Ampere Blockchain Model Whitepaper

Ryen Krusinga

September 22, 2023

Contents

1 Introduction

We describe a blockchain-based medical app capable of combining patient data from different institutions and formats into a coherent, decentralized whole. Our format allow for natural AI-assisted verification of data consistency and provides a secure, institution-independent way of representing personal medical information.

2 Components Overview

The app consists of several interconnected services running on cloud servers, whitelisted institutional computer networks, and whitelisted private devices. These components include:

- **Blockchain core software:** A code library that allows authorized app instances to read from and/or write to a patient's blockchain.
- **Data/AI layer software:** Code that formats patient data for use on the blockchain and checks data consistency with assistance from AI.
- **Patient database software:** Code that fingerprints patients and maps them to their corresponding blockchain.
- **FHIR/EHR bridging software:** A service that interfaces with EHR software from different institutions in order to obtain patient data and verify it.
- **Cloud service bridging software:** Software components that interface with cloud computing providers such as Microsoft Azure and Amazon Web Services. These components abstract away cloud services so that core app components can run on different providers. These cloud services include:
  - Serverless coding environments like Azure Functions and AWS Lambda
  - Scalable, replicated databases
  - API calls to AI models and other verification code.

- **Key and version management software:** A service that manages whitelisted app instances, distributes and revokes cryptographic keys, and distributes code updates to nodes running the app.
- **Frontend interface software:** A user-friendly desktop app for interacting with the blockchain service.
- **Logging, error reporting, and analytics software:** In order to constantly improve the app, we will need to extract usage analytics, data logs, and error reports from nodes implementing our technology.

In the future, as the app grows in complexity, other services may be needed, such as a load-balancing service, a shard management service like Kubernetes, and so forth.

3 Blockchain Design

Our blockchain model consists of a header, a subheader, and a data frame. The data frame has a fractal structure, in the sense that each data frame can contain other data frames, containing other data frames, and so on, each frame at every level implementing its own hash function, linearization function, constructor, and so forth. Not much nesting is likely to be needed, but this makes the model quite extensible. All frames should begin with their type and version number, as this determines which code should be used to handle the contents.

Elements of the design shall include:

- Block headers (not included in the overall block hash) consisting of:
  - The overall block hash
  - Sufficient digital signatures of the hash from the providing node and all verifying nodes
  - Other metadata as needed
- Block subheaders (included in the overall hash) consisting of:
  - Block format version number
  - Block size

  - Timestamp the block was added
  - Block manifest: a list of data sub-blocks contained inside, including their type and size.
  - The hash of the previous block in the central chain
  - A list of hashes of other blocks and code entities in (something like) prefix notation, describing the computation (if any) that was done to generate the data in the current block (e.g., calling a specific AI model).
  - Other metadata as needed

- The data section, consisting of concatenated data blocks containing:
  - The data block type
  - The version number of the given type
  - The timestamp (or timestamps) of when the data was physically collected, or first recorded (this could be very different than the timestamp when the block was added to the chain).
  - The size of the data segment
  - The actual data

The entire blockchain can be stored in a binary file consisting of a metadata segment followed by a list of all the blocks in sequence. For any given patient, this file ought not be much longer than a few dozen kilobytes, or perhaps a megabyte or so at most, depending on the type of data inside.

Each patient, doctor, institution, and other app-related entity will have its own blockchain. Since these blockchains are privately maintained by whitelisted provider nodes, hyper-efficiency of the blockchain protocol is not paramount. This gives us room to allow the block protocol to have the complex structure (as described above) necessary to implement the desired functionality.

3.1 Block Algorithm, Data Validation Protocol, and AI Integration

The block algorithm shall be approximately as follows:

1. **Patient encounter:** A whitelisted institution sees a patient. They make an EHR entry.

2. **Consent:** Through the FHIR system or an EHR-specific extension, the blockchain software sees that a patient has a new entry. If the patient has given consent to use the new technology, then the software can proceed to add it to the blockchain.

3. **Intake:** A data entry block is created, linked to the latest block in the central chain. It stores the EHR note as-is to the blockchain, with no modification yet. It can be in any format.

4. **Further Processing:** A second, processed block is created and added to the blockchain after the unprocessed intake block. This processed block reformats the data to our own specification, possibly using the assistance of AI.

5. **Consistency checking:** The information in the processed block is checked against the history of the blockchain, possibly with the help of AI, in order to determine if there are any errors. This step is the most complex. Possibly more blocks are generated as a result, each depending on the blocks computed before.

6. **Updating the central chain:** A new central summary node is created that computes an up-to-date overall patient status based on the new information.

3.2 Philosophy Behind the Design

The flexible data formats, versioning system, and multi-block intake procedure all combine to make this blockchain maximally flexible in its ability to represent patients. It can accomodate patient information of any type in essentially any format. New formats and new data types can be added in the blockchain's future, along with better and better versions of the verification protocol. Furthermore, by storing intermediate computations as their own blocks, and by using the prefix-notation hash list to describe how each block was produced, the blockchain achieves full transparency of how it obtained, reformatted, and modified its data.

To step back even further: what does it mean, at the widest level, to store data *about* something out in the world? This may seem like a trivial

question, but it is not. Digital data is nothing but bits in computer memory. It does not have intrinsic meaning. Meaning derives from (1) how it is used in the computer program that handles it, and (2) how humans interpret the output of that computer program. This creates a problem of *drift*, which is when data becomes obsolete either because (a) the format has become useless as time progresses (due, perhaps, to newer computer technologies or professional practices), or (b) the literal meaning of the data has shifted (e.g., old medical categories are discovered to correspond to entirely different bodily processes than previously believed or agreed upon; social interpretations of gender versus sex is one example). Adding to these problems is the fact that one can never have *all* the data about a person; that would require an atomic-resolution movie of their entire life since birth. There is always more data to collect; a practical line must be drawn somewhere, and that line can shift.

By designing the blockchain in the way described above, we can address all these problems by storing not just a rigid set of information fields mapped to values, as in current EHR systems, but multiple views to the same information in flexible, changeable formats, all the while maintaining computational transparency and drawing upon the flexibility of the latest AI technology as a proxy for human intuition.

4 Security Design and Key Distribution Model

The goals of our security model include:

- HIPAA compliance: patient blockchain data should only exist in an unencrypted state on HIPAA-compliant servers.
- Whitelisting: only designated institutions should be able to read and write to the blockchain.
- Blockchain validation: new data added to the blockchain must be cryptographically signed by the institution that provided the data. Insofar as possible, new blocks must be validated and signed off on by a selection of other trusted blockchain nodes.
- Patient control over data: there should be a secure means for patients to read the contents of their blockchain.

- Robustness to attack: it should be exceedingly difficult or impossible for attackers to make unauthorized modifications to the blockchain or to read protected patient information.

- Key revokability: there should be a decentralized protocol in place to revoke any previously whitelisted keys that may have been compromised.

To achieve these ends, several pieces of core functionality may be implemented:

- **Total blockchain encryption:** A patient's blockchain information shall be stored in a totally encrypted state using standard symmetric encryption such as AES. The encryption keys themselves shall be stored in a separate encrypted locker file (see multi-key authentication below).

- **Block validation procedures:** Before signing a new proposed block, whitelisted nodes should independently verify the integrity of the block, insofar as is possible. When blocks are partially AI-generated, this becomes tricky. New AI-handling protocols will need to be developed, such as, for example, cryptographic "receipts" from the providers of the AI model in question.

- **Multi-factor and multi-key authentication:** In addition to using standard multifactor authentication techniques, such as mobile texts or emailed codes, patient data can be further protected by "multi-key" authentication. By making the encryption keys to a locker file (see above) depend on the hash of multiple concatenated keys held by boths patients and providers, patients may use the web portals of trusted institutions to view their blockchain info. While all decryption should be done remotely on trusted, HIPAA-compliant servers, this model nevertheless allows patients, if they wish, to download their encrypted personal blockchain and then view it through their desired web portal by uploading it and entering their personal key. The advantages of this are: (1) if the patient's personal computer and personal key are both compromised by an identity thief, the data still cannot be accessed unless the thief also has access to the patient's web portal account, including all the means of multifactor authentication necessary to login to the portal; (2) whitelisted medical institutions can still read and

write to the patient's blockchain without knowing the patient's personal key via multiple locker files.

- **Everything on the blockchain:** There should be blockchains not just for patients, but also for doctors, institutions, code versions, and all other entities related to the app. The collection of whitelisted public keys should itself exist on the blockchain in a decentralized way, allowing modifications to this list to also be made in a decentralized manner. (This is similar to PGP. However, it is unclear at this time what role, if any, a centralized set of whitelisted keys belonging to our organization should play in the blockchain. Our company is the authority that allows institutions to use our blockchain app technology, after all, so perhaps instead we should retain the sole ability to add or revoke new nodes. This is a matter for further research.) There should also probably be a global blockchain for mapping patient identities to their corresponding blockchain file.

5 Data/AI Layer Design

5.1 Data frames

Every new type and version of a data frame must have a corresponding description entry in the codebase specifying its purpose and its data format. Functions capable of handling the specific type and version will be registered in a global data handling table.

Broadly, each type of data frame should implement at least the following functions:

- A hash function that maps a given data frame instance to a deterministic cryptographic hash.
- Constructors, possibly using AI, capable of taking data from various other formats and mapping them into the current format.
- Version casting constructors capable of mapping data in the current frame to past versions of the data format. Only necessary if backwards version compatibility is needed.
- At least two linearization/serialization functions: one for human readability, one for efficiently inputting the data into an AI.

5.2 AI interface for consistency checking and recommendation

Broadly speaking, the consistency checking algorithm for a given blockchain would be as follows:

1. Find all past blocks containing the type of data being checked (or the most recent summary block on the central chain).
2. Use the linearization function of each relevant data frame to extract a sequence of AI-readable inputs.
3. Provide the AI with a prompt asking it to perform the given check, along with the linearized data, and give an answer in a particular format.
4. Put the AI's response, if applicable, in a new data frame using a constructor function designed for this scenario.

This can be adapted for making recommendations, performing logical inferences using classical reasoning techniques, and other potential tasks.

6 Patient Database Design

Patients must be mapped to their corresponding blockchain reliably, so that duplicate chains are avoided, as well as the problem of mistakenly attaching one patient's information to another's blockchain. This requires establishing a patient's identity. Identity establishment should be robust to errors and (among other things) slight variations in name spelling (such as, for example, including a middle name or not), and should handle the rare cases where there is more than one person with the same name and birthday.

Within a single EHR system, this is done via a universally unique identifier (UUID) attached to each patient, along with the doctor asking clarifying questions about identity.

The problem is that different EHR systems may use entirely different UUIDs for the same patient. Thus, for a decentralized app, identity must be established based on attributes alone. Therefore, patient data must be "fingerprinted" in a way that reliably establishes identity even in the presence of formatting differences, form errors, and so on. This is not likely to be a major concern, but it is something we will need to deal with.

7 Cloud and FHIR/EHR Bridging

This refers to background code that interacts with cloud and medical APIs, providing a layer of abstraction between the core app functionality and the world. Designs are based on company-specific API protocols. This code should strive to be as minimal and out-of-the-way as possible, relying heavily on pre-existing libraries.

8 Frontend Interface

We are currently using an electron/react paradigm, with the added benefit that the app can easily be run remotely through a browser.

The interface should allow the viewing of a patient's blockchain, possibly making recommendations or drawing attention to errors.

9 Logging, error reporting, and analytics

This type of code is integrated throughout all the other code modules. A central service may be created to handle such information and send it to the developers.

10 Minimal beta version goals and component set

The app described above is a complex multifaceted piece of software. Developing a beta version requires paring down functionality to the absolute minimum necessary to demonstrate the promise of the technology.

The demonstration goals of beta are:

1. Show the ability to create a blockchain based on pre-existing patient data accessed through an FHIR interface, stored in multiple formats.

2. Show the ability for multiple app instances to add blocks to the chain in a secure, decentralized manner.

3. Show the ability to use AI to check errors in patient data and generate summary blocks.

4. View the blockchain on a portable web app.

10.1 Necessary functional modules

To achieve these goals, the following modules must be implemented in some fashion:

1. **Minimal serverless wrapping code:** Code that wraps app functionality for all services listed here, allowing services to run in a serverless fashion through Azure Functions.

2. **Basic FHIR service:** Must be able to obtain all past FHIR entries for a given patient, as well as detect new entries as they arise. Must be able to connect to actual FHIR test servers. Must provide an API interface to the rest of the app abstracting away the details of FHIR calls.

3. **Basic patient database service:** Must be able to create and connect to secure databases, both local and remote. Must be able to store and retrieve blockchain data based on a patient's fingerprint. Must be able to cache data and automatically keep itself up-to-date if remote changes are made.

4. **Basic AI service:** Must be able to call GPT-4 in a HIPAA-compliant way through Microsoft Azure. Must provide an API interface to the rest of the app for making such calls.

5. **Basic resource management service:** Must keep track of API URLs for all app and service instances. Must allow app instances to locate themselves in the wider ecosystem. Must manage digital certificates and connection security. Must manage running processes and restart them if necessary. Must keep track of whitelisted keys. Must track code versions on a basic level. Must provide an API interface for adding new whitelisted app instances.

6. **Blockchain core functionality:** Must be able to add blocks, verify blocks, and linearize blockchains for AI use. Must have a precise fault-tolerant consensus protocol.

7. **Data Frame core functionality:** A minimal set of data frames must be specified and implememented as described in the data layer design section. Their format information must be stored as data. They must be sufficient to store most types of relevant patient data.

8. **Logging ability:** Components must log their actions for the purposes of rapid debugging.

9. **Minimal frontend interface:** Must be able to load the blockchain for particular patients, visualize the blockchain information in some way, and perform actions on the blockchain such as human-initiated AI consistency checking.

10.2 Implementation details

All modules shall be implemented in Python with the exception of the frontend interface, which uses a stack of Typescript/Javascript/Node.js/React/Electron.

Modules shall be implemented in a way that takes maximum advantage of the serverless paradigm. Modules shall be conceived of as (stateless?) event-driven functions that interact with other resources such as databases. The internal logic shall only be dependent on internal APIs, abstracting away execution environment details. Cloud interfacing services are code wrappers for instances of the internal functions.

Individual functions shall be as short and self-contained as possible (partially to allow them to fit within chatGPT's context window for development). They shall follow general style guidelines. They shall be tested where possible (requiring a basic mocking system around them). They shall rely on standard libraries where possible.

10.3 Predicted difficulty and implementation scheduling

The core set of functional modules are 4, 5, 6, and 7. Of those, 5, 6, and 7 are likely to be the largest and most technically involved modules at first. However, with careful abstraction, their logic can be mostly internal, allowing for ease of development by reducing the reliance on outside libraries and execution paradigms.

The least immediately important modules are 8 and 9, so those can be saved for last (though 8 can somewhat be done as I go along).

1, 2, 3, and 4 all interface with external APIs and libraries, and thus have the largest learning curve even if they aren't the biggest modules. They also are prerequisites for the core functionality of the app, and may guide some design decisions about core functionality. Therefore, I plan to work on these first. I have already implemented a test version of 4 to call chatGPT's API, so that code can be adapted. That leaves 1, 2, and 3 as the first I should implement. Since getting FHIR data is the most important step, I will work on 2 first, then 1, then 3.

10.4 Minimal Module Design Objectives

The design objectives for the first three modules to be implemented:

- **FHIR module (2):** Provide a minimal set of functions for (a) obtaining a list of patients from FHIR, (b) obtaining the information and complete medical histories of said patients, and (c) automatically finding new patient entries as they arise. Must be demonstrated on actual FHIR test servers.

- **Serverless module (1):** Must provide a wrapper for the core functions of the app, such as the database module, AI module, and blockchain module, that allows the code to work in a serverless framework (in this case, Azure Functions). This involves connecting to Azure resources like database services and AI services. Must demonstrate ability to run general functions in a serverless manner, retrieve and store data from Azure resources, and make API calls to other Azure functions and AI services.

- **Database module (3):** Internal API interface must be machine-agnostic. The module must provide a database abstraction that can switch automatically between local and remote databases. It must keep any local databases up-to-date. It must implement some patient identity mapping function. Must demonstrate ability to store mappings between patient identity and large patient files (eventually, the blockchain files).