POLITECNICO DI MILANO
Master of Science in Computer Science and Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria

Quality-Aware Transaction Validation in Blockchains

Supervisor: Prof.ssa Cinzia Cappiello
Co-supervisor: Prof. Florian Daniel

M.Sc. Thesis by:
Lorenzo Maria Bonelli, matriculation number 876070

Academic Year 2018-2019

Abstract

The first blockchain, Bitcoin, was born in 2009 as an infrastructure for managing a cryptocurrency, but the possibilities offered by this technology are many more. With the introduction of smart contracts, pieces of code, which run on the blockchain infrastructure inheriting its key features, such as not being subject to any downtime, diddling or third party control, the blockchains became the perfect environment for developing applications with special requirements. Ethereum has been the first blockchain to introduce Touring-complete smart contracts and several businesses benefit from this innovation, from financial services to IoT, from supply chain management to asset management. When the blockchain is used to implement complex business processes, the quality of the stored data becomes critical: data are a valuable asset just if they are qualitative, which means that they fit the specific case, and since they are used to drive the application logic, to take critical decisions and to trigger actions, assessing their quality before storing them is crucial; while this is valid in any applications, it is even more important in the context of blockchains, where the data are permanent and immutable after being stored.

While the blockchain technology provides some quality guarantees for the data stored, such as anti-tampering, thanks to the hashes linking blocks, and non-repudiation, thanks to the use of cryptographic signatures, the payload of transactions is not subject to any analysis or approval by the standard consensus protocol and analyzing these data requires implementing data quality controls as specific smart contracts. Writing code to run on the blockchain poses some strict constraints, for example smart contracts implement passive application logic, cannot access any data stored off-chain and every operation costs money, so the purpose of this work is studying how data quality controls can be efficiently implemented to enrich existing code, evaluating the impact they have on both cost and performance. Considering two case studies, this thesis first proposes two smart contracts to perform data quality controls and, after performing a cost-benefits analysis, decides whether it's convenient or not to add those checks.

Sommario

La prima blockchain, Bitcoin, è nata nel 2009 come un'infrastruttura per la gestione di una criptovaluta, cioè una moneta digitale, ma le possibilità offerte da questa tecnologia sono molto maggiori. Con l'introduzione dei cosiddetti *smart contracts*, cioè codice che viene eseguito sulla blockchain ereditandone le proprietà principali, come il fatto di non essere soggetta a guasti che causino periodi di inattività, a manomissioni o al controllo da parte di una singola entità, la blockchain è diventata un ambiente ideale per sviluppare applicazioni con specifici requisiti.
Ethereum è stata la prima blockchain ad offrire un linguaggio di programmazione Touring equivalente e molti settori hanno ricevuto benefici da questa innovazione: dai servizi finanziari all'ioT, dalla gestione della supply chain a quella dei beni. Quando la blockchain viene usata per implementari complessi processi di business, la qualità dei dati salvati diventa un aspetto critico: i dati sono una risorsa preziosa se sono di qualità, cioè adatti al caso specifico, e dal momento che vengono usati per guidare la logica dell'applicazione, per prendere decisioni critiche e per innescare azioni, valutare la loro qualità prima di immagazzinarli diventa cruciale; nonostante questo sia valido in ogni applicazione, assume un'importanza ancora maggiore nel contesto delle blockchains, dove i dati salvati sono permanenti e immutabili. Nonostante la blockchain offra, per sua stessa natura, alcune garanzie sulla qualità dei dati salvati, come il fatto di non essere manomettibili, grazie agli hashes che uniscono i blocchi, e la non repudiabilità, grazie all'uso di firme crittografate, il *payload* delle transazioni non è soggetto ad alcun tipo di controllo o approvazione da parte del protocollo di consenso, per cui analizzare questi dati richieste l'implementazione di appositi *smart contracts* che facciano controlli sulla loro qualità. Scrivere codice da eseguire sulla blockchain è però un'attività molto complessa, soggetta ad alcuni limiti importanti: gli *smart contracts* implementano solo logica passiva, non possono accedere a dati salvati al di fuori della blockchain e ogni singola operazione ha un costo economico, dunque lo scopo di questo lavoro è studiare come dei controlli di qualità dei dati possano essere implementati in modo efficiente all'interno di applicazioni per la blockchain, valutando l'impatto che hanno su costi e performance. Utilizzando due casi di studio, questa tesi presenta due *smart contracts* per fare controlli sulla qualità dei dati e, dopo aver fatto un'analisi di costi e benefici, si prefigge di stabilire se sia conveniente o meno implementarli.

Contents

Chapter 1

Introduction

The use of blockchain technology is no more limited to virtual currency transactions, since programmable blockchains offer an infrastructure to build applications on by writing code that leverages the key principles of this technology. The programs deployed on this infrastructure are called *smart contracts* and they are complete programs, implementing their own logic and dealing with data, whose quality becomes essential in complex business scenarios. In a context where data are stored in a permanent and immutable way, being used to drive the application logic and to take critical decisions, assessing their quality before storing them is crucial.

1.1 Context: data quality controls in blockchain applications

Blockchains are one of the most innovative and game-changing technologies of the last decade. A *blockchain* is a shared, distributed ledger, namely a log of transactions providing persistency and verifiability of transactions themselves [20]. In other terms, blockchains can be considered as decentralized databases offering some very appealing properties, such as the immutability of stored transactions and the possibility for participants to establish trust without a third entity intermediating [25].

While the first blockchain, Bitcoin, was born for managing and transfering a cryptocurrency, the introduction of smart contracts has pushed the use of this technology one step further. They are pieces of code which are deployed on the blockchain by a transaction and create an instance of the given code persistently running in the blockchain's environment.

Ethereum, specifically, can be considered as the largest innovation in this field after Bitcoin's birth: launched in 2014, it is a public blockchain-based distributed computing platform, providing a decentralised virtual machine, known as the Ethereum Virtual Machine (EVM), as runtime environment to execute Touring-complete smart contracts [25].
This innovation made Ethereum very appealing for many applications, because running code on the blockchain can provide several benefits: it is not subject to any downtime, diddling or third party control and it guarantees some key properties such as data persistence and anti tampering.
However, when the outcome of a smart contract is written on the blockchain, it is not subject to any check by the platform, so in order to avoid to permanently store non-qualitative data, quality controls should be implemented using additional code running on-chain. This poses some relevant drawbacks because smart contracts just implement passive application logic, they cannot access any data stored *off-chain* and any operation costs money.

This thesis, starting from the theoretic approach to data quality assessment on the blockchain reported in 4.1, studies how data quality controls can be practically implemented on Ethereum and tests the approach by considering two realistic scenarios and providing suitable smart contracts for data quality assessment. In order to cover as many cases as possible, we consider an application where data are streamed from a sensor and another where the data are entered and submitted as a single transaction by a human entity. On top of that, we'll evaluate different metrics for each case, so that the proposed smart contracts will cover different level of complexity.
The proposed code will be run in order to measure the cost, in terms of money and time, for deploying and invoking the data quality controls to evaluate whether it's reasonable or not to implement similar controls in blockchain applications.

1.2 Scenario and Problem Statement

The main application of blockchain, at least at its dawn, was the exchange of a digital asset, the Bitcoin cryptocurrency, and each transaction represented a transfer of the asset from one address to another. An address is a 160-bit univocal code used for identifying accounts [26]. In this context, a record of the transaction is written on the blockchain as a permanent and immutable information and the cryptocurrency balance of the two addresses is updated accordingly.

Once business processes are moved to the blockchain, however, the necessity to perform operations on the data, including assessing their quality, arises. This is because data become the core asset the applications are build on: they are used for implementing strategies, taking decisions and triggering actions, so using non-qualitative data can lead to several losses, from potential opportunities to economical ones.
The possible business applications running on the blockchain are virtually infinite, so we decide to consider two real-life scenarios requiring extensive data quality controls, so that we can study several different situations, from easier one, where the metrics are assessed by comparing the received value with a numeric threshold, to the most difficult ones, which need to access data stored off-chain.

1.2.1 Scenario 1, Drugs Transportation

The transportation of medicinal products for human use, in the countries of the EU, is regulated by the GDP Regulation C343 01. A logistic company, to be allowed to transport these kinds of goods, should strictly comply to the guidelines provided by the document. The most critical aspect during transportation is the temperature; goods should be carried in a thermally isolated truck and the temperature should be periodically checked to make sure that the products' quality is not altered.

The GDP Regulation cited above requires carriers to collect data during all the phases of transportation, to store them in a secure way and to guarantee that, once they are stored:

- They cannot be modified or altered in any way;
- They can be accessible at any time;
- They can be available for at least five years from their collection;
- They can be produced to customers to prove that all the best practices have been followed during the process.

A logistic company with medical products transportation as its core business would like to benefit from the new possibilities offered by the blockchain technology by implementing the quality monitoring system, currently running on a local server in their office, on the Ethereum blockchain. In this way, the data collected from their trucks and warehouses can not be lost or tampered; by doing this, the company can protect itself from possible

accusations of bad transportation conditions, avoid that any employee can hide negligence and also make sure that the data are always available upon request.

All the company's trucks and warehouses are equipped with sensors measuring the temperature, either inside the truck or in different areas of the warehouse, one time per minute and transmitting this information to a base station located in the company's main office. Since transmitting and writing on the blockchain a transaction every minute would be very expensive, the sensor collects the measurements and transmits a block of 60 temperature records one time per hour; this granularity is fine enough for the application.

Before storing the data collected permanently on the blockchain, some data quality controls are required in order to make sure that the drugs have been properly preserved.

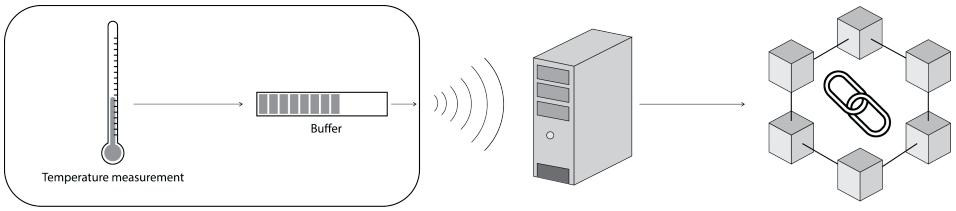

Figure 1.1: General scheme of scenario 1

1.2.2 Scenario 2, Drugs Prescription:

A general doctor has up to tens of hundreds of patients and prescribes tens of medications a day. Each patient has its clinical background and (possibly) a list of ongoing treatments, so it can be difficult for the doctor to know if a new prescription can cause any harm to him. This can be the case of a patient who is already taking another drug incompatible with the new prescription, but also of a patient who, in the past, had an allergic reaction to a drug of the same category.

To help doctors in avoiding errors in prescriptions, to protect them in case of false accusations, but also to prevent them from hiding negligence, we want to record prescriptions on a blockchain and to run some quality controls before writing the transaction (i.e. accepting the prescription).

Each new prescription consists in the patient's Social Security Number

(SSN) and a drug; the quality controls should check that the new record has all the requested fields, that both the patient's SSN and the Drug name exist and that the new prescription is compatible with the ongoing treatments by the patient (if any) and that there's no record of intolerances to similar drugs in his clinical background.

In order to keep only the ongoing treatments on-chain and to save space storage, which has a relevant cost on the blockchain, we want the smart contract to be able to delete prescriptions once they are no longer needed for the patient.

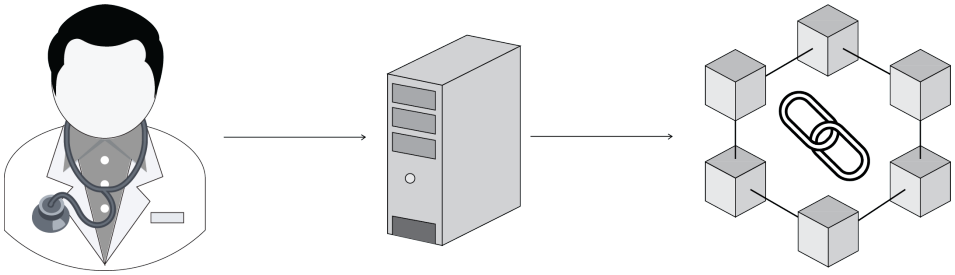

Figure 1.2: General scheme of scenario 2

1.3 Methodology

This thesis investigates whether it is reasonable and convenient to implement data quality controls for blockchain applications. We consider two real life case studies, described in section 1.2, and we implement some modular, reusable smart contracts to assess the quality of data which have to be stored permanently on the blockchain. The purpose is to offer smart contracts, which are as general and tunable as possible, so that they can be integrated with a very little effort into any business application similar to the proposed ones.

While code reusing is a key principle of any programming language, this assumes an even higher importance in blockchain programming for two main reasons: first of all, writing code is always an error-prone activity, but this is dramatically more relevant in this context because smart contracts may store Ether, the cryptocurrency associated to the Ethereum infrastructure,

so a code vulnerability can lead to an exploit resulting in the loss, or stealing, of a potentially massive amount of money in the form of cryptocurrencies. On top of that, every activity on the blockchain has a cost: each time an address sends a transaction, it has to pay an amount proportional to the complexity of the code to run, both in terms of logical operations and data storage, so the more instructions are in the code, the higher is the amount to pay. For this reason, keeping the code as simple as possible and avoid redundant code is the main goal when developing smart contracts and, since there are not many best practises to follow in this relatively new field, writing reusable code can be helpful for people who want to implement business application on the Ethereum platform.

When it gets to write reusable code, there are two possible ways to integrate it with upcoming applications: the first option is to present the contracts as a library, usually a Github repository, that other programmers can access and use when writing their code. The second option is to deploy the smart contracts on the Ethereum blockchain and to provide their addresses and their functions, so that anyone who needs to execute that code can just add a call to the given smart contract in their code, but this requires the application to be able to invoke the controls passing exactly the expected parameters, which may result a difficult task in some cases.

For help programmers in integrating our code into their application, we think that the best solution is providing our smart contract as code available online which anyone can take, modify and adapt to perfectly match the specific needs. After writing the smart contracts for assessing the relevant metrics for the two case studies, we will test our solution and the parameters we will consider are the costs in terms of money and time to decide if the cost of implementing data quality controls is proportional to the benefits brought. It is important to notice that some applications could not exist without data quality controls: for example the two case studies proposed would be totally useless if no data quality assessment is performed, so our study reduces to analyzing whether it is reasonable or not to deploy those application on the blockchain, but we'll expand our evaluation to decide whether it would be reasonable, for a business application where data quality controls are not mandatory, to implement them or not.

When programming smart contracts for Ethereum, the code is usually written in an high level language and then compiled into EVM bytecode which can be executed by the EVM. Although there exist some different high level languages for programming Ethereum smart contracts, this thesis will

focus on Solidity, which is the most widely used one, using the Remix IDE, a web application for writing, debugging, deploying and calling Ethereum smart contracts. The code, during development, will be deployed on the Ropsten test network, which is a testing environment working almost the same way as the Ethereum main network but where the cryptocurrency used is virtual, which means that deploying and invoking smart contracts is basically free. Once ready, the code will be tested and its performance will be evaluated thanks to special functions and the transaction logs, that carry several information regarding the transaction itself.

1.4 Contributions

The main contributions of this work are:

- an analysis of the limitations imposed by the blockchain technology when implementing Ethereum smart contracts;
- an innovative approach to allow programmers to enrich smart contracts with data quality assessment capabilities;
- the design and implementation of reusable smart contracts, namely programs running on the blockchain, for performing data quality controls on top of the Ethereum infrastructure;
- the performance evaluation of the written smart contracts, both in terms of cost and time overhead.

1.5 Structure of Thesis

This thesis is structured in seven chapters.
Chapter 2 provides the reader with a review of the state of the art, in order to place our work in the right context, starting from considering blockchain from a wider point of view and then focusing more on smart contracts and data quality.
Chapter 3 provides all the basic concepts needed in order to understand our work, highlighting the technological innovations that characterize the blockchain infrastructure. In the second part of the chapter we present the goals and requirements of our work.
Chapter 4 focuses on the approach adopted in our work, presenting the design decisions made and the architecture which has been used.

Chapter 5 contains the core aspects of our work, explaining which are the features we kept in mind while developing our solution.
Chapter 6 gets into details of our implementation: first it presents the technologies we used and then it explains how we practically implemented our work. The second section focuses on the evaluation, from its design to testing, ending with our considerations on the collected results.
Chapter 7 draws the conclusions of the thesis, highlights the limitations of our solution and pointing out which aspects of our work can be further developed by future work.

Chapter 2

State of the Art

This chapter analyzes the state of the art regarding blockchain programming, with a specific focus on data quality assessment. First of all we provide a short overview of blockchains in order to understand their strengths and weaknesses; afterwards, we point out the main reasons why people started programming smart contracts, expanding the possible uses of blockchains. After that, we present and analyze the most relevant studies and contributions related to the implementation of smart contracts running business applications and in the last section we introduce the problem of data quality assessment, explaining why it is important for smart contracts to implement quality controls.

2.1 Blockchain: early stage

The first blockchain ever implemented is Bitcoin. It was announced by a paper posted to a cryptography mailing list in 2008 by a pseudonymous person, or group of people, called Satoshi Nakamoto [11]. The new technology was described as a "system for electronic transactions without relying on trust". [20] This sentence contains the two key features of blockchain, pointing out the revolutionary aspect of it. First of all, the Bitcoin blockchain was designed as a platform for exchanging a cryptocurrency, Bitcoin, which is a virtual currency. The other fundamental aspect is that the platform enforces trust between the parties without counting on a central authority. This means that everything happening on the blockchain, namely each transaction, is non-centralized, so basically the money exchange between two parties is not regulated by any third entity, as bank do with physical currencies.

This aspect of the technology is what makes it suitable for virtually infinite applications other than money exchange, but as any new, game-changing technology it takes time to fully exploit it and it's a common opinion that just a small part of blockchain capabilities has been used so far [17].
Bitcoin theorization and implementation are based on several concepts belonging to programming, mathematics, game theory and economics, but there are some contributions which tried to explain this technology in a more simple way. As we already pointed out, the blockchain is a ledger of transactions that, after being sent by the initiator, are validated by the network and appended to the ledger itself. A simplified, yet very clear, explanation of the validation mechanism of Bitcoin transactions, as provided by [24], is that a transaction is first cryptographically signed with the transaction initiator's signature to ensure integrity, then, if it is properly formed and valid, it is sent to some nodes on the network, which in turn forward it to some other nodes and so on until it reaches all the nodes of the blockchain. This is called *flooding mechanism*.
When the transaction reaches a so called *mining node* it is verified and included in a block. Miners are the nodes in charge of aggregating transactions and appending them into the ledger where they are permanently stored, being accepted as valid by the whole network. Since the ledger is shared and stored on all the participant nodes, it is impossible to alter a block after it is written.
Bitcoin trading is not strictly related to this work, which focuses more on programming code on top of blockchains, but since it is the first example of this technology it is still relevant for our work to analyze some of its key aspects.
A very interesting overview of the history of Bitcoin is provided in [11], following the technology from its birth in 2008 to 2017: in 2012 more than 1000 merchants were accepting bitcoin as payment method and this boosted the interest in this field, up to the point that companies such as Microsoft and even countries, such as Japan, started accepting this currency as a valid payment method. Since blockchains are strictly related to cryptocurrencies ownership, a short overview of the largest attacks to Bitcoin exchanges well explains how difficult and risky writing code for blockchain applications is: in 2014 the Japanese company Mt.Gox announced that 650,000 bitcoins, equivalent to over $ 300 million at the time, were stolen from their customers' accounts. The following year the British exchange Bitstamp reported 19,000 bitcoins stolen ($ 5 million), while in 2016 Bitfinex reported

120,000 bitcoins stolen ($ 60 million)[11].

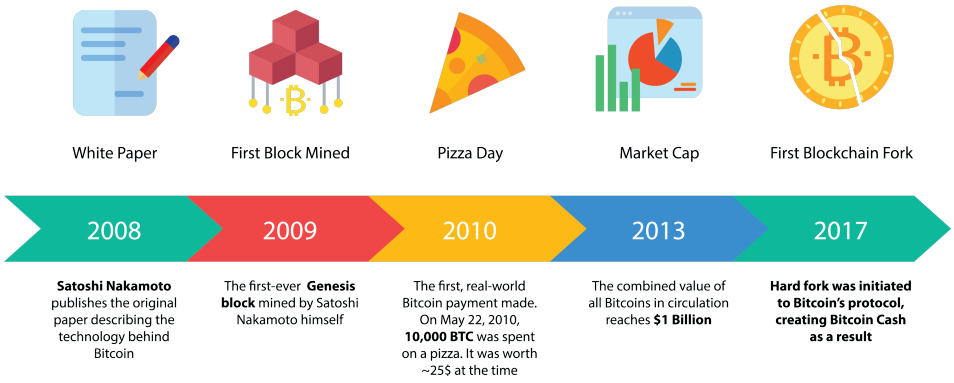

Figure 2.1: Evolution of Bitcoin, adapted from blog.coingate.com

2.2 Ethereum and Blockchain 2.0

With the birth of Ethereum in 2014, the era of blockchain 2.0 began. While Bitcoin was designed to manage and trade cryptocurrencies, Ethereum offered new capabilities, further pushing this technology towards new fields of application. The scope of this new blockchain was to build the generalised technology on which all the transaction-based state machine concepts can be built, offering to the users an end-to-end system for building software [26]. Ethereum offers, on top of the transfer of its own cryptocurrency, Ether, the execution of smart contracts, computer programs that run on the blockchain environment and are executed through transactions, interacting with cryptocurrencies and offering interfaces for the users to handle input [25].
In blockchain 2.0 transactions are not strictly financial but are used to support transactions like querying data and storing information [30].
Ethereum's runtime environment is a decentralized virtual machine, called Ethereum Virtual Machines (EVM), which handles the computation and state of the smart contracts. It is built on a stack-based, Turing-complete, language, offering a predefined set of instructions, called opcodes, with their arguments [5] making it suitable for a wide range of applications.
A smart contract can be written in high level languages such as Solidity,

Serpent and LLL and then compiled into bytecode to be executed by the EVM [9].
Each entity interacting with a blockchain is represented by an unique address and Ethereum offers two types of addresses: account addresses and contract addresses. Smart contracts, in fact, are deployed on the blockchain and assigned to a 160 bit unique address; from that moment on, the code can not be changed. They are invoked by sending a transaction to that address, which is forwarded to the network and mined to reach consensus on the result [1]. A contract address has a state (balance and private storage) and an executable code; the state is stored on the blockchain and updated each time the contract is invoked.
The capabilities of programmable blockchains allow them to be used, at least potentially, for a wide range of applications. Within the wide literature on the topic, one relevant branch includes smart contracts for decentralizing user's privacy by protecting personal data, letting people take the ownership of their own personal information. In this scenario, the blockchain works as access manager and the control of personal information shifts from companies, which collect, use and trade them, to the user itself, which can now decide who to give these data to and potentially getting even a profit from this [30].
Another interesting application field for smart contracts is business processes monitoring, where several entities should cooperate and trust between the parties is not guaranteed. In this scenario blockchain capabilities can bring relevant benefits because it works as a common ground for data sharing between the parties by keeping an unique shared state [24].
One additional use of this technology can be protecting data which are hosted within the cloud. An innovative work in this direction is presented in [13]; starting from the consideration that many applications and data in healthcare sector are being moved to the cloud, causing several security and privacy concerns, the authors propose the blockchain as a possible way to achieve decentralized consensus, consistency and resilience to data modifications, making sure that the medical history is complete, timely, accurate and anti-tampering [4][28]. More detailed instructions on how to achieve this, together with a case-study of a blockchain-based healthcare application are provided in [29], where the authors also address the main challenges of this approach, such as minimizing integration complexity and data storage requirements.

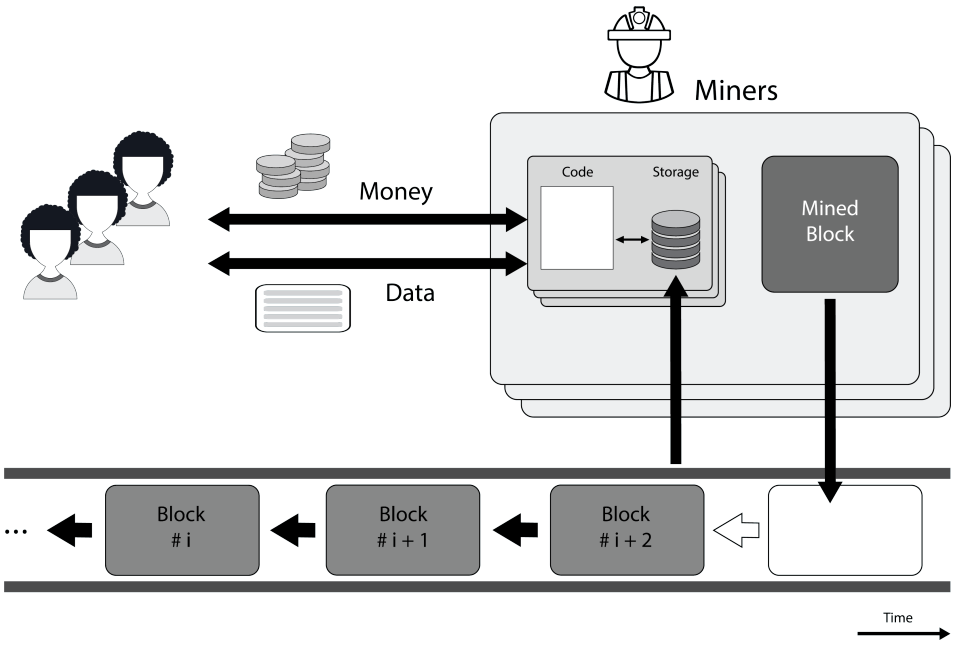

Figure 2.2: Personal data management on the blockchain

2.3 Smart contracts coding

When it comes to programming, the blockchain requires a different approach with respect to other infrastructures. There are different high level languages to write code in, which will be then compiled into bytecode to run on the EVM, but this technologies imposes several constraints to consider when writing code.

First of all, every instruction the EVM executes has a cost, which is paid to reward the miners for their work. Deploying a contract has a cost, proportional to the size of the code, each transaction has a cost, proportional to the data to be included, the computations the contract has to perform and the portion of memory to be used; this means that users are encouraged to write quality code, carefully studying every single line. A bad implemented operation leads to a waste of money each time it is executed, so if we consider that a function can be potentially invoked tens of thousands of times, optimizing code can lead to a huge money save on the long term[2].

Although Solidity, the most used programming language for Ethereum smart contracts, provides some useful functions and variables, such as *msg, block, tx* which allow programmers to access information regarding the transaction,

it does not provide many data types commonly used by other programming languages for managing storage and data manipulation, forcing the user to address tasks he may be not used to deal with, such as writing a method to concatenate strings [25].
Some other limits of this technology is the fact that the payload of transactions is not subject to any check or analysis by the consensus protocol, which is the reason why our work is relevant in this scenario: we want to perform quality controls on the application data before storing them permanently on the blockchain and this is possible just by implementing suitable smart contracts to be invoked for this purpose. With respect generic software, smart contracts: [8]

- Implement *passive application logic*, which means that they have to be invoked by a transaction in order to work and perform their task, they can't be triggered automatically when some conditions are met;

- Can not access data stored on the blockchain directly, they can just access data inside the transaction payload, local variables and data that other contracts make available through apposite functions;

- Can not access data stored *off-chain*, unless they invoke specific services, called Oracles, which in turn push data into the blockchain.

One further challenge posed by smart contracts is that the data integrity, immutability and persistence, some of the key features of the blockchain, can sometimes turn into an issue to take care of. This is especially relevant when it comes to dealing with personal data, which are protected by several laws. The GDPR gives EU citizens the right to request their data to be deleted, which goes in contrast with blockchain technology [13]. It is also relevant in this context to notice that this technology was originally designed to record transactional data, which means linear and small information. Data coming from business processes, such as healthcare data, can be relational, can require searching and are potentially very large, so many studies are being made on how this technology can deal with those kinds of data.
Currently, one critical aspect of blockchain is also scalability; its structure and the POW mechanism act as a bottleneck, so currently public infrastructures like Ethereum can handle, on average, between 3 and 20 transaction per second [27]. VISA payment service, for example, can process around 2000 transactions per second, at least 100 times more. As the number of applications running on the blockchain increases, this issue gets more and

more relevant, since the wait to have a transaction written on the ledger will constantly increase if the throughput remains the same. A possible solution is provided by Bitcoin lightning network, which moves some transactions *off-chain*: it establishes a multisignature transaction between two parties as a channel to transfer currency *off-chain*. In this way the transaction writes on the blockchain just when both the participants wish to confirm and close the transaction, lightning the amount of tasks to be performed *on-chain* [21].
Since blockchains are not suitable for storing large amount of data, given the fact that all the participant nodes in the network keep a copy of the distributed ledger, this can pose some limits on the kind of applications which can be developed on top of this infrastructure. A possible solution can be storing large data *off-chain* and keeping just their hash, or other meta-data, *on-chain* [25].
Another important aspect affecting smart contracts coding is that they deal directly with money. Virtual currencies have real value, so implementing code which stores and trade money raises new security challenges. On top of that, programmers should use an *economic thinking* when dealing with this matter, and it's possible that they did not acquire this skill yet. If, in traditional programming, a buffer overflow, a bug happening when a program writes data in a memory area it should not supposed to write into, is often benign, in this scenario it can lead to immediate exploit causing a potentially huge money loss [12].
On top of technology limitations, since writing code for the blockchain is a relatively new activity, with a small literature and a very few libraries to use, it is very common that smart contracts are badly implemented. A deep study on this can be found in [1], where the authors studied tens of smart contracts found in papers on the topic, finding out that almost the totality of them is subject to at least one of the following issues:

- Codifying issues
- Security issues
- Privacy issues
- Performance issues

Several solutions have been proposed to help programmers in writing clean and correct code, such as the semi-automation of smart contracts creation

by translating a human-readable contract representation into rules [14]. Delmolino et al. [12] wrote a tutorial to help programmers implementing correct smart contracts, while Bhargavan et al. [7] proposed to verify the correctness of the code by using formal methods. Chen et al. [10] tried to address the topic of under optimized smart contracts by identifying seven programming patterns which can lead to extra, unnecessary costs.

Category	Contracts	Transactions
Financial	373	624,046
Notary	79	35,253
Game	158	58,257
Wallet	17	1,342
Library	29	37,034
Unclassified	155	3,679
Total	811	759,611

Figure 2.3: Transaction by category as classified in [5]

2.4 Data Quality

Data are an highly valuable asset in almost every business, but with the birth of big data, as the amount of information available is growing quickly, the necessity of performing data analysis increases. This is because the sources of such data are several and different, which means that the data collected are rough and need to be analyzed in order to extract relevant information. In this scenario the techniques for data quality assessment can support this activity and become an indispensable tool when dealing with data [3].
Data Quality (DQ) is defined as the suitability of data for the respective

data processing application [16] and has been studied for decades; in 1993, Wang et al. [23] were already studying the parameters to consider in order to determine quality indicators to filter out data unnecessary for the specific application and, although the topic of information quality was already discussed, their work consists in a more formal and general analysis which is still relevant nowadays.
Data quality is a multidimensional concept because it considers several aspects by means of different metrics to assess. There are plenty of metrics which can be considered when dealing with DQ, but the literature agrees that there are some of them which are relevant to almost any applications:

- accuracy: it is defined as a measure of how close a data value v is close to some other value vẃhich is considered correct [22];
- completeness: it is defined as the degree with which a dataset includes the data describing the corresponding real-world objects [6];
- timeliness: it is defined as the measure of how much the data capture is close to the real-life event being captured;
- consistency: it measures the satisfaction of semantic rules defined over a dataset [6];
- precision: it defines how two successive measurements are closed to each other and it is computed as the inverse of the standard deviation calculated over the measured values.

Other metrics can be included depending on the specific context [13].
As smart contracts can implement business applications on top of the blockchain technology, we want them to be able to perform data quality analysis tasks in order not to store, and subsequently use, low quality data; these controls on the information quality have to be implemented as specific smart contracts, since the technology does not offer any tool for this purpose. Data quality controls in blockchain applications have not been very discussed, but a relevant paper on the topic, proposing a first approach to the implementation of quality assessing smart contracts, is the one by Cappiello et al. [8], where the authors discuss the topic at three levels of abstraction. They first make explicit the necessity to develop these kind of controls in blockchain applications, then they analyze how this technology affects the way in which data are used and made available for quality assessment. In sight of the considerations made, they propose a set of possible implementations which

take into account all the constraints imposed by the blockchain, applying them to two case studies taken from real-word applications. As in our work, they specifically focus on Ethereum as infrastructure and Solidity as coding language.

2.5 Summary

While many works discuss smart contracts programming, the necessity of implementing data quality controls to support such applications is usually not considered. The paper which brings the attention to this topic is the one by Cappiello et al. [8]; not only it studies how these controls can be implemented considering all the features and limitation of the blockchain, but it also provides some implementation guidelines, formalizing how to implement different functions according to the data to work with. On top of that, it provides some code fragments for the distinct implementation options.
Starting from their work, we want to get more into implementation details in this thesis, providing fully working smart contracts for data quality assessment. Additionally, we want to address the problem of evaluating the performance of such implementations, which is something that has not been discussed yet.

Chapter 3

Smart contracts on Ethereum: basic concepts

In this chapter we are going more into details of smart contract coding, by analyzing and explaining the underlying technology. We introduce here some useful concepts to better understand how smart contracts work, since they are the tool we are going to use to implement our solution. The real improvement between Bitcoin and Ethereum is the capability of the latter to execute code: the Ethereum Virtual Machine (EVM) is a Touring-complete virtual machine which has a unique global state and works with consensus. The ability of blockchain to reach consensus with no central authority is the really revolutionary idea which allowed realizing what up to that time was just a theoretic idea, a system where untrusted parties can trust each other without the need of a central authority.

3.1 The Byzantine Generals Problem

The problem of reaching consensus between entities that don't trust each other is usually called "The Byzantine Generals Problem" [18]. Some generals of the Byzantine army are camped outside an enemy city, each of them has its own troops and they are located in different areas. They have to reach consensus on whether to attack or retreat, because just a a coordinated action of all the troops can lead to a victory, while a single general behaving differently from the others would cause a defeat. The generals don't have a central authority to follow, the so called trusted authority which tells the participants what is true and what is not, so each general takes a decision and shares it with the other generals using messengers, which are the only

possible way to communicate. Theoretically, if all the messages get through, every party would get aware of the decisions of every other participant and they would base their actions on the same shared information, reaching consensus. Practically, this holds false most of the times, because three main problems arise:

- A messenger can be intercepted by the enemies or have an accident while bringing a message from a general to another. This would result in the loss of information, so the parties should take a decision without a full knowledge of the situation.

- A messenger can be intercepted by the enemies and substituted with a fake one, bringing a message which is misleading on purpose, resulting in the parties to base their decision on wrong information and losing the fight.

- One of the generals can be a traitor and provide a wrong plan on purpose, retiring its troops while the other Generals are attacking, causing a defeat for the Byzantine army. This is the perfect example of the problem of not having a central authority in a system of untrusted parties: the bad behaviour of a single person can badly affect thousands of people.

Since Bitcoin is a distributed ledger designed as a network of computers sharing a unique, common status, the nodes should reach consensus each time a new block is appended to the ledger, so the same problem applies to Blockchain: each general is a node of the network and they need to reach consensus on the current state of the system. All the participants have to agree, executing the same function in order to avoid failure. The property of a system to be able to resist to the class of failures caused by the Byzantine General Problem is called Byzantine Fault Tolerance (BFT) and it's the key feature of Blockchain, because a Byzantine Fault Tolerant System (BFTS) is able to continue operating properly even if some of the nodes have communication failures or malicious behaviour. The algorithm used by Bitcoin to reach consensus, becoming a BFTS, is called Proof Of Work (PoW) and it's considered as one of the most ingenious solutions to solve the Byzantine Generals Problem. The same algorithm has been later adopted by other blockchains such as Ethereum.

3.2 Consensus

A blockchain is a network of node computers which can create, receive, store and send data and since there's no central authority, the nodes start from the assumption that no one else in the network can be trusted, so the system uses PoW to be able to reach consensus even in presence of non-compliant nodes. Specifically, Proof Of Work is used to validate transactions and append them to the blockchain. Once a transaction is submitted, it is first broadcasted to the network, where all the nodes independently verify that it is valid against a series of criteria, such as the transaction amount is within the allowed range. After this step, transactions go into the so called *mempool*, where they wait to be included into a block to be permanently stored on the blockchain. The creation of a valid block is carried out by the so called miner nodes, which compete against each other to be the first to create and append the next block by solving complex computational puzzles, which are very difficult and resource consuming to solve, but very easy to verify the correct solution. Specifically, the mining node needs to try and guess a pseudo-random number, called nonce, which combined with the data contained in the block and passed through a hash function, must produce a result matching some given conditions, such as starting with four zeroes. Once a matching solution is found, it is validated by the other nodes and the miner node gets a reward for its work. The generated solution is called block hash and each validated block has its own block hash, which represents the work done by the miner, this is where the definition Proof of Work comes from.

PoW contributes in protecting the network against several possible attacks, because an attack would require a lot of computational power and a lot of time to be performed, resulting in the incurred cost to be larger than the potential reward.

The drawback of introducing Proof of Work is that submitting a transaction to the network has a cost, because the blockchain should reward the miners for the tasks performed. The fees charged for submitting a transaction are proportional to the computational complexity of the code to be executed and to the amount of data to be stored on the blockchain itself. For this reason, file storage applications or complex algorithms are not suitable for execution on EVM because the execution costs would be prohibitive, outweighing the benefits of running code on this infrastructure. The goal

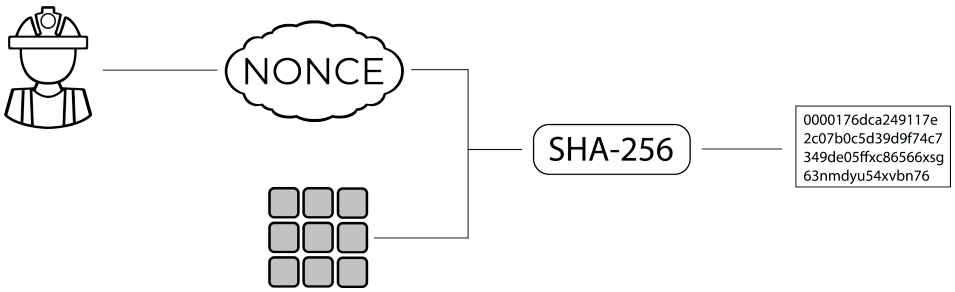

Figure 3.1: Mining mechanism in PoW based blockchains

of this thesis is deciding whether it's possible to perform data quality controls in Ethereum smart contracts in an economically feasible way or not, so we are going to see in details how the cost of running a smart contract is calculated.

3.3 Transactions

There are two types of transactions in Ethereum: those resulting in a message call and those resulting in the creation of a new account with associated code, informally a contract creation. Both of them specify some common fields [26]:

nonce: scalar value equals to the number of transactions sent by the sender (T_n).

gasPrice: scalar value equal to the number of Wei to be paid per unit of *gas* for all computation costs incurred as a result of the execution of the transaction (T_p).

gasLimit: scalar value equal to the maximum amount of *gas* that should be used in executing this transaction. It is paid before any computation is done and can't be increased later (T_g).

to: 160-bit address of the message call's recipient or, for contract creation transactions, $\emptyset$ (T_t).

value: scalar value equal to the number of Wei to be transferred to the message call's recipient or, for contract creation transactions, an endowment to the newly created account (T_v).

v, r, s: values of the signature of the transaction, used to determine the sender of it (T_s).

Additionally, a contract creation contains:

> **init:** unlimited size byte array specifying the EVM-code for the account initialisation procedure (T_i).
> It returns the **body**, a second fragment of code executing each time the account receives a message call.

In contrast, a message call transaction contains:

> **data:** unlimited size byte array specifying the input data of the message call (T_d).

To better understand the parameters above, an overview on how the transaction fees are managed can be helpful. The transaction fee is expressed in *gas*, which measures the amount of work miners have to do in order to include the transaction in a block. Gas units are purchased from the transaction's sender when submitting the call and they are paid in *wei*, which is the smallest fraction of Ether (ETH), the currency used on the Ethereum network. 1 wei is equivalent to 10^{-18} ETH. The sender, when submitting the transaction, can decide how many wei to pay for each gas unit and the more they decide to pay, the faster their transaction will be processed and written on the blockchain, because miners give higher priority to transactions they can get higher rewards from.
It is important to notice that T_g is not the actual amount of gas units needed to get the transaction written on the blockchain, but it's the maximum number of gas units the sender is willing to pay for it. The need to set a limit is given by the fact that the EVM is Touring-complete so it's prone to the halting problem, which is even amplified because of the complexity of writing code executing on the Ethereum infrastructure; when a transaction is submitted, it's not possible to know in advance whether the computation will terminate or not, so it's necessary to set T_g in order to avoid halting the system: when it reaches the threshold, the transaction automatically reverts. When sending a transaction, setting a proper value for T_g is crucial, because if during the transaction the remaining gas gets to zero, the transaction is reverted but the sender still has to pay for the work of the miner, while if the computation terminates with some residual gas, that amount is refunded to the sender. On the one hand, setting a low T_g can cause the transaction to revert before finishing even if the code would terminate; on the other hand, setting a very high value for T_g can seem a good solution, because the residual gas amount is refunded, but submitting a non terminating code, which is not that uncommon in smart contract programming,

would result in the loss of all the gas purchased.

The way how *gas* is spent is crucial for understanding the reasons behind this thesis;

- when the execution begins, the remaining gas is set to

$$T_g \text{ - } 21000 \text{ - } 68 * \textbf{ TXDATALEN}$$

- once the code is executed, every computational step decreases the remaining gas of an amount of gas units depending on the complexity of that specific operation. A specific cost is associated to each computation, a complete table can be found in figure 3.2. As an example, summing two integers costs 3 gas units, multiplying two integers costs 5, storing a word on the blockchain costs 20000. Taking a look at these examples, it is evident how any line of code can have massive impact on the execution cost.

3.4 Oracles

It is important to notice that smart contracts cannot access *off-chain* data, however for some applications it is vital to get some information from the outside word in order to perform calculations. For this purpose, many blockchains, including Ethereum, support *oracles*, a layer which can query, verify and authenticate external data sources providing the result to the invoking contract, working as a bridge between the blockchain and the external word. The information typically retrieved by oracles includes exchange rates, asset prices and real-time information.
The capability to fetch *off-chain* data massively enlarges the range of applications of smart contracts, because just a very limited number of program could work using just the data stored *on-chain*. It can look like using data not coming from the blockchain can violate the fundamental principles of this infrastructure, such as decentralization and the lack of a third party authority, since we are basing our computations on data provided by an external data source. Actually, we saw that the consensus algorithm aims to allow transactions between parties that don't trust each other, so it is not important where the information comes from, as long as the two following properties are satisfied:

non-repudiation: once a participant says something, he's not able to repudiate that.

integrity: nobody is able to alter what has been said.

Many different blockchain oracles exist and they are usually classified according to three qualities: the data source, the direction of the information and how the trust is established; a single oracle can belong to multiple categories. Specifically, we can find [19]:

Software oracles: they interact with online information sources, such as databases, servers and websites, providing data in real-time. Typical information accessed by software oracles are exchange rates, asset prices and a large variety of real-time information.

Hardware oracles: they interact with the real world, taking data from the physical word to make it available to smart contracts. Some possible sources of information con be electronic sensors and information reading devices such as RFID. They work as a translator between real-world events and digital value that can be used by smart contract, an example can be the temperature of a room.

Inbound oracles: they fetch information from an external source and make it available for a smart contract running on the blockchain. An example is the current conversion rate between Euro and US Dollars.

Outbound oracles: they take information from the smart contracts and send it to the external world. An example is a smart lock, which opens when a given condition, evaluated *on-chain*, is satisfied.

Centralized oracles: they only rely on a single provider of information, because the oracle is controlled by a single entity, which can act maliciously, having a direct impact on the contract. On top of that, this means that there's a single point of failures, so the contract is not resilient to vulnerabilities and attacks, because the failure of the information provider would prevent the code from executing.

Decentralized oracles: they are designed in a way similar to blockchains, so the smart contract relies on a series of oracles to determine the validity and accuracy of data, avoiding to rely on a single source of truth. With this architecture, trust is distributed between many participant in order to avoid a single point of failure.

Name	Value	Description
G_{zero}	0	Nothing is paid for operations of the set W_{zero}
G_{base}	2	The amount of gas to pay of operations of the set W_{base}
$G_{verylow}$	3	The amount of gas to pay of operations of the set $W_{verylow}$
G_{low}	5	The amount of gas to pay of operations of the set W_{low}
G_{mid}	8	The amount of gas to pay of operations of the set W_{mid}
G_{high}	10	The amount of gas to pay of operations of the set W_{high}
$G_{extcode}$	700	The amount of gas to pay of operations of the set $W_{extcode}$
$G_{balance}$	400	This is paid for a BALANCE operation
G_{sload}	200	This is paid for a SLOAD operation
$G_{jumpdest}$	1	This is paid for a JUMPDEST operation
G_{sset}	20000	This is paid for a SSTORE operation when the storage value is set to non-zero from zero
G_{sreset}	5000	This is paid for a SSTORE operation when the storage value's zeroness remains unchanged or is set to zero
$R_{rsclear}$	15000	This is the refund given when the storage value is set to zero from non-zero
$R_{selfdestruct}$	24000	This is the refund given for self-destructing an account
$G_{selfdestruct}$	5000	This is paid for a SELFDESTRUCT operation
G_{create}	32000	This is paid for a CREATE operation
$G_{codedeposit}$	200	This is paid per byte for a CREATE operation to succeed in placing code into the state
G_{call}	700	This is paid for a CALL operation
$G_{callvalue}$	9000	This is paid for a non-zero value transfer as part of the CALL operation
$G_{callstipend}$	2300	This is a stipend for the called contract subtracted from $G_{callvalue}$ for a non-zero value transfer
$G_{newaccount}$	25000	This is paid for a CALL or for a SELFDESTRUCT operation which creates an account
G_{exp}	10	This is a partial payment for an EXP operation
$G_{expbyte}$	50	This is a partial payment when multiplied by $[\log_{256}(exponent)]$ for the EXP operation
G_{memory}	3	This is paid for every additional word when expanding memory
$G_{txcreate}$	32000	This is paid by all contract-creating transactions after the *Homestead transition*
$G_{txdatazero}$	4	This is paid for every zero byte of data or code for a transaction
$G_{txdatanonzero}$	68	This is paid for every non-zero byte of data or code for a transaction
$G_{Gtransaction}$	21000	This is paid for every transaction
G_{log}	375	This is a partial payment for a LOG operation
$G_{logdata}$	8	This is paid for each byte in a LOG operation's data
$G_{logtopic}$	375	This is paid for each topic of a LOG operation
G_{sha3}	30	This is paid for each SHA3 operation
$G_{sha3word}$	6	This is paid for each word for input data to a SHA3 operation
G_{copy}	3	This is a partial payment for *COPY operations, multiplied by the number of words copied
$G_{blockhash}$	20	This is a payment for a BLOCKHASH operation

Figure 3.2: Costs of operations on Ethereum blockchain

Chapter 4

Data Quality Assessment in Blockchain Applications

In this section we are going to explain how we approach the problem theoretically, showing in section two the goals and requirements of our work and in section three its core aspects.

4.1 Data quality assessment

This section focuses on the theoretical approach to data quality dimensions assessment as introduced in [8], first by conceptualizing the information needs of typical metrics and the possible policies from a more general point of view, then by analyzing the data sources and implementation options offered by the blockchain.

4.1.1 Data quality controls

Data quality is usually defined as *data fitness for use* [22], which means how they meet the requirements imposed by the specific application, and it is pictured by several different dimensions, which can be relevant or not depending on the specific context. Within the whole set of possible metrics, we will focus on *accuracy, completeness, timeliness and precision* as defined in Chapter 2; the first three dimensions are relevant in every information system, while the latest one is particularly relevant in the context of IoT, where data are streamed by sensors.
The way in which data quality assessment algorithms are defined depends on the type of data to deal with and some additional data, such as the

reference value v' when assessing accuracy, may be required; considering the additional data needed to measure the quality of a variable, four situations may occur:

1. quality assessment does not require any additional information so the analyzed value is sufficient. Possible cases are the *accuracy*, that can be measured using just constants and business rules and completeness, computed by checking whether the value is present or missing;

2. quality assessment of a given value requires one or more past values of the same variable; for example, in IoT scenarios, the precision of a measured value is assessed considering previous measurements;

3. quality assessment of a given value requires single values of different other variables. For example, the consistency of a sensed temperature may be assessed against the values of pressure and humidity measured by other sensors;

4. quality assessment of a given value requires multiple values of a number of other variables.

Measuring data quality at runtime makes it possible for the application to implement different policies reacting to the results of quality assessment; how to react to the measured metrics is, of course, an application-specific matter, but we can distinguish five *policies* to adopt:

- *accept value*: sometimes we can decide to accept non qualitative data that violate the requirements. One possible case can be the values measured during sensor configuration, because we know that they are not relevant for the application;

- *do not accept value*: we decide to refuse non qualitative data, avoiding writing them into the system. In this case we consider a better scenario having less, qualitative, data with respect to having a larger dataset including low quality information;

- *log violation*: in some cases it can be necessary to accept a value but flagging it as low quality. The flagged value, then, can by used by the application for further computations.

- *raise event*: for some applications a low quality value represents a critical situation requiring an intervention, so the system would raise an event to notify some other system or person;

- *defer decision*: sometimes one single violation is not sufficient to take a final decision on how to react, so the decision is deferred.

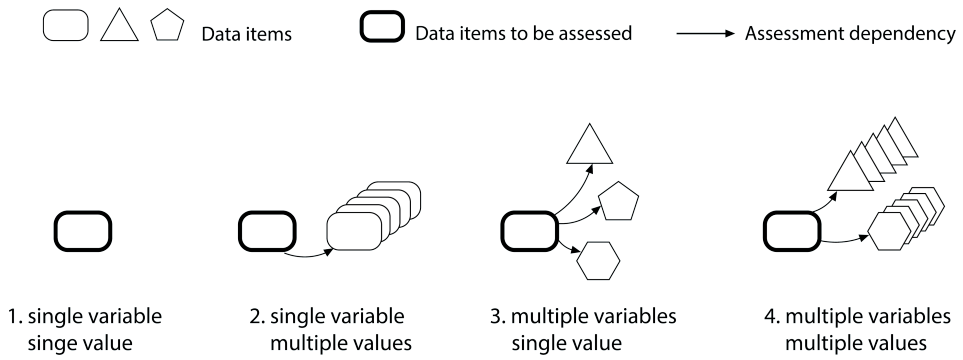

Figure 4.1: Dependencies among variables and values in data quality assessment

4.1.2 Logical implementation

The information needed for data quality assessment that we pointed out in the subsection can be provided to smart contracts in different ways, which in turn require different approaches. In this sense, the contribution provided by Cappiello et al.[8] is very relevant, because the authors identified four blockchain-specific patterns these data can be made available through:

- a single transaction brings all the values needed inside its payload, data are synchronized and the metric assessment can be computer as soon as the transaction is received;

- multiple ordered transactions bringing different pieces of information needed for metrics evaluation, the last of which is the value to assess quality of;

- multiple interleaved transactions bringing different pieces of information to be used for data quality assessment, but no specific order is guaranteed so correlating transactions is a necessary step in order to perform the task;

- external data sources, such *off-chain* data stored on a network node or web-accessible information, are needed in order to retrieve the data

used for quality assessment. In this case the smart contract should make use of an oracle, complicating metric computation and increasing costs.

As pointed out in Section 2.3, the blockchain technology poses some limitations on how data quality controls can be implemented inside smart contracts. Considering the four data access patterns we listed above, the possible implementation options for quality controls are:

1. *stateless smart contract*: if all the data needed for assessment are in the transaction payload, the control logic can be implemented by one simple stateless contract accepting the input data;

2. *stateful smart contract*: if data coming from different ordered transactions are needed for assessing a value, a stateful smart contract with functions able to provide persistent storage is needed;

3. *stateful smart contract with correlation*: if there's no guarantee on transaction order, the contract should implement, on top of storage capabilities, functions for correlating values in order to decide when assessment can be performed;

4. *smart contract and oracle*: when there is need to use *off-chain* data to assess quality, an oracle should be used additionally to the smart contract.

4.2 Goals and requirements

The goal of this thesis is to discuss whether it is reasonable or not to introduce data quality controls in Ethereum Smart contracts. There's not a single answer to this question, because it depends on many aspects such as the business domain of the specific application, the time and cost constraints and whether the smart contract can be implemented without these controls or not, but we consider two real-life, relevant scenarios to perform a wide analysis and a cost-benefit evaluation.
We propose data quality controls as Solidity code that can be reused in many different applications as it is or with just small modifications, because as writing smart contracts is an error prone activity, reusing code which has been already widely used, tested and improved can bring huge benefits to programmers. We decide to cover the most relevant metrics, the one

which are used more often, so that our code can be valid in as many cases as possible; the metrics considered are accuracy, completeness, consistency, precision and timeliness, as defined in Chapter 2.
On top of that, we consider two very different cases: the first one works with data streamed from a sensor, which are measured each minute and transmitted to our sensor once per hour; the second one receives a series of different data inserted by a user so performs the metrics evaluation on demand. To further increase the number of cases considered we implemented functions working with data stored locally inside the contract, functions using data stored on the blockchain and even functions using oracles to fetch *off-chain* data.
In Chapter 3 we explained in details Ethereum's reward mechanism, pointing out how any single operation performed has a cost. Participants have to pay a fee when deploying a smart contract, but also each time they call a function by means of a transaction and this fee is proportional to the number and type of operations the code executes and also to the amount of memory used and the number of read and write operation. The more complex the function , the more the computational power miners need to use for appending the transaction to the next block, the higher the transaction fee we have to pay them as a reward. We also know that different operations may have hugely different costs, so any single programming decision is relevant in making the code as efficient as possible, both in terms of cost and execution time.
Since miners give priority to the transactions with higher gas price, execution time and cost are in inverse proportion, so the constraints on the two parameters are a crucial information in order to decide whether using these data quality controls is convenient or not for a specific application.

To help the reader in clearly understanding what is the result we want to achieve, in this paragraph we are pointing out the requirements to meet with our solution.

Functional requirements:

- The solution should allow users to perform data quality metrics assessment on top of their Ethereum smart contracts.
- The solution should be flexible enough to be adapted to many different applications.
- The solution should consider different scenarios, in order to highlight how different approaches and constraints impact the cost

and overhead of such controls.

Non-functional requirements:

- The solution should be as efficient as possible, in terms of both time and cost, following all the known best practises in smart contract programming.

Architectural requirements:

- The solution should be able to be integrated with an already existing smart contract, providing a constructor to tune it according to the case specific needs

Technological requirements:

- The solution should be written in Solidity language and run on Ethereum Blockchain
- The solution should be provided as a contract with a function to be called passing the data for which the quality has to be assessed

4.3 Core aspects

In this section we present the two core aspects our work bases on and that we always keep in mind when developing our implementation.

4.3.1 Reusability

The first goal of this work is to provide smart contracts for data quality assessment that can be reused, because as discussed writing code for blockchain applications is an error prone activity, so using code which has been already tested and evaluated can pose several benefits with respect to writing code from scratch.
Since the possible applications running on Ethereum are virtually infinite, it is impossible to cover all the cases here, but we tried to provide two smart contracts which can cover as many cases as possible. For some applications it would be enough to deploy the code as it is, for some other cases some modifications may be necessary, for this purpose we decided to:

- Divide the code in atomic functions, so that programmers can reuse just specific parts of code;

- Provide rich comments to each function in order to help the reader in correctly understanding their functioning.

Our first scenario considers drug transportation, but the same smart contract can be used for many different cases where a datum is streamed from a sensor and its value should be kept under control, for example temperature, humidity, radiations, voltage, hearth rate. Although the business applications will be developed in different ways, our smart contract can be applied to all the cases above, which are basically IoT scenarios, to assess data quality. We have provided our smart contract with a function to be invoked to trigger the data quality assessment, which not only gets the data to evaluate but also gets all the other useful parameters, such as the reference values and the thresholds to detect out of range values. Parameterizing our contract makes it possible for it to be used in different scenarios without changing the code, the only requirement is that the data to be evaluated are passed as an array of value-timestamp pairs, which is a prerequisite very easy to meet.

The second scenario, drug prescription, is more specific, which means that the proposed smart contract won't be reused as it is for different applications. Nevertheless, since it is more complex than the previous one, because it features advanced data managing, including structures and mapping, complex logic and even the use of an oracle for fetching data located *off-chain*, some parts of the code can still be reused or at least can provide helpful hints on how to deal with similar situations.

4.3.2 Code Optimization

As discussed, deploying a contract on the blockchain has a cost; also, each single transaction has a cost. Every single operation requires a fee payment, from adding two integer values to reading a datum from the contract memory, so writing smart contracts require an high level of understanding of the underlying technology, the blockchain, in order to optimize each line of code. This is not easy, especially if we consider that the programming languages used for coding smart contracts force the programmer to deal with aspects he may not be familiar with, such as writing functions for very simple operations standard programming language provide built-in functions for, or choosing between three different possible integer data types: 8 bit, 16 bit, 32 bit.
We'll get into details of each single aspects of our smart contracts, improving

them as much as possible in order to provide the best practises for coding similar applications.

4.4 From requirements to design

We started this section by approaching data quality assessment in blockchain applications from a more theoretic perspective, analyzing the information needs of typical metrics, how data sources can be made available and the implementation options offered by the technology. All the considerations made in Section 4.1 have been used to identify the goals and requirements our solution should meet, expressed in Section 4.2.
Pointing out the requirements of the work is crucial in order to drive the design decisions we deal with in the next chapter.

Chapter 5

Data Quality Assessment in the considered scenarios

As outlined in Chapter 1, this work aims to integrate data quality controls inside blockchain applications, in order not to store low quality data which can affect the functioning of the program. To validate our approach we make use of two case studies where applications have to assess the quality of received information before storing it: in the first one a drug transportation company wants to store *on-chain* the temperature measurements recorded by the sensors placed on the trucks and in the warehouses, while in the second one the prescriptions made by a doctor are stored on the blockchain to prevent the doctor from hiding negligence and to avoid that patients can move false accusations towards him. In this chapter we present the design decisions taken and the architectures adopted, which constitute the basis of the implementation we present in Chapter 6.

5.1 Design Decisions

The topic of data quality control is very new and unexplored, so we have to take many design choices from scratch. First of all, we decide to start from two concrete scenarios, to highlight why a possible user should decide to develop a business application on the blockchain. We use the two scenarios proposed by [8] and refine them to have a common ground between them: drugs. The first case considers the transportation of drugs, which is a very delicate task subject to many constraints, so verifying that they are always met is vital.
In this scenario the data to assess the quality of are streamed from sensors

placed on the trucks, so by studying this scenarios we are potentially covering all those cases where an user has to assess the quality of data streamed from sensors, namely IoT applications.
In the second scenario, a doctor has to prescribe drugs to his patients and the system checks the validity of a prescription before confirming it. This case is particularly relevant because it is a very complex scenario considering several different aspects: it uses different data structures, it loads and stores data from the contract memory and it even makes use of an oracle to fetch *off-chain* data.
When coding the smart contract, writing one unique function receiving the input data and assessing the final metrics would result in a shorter code, which can lead to some savings in terms of deploying cost, but we prefer to follow the programming best practises and divide the contract into several functions, each performing a specific atomic task. This decision brings two main benefits to our work: it makes the code more clear and easy to read and it will also allow us to measure the performance of each single function, which means having many more data to perform our analysis on.
We also have to decide how to implement oracles; since developing a new oracle is a very complex activity and it is out of the scope of this work, we decided to use Provable, previously called Oraclize, which is the most diffused and well-known oracle service available.

5.2 Architecture

In this work we propose the architecture presented in figure 4.2.
The Data Quality assessment smart contract, represented by the dashed rectangle in the figure, is invoked when application data are available for storing. We aim to offer smart contracts which can be reused by almost any applications as it is, or with just small changes, in order to facilitate the task of assessing DQ for the programmers wishing to implement applications running on top of the Ethereum infrastructure. Our smart contracts will be invoked by providing the data to be written on the blockchain and, optionally, any other information needed in order to assess the quality metrics, such as parameters and thresholds. The smart contract can, optionally, fetch data from the contract storage and from *off-chain*, by using oracles. The smart contract assesses all the DQ metrics and stores them, together with the data, inside the contract memory. Since the two use cases have some differences, we are going to get into details of each of them in the

following subsections.

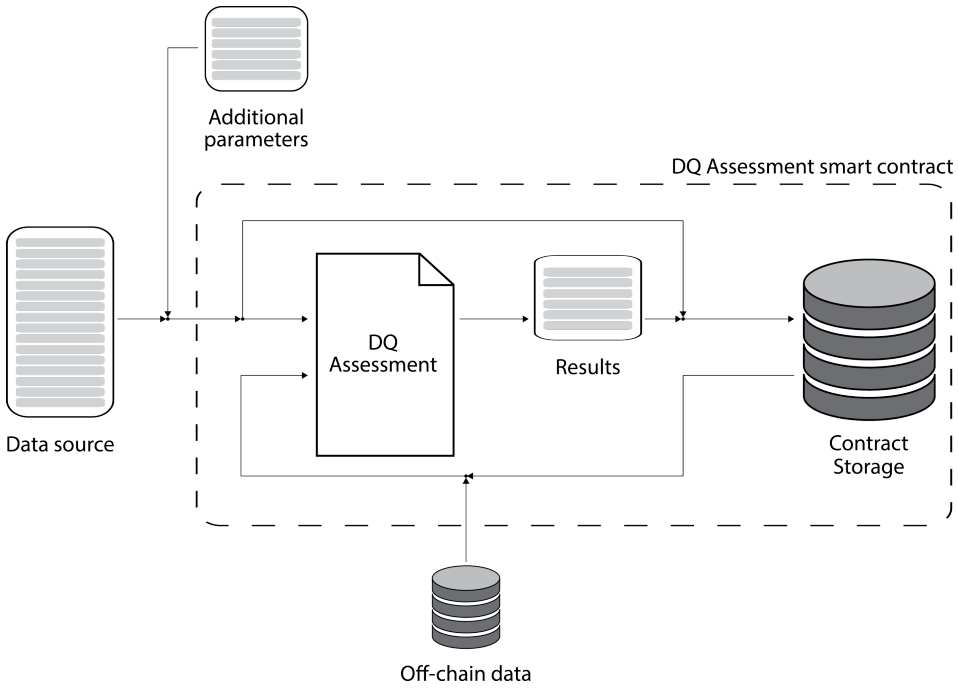

Figure 5.1: General architecture of a smart contract for assessing DQ

5.2.1 Scenario 1: drug transportation

In this case the data are sent every 60 minutes from the temperature sensors to the base station and then sent to the smart contract as the payload of a transaction. Our smart contract should assess the data quality metrics and store the results permanently on the blockchain, together with the data received. An out of range result would probably trigger some actions, but considering this is out of the scope of this work, so our smart contract should just receive data, evaluate the metrics and store the results. To do so, we decided to implement our contract as a series of separate, independent functions, one for each metric to assess, and one function working as the contract logic: it gets the data to be evaluated, invokes the assessment functions, collects the results and write them on the blockchain. Here we list the necessary functions and what they are required to do

Function A: interface

- Get the data to assess quality of;

- Get all the other parameters needed to evaluated the metrics, such as the expected value, the tolerance, the thresholds triggering the value change;
- Invoke the functions assessing the single metrics, providing all the data needed;
- Store the results permanently on the blockchain

Function B: accuracy

- Receive the data to be evaluated together with any other relevant parameters.
- Assess accuracy and return the result (true/false)

Function C: precision

- Receive the data to be evaluated together with any other relevant parameter.
- Assess precision and return the result (true/false)

Function D: completeness

- Receive the data to be evaluated together with any other relevant parameter.
- Assess completeness and return the result (true/false)

Function E: timeliness

- Receive the data to be evaluated together with any other relevant parameter.
- Assess timeliness and return the result (true/false)

In order for these functions to correctly work, some auxiliary functions may be needed.

5.2.2 Scenario 2: drug prescription

In this case, the smart contract to develop is invoked manually by the doctor when needed.
The logic of this scenario is more difficult than the previous one's, so many more functions are needed. Specifically, while in the previous case the task to perform is always the same, so we have just one function working as interface

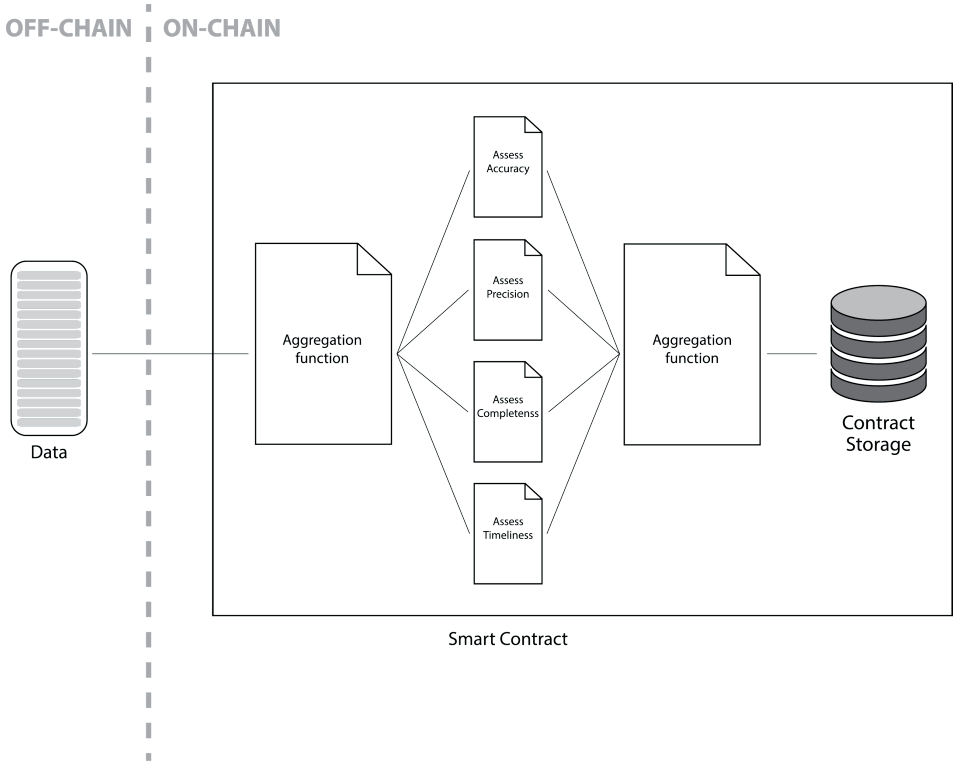

Figure 5.2: Architecture of the smart contract for scenario 1

towards the *out-of-contract* world, when a doctor prescribes a drug, there are at least two different cases to consider: prescribing a new drug and updating an existing prescription. On top of that, the contract should offer the possibility to remove a prescription once the treatment ends; this means that the smart contract to implement should offer at least three different functions to be accessed from, specifically:

Function A: new prescription

- Get the new prescription and patient's ID
- Check if the new prescription is valid, using the necessary functions
- Store the new prescription on the blockchain (if validated)

Function B: update prescription

- Get the new prescription and patient's ID
- Update the stored prescription with the new information

Function C: remove prescription

- Get the prescription to remove and patient's ID
- Remove from the memory the given prescription

To further facilitate the integration with existing code, we create a function acting as a selector for functions A and B: it receives the drug and the patient ID and checks if a record for the same drug and the same patient is already in memory; if it is, he calls function B, otherwise it calls function A. The three functions will invoke other functions to execute their control logic, but their implementation is out of the scope of this section and will be discussed in Chapter 6.

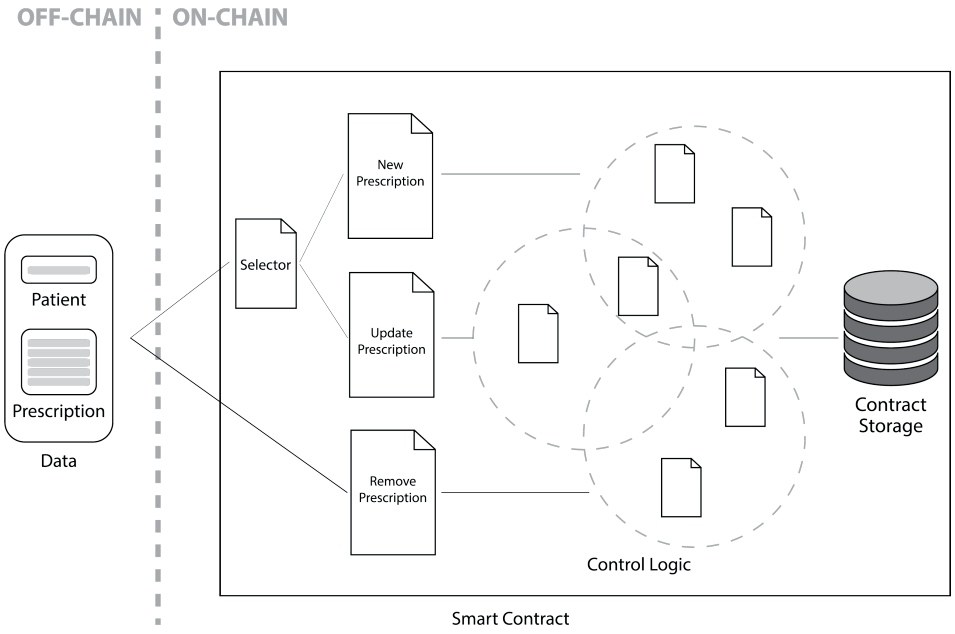

Figure 5.3: Architecture of the smart contract for scenario 2

Chapter 6

Implementation and Evaluation

6.1 Implementation

In this section we will get more into the details of our concrete work, the code, explaining how we practically implement our solution, all the design decisions we take and all the technologies and tools we use.

6.1.1 Technologies

When approaching our work, we first have to decide the blockchain to develop our smart contracts on. We decide to adopt Ethereum because it is the best known and the most widely used one for programming, so there are several tools available for writing code, different programming languages and a relevant literature on the topic. On top of that, Ethereum is a public blockchain, which means that anyone can access it and submit transactions, so our code can virtually be deployed by anyone.
The second decision we have to make is the programming language, since there are different possible alternatives to choose from and they all compile to EVM bytecode to be deployed on Ethereum. The two main options are Solidity and Vyper, the former has been first introduced in 2014 and since then it has been the most used programming language for Ethereum smart contracts; it is an object-oriented language inspired by C++, Python, and JavaScript. Vyper is a more recent language, available just in beta version at the time of writing, it is contract-oriented and has its focus on security. The purpose of Vyper is to provide the programmer with the ability to write

smart contracts without any undesired vulnerabilities or loopholes [15].
Although Vyper offers some interesting features, Solidity is still the standard for writing code to be deployed on the EVM, there are some useful libraries available online which can facilitate the task of programming and there is a considerable literature, which can help programmers in writing clean and optimized code, so we decide to use Solidity for our work.
The following decision is the development environment and we decide to use Remix, which is an IDE for Ethereum programming focused specifically on Solidity. An IDE is an Integrated Development Environment, namely an application offering a series of tools to help programmers in writing and executing code. Remix offers every tool needed to program in Solidity: a source code editor, a compiler, a debugger and can deploy and run the code; it can be installed locally but it can also be accessed from the web, which means that it can be used from any internet-connected devices, without any hardware or software requirements. To submit transactions on the blockchain, Remix requires the user to have an Ethereum account, which consists in an unique address used to store, send and receive Ether and to invoke smart contracts by submitting transactions to the correspondent address. To facilitate this task, the IDE offers full integration with MetaMask, which is the most widely adopted crypto wallet for Ethereum. MetaMask is an Ethereum wallet, which can be installed on many browsers as an extension and on iOS and Android devices to manage virtual assets and to directly interact with dApps, the decentralized applications running on top of blockchains.
Remix as IDE and MetaMask as wallet constitute the perfect environment for writing and testing code because they allow the use of test networks, namely copies of the Ethereum blockchain used for development purposes since they use a valueless version of Ether, made available upon request by a faucet contract. This means that accounts own Ether and transactions cost Ether, just like the Ethereum main net, but the currency used on the Testnet has no countervalue in the real world, so developers can deploy and test code without incurring in any concrete cost and avoid that untrusted code causes any financial loss.
The Ethereum testnet we use is Ropsten, since it is the only one using the same consensus protocol of the main net, proof-of-work.

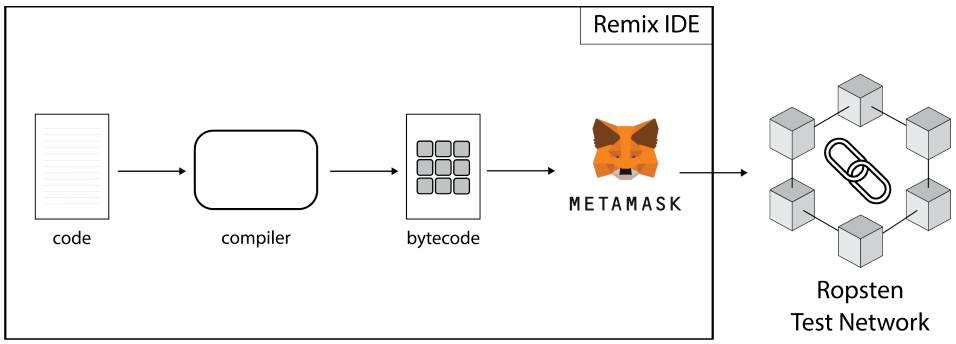

Figure 6.1: The environment used for coding

6.1.2 Implementation

In this subsection we'll get into the code to describe how the smart contracts are implemented. In order to keep the description as clear as possible, we are going to consider the two scenarios separately.

Scenario 1, Drug Transportation

Here we describe the smart contract developed for the first scenario, described in Section 1.2, where a transportation company working with drug want to store on the blockchain the temperature measurements made by the sensors located on the trucks. First of all, we code the data structures needed for this smart contract:

- *Measure*, it is the datum received from the sensor, consisting in a temperature-timestamp pair;
- *StoredRecord*, it is the datum stored on the blockchain after assessing the quality of a batch of measures, it consists in the array of measures and the value of the assessed metrics.

Once the data type are defined, we instantiate two global variables: the first one is an array of *StoredRecord* acting as the contract memory, the second one is an integer that simply counts the number of records that have been already stored in the contract memory; the purpose of this variable is facilitating the insertion of a new record by allowing direct access to the correct memory location.

We decide to divide the contract logic from the metrics assessment by having one function to be called by the business application, which in turn

```
9  struct Measure {
10   uint temperature;
11   uint time;
12 }
13
14 struct StoredRecord {
15 Measure[] smeasures;
16 bool saccuracy;
17 bool scompleteness;
18 bool sprecision;
19 bool stimeliness;
20 }
21
22 StoredRecord[] public mem;
23 uint public count;
```

Figure 6.2: Data structures used in scenario 1

invokes four other functions, one for each metric to assess. The function called by outside the contract to perform the data quality assessment is called *evaluateStream* and it receives as input:

- *values*, the array of measures to assess data quality of. It is a variable size array, which means that the same function can deal with any possible window size;

- *accuracyTemperature*, the reference value for the temperature field. It represents the expected value and it is used to check the accuracy of the measured values. It is provided by the user so that our contract can apply to different situations, for example different drugs may require different transportation temperatures;

- *accuracyTolerance*, the tolerance the application can accept on the measured value. If the absolute value of the difference between the measured temperature and the expected value is not larger than the tolerance, the datum is still accepted. Some flexibility is introduced here because the temperature measured by the sensor can be subject to small uncertainty given by the sensor sensitivity and by the external environment. This parameter is provided by the user so that the contract can adapt to different application-specific requirements.

- *accuracyTrigger*; it is common, when dealing with data streams, to consider a single value out of the range as an error, usually caused by a sensor malfunctioning or an accidental alteration of the value during transmission; for this reason the application may require that more than one out-of-range value has to be consecutively recorded in

order to mark the data batch as not accurate, that number is given by the *accuracyTrigger* parameter. *accuracyTrigger* indicates how many consecutive measures out of the range are needed to determine that the current batch of data is not accurate. The business application using our smart contract will most likely implement actions based on the assessed metrics, for example in this specific case if a data batch is marked as not accurate, the drugs would be discarded because they have not been properly preserved.

- *completenessLenght*, how many measures are expected in the data batch. It is compared against the length of the received batch to assess completeness.

- *precisionTolerance*, the range of accepted values for the standard deviation measured on the received data batch. If the value is higher than the threshold, the batch is marked as not precise.

- *timelinessDelta*, the maximum accepted delta between the datum timestamp and the time the data batch is received. It determines how "old" the datum should be to be not considered as *on time*. Some time margin is necessary not only because a small delay in getting a temperature measure won't cause any issue to the product, but also because the concept of time in the blockchain is slighlty different from the *real* time. This is because the only time indication on Ethereum is given by the timestamp of the block the transaction is appended to, so it can differ from the current time of 10 to 20 seconds, which is the average time spent to mine a new block.

- *timelinessTrigger*, as happens with accuracy, a single timestamp out of the range can be attributed to an error of the sensor or during data transmission, so more than one consecutive out-of-range timestamp is required in order to mark the timeliness of the current batch as false.

The function pushes an empty *StoredRecord* into the memory and then it fills up its fields: the four boolean variables associated to the metrics are assigned by invoking the corresponding functions, one for each metric, while the array of *Measure* is filled up by copying the *values* array into memory. Using a *for cycle* to save the data in memory may look like a bad implementation choice since every iteration has to read a value, but Solidity has several limitations when dealing with arrays and one of those limitations is

that it can not directly copy arrays to memory. When writing a cycle in Solidity it is vital to carefully check that every entry and exit condition is properly managed, because submitting a transaction where the exit condition is not met will cause the program to consume all the transaction gas, causing significant losses of both time and money. Before returning, the function increments the value of *count*.

```
70  function Accuracy(Measure [] memory values, uint mtemperature, uint tolerance, uint trigger) public pure
71  returns (bool check) {
72      uint8 conta = 0;
73      for (uint8 i=0; i<values.length; i++) {
74          if (values[i].temperature>(mtemperature+tolerance) || values[i].temperature<(mtemperature-tolerance)) {
75            conta++;
76            if (conta == trigger) {
77                return false;
78            }
79          }
80          else conta = 0;
81      }
82      return true;
83  }
```

Figure 6.3: The code assessing accuracy

The four functions to assess the metrics have a very simple functioning, they receive the data to be evaluated together with the other parameters needed, if any, and return a boolean value. *Accuracy* checks that the measured temperatures are in the given range, *Precision* checks that the standard deviation is smaller than the threshold, *Completeness* checks that the data batch contains as many elements as expected and *Timeliness* checks that the measurement timestamps are in the acceptable range.

Scenario 2, Drug Prescription

Here we describe the smart contract developed for the second scenario, described in Section 1.2, where a general doctor stores on chain all the prescriptions made to the patients. As we did for the first contract, we start by coding the data structures needed for the smart contract:

- *Prescription* is a data structure consisting in a drug, a formula, a daily dose, the category the drug belongs to and the category, if any, the drug is incompatible with;

- *Patient* consists in a personal identification string and the array of drugs prescribed to the patient;

then we instantiate a global variable, called *patientsList*, consisting in the list of all the patients, so acting as the contract memory. On top of that we create a mapping from string to integer, called *entry*, to be able to access

patientsList by the patient's personal ID; since an array element can be accessed just from its index, the mapping associates the patient's personal ID to the index the patient is located at in the array.

```
15  struct Patient {
16    string cf;
17    Prescription [] prescrs;
18  }
19
20  struct Prescription {
21    string drug;
22    string formula;
23    uint dailydose;
24    uint category;
25    uint incompatibility;
26  }
27
28  mapping (string => uint) entry;
29  Patient [] public patientsList;
```

Figure 6.4: Data structures used in scenario 2

We decide to create an unique function as a bridge between the business application and the smart contract application logic, to facilitate the integration with other systems; it is called *AddPrescription* and it receives as input:

- *prescriptionCF*, which is the patient's personal ID;
- *newPres*, the prescription as entered by the doctor, it can be either a new prescription or the update of an existing one;

This function first determines whether *prescriptionCF* exists in memory by invoking the *ExistsPatient* function and, if it does, it invokes the *ExistsPrescription* function which checks if the current drug has been already prescribed to the patient. Depending on the return value, the current function will determine the next steps. If the drug was already prescribed to the patient, it will invoke a function to update the prescription: it searches for the prescription in the contract memory and sets the new values for the *formula* and *dailydose* fields. Up to this point, the contract is pretty straightforward and does not require particular reasoning on how to implement the requested function.
What is more complex is the function to register a new prescription, because in this case the contract should consult an external source located *off-chain* to check that the drug exists and to retrieve its category and the drugs it is incompatible with, in order to assess the consistency with the drugs already prescribed to the given patient. To consult a source located *off-chain*,

Solidity offers the so called *oracles*, introduced in Section 3.4, ad-hoc smart contracts which can retrieve data from a data source located outside the blockchain and push them into the blockchain itself. Programming an oracle from scratch is a very difficult task and it is out of the scope of this work, so we decided to use Provable, which is an oracle service offering an easy and fast integration with smart contracts and it is the most widely used oracle service on the market. This service offers a library of functions which can be invoked by the programmer to read data from an external source and push them inside the smart contract; in exchange, Provable will take a fee for its work similar to the one miners earn. The functioning scheme of these oracles is pretty easy: the smart contract invokes a function, called *oraclize_query*, by passing a formatted string including the following information:

- The type of source to query, in our application we decide to store the records on a web page, so the source is an URL;

- How the data source is formatted. For URL sources the two options are XML and JSON, there is no relevant difference between the two options so we decide to store the information in an XML file;

- The URL to read data from, including the complete path to the data needed.

Once the function is called, the oracle will fetch the data at the given address, encode them in a string and send it back to the contract that called the oracle. For this purpose, the programmer should implement a function called *___callback* in its contract, which the oracle will send a transaction to when the data have been retrieved by including them as the transaction's payload.

The use of an oracle requires the programmer to take care of some aspects related to this choice; the first thing to consider is that the user passes a new prescription as parameter when sending a transaction to our smart contract and our smart contract has to consult the oracle before deciding whether the prescription is valid and can be written or not. The control flow goes from our smart contract to the oracle, and then again to the code inside the *___callback* function, but the prescription cannot be passed along this invocation chain because the function used to consult the oracle can just receive a query as parameter.

This means that we have to store the prescription before invoking the oracle

in order to be able to retrieve it after the re-entrance function is triggered; there are two main options to do so:

1. storing the prescription in a global variable as soon as it is received, write it in the patient's records just if it is confirmed after oracle invocation. The positive aspect of this solution is that, if the prescription is rejected, there are no further operations to perform because the global variable will be overwritten when the doctor will submit the next prescription and nothing has been stored on the patient's records. As a drawback, if the prescription is accepted, the smart contract will have to copy the prescription also on the patient's records, resulting in two writes of the same data;

2. recording the prescription on the patient's history as soon as it is submitted, and remove it if it is marked as invalid after the oracle invocation; this will save time and money in case the prescription is confirmed, because it will perform just one write operation, but will cause the need to remove the record from the memory if the prescription is not accepted.

Since we assume that most of the prescriptions submitted by a doctor would be correct, we decide to record the data as soon as they are submitted and incur in an extra computation if the prescription is marked as invalid after oracle consultation. In both cases, we need to store the current patient's personal ID before oracle invocation in order to access the correct record after the ___*callback* function is called.

The second aspect to carefully consider when using an oracle in Solidity programming is handling the exceptions properly, such as checking that the contract has a sufficient balance before calling the oracle, reverting the transaction if the ___*callback* function is invoked by a contract which is not the Provable address and managing the case in which the oracle returns an empty string, meaning that the URL provided does not bring to any valid information.
Forgetting to handle any of these exceptions can cause tragic consequences and the number of exploits towards Ethereum smart contracts is huge; in our code, for example, not checking the sender's address when the return function is called, means allowing everyone to push data into our contract, which in turn are used by our contract to take decisions, allowing an attacker to easily influence the behaviour of our code. The public nature of

blockchain technology, in this situation, turns into a potential risk for smart contracts' security: every transaction recorded on the Ethereum main net, as well as on the Testnets, is publicly available at *www.etherscan.io*, which means that everyone can search for Provable's address and see every single transaction ever sent to and from that address. An attacker can send transactions to the *___callback* function of those addresses and if any of them is not implementing a check on the sender's address the attacker can inject any information inside the contract just by using information publicly available.

The function we wrote to add a new prescription first stores the patient's ID in a global variable, then pushes the new prescription in *patientsList* at the index associated to the given customer, which is retrieved by accessing the *entry* mapping; after that, it invokes the oracle by passing the concatenation of the URL where the drug list is located and the drug name, in order to access the correct field inside the XML file. In order to facilitate the integration with the oracle, which can only return a string, we stored the *drug category - incompatible category* pair associated to each drug as two integers separated by a comma:

$$< drugA > 1{,}3 < /drugA >$$

When the Provable service retrieves the requested information, it calls the return function by passing the result string and our contract logic continues: first of all the *___callback* function checks if the result equals to the empty string, which means that the oracle was not able to retrieve any data at the URL provided. In this case, the prescription is removed from the memory, otherwise the contract checks if the returned data, the drug category and the category it is incompatible with, are in contrast with any of the drugs which are currently prescribed to the patient. If no incompatibility is found, the code updates the value of the two fields *category* and *incompatibility* in the stored record, otherwise it removes the prescription. Some of the auxiliary functions used during the process are *DeleteElem*, which gets a patient ID as input and removes the last prescription from the array associated to that ID, *UpdateCategory*, which updates the values of *category* and *incompatibility* stored in memory, and *ExistsCategory* which checks if the new prescription is in contrast with the existing ones.

This last function requires some attention because it has to get through the memory, which is managed as a two levels array, to check for incompatibilities, and Solidity poses several limitations when dealing with arrays. While the *entry* mapping allows us to directly access the index associated

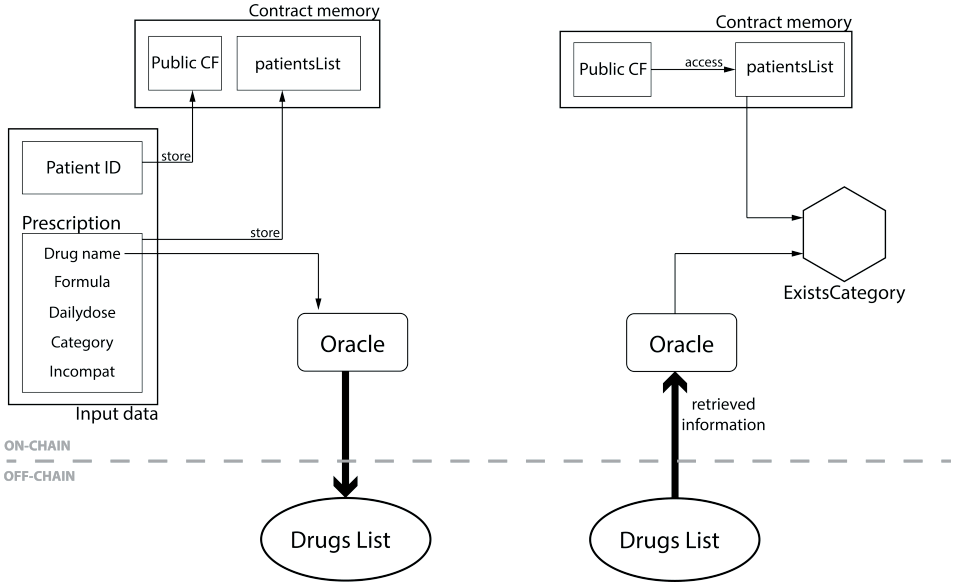

Figure 6.5: How data are passed along the smart contract

to a given patient ID, our function has to scan all the array of prescriptions using a *for* cycle in order to look for a record in contrast with the new one the doctor submitted. This can look like an inefficient solution, but given the few functions provided by Solidity to manage arrays, the only alternative option is having, for any patient, two arrays where we store all the categories of drugs currently prescribed to the patient and all the incompatibilities posed by the prescribed drugs. In this way, if a person has more than one drug of the same category prescribed, our *ExistsCategory* would scan a shorter array, saving both money and time; as a side effect we would have to store additional data in memory, which is an expensive task.
Another drawback of this approach is that prescriptions are usually limited in time and, when a record is removed, we would have to remove the corresponding values from the two arrays. This is not an easy task because, due to the limitations posed by Solidity, removing a record which is not the last in the array would require us to shift all the following values in order to avoid empty records; this operation requires several read and write operations, which are expensive to perform.
Considering that it is very rare that a patient has more than one drug prescribed from the same category, our solution is the more efficient in the given scenario.

6.2 Evaluation

The main goal of this work is evaluating the impact adding data quality controls has on the smart contracts' performance, both in terms of time and costs.

6.2.1 Design of Evaluation

To evaluate how data quality controls affect the code performance, we deploy, for each scenario, our smart contract, which includes the controls, and a smart contract that just stores the received data, then we measure the performance of each of them. As we saw in Section **??**, each transaction has a fee, expressed in *gas*, which has to be paid by the user to get the transaction mined and appended to the ledger. The amount of *gas* needed to submit a transaction depends on the number and type of operations to be performed and is fixed.
When the user sends a transaction, he decides how much to pay for each *gas unit*, determining the real cost of the transaction. Since the gas is used as a reward for the miners, we saw that the more we pay for the gas, the faster the transaction should be mined, because miners are more willing to work on blocks they get an high reward for; for this reason, in blockchain transactions, the price paid for submitting a transaction (in wei, the smallest fraction of an Ether) and the transaction time are usually correlated. The problem is that the price paid is not the only variable affecting the transaction time; the number of transactions pending can hugely affect the wait to have your transaction mined into a block in a way that can not be foreseen, so comparing the performances of different smart contracts is not an easy task.

The first metric that we can use as a comparison is the amount of gas needed to mine a transaction: it is deterministic, so we can measure the difference in terms of computation power needed to deploy two different smart contracts on the blockchain with no uncertainty. In order to find a way to compare also the cost in *wei* and the transaction time, we start by trying to determine a strict correlation between the two metrics; to do so, we take the same transaction and submit it for ten times keeping the same *gas price*, measuring the time it takes for it to be mined. We repeat this process with 10 different *gas prices*, from 0.01 Gwei to 10 Gwei. This process is repeated in different days of the week at different times of the day, because

the amount of traffic on the blockchain strongly influences the transaction time. The data collected does not provide a strong link between the *gas price* and the mining time, for example we registered one transaction where we paid 0.01 Gwei for each gas unit that took less time than a transaction where we paid 10 Gwei for each gas unit.
To deal with this uncertainty we decide to set a fixed *gas price*, compatible with our performance requirements, and to submit the transactions many times in order to evaluate the *average* performances. Considering our two use cases, we decide that a reasonable transaction time could be [0-30 seconds] for the drugs transportation scenario, [0-90 seconds] for the drugs prescriptions scenario. For the first scenario, acting quickly after founding out that the correct preservation conditions are not being met is important in order to prevent the drugs from being wasted, but a 30 seconds wait won't cause any worsening in the drug conditions. For the second scenario, the doctor has many appointments in a day, so a visit is usually pretty short. This smart contract requires the use of an oracle so the transaction time is longer than the previous case and 90 seconds is still a reasonable wait considering the total length of an appointment. Since the measured data are very different one another, we also set a constraint on the maximum percentage of transactions that could exceed our range: 10%. The first *gas price* satisfying these constraints, according to our measurements, is 1 Gwei, while lower prices would cause too many transactions exceeding our range; for higher prices, the improvements in terms of transaction time wouldn't justify the increase of costs: by doubling the cost, the average mining time decreased just by 6%, so we decided to keep 1 Gwei as the correct *gas price* for our evaluation. It is important to notice that the only concept of time in blockchains is the timestamp associated to a block, representing the Unix time the given block is appended to the ledger; for this reason, every measurement is subject to a 3-6 seconds uncertainty, which is the average time between two block confirmations.
The amount of *gas units* consumed by a transaction can be consulted at *www.etherscan.io*, searching by the contract address, while some more effort is needed in order to measure the transaction time. This has to be measured in the following two cases:

- Contract creation
- Contract interaction, namely function invocation

In both cases, we have to schedule the transactions because the transaction

submission timestamp has to be compared to the block timestamp in order to measure the transaction time as the difference between the two numbers. In the first case, contract creation, we store the transaction submission timestamps in an excel file when scheduling the submissions; after all the transactions are sent, we retrieve the block timestamps from www.etherscan.io and calculate the difference between the two values.
In the second case, contract interaction, we pass the submission timestamp to our function as a parameter together with the data to be stored, so we calculate the total transaction time internally, measured as the difference between the block timestamp and the input value, emitting an event as the last command before the function returning. Events log can be found on www.etherscan.io inside the transaction information, an example is provided in Figure 6.6.

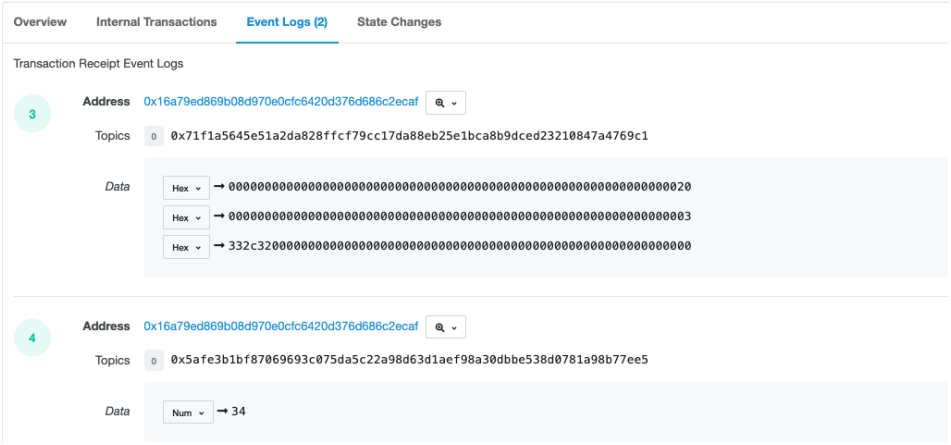

Figure 6.6: Transaction's event log as displayed on www.etherscan.io

6.2.2 Metrics

Since we decide to keep a fixed *gas price* for our evaluation, the metrics we measure are the *gas amount*, which represents the computational complexity of the smart contract and the cost for the executor, and the average transaction time. Since the transaction time is subject to huge variations due to the number of transactions waiting for mining, we repeat our measurements in 10 different days, five times during business days, five times during the weekend, where the traffic is lower according to the data we collected. In

this way, by using the average of our measurements, we should have data which can well represent the expected performance in the long term.
For each round we test two different smart contracts for each scenario: the first one just stores the received data, the second one performs the data quality controls we coded before storing data. For testing the contracts, we send the following transactions:

- **Drug transportation, contract creation:** ten different contract creation transactions;

- **Drug transportation, contract interaction:** four different data sets in order to cover different cases: some of our data quality controls have *for cycles* with exit conditions, so different data sets can determine different iterations resulting in different transaction times and *gas prices*, so it is important to consider different inputs in order to have a correct estimation of how the contract can perform once deployed. For each data set, we send ten different transactions;

- **Drug prescription, contract creation:** ten different contract creation transactions;

- **Drug prescription, contract interaction:** two different data sets in order to cover different cases: an accepted drug and a refused one go through different functions, so both cases have to be covered. For each data set, we send ten different transactions.

6.2.3 Results

The data collected show that adding data quality controls to a smart contract has a considerable impact on both the cost and the performance, but the result is highly affected by the size and computational complexity of the code we are adding quality assessment functions to.
The cost, both in time and money, of deploying a contract, basically corresponds to the size of the code; in the drug transportation scenario, the smart contract storing the measurements with no data quality controls is very short in size, so adding our functions for quality assessment almost triples the size, resulting in a +130% in cost and +100% in transaction time. On the other hand, adding data quality controls to the drug prescription scenario has a smaller impact on the code size, around 60% more; the measured cost increase of 50% is consistent with our expectations. The

unexpected datum we collected is the increase in transaction time in this second scenario, which has been measured in +156%. The inconsistency of this datum has to be addressed to the network traffic, since some of the test transaction took a very long time to be mined and this had a major impact on the average transaction time.

Getting to contract interaction transactions, it is more difficult to foresee the actual impact of adding data quality controls, because the cost and the transaction time are affected not only by the number of operations but also by their type. To perform data quality controls, a function inside a smart contract may require three different data sources: data received as parameters, data stored *on-chain* and data stored *off-chain*.
Our drug transportation scenario works directly with the array of measurements which is passed as a parameter, so it can perform quality assessment with a very small overhead: +40% in transaction time and just +3% in costs.
On the other hand, the smart contract for drug prescription requests data stored *off-chain* using an oracle and then compares the result against data stored in the contract memory; this is an extremely costly operation, mainly because of the use of the oracle, which is a third party service. Using an oracle has a very high impact both on the transaction time, because the smart contract has to wait for the service to return after a time that is determined arbitrarily by the provider, and on the costs, because the oracle requires a very high fee to perform its task. In our measurements, adding data quality controls using an oracle, results in +150% in time and +642% in costs in the case of a correct prescription, +280% in time and +316% in costs in the case of a wrong one. This overhead is extremely large and can lead to unsustainable costs for who decides to implement this smart contract.

6.2.4 Discussion

The collected results show that the overhead brought by the implementation of data quality controls in smart contracts is subject to large variation depending on the specific application. Since the contracts are deployed once, we think that it is not relevant the impact quality assessment has on this specific task, while it is very interesting to study how they affect the contract interaction. A contract can be invoked tens, hundreds, or thousand

Gas price = 1 Gwei		DQ			No DQ			Delta		
		Time	Gas Units	Cost (Gwei)	Time	Gas Units	Cost (Gwei)	Time	Gas Units	Cost (Gwei)
Transport	Creation	38	1020963	1020963	19	440936	440936	19	580027	580027
	New record stored	14	2845480	2845480	10	2771087	2771087	4	74393	74393
Prescription	Creation	23	2845480	2845480	9	1892948	1892948	14	952532	952532
	Drug accepted	50	430207	965625	20	130115	130115	30	300092	835510
	Drug refused	76	305968	541888	20	130115	130115	56	175853	411773

Figure 6.7: The data collected during testing

	% Increase adding DQ controls		
	Time	Gas Units	Cost (Gwei)
Transport	**100%**	**132%**	**132%**
	40%	**3%**	**3%**
Prescription	**156%**	**50%**	**50%**
	150%	**231%**	**642%**
	280%	**135%**	**316%**

Figure 6.8: The overhead caused by deploying data quality controls

of times a day, so even a small cost increase on the single transaction can lead to a dramatic extra cost for the contract owner at the end of the day. For this reason, we think that everyone considering to deploy an application on the blockchain should carefully evaluate all the costs which should arise from this decision.

In our opinion, if data quality controls are not mandatory for the specific application, the user should better deploy the application on the blockchain to enjoy the benefits coming from the adoption of this technology and then decide whether implementing data quality assessment is feasible in terms of costs and transaction time. Some specific applications may require very fast transactions, so an even small overhead can violate this constraint; some other applications, such as the one we considered in our work, don't have very strict constraints on transaction time, so deciding whether to implement controls or not just reduces to a merely economic consideration. If, instead, the application can not work without data quality assessment, this analysis has to be made before deciding to implement the application on the blockchain. If the controls have a small impact on the costs and performance, it is possible that the trade-off between the costs and the benefits

brought by the use of the blockchain is still advantageous for the user. If the controls require large access to the memory or even the use of oracles, like our drug prescription scenario, the costs to incur in can frequently overcome the benefits, so deploying the application on the Ethereum platform may not be the best solution for the specific case.
Nevertheless, the considerations about the sustainability of using data quality controls can not be reduced just to the analysis of the cost of the single transaction; in our drug prescription scenario, a transaction to submit a new prescription, including data quality controls, has a very high cost if compared to the other transactions we submitted for testing, but with the current exchange rate between Ether and Euro (April 2020), the extra cost for the single transaction is around 0.15 USD. If the application should invoke the contract ten times a day, the cost can still be perfectly sustainable for every user if compared to the benefit of implementing a blockchain application. If, on the other hand, the application should invoke the contract one thousand times a day, the cost can be sustainable just if the application is necessary for generating profit.
In conclusion, if we consider applications which can work also without data quality controls, implementing them can be feasible just if they work with *on-chain* data. If, instead, they need to use oracles, implementing quality assessment can be sustainable just for a minor percentage of the cases.

Chapter 7

Conclusion and Future Work

This chapter will draw the conclusions of the work that has been done in the previous ones. The first section describes, in short, the problems addressed by the thesis, the methodology used and the lessons learned. The second section summarizes the output of the work, explaining how it contributes in advancing the status of the art and the last two sections presents the limits of our contribution, providing some indications on the aspects that would deserve further studies.

7.1 Summary and Lessons Learned

The adoption of blockchain technology is having large impact on the development of applications requiring a distributed paradigm and smart contracts are the core entity of this transformation.
Yet, our research has shown how developing smart contracts is still a difficult, error prone activity and the lack of widely adopted conventions in blockchain programming limits optimization and reusability. Also, writing reliable smart contracts requires a deep understanding of the underlying technology, blockchains, and language specific coding skills to deal with the pay-per-invocation paradigm. On top of that, most of the literature focuses on functionality and transaction rates, with just a few mentions to the costs associated to deploying and running distributed application.
The approach adopted in our work, instead, focuses mostly on costs, both economical and in terms of time, because no application can ignore this aspect: an application that is not economically sustainable is, from a practical point of view, an useless application.
All the above mentioned considerations have been the starting point for ap-

proaching the topic of this work: studying how data quality controls can be implemented inside smart contracts and how they affect the cost and performance of distributed applications. Data quality assessment is fundamental for almost every application, but has been poorly discussed related to dApps. Nevertheless, since the number of applications running on the blockchain are expected to constantly grow, it is vital to enable smart contracts to perform data quality controls, because data are a valuable asset just if they are suitable for the specific case. This work aims to providing useful guidelines, including reusable code, for implementing data quality controls in distributed applications and to study their overhead.

7.2 Outputs and Contributions

The main outputs of this work are:

- an analysis of the issues related to blockchain programming, including the limitations posed by this technology;
- two working smart contracts for assessing the quality of the application data, covering a wide range of possible applications;
- an analysis of the costs introduced by data quality controls, both in terms of money and execution time.

The main contributions of this work are:

- an extensive review of the state-of-the-art of blockchain programming, focusing on the possible applications;
- the design and implementation of data quality controls that can be reused and extended to other scenarios;
- a study on the overhead associated to such controls.

7.3 Limitations

This work explores a relatively new field and provides some useful and valuable assets, such as working smart contracts for data quality assessment and an analysis of the overhead associated to them, that can be a precious resource for programmers approaching this topic, but it is still subject to some limitations. First of all, due to time restriction, the sample size is relatively

small; while this has a little or no impact on the measured cost associated to our code, we noticed that the transaction time of the exact same contract interaction was subject to large variations, mainly due to the number of transactions waiting to be mined. Since network traffic has a large impact on transaction time, a larger dataset can provide a more accurate estimation of the average transaction time on the long term. Another limit of this work is that it uses a third party oracle, where the cost and the time needed to fetch *off-chain* data are imposed by the service provider. We decided to use an existing oracle service because implementing an oracle from scratch is a very complex and time consuming activity, out of the scope of this work. Nevertheless, developing and using an oracle for the specific purpose can lead to savings both in transaction time and costs. Lastly, even if we chose two scenarios covering as many real world applications as possible, the number of cases that can use our code as it is, or with just some minor changes, is still limited.

7.4 Future Work

Future improvements on this work can include the increment of the considered scenarios, in order to expand the library of reusable code and expanding the range of considered applications. Another possible direction can be the deploying of the same scenarios in blockchains different than Ethereum, both public and private ones, in order to study how the underlying infrastructure can affect the performance of data quality controls.

Bibliography

[1] Maher Alharby and Aad Van Moorsel. Blockchain-based smart contracts: A systematic mapping study. *arXiv preprint arXiv:1710.06372*, 2017.

[2] Andreas M Antonopoulos and Gavin Wood. *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[3] Danilo Ardagna, Cinzia Cappiello, Walter Samá, and Monica Vitali. Context-aware data quality assessment for big data. *Future Generation Computer Systems*, 89:548–562, 2018.

[4] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. Medrec: Using blockchain for medical data access and permission management. In *2016 2nd International Conference on Open and Big Data (OBD)*, pages 25–30. IEEE, 2016.

[5] Massimo Bartoletti and Livio Pompianu. An empirical analysis of smart contracts: platforms, applications, and design patterns. In *International conference on financial cryptography and data security*, pages 494–509. Springer, 2017.

[6] Carlo Batini, Monica Scannapieco, et al. Data and information quality. *Cham, Switzerland: Springer International Publishing. Google Scholar*, page 43, 2016.

[7] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, pages 91–96, 2016.

[8] Cinzia Cappiello, Marco Comuzzi, Florian Daniel, and Giovanni Meroni. Data quality control in blockchain applications. In *International Conference on Business Process Management*, pages 166–181. Springer, 2019.

[9] Si Chen, Rui Shi, Zhuangyu Ren, Jiaqi Yan, Yani Shi, and Jinyu Zhang. A blockchain-based supply chain quality management framework. In *2017 IEEE 14th International Conference on e-Business Engineering (ICEBE)*, pages 172–176. IEEE, 2017.

[10] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 442–446. IEEE, 2017.

[11] Usman W Chohan. A history of bitcoin. *Available at SSRN 3047875*, 2017.

[12] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *International conference on financial cryptography and data security*, pages 79–94. Springer, 2016.

[13] Christian Esposito, Alfredo De Santis, Genny Tortora, Henry Chang, and Kim-Kwang Raymond Choo. Blockchain: A panacea for healthcare cloud-based data security and privacy? *IEEE Cloud Computing*, 5(1):31–37, 2018.

[14] Christopher K Frantz and Mariusz Nowostawski. From institutions to code: Towards automated generation of smart contracts. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 210–215. IEEE, 2016.

[15] Mudabbir Kaleem and Aron Laszka. Vyper: A security comparison with solidity based on common vulnerabilities. *arXiv preprint arXiv:2003.07435*, 2020.

[16] Anja Klein and Wolfgang Lehner. Representing data quality in sensor data streaming environments. *Journal of Data and Information Quality (JDIQ)*, 1(2):1–28, 2009.

[17] Bojana Koteska, Elena Karafiloski, and Anastas Mishev. Blockchain implementation quality challenges: a literature. In *SQAMIA 2017: 6th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications*, pages 11–13, 2017.

[18] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *ACM Transactions on Programming Languages and Systems*, pages 382–401. 1982.

[19] Vallery Mou. Blockchain oracles explained. `https://www.binance.vision/blockchain/blockchain-oracles-explained`, 2020.

[20] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[21] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

[22] Thomas C Redman and A Blanton. *Data quality for the information age*. Artech House, Inc., 1997.

[23] Richard Y Wang, Henry B Kon, and Stuart E Madnick. Data quality requirements analysis and modeling. In *Proceedings of IEEE 9th International Conference on Data Engineering*, pages 670–677. IEEE, 1993.

[24] Ingo Weber, Xiwei Xu, Régis Riveret, Guido Governatori, Alexander Ponomarev, and Jan Mendling. Untrusted business process monitoring and execution using blockchain. In *International Conference on Business Process Management*, pages 329–347. Springer, 2016.

[25] Maximilian Wohrer and Uwe Zdun. Smart contracts: security patterns in the ethereum ecosystem and solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 2–8. IEEE, 2018.

[26] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[27] Xiwei Xu, Cesare Pautasso, Liming Zhu, Vincent Gramoli, Alexander Ponomarev, An Binh Tran, and Shiping Chen. The blockchain as a software connector. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 182–191. IEEE, 2016.

[28] Jie Zhang, Nian Xue, and Xin Huang. A secure system for pervasive social network-based healthcare. *Ieee Access*, 4:9239–9250, 2016.

[29] Peng Zhang, Jules White, Douglas C Schmidt, and Gunther Lenz. Applying software patterns to address interoperability in blockchain-based healthcare apps. *arXiv preprint arXiv:1706.03700*, 2017.

[30] Guy Zyskind, Oz Nathan, et al. Decentralizing privacy: Using blockchain to protect personal data. In *2015 IEEE Security and Privacy Workshops*, pages 180–184. IEEE, 2015.

Appendix A

Source code

The source code of our smart contracts can be found at *https://github.com/N0be/msc*