A Reference Architecture for Blockchain-based Resource-intensive Computations managed by Smart Contracts

Thesis · October 2018

CITATIONS
2

READS
9,083

1 author:

Anastasios Kalogeropoulos
Technische Universität München
4 PUBLICATIONS **2** CITATIONS

SEE PROFILE

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

A Reference Architecture for Blockchain-based Resource-intensive Computations managed by Smart Contracts

Anastasios Kalogeropoulos

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

A Reference Architecture for Blockchain-based Resource-intensive Computations managed by Smart Contracts

Eine Referenzarchitektur für blockchain-basierte ressourcen-intensive Berechnungen die durch Smart Contracts verwaltet werden

Author:	Anastasios Kalogeropoulos
Supervisor:	Prof. Dr. Helmut Krcmar
Advisor:	M.Sc. Dian Balta (fortiss)
Submission Date:	June 15, 2018

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, June, 11, 2018 Anastasios Kalogeropoulos

Acknowledgements

I would like to express my deepest gratitude to my advisor, M.Sc. Dian Balta for his constant advice, guidance and support of my thesis at the chair of Information Systems, Department of Informatics, Technical University of Munich. I would also like to thank Prof. Dr. Helmut Krcmar for supporting this research project.

A special thanks to my colleague Saurabh Narayan Singh for our interesting discussions during this project and for sharing with me all his knowledge regarding the blockchain technology. Furthermore, I thank my friend Giorgio Tabarani for his encouragement and support.

I would also like to thank my brother and my parents for their continuous support over these years.

Last but not least, I would like to thank one of the most important persons in my life for her love and understanding.

Abstract

Blockchain technology has received great attention recently. It enables data to be exchanged between different contracting parties that act anonymously within a network and serves as a promising alternative to the current organisational and technical infrastructure. Bitcoin, the first and the most popular blockchain-based application, identified that the potential of this technology can be used in many more application areas beyond financial services. Blockchain transactions are administered by smart contracts, software programs that are executed automatically as soon as a given condition is met. They offer the possibility of creating a new form of agreement between parties without the need for intermediaries. The growing complexity of blockchain-based applications faces many challenges such as scalability and security. Therefore, to overcome these issues, infrastructures are needed in order to support processing of resource-intensive computations and manage private data using the blockchain technology.

In this thesis, we address the limitations of systems that need to perform resource-intensive computations and handle private data on the blockchain. To identify the requirements of such systems we conduct a stakeholder analysis and we design a reference architecture that coordinates different components to delegate the execution of computations in a secure off-chain mechanism. We develop a prototype that utilizes different technologies in combination with a system of smart contracts that orchestrate the data and software acquisition, enforce the execution of the software over the given data into a blockchain-unrelated component and make the computation output available to the recipient. To evaluate the designed architecture and the developed prototype, we describe a use case scenario that predicts chronic kidney diseases based on patient symptoms and we show that the proposed system is promising and satisfies the identified requirements.

Contents

List of Figures

List of Tables

List of Abbreviations

API Application Programming Interface.

Blob Binary Large OBject.

DAG Directed Acyclic Graph.

DAO Decentralized Autonomous Organization.

Dapp Decentralized Application.

DHT Distributed Hash Table.

DSRM Design Science Research Methodology.

EVM Ethereum Virtual Machine.

FHE Fully Homomorphic Encryption.

FOSS Free and Open-Source Software.

GNT Golem Network Tokens.

HTML Hypertext Markup Language.

HTTP HyperText Transfer Protocol.

IaaS Infrastructure as a Service.

IOT Internet Of Things.

IP Internet Protocol.

IPFS InterPlanetary File System.

ML Machine Learning.

MVC Model View Controller.

PaaS Platform as a Service.

PoC Proof Of Concept.

PoCo Proof-of-Contribution.

PV Photovoltaic.

RLC Runs On Lots of Computers.

RPC Remote Procedure Call.

SHA-256 Secure Hashing Algorithm-256.

UML Unified Modeling Language.

VM Virtual Machine.

1 Introduction

1.1 Motivation

Blockchain technology has been used in cryptocurrencies for several years but there are many more application areas for this disruptive technology [16]. Bitcoin [40] is the first and largest digital currency but the blockchain capabilities extend far beyond that [14], enabling existing applications to be improved and new to emerge.

Blockchain is a shared database composed of blocks which contain transactions executed among parties in a network. Once information is stored on the blockchain, it can never be erased, therefore it automates auditing and makes applications transparent and secure. Due to its characteristics, it has the potential to create a competitive market in both financial and non-financial areas providing lower costs and improved security in a large network where transactions of untrusted participants take place and they can be verified at any time without the need of a trusted central authority [58]. All the transactions are regulated by smart contracts, which are computer programs executed on a blockchain and transfer any kind of asset. Examples include projects that support on-chain data storage and computations using smart contracts, a widely used one is the Ethereum platform [22]. Thus, blockchain applications are getting more complex and go beyond currency transactions towards an innovation that requires an infrastructure for storing data and performing computations similar to cloud services platforms.

In order to tackle the requirements of the blockchain applications and the smart contracts, one has to face several challenges from an economic and technical perspective. Computations on-chain are resource-intensive [1] and often described as inefficient [27]. For example, in case of the Ethereum platform, smart contract executions might be restricted if they consume too much computing power. These limitations prevent the scalability of systems and therefore the processing of complex algorithms like machine learning or data analysis on-chain. Apart from the computational issues, there are privacy and confidentiality concerns one has to worry about. All actions performed in a smart contract are recorded on the blockchain and everyone in the network has access

[1] A set of related processes that require significant system resources or time, or requires exclusive access to large amounts of data.

to the data that is stored on it. As a result, applications that require privacy preserving approaches to handle private data, for instance healthcare systems, are currently limited to private networks.

To counter the challenge of processing resource-intensive computations, two approaches exist: on-chain and off-chain. The on-chain approach offers decentralized computing and provides a system that limits the processing of large amounts of computations and data. Regarding off-chain computations, there is no systematic approach, although rudimentary techniques are available that include processing on Virtual Machines (VMs) or Docker containers and charging for the service on-chain. Therefore, if would be of great interest for academia and practice systems that perform resource-intensive computations off-chain using the blockchain technology.

1.2 Purpose and Research Questions

The goal of this thesis is to design a reference architecture for systems that delegate the execution of resource-intensive computations into a secure off-chain mechanism using smart contracts, Free and Open-Source Software (FOSS) and public repositories. The requirements and the architecture of the system will be evaluated by a PoC for a chronic kidney disease prediction scenario but the solution can be applied to many use cases. This thesis was based on the following three research questions:

- **Research question 1: Which are the requirements of the system in order to perform resource-intensive computations off-chain?**

  To address this question, related work will be studied based on the following initial premises. First, smart contracts should orchestrate the data acquisition and enforce the execution of the FOSS on the given data within the container. Second, the computation should take place only if pre-defined conditions are met (e.g. the data and/or the software can be found on the given address so that they can be accessed from the container). Third, the output of the computation should be made available to the recipient directly by the smart contract.

  The expected outcome of this process will be the key requirements that the system should fulfil in order to execute resource-intensive computations off-chain.

- **Research question 2: What is a reference architecture of a blockchain-based system for resource-intensive computations?**

  The reference architecture [45] will be based on the requirements of the system and it will be designed in a way that adapts to its technical and operational needs

as well as economic constraints. The entities, models and functions of the system will be described and it will define their functionality and the relationships among them. It will also focus on how these elements interact with each other, e.g. how the off-chain container can access the data and the FOSS.

The expected outcome will be a reference architecture of a system that allows for off-chain resource-intensive computations steered and administered on-chain.

- **Research question 3: How can the architecture of the system be implemented in a prototype?**

  To answer this question, a functional prototype will be developed using the blockchain technology and it will be tested using real data and FOSS.

  The expected outcome of this work will be a prototype that implements the developed architecture.

1.3 Outline

The rest of this work is organised as follows. Chapter 2 presents the theoretical background and the basic concepts about the blockchain technology, cryptography and distributed systems as well as related work and chapter 3 discusses the application area of this work and the methodology we followed. Chapter 4 presents the use case scenario and the requirements of the system, while chapter 5 discusses the design of the system from four different views. At the end, chapter 6 presents the tools and technologies used to implement the prototype as well as the evaluation of the system based on the designed architecture and the developed PoC, while chapter 7 presents the conclusions and addresses the limitations and the future work.

2 Theoretical Background and Related work

This chapter presents the blockchain technology and the landscape of its applications. The basic concepts of cryptography and distributed systems that are necessary to understand the blockchain technology are discussed and related work is reviewed.

2.1 Blockchain Overview

Blockchain was first introduced as the underlying technology of Bitcoin [40] and its purpose was to provide a solution to the "double-spending" problem [21]. It can be described as a peer-to-peer distributed ledger that is cryptographically secure, append-only, immutable and can be updated only via consensus among peers. The ledger stores records of transactions between non-trusted anonymous participants in a network, referred to as nodes. Rather than having a central administrator, the ledger is replicated across the entire network and once a transaction occurs every copy is updated, eliminating the need of a third party to perform the task of validating the transaction.

Transactions that occurred are grouped into blocks which are linked to form a chain and they are secured using cryptography. A block contains a timestamp, transaction data and a hash of the previous block, thus creating a chain of blocks linked with each other as shown in Figure 2.1.

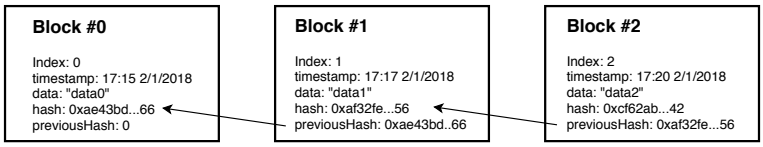

Figure 2.1: Structure of a generic blockchain.

In order to add new blocks to the ledger, the transactions need to be validated by the nodes within the network. Miners, which are members in the network with a lot of computing power, compete to validate the transactions by solving complex problems. The first miner to solve the problems and validate the block receives a reward. In case of the Bitcoin, a miner would receive Bitcoins. Protocols, such as proof-of-work [40] and proof-of-stake [5], are used to achieve distributed consensus, validate new blocks and add them to the chain.

Due to its characteristics, the blockchain technology can be applied in many use cases and it can be used to build the underlying infrastructure of many systems. The landscape of the blockchain applications can be described with three major tiers [49], currencies, smart contracts and DAOs, and it is discussed in the following sections.

2.1.1 Currencies - Bitcoin

The first tier was introduced with the invention of Bitcoin and describes applications that require financial transactions to be executed on the blockchain. All the alternative currencies that exist today belong to this category, for instance the Litecoin [35], which is a variation of Bitcoin with a few improved features.

Current digital economy depends on certain trusted authorities. Online transactions rely on third-party entities for the security and the privacy of the exchanged assets. However, these entities can be hacked or compromised and this is where the blockchain technology can be used in order to eliminate intermediaries without compromising the privacy of the transaction details and the parties involved in it. Bitcoin paved the way for creating financial transactions over the internet in which users can receive funds immediately in their digital wallet and send funds to others using a private key.

2.1.2 Smart contracts and Ethereum

The emerging need for more complex blockchain applications introduced the concept of smart contracts and allowed the blockchain technology to be applied beyond financial transactions. While the first tier is for decentralization of currencies and payments, the second tier relates to the transfer of any kind of assets beyond currencies using the blockchain.

Smart contracts are described by Nick Szabo [50] as computer programs that reside on the blockchain and contain business logic and data. A widely used project that uses smart contracts is the Ethereum Project. Ethereum allows developers to program their own smart contracts, create applications that run on the blockchain and exchange any kind of data between nodes in a network [9].

2.1.3 DAOs

The third tier relates to decentralized autonomous entities governed by smart contracts that run on top of the blockchain, known as DAOs. DAO's financial transaction records and rules are maintained on the blockchain and instead of human interacting in person and controlling assets using paper contracts, smart contracts are used to govern the organizations. Ethereum blockchain led the way with the introduction of DAOs and a project of such an organization was "The DAO", although it did not manage to succeed. [47].

2.2 Cryptography Basics

Cryptography [11] is a set of techniques for secure communication between participants in a network by hiding private information in order to prevent third parties from reading private messages. It is one of the core aspects of the blockchain technology. Its role is to contribute to the security of the network by making it hard to manipulate data and secure its users by enabling transparent transactions while maintaining the participants private.

Blockchain technology utilizes public-key cryptography [24], also known as asymmetric cryptography, in the encryption and decryption of sensitive data as well as in digital signatures in order to prove a message's authenticity. Users generate a key pair consisting of a public and a private key. The public key represents the user's account and is used to encrypt messages while the private key should be kept secret and it is used to decrypt a message encrypted with the corresponding public key. An example of asymmetric encryption is shown in Figure 2.2.

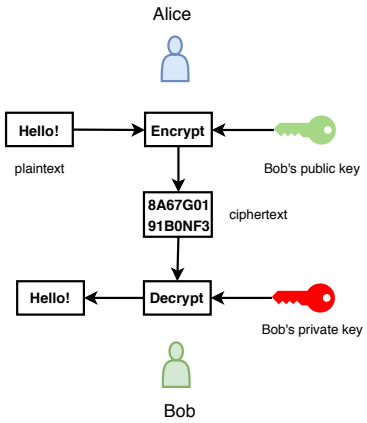

Figure 2.2: Asymmetric encryption example.

Bob sends his public key to Alice and keeps the private key secret. To send Bob a message, Alice encrypts it using his public key and sends him the ciphertext. Then, Bob uses his private key to decrypt the message.

Public-key cryptography works also the other way around. Any message encrypted with a private key can only be decrypted with the corresponding public key. This process is used to create digital signatures [29]. Digital signatures help protect against fake or spoofed messages. In blockchain, the private key is used to sign transactions, while the public key is to receive cryptocurrencies. Every transaction that is executed on the blockchain is digitally signed by the sender using their private key. A simplified process of digitally signing and verifying a transaction on blockchain is illustrated in Figure 2.3.

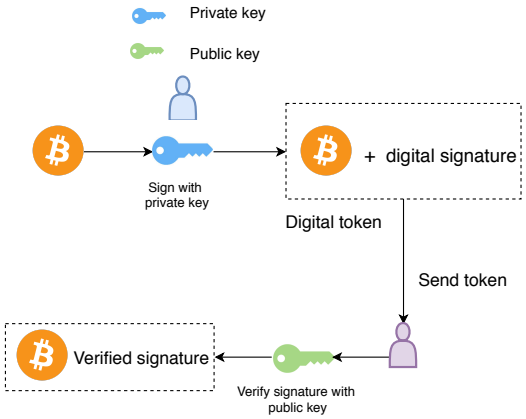

Figure 2.3: Simplified digitally signed transaction on blockchain.

Another key component of blockchain technology used to determine the integrity of the data is hashing [59], a cryptographic method for transforming data of an arbitrary size to data of a fixed size, referred to as hash. It is impossible to find out what the original data is from a hash and its output is deterministic, meaning that the same input data will always produce the same hash. Blockchain technology exploits hashing to ensure immutability of the transaction records. The hash is used to agree between all parties in the network that the current state of the blockchain is the same to everyone. An example of a popular cryptographic hash function which is also used by Bitcoin in the process of mining is the Secure Hashing Algorithm-256 (SHA-256).

2.3 Distributed Systems

Blockchain at its core is a decentralized distributed system, therefore understanding distributed systems is essential in order to understand the blockchain technology.

The evolution of computer systems has been changed a lot since 1945 where the era of modern computer began. At that time, computers were large and expensive and they were operated independently from one another. Since then, many technological advances changed this situation and nowadays it is feasible and easy to put computer systems together composed of many computers in a network to form a distributed system. While there is no single definition of what a distributed system is, according to the book of Tannenbaum and Van Steen (2007) [51], it is defined as:

> "A distributed system is a collection of independent computers that appears to its users as a single coherent system."

Distributed systems can be described as autonomous computational entities with their own memory, called nodes, spread across different geographies and communicating with messages. A diagram of a distributed system with three nodes is illustrated in Figure 2.4.

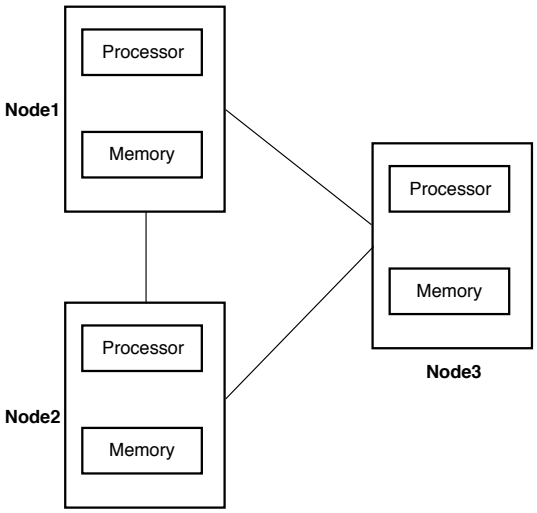

Figure 2.4: Block diagram of a distributed system.

An important feature of a distributed system is the notion of fault tolerance [12]. Nodes in the system may fail and the data stored in these nodes might be lost. There are two types of failures a node may undergo, a crash failure or a Byzantine failure [34]. A crash failure occurs when a node stops working abruptly. Byzantine failures

occur as a result of malicious actions of an adversary. The design goal of a distributed system is to automatically recover from these failures without affecting the overall functionality of the system.

Replication plays an important role in distributed systems in order to increase reliability and improve their performance. Various replication techniques can be applied in order to maintain resources at different nodes, avoid a single point-of-failure and scale the system [28]. However, the problem of having multiple copies may lead to consistency problems, hence consistency techniques must be carried out to ensure these copies are the same [6].

To achieve availability in distributed systems in the presence of failures, protocols that enable the system as a whole to continue function properly are needed [42]. These protocols require cooperation among processes and a fundamental problem of this task is the problem of agreeing on data values during computations. To reach a consensus, a simple approach can be followed. Processes can vote and agree on the majority value. In the absence of faults, this solution works fine but in case of faulty processes the result of the agreement can be influenced by their votes. A famous algorithm used to solve consensus in a network of unreliable processes is Paxos [33].

2.4 Related Work

The emergence of the blockchain technology and the increasing demand of its applications requires for smart contracts to become more complex and go beyond currency transactions and therefore there is a need for infrastructure that stores data and performs resource-intensive computations.

A mapping of the various infrastructure layers of the blockchain technology that provide storage, processing and communication in a decentralized computing world is shown in Figure 2.5.

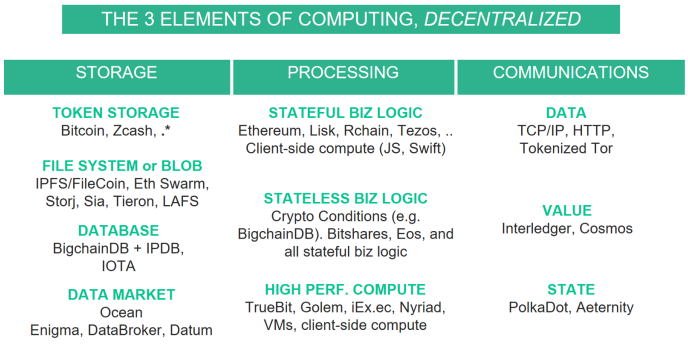

Figure 2.5: Blockchain infrastructure landscape. Image taken from [7].

The rest of this section presents projects that offer infrastructure for data storage and computations using the blockchain technology.

2.4.1 Ethereum

The Ethereum platform [22] is a blockchain system that supports the development of Dapps [44] using smart contracts that are executed directly on-chain using the Ethereum Virtual Machine (EVM). The currency of exchange in Ethereum is called Ether. Users can execute different types of transactions, from transfer of funds between accounts to execution of smart contracts. There are two types of accounts, externally owned and contract accounts [9]. Externally owned accounts are controlled by private keys and they are used to perform transactions, either transferring funds or triggering smart contract code. Contract accounts have associated code that is triggered by transactions or calls received from other contracts, as shown in Figure 2.6.

Figure 2.6: Ethereum accounts.

Ethereum offers a built-in Turing-complete programming language designed for developing smart contracts that run on the EVM, called Solidity [15]. A contract written in Solidity is compiled into EVM bytecode and then deployed at a specific contract address. Operations in contracts are executed by every node in the network and each operation costs a certain amount of a unit called gas [60], which can be purchased in exchange of Ether. Hence, to protect the nodes from executing infinite loops, there is a mechanism to limit the resources used by the smart contracts, referred to as gas limit [60]. Transactions have a gas limit value which specifies how much gas they can consume. Since smart contracts run by any node on a network, they are required to be deterministic in order to achieve the same result. If the result differs between some nodes, consensus in the network cannot be reached and the transaction will fail.

2.4.2 BigchainDB

The growth of applications that are using the blockchain technology requires for an infrastructure with high throughput of transactions, low latency and storage capacity. A blockchain database that is designed to offer scalability to these applications is BigchainDB [36]. It offers high throughput, low latency, decentralized control, immutable storage and transfer of assets among others.

BigchainDB is built, at the time of writing, on top of an existing distributed database, namely RethinkDB [57]. As Figure 2.7 illustrates, internal communication in the network is handled by RethinkDB. Clients control the BigchainDB nodes, each one wraps a RethinkDB node. The BigchainDB node provides an Application Programming Interface (API) to the users in order to make transactions and register, issue, create or transfer assets.

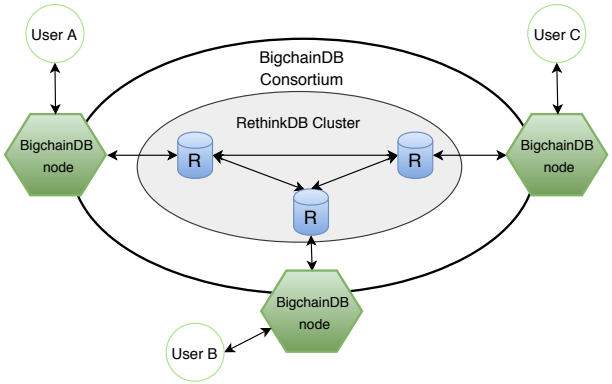

Figure 2.7: BigchainDB architecture.

2.4.3 IPFS

IPFS [4] is a decentralized peer-to-peer file-sharing application. It combines different technologies like Distributed Hash Table (DHT) [19] and Merkle [37] Directed Acyclic Graph (DAG) data structures and uses a protocol similar to BitTorrent to decide how to store and manage the data around the network. It also supports file versioning by using data structures similar to Git. It does not require for every node to store all the data that has been published in the network. Instead, users can choose which data

they want to persist.

Nowadays, the Internet is based on HTTP which relies on location addressing and uses Internet Protocol (IP) addresses to identify the specific server that hosts the requested data. Despite the fact that HTTP is the backbone of the World Wide Web, it has some flaws that limit its performance and IPFS was designed to solve them [30].

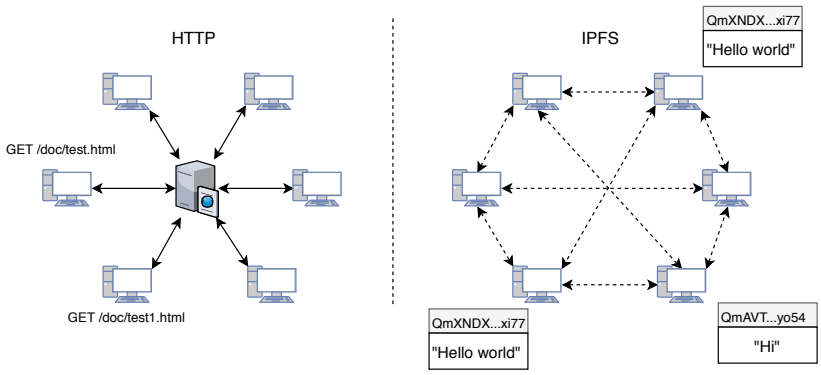

Figure 2.8: HTTP vs. IPFS

As shown in Figure 2.8, IPFS distributes files across the network and each file is addressed by its content. When stored in the system, they are assigned with a cryptographic hash based on their content, as opposed to traditional location addressing used by HTTP where a single server hosts many files and information has to be fetched by accessing this server.

2.4.4 Other related projects

The Golem Project [54] offers a decentralized supercomputer where people provide their hardware in return for Golem Network Tokens (GNT). It allows for developers to securely distribute and monetize their software and for individual users to rent resources of other user's machines in order to complete resource-intensive computations. It can be considered both Infrastructure as a Service (IaaS) as well as Platform as a Service (PaaS).

Another project that provides a blockchain-based decentralized cloud for executing tasks is the iExec [31]. It offers a marketplace service that allows anyone to offer their computing resources. Applications and developers pay a fee in iExec's internal

cryptocurrency, namely Runs On Lots of Computers (RLC), to access these resources. The iExec team develops a Proof-of-Contribution (PoCo) protocol that allows off-chain consensus and certifies providers for the usage of their resources on the blockchain.

An application which is trying to solve the scaling issues of the Ethereum blockchain [13] is Truebit [52]. It outsources the verification of computations into participants in a market of off-chain computers rather than every node computing every smart contract. These participants are called *solvers* and they submit a solution to the problem for a reward. In case of a dispute with a solver's result, the problem will be executed on-chain and the miners will settle it.

Different development platforms come with different benefits. As McConaghy states in his article [7], there is no "one technology that fits all". The aforementioned projects belong to the category of high performance compute, as shown in Figure 2.5. However, there are simpler rudimentary techniques for processing on VMs or off-chain containers and storing the result into Binary Large OBjects (Blobs) on IPFS or BigchainDB, but no systematic approach available.

3 Research Approach

This chapter presents the application area of this thesis followed by the methodology that was used to collect the necessary information in order to identify the requirements of the system, design a reference architecture and implement a prototype.

3.1 Context

This thesis exploits the blockchain technology and focuses on delegating resource-intensive computations into a secure off-chain container by keeping the data and the computed result private. The application area of this work is discussed below.

- **Predicting energy consumption of households**

  Due to the evolution of the Internet Of Things (IOT) [1], many households tend to use smart devices with Internet connection to control many of the house operations, like lighting, heating and appliances. Regarding electricity, smart meters can be installed in the house in order to track the consumption and collect information about the energy usage of the devices. Several Machine Learning (ML) approaches have been implemented to produce accurate energy consumption forecasts for households [8]. However, this process requires confidentiality of the data collected from the smart meters and hardware that can execute computationally intensive ML algorithms. The system proposed in this thesis could be used by landlords to estimate the energy use of their house and apply energy-saving solutions by uploading their smart meter's data confidentially into the system and delegate the prediction of the energy use into a trusted container executing appropriate software.

- **Chronic kidney disease prediction using symptoms**

  In healthcare, medical records contain private information about patients that allow health care providers to evaluate the patient's health and wellness. These records can be used in combination with ML algorithms to predict future diseases. For instance, Milandeep Arora et al (2016) [46] presented an algorithm to detect chronic kidney diseases for patients based on their symptoms. However, this approach requires for the patient records to be shared with experts who own and control software that predicts chronic kidney diseases. The system developed in this thesis could be used by health care providers or patients to detect chronic kidney diseases by executing ML algorithms over their medical records in a secure container and receive the predictions about the status of their health in a privacy preserving approach.

- **Training ML models**

  Several cases require training a ML algorithm in order to create a ML model. For instance, a software company that builds spam-detecting software for emails wants to train a ML model that predicts whether an email is spam or not. The more complex the case is, the more data is required as well as sufficient computer resources are needed to create the model. For creating a highly-accurate model, high quality data is needed. As discussed in [53], collecting and preparing the data is an important and difficult step in the training process and computer resources are expensive. Thus, data providers do not want to offer their preprocessed data for free and publicly accessible to everyone. The system proposed in this thesis could be used to host datasets cryptographically secure as well as containers that can train models using these datasets with a fee. The software company could then access the system to rent datasets and containers in order to train a ML model and receive the artifact.

- **Forecasting the power generation of photovolatic cells using ML**

  Several methods exist to convert energy into electrical energy, although many of the sources are non-renewable and exist in finite amount. A type of energy that is renewable and abundant is solar energy and can be exploit by Photovoltaic (PV) modules. As discussed in [2], forecasting PV generation becomes vital and the process of generating ML models that predict the PV generation at fortiss GmbH is presented in the preceding work. However, this process requires for different types of weather data to be used and collecting this data might not be an easy task. The system proposed in this work could be used by fortiss GmbH to rent available datasets and containers for a small fee in order to perform the training of the ML models and receive the artifacts.

Although the preceding applications belong to different domains, they all have to face the same challenges. As of today, they are limited to private networks since they need systems with privacy preserving approaches to share data with several parties for different purposes and use computer resources to perform resource-intensive computations.

3.2 Design Science Research Methodology

In this thesis we followed a DSRM [43] to identify the problem, design a reference architecture and develop a prototype of a system that allows for off-chain resource-intensive computations administered on-chain. The activities undertaken as part of this methodology are summarized graphically in Figure 3.1 and they are described in detail in the rest of this section.

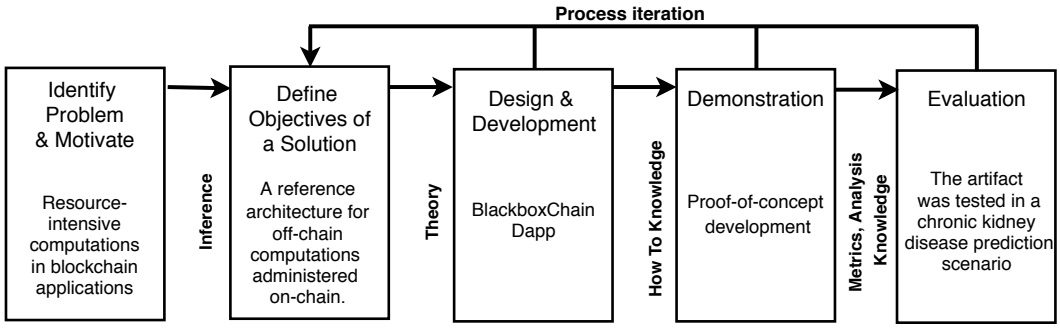

Figure 3.1: DSRM process model.

3.2.1 Problem Identification and motivation

The requirements of blockchain applications are growing, thus computations on-chain are considered resource-intensive and sometimes inefficient. Further, the Ethereum platform prevents the scalability of applications because of security reasons and transaction costs. Therefore, the processing of algorithms like ML or data analysis is limited. Additionally, confidentiality of transactions and privacy of the data are major issues, since actions taken in a smart contract and data recorded on the blockchain are publicly available to anyone in the network.

3.2.2 Define the objectives for a solution

Our objective was to design a reference architecture for a system that performs off-chain resource-intensive computations steered and administered on-chain. The major challenges included the smart contract design in order to orchestrate acquisition of the data, enforce the execution of the software on the given data within an off-chain container and make the output available to the recipient. The system was required to provide lower execution costs by delegating resource-intensive computations into a blockchain-unrelated component, strong privacy of the transaction details and restricted access to the computed result.

3.2.3 Design and Development

The artifact was designed by gathering the user requirements first and it was adapted to the system's technical and operational needs as well as economic constraints. The design includes the entities, models and functions of the system as well as their functionality and the relationships among them. It also focuses on the interaction between these elements, for example how the off-chain container can access the data and the software.

3.2.4 Demonstration

The implemented artifact is a Dapp running on the Ethereum blockchain. It provides an interface for users to upload dataset, software and container resources on the blockchain. IPFS is used to host all the necessary files and BigchainDB stores metadata and IPFS addresses. Smart contracts store the BigchainDB transaction identifiers and they are used to verify that the information sent to the container is valid. For the purpose of this work, the off-chain computations take place in Docker containers and the output is stored under IPFS addresses which are written on the blockchain.

3.2.5 Evaluation

Once the prototype was developed, a use case scenario with different stakeholders was built in order to predict chronic kidney diseases on patients. The system was tested using private medical records and a ML algorithm that detects chronic kidney diseases based on symptoms. The medical records were sent encrypted in a Docker container along with the open-source software and the computation output was made available cryptographically secure to the recipient. The system was found to meet the requirements of the actors, however there exists some limitations that are described in chapter 7. The entire scenario is discussed in detail in chapter 4.

3.2.6 Communication

At the time of writing, this thesis has not been published in a paper or in an academic journal, therefore this activity is not discusses in this work.

4 Analysis and System Requirements

This chapter begins with a detailed analysis of a use case scenario that was used to extract the requirements of the application, followed by a description of the functional and non-functional requirements of the system by conducting a stakeholder analysis. At the end, we present the architecture of Dapp and we show how our system conforms to this architecture.

4.1 Detailed Use Case Analysis

Among the use case scenarios that described in section 3.1, we chose the chronic kidney disease prediction scenario to analyse our system and derive the stakeholder requirements.

4.1.1 Chronic Kidney Disease Prediction Scenario

This section describes the chronic kidney disease prediction scenario which was used to collect the user requirements. We chose this case because the prediction process requires for privacy of patient's records as well as for the execution of resource-intensive ML computations to detect chronic kidney diseases of patients based on their symptoms.

For the purpose of this scenario, we define three actors: a laboratory, a research institute and a container provider and the interactions between them and the application. We assume that the laboratory has collected data about patient symptoms but it lacks the technical knowledge to develop an algorithm that detects chronic kidney diseases for a patient as well as the computer resources to execute it. The research institute is focused on ML techniques to develop algorithms for healthcare applications and provides an algorithm that uses patient's symptoms to detect chronic kidney disease. Further, the container provider offers an off-chain container which is configured to support machine learning computations and runs on a computer with enough computing resources. A use case diagram for the preceding scenario is illustrated in Figure 4.1 and contains the three actors (a laboratory, a research institute and a container provider), each one with different requirements.

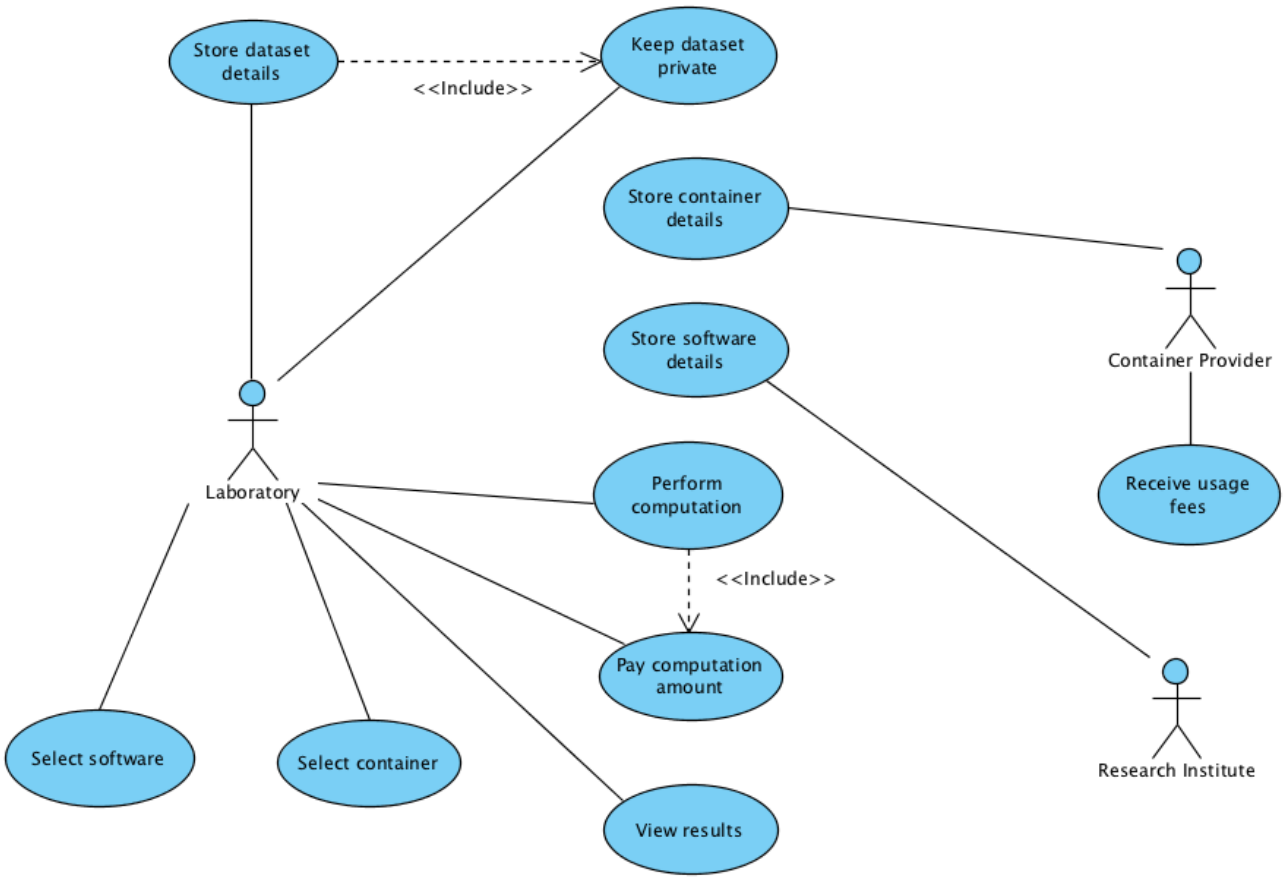

Figure 4.1: UML use case diagram for the chronic kidney disease scenario.

The stakeholder requirements of the chronic kidney disease scenario are summarized in Table 4.1. The laboratory has a double role, as data provider and individual user, since it uses its own dataset and chooses the software and the container for the computation. It also requires for privacy of the uploaded dataset and the computation output. The software offered by the research institute can have a usage fee which will be transferred to the research institute by the system after its usage. Further, the container provider offers an off-chain mechanism to host computations and requires for receiving a fee for using its services. The laboratory is responsible for paying these fees to the owners.

As a ...	I want/need to ...
Laboratory / Individual user	Be able to upload datasets and metadata in the system.
	Prevent someone from reading my dataset apart from the container assigned for the computation.
	Be able to select available dataset, software and container and use the system to perform computations.
	Identify myself in a cryptographically secure way upon accessing my computation results on the blockchain, so that no unauthorized entities can access them.
Research Institute	Be able to upload software, metadata and cost of usage (optional) in the system.
	Receive funds after a successful computation if my software was used by someone else (and cost exists).
Container provider	Be able to offer containers, metadata and cost of usage in the system.
	Receive funds after a successful computation if my container was used by someone else.

Table 4.1: Stakeholder requirements of the chronic kidney disease scenario.

4.2 System Requirements

The proposed system is designed to support different stakeholders to interact with the application, namely data providers, software providers, container providers and individual users. After conducting a stakeholder analysis, as the study of Balta, Greger, Wolf and Krcmar (2015) [3] suggests, the different actors and their requirements were identified as well as the interactions between them. The requirements collected from the application domain and the use case scenario described in section 4.1.1 can be distinguished between two types, functional and non-functional. The dependencies between the different stakeholders and their functional requirements are presented in Figure 4.2 using a dependency network diagram [3].

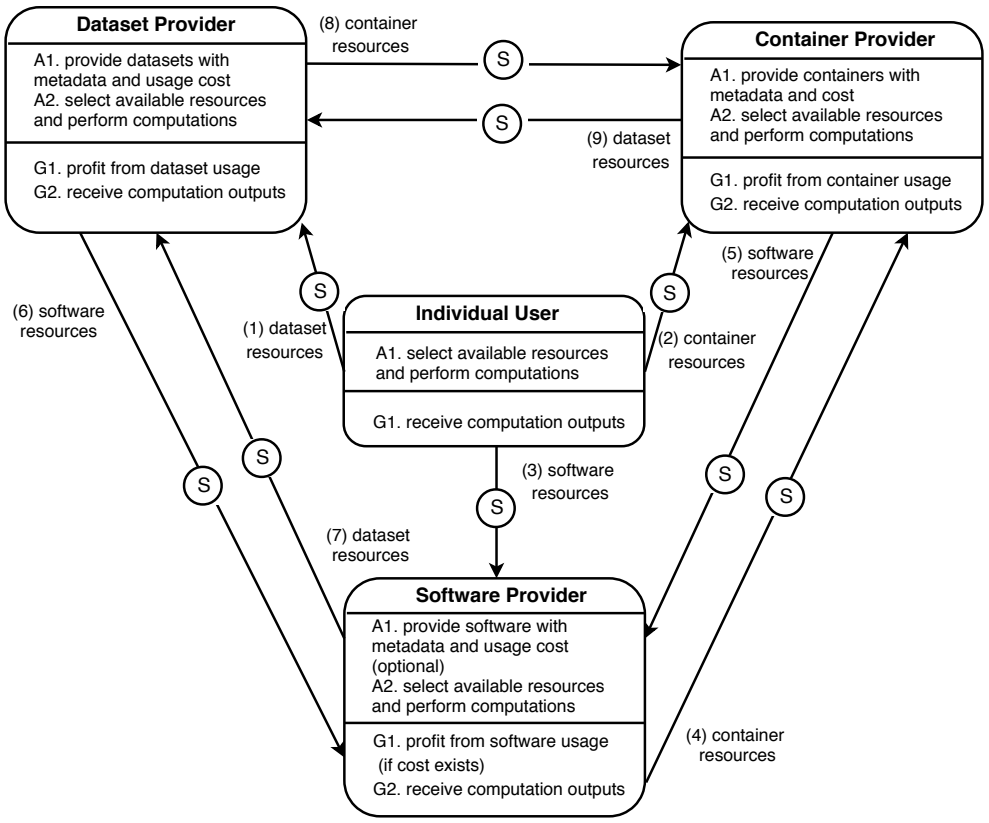

Figure 4.2: Interactions, dependencies and functional requirements between actors.

Regarding our stakeholder analysis and the functional requirements of the system, dataset providers offer resources that are used inside containers during the computation phase. They submit their datasets with metadata and usage fee (activity A1 of the dataset provider) and they select available resources already uploaded to the application to perform computations for their own purpose (activity A2 of the dataset providers). Their goal is to collect the usage fees from their datasets (goal G1 of dataset provider) and receive outputs in case they ordered computations (goal G2 of dataset providers). Datasets are critical in the relationship between individual users and other providers (dependencies 1, 7, 9 "dataset resources").

Container providers offer containers that host computations (activity A1 of container provider). They also select available resources in the system to perform computations for their own purpose (activity A2 of container provider). Containers are important in the relationship between individual users, software and dataset providers and the dependencies are modelled based on container resources (dependencies 2, 4, 8 "container

resources"). Container providers pursue their goals of collecting fees by offering their containers (goal G1 of container providers) and receive computation results (goal G2 of container providers).

Software providers offer open-source software that is used during computations. It is needed by providers and individual users in order to run inside containers over datasets and produce results. The dependency is modelled based on software resources (dependencies 3, 5, 6). Providers upload metadata about their software with an optional usage fee. If the fee is set, it acts as a "reward" and the provider will earn the specified amount (goal G1 of software provider), otherwise the resource is offered for free. Further, software providers can choose available datasets and containers to perform computations according to their needs (activity A1 of software provider) and receive the results (goal G2 of software provider).

Individual users are dependent on providers. Their dependencies (dependencies 1, 2, 3) are modeled as individual users need dataset, software and container resources to order computations and receive the outputs (goal G1 of individual users).

Regarding the non-functional requirements, the system should be able to provide security of the transaction details. Smart contracts should be simple and execute in a way that reduces the transaction costs to the minimum necessary. Another important requirement is that the contracts should be upgradable without losing the already stored data on the blockchain, i.e. they should be designed in a way that isolate the data from the logic of the application.

4.3 System Analysis

The system designed in this thesis conforms to the Dapp architectural pattern. The architecture of a Dapp differs from a typical web application. The differences lie mainly in the business logic and the data storage layers, as Figure 4.3 illustrates. Decentralized applications implement the logic in the smart contracts and store the data on the blockchain, whereas web applications follow a different architectural pattern by storing the data in a traditional database or locally in the machine.

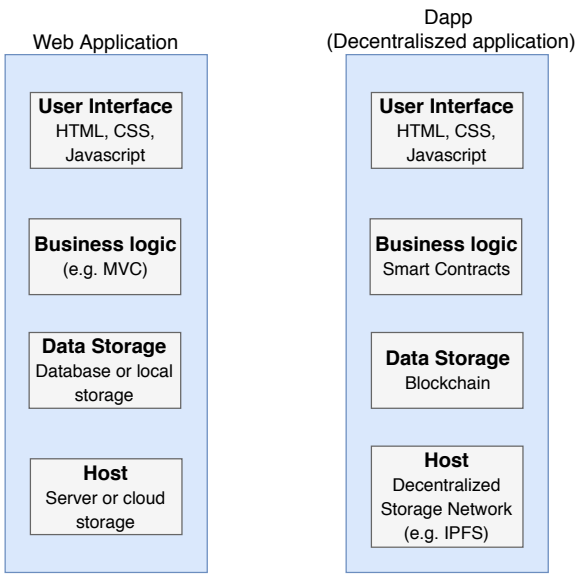

Figure 4.3: Web application vs. Dapp architecture.

The proposed system is a Dapp since it operates autonomously and its code is open-source. Its back-end runs on a public decentralized blockchain, that is the Ethereum blockchain and the front-end can be hosted on a decentralized storage network such as IPFS. The components and the users of system need Ether, the Ethereum's cryptocurrency, to interact with the blockchain, execute transactions and make payments.

5 System Design

This chapter describes the software architecture of the BlackboxChain application using the "4+1 architectural view model" [32]. The architecture is presented from the logical, development, process and physical perspective by providing an analysis of each case followed by a UML diagram.

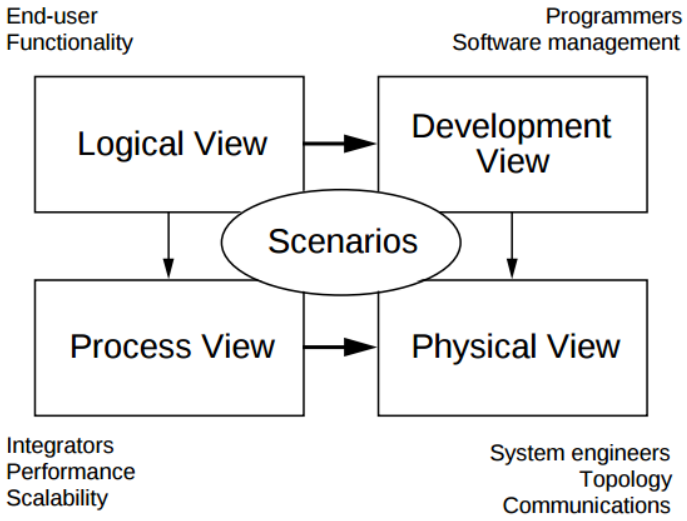

Figure 5.1: The "4+1" view model. Image taken from [32].

The "4+1" view approach was used to organize the application's architecture representations into different views that meet the user requirements, as shown in Figure 5.1. The rest of this chapter describes the four main views in detail. The scenario was described in chapter 4.

5.1 Logical View

The logical view describes the system of smart contracts used in this application. Their architecture was based on the *Hub and Spoke Topology Design* [56]. This approach isolates the storage from the logic and offers contract upgradability without losing the already stored data on the blockchain.

The architecture of the system utilizes two types of smart contracts, namely *registry contracts* and *manager contracts*. A detailed description of the registry contracts is given below and their source code can be found on Appendix A.

Registry contracts are responsible for writing to and reading from the blockchain. They contain setter and getter methods to store and access the necessary information.

- Dataset, software and container registry contracts store information regarding the dataset, software and container resources. These contracts are responsible for storing the BigchainDB transaction identifiers, which contain IPFS addresses of the corresponding files and metadata, as well as the checksum of the stored information in order to detect errors and corrupted data. Additionally, the cost (in Ether) for renting these resources and the owner's Ethereum public address are stored.

- The *ComputationRegistry* contract stores all the necessary information needed to perform a computation in the container. The user's public key is stored on the blockchain and it is required by the off-chain container to encrypt a one-time password which in turn is used to encrypt the computation result. Each computation has a status *PLACED*, *SUCCEEDED* or *CANCELLED*. Once a new computation arrives, its status is set to *PLACED*. Upon a successful execution, its status changes to *SUCCEEDED*, otherwise to *CANCELLED*.

- The *ResultRegistry* contract maintains the IPFS addresses which host the computation outputs for each user as well as the one-time passwords used to decrypt these outputs.

The second type of smart contracts, namely *manager contracts*, implement the logic and call methods from the registry contacts in order to add to, update and read from the blockchain. A detailed explanation about this type of contacts is given below and their source code can be found on Appendix A.

- The *RegistryManager* contract adds new dataset, software and container entries on the blockchain by calling the methods from the corresponding *registry contracts*. Each entry has a unique identifier which is computed by this contract. Anyone in the network with an Ethereum account can call the methods of this contract and interact with the blockchain.
- The *ComputationManager* contract is responsible for adding new computations and handling payments. Anyone in the network with an Ethereum account can add a new computation but only a process with specific Ethereum public address can change the status of a computation and trigger payments. This process acts as a trusted third-party in the system that brings off-chain information on-chain and it is described in detail in section 5.2. The methods of this contract are called by the trusted third-party to distribute the paid amount to the dataset, software and container providers whose resources were used in case of a successful computation, or return the funds to the user who initiated the computation in case of a failure.

The UML class diagram of the system of smart contracts is illustrated in Figures 5.2 and 5.3

The *ResultManager* contract triggers events according to the state of the computation and calls the methods of the *ResultRegistry* contract to store information about the results on the blockchain. It accepts calls only from a trusted third-party to store the data, i.e. the *oracleAddress*.

The *ResultRegistry* contract keeps track of the computation outputs and accepts calls only from authorized entities, i.e. the *ResultManager* contract.

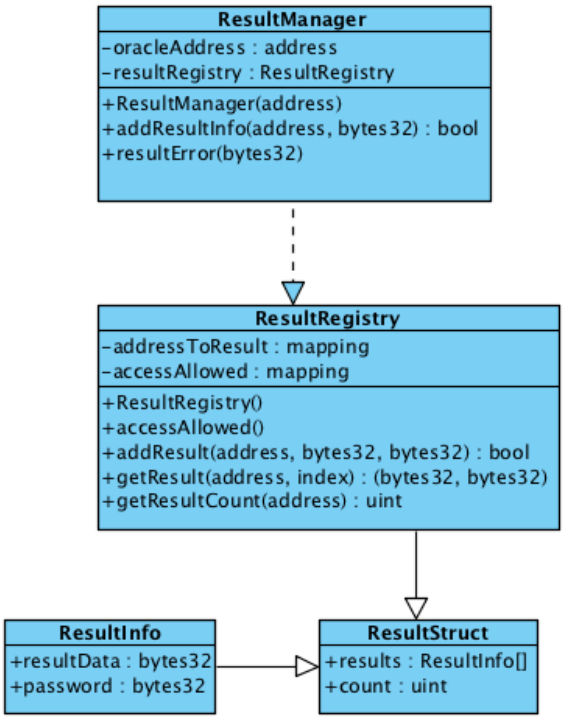

Figure 5.2: UML class diagram of the *ResultManager* and *ResultRegistry* contracts.

Figure 5.3 illustrates the rest of the registry and manager contracts as well as the interactions between them. The *RegistryManager* contract calls the registry contracts to store information about the resources on the blockchain. The *ComputationManager* accepts calls from a trusted third-party, i.e. the *oracleAddress*, in order to fetch information about resources from the registry contracts.

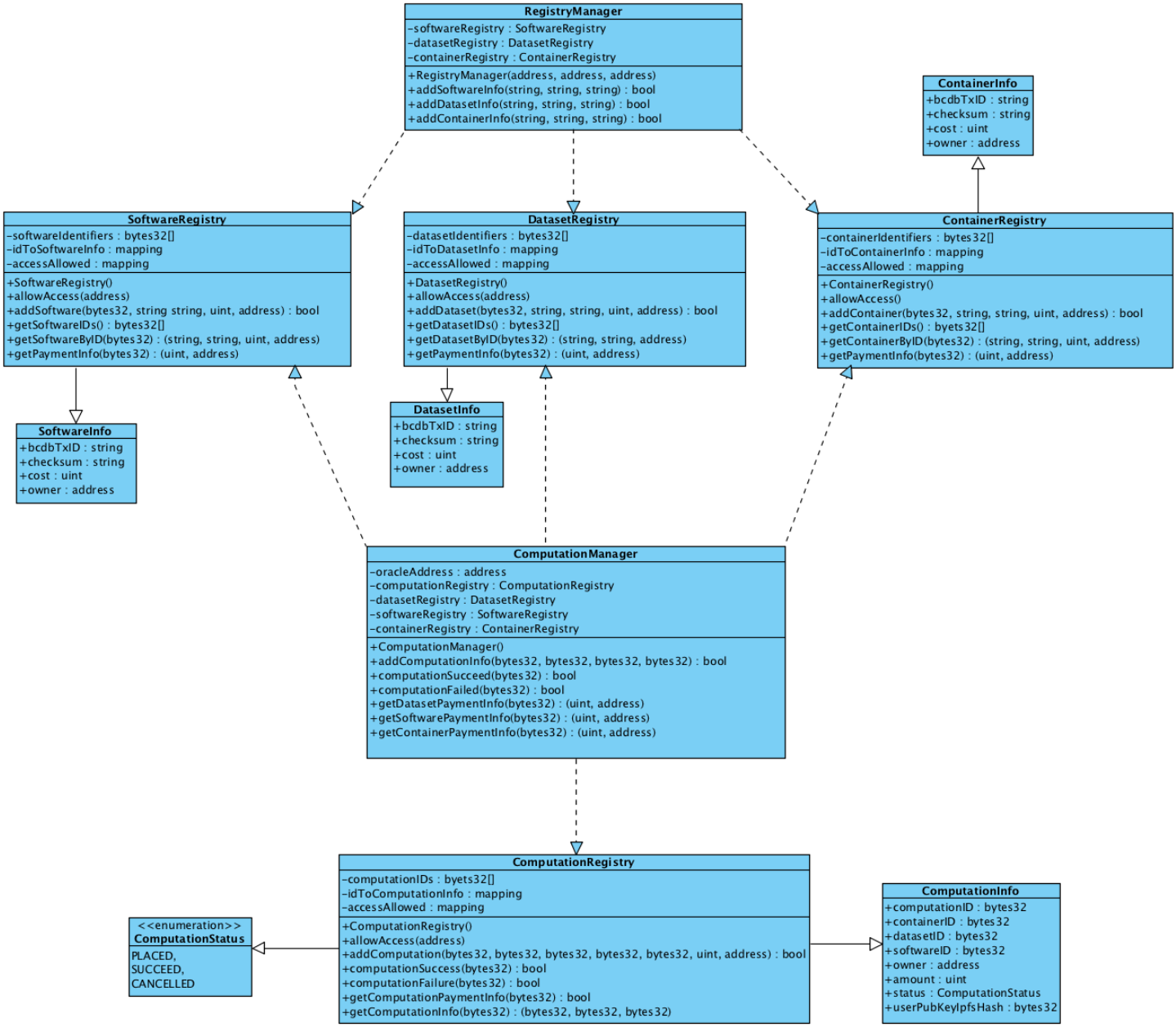

Figure 5.3: UML class diagram of the registry contracts and the *ComputationManager* contract.

5.2 Development View

The development view describes the components and modules used to assemble the BlackboxChain application. Table 5.1 summarizes these elements and they are described in detail below.

Component/Module	Description
Web browser	Component used to provide a user-friendly interface to interact with the application and the blockchain.
BigchainDB	Module used to store metadata and IPFS addresses about dataset, software and container resources in the BigchaindDB network.
Ethereum Blockchain	Component used to deploy smart contracts and store information about dataset, software, container resources and computation results.
IPFS	Module used to host dataset, software and container files in the IPFS network.
ValidationOracle	Third-party process that validates computations and handles the communication between the smart contacts and the off-chain containers.
DockerHost	Module that hosts containers to perform computations.

Table 5.1: Components and modules of the BlackboxChain application.

- **Web browser** - This component provides a web interface for the users to interact with the application. Users need an Ethereum wallet browser plug-in, namely Metamask [38] to manage their accounts. The web interface provides the functionality to store datasets, software and containers on the blockchain, as well as start the execution of a computation into an off-chain container. It also displays the computation results for the Ethereum public address that is connected to the system.
- **BigchainDB** - BigchainDB was presented in section 2.4.2. It provides an API

to store assets and query the database. It is used by the application to store metadata of the resources like filenames, file specifications, IPFS addresses and usage costs.

- **Ethereum Blockchain** - The system of smart contracts is deployed on the Ethereum blockchain. This component provides an Application Binary Interface (ABI) to the users to call the contract methods and read or write information on the blockchain.

- **IPFS** - IPFS was described in detail in section 2.4.3. This module provides an API to add and read files from the IPFS network. Dataset, software and container providers add their files to the IPFS network manually before they interact with the BlackboxChain application and then they store the IPFS addresses on BigchainDB using the web interface.

- **ValidationOracle** - Smart contracts cannot access data outside of the blockchain, thus there is a need for data feeds, called oracles, designed to submit external data to the smart contracts. In the developed system, the *ValidationOracle* plays the role of an oracle, therefore it is an important component of the application. One of its main tasks is to validate the information before it is sent to the off-chain container. To achieve this, it computes the checksum of the data and compares it with the checksum stored on the blockchain. Further, it retrieves the status of the container which is assigned to execute the computation. The execution starts only if the checksum values are equal and the container is running. In this case the oracle set the status of the computation to *SUCCEEDED* by calling the *ComputationManager* contract, otherwise to *CANCELLED*. Another important task of this component is to receive the computation output from the container and write it on the blockchain by calling the *ResultManager* contract.

- **DockerHost** - For the purpose of this thesis, computations take place in Docker containers. Providers host running containers in their machines and they offer them for a fee in order to perform computations. Files regarding the containers, such as their the specifications as well as the scripts that orchestrate the execution of the computations are hosted by IPFS.

The UML component diagram for the BlackboxChain application is shown in Figure 5.4.

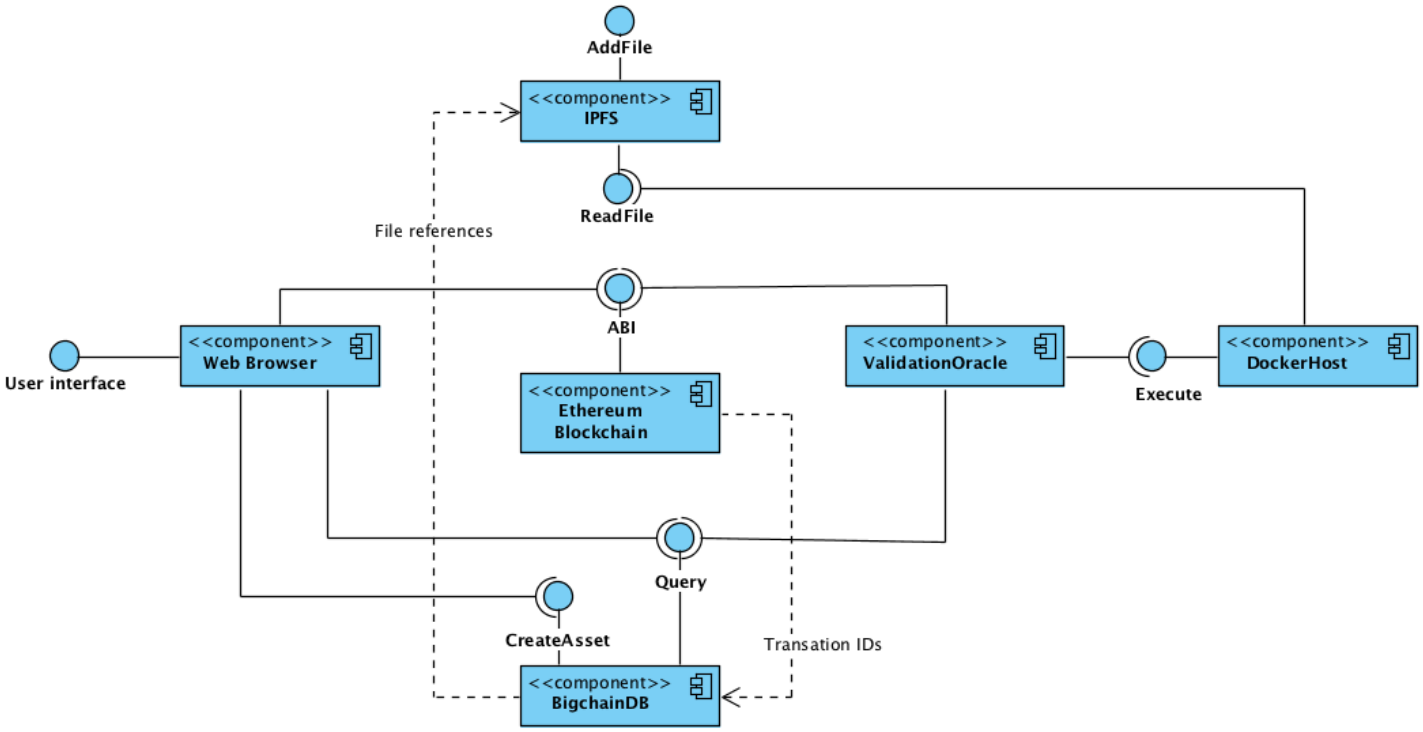

Figure 5.4: UML component diagram of the BlackboxChain application.

5.3 Process View

The process view contains the interactions between the actors and the different components of the system. This section describes the process for two different actors, providers and individual users.

The process of interacting with the system from a provider's perspective in order to offer dataset, software and/or container resources is illustrated in Figure 5.5. In step 1, the providers store their files on IPFS. These files can be datasets, software, or container specification files. Datasets have to be encrypted by the provider before added on IPFS. Once these files are added to the IPFS network, their metadata as well as the returned IPFS address is stored on BigchainDB by the web interface, as shown in step 4. The checksum of this information is computed at the front-end at step 6

and it is stored on the blockchain along with the BigchainDB transaction identifier at step 7. Once this process is finished, the uploaded resources are made available to the users.

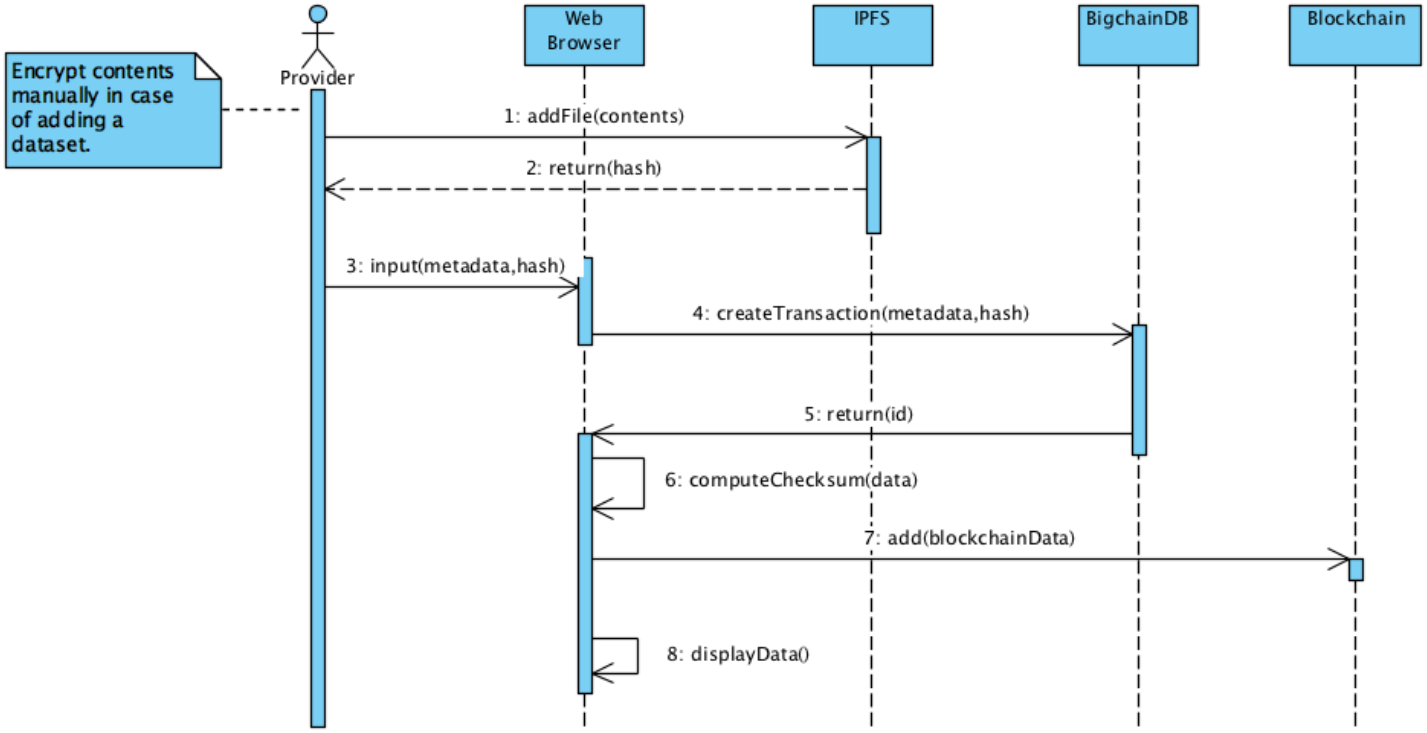

Figure 5.5: UML sequence diagram from a provider's view.

The process of interacting with the system as an individual user as well as all the steps needed to perform a computation are illustrated in Figure 5.6. Providers follow the same process in case they want to act as individual users and order computations. In step 1, the oracle service is running and awaits for new computation events. Using the front-end, users select which dataset, container and software they want to use and the information about the computation is stored on the blockchain in step 3, including the amount in Ether for using these resources. Users provide also their public key which is required by the container during the process of encrypting the computed

result. In step 4, the oracle is notified with a new computation arrival and in the next step it reads the checksum values from the blockchain. The checksum of the computation data is calculated in the oracle by querying the BigchainDB and it is compared with the blockchain value. Also, the status of the selected container is checked in step 10. If the checksums are equal and the selected container is *Running*, it is notified to start the execution of the computation, as step 12 shows. In this case, the computed result is encrypted with a one-time password, this password is encrypted with the user's public key and the ciphertexts are stored on IPFS. In step 14, the oracle receives the IPFS addresses and in step 15 it stores these addresses on the blockchain. Then, the corresponding providers should receive the specified amount in Ether for offering their resources. On the other hand, as step 19 shows, an error message is received and the funds are returned to the initiator of the computation. In both cases, the front-end notifies the user with either the IPFS addresses that contain the result or with an error message, as steps 17 and 22 show.

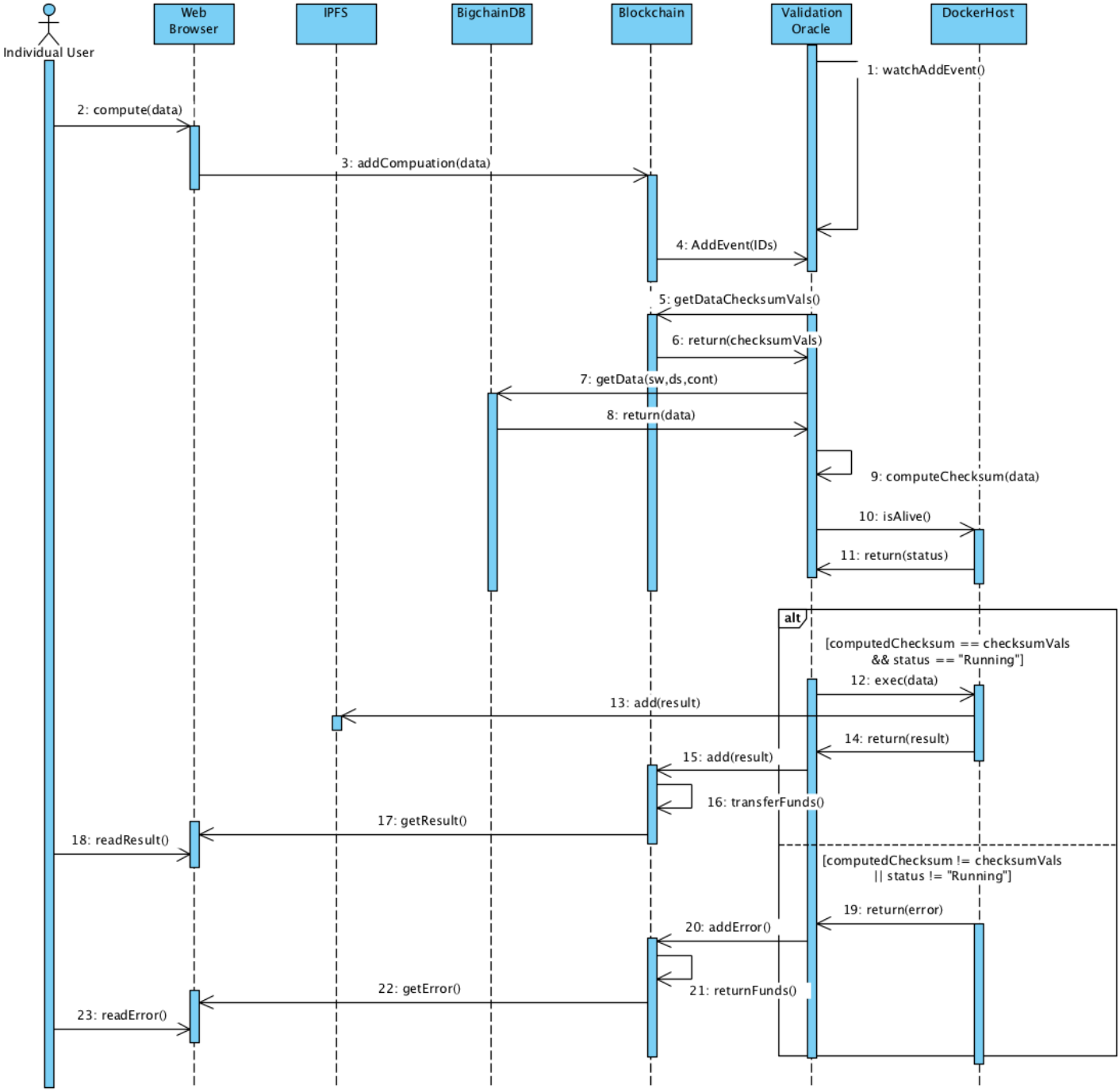

Figure 5.6: UML sequence diagram from a provider's / individual user's view.

5.4 Physical View

The physical view describes how the software and the hardware of the BlackboxChain application are connected and how the system is installed when it operates. Figure 5.7 illustrates the UML deployment diagram of the application.

- **Web front-end** - The front-end is running on a browser on a computer. It can also be hosted on IPFS in order to be decentralized and publicly accessible. It communicates with the Ethereum blockchain using Remote Procedure Calls (RPCs) and with a BigchainDB node over HTTP.
- **EVM** - The smart contracts used by the system are deployed into the EVM which acts as the back-end of the application.
- **BigchainDB Cluster** - This component contains a set of connected BighchainDB nodes. The oracle and the front-end connect to one of these nodes in order to query the database or add user data in the network.
- **IPFS** - A running IPFS daemon is required for applications to communicate with the rest of the IPFS network. Docker containers connect to a running IPFS daemon over HTTP to add and read files in the network.
- **ValidationOracle.js** - This is a Node.js [41] process that is deployed on a computer. It uses RPC to communicate with the EVM and HTTP to connect with a BigchainDB node and the DockerHost.
- **DockerHost** - This device describes a set of running Docker containers hosted on a computer. Each container is isolated from the surrounding environment and it communicates with an IPFS daemon and the ValidationOracle.js process using HTTP.

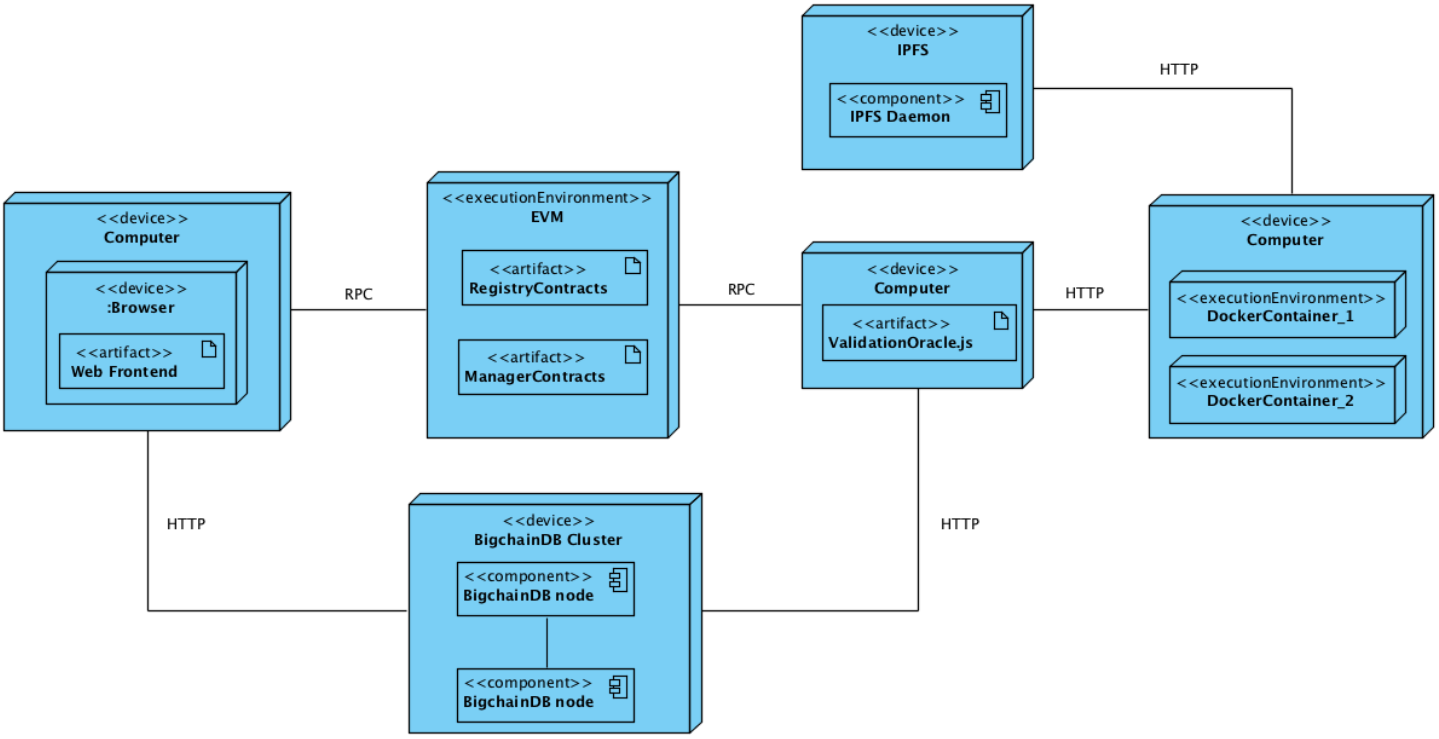

Figure 5.7: UML deployment diagram of the BlackboxChain system.

6 Prototypical Implementation and Evaluation

This chapter describes the technologies and tools that were used for the PoC implementation, followed by the evaluation of the system based on the designed architecture and the developed prototype.

6.1 Tools and Technologies

6.1.1 AngularJS

AngulaJS [17] is an open source Javascript framework for building client applications in Hypertext Markup Language (HTML) and Typescript and supports the Model View Controller (MVC) architecture [18]. Hence, it provides project structure organization by enforcing the isolation of data from the interfaces and the logic in order to increase reusability of the components into other interfaces. It extends HTML by custom elements, attributes and classes, known as directives and it provides features for binding the data with the view. Thus, it helps developers to emphasize on building applications that are maintainable, testable and easy to extend. The AngularJS framework was used to implement the user interface of the BlackboxChain application.

6.1.2 Docker

Docker [20] is an open source platform that allows applications to be deployed and run inside isolated lightweight entities, called containers. Developers use containers to create a standardized application environment which is portable and run everywhere where the Docker Engine runtime is installed. Internally, Docker uses a client-server architecture and a daemon acts as a server that builds and runs containers. The server listens on a Unix socket, making itself unreachable over the network. This security feature prevents clients who connect to the server from taking control of the entire host. In the BlackboxChain application, the off-chain computation takes place in already configured and running Docker containers.

6.1.3 Truffle

Truffle [55] is a development framework that makes it easier and simpler to test and deploy Ethereum smart contracts. It provides an Ethereum blockchain simulator for development purposes to test and deploy smart contracts by creating a set of Ethereum accounts with fake Ether. It offers contract compilation as well as automated testing using frameworks such as Mocha [39] and Chai [10]. Further, it offers easy contract deployment in any private, public or test blockchain. In this work, Truffle was used for the entire development of the smart contracts in order to interact with the blockchain and test their functionality.

6.2 System Evaluation

In this section we present the evaluation of the system based on how the developed architecture fulfils the stakeholder requirements, followed by the evaluation based on the developed prototype. For the evaluation, the chronic kidney disease prediction scenario was used, as described in section 4.1.1.

6.2.1 Evaluation based on the Reference Architecture

The stakeholder requirements as well as the decisions taken to fulfil them are summarized in Table 6.1. The designed architecture combines different technologies and satisfies all the requirements of each actor.

To support the requirements of each stakeholder, blockchain technology in combination with BigchainDB and IPFS store all the necessary information about datasets, software and containers. To keep the datasets private, a one-time password is used to encrypt the data and the public key of the container which is assigned to execute the computation is used to encrypt the password. This process has to be done by the data provider prior to uploading the information in the system, although several scripts are implemented to ease this process. Further, all the uploaded resources are displayed to the actors by the front-end and they can be used for computations. The blockchain infrastructure provides trust between the stakeholders and offers mechanisms to handle payments upon a successful computation or to return money in case of failures. It also offers identification and wallet management for the users in order to interact with the system and view their computation results.

As a ...	I want/need to ...	Fulfilment
Laboratory / Individual user	Be able to upload datasets and metadata in the system.	IPFS, blockchain technology and BigchainDB are used to store information regarding datasets.
	Prevent someone from reading my dataset apart from the container assigned for the computation.	Encryption of the dataset, decrypt only inside the container before computation.
	Be able to select available dataset, software and container and use the system to perform computations.	User-friendly interface to interact with the system.
	Identify myself in a cryptographically secure way upon accessing my computation results on the blockchain, so that no unauthorized entities can access them.	Fulfilled by the blockchain infrastructure and the smart contracts and cryptography.
Research Institute	Be able to upload software, metadata and cost of usage (optional) in the system.	IPFS, blockchain technology and BigchainDB are used to store information regarding software.
	Receive funds after a successful computation if my software was used by someone else (and cost exists).	Use smart contracts to enforce the payments.
Container provider	Be able to offer containers, metadata and cost of usage in the system.	IPFS, blockchain technology and BigchainDB are used to store information regarding containers.
	Receive funds after a successful computation if my container was used by someone else.	Use smart contracts to enforce the payments.

Table 6.1: Evaluation of the system architecture based on the requirements of the chronic kidney disease scenario.

To address the non-functional requirements, blockchain technology in combination with encryption is used to provide security of the transaction details. Container providers exchange their cryptographic public keys with dataset providers and receive the public keys of individual users. Upon a successful computation, the output is encrypted in the container before it is stored on IPFS with a one-time password and it is made available encrypted with the public key of the actor who initiated the computation, in our case that is the public key of the laboratory.

6.2.2 Evaluation based on the PoC Implementation

The evaluation of the requirements of the chronic kidney disease scenario based on the prototype implementation is summarized in Table 6.2. The table presents the functional requirements of the system as well as the decisions taken in the PoC implementation in order to satisfy them. At the end of this section, Figures 6.1, 6.2, 6.3, 6.4 and 6.5 illustrate the page layouts of the BlackboxChain user interface to insert datasets, software and containers, view available resources and display the computation results.

The developed system of smart contracts play an important role in the functionality of the prototype. The registry contracts are used to store information regarding datasets, software and containers, as described in 5.1. In our scenario, the laboratory stores the dataset with the patient symptoms encrypted on IPFS and enters their metadata along with the IPFS address in BigchainDB. To detect errors, we compute the checksum of the saved information and we store it in the *DatasetRegistry* contract along with the BigchainDB transaction identifier. The research institute and the container provider also have to upload their resources on IPFS and then enter the metadata on BigchainDB in order to upload information regarding software and containers in the *SoftwareRegistry* and *ContainerRegistry* contracts respectively.

The process of encrypting the dataset is not done automatically by the system, although scripts are provided to perform this task. In our scenario, the laboratory has to download the container's public key, generate a one-time password and encrypt its dataset with this password. Then, the password must be encrypted using the public key and the two generated files must be added on IPFS.

Software and container providers must be paid upon a successful computation in case they have set a fee for the usage of their resources. The *ComputationManager* contract enforces the payment to the corresponding providers, or returns the amount to the actor who initiated the computation in case of a failure. The *ValidationOracle* is the trusted third-party that provides the necessary details to the *ComputationManager* contract regarding the computation and checks the output for errors. In our scenario, the container provider will charge the laboratory for using the container, but the research institute does not have to set any fee for the software. However, this feature is supported by the system in case it is necessary.

The prototype offers a front-end for the users to interact with the system, upload their data and view available resources. All the actors described in our scenario can view the uploaded dataset, software and container information and perform computations using the front-end. In order to interact with the blockchain and identify themselves, the Metamask plug-in is used to provide them with a wallet with unique Ethereum private and public addresses. The *ResultRegistry* contract stores the

IPFS addresses of the computation outputs and the one-time passwords according to Ethereum public addresses. Thus, in our scenario, the laboratory has to connect in our application with its Ethereum public address in order to access the predictions before it decrypts them using its cryptographic public key and the provided password.

As a ...	I want/need to ...	Fulfilment
Laboratory / Individual user	Be able to upload datasets and metadata in the system.	Store files on IPFS, create BigchainDB transactions and call *DatasetRegistry* contract's methods.
	Prevent someone from reading my dataset apart from the container assigned for the computation.	Encrypt datasets using the provided scripts.
	Be able to select available dataset, software and container and use the system to perform computations.	AngularJS front-end provides an interface to interact with the system.
	Identify myself in a cryptographically secure way upon accessing my computation results on the blockchain, so that no unauthorized entities can access them.	Store results on IPFS and in the *ResultRegistry* contract and use Metamask to manage wallet.
Research Institute	Be able to offer software, metadata and cost of usage (optional) in the system.	Store files on IPFS, create BigchainDB transactions and call *SoftwareRegistry* contract's methods.
	Receive funds after a successful computation if my software was used by someone else (and cost exists).	Call *ComputationManager* contract to store computations and *ValidationOracle* to validate the process.
Container provider	Be able to offer containers, metadata and cost of usage in the system.	Store container files on IPFS, create BigchainDB transactions and call *ContainerRegistry* contract's methods.
	Receive funds after a successful computation if my container was used by someone else.	Call *ComputationManager* contract to store computations and and *ValidationOracle* to validate the process.

Table 6.2: Evaluation of the prototype based on the requirements of the chronic kidney disease scenario.

Regarding the non-functional requirements, asymmetric encryption in combination with one-time passwords prevent unauthorized access to the uploaded datasets and computed results. Also, the system of smart contracts is designed in a way that provides isolation of the logic and the data using the "Hub and Spoke Topology Design", as described in section 5.1. Thus, the data stored on the blockchain will still exist and the system will continue to operate properly in case of changing the logic.

Figure 6.1 illustrates the page layout with sample data where users can insert information about software. It contains fields such as the name of the software, its IPFS address, specification about its input parameters and its usage cost.

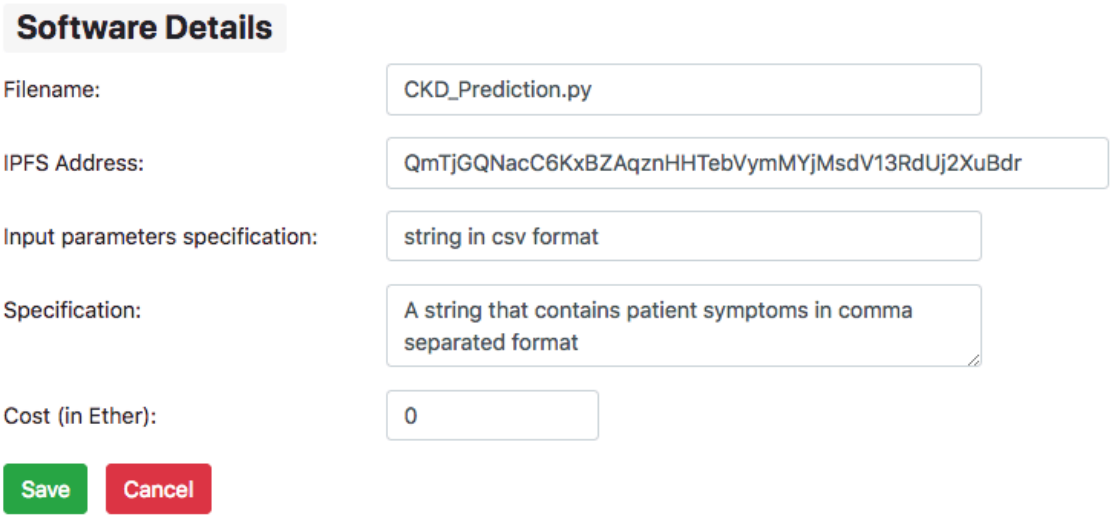

Figure 6.1: BlackboxChain: Page layout to insert new software.

Figure 6.2 illustrates the page layout with sample data where users can insert information about containers. It contains fields such as the docker container identifier, the IPFS address of the container filesystem, the public key of the container, a specification and its usage cost.

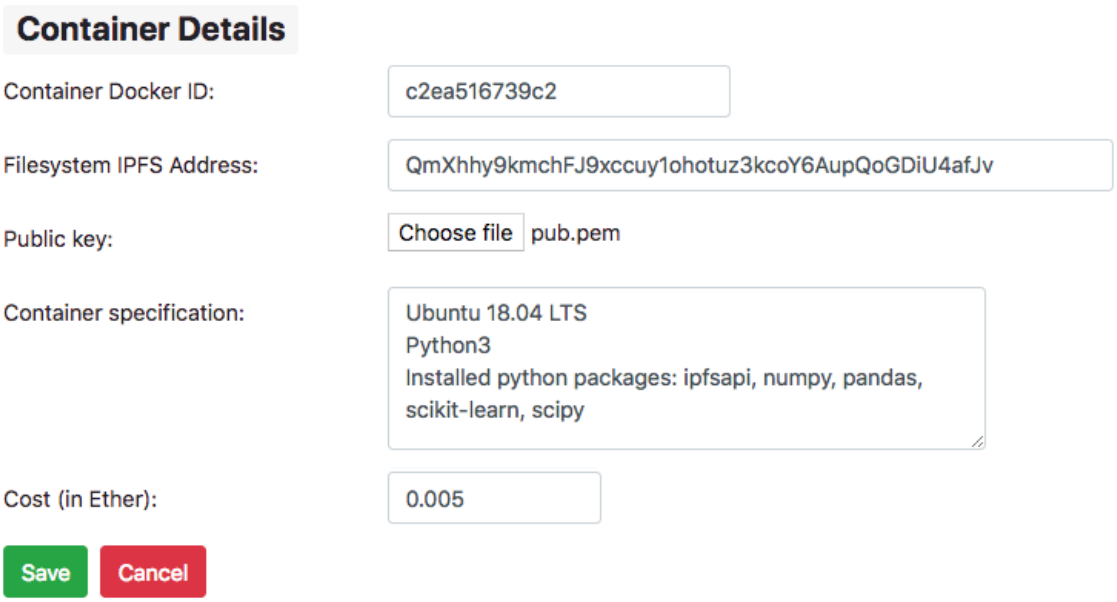

Figure 6.2: BlackboxChain: Page layout to insert new container.

The page layout in order to insert information about datasets is illustrated in Figure 6.3. It contains fields such as the name of the dataset, the IPFS address under which the dataset is stored, the IPFS address under which the one-time password is stored in order to decrypt the dataset, specification about the dataset and its usage cost.

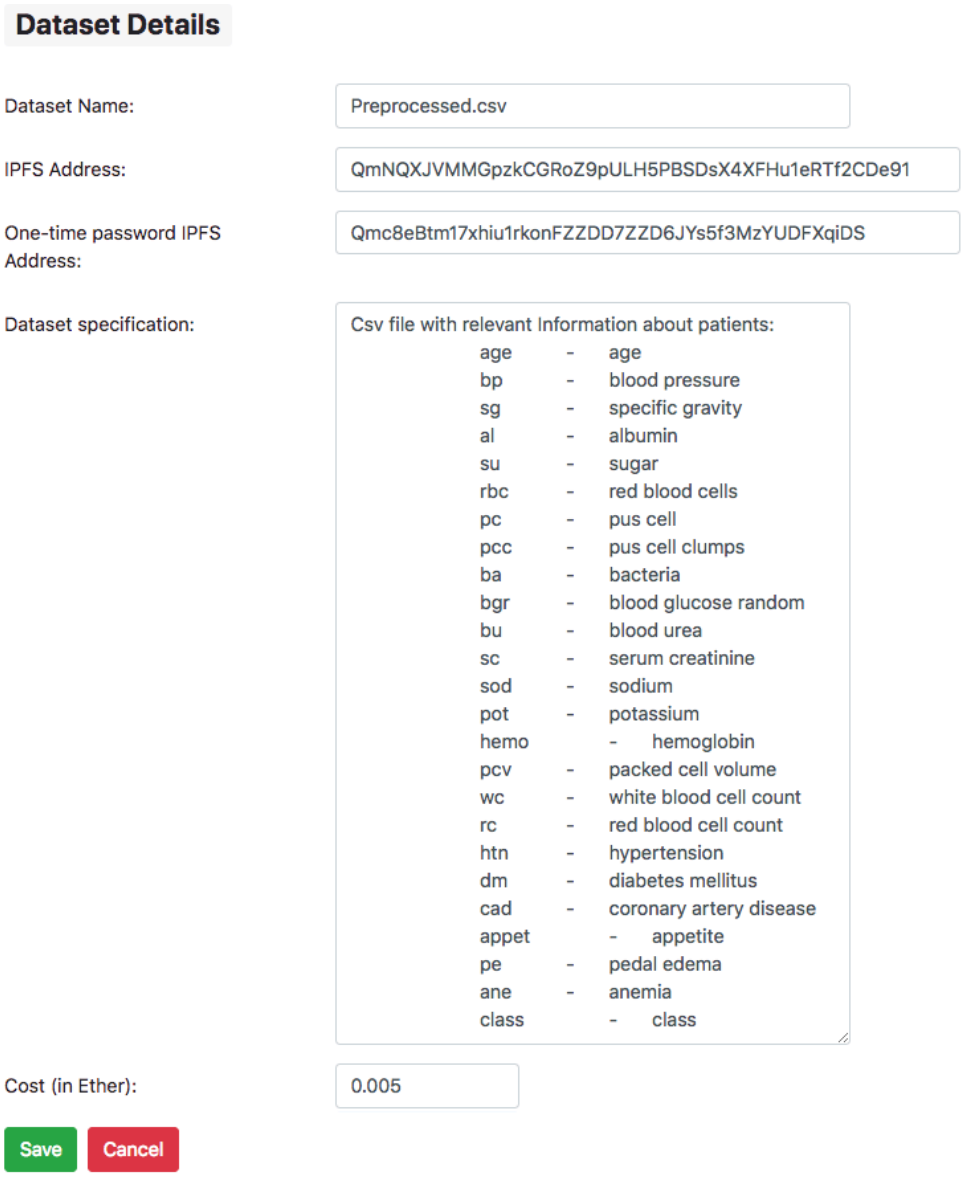

Figure 6.3: BlackboxChain: Page layout to insert new dataset.

Figure 6.4 illustrates the layout of the page that displays available datasets, software and containers to the users as well as a field to input their cryptographic public key. Using this page, users can select which resources they want to use and they can trigger an execution of a computation by clicking on the *Run* button.

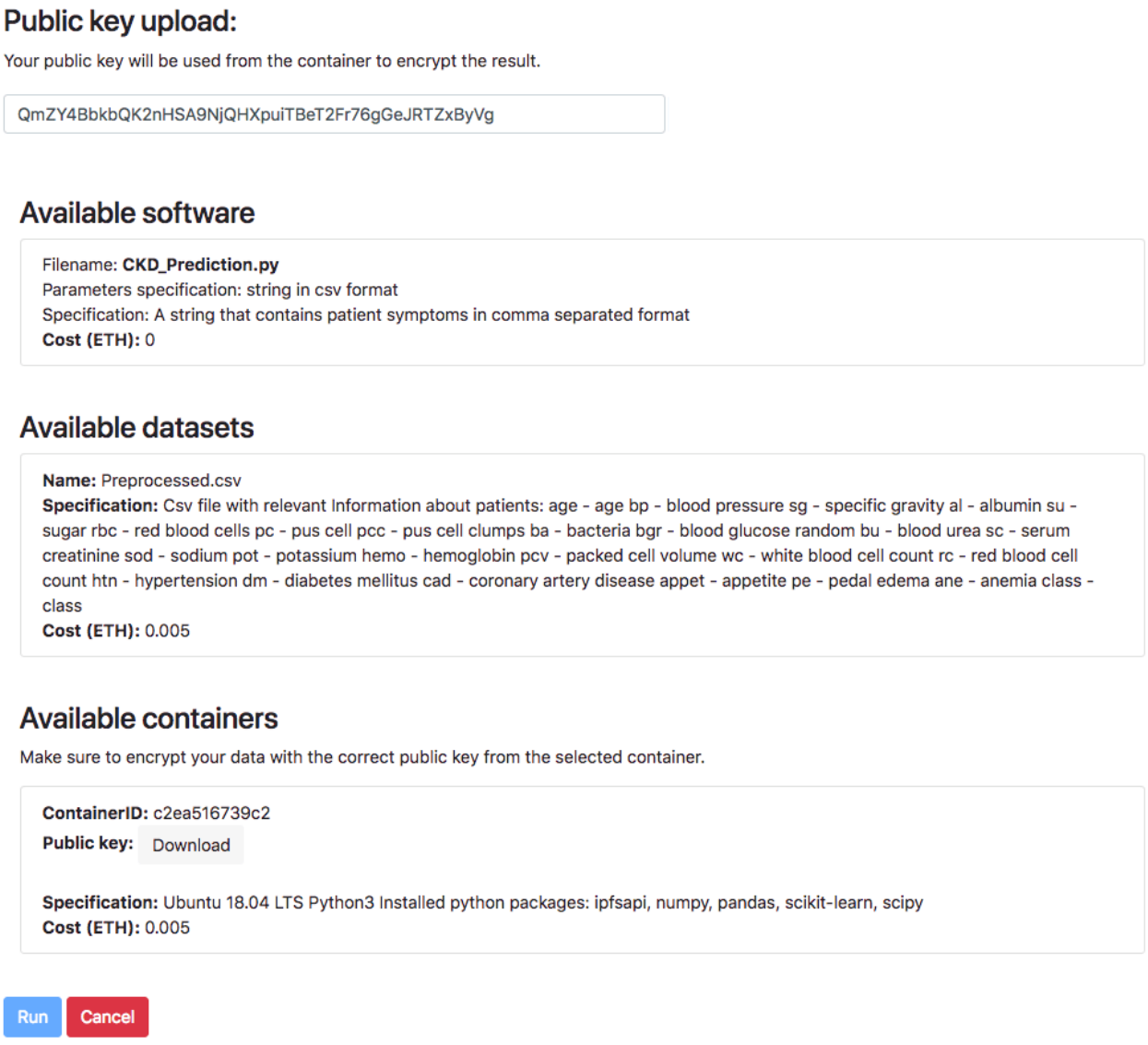

Figure 6.4: BlackboxChain: Page layout to view available datasets, software and containers.

Figure 6.5 depicts the layout of the page that displays the computation outputs based on the current Ethereum public address. It contains the IPFS addresses of the dataset and the one-time password which can be used to decrypt the dataset. Recipients can use the provided scripts in combination with their private key to decrypt this information and access the data.

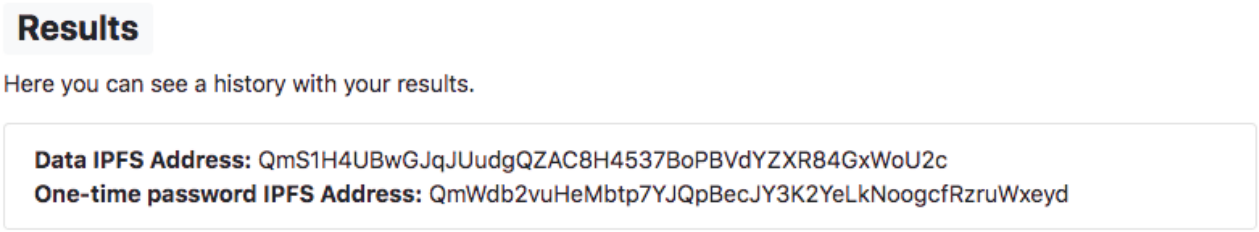

Figure 6.5: BlackboxChain: Page layout to view computation outputs.

7 Conclusions

This thesis proposed a reference architecture for systems that delegate the execution of resource-intensive computations into a secure off-chain mechanism administered by smart contracts using the blockchain technology. A stakeholder analysis was conducted in order to understand the requirements for implementing such an application and a prototype was implemented based on the proposed architecture. The prototype was developed using a system of smart contracts written in the Solidity programming language in combination with IPFS and BigchainDB to host files and metadata. Furthermore, an oracle was used as a trusted third-party data-feed to provide the smart contracts with external data. The reference architecture and the prototype were evaluated based on a chronic kidney disease prediction scenario and the results showed that the developed system is promising and it satisfies the stakeholder requirements.

7.1 Limitations

Although the evaluation criteria were fulfilled, it is worth pointing out the limitations of the proposed system. Protecting data from unauthorised access is important in the proposed system. Encryption enhances the data protection, however the encrypted data has to be decrypted inside the container before it becomes usable. Despite the fact that many encryption schemes that support computing over encrypted data have been developed, namely Fully Homomorphic Encryption (FHE) [25] and Zero-Knowledge Proofs [23], they still remain in theory and applying these schemes in practice is difficult and time consuming, as previous research has shown [26, 48].

7.2 Future Work

In the future, instead of using one trusted oracle to provide off-chain information to the smart contracts, the system can be extended to support a decentralized network of oracles. Multi-signature smart contracts can be used to verify the integrity of the oracles and ensure that all the external data brought on-chain as well as the computation results are valid. Additionally, the current prototype implementation supports software that accepts a single dataset file as input, therefore the system of smart contracts

and the user interface can be extended to support algorithms that read from multiple datasets. Furthermore, regarding the docker hosts, proper administration controls can be applied to ensure that users and containers access only resources needed for their operation.

Bibliography

[1] L. Atzori, A. Iera, and G. Morabito. "The internet of things: A survey." In: *Computer networks* 54.15 (2010), pp. 2787–2805. ISSN: 1389-1286.

[2] A. Bajpai. "Forecasting the Power Generation of Photovoltaic Cells using Machine Learning." Thesis. 2018.

[3] D. Balta, V. Greger, P. Wolf, and H. Krcmar. "E-government stakeholder analysis and management based on stakeholder interactions and resource dependencies." In: *System Sciences (HICSS), 2015 48th Hawaii International Conference on*. IEEE, pp. 2456–2465. ISBN: 147997367X.

[4] J. Benet. "IPFS-content addressed, versioned, P2P file system." In: *arXiv preprint arXiv:1407.3561* (2014).

[5] I. Bentov, C. Lee, A. Mizrahi, and M. Rosenfeld. "Proof of Activity: Extending Bitcoin's Proof of Work via Proof of Stake [Extended Abstract] y." In: *ACM SIGMETRICS Performance Evaluation Review* 42.3 (2014), pp. 34–37. ISSN: 0163-5999.

[6] K. P. Birman. "Consistency in Distributed Systems." In: *Guide to Reliable Distributed Systems: Building High-Assurance Applications and Cloud-Hosted Services*. Ed. by K. P. Birman. London: Springer London, 2012, pp. 457–470. ISBN: 978-1-4471-2416-0. DOI: 10.1007/978-1-4471-2416-0_15.

[7] *Blockchain Infrastructure Landscape: A First Principles Framing*. https://blog.bigchaindb.com/blockchain-infrastructure-landscape-a-first-principles-framing-92cc5549bafe. Accessed: 2018-05-16. 2017.

[8] R. Bonetto and M. Rossi. "Machine Learning Approaches to Energy Consumption Forecasting in Households." In: *arXiv preprint arXiv:1706.09648* (2017).

[9] V. Buterin. "A next-generation smart contract and decentralized application platform." In: *white paper* (2014).

[10] *Chai*. http://www.chaijs.com/. Accessed: 2018-05-16.

[11] J.-S. Coron. "What is cryptography?" In: *IEEE security & privacy* 4.1 (2006), pp. 70–73. ISSN: 1540-7993.

[12] F. Cristian. "Understanding fault-tolerant distributed systems." In: *Communications of the ACM* 34.2 (1991), pp. 56–78. ISSN: 0001-0782.

[13] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, and E. G. Sirer. "On scaling decentralized blockchains." In: *International Conference on Financial Cryptography and Data Security*. Springer, pp. 106–125.

[14] M. Crosby, P. Pattanayak, S. Verma, and V. Kalyanaraman. "Blockchain technology: Beyond bitcoin." In: *Applied Innovation* 2 (2016), pp. 6–10.

[15] C. Dannen. *Introducing Ethereum and Solidity*. Springer, 2017. ISBN: 1484225341.

[16] M. Dapp, D. Balta, and H. Krcmar. *Blockchain - Disruption der Verwaltung? Eine Technologie zur Neugestaltung der Verwaltungsprozesse*. 2017. DOI: 10.13140/RG.2.2.31889.12644.

[17] P. B. Darwin and P. Kozlowski. *AngularJS web application development*. Packt Publ., 2013. ISBN: 1782161821.

[18] J. Deacon. "Model-view-controller (mvc) architecture." In: *Online][Citado em: 10 de março de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf* (2009).

[19] *Distributed Hash Tables*. https://blog.keep.network/distributed-hash-tables-49721094403d. Accessed: 2018-05-17.

[20] *Docker - Build, Ship, and Run Any App, Anywhere*. https://www.docker.com/. Accessed: 2018-05-18.

[21] *Double-Spending*. https://www.investopedia.com/terms/d/doublespending.asp. Accessed: 2018-05-14.

[22] *Ethereum Project*. https://ethereun.org. Accessed: 2018-05-16.

[23] U. Feige, A. Fiat, and A. Shamir. "Zero-knowledge proofs of identity." In: *Journal of cryptology* 1.2 (1988), pp. 77–94. ISSN: 0933-2790.

[24] S. L. Garfinkel. "Public key cryptography." In: *Computer* 29.6 (1996), pp. 101–104. ISSN: 0018-9162.

[25] C. Gentry. *A fully homomorphic encryption scheme*. Stanford University, 2009. ISBN: 1109444508.

[26] C. Gentry, S. Halevi, and N. P. Smart. "Homomorphic evaluation of the AES circuit." In: *Advances in cryptologycrypto 2012*. Springer, 2012, pp. 850–867.

[27] D. Harz and M. Boman. "The Scalability of Trustless Trust." In: *arXiv preprint arXiv:1801.09535* (2018).

[28] A. A. Helal, A. A. Heddaya, and B. B. Bhargava. *Replication techniques in distributed systems*. Vol. 4. Springer Science & Business Media, 2006. ISBN: 0306477963.

[29] *How do Digital Signatures Work? A Look Behind the Scenes*. Accessed: 2018-05-20. 2015.

[30] *HTTP is obsolete. It's time for the distributed, permanent web*. `https://ipfs.io/ipfs/QmNhFJjGcMPqpuYfxL62VVB9528NXqDNMFXiqN5bgFYiZ1/its-time-for-the-permanent-web.html`. Accessed: 2018-05-12.

[31] "iExec - Blockchain-Based Decentralized Cloud Computing." In: ().

[32] P. B. Kruchten. "The 4+ 1 view model of architecture." In: *IEEE software* 12.6 (1995), pp. 42–50. ISSN: 0740-7459.

[33] L. Lamport. "Paxos made simple." In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[34] L. Lamport, R. Shostak, and M. Pease. "The Byzantine generals problem." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982), pp. 382–401. ISSN: 0164-0925.

[35] *Litecoin - Open source P2P digital currency*. `https://litecoin.org/`. Accessed: 2018-05-17.

[36] T. McConaghy, R. Marques, A. Müller, D. De Jonghe, T. McConaghy, G. McMullen, R. Henderson, S. Bellemare, and A. Granzotto. "BigchainDB: a scalable blockchain database." In: *white paper, BigChainDB* (2016).

[37] *Merkle Tree Introduction*. `https://medium.com/@evankozliner/merkle-tree-introduction-4c44250e2da7`. Accessed: 2018-05-12.

[38] *MetaMask*. `https://metamask.io`. Accessed: 2018-05-10.

[39] *Mocha - the fun, simple, flexible JavaScript test framework*. `https://mochajs.org`. Accessed: 2018-05-16.

[40] S. Nakamoto. "Bitcoin: A peer-to-peer electronic cash system." In: (2008).

[41] *Node.js*. `https://nodejs.org/`. Accessed: 2018-05-16.

[42] M. Pease, R. Shostak, and L. Lamport. "Reaching agreement in the presence of faults." In: *Journal of the ACM (JACM)* 27.2 (1980), pp. 228–234. ISSN: 0004-5411.

[43] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee. "A design science research methodology for information systems research." In: *Journal of management information systems* 24.3 (2007), pp. 45–77. ISSN: 0742-1222.

[44] S. Raval. *Decentralized Applications: Harnessing Bitcoin's Blockchain Technology*. " O'Reilly Media, Inc.", 2016. ISBN: 1491924527.

[45] A. Reidt, M. Duchon, and H. Krcmar. "Referenzarchitektur eines Ressourcen-Cockpits zur Unterstützung der Instandhaltung." In: *aw&I Report* 1 (2017), pp. 43–60. ISSN: 2566-8633.

[46] M. A. Sharma and E. Ajay. "Chronic Kidney Disease Detection by Analyzing Medical Datasets in Weka." In: *International Journal of Computer Application* 6.4 (2016). ISSN: 2250-1797.

[47] D. Siegel. "Understanding the DAO attack." In: *Web. http://www. coindesk. com/understanding-dao-hack-journalists* (2016).

[48] N. P. Smart and F. Vercauteren. "Fully homomorphic encryption with relatively small key and ciphertext sizes." In: *International Workshop on Public Key Cryptography*. Springer, pp. 420–443.

[49] M. Swan. *Blockchain: Blueprint for a new economy*. " O'Reilly Media, Inc.", 2015. ISBN: 1491920475.

[50] N. Szabo. "Smart contracts: building blocks for digital markets." In: *EXTROPY: The Journal of Transhumanist Thought,(16)* (1996).

[51] A. S. Tanenbaum and M. Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007. ISBN: 0132392275.

[52] J. Teutsch and C. ReitwieSSner. "A scalable verification solution for blockchains." In: *:(2017). url: https://people. cs. uchicago. edu/ teutsch/papers/truebit. pdf* (2017).

[53] *The Big (Data) Problem With Machine Learning*. `http://www.digitalistmag.com/digital-economy/2017/05/11/big-data-problem-with-machine-learning-05084700`. Accessed: 2018-05-16. 2017.

[54] *The Golem Project: crowdfunding whitepaper.* `https://golem.network/doc/Golemwhitepaper.pdf`. Accessed: 2018-05-17.

[55] *Truffle Suite - Your Ethereum Swiss Army Knife*. `http://truffleframework.com`. Accessed: 2018-05-16.

[56] *Upgradable Solidity Contract Design Rocket Pool*. `https://medium.com/rocket-pool/upgradable-solidity-contract-design-54789205276d`. Accessed: 2018-05-10.

[57] L. Walsh, V. Akhmechet, and M. Glukhovsky. "Rethinkdb-rethinking database storage." In: (2009).

[58] *What is Blockchain Technology?* `https://www.coindesk.com/information/what-is-blockchain-technology/`. Accessed: 2018-05-14.

[59] *What Is Hashing? Under The Hood Of Blockchain*. `https://blockgeeks.com/guides/what-is-hashing/`. Accessed: 2018-05-15. 1969.

[60] G. Wood. "Ethereum: A secure decentralised generalised transaction ledger." In: *Ethereum project yellow paper* 151 (2014), pp. 1–32.

A Appendix

```
1  pragma solidity ^0.4.19;
2  /*
3  * The SoftwareRegistry contract.
4  * Store software resources.
5  */
6  contract SoftwareRegistry {
7
8      bytes32[] softwareIdentifiers;
9
10     // Software data structure
11     struct SoftwareInfo {
12         string bcdbTxID; // bigchainDB transaction id with the metadata
               of the software
13         string checksum; // hash of the metadata
14         uint cost; // cost of using the software
15         address owner; // address of the software provider
16     }
17
18     //softwareId => SoftwareInfo
19     mapping(bytes32 => SoftwareInfo) idToSoftwareInfo;
20     mapping(address => bool) accessAllowed;
21
22     function SoftwareRegistry() public {
23         accessAllowed[msg.sender] = true;
24     }
25
26     // allow an address to access contract's methods
27     function allowAccess(address _address) onlyIfAllowed public {
28         accessAllowed[_address] = true;
29     }
30
31     // RegistryManager contract calls this method to add a new software
```

```
32      function addSoftware(bytes32 _id, string _bcdbTxID, string _checksum,
            uint _cost, address _owner) onlyIfAllowed public returns(bool
            success) {
33          softwareIdentifiers.push(_id);
34
35          idToSoftwareInfo[_id].bcdbTxID = _bcdbTxID;
36          idToSoftwareInfo[_id].checksum = _checksum;
37          idToSoftwareInfo[_id].cost = _cost;
38          idToSoftwareInfo[_id].owner = _owner;
39          return true;
40      }
41
42      modifier onlyIfAllowed () {
43          require(accessAllowed[msg.sender] == true);
44          _;
45      }
46
47      // returns the identifiers of the software
48      function getSoftwareIDs() public view returns(bytes32[]) {
49          return softwareIdentifiers;
50      }
51
52      // returns the information about the given software
53      function getSoftwareByID(bytes32 _id) public view returns(string
            _bcdbTxID, string checksum, uint cost, address owner) {
54          return (idToSoftwareInfo[_id].bcdbTxID, idToSoftwareInfo[_id].
                checksum, idToSoftwareInfo[_id].cost, idToSoftwareInfo[_id].
                owner);
55      }
56
57      // return specific information about the given software
58      function getPaymentInfo(bytes32 _id) public view returns(uint _cost,
            address _owner) {
59          return (idToSoftwareInfo[_id].cost, idToSoftwareInfo[_id].owner);
60      }
61  }
```

```
1   pragma solidity ^0.4.19;
2   /*
```

```
3  * The DatasetRegistryContract.
4  * Store dataset resources.
5  */
6  contract DatasetRegistry {
7
8      bytes32[] datasetIdentifiers;
9
10     // Dataset data structure
11     struct DatasetInfo {
12         string bcdbTxID; // bigchainDB transaction id with the metadata
               of the dataset
13         string checksum; // hash of the metadata
14         uint cost; // cost of using the dataset
15         address owner; // address of the dataset provider
16     }
17
18     mapping(bytes32 => DatasetInfo) idToDatasetRegistry;
19     mapping(address => bool) accessAllowed;
20
21     function DatasetRegistry() public {
22         accessAllowed[msg.sender] = true;
23     }
24
25     // allow address to access contract's methods
26     function allowAccess(address _address) onlyIfAllowed public {
27         accessAllowed[_address] = true;
28     }
29
30     // RegistryManager contract calls this method to add a new dataset
31     function addDataset(bytes32 _id, string _bcdbTxID, string _checksum,
           uint _cost, address _owner) onlyIfAllowed public returns (bool
           success) {
32         datasetIdentifiers.push(_id);
33
34         idToDatasetRegistry[_id].bcdbTxID = _bcdbTxID;
35         idToDatasetRegistry[_id].checksum = _checksum;
36         idToDatasetRegistry[_id].cost = _cost;
37         idToDatasetRegistry[_id].owner = _owner;
38         return true;
```

```
39      }
40
41      modifier onlyIfAllowed() {
42          require(accessAllowed[msg.sender] == true);
43          _;
44      }
45
46      // returns the identifiers of the datasets
47      function getDatasetIDs() public view returns (bytes32[]) {
48          return datasetIdentifiers;
49      }
50
51      // returns the information about the given dataset
52      function getDatasetByID(bytes32 _id) public view returns (string
            _bcdbTxID, string checksum, uint cost, address owner) {
53          return (idToDatasetRegistry[_id].bcdbTxID, idToDatasetRegistry[
                _id].checksum, idToDatasetRegistry[_id].cost,
                idToDatasetRegistry[_id].owner);
54      }
55
56      // return specific information about the given dataset
57      function getPaymentInfo(bytes32 _id) public view returns (uint _cost,
             address _owner) {
58          return (idToDatasetRegistry[_id].cost, idToDatasetRegistry[_id].
                owner);
59      }
60  }
```

```
1   pragma solidity ^0.4.19;
2   /*
3   * The ContainerRegistryContract.
4   * Store container resources.
5   */
6   contract ContainerRegistry {
7
8       bytes32[] containerIdentifiers;
9
10      // Container data structure
11      struct ContainerInfo {
```

```
12          string bcdbTxID; // bigchainDB transaction id with the metadata
                of the container
13          string checksum; // hash of the metadata
14          uint cost; // cost of using the container
15          address owner; // address of the container provider
16      }
17
18      mapping(bytes32 => ContainerInfo) idToContainerInfo;
19      mapping(address => bool) accessAllowed;
20
21
22      function ContainerRegistry() public {
23          accessAllowed[msg.sender] = true;
24      }
25
26      // allow an address to access contract's methods
27      function allowAccess(address _address) onlyIfAllowed public {
28          accessAllowed[_address] = true;
29      }
30
31      // RegistryManager contract calls this method to add a new container
32      function addContainer(bytes32 _id, string _bcdbTxID, string _checksum
            , uint _cost, address _owner) onlyIfAllowed public returns(bool
            success) {
33          containerIdentifiers.push(_id);
34
35          idToContainerInfo[_id].bcdbTxID = _bcdbTxID;
36          idToContainerInfo[_id].checksum = _checksum;
37          idToContainerInfo[_id].cost = _cost;
38          idToContainerInfo[_id].owner = _owner;
39          return true;
40      }
41
42      modifier onlyIfAllowed () {
43          require(accessAllowed[msg.sender] == true);
44          _;
45      }
46
47      // returns the identifiers of the containers
```

```
48      function getContainerIDs() public view returns(bytes32[]) {
49          return containerIdentifiers;
50      }
51
52      // returns the information about the given container
53      function getContainerByID(bytes32 _id) public view returns(string
            _bcdbTxID, string checksum, uint cost, address owner) {
54          return (idToContainerInfo[_id].bcdbTxID, idToContainerInfo[_id].
                checksum, idToContainerInfo[_id].cost, idToContainerInfo[_id].
                owner);
55      }
56
57      // return specific information about the given container
58      function getPaymentInfo(bytes32 _id) public view returns(uint _cost,
            address _owner) {
59          return (idToContainerInfo[_id].cost, idToContainerInfo[_id].owner
                );
60      }
61  }
```

```
1   pragma solidity ^0.4.19;
2   /*
3   * The RegistrytManager contract.
4   * Contains the logic for the registry contracts.
5   */
6   import "./SoftwareRegistry.sol";
7   import "./DatasetRegistry.sol";
8   import "./ContainerRegistry.sol";
9
10  contract RegistryManager {
11      SoftwareRegistry softwareRegistry; // address of SoftwareRegistry
            contract
12      DatasetRegistry datasetRegistry; // address of DatasetRegistry
            contract
13      ContainerRegistry containerRegistry; // address of ContainerRegistry
            contract
14
15      // store registry contract addresses
16      function RegistryManager(address _datasetRegistryAddress, address
```

```
        _softwareRegistryAddress, address _containerRegistryAddress)
        public{
17          softwareRegistry = SoftwareRegistry(_softwareRegistryAddress);
18          datasetRegistry = DatasetRegistry(_datasetRegistryAddress);
19          containerRegistry = ContainerRegistry(_containerRegistryAddress);
20      }
21
22      // call SoftwareRegistry contract to add a new dataset
23      function addSoftwareInfo(string _bcdbTxID, string _checksum, uint
            _cost) public returns(bool success) {
24          bytes32 id = keccak256(_bcdbTxID, _checksum, now); // compute new
                software identifier
25          softwareRegistry.addSoftware(id, _bcdbTxID, _checksum, _cost, msg.
                sender);
26          return true;
27      }
28
29      // call DatasetRegistry contract to add a new dataset
30      function addDatasetInfo(string _bcdbTxID, string _checksum, uint
            _cost) public returns(bool success) {
31          bytes32 id = keccak256(_bcdbTxID, _checksum, now); // compute new
                dataset identifier
32          datasetRegistry.addDataset(id, _bcdbTxID, _checksum, _cost, msg.
                sender);
33          return true;
34      }
35
36      // call ContainerRegistry contract to add a new container
37      function addContainerInfo(string _bcdbTxID, string _checksum, uint
            _cost) public returns(bool success) {
38          bytes32 id = keccak256(_bcdbTxID, _checksum, now); // compute new
                container identifier
39          containerRegistry.addContainer(id, _bcdbTxID, _checksum, _cost,
                msg.sender);
40          return true;
41      }
42  }
```

```
1  pragma solidity ^0.4.19;
```

```
2  /*
3  * The ComputationRegistry contract.
4  * Store computations.
5  */
6  contract ComputationRegistry {
7
8      bytes32[] public computationIDs;
9
10     // The computation can have 3 states
11     enum ComputationStatus {PLACED, SUCCEEDED, CANCELLED}
12
13     // Computation data structure
14     struct ComputationInfo {
15         bytes32 computationID; // identifier of the computation
16         bytes32 containerID; // identifier of the container that will be
               used in the computation
17         bytes32 datasetID; // identifier of the dataset that will be used
                in the computation
18         bytes32 softwareID; // identifier of the software that will be
               used in the computation
19         address owner; // address of the recipient of the computation
               output
20         uint amount; // total amount paid for the computation
21         ComputationStatus status; // status of the computation
22         bytes32 userPubKeyIpfsHash; // IPFS address containing the
               cryptographic public key of the owner
23     }
24
25     mapping(bytes32 => ComputationInfo) public idToComputationInfo;
26     mapping(address => bool) accessAllowed;
27
28     function ComputationRegistry() public {
29         accessAllowed[msg.sender] = true;
30     }
31
32     // allow an address to access contract's methods
33     function allowAccess(address _address) onlyIfAllowed public {
34         accessAllowed[_address] = true;
35     }
```

```
36
37      // called by an allowed address to add a new computation
38      function addComputation(bytes32 _id, bytes32 _userPubKeyIpfsHash,
            bytes32 _datasetID, bytes32 _softwareID, bytes32 _containerID,
            uint _amount, address _owner) onlyIfAllowed public returns (bool
            res) {
39
40          idToComputationInfo[_id].computationID = _id;
41          idToComputationInfo[_id].containerID = _containerID;
42          idToComputationInfo[_id].userPubKeyIpfsHash = _userPubKeyIpfsHash
                ;
43          idToComputationInfo[_id].datasetID = _datasetID;
44          idToComputationInfo[_id].softwareID = _softwareID;
45          idToComputationInfo[_id].owner = _owner;
46          idToComputationInfo[_id].amount = _amount;
47          idToComputationInfo[_id].status = ComputationStatus.PLACED; //
                set initial status to PLACED
48          computationIDs.push(_id);
49
50          return true;
51      }
52
53      // transfer funds to the corresponding providers after a successful
            computation
54      function computationSuccess(bytes32 _computationID) onlyIfPlaced(
            _computationID) onlyIfAllowed public returns (bool success) {
55          idToComputationInfo[_computationID].status = ComputationStatus.
                SUCCEEDED;
56          return true;
57      }
58
59      // cancel a placed computation
60      function computationFailure(bytes32 _computationID) onlyIfPlaced(
            _computationID) onlyIfAllowed public returns (bool success) {
61          idToComputationInfo[_computationID].status = ComputationStatus.
                CANCELLED;
62          return true;
63      }
64
```

```
65      modifier onlyIfAllowed () {
66          require(accessAllowed[msg.sender] == true);
67          _;
68      }
69
70      modifier onlyIfPlaced(bytes32 computationID) {
71          require(idToComputationInfo[computationID].status ==
                ComputationStatus.PLACED);
72          _;
73      }
74
75      // return information about the given computation
76      function getComputationInfo(bytes32 _computationID) public constant
            returns(bytes32 _dsID, bytes32 _swID, bytes32 _contID) {
77          return(idToComputationInfo[_computationID].datasetID,
                idToComputationInfo[_computationID].softwareID,
                idToComputationInfo[_computationID].containerID);
78      }
79
80      // return specific information about the given computation
81      function getComputationPaymentInfo(bytes32 _computationID) public
            constant returns(address owner, uint amount) {
82          return(idToComputationInfo[_computationID].owner,
                idToComputationInfo[_computationID].amount);
83      }
84  }
```

```
1   pragma solidity ^0.4.19;
2   /*
3   * The ResultRegistry contract.
4   * Store computation results.
5   */
6   contract ResultRegistry {
7
8       // Result data structure
9       struct ResultInfo {
10          bytes32 resultData; // IPFS address of computation output
11          bytes32 password; // IPFS address of one-time-password to decrypt
                the result
```

```
12      }
13
14      // array of results
15      struct ResultStruct {
16          ResultInfo[] results;
17          uint count;
18      }
19
20      mapping(address => ResultStruct) addressToResult;
21      mapping(address => bool) accessAllowed;
22
23      function ResultRegistry() public {
24          accessAllowed[msg.sender] = true;
25      }
26
27      // allow an address to access contract's methods
28      function allowAccess(address _address) onlyIfAllowed public {
29          accessAllowed[_address] = true;
30      }
31
32      // ResultManager contract calls this method to add a new result
33      function addResult(address _owner, bytes32 _newResultData, bytes32
            _newPassword) onlyIfAllowed public returns(bool){
34
35          addressToResult[_owner].results.push(ResultInfo(_newResultData,
                _newPassword));
36          addressToResult[_owner].count = addressToResult[_owner].count +
                1;
37          return true;
38      }
39
40      modifier onlyIfAllowed () {
41          require(accessAllowed[msg.sender] == true);
42          _;
43      }
44
45      // returns the a specific result from the given address
46      function getResult(address owner, uint index) public view returns(
            bytes32, bytes32) {
```

```
47          require(msg.sender == owner);
48          return (addressToResult[owner].results[index].resultData,
                addressToResult[owner].results[index].password);
49      }
50
51      // returns the number of results of the given address
52      function getResultCount(address owner) public view returns(uint) {
53          require(msg.sender == owner);
54          return addressToResult[owner].count;
55      }
56  }
```

```
1   pragma solidity ^0.4.19;
2   /*
3   * The ComputationManager contract.
4   * Contains the logic for the ComputationRegistry contract.
5   */
6   import "./ComputationRegistry.sol";
7   import "./DatasetRegistry.sol";
8   import "./SoftwareRegistry.sol";
9   import "./ContainerRegistry.sol";
10
11  contract ComputationManager {
12
13      address oracleAddress; // address of trusted third-party
14
15      // event for new computations
16      event ComputationAdded (
17          bytes32 indexed computationID, // computation identifier
18          bytes32 indexed containerID, // container identifier used by the
                computation
19          bytes32 indexed datasetID, // dataset identifier used by the
                computation
20          bytes32 softwareID, // software identifier used by the
                computation
21          bytes32 userPubKeyIpfsHash // IPFS address of the recipient's
                public key
22      );
23
```

```
24      ComputationRegistry computationRegistry; // address of
            ComputationRegistry contract
25      DatasetRegistry datasetRegistry; // address of DatasetRegistry
            contract
26      SoftwareRegistry softwareRegistry; // address of SoftwareRegistry
            contract
27      ContainerRegistry containerRegistry; // address of ContainerRegistry
            contract
28
29      // store registry contract and oracle addresses
30      function ComputationManager(address _computationRegistryAddress,
            address _datasetRegistryAddress, address _softwareRegistryAdderss,
             address _containerRegistryAddress, address _oracleAddress)
            public {
31          computationRegistry = ComputationRegistry(
                _computationRegistryAddress);
32          datasetRegistry = DatasetRegistry(_datasetRegistryAddress);
33          softwareRegistry = SoftwareRegistry(_softwareRegistryAdderss);
34          containerRegistry = ContainerRegistry(_containerRegistryAddress);
35          oracleAddress = _oracleAddress;
36      }
37
38      // call ComputationRegistry contract to add a new computation
39      function addComputationInfo(bytes32 _uPubKeyIpfsHash, bytes32
            _datasetID, bytes32 _softwareID, bytes32 _containerID) public
            payable returns(bool success) {
40          bytes32 id = keccak256(_datasetID, _softwareID, _containerID, now
                ); // compute new computation identifier
41
42          bool stored = computationRegistry.addComputation(id,
                _uPubKeyIpfsHash, _datasetID, _softwareID, _containerID, msg.
                value, msg.sender);
43          if(stored) {
44              ComputationAdded(id, _containerID, _datasetID, _softwareID,
                    _uPubKeyIpfsHash); // trigger event for success
45              return true;
46          }
47          return false;
48      }
```

```
49
50      // called by the oracle to transfer funds to the providers after a
            successful computation
51      function computationSucceed(bytes32 _computationID) onlyOracle public
             returns(bool success) {
52          bool res = computationRegistry.computationSuccess(_computationID)
                ;
53          if(res) {
54              var (ds_id, sw_id, cont_id) = computationRegistry.
                    getComputationInfo(_computationID);
55
56              var (ds_cost, ds_owner) = getDatasetPaymentInfo(ds_id);
57
58              var (sw_cost, sw_owner) = getSoftwarePaymentInfo(sw_id);
59
60              var (cont_cost, cont_owner) = getContainerPaymentInfo(cont_id)
                    ;
61
62              ds_owner.transfer(ds_cost);
63              sw_owner.transfer(sw_cost);
64              cont_owner.transfer(cont_cost);
65              return true;
66          }
67          return false;
68      }
69
70      // called by the oracle to return funds to the recipient of the
            computation output after a failed computation
71      function computationFailed(bytes32 _computationID) onlyOracle public
            returns(bool success) {
72          bool res = computationRegistry.computationFailure(_computationID)
                ;
73          if(res) {
74              var(owner, amount) = computationRegistry.
                    getComputationPaymentInfo(_computationID);
75              owner.transfer(amount);
76              return true;
77          }
78          return false;
```

```
79      }
80
81      modifier onlyOracle {
82          require(msg.sender == oracleAddress);
83          _;
84      }
85
86      // call DatasetRegistry contract to get information about the given
            dataset
87      function getDatasetPaymentInfo(bytes32 _datasetID) private constant
            returns (uint cost, address owner) {
88          var (ds_cost, ds_owner) = datasetRegistry.getPaymentInfo(
                _datasetID);
89          return (ds_cost, ds_owner);
90      }
91
92      // call SoftwareRegistry contract to get information about the given
            software
93      function getSoftwarePaymentInfo(bytes32 _softwareID) private constant
             returns (uint cost, address owner) {
94          var (sw_cost, sw_owner) = softwareRegistry.getPaymentInfo(
                _softwareID);
95          return (sw_cost, sw_owner);
96      }
97
98      // call ContainerRegistry contract to get information about the given
             container
99      function getContainerPaymentInfo(bytes32 _containerID) private
            constant returns (uint cost, address owner) {
100         var (cont_cost, cont_owner) = containerRegistry.getPaymentInfo(
                _containerID);
101         return (cont_cost, cont_owner);
102     }
103 }
```

```
1  pragma solidity ^0.4.19;
2  /*
3  * The ResultManager contract.
4  * Contains the logic for the ResultRegistry contract.
```

```
5  */
6  import "./ResultRegistry.sol";
7
8  contract ResultManager {
9
10     address oracleAddress; // address of trusted third-party
11
12     // event for new result
13     event ResultAdded (
14         bytes32 resultData, // IPFS address of the computation output
15         bytes32 password, // IPFS address of the one-time password to
               decrypt the result
16         address owner // address of the recipient
17     );
18
19     // event in case of a failure
20     event ResultError(bytes32 errorMsg);
21
22     ResultRegistry resultRegistry; // address of the ResultRegistry
           contract
23
24     function ResultManager(address _resultRegistryAddress, address
           _oracle) public {
25         oracleAddress = _oracle;
26         resultRegistry = ResultRegistry(_resultRegistryAddress);
27     }
28
29     // call ResultRegistry contract to add a new result
30     function addResultInfo(address _owner, bytes32 _newResultData,
           bytes32 _newPassword) onlyOracle public returns(bool success) {
31         bool res = resultRegistry.addResult(_owner, _newResultData,
               _newPassword);
32         if(res) {
33             ResultAdded(_newResultData, _newPassword, _owner); // trigger
                   event for success
34             return true;
35         }
36         return false;
37     }
```

```
38
39      // trigger event in case of error in the execution
40      function resultError(bytes32 _errorMsg) onlyOracle public {
41          ResultError(_errorMsg); // trigger event fo error
42      }
43
44      modifier onlyOracle {
45          require(msg.sender == oracleAddress);
46          _;
47      }
48  }
```