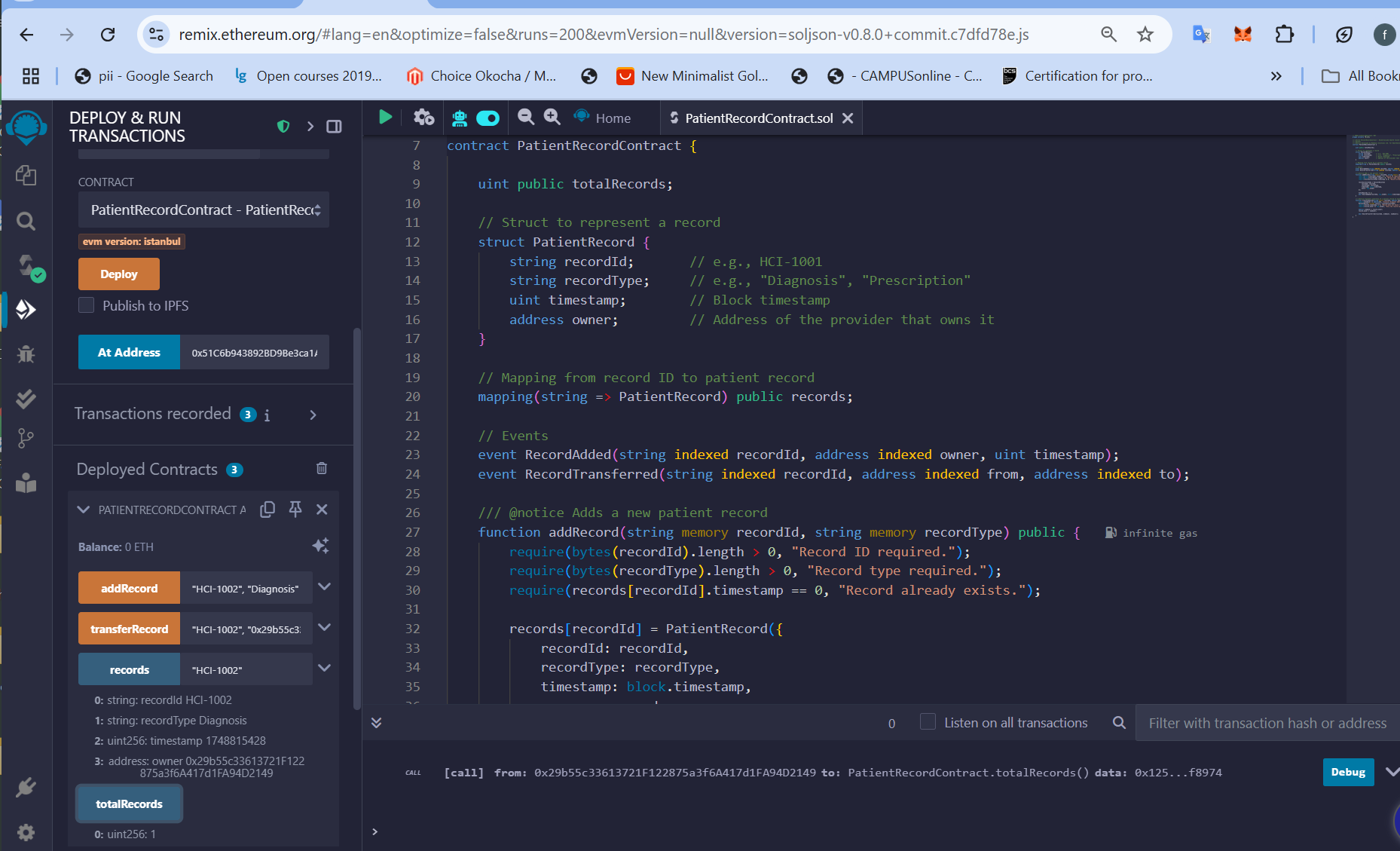
* **Test Action**:  
   From **Account 1**, called:  
   transferRecord("0x29b55c33613721F122875a3f6A417d1FA94D2149")
* **Expected Outcome**: Ownership changed successfully; confirmed by calling records() again.



**Figure 45**: MetaMask transfer transaction

**Figure 46:**EtherScan:https://sepolia.etherscan.io/tx/0x86ebea2440193662d95088cdb5844a3cc599008aa50f2f5eadd4b2d445230747

**Figure 51**: Etherscan verification of transfer parameters

  
**Figure 47**: records() showing Account 2 as new owner

**Outcome**: Demonstrates access control, correct use of msg.sender, and real-time ownership transfer.

## **5.7 Challenges and Resolutions**

The creation of the PatientRecordContract smart contract and a corresponding Python-based blockchain simulation faced a number of technical and contextual obstacles. This section describes the significant challenges found during implementation, how they affected the project's progress, and the methods used to solve them.

### **5.7.1. Remix IDE Deployment Errors**

A serious issue developed during the smart contract's initial deployment on the Sepolia testnet via the Remix IDE coupled to MetaMask. Deployment attempts can fail, resulting in misleading error messages like "invalid opcode" or MetaMask freezing during transaction confirmation.

Resolution: The problem was caused by unstable wallet-network synchronisation and insufficient Sepolia ETH. This was rectified by validating the testnet settings, re-authorizing MetaMask in Remix, and paying the deployer account via the Sepolia faucet. Best practices in decentralized app development emphasise the significance of validating the wallet-network handshake before contract interactions (Wang et al., 2022).

### **5.7.2. Incorrect Function Argument Formatting**

When using functions like addRecord("HCI-1002", "Diagnosis"), Remix originally produced type errors. This was especially difficult for transferRecord(), because inputs like addresses required to carefully adhere to the Solidity-defined address type syntax.

**Resolution**: Inputs in Remix were carefully manually formatted to ensure addresses were wrapped in double quotes and matched expected string types. Errors were deleted once the parameters were checked against the function signature. This experience is consistent with the work of Jiang et al. (2021), who observe that Solidity's tightly typed structure necessitates stringent user input management, particularly during external contact.

### **5.7.3. Ownership Verification Issues**

After successfully using transferRecord(), it was not immediately clear whether ownership had moved. In Remix, the records() function initially returned contract-like hashes or unreadable forms, causing misunderstanding about the genuine owner.

**Resolution:** The output was studied more thoroughly, indicating that the owner field had the right Ethereum address. Etherscan transaction records also helped with verification. As mentioned in security-focused Ethereum development research, on-chain verification using public explorers such as Etherscan is an important secondary confirmation (Chen et al., 2020).

### **5.7.4. Proof-of-Work Bottlenecks in Python Simulation**

During the Python blockchain simulation, the Proof-of-Work (PoW) algorithm created substantial delays. Mining blocks with a difficulty of 4 became computationally expensive on a standard system, especially after several block additions during testing.

**Resolution:** To reduce block creation time, a ceiling on mining attempts (max\_attempts=100000) was introduced. This strategy preserved the conceptual learning goal while retaining testability. This technique is consistent with pedagogical blockchain designs in which performance restrictions should not impede key teaching objectives (Narayanan et al., 2016).

### **5.7.5. Incomplete Documentation Tracking**

Another problem was organsing test result proof, such as Etherscan URLs and screenshots, across two MetaMask accounts. Some transactions were difficult to find because of overlapping timestamps or confusion between the deployer and recipient account views.

**Resolution:** An organised screenshot and link naming system was implemented, and each transaction was identified by its hash and confirmed in both account histories. This emphasises the necessity of auditable trails in decentralised systems, as underlined by De Angelis et al. (2018), particularly in healthcare-related smart contracts.

## **6.0 Limitations and Future Enhancements**

While this study exhibits blockchain's underlying potential for maintaining sensitive healthcare records, various constraints were observed that hinder its scalability and real-world applicability. This section identifies the most major constraints and presents a road map for future improvements.

### **6.1 Limited Access Control**

Currently, any Ethereum address can access the contract functions. Although the owner is tracked each record, the system lacks role-based access control, limiting function calls to only verified healthcare providers.

**Future Enhancement:** Implement require(isProvider[msg.sender]) logic and a provider whitelist mapping to enforce only registered providers can add or transfer records. This mirrors best practice in healthcare blockchain systems, which must integrate decentralised identity management (Yue et al., 2016).

### **6.2 No Data Encryption or Off-Chain Storage**

All patient record data, including record type and timestamp, is recorded in plaintext on the blockchain. This presents data privacy risks, especially under compliance frameworks such as GDPR and HIPAA.

**Future Enhancement:** Integrate off-chain storage (e.g. IPFS) and store only record hashes or pointers on-chain. Encryption and access tokens could then be handled externally. Such hybrid storage designs are common in blockchain-based EHR systems (Azaria et al., 2016).

### **6.3 Gas Cost and Scalability Constraints**

The current design would scale poorly in a production environment. Each addRecord() or transferRecord() operation incurs a gas cost, and as record volume grows, user cost and network congestion will increase.

**Future Enhancement:** Adopt a layer-2 solution like Optimism or zk-Rollups to reduce fees, or consider permissioned blockchains (e.g. Hyperledger Fabric) for more controlled environments (Androulaki et al., 2018).

### **6.4 Basic Blockchain Simulation (Python)**

The Python blockchain designed for conceptual understanding lacks network broadcasting, real mining, and distributed node functionality. It is completely local and deterministic.

**Future Enhancement:** Upgrade the Python blockchain to support peer-to-peer networking, or replace it entirely with a private Ethereum testnet (e.g. using Ganache or Geth) to allow full decentralisation features. This would better simulate real blockchain properties such as consensus, latency, and node conflicts (Xie et al., 2019).

**6.5 No User Interface (UI)**

All interactions are currently carried out via the Remix IDE or Python terminal. This reduces accessibility and usability for non-technical healthcare providers.

**Future Enhancement:** Build a front-end UI using React and Web3.js to allow seamless interaction with the deployed smart contract. This will improve usability and reflect real-world healthcare record systems, where end-users require visual interfaces to manage patient data securely (Zheng et al., 2020).

### **Appendix A: Fund Transfer to Account 1**

This appendix describes the first transaction in which Account 1 received testnet funds on the Ethereum Sepolia network. The transaction was recorded on the blockchain and may be checked independently via the given Etherscan link.

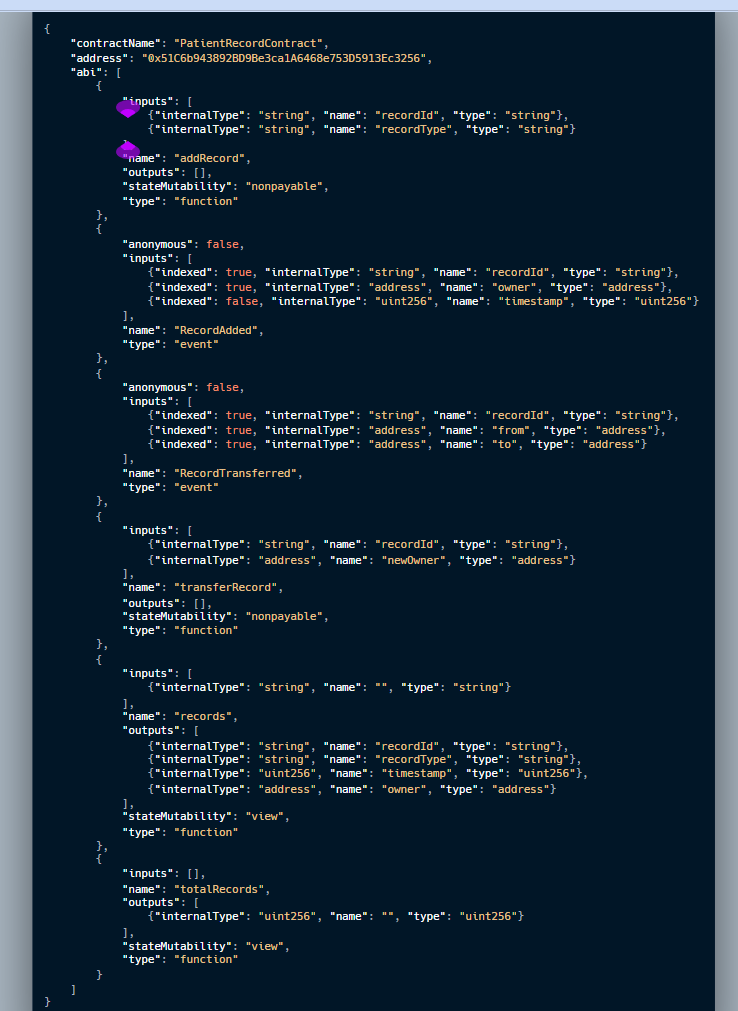
* **Transaction Description:** Fund transfer to Account 1 (Sepolia testnet)
* **Transaction Hash: 0x0af4bf78980612914214d32ebfb451e634c3bb228b95b270c17f3d5c04f74e4b**
* **Explorer Links:** [**https://sepolia.etherscan.io/tx/0x0af4bf78980612914214d32ebfb451e634c3bb228b95b270c17f3d5c04f74e4b**](https://sepolia.etherscan.io/tx/0x0af4bf78980612914214d32ebfb451e634c3bb228b95b270c17f3d5c04f74e4b)
* **Transaction hash and contract address**

**https://sepolia.etherscan.io/tx/0xa3997d729a6d887a8981cc5d18cd0f17e5eb312542cfbc073f63698f6577c39d**

### **Appendix B: Smart Contract Metadata and ABI**

This appendix contains the whole ABI (Application Binary Interface) and metadata for the deployed PatientRecordContract. The ABI specifies the methods and events provided by the contract and is required for front-end integration and blockchain interaction.

* **Contract Name:** PatientRecordContract
* **Deployed Address:** 0x51C6b943892BD9Be3ca1A6468e753D5913Ec3256
* **Deployment Network:** Ethereum Sepolia Testnet
* **Metadata File:** PatientRecordContract.json (saved using Python)

**ABI and Metadata (JSON Format)**  


### **References**

**Androulaki, E., Barger, A., Bortnikov, V. et al.,** 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. *Proceedings of the Thirteenth EuroSys Conference*, pp.1–15. Available at:<https://doi.org/10.1145/3190508.3190538> [Accessed 2 June 2025].

**Antonopoulos, A.M. and Wood, G.,** 2019. *Mastering Ethereum: Building Smart Contracts and DApps*. O’Reilly Media.

**Azaria, A., Ekblaw, A., Vieira, T. and Lippman, A.,** 2016. MedRec: Using Blockchain for Medical Data Access and Permission Management. *2016 2nd International Conference on Open and Big Data (OBD)*, IEEE, pp.25–30. Available at:<https://doi.org/10.1109/OBD.2016.11> [Accessed 2 June 2025].

**Buterin, V.,** 2020. Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform. [online] Available at:<https://ethereum.org/en/whitepaper/> [Accessed 2 June 2025].

**Carbon,** 2025. *Carbon – Create and Share Beautiful Images of Your Source Code*. [online] Available at:<https://carbon.now.sh/> [Accessed 2 June 2025].

**Chen, T., Li, X., Luo, X. and Zhang, X.,** 2020. Under-Optimized Smart Contracts Devour Your Money. *24th International Conference on Financial Cryptography and Data Security*, pp.442–460. Available at: <https://www4.comp.polyu.edu.hk/~csxluo/Gasper.pdf> [Accessed 2 June 2025].

**De Angelis, S., Aniello, L., Lombardi, F., Margheri, A. and Sassone, V.,** 2018. PBFT vs Proof-of-Authority: Applying the CAP Theorem to Permissioned Blockchain. *Italian Conference on Cyber Security (ITASEC)*, pp.1–11.

**Engelhardt, M.A.,** 2017. Hitching Healthcare to the Chain: An Introduction to Blockchain Technology in the Healthcare Sector. *Technology Innovation Management Review*, 7(10), pp.22–34. Available at:<https://doi.org/10.22215/timreview/1111> [Accessed 2 June 2025].

**Etherscan,** 2024. Etherscan Blockchain Explorer. [online] Available at: [https://etherscan.io](https://etherscan.io/) [Accessed 2 June 2025].

**Esmaeilzadeh, P.,** 2019. The Role of Privacy Concerns in Blockchain Adoption for Healthcare. *Journal of Medical Systems*, 43(9), pp.1–9. [Accessed 2 June 2025].

**Fan, K., Jiang, W., Li, H. and Yang, Y.,** 2018. Blockchain-based Privacy-Preserving Medical Data Sharing. *Future Generation Computer Systems*, 94, pp.574–582. Available at:<https://doi.org/10.1016/j.future.2018.11.046> [Accessed 2 June 2025].

**Gervais, A., Karame, G.O., Wüst, K., Glykantzis, V., Ritzdorf, H. and Capkun, S.,** 2016. On the Security and Performance of Proof of Work Blockchains. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp.3–16. Available at:<https://doi.org/10.1145/2976749.2978341> [Accessed 2 June 2025].

**Jiang, Z., Wu, H., Wang, J. and Liu, D.,** 2021. Smart Contract Vulnerabilities: Vulnerable Does Not Mean Exploited. *Blockchain: Research and Applications*, 2(3), pp.100–118. Available at:<https://doi.org/10.1016/j.bcra.2021.100015> [Accessed 2 June 2025].

**Kuo, T.T., Kim, H.E. and Ohno-Machado, L.,** 2017. Blockchain Distributed Ledger Technologies for Biomedical and Health Care Applications. *Journal of the American Medical Informatics Association*, 24(6), pp.1211–1220. Available at:<https://doi.org/10.1093/jamia/ocx068> [Accessed 2 June 2025].

**Li, X., Jiang, P., Chen, T., Luo, X. and Wen, Q.,** 2020. A Survey on the Security of Blockchain Systems. *Future Generation Computer Systems*, 107, pp.841–853. Available at:<https://doi.org/10.1016/j.future.2017.08.020> [Accessed 2 June 2025].

**MetaMask,** 2024. MetaMask: A Crypto Wallet & Gateway to Blockchain Apps. [online] Available at: [https://metamask.io](https://metamask.io/) [Accessed 2 June 2025].

**Mettler, M.,** 2016. Blockchain Technology in Healthcare: The Revolution Starts Here. *2016 IEEE 18th International Conference on e-Health Networking, Applications and Services (Healthcom)*, pp.1–3. Available at:<https://doi.org/10.1109/HealthCom.2016.7749510> [Accessed 2 June 2025].

**Nakamoto, S.,** 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. [online] Available at:<https://bitcoin.org/bitcoin.pdf> [Accessed 2 June 2025].

**Narayanan, A., Bonneau, J., Felten, E., Miller, A. and Goldfeder, S.,** 2016. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton: Princeton University Press.

**Patel, V.,** 2019. A Framework for Secure and Decentralized Sharing of Medical Imaging Data via Blockchain Consensus. *Health Informatics Journal*, 25(4), pp.1398–1411. Available at:<https://doi.org/10.1177/1460458218769699> [Accessed 2 June 2025].

**PK910,** 2025. *Sepolia Faucet*. [online] Available at:<https://sepolia-faucet.pk910.de/> [Accessed 2 June 2025].

**Remix,** 2024. Remix - Ethereum IDE.[online] Available at: [https://remix.ethereum.org](https://remix.ethereum.org/) [Accessed 2 June 2025].

**Reyna, A., Martín, C., Chen, J., Soler, E. and Díaz, M.,** 2018. On Blockchain and Its Integration with IoT: Challenges and Opportunities. *Future Generation Computer Systems*, 88, pp.173–190. Available at:<https://doi.org/10.1016/j.future.2018.05.046> [Accessed 2 June 2025].

**Szabo, N.,** 1997. Formalizing and Securing Relationships on Public Networks. *First Monday*, 2(9). Available at:<https://doi.org/10.5210/fm.v2i9.548> [Accessed 2 June 2025].

**Wang, Y., Han, J. and Xie, B.,** 2022. Development Tools and Environment Setup for Ethereum Smart Contracts. *IEEE Transactions on Software Engineering*, 48(7), pp.2149–2165.

**Xia, Q., Sifah, E.B., Asamoah, K.O., Gao, J., Du, X. and Guizani, M.,** 2017. MeDShare: Trust-less Medical Data Sharing Among Cloud Service Providers via Blockchain. *IEEE Access*, 5, pp.14757–14767.

**Xie, J., Tang, Q. and Zhu, H.,** 2019. A Survey of Consensus Protocols and Their Applications in Blockchain. *IEEE Transactions on Emerging Topics in Computing*, 8(4), pp.1390–1403.

**Yli-Huumo, J., Ko, D., Choi, S., Park, S. and Smolander, K.,** 2016. Where is Current Research on Blockchain Technology?—A Systematic Review. *PloS ONE*, 11(10), p.e0163477. Available at:<https://doi.org/10.1371/journal.pone.0163477> [Accessed 2 June 2025].

**yWorks,** 2025. *yEd Live – Diagram Editor*. [online] Available at:<https://www.yworks.com/yed-live/> [Accessed 2 June 2025].