University of Zurich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Design and Implementation of a SC-based System for the Tracking within a Cheese Supply Chain

Matteo Gamba
Zurich, Switzerland
Student ID: 19-752-443

Supervisor: Eder J. Scheid, Jan von der Assen, Christian Killer
Date of Submission: August 25, 2022

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Zusammenfassung

Da ein Grosser Teil der Schweizer Käseproduktion exportiert werden, stellt gefälschter Käse, der im Ausland unter geschütztem Namen verkauft wird, eine ernsthafte wirtschaftliche Bedrohung für die Schweizer Käseindustrie dar. Aus diesem Grund hat sich eine Gruppe privater und staatlicher Organisationen in der Schweiz zusammengetan, um die *CheeseChain* aufzubauen, eine Blockchain-basierte Lösung zur Erhöhung der Transparenz und des Vertrauens entlang der Wertschöpfungskette des Tête-de-Moine, einem Schweizer Käse, sowie zur Authentifizierung des Produkts mithilfe eines DNA-basierten Authentifizierungssystems und zur Veröffentlichung der Ergebnisse in der öffentlichen Blockchain (BC). In dieser Arbeit wird eine Smart-Contract-Lösung zur Verfolgung eines Käses durch seine Lieferkette und beinhaltet ein Frontend, das die Interaktion mit dem Smart Contract (SC) vereinfacht, sowie einen Server, mit dem Daten aus den bestehenden Systemen über Netzwerkanfragen in die BC geschrieben werden können. Der SC abstrahiert eine Lieferkette in Lieferkettenteilnehmer, Produktionschargen und einzelne Produktionsschritte, die an eine Charge angehängt werden können, wodurch der SC mit nur geringfügigen Änderungen des Quellcodes auch für andere Anwendungsfälle der Lieferkettenverfolgung einsetzbar ist. Dies wird durch benutzerdefinierte Datenstrukturen, so genannte Structs, für Produktionschargen und -schritte erreicht, wobei jede Charge den Zeitstempel der Erstellung, weitere Informationen über das Produkt und einen Zeiger auf den letzten innerhalb der Lieferkette durchgeführten und für die Charge registrierten Schritt enthält. Jeder Schritt selbst verweist auf seinen Vorgänger und speichert daneben eine Beschreibung des Schrittes, den Teilnehmer, der den Schritt durchgeführt hat, den Zeitstempel und die Koordinaten. Dies bildet eine unveränderliche und nur durch Anhängen veränderbare, rückverfolgbare verknüpfte Liste von Schritten, die über das benutzerdefinierte Frontend eingesehen werden kann, das für die manuelle Interaktion mit dem SC durch die Teilnehmer der Lieferkette und die Abfrage der Lieferkettenhistorie für einen bestimmten Käse durch den Kunden und die Überprüfung der Echtheit eines Produkts konzipiert ist. Darüber hinaus implementiert der SC einen Zugangskontrollmechanismus, der alle Funktionen schützt, die den Zustand des SCs verändern. Sie können nur von registrierten Teilnehmern der Lieferkette aufgerufen werden, die vom Systemadministrator verwaltet werden, der gleichzeitig der Bereitsteller des Vertrags und die einzige Instanz ist, die alle verfügbaren Funktionen aufrufen darf. Der oben erwähnte Server dient als Abstraktion über den SC und erleichtert die Injektion von Daten, die in einem privaten System verfügbar sind, um automatisch über eine API in die BC geschrieben zu werden. Die Evaluierung des Systems hat gezeigt, dass die Auswahl der am besten geeigneten BC für eine bestimmte Anwendung sorgfältig und mit Bedacht erfolgen muss, da sie sich direkt auf die Systemleistung und die Betriebskosten auswirkt, die von der Blockzeit der

ausgewählten BC bzw. dem Token- und Gaspreis abhängen. Darüber hinaus wird der vorgestellte SC aus Sicht der Sicherheit von SCs als risikoarm eingestuft, da es einen Mechanismus zur Zugangskontrolle gibt und keine monetären Anreize zur Ausnutzung des Systems bestehen. Das grösste Sicherheitsrisiko besteht in der falschen Verwaltung und Weitergabe privater Schlüssel durch die registrierten Teilnehmer der Lieferkette oder den Administrator, wodurch der Inhaber die Möglichkeit hat, fehlerhafte Informationen in den SC zu schreiben.

Abstract

With a major part of the Swiss cheese produced being exported, counterfeit cheeses selling under their protected name abroad pose a serious economic threat to the Swiss cheese industry. Consequently, a group of Swiss private and federal entities have teamed up to build the *CheeseChain*, a blockchain-based solution to increase transparency and trust along the Tête-de-Moine (a Swiss cheese) value chain, as well as provide proof-of-origin using a PCR-based system and publish the results to the public Blockchain (BC). This thesis presents a smart contract (SC) solution to track a cheese through its supply chain including a frontend to facilitate interactions with the SC and a server that allows data from the existing systems to be written to the BC via network requests. The SC abstracts a supply chain into supply chain participants, production batches, and individual production steps that are appended to a batch, which makes the SC applicable to supply chain tracking use cases other than cheese with only minimal modifications of the source code. This is achieved using custom data structures, called structs, for production batches and steps, where every batch contains the timestamp of creation, further information about the product, and a pointer to the last step performed within the supply chain and registered on the batch. Each step itself points to its predecessor and stores a description of the step, the participant who performed the step, the timestamp, and the coordinates alongside. This forms an immutable and append-only backward traceable linked list of steps that can be inspected via the custom frontend designed for manual interaction with the SC by supply chain participants and the retrieval of supply chain history for a specific cheese by the customer allowing for verification of a product's authenticity. Further, the SC implements an access control mechanism that guards all functions that change the SC's state. All functions are only callable by registered supply chain participants, which are managed by the system administrator, who is at the same time the deployer of the contract and the only entity that is allowed to call all available functions. The server provides an abstraction over the SC and facilitates injecting data available in a private system to be written into the BC automatically through an API. Evaluating the system has shown that choosing the best suitable BC for a specific application has to be done diligently and carefully since it directly implies the system's performance and operational costs, which depend on the selected BC's block time, respectively token and gas price. Further, the presented SC is evaluated to carry a low risk from a SC security standpoint, due to the access control mechanism in place and the absence of monetary incentives to exploit the system. The biggest security risk is introduced by the mismanagement and leakage of private keys by the registered supply chain participants or the administrator, which gives the holder the ability to write faulty information into the SC.

Acknowledgments

I would like to thank my primary supervisor Eder John Scheid for his invaluable assistance and insights leading to the writing of this paper. Your continuous feedback and input always gave me a clear view of the current standpoint, potentials for improvement, as well of the remaining road map for the thesis. Your support is greatly appreciated and was crucial for this thesis.

Further, I would like to express my gratitude to Prof. Dr. Burkhard Stiller, the whole Communication Systems Group (CSG) of the University of Zurich and the CheeseChain consortium for allowing me to solve this real-world economical problem as part of this work.

Finally, I would also like to thank my friends and family for providing me with a supportive environment, which has allowed me to thrive as a person and succeed in my educational career.

Contents

Chapter 1

Introduction

Blockchain (BC) offers many benefits in different areas [60]. One such area is supply chain tracking (SCT) [46], in which a BC allows untrusted parties to share data in an immutable manner containing information on (1) how the product left the producer (*e.g.*, to meet regulatory standards), (2) how was its conditions during transport (*e.g.*, temperature collected by sensors), and (3) how it arrived at the final destination. Further, such integrity is helpful for consumers and governmental bodies to verify the origin of the product they acquired and that it was not tampered with during any step in the supply chain. In the context of agricultural products such as cheese, governmental entities (*e.g.*, Federal Office for Agriculture) may also include information regarding tests conducted on the specific product. For example, the test results show (*e.g.*, positive or negative) if the cheese was made with the correct bacteria culture sent to the cheese makers.

Counterfeits of Swiss cheeses with protected designation of origin (PDO) are seen as a real threat to the Swiss cheese industry [50]. It is estimated that 10% of the Swiss Emmentaler AOP sold worldwide happen to be counterfeits, which causes a yearly loss of 20 Million Swiss Francs [10]. A group of Swiss private and federal entities proposed to create a solution to address that, called the CheeseChain solution. It is a Blockchain-based solution to target the problem mentioned above and increase transparency and trust within the Tête-de-Moine PDO value chain using a DNA-based intrinsic product authentication system [17].

1.1 Description of Work

This thesis aims to design and develop a BC-based Smart Contract (SC) that will contain public information regarding the cheese, *e.g.*, proof-of-origin test results, production steps, and stakeholders involved in the production. Further, as the goal is to increase transparency in the supply chain, the system contains an easy-to-use web-based frontend platform for consumers to verify their product using a unique identifier of the cheese, *e.g.*, the lot number.

By improving fraud detection and in-transit identification, the system will allow instant access to a verifiable audit trail from the milk's processing to the consumer, benefiting all actors in the value chain. Fromarte, the umbrella association of Swiss artisan cheese makers, foresees this project as a future-oriented solution for their quality management and an enhancement of the cheeses produced within Swiss artisan structures. By integrating with their widely used quality management system, our approach will benefit from the option to validate adherence to prescribed guidelines. This thesis is conducted within the CheeseChain project [17], a project developed in conjunction with the Swiss Confederation's center of excellence for agricultural research (Agroscope). It targets developing and implementing a platform to improve transparency and trust along the Tête-de-Moine value chain. This significant advance for proof of origin, immutability and transparency is achieved by combining biological data with storage in a BC and partner-specific databases, *e.g.*, from Fromarte.

Lastly, an evaluation and discussion of the working implementation are performed considering the fulfillment of requirements and possible areas of improvement (*e.g.*, costs of interaction and frontend usability).

1.2 Thesis Outline

Following the introductory and motivational information, some theoretical Background about BCs, Smart Contract (SC) languages and the SC interaction solution is given in Chapter 2. This chapter forms the basis for understanding the subsequent design and implementation. Chapter 3 discusses the current state of the art regarding SC and BC solutions around supply chains documented in peer-reviewed papers. With that knowledge, a suitable technology stack (BC and SC language) is chosen. Chapter 4 presents the design and implementation of the architecture of the overall system, including the SC, a backend and a frontend. The developed system is then evaluated in Chapter 5 in terms of cost, performance and security. Subsequently, Chapter 6 concludes the thesis. It summarizes the work and proposes suggestions for improvement of the system.

Chapter 2

Background

This chapter forms the technical and theoretical basis and aims to provide all the background information needed to comprehend later chapters of the thesis. More specifically, the BC and SC concept, related technical details and the CheeseChain project are presented.

2.1 Blockchain and Distributed Ledger Technologies

As stated in [4], "*Blockchain at its core is a peer-to-peer distributed ledger that is cryptographically secure, append-only, immutable (extremely hard to change) and updateable only via consensus or agreement among peers*". Hence, a BC is essentially a distributed database that allows a community of users to store all digital events (transactions) and shares them among all participants [27]. Under the regular operation of the BC network, no transaction can be changed once included in a block [68].

The idea of BC was first introduced in 2008 with the Bitcoin Whitepaper written under the pseudonym Satoshi Nakamoto [51]. Since then, the interest in the technology has been increasing given its central attributes: Security, anonymity and data integrity without the interference of a third party. Despite attracting the attention of many different industries, the financial industry currently dominates the mindshare in the blockchain space [69].

2.1.1 Technical Structure

The term BC describes the technical structure of the system. It is a chain of blocks, where each block is linked to the previous block's cryptographic hash, making it tamper evident [67]. The block hash can be considered a unique identifier for a certain block obtained by hashing all the block's content. Changing a transaction within a block changes the block hash [31] and disconnects the parent from the child block since the block hash that the parent is pointing to does not exist anymore. Figure 2.1 visualizes a typical example of this structure.

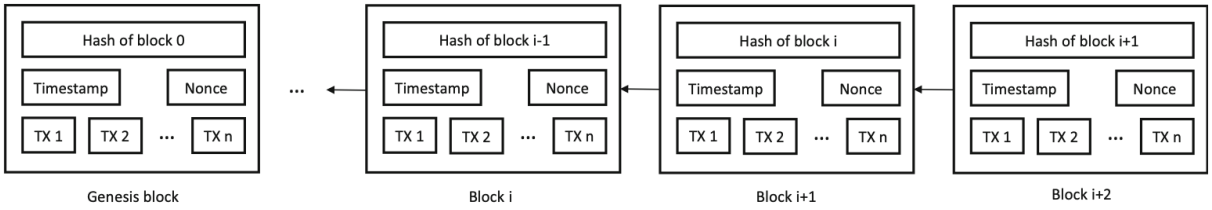


Figure 2.1: Chain of Blocks [68]

In the Bitcoin system, each block consists of six parts (see Figure 2.2): A 4-byte version number, the SHA-256 hash value (PreHash) of the previous block, the Merkle root, timestamp, nonce and Target Hash obtained after verifying all transactions [70]. The BC is extended block by block and represents the entire transaction history [55].

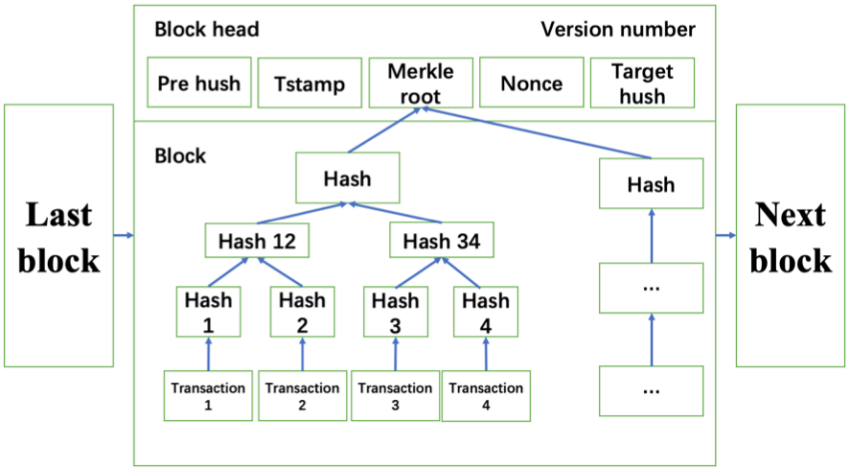


Figure 2.2: Block Structure [70]

Transactions are created and exchanged by peers of the BC network and are the only way to modify its state. Typical types of transactions are monetary exchanges or the execution of SCs, essentially arbitrary pieces of code deployed onto the BC [67]. In the example of Ethereum, transactions are network messages that include a sender, recipient, value (amount in Ether) and data payload. The sender and recipient must be specified, while the value and the data payload can be left empty [3].

2.1.2 Public Key Cryptography

BCs use cryptographic proofs instead of trusting a third party for two willing parties to execute a transaction over the Internet. Every transaction is protected through a digital signature leveraging public-key/asymmetric cryptography [36]. Every transaction is sent to the *public key* of the receiver and digitally signed with the *private key* of the sender, creating a digital signature [27].

Public and private keys always come in pairs. In the case of Ethereum, they form an *Ethereum Account* controlled by everyone possessing the private key. This is the case

because a public key is always derived from the corresponding private key through an asymmetric algorithm [3]. More specifically, in Ethereum, only the last 20 of the 42 character long public key are used to create the *Ethereum Address*, which can be identified by the *0x* prefix [11]. Therefore, the public key derives the address, essentially the public account identifier every participant can see in the network. The private key is the password protecting the account.

These mathematical functions used in BC applications have a unique property: They are easy to calculate but hard to reverse. Every public key is derived from the private key, but the private key can not easily be derived from the public key. This characteristic enables the creation of unforgeable digital signatures. For a transaction to go through, the sender must prove the ownership of the *private key* [3]. Bitcoin and Ethereum use the Elliptic Curve Digital Signature Algorithm (ECDSA), which takes a message, the private key and a random secret as input. Reversing the process reveals the signer's public address and the message, while the private key and the secret remain unknown [73]. Elliptic curve mathematics (ECDSA's super set) allows for *anyone* to verify a transaction or message by comparing the digital signature to the public address and the transaction details. This process proves that the transaction could have only come from somebody possessing the private key behind the public address. [3].

2.1.3 Deployment Types

BCs can be classified regarding their deployment type regarding read and write permissions. The classification is done across two dimensions: permissioned vs. permissionless and public vs. private [59]. The former relates to data visibility (*i.e.*, which information can be viewed by whom). The latter describes data writability (*i.e.*, who is allowed to append data and change the state of the BC) [67]. Among these two dimensions, four categories emerge [59]:

- **Public permissionless BCs:** All peers have read/write rights.
- **Public permissioned BCs:** All peers have read rights, but writing is limited to selected peers.
- **Private permissionless BCs:** Only selected peers within a closed network have read-/write rights.
- **Private permissioned BCs:** Read/write access is managed by a centralized organization.

Bitcoin and Ethereum are examples of public permissionless BCs. Since they are open and decentralized, any peer can join and leave the network as a reader or writer at any time [67]. In contrast to permissionless BCs, where no central authority exists and every peer stores an identical copy of the current state of the BC, private permissioned BCs have a central authority that has the right to overwrite and roll back any transaction. Furthermore, the master copy of transaction records is not distributed among all participants [49] but

instead maintained by a selected set of nodes. A permissioned BC is advantageous in specific business applications since it is better at maintaining privacy and fitting business governance needs than a permissionless BC [1].

2.1.4 Consensus Mechanism (CM)

In distributed computing, distrust among network participants is inevitable. To ensure network reliability and consistency, the participants, so-called nodes, negotiate to agree upon a state of the BC, known as consensus. This includes an agreed understanding of account balances and transaction orders and is achieved through relevant protocols, or so-called consensus mechanisms [70]. Most importantly, it prevents users from *double spending* their coins and makes it difficult to attack a network [23]. Since every new transaction aims to alter the state of the BC, CMs are crucial for new blocks to be created and appended to the BC [2]. There exist several different CMs *e.g.*, Proof of Work (PoW), Proof of Stake (PoS), Delegated Proof of Stake (DPoS) and Proof of Authority (PoA). Out of all options, most existing BCs leverage the computationally expensive PoW algorithm [40]. The following paragraph describes selected consensus algorithms.

Proof of Work (PoW): The core idea of the PoW CM is to ensure data consistency and consensus security by allowing the computing power of the distributed network nodes to compete [70]. These validating nodes are called *miners*, and the process is known as *mining*. Each miner is constantly calculating the block header's hash in a trial and error approach [71]. The goal is to find a hash value that is smaller or equal than the *target hash* by hashing the block header values iteratively with a random number, the *nonce*. The BC network defines the target hash, which implicitly defines the mining difficulty. The lower the target, the higher the difficulty since fewer hash values match the criteria [43]. Once a node succeeds, it broadcasts the block to the entire network to be appended. Each node must confirm the correctness of the nonce first before appending the block to their local copy of the BC [71]. In PoW, it is hard to find the right solution, but easy to verify it. The values in the header are hashed and compared to the target hash [7]. Since mining is time and resource-consuming, an incentive mechanism exists, rewarding the miner with the BCs' native token [54]. [70] describes the mining process as follows:

1. Each node selects a certain number of transactions from the current memory pool.
2. Verify the legitimacy of the selected transaction and then package it.
3. Find the right nonce, so that Hash(PreHash, Merkle Root, Timestamp, Nonce) $\leq$ Target Hash.
4. Upon success, create a block and broadcast it to the network.

Should more than one miner find the nonce which meets the requirements, the BC undergoes *bifurcation*. Figure 2.3 visualizes how two *forks* coexist until one is eventually cut off. Bitcoin follows the principle of the longest chain, which requires six more blocks to be successfully appended to the current block before a fork is recognized [70].

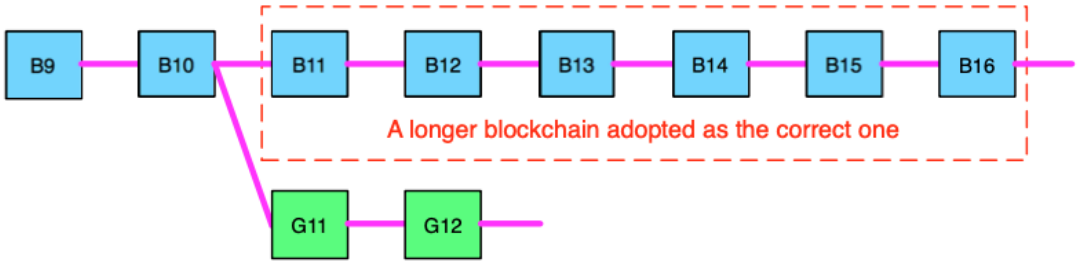


Figure 2.3: A Scenario of BC Branches [71]

The pace at which Blocks are appended is network-specific. Bitcoin has a *block time* of 10 minutes while Ethereum averages around 15 seconds [31]. As more miners join a network and computers become faster, blocks are found faster. To ensure a predictable, average block time of 10 minutes in Bitcoin, the target hash and, therefore, the difficulty is automatically adjusted every 2016 blocks [43].

Proof of Stake (PoS) was proposed in 2011 in a Bitcoin talk forum as an alternative solution to the resource-intensive PoW mechanism [70]. Instead of demanding miners to find a nonce in unlimited numerical space, a validator for the next block is randomly selected proportionally to the node's stake of native tokens in the network [48]. The rationale is that nodes with a higher stake in the network are interested in the network's success and therefore are unlikely to perform malicious acts and attacks on the BC [71]. If a validator were to approve a malicious block, he/she would lose the entire stake. In the case of Ethereum, a stake of 32 Ether is needed to qualify as a validator node. In PoS new blocks are not mined but *forged* or *minted* and validating nodes are called *forgers* [57]. Some benefits of PoS over PoW are higher decentralisation, security and speed [22, 70].

Delegated Proof of Stake (DPoS) is a variation of PoS that seeks to reach consensus more efficiently by speeding up transactions and block creation without compromising the decentralized incentive structure of the BC. In DPoS, nodes 'vote' to select a *witnesses* (users they trust to validate transactions), from which the ones with the most votes earn the right to validate transactions. The primary incentive against malicious behavior for a witness is their loss of income and reputation. Votes are weighted according to the size of the voter's stake and can be delegated to other users to choose a trustworthy witness on their behalf. Users of DPoS systems can vote for another group of users, the *delegates*, a trusted party responsible for maintaining the network. They oversee the governance and performance and can propose changes to the protocol, like altering the block size or rewards for validations. Lisk, EOS, Steem, BitShares and Ark are examples of BCs that have adapted the DPoS CM [22]. Overall, DPoS dramatically reduces the number of participating nodes in verification and therefore reaches consensus at a much higher pace than PoW and traditional PoS [70]. However, all of that comes at the cost of more centralization [59].

Proof of Authority (PoA) PoS algorithms work under the assumption that validators are incentivized to act in the network's interest since they would be losing their stake otherwise. Therefore, it makes sense that a more significant stake automatically makes the node more trustworthy. However, this assumption is not entirely correct. Two equal stakes might be equal from a monetary perspective, but they could be valued differently by

their holders. This difference in perceived value is what PoA aims to improve. Instead of tokens, network participants stake their identity, which can help ensure that all validators have an equal incentive for the network's success [9].

In PoA, every round, one of the N authorities is elected as the *mining leader* in charge of block validation. For this mechanism to work properly, at least $\frac{N}{2+1}$ nodes must act honestly. In short, PoA improves performance, cost, scalability and power consumption but compromises decentralization, privacy and censorship resistance [33, 32]. Current projects using PoA are Ethereum's Kovan Testnet, PoA Network (an Ethereum sidechain) and VeChain's VeChainThor platform.

2.2 Ethereum

Like Bitcoin, Ethereum is an open-source BC network that allows to build an economic system in software and provides account management and a native unit of exchange, so-called *tokens*, out of the box [31].

Ethereum shares many common elements with other open BCs: A peer-to-peer network connecting nodes, a PoW CM and cryptographic primitives, such as digital signatures, hashes and a digital currency (*ether*). Unlike Bitcoin, Ethereum's primary purpose is not to be a digital currency payment network. Ethereum is often also described as *the world computer*. It is a globally decentralized computing infrastructure that executes programs called *Smart Contracts* on a *virtual machine*. A BC network is used to synchronize and store the system's state changes, along with *ether* as a form of payment and constraint for execution resource costs. *Solidity*, Ethereum's programming language is Turing complete, meaning that Ethereum has everything needed to function as a general-purpose computer [3].

2.2.1 Smart Contracts (SC)

BCs have various features that are not shared by all implementations to the same extent. Since Ethereum's main feature is the ability to execute business logic, this chapter will explore how that is realized.

In Ethereum terms, that equals having a SC running on the *Ethereum virtual machine (EVM)*. [31] defines SC as some business logic that runs on the network, semi-autonomously moving value and enforcing payment agreements between parties. They are similar to *classes* in conventional object-oriented programming.

Smart Contracts Languages

Programming a BC has many similarities with traditional programming. A programming language, more specifically a *Smart Contract Language (SCL)* is needed to express the

Table 2.1: SCLs Overview [66]

Language	Blockchain	Syntax similar to	Built with	Turing complete
Solidity	Ethereum	JavaScript	C++	+
Vyper	Ethereum	Python	Python	-
Go	Hyperledger Fabric Neo	-	Rust	+
C / C++	EOS Solana	-	Rust	+
C#	Neo	-	Go	+
Rust	Solana Near Polkadot	-	C / C++	+
Haskell	Cardano	-	GHC	+
Clarity	Bitcoin	Lisp	C#	+

logic to be executed, which can be achieved with various toolsets. Table 2.1 gives an overview over selected programming languages.

Among the SCLs listed in Table 2.1 some were created explicitly for BC development, while others were adjusted from existing languages to work with SCs. When choosing the right SCL, there is no absolute answer. It depends on which BCs support the SCL, which traditional programming languages the developer is familiar with, and whether the SCL is Turing complete or not. A downside of Turing completeness is that there is no theoretical guarantee for how much time is needed to solve a given problem. Since Ethereum executes SCs serially, an infinite loop would prevent other SC from being executed. This problem is solved in Ethereum by setting a *gas limit* which essentially bounds the maximum complexity of a single execution. Furthermore, Turing incomplete SCLs support much more in-depth static analysis and enhanced security. As a result, *Turing incompleteness* has become the new *feature* of SCL [66].

Smart Contract interaction Solutions

In decentralized applications (*dApps*) SCs serve as the backend providing the business logic [21]. To write data to the BC and send a transaction, we need a way to interact with the SC. While transactions can theoretically be initiated from any node in the network, most users typically interact with SCs on the BC via the web browsers, contacting another node in the background. This interaction can be achieved either via a browser extension like *Metamask* or programmatically from code running in the browser by leveraging libraries like *Web3.js* or *Ethers.js* [3, 65]. These libraries allow users to interact not only with SC but with the BC as a whole (*e.g.*, read data, make transactions and deploy SCs) [18].

Solidity Example

```
pragma solidity ^0.8.4;

contract Coin {
    // The keyword "public" makes variables
    // accessible from other contracts
    address public minter;
    mapping (address => uint) public balances;
    // Events allow clients to react to specific
    // contract changes you declare
    event Sent(address from, address to, uint amount);
    // Constructor code is only run when the contract
    // is created
    constructor() {
        minter = msg.sender;
    }
    // Sends an amount of newly created coins to an address
    // Can only be called by the contract creator
    function mint(address receiver, uint amount) public {
        require(msg.sender == minter);
        balances[receiver] += amount;
    }
    // Errors allow you to provide information about
    // why an operation failed. They are returned
    // to the caller of the function.
    error InsufficientBalance(uint requested, uint available);

    // Sends an amount of existing coins
    // from any caller to an address
    function send(address receiver, uint amount) public {
        if (amount > balances[msg.sender])
            revert InsufficientBalance({
                requested: amount,
                available: balances[msg.sender]
            });
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        emit Sent(msg.sender, receiver, amount);
    }}
```

Listing 2.1: Solidity Example [25]

The Solidity code described in Listing 2.1 is retrieved from the official Solidity [25] website and is an example of what Solidity code looks like. It is the simplest example of a token that can be created on the Ethereum BC and nicely introduces relevant concepts.

As with classical classes, this SC's lifetime starts with the constructor's execution. In this case, the function assigns the address of the creator (contract or person) of the contract to the variable minter. This variable has the type `address` which is a 160-bit value that does not allow any arithmetic operations and is used to store contract or account addresses. The keyword `public` exposes the variable to other contracts and users. `msg`, `tx` and `block` are special variables that provide access to BC values.

The `mint` function takes an address and an unsigned integer (uint) as parameters and assigns the specified amount of new tokens to the desired address. The tokens belonging to a specific address are not stored on the address but directly in the token contract.

`Mapping` is a more complex data structure that serves this purpose perfectly and can be thought of as a hash map. As specified in parenthesis in the variable `balances`, an address is mapped to a uint, the token amount of the corresponding address. `Require` serves as a guard that reverts the transaction if the condition is not met.

The `send` function implements the logic of transferring tokens from one address to another and emits an *event* by doing so. The event has to be specified in the contract and allows clients or block explorers to be notified when certain things on the BC happen.

An `error` is similar to the `require` function but is a way to provide more detailed information why a transaction did not complete [25].

2.2.2 Ethereum Addresses

An Ethereum account is an entity with an ether (ETH) balance that can send transactions on Ethereum. Accounts can be user-controlled or deployed as SCs. Ethereum has two types of accounts:

- **Externally Owned Accounts (EOAs):** Controlled by anyone who has the private key.
- **Contracts:** A SC deployed to the network, controlled by code.

Accounts always come with their corresponding 42-character-long address, serving as their identifier within the network. For EOAs, this represents the last 20 bytes of the public key, prefixed with 0x. The contract address is derived from the deployer address and the number of transactions originating from that address, the *nonce* [19].

2.2.3 Gas and Fees

Gas fees exist to secure the Ethereum network from bad actors spamming it by charging a fee for every computation on the network. The fees are paid in ETH, and gas prices are denoted in gWei, a denomination of ETH and equal to 10^{-9} ETH.

Gas represents the fundamental unit of computation. To avoid accidental or hostile infinite loops, every transaction must set the maximum amount of gas it wants to use.

Transaction fees can be calculated as presented in Equation 2.1. The *base fee* represents the minimum amount of gas needed for a transaction to be eligible for inclusion in a block. At the same time, the *tip* (also called *priority fee*) is the miner's reward for including it in a block.

$$GasUnits(limit) \times (BaseFee + Tip) \tag{2.1}$$

Gas limit allows the user to set a maximum amount of gas he/she is willing to consume on a transaction. More complex transactions involving SCs require more computational work and a higher gas limit than a simple payment (21'000 units).

To emphasize the irreversible nature of BC transactions, we quote from [20]:

"*For example, if you put a gas limit of 50,000 for a simple ETH transfer, the EVM would consume 21,000, and you would get back the remaining 29,000. However, if you specify too little gas, for example, a gas limit of 20,000 for a simple ETH transfer, the EVM will consume your 20,000 gas units attempting to fulfill the transaction, but it will not complete. The EVM then reverts any changes, but since the miner has already done 20k gas units worth of work, that gas is consumed [20].*"

In times of high network congestion, the base fee rises to lower the congestion. At the same time, miners prioritize transactions with a high tip. As a result, we experience spikes in transaction fees during times of high network usage [20].

2.3 The CheeseChain Project

Since the design and implementation of a SC-based SCT solution for the CheeseChain project is the primary focus of this work, this section provides a background on the CheeseChain project and its stakeholders.

The CheeseChain is a BC-based application to enhance transparency and trust among the Tête-de-Moine PDT value chain by combining DNA-based proof-of-origin laboratory tests, data from partner-specific databases and BC technology.

This combination is integrated with a BC-based approach for the first time, which produces a tamper-proof and immutable audit trail. The increased transparency and verifiability of the published data guarantee the authenticity of the food product, here cheese. The in-product DNA-based certification system developed by Agroscope has been used by Tête-de-Moine since 2013.

Including information on material flows and laboratory data will allow for efficient sampling by the trademark owner and simplified access for authorities, distributors and eventually, customers [17].

Figure 2.4 illustrates three groups of stakeholders served by the CheeseChain. The most important ones are the above-defined primary stakeholders, followed by national and international suppliers, retailers and consumers, as well as the responsible parties for custom clearance [41].

The primary stakeholders of this project are *Agroscope*, the federal competence center for agricultural research, *Interprofession Tête-de-Moine*, the trademark owner of Tête-de-Moine PDT and *Fromarte*, the umbrella association of Swiss artisan cheese makers. The latter foresees this project as a future-oriented solution for their quality management and enhancement of the cheeses produced within Swiss artisanal structures [41].

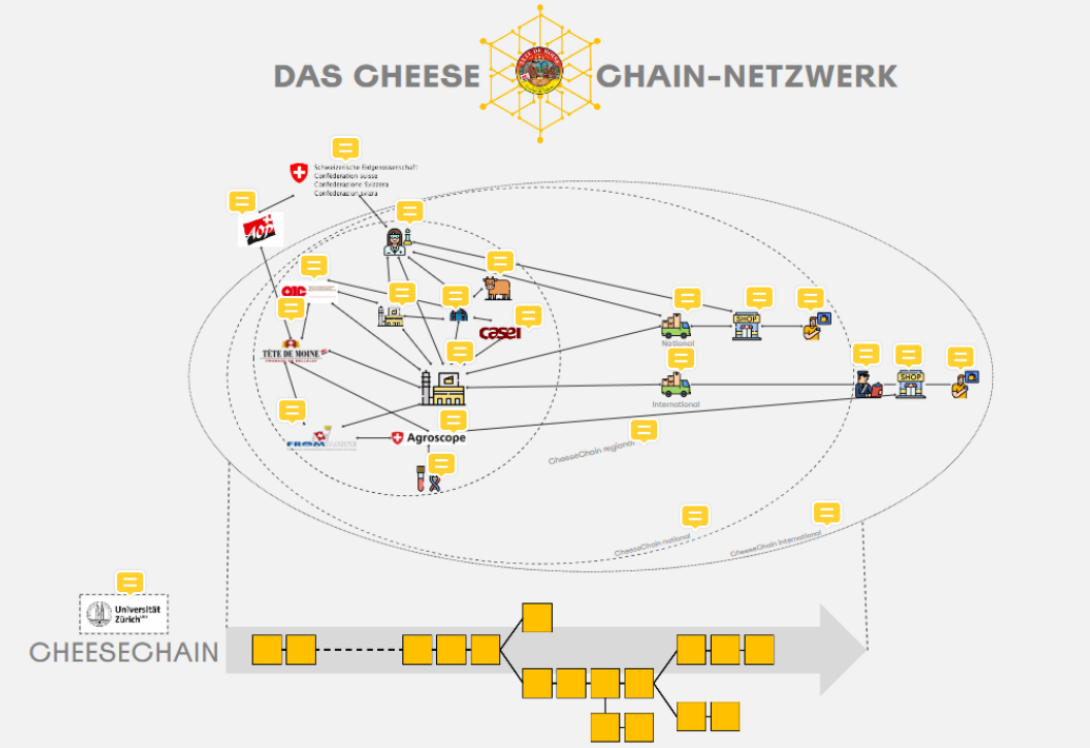


Figure 2.4: Illustration of the CheeseChain Network [41]

Chapter 3

Related Work

This chapter aims to identify, organize and analyze papers in peer-reviewed journals about BC in a supply chain context. The focus lies in understanding the main BC applications in supply chain research and the significant disruptions and challenges.

3.1 Solutions

[8] proposes to use BC and SCs to tackle the traceability of temperature-sensitive pharmaceutical goods *e.g.*, vaccines. The work uses the following elements to check the recorded temperature data during transport against regulatory requirements and make the results publicly available:

- **Ethereum Blockchain Network:** is used to record and verify temperature data.
- **Smart Contract:** is issued for each shipment and used to compare the recorded data against regulations.
- **Database:** a relational database to store raw temperature data and user credentials.
- **Server:** lies between the BC network and frontend users, creates and modifies SCs and stores data in the database.
- **Mobile Devices:** are used by end-users to register new shipments and track/send temperature data records to the server.
- **Sensors:** based on Bluetooth technology that record and send thermal data in a fixed interval to a mobile device.

As a first step, the sensors and shipment must be associated. This is done by storing the sensor's MAC address and the shipment's track-and-trace number in the database and in the Internet of Things (IoT) device's storage. These numbers are encoded as a Quick Response (QR) code, retrieved by scanning with a mobile device and sent to the

server. The server configures a new SC with the corresponding regulatory temperature requirements for every new shipment containing medical products and deploys it. Now the sensor can be placed into the packet to record the temperature every 10 minutes and store it in the internal memory.

After the shipment is received at the destination, the track-and-trace number is scanned. The mobile client sends it to the server, which responds with a MAC-Address. Then the client automatically downloads all the raw temperature data via Bluetooth and sends it to the SC. The SC evaluates the regulatory compliance and stores the result on-chain. For cost-efficiency reasons, only a URL that points to the raw temperature data stored in a DB is stored inside a contract. Figure 3.1 visualizes the proposed architecture. Within such a context, [61] further expands the architecture by introducing a BC-agnostic interoperability API so that the solution can work with different BCs without requiring specific knowledge.

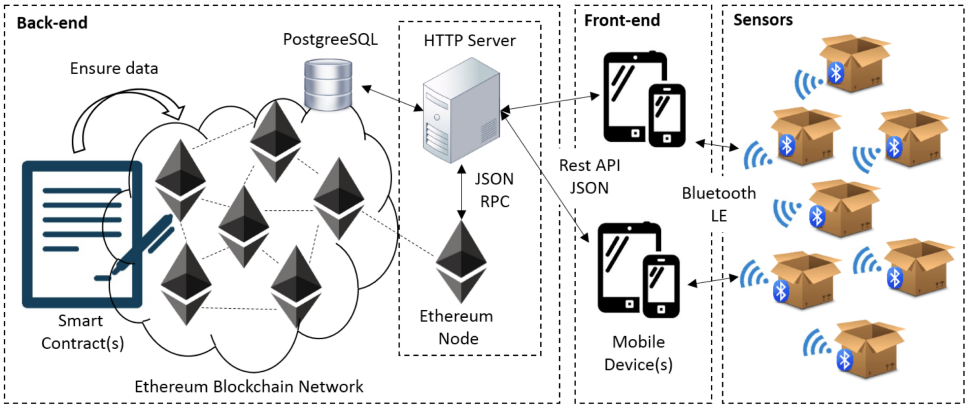


Figure 3.1: Modum.io BC Architecture [8]

FoodChains [30] tackles the challenge of SCT of food with the help of BC. More specifically, it tries to increase transparency, trust, digitization and sustainability within the value chain of Swiss dairy products. The project is conducted by the Communication Systems Research Group (CSG) at the Department of Informatics at the University of Zurich in collaboration with Molkerei Fuchs, a Swiss milk producer. The proposed system records raw product, transport, manufacturing, and sold product data, enabling geographical and temporal traceability of milk and milk products, thus increasing trust in the brand. For cost-efficient data storage, only hashes of their raw data are stored on Ethereum, their BC of choice, while the complete data is accessible in an off-chain DB. This information is accessible to all stakeholders within the supply chain, and as configured to the public. An Android Application with an integrated QR-code scanner is used to communicate with the system.

The authors of [58] propose a high-level solution to food fraud within the food supply chain and implemented a simple SC on the Ethereum BC as a proof of concept. They claim that tampering or misrepresentation of food occurs the most in milk-related products, accounting for around 49 billion dollars of cost, and can be attributed to 420'000 deaths yearly. The leader in regards to unreported origin foods is seafood with 24-36%. To increase transparency, efficiency, security and food safety, they propose the following 4-stage system:

1. *Stage 1:* Farmers store food data on the BC (*e.g.*, type of crop, procedure used for sewing, manufacturing, time and date).
2. *Stage 2:* Food processors place bids through a SC and store processing data on the BC.
3. *Stage 3:* Wholesalers place bids through a SC for processed food and update transportation information details such as time and location.
4. *Stage 4:* Consumers verify the history on-chain.

Within the same context of food, [5] implemented a prototypical system to identify and verify the quality of agricultural products. The Ethereum BC was used to ensure transparency and allow anyone to access the network. Instances of the proposed SC were created for each physical product and deployed to the BC. A QR code containing the Ethereum address of the SC was used as the reference to the virtual instance of the product. With every transaction, the ownership and location of the product were updated, enabling traceability through the supply chain. In addition, they propose a token-based mechanism that indicates the farmers' reputation, which serves as a solution for the misuse of paper certifications. Farmers could place a certification request for every process and gain tokens for each certification done by peers (peer farmers, agricultural officers, or related persons). Finally, consumers are eligible to trace back a product and identify every owner throughout the supply chain via their public Ethereum address and rate the product quality.

While the previously described papers primarily focus on traceability and sparingly on the credibility of actors within the supply chain, [64] developed a complete solution that ensures not only *traceability* but also *trust* and a smooth *delivery* experience. Different from storing off-chain data in private BCs like in the case of [30], [64] upload it all to the Interplanetary File System (IPFS) and store the IPFS hash of that data on the BC. This approach decentralizes the data storage and provides an efficient, secure and reliable solution. Ethereum and its platform native SCs assure an efficient, secure and trusted environment for supply chain activities. The system categorizes the actors within the supply chain in the following entities: *Farmer, processor, distributor, retailer, consumer, logistic company (LC), and arbitrator.* Figure 3.2 visualizes the architecture of the proposed system. The arbitrator sits at the same level as the storage layer to represent him as an off-chain entity. His job is to monitor and manage the network, which includes handling disputes.

Traceability is achieved through the use of three SCs, *i.e.*, registration contract (RC), add to lot contract (ALC) and add transaction contract (ATC). The RC registers the supply chain entities and the products available to each entity. This product registration process contains the address of ALC, where additional lot and product details are specified. Lastly, ALC has a reference to ATC. Every entity has its own set of transactions that can be performed. In the case of a malicious entity, only the arbitrator has the authority to remove it.

In order to *trade* registration through the RC must have taken place. After that, a trading contract between the seller and the buyer can be initiated. The buyer specifies the product

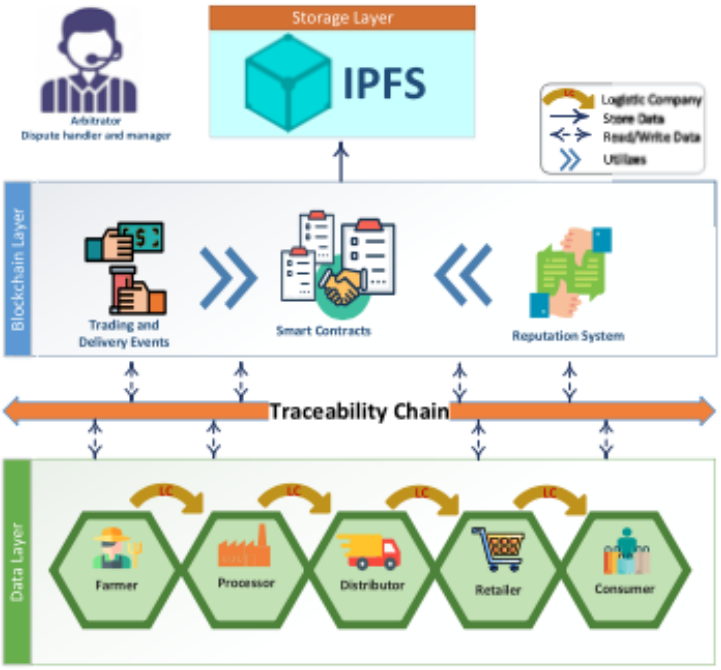


Figure 3.2: BC-Based End-to-End Solution for Agricultural and food supply chain [64]

he is interested in and the product code, and the seller uploads a current picture of the product and the price on IPFS. Next, both parties must deposit a security amount into the contract as punishment in case of a dispute. In case of dispute, all funds are sent to the arbitrator and distributed according to the off-chain settlement.

The system further includes a process to evaluate the trustworthiness of sellers through a *reputation contract* triggered after every trade event. The buying entity can request that stored reputation before every trade. All supply chain entities except the farmer and the end customer act as buyers and sellers and profit from the BC-stored immutable reviews.

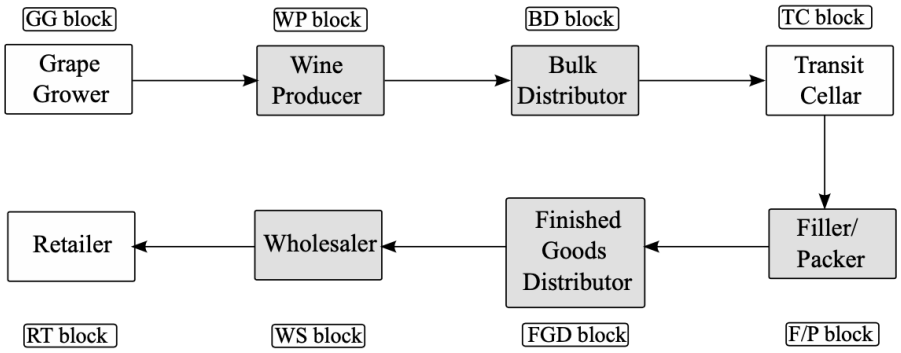


Figure 3.3: Data Flow Between Entities [6]

Still within the context of food supply chain traceability, but with the focus on anti-counterfeiting [6] designed a prototype that benefits the wine industry. Wine counterfeiting is very prominent, while the most prevalent type of fraud is the relabelling of cheaper wines to expensive and highly collectible ones. The implementation works on a private

BC, where a subset of the entities within the supply chain are part of the consensus and act as miners. Figure 3.3 shows the involved entities and data flow throughout the system and highlights the mining entities in gray color.

Every transaction is represented in its block, with the grape grower generating the genesis block. Every individual bottle of wine can be traced back by its unique id that points to the batch number, which traces back every step until the genesis block is reached. The purchase information of every bottle is recorded, ensuring that no ID can be sold twice. Data is stored in each block either as plain text or as cypher text which is encrypted and decrypted with a shared secret key distributed among all entities in the system.

[72] defines *traceability* as the core component in preventing counterfeiting of products. Their implementation focuses on information storage, inquiry and anti-counterfeiting along a medication supply chain. According to the World Health Organization (WHO), 10% of medications sold worldwide are counterfeit, and the percentage is as high as 30% in some developing counties.

As a solution [72] proposes a BC-based system where, equally to the last described solution, each transaction represents a block that includes the BC addresses of the two involved parties in the trade, the transaction time and the specific contents of the transaction. After a medication has passed the quality inspection in the manufacturing stage, the manufacturer publishes information about the product on the BC. Following the creation of the genesis block and the quality inspection, the medicine goes through the hands of intermediary entities before reaching the consumer. These intermediary entities, such as wholesalers and retailers, can update the medications' current status through the BC. All stages in the supply chain can then be queried by the authorized entities, while regulators join the network to monitor and verify all relevant mechanisms. An access control policy model based on SCs prevents the information from being altered or disclosed by unauthorized entities. In order to be assigned a role, the entities along the supply chain must register for a role first.

3.2 Comparison and Discussion

The above-cited papers have successfully introduced systems and mechanisms to store relevant supply chain data in an immutable manner on a public or private BC, together with different access control mechanisms. The proposed solutions range from very vague frameworks to very practical and specific implemented prototypes. Their overview is presented in Table 3.1.

Four out of the six presented works chose to use Ethereum as the underlying infrastructure powering their application, out of which the two more sophisticated solutions implemented some off-chain elements into their system for cost-efficiency purposes. They only store the hash of the full data on-chain. This approach maintains immutability while bringing down storage costs. While [30] used a private DB for storing the large amount of data, [64] pursued the decentralized ideology and chose IPFS instead. Whether it is better to use a public or a private BC approach cannot be defined but depends on the goal of the

Table 3.1: An Overview of Related Work

Ref	Use Case	Blockchain	Off-chain Elements	Interaction Type
[8]	SCT & regulatory compliance	Ethereum	Storage of raw data & user credentials	QR codes, sensors and Android app
[30]	SCT	Ethereum	Storage of full data	QR codes and Android app
[58]	Food Safety & trading	Ethereum	-	Through SC
[5]	Food SC	Ethereum	-	QR codes and user interface
[64]	SCT, trading & accountability	Ethereum	IPFS & disputes	not specified
[6]	SCT & anti-counterfeiting	Private BC	-	Custom GUI
[72]	SCT & anti-counterfeiting	not defined	-	web client

particular system. Public BC have relatively high transaction and storage costs but no infrastructure costs; the opposite holds for their private counterparts [42, 45].

The more elaborate systems like [30, 64, 6, 72] stand out by providing or proposing a user interface to connect to their system instead of relying on communication through SCs directly. QR codes were used in [30, 5] to simplify the interaction further.

Traceability and transparency are the main focus points in BC-based supply chain solutions. These efforts benefit supply chains in terms of improved efficiency of transactions and storage as well as sustainability and security. The temporal and geographical traceability further improves trust in brands and among supply chain entities. Research by [58, 64] has shown that trading mechanisms can be constructed that encourage trading even in low-trust environments. The solutions proposed come at the cost of decentralization, as a centralized entity handles disputes. Decentralized dispute handling would solve that problem while possibly introducing new magnitudes of complexity to the system.

As a solution to increase transparency and trust benefiting transactions within the supply chain, a token-based trustworthiness score and a SC-powered review system are proposed. Granting these trust-tokens based on reviews from off-chain or entities outside of the supply chain exposes the system to the risk of corruption and manipulation.

However, maliciously acting supply chain entities might still be incentivized to inject false information into the BC. This poses a significant threat to the trustworthiness and credibility of the available information on-chain.

Under the assumption of trustworthy supply chain entities participating on-chain, these systems provide excellent technical solutions for enhanced supply chain traceability and anti-counterfeiting possibilities. However, no mechanism is specified to connect a physical

product to the digital on-chain twin. In other words, if the on-chain data is correct, but the physical product is a forgery that claims to be the product traceable on-chain, the systems would lose all its value-added. The authors of [47] introduce a solution for that issue using the example of counterfeits of 3D printed manufacturing materials. They printed the QR code linking to the transaction hash, identifying the physical product on-chain in fluorescent material. Information about the fluorescent color is stored together with the other data on-chain. It can be verified with a standard phone camera setup, which detects the color, quantifies it, and compares it to the stored data. This is one of many possible solutions for physical signatures connecting digital representations securely and trustworthy to their physical counterparts.

Chapter 4

Design and Implementation

This chapter describes the application scenario of the designed system within the CheeseChain in Section 4.1 to understand the requirements and evaluate the value provided. Further, Section 4.2 aims to give an overview of the system architecture and explain each system component's design. Finally, Section 4.3 explains the associated source code and the technologies used in development.

4.1 Application Scenario

The implementation is based on a specific SCT scenario in the production of the Tête-de-Moine PDO cheese, tailored to serve the main stakeholder of the system, the customer and other stakeholders involved in the production process.

Figure 4.1 shows the primary use case of the system. A **stakeholder involved in production** of the cheese, *e.g.*, milk or cheese producer records information about the production step in Fromartre's quality management tool. This information is recorded within a form for each lot of cheese and contains information about every production step and additional information. Once the production step is completed, the form is *frozen* and cannot be modified. This is when the system will be called with the information needed. From that point, the lot number printed on the product package can be used to retrieve details about the production process.

After the cheese is produced and has entered the shelves of grocery stores, **Agroscope** picks a sample and conducts laboratory tests on the cheese. Of primary interest is the result of the PCR test that compares the bacteria culture of the sample cheese with the one Agroscope provided for the production of the cheese. A mismatch of the colonies would mean that the tested product is a counterfeit. The result is then communicated to the **interprofession Tête-de-Moine**, who is responsible for feeding the information into the system, which publishes it on-chain and makes it accessible to the public. Now the test result is visible next to the production history of a given lot. An access control mechanism (ACM) is implemented with the primary features discussed above to give only authorized entities the right to make changes *e.g.*, record new steps or add laboratory results. The

administrator is in charge of registering those roles, overseeing the overall system, and has the authority to make changes if needed.

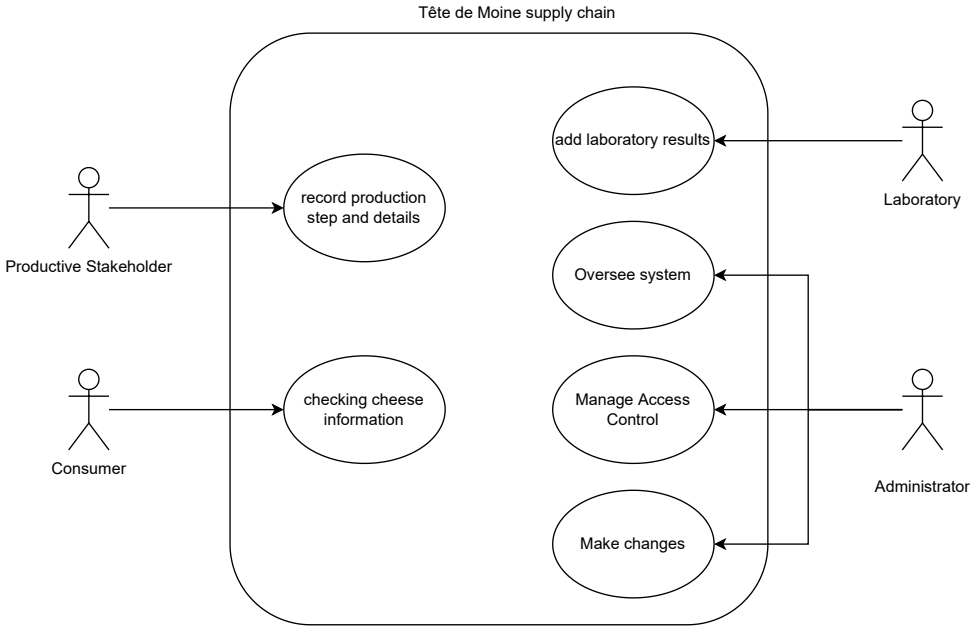


Figure 4.1: Specific Use Case

4.2 Design

In the following section, an overview of the system is provided. In contrast to Section 4.3, which focuses on the implementation, this chapter is technology-agnostic, showing the components and their respective responsibilities.

The system's design followed a top-down approach, first focusing on building a general purpose SC framework for SCT on the BC. That framework was then narrowed down in the implementation to fit the specific use-case of the CheeseChain project described in Section 4.1. Figure 4.2 shows a visualization of the system's architecture.

As visually separated, the system consists of three main parts: **external, off-chain, and on-chain**. On-chain and off-chain aim to serve end consumers while the external part forms the *private system* and mainly serves internal stakeholders of the supply chain. From now on, the public system will be referred to as the *Public CheeseChain Solution (PUC)*.

4.2.1 External

The external part of the system forms the private system, which includes a private consortium BC and the Fromartre DB. It is an integral part of the overall system but is not developed by the author of this thesis. The PUC depends on the external part and

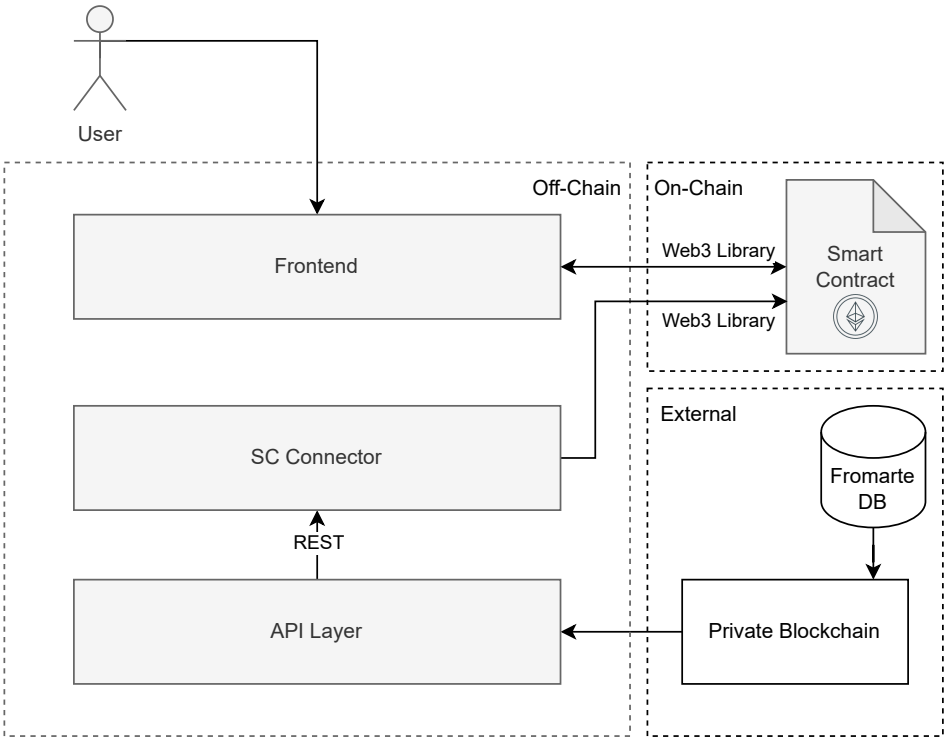


Figure 4.2: Proposed SCT System Architecture

extends its feature set to serve the public. The previously mentioned Fromartre quality management system stores the information in the DB referred to as *Fromartre DB* in Figure 4.2.

The data is pulled via its GraphQL Application Programming Interface (API) after freezing a form and fed into the private BC. The data available in the private system is an immutable copy of the data stored in the Fromartre DB and a superset of the data available in PUC; therefore, it is not accessible to the public in its entirety. All information in the private BC is intended to enhance transparency and trust along the Tête-de-Moine value chain. The selected subset of data, the public data, is injected into PUC through API endpoints exposed by its API layer.

4.2.2 On-Chain

All PUC elements are built to enhance the functionality of the on-chain part, which is the core of the system and *consists only of one SC*. The SC is the core of this thesis and presents an extendable and adaptable SC implementation that can be used in a broad range of SCT use cases. The SC offers the following features and operates under the following assumptions and limitations: The SC implements and exposes publicly callable methods whose execution is restricted by an ACM.

Assumptions and Limitations

All features of the SC are underpinned via the following assumptions and limitations:

- The production of a product has a determined starting point.
- Every step is performed only by a single entity.
- All production steps are performed in sequence and only one at a time.
- The product itself or its corresponding production batch, *e.g.*, a lot of cheese, must be uniquely identifiable by a number.

Features

Under the assumptions and limitations listed above, the SC allows stakeholders to register a new production entity and append any number of sequential production steps to it, and retrieve each step and entity independently. Additionally, information can be registered on a production entity. In the context of the CheeseChain, this translates into the following supported actions:

- **Add Milk Batch:** A milk producer registers a new batch of milk and informs the cheese producer about the milk batch identifiers at the time of purchase.
- **Add Lot:** By providing a list of milk batches being used in the production, a designated lot of cheese can be registered. This generates and stores a unique identifier for the new lot and allows appending production steps.
- **Get Lot:** Returns information about a lot, identified by its id, and the option to trace back its entire supply chain history.
- **Add Step:** By providing the lot identifier, a new step can be appended to the referenced lot. The entity performing the step can implicitly be derived and identified through the BC address that added the step. For every step, a unique id is generated.
- **Get Step:** Returns information about a step, identified by its id.
- **Add Laboratory Results:** Once Agroscope has completed the laboratory tests, the results can be stored in the SC by specifying the outcome (true: bacterial colonies match, false: bacterial colonies do not match) and the corresponding lot identifier.

The SC design's central and most important feature lies in the ability to trace back the entire history of sequential append-only steps and generate a unique history for each lot. Figure 4.3 depicts how this is achieved. Every lot has three properties:

- **Time Stamp:** The time it was registered, respectively added to the SC;

- **Last Step:** The ID of the last step added to the lot; and
- **Test Result:** Additional information on the lot. In this case, the test results of Agroscope's laboratory test.

Tracing back a given lot's production history starts with identifying the latest added step, which is stored inside the *last step* property of the lot struct. Each step is uniquely identified by an ID and can be retrieved from a mapping (a dictionary-like data structure) that maps the ids to the corresponding struct. Every step consists of five properties, out of which the `previousStep` points to its predecessor. The entire history of a lot is obtained by identifying the last step added to the lot and then tracing back all steps until a step has a `previousStep` of 0. In this case, the first step added to the lot is found, and the entire history is retrieved. All properties present on the `Step` struct are explained below:

- **Time Stamp:** The time it was registered, respectively added to the SC
- **Previous Step:** The ID of the step proceeding the current step
- **Owner:** The BC address of the entity or person that registered (and performed) the step
- **Description:** A short description of what actions were included in the step
- **Coordinates:** Identifies the geographical location where the step was performed

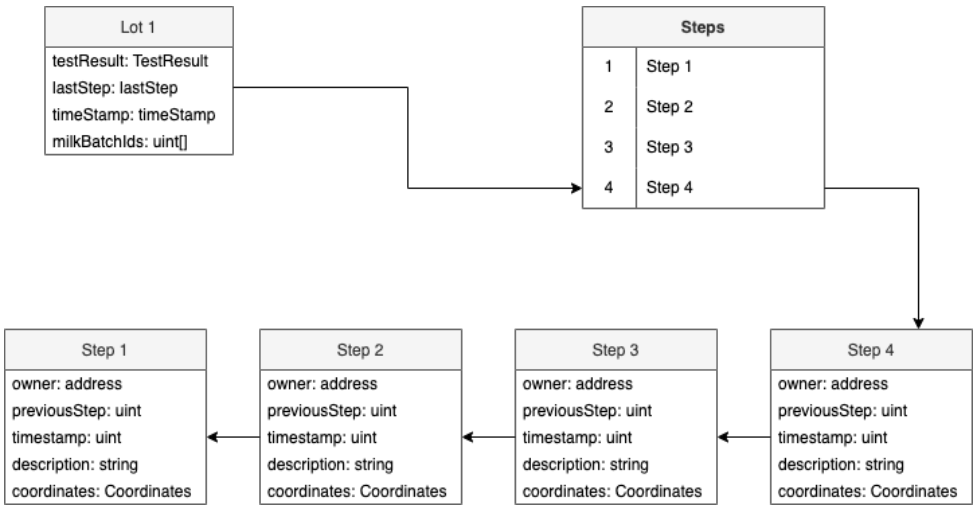


Figure 4.3: Tracing Back Production Steps in a SC

Access Control Mechanism

Since the SC is part of a public system on a public permissionless BC with no encryption for the stored data, all information present in the SC is accessible to the public and can be read by any interested party. On the other hand, state-changing write actions are controlled by an ACM, which defines five different scopes, referred to as *roles*.

- **View Only:** This is an implicit role not defined in the SC that every user of the PUC enjoys that is not registered as a system participant by the administrator. This role grants all read rights but no write rights on the SC.

- **Milk Producer:** A milk producer has one single task in the supply chain: to produce and register the produced milk. Hence, this role is only allowed to register new batches of milk.

- **Basic:** Any entity involved in the production other than the production of milk enjoys write rights for adding a lot and a step.

- **Laboratory:** Similar to the milk producer, the laboratory also has one single responsibility in the supply chain. That is to conduct a polymerase chain reaction (PCR) test and communicate the results publicly on-chain. Therefore, this role only has the right to add test results for specific lots to the SC.

- **Admin:** The entity or person deploying the SC is implicitly awarded the *administrator* role, which grants her the right to perform all read and write actions to the SC and, as such, oversees the system.

Which role is assigned to a user is determined by the administrator only. In addition to the features presented above, the SC implements functions to register a user, remove a user and change a user's role, which are only executable by the system administrator.

- **Add Participant:** Registers a user's BC address with the assigned role and name.

- **Change Participant Role:** Allows a participant's role to be changed.

- **Remove Participant:** Removes a participant, leaving him with read-only rights.

Table 4.1 gives an overview of all write actions that can be performed on the SC and sets them against the access rights defined for each role.

Table 4.1: ACM Roles and Their Functions

Function	**View Only**	**Milk Producer**	**Basic**	**Laboratory**	**Administrator**
Add Milk Batch	✗	✓	✗	✗	✓
Add Lot	✗	✗	✓	✗	✓
Add Step	✗	✗	✓	✗	✓
Add Participant	✗	✗	✗	✗	✓
Remove Participant	✗	✗	✗	✗	✓
Change Role	✗	✗	✗	✗	✓
Add Lab Results	✗	✗	✗	✓	✓

4.2.3 Off-Chain

The off-chain part relies on the on-chain as well as on the external part. It connects the different parts of the system and enhances the User Xperience (UX). The *Frontend, Smart Contract Connector (SCC)* and *API Layer* components together form the off-chain part of the system; the SCC and the API layer respectively work together as the server. Together they abstract the SC and isolate it from the private system.

API Layer

In order to route requests to the public system, handle data payloads and forward them to the correct location, an API layer is put into place. It is the mediator between the private BC and the SCC, commonly referred to as the *controller* in the software architectural pattern Model-View-Controller (MVC). In order to fulfill this task, the component exposes and implements a set of endpoints to perform all read and write actions available on the SC.

Smart Contract Connector

The SCC bridges the gap between off-chain and on-chain components by processing the received data from the API layer to the proper data structure that the SC accepts. Further, state-changing (write) transactions are signed by leveraging a BC library and sent to a remote BC node via Hypertext Transfer Protocol (HTTP).

Frontend

The frontend offers a convenient and user-friendly way to interact directly with the SC and, therefore with the system core. It is the entry point for consumers or supply chain entities that want to make manual changes. It consists of the following three components:

- **User Interface:** The User Interface (UI) provides a visual representation of all write actions that can be performed on the SC and the functionality to retrieve a lot's complete history and PCR test results. They change based on the user's role to show only the allowed actions.
- **BC Wallet:** Having a BC wallet connected to a frontend is necessary to perform actions that write to the BC. Read actions can be done without a wallet connected. Further, the BC address of the connected wallet eliminates the need for a conventional log-in since the authentication is done by proving the possession of the private key.
- **Smart Contract Connector:** It is a more lightweight version of the SCC component inside the server. The main difference is that it allows for direct communication with the SC instead of calling a different component to perform that task.

4.3 Implementation

This section highlights the key aspects of the system's implementation. Thus, the technologies used to build the components introduced in Section 4.3 are presented. With the help of code snippets, it is shown how their interplay is achieved from a technical standpoint.

4.3.1 Smart Contract

The SC is written in Ethereum's native programming language, Solidity [3], and is intended to be deployed to a Testnet for testing and on the Mainnet in production. We use Hardhat, a development environment that facilitates compiling, deploying, testing and debugging Ethereum software [37], for local development. Hardhat comes with *Hardhat Network*, a local Ethereum Network, similar to Ganache or `geth -dev`.

In order to make use of the SC functionality, it must first be deployed, which grants the Ethereum address of the deployer the administrator role. Administrator rights are assigned implicitly by setting the public `administrator` variable to the `msg.sender` as presented on line 4 in Listing 4.1. Therefore, there can only exist one administrator per SC.

```
1 address public administrator;
2
3 constructor() {
4         administrator = msg.sender;
5     }
```

Listing 4.1: Constructor Function

Thereon, a set of read and write functions are exposed for interaction. These will be discussed in more detail in the following sections. All state changes are stored in the SC itself. This means that the SC is the PUC's single source of truth.

Towards understanding the core functions of the SC *e.g.*, `addParticipant`, `addLot` or `addStep`, one must first understand the variables, structs, enums, mappings and function modifiers.

Public variables

In addition to the `administrator`, the SC defines four more publicly readable variables:

```
1 uint public totalSteps; //starts at 0
2 uint public totalLots; //starts at 0
3 uint public totalMilkBatches; //starts at 0
4 Participant public laboratory; // only one laboratory can exist
```

Listing 4.2: Public Variables

`TotalSteps`, `totalLots` and `totalMilkBatches`, all work as incremental counters starting at 0. The values are unique identifiers for lots, steps and batches of milk. At the same time, they provide an insight into how many entities were recorded by the SC.

Structs

In order to define the use case of the CheeseChain more accurately, a set of custom data types, called structs, are defined. For instance, every participant is registered as a `Participant` struct, which includes a name, a role and a corresponding BC address. Listing 4.3 introduces all structs used inside the SC.

```
1  struct Participant {
2      string name;
3      Role role;
4      address owner;
5  }
6
7  struct Coordinates {
8      string latitude;
9      string longitude;
10 }
11
12 struct MilkBatch {
13     uint timestamp;
14     address producer;
15     Coordinates coordinates;
16 }
17
18 struct TestResult {
19     bool result;
20     uint timestamp;
21 }
22
23 struct Lot {
24     TestResult testResult;
25     uint lastStep;
26     uint timestamp;
27     uint[] milkBatchId;
28 }
29
30 struct Step {
31     address owner;
32     uint previousStep;
33     uint timestamp;
34     string description;
35     Coordinates coordinates;
36 }
```

Listing 4.3: Structs

Enums

Listing 4.4 shows the definition of the `Role` enum, which is used to define a participant's role inside the `Participant` struct. Using enums over any other data structure for roles *e.g.*, integers or strings, allows for compile-time checking and prevents errors from passing invalid values during development. Internally the roles are represented as integers from 0 to 3.

```
1 enum Role {
2     ViewOnly,
3     Basic,
4     Laboratory,
5     MilkProducer
6 }
```

Listing 4.4: Enums

Mappings

Mappings act as the main data storage element inside the PUC SC. In this case, they always take a Solidity primitive data type (uint or address) and point to a custom data type defined by a struct. This structure is ideal for storing an unknown amount of data that can and should be uniquely identifiable through a single key. This is the case with `MilkBatches`, `Lots`, `Steps` and `Participants`, which are uniquely identifiable via an unsigned integer (uint), respectively by an Ethereum address. Listing 4.5 defines all mappings used in the SC.

```
1 mapping(uint => Step) public steps;
2 mapping(uint => Lot) public lots;
3 mapping(uint => MilkBatch) public milkBatches;
4 mapping(address => Participant) public participants;
```

Listing 4.5: Mappings

Function Modifiers

In Solidity, function modifiers allow appending functionality to a function in a declarative manner. They are beneficial to *reduce code redundancy* since they can be reused for multiple functions once defined. Their main use case is to check a condition prior to executing a function [29, 25]. This is useful when implementing an ACM that checks if the caller is allowed to call a specific function. If this is not the case, the function call is reverted with the appropriate error message defined in the function modifier. Should a participant with any role different from administrator or basic call the `addLot` function in Listing 4.12, the transaction would revert with the error message defined in the `onlyBasicParticipant` function modifier in Line 5 of Listing 4.6. As shown in Listing 4.6, a function modifier is implemented for every role defined in the SC.

```
modifier onlyAdministrator {
    require(msg.sender == administrator, 'This function is only callable
        by an admin!');
    _;
}
modifier onlyBasicParticipant {
    require(isAdministrator(msg.sender) || isBasicParticipant(msg.sender
       ), 'Msg.sender is not basic or admin');
    _;
}
modifier onlyLaboratory {
    require(isAdministrator(msg.sender) || isLaboratory(msg.sender), '
       Msg.sender is not lab or admin');
    _;
}

modifier onlyMilkProducer(){
    require(isAdministrator(msg.sender) || isMilkProducer(msg.sender), '
       This function is only callable by a milk producer!');
    _;
}
```

Listing 4.6: ACM Function Modifiers

Listing 4.6 shows the implementation of all function modifiers belonging to the ACM. In addition, five more function modifiers are defined in Listing 4.7 and will be addressed within their context of use further below.

```
modifier notEmptyAddress(address _address) {
    require(_address != address(0), 'The address cannot be a 0 address!'
       );
    _;
}

modifier participantDoesntExist(address _address){
    require(participants[_address].owner == address(0), "A participant
       with this address exists already");
    _;
}

modifier participantExists(address _address){
    require(participants[_address].owner != address(0), "A participant
       with this address does not exist");
    _;
}

modifier milkBatchExists(uint[] calldata _batchIds){
    require(_batchIds.length > 0, "Please provide at least one milk
       batch identifier!");
    for (uint i=0; i < _batchIds.length; i++){
        require(milkBatches[_batchIds[i]].timestamp != 0, "Please
           provide only existing milk batch identifiers");
    }
    _;
}

modifier lotExists(uint _lotId){
```

```
25      require(lots[_lotId].timestamp != 0, 'The lot with the given number
            does not exist!');
26      _;
27  }
```

Listing 4.7: Additional Function Modifiers

Add Participant

Upon deployment of the SC, the *administrator* is the only entity having write permissions. In order to support a real SCT use case with different entities responsible for different steps in production, these must first be registered to the SC.

```
1  event ParticipantAdded(Participant participant);
2
3  function addParticipant(Participant calldata participant)
4      onlyAdministrator
5      notEmptyAddress(participant.owner)
6      participantDoesntExist(participant.owner)
7      public {
8          participants[participant.owner] = participant;
9          emit ParticipantAdded(participant);
10     }
```

Listing 4.8: Add Participant

A participant is added by calling the `addParticipant` function in line 3 in Listing 4.8 with a `Participant` struct, which is defined by a `name`, a `role` and an `address`. The name, different from the Ethereum address, serves as a human-readable identifier for a particular participant and can be an arbitrary string. The role on the other side must be of the `Role` enum type. A set of requirements set by the modifiers must be satisfied for the function to execute successfully. The function can only be called by the administrator, the address provided for the participant can not be an empty address, respectively a `0x0`, address, and no existing participant is allowed to have the same address.

Remove Participant

Removing a participant by calling `removeParticipant` in Listing 4.9 only requires the address, which must be present in the `participants` mapping. The function removes the identified entry from the mapping and is limited to only being callable by the administrator.

```
1  function removeParticipant(address _address) onlyAdministrator
       participantExists(_address) public {
2      delete participants[_address];
3  }
```

Listing 4.9: Remove Participant

Change Participant Role

Should the need ever appear to change a participant's role, this can be achieved by calling the `changeParticipant` function in line 3 of Listing 4.10. The function takes the user address and the new role as parameters. If the address does not match any registered participants, the function reverts.

```
1 event RoleChanged(Participant participant);
2
3 function changeParticipantRole(address _address, Role _newRole)
    onlyAdministrator participantExists(_address) public {
4     participants[_address].role = _newRole;
5     emit RoleChanged(participants[_address]);
6 }
```

Listing 4.10: Change Participant Role

Adding Milk Batch

As milk is produced before the cheese, it is not clear in which lot of cheese the milk will be used. For this reason, milk production is a special step within PUC. To reflect this property, the `addMilkBatch` function in line 3 of Listing 4.11 allows the registration of a batch of produced milk before the initialization of a lot of cheese. Only the administrator or a milk producer can add a milk batch to the SC.

```
1 event NewMilkBatch(uint indexed _milkBatchId, uint _timestamp);
2
3 function addMilkBatch(Coordinates coordinates) onlyMilkProducer external
    {
4     totalBatches += 1;
5     milkBatches[totalBatches] = MilkBatch(block.timestamp, msg.sender,
        coordinates);
6     emit NewMilkBatch(totalBatches, block.timestamp);
7 }
```

Listing 4.11: Adding a Milk Batch

Adding Lot

Registering a lot of cheese requires knowledge about the origin of the milk used to produce the lot. Only basic participants and the administrator have the authority to perform this action.

```
1 event LotAdded(uint indexed _lotId, uint _timestamp);
2
3 function addLot(uint[] milkBatchIds) onlyBasicParticipant external {
4     totalLots += 1;
5     TestResult memory test = TestResult(false, 0);
6     lots[totalLots] = Lot(test, 0, block.timestamp, milkBatchIds);
7     emit LotAdded(totalLots, block.timestamp);
8 }
```

Listing 4.12: Adding a Lot

Listing 4.12 shows how the `totalLots` variable is used as a unique identifier for a newly created lot. Additionally, an empty `TestResult` and the `timestamp` of the current block are added to the lot. Upon successful creation, the `lotAdded` event is emitted, which gives any entity interested in knowing when the function was executed the option to watch the SC by subscribing to the given event.

Adding Step

The `addStep` function on line 3 of Listing 4.13 is called with the `lotNumber` (specifying the lot to append the step to), a description, as well as coordinates where the step took place. The `lotExists` function modifier makes sure the lot exists, while `onlyBasicParticipant` checks if the caller is authorized to perform this action. `TotalSteps` follows the same mechanic as `totalLots` and is used as the step identifier. Line 6 in Listing 4.13 saves a new step to the `steps` mapping. Within this newly constructed step, `previousStep` is set to the `lastStep` recorded in the lot referenced. Lastly, `lastStep` of the lot is updated to the step identifier of the current step, which constructs a chain of steps, and the `StepAdded` event is emitted.

```
1 event StepAdded(uint indexed _stepId, uint _timestamp);
2
3 function addStep(uint lotNumber, string calldata description,
     Coordinates calldata coordinates)
4     onlyBasicParticipant lotExists(lotNumber) external {
5     totalSteps += 1;
6     steps[totalSteps] = Step(msg.sender, lots[lotNumber].lastStep, block
          .timestamp, description, coordinates);
7     lots[lotNumber].lastStep = totalSteps;
8     emit StepAdded(totalSteps, block.timestamp);
9 }
```

Listing 4.13: Adding a Step

Adding Laboratory Result

When a laboratory test result becomes available, it can easily be added by calling the `addLabResult` function, which takes the `lotNumber` and a boolean: whether the bacterial colonies were a match or not. This action can only be performed by the one participant registered as the laboratory and emits the `LabResultAdded` event.

```
1 event LabResultAdded(uint indexed _lotId, bool indexed _result, uint
     indexed _timestamp);
2
3 function addLabResult(uint lotNumber, bool result) onlyLaboratory public
      {
4     lots[lotNumber].testResult = TestResult(result, block.timestamp);
5     emit LabResultAdded(lotNumber, result, block.timestamp);
6 }
```

Listing 4.14: Adding a Laboratory Result

Tests

Writing automated tests when building SC is of the utmost importance since SCs are immutable once deployed and cannot be easily updated. Deploying the SC costs money, and often user's money or information is at stake. SC upgradeability can nevertheless be realized with the *proxy pattern*, which uses a proxy SC to forward function calls to the SC containing the corresponding implementation [53]. The proxy does not change, while the pointer to the implementation SC changes with every new version of the implementation SC being deployed.

Tests for every functionality in the SC were written to ensure correctness. Typescript is the programming language of choice, while *Ethers.js* is the library used to interact with the SC. As a test runner *Mocha* is used in combination with the *Chai* assertion library, which runs on *Node.js* [12]. Listing 4.15 shows the required imports to set up the testing environment.

```
1 const { ethers } = require("hardhat");
2 const { expect } = require("chai");
```

Listing 4.15: Importing Required Packages

The tests are contained in a file called `CheeseChain.test.js` inside the `test` folder in the root directory of the project. Listing 4.16 shows what the interface of the written test suite looks like:

```
1  describe('CheeseChain', function() {
2    // add a test hook
3    beforeEach(function() {
4      // ...some logic before each test is run
5    })
6    // test a functionality
7    it( "Should revert addLot() when not called by admin or basic",
8      async function() {
9
10     // add an assertion
11      await expect(cheeseChain.connect(lab).addLot())
12         .to.be.revertedWith(
13       "Msg.sender is not basic or admin"
14     );
15   })
16   // ...some more tests
17 })
```

Listing 4.16: Test Suite Interface

Before each test is run, some Ethereum addresses are retrieved from the local BC and stored inside variables. Then a new contract is deployed, and one participant for each role is registered since the tests need to handle calling the functions from all roles.

Running `npx hardhat test` in the terminal spins up a local Ethereum network and executes all the files in the `test` folder against the SC. As output, a list of all tests, including information on how long they took to run and if they passed.

```
// get accounts
[admin, lab, basic, viewOnly, dummy] = await ethers.getSigners();

// admin is automatically registered
const CheeseChain = await ethers.getContractFactory('CheeseChainV2',
    admin)
cheeseChain = (await CheeseChain.deploy()) as CheeseChainV2
await cheeseChain.deployed()

await cheeseChain.addParticipant({
  name: 'Example Laboratory',
  role: Role.Laboratory,
  owner: lab.address,
})

await cheeseChain.addParticipant({
  name: 'Example Basic',
  role: Role.Basic,
  owner: basic.address,
})

await cheeseChain.addParticipant({
  name: 'Example ViewOnly',
  role: Role.ViewOnly,
  owner: viewOnly.address,
})

await cheeseChain.addParticipant({
  name: 'Example Milk Producer',
  role: Role.MilkProducer,
  owner: milk.address,
})s
```

Listing 4.17: Execution Before Each Test

4.3.2 Server

The server consists of the *API layer* and the *smart contract connector*. It acts as the interface between the private and the public system, which is its only responsibility. It is implemented in *Node.js* using *TypeScript* and leveraging *Express.js* as the backend framework. Figure 4.4 depicts the server's directory structure. The files and their associated code will be explained in the next sections.

API Layer

The API Layer is a lightweight controller that routes incoming HTTP requests to the corresponding middleware functions implemented by the SCC. All actions that can be taken on the SC are exposed as endpoints, served by Express.js and live within the `server.ts` file.

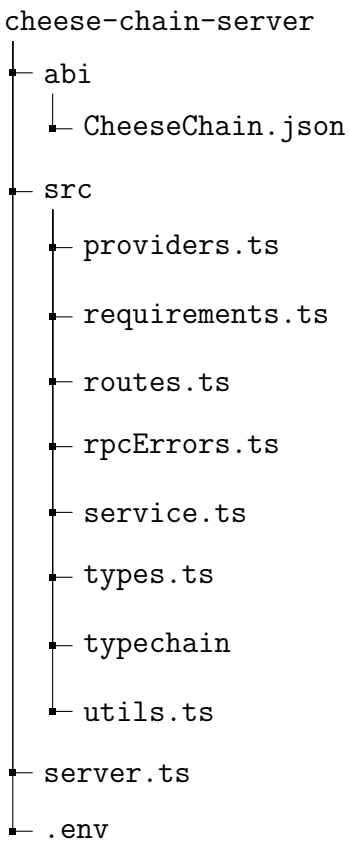


Figure 4.4: Server Directory Structure

```
1 const app: Express = express();
2 const PORT = process.env.PORT || 3000;
3
4 app.listen(PORT, () => {
5   console.log(`Server is listening on port ${PORT}`);
6 });
7
8 app.get("/contract", requireContract, getContractAddressRoute);
```

Listing 4.18: Getting the Contract Address

Listing 4.18 shows a simple example of an endpoint implemented with *Express.js*. The method called on `app` specifies the REST method for the endpoint, which is provided as the first parameter. All later parameters are `middleware` functions that are called in sequence, from left to right. The middleware functions are part of the SCC; therefore, they are described in the next section.

```
1 const connectContract = [requireContract, attachContract]
2
3 // first steps
4 app.post("/deploy", deployContractRoute);
5
6 // general functions
7 app.get("/block", getBlockRoute);
8 app.get("/contract", requireContract, getContractAddressRoute);
9
```

```
// contract variables
app.get("/total-lots", connectContract, getTotalLotsRoute);
app.get("/total-steps", connectContract, getTotalStepsRoute);
app.get('/total-milk-batches', connectContract, getTotalMilkBatchesRoute
    )
app.get("/admin", connectContract, getAdminRoute);

// mapping information
app.get("/lots/:id", connectContract, getLotRoute);
app.get("/steps/:id", connectContract, getStepRoute);
app.get('/milk-batches/:id', connectContract, getMilkBatchRoute)
app.get("/participants/:address", connectContract, getParticipantRoute);

//core functions
app.post('/add-milk-batch', requireContract, attachContract,
    addMilkBatchRoute)
app.post("/add-lot", connectContract, addLotRoute);
app.post("/add-step/:id", connectContract, addStepRoute);
app.post("/add-basic", connectContract, addBasicRoute);
app.post("/add-lab", connectContract, addLabRoute);
app.post("/remove-participant", connectContract, removeParticipantRoute)
    ;
app.put("/change-role", connectContract, changeRoleRoute);
app.post("/add-lab-result", connectContract, addLabResultRoute);
```

Listing 4.19: All REST Endpoints

Listing 4.19 shows a complete list of all endpoints implemented by the API layer, together with the middleware functions the requests are routed to.

Smart Contract Connector

The SCC is the server's largest and most important component, consisting of the entire `src` directory in Figure 4.4. It allows the incoming requests to be transformed into transactions, signs them, and forwards them to the BC. The two main Express middleware functions are `requireContract` and `attachContract` which are concatenated into the `connectContract` array, since they are always executed together and in the same order.

```
export const requireContract = (
  req: Request,
  res: Response,
  next: NextFunction
) => {
  try {
    const contractAddress = process.env.CONTRACT_ADDRESS;
    if (contractAddress === undefined) {
      throw new Error("No contract address defined");
    }
  } catch (error) {
    let message;
    if (error instanceof Error) message = error.message;
    else message = String(error);
    res.status(428).json({
      error: message,
```

```
17      });
18    }
19    req.contractAddress = process.env.CONTRACT_ADDRESS;
20    next();
21  };
```

Listing 4.20: RequireContract Middleware

Listing 4.20 shows how `requireContract` ensures that a contract address is present in the execution environment before executing middleware that requires the presence of a SC. The next middleware executed is, should a SC address indeed be present, typically `attachContract`. Otherwise, a 428 response *precondition required* is returned together with an appropriate error message.

```
1  import { Contract } from "ethers";
2  import { connect } from './providers'
3  import { CheeseChain } from './typechain'
4
5  const [provider, wallet, contractAt] = connect()
6
7  export const attachContract = async (
8    req: Request,
9    res: Response,
10   next: NextFunction,
11 ) => {
12   const CheeseChain: Contract = contractAt(
13     'CheeseChain',
14     process.env.CONTRACT_ADDRESS,
15   )
16   const cc: CheeseChain = (await CheeseChain.connect(wallet)) as
       CheeseChain
17   req.cc = cc
18   next()
19 }
```

Listing 4.21: AttachContract Middleware

`AttachContract` creates an Ethers.js contract instance and connects a wallet to it, which is necessary to sign transactions that write to the contract but also facilitates reading the contract. The Ethers SC instance, which is now connected to a deployed SC instance, is attached to the Express request object within line 16 in Listing 4.21 to be accessible to the following middleware functions. `CheeseChain`, which is imported in line 2 of Listing 4.21, is generated by a Hardhat plugin that generates TypeScript types from SCs and allows static type checking when writing code.

Listing 4.22 shows how `providers.ts` exports the `connect` function, which returns an array of constants that abstract some steps necessary to connect to a deployed SC instance.

```
1  import { ethers, providers, utils, Wallet } from 'ethers'
2  import { getContract } from './utils'
3
4  const connect = (): [providers.Provider, Wallet, any] => {
5      const provider = process.env.CHAIN_ID
6      ? new providers.JsonRpcProvider(process.env.NODE_URL, process.env.
         CHAIN_ID)
```

```
7       : new providers.JsonRpcProvider(process.env.NODE_URL)
8
9       // use private key for wallet
10      const wallet = new ethers.Wallet(process.env.PRIVATE_KEY!, provider)
11
12      const contractAt = getContract(wallet)
13
14      return [provider, wallet, contractAt]
15  }
16
17  export { connect }
```

Listing 4.22: Providers.ts

In order to connect to a network, one must connect to a node of the corresponding network. This is achieved by setting the `NODE_URL` environment variable to the node's Unique Resource Identifier (URI), which Ethers needs to instantiate a provider. This abstraction enables talking to the BC. `CHAIN_ID` is an optional variable, which defaults to the chain id returned by the connected node. Line 5 in Listing 4.23 shows how `getContract`, a curried function inside `utils.ts`, finds the SC application binary interface (ABI) inside the `ABI` directory and connects a wallet, as well as a contract address. This constructs an object with methods on it for each SC function.

```
1  import { Wallet, ethers, utils, Contract } from 'ethers'
2
3  export const getContractJSON = (contractName: string): any => require
       (`../abi/${contractName}.json`)
4
5  export const getContract = (wallet: Wallet) => (contractName: string,
       contractAddress: string): ethers.Contract => {
6      const contractJson = getContractJSON(contractName)
7      return new ethers.Contract(contractAddress, contractJson.abi, wallet
           )
8  }
```

Listing 4.23: Utility Functions

Once a contract instance is successfully added to the Express request object, the next middleware function can interact with the SC. As visible in Listing 4.19 all of the last functions end with 'Route', which signifies the route the request is routed to. All available `routes` are implemented in `routes.ts` as shown on the example of the `addLotRoute` in Listing 4.24.

```
1  import { Request, Response } from "express";
2  import { addLot } from "./service";
3
4  const createRoute = (call: Function) => async (req: Request, res:
       Response) => {
5    try {
6      const value = await call(req, res);
7      res.send(JSON.stringify(value));
8    } catch (error) {
9      console.log(error);
10   }
11 };
```

```
12
13 export const addLotRoute = createRoute(addLot);
```

Listing 4.24: Create a Route

A route is created by passing in the function that interacts with the contract into the `createRoute` function, which is responsible for calling the passed function and parsing and returning the response in JavaScript Object Notation (JSON) representation. For every route, a corresponding service function is implemented inside `service.ts` which interacts with the SC via the instance attached to the Express request object. Line 5 in Listing 4.25 checks if the function parameters passed are appropriate and returns a 400 status code otherwise.

```
1  export const addLot = async (
2    req: Request<{}, {}, AddLotBody>,
3    res: Response,
4  ) => {
5    if (!req.body.milkBatchIds || typeof req.body.milkBatchIds !== 'object
       ') {
6      res.status(400).json({
7        error: 'Invalid body! Please Provide milk batches as an array of
           string.',
8      })
9      return
10   }
11   const { milkBatchIds } = req.body
12   try {
13     const tx = await req.cc!.addLot(milkBatchIds)
14     const receipt = await tx.wait()
15     return { lotId: receipt.events![0].args!._lotId.toNumber() }
16   } catch (error) {
17     handleRpcErrors(
18       error,
19       [rpcErrors.onlyBasic, rpcErrors.onlyValidMilkBatch],
20       res,
21     )
22   }
23 }
```

Listing 4.25: Add Lot Service

`HandleRpcErrors` in line 17 in Listing 4.25 handles custom errors thrown by the function modifiers, as well as other remote procedure calls (RPC) errors that might appear when calling a SC.

4.3.3 Frontend

The frontend is a single-page application built with *TypeScript* and *React*, a JavaScript UI library. The reasoning behind the choice of the technological stack is due to the author's prior experience. By design, the frontend is not a required component of the system for it to function. However, it enhances the user experience and allows the data stored inside the SC to be accessible and writable by users with minimal experience within the BC

ecosystem. As demonstrated in Section 4.3.2, all actions on the SC can be performed via the server or directly through the SC. The main use case for the frontend is a simple retrieval of the lot history by the end customer.

Connect or Deploy a Smart Contract

To interact with a SC, the SC must be identifiable by the frontend. This means that the address on the BC and a specification of the methods the SC exposes must be introduced into the codebase. The SC's ABI is supplied and encoded in JSON representation [24]. Essentially, the ABI specifies how contract calls must be encoded for the EVM and, reversely, how to decode transaction data [34]. Regarding the contract address, the frontend supports deploying a new contract or connecting to an existing one.

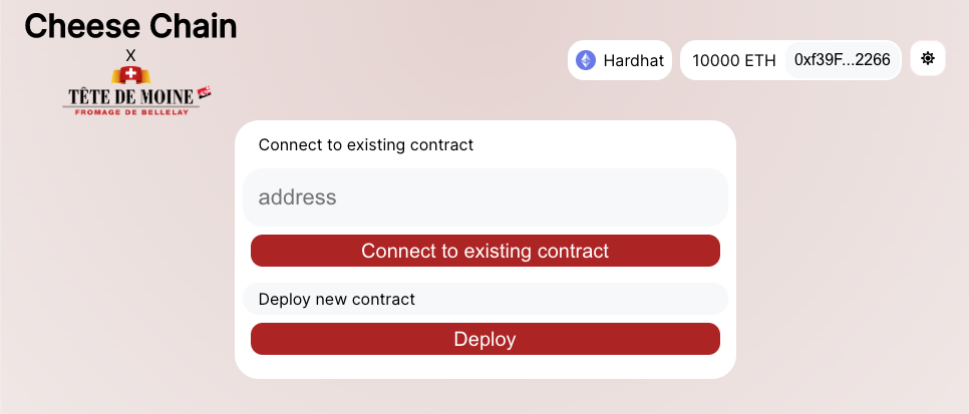


Figure 4.5: Connect or Deploy a Smart Contract

Connecting the frontend to an already deployed SC is done by adding the corresponding contract address as an environment variable within the `.env` file in the root directory of the frontend. As visible in the example in Listing 4.26 the environment variable must be named `REACT_APP_CONTRACT` in order to work.

```
1 REACT_APP_CONTRACT=0xe7f1725E7734CE288F8367e1Bb143E90bb3F0512
```

Listing 4.26: Add an Existing Contract Address as an Environment Variable

Wallet and Ethereum API

frontend that knows a SC's address and its ABI is not everything needed to make transactions or retrieve data in the browser. To retrieve data, an Ethereum *Provider* is needed, which is an abstraction of a connection to the Ethereum network, providing a standard interface to Ethereum node functionality [26]. Sending transactions further requires a *Signer*. A Signer signs transactions with the user's private key before broadcasting them to the Ethereum network [63]. In the current implementation *Alchemy* is used as the node provider, while *Metamask*[52] is used as the Signer. *Ethers.js* is the library used to connect the frontend to the Ethereum node provided by Alchemy or Metamask. Users with Metamask installed in their browser would connect to the node provided by Metamask, while users lacking the browser extension connect to the Alchemy node.

Smart Contract Connector

Similar to the server, the frontend also requires a SCC. Unlike the server implementation, where a connection to the SC is established through middleware functions, the frontend uses the React Context API and a service file implementing all interactions with the SC.

```
<ContractProvider>
    <FunctionsProvider>
        <ContractInteractions />
        <LotHistory />
    </FunctionsProvider>
</ContractProvider>
```

Listing 4.27: Frontend SCC Architecture

The architecture presented in Listing 4.27 consists of two React context providers (ContractProvider and FunctionsProvider), which allow passing a value to all child components without having to use *props*. The `ContractProvider` passes an Ethers contract instance connected to a deployed SC. In contrast, the `FunctionsProvider` makes it easy to call and await the methods exposed by the contract instance and react to responses or errors accordingly.

```
export const ContractContext = createContext<Contract | undefined>(
    undefined);
ContractContext.displayName = "ContractContext";

export const useContract = (): Contract => {
  const contract = useContext(ContractContext);
  if (!contract) {
    throw new Error("contract missing");
  }
  return contract;
};

export const ContractProvider = ({ children }: { children: ReactNode })
    => {
  const { library } = useWeb3React<Provider>();
  const [signer, setSigner] = useState<Signer>();
  const [contract, setContract] = useState<Contract | undefined>();

  useEffect((): void => {
    if (!library) {
      setSigner(undefined);
      return;
    }

    setSigner(library.getSigner());
  }, [library]);

  useEffect(() => {
    const envContract = process.env.REACT_APP_CONTRACT;

    const CheeseChain = new ethers.ContractFactory(
      CheeseChainArtifact.abi,
      CheeseChainArtifact.bytecode,
```

```
32        signer
33      );
34
35      if (!signer) return;
36      if (envContract === "" || envContract === undefined) return;
37
38      CheeseChain.attach(envContract).then((instance: Contract) =>
39        setContract(instance)
40      );
41    }, [setContract, signer]);
42
43    return contract === undefined ? (
44      <RegisterContract setContract={setContract} />
45    ) : (
46      <ContractContext.Provider value={contract}>
47        {children}
48      </ContractContext.Provider>
49    );
50  };
```

Listing 4.28: ContractProvider.tsx

Within the `ContractProvider`'s useEffect in line 26 of Listing 4.28, the SC address available in the environment variable is used to construct an Ethers contract instance together with the corresponding byte code and API of the already deployed SC. This instance is stored in the state variable `contract` in line 15 and is passed to the component tree to make it accessible to all its children. This instance allows all SC functions to be called using the corresponding methods, the same way as within the server's SCC. Said architecture assures the presence of a SC connected to the frontend and conditionally renders the `RegisterContract` or the `ContractContext.Provider` on line 43. Should no instance be present, the `RegisterContract` component shown in Figure 4.5 is rendered, prompting the user to either connect an existing SC or deploy a new one. If a contract address is defined in the environment variable and a signer is present, components allowing for SC interaction are rendered. The same state change is triggered once a new contract is deployed or an existing contract is registered. This causes a re-render and another call of the aforementioned `useEffect` hook. The ContractProvider's children gain access to the contract instance via the `useContract` hook defined in line 4.

```
1  export const ContractFunctionsContext = createContext<
2    ContractFunctions | undefined
3  >(undefined);
4  ContractContext.displayName = "ContractContext";
5
6  export const useFunctions = (): ContractFunctions => {
7    const functions = useContext(ContractFunctionsContext);
8    if (!functions) {
9      throw new Error("no functions");
10   }
11   return functions;
12 };
13
14 const FunctionsProvider = ({ children }: { children: ReactNode }) => {
15   const contract = useContract();
16   const [contractFunctions] = useState<ContractFunctions>({
```

```
    addLot: addLot(contract),
    addStep: addStep(contract),
    addParticipant: addParticipant(contract),
    removeParticipant: removeParticipant(contract),
    addResult: addResult(contract),
  });

  return (
    <ContractFunctionsContext.Provider value={contractFunctions}>
      {children}
    </ContractFunctionsContext.Provider>
  );
};
```

Listing 4.29: FunctionsProvider.tsx

The `FunctionsProvider` shown in Listing 4.29 is the second context provider inside the frontend's SCC and exposes all SC functions that are called within the frontend alongside the appropriate response and error handling code. These functions are implemented in the `interactionService.ts` file and are all curried functions that take a contract instance as the first parameter. Listing 4.30 shows an example callable by child components with `addLot(milkBatchId)`, after having accessed it through the custom `useFunctions` hook.

```
export const addLot = (contract: CheeseChain) => async (
  milkBatchId: number,
) => {
  try {
    const tx = await contract.addLot([milkBatchId])
    await tx.wait()
  } catch (e) {
    window.alert(serializeError(e).message)
  }
}
```

Listing 4.30: AddLot Service Function Inside InteractionService.ts

Interacting with the Smart Contract

Interacting with the SC is straightforward via the frontend. The upper of the two visually separated sections in Figure 4.6 represents all available write actions on the frontend. In contrast the lower section facilitates the read operation by retrieving the production history of a given lot. Only the actions the role is authorized to perform are shown depending on which role is assigned to the connected Ethereum account. In the case where no wallet is connected or no role is assigned to the address, the *view only* role applies implicitly and only the read lower read section is rendered. Sending a transaction and writing to the SC requires one more step: accepting the Metamask signature prompt that pops up upon requesting to send a transaction. Figure 4.7 shows an example of a lot's history that includes all steps and their associated information, as well as the laboratory test results performed by Agroscope.

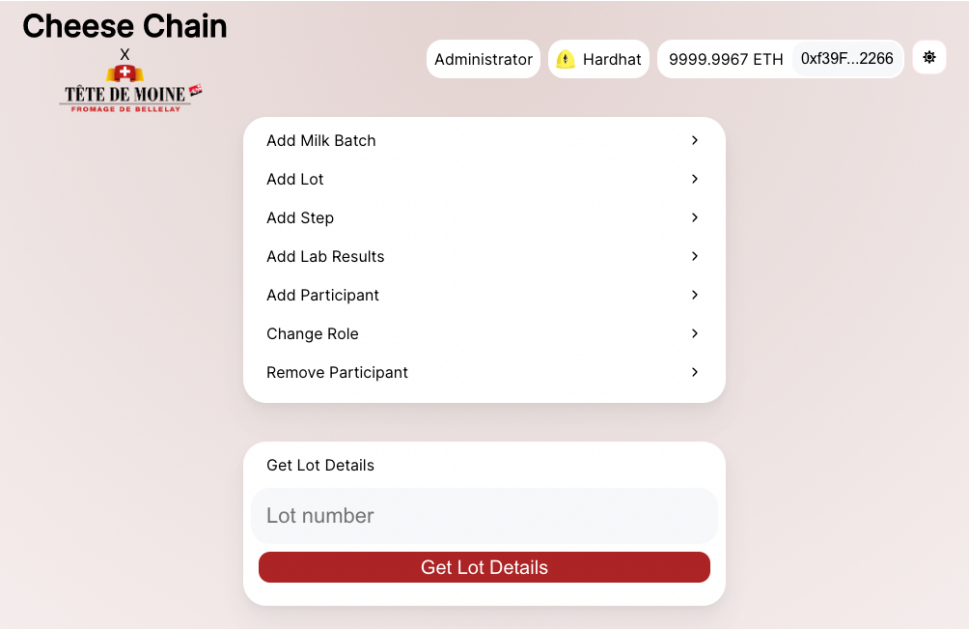


Figure 4.6: Interacting With the Smart Contract

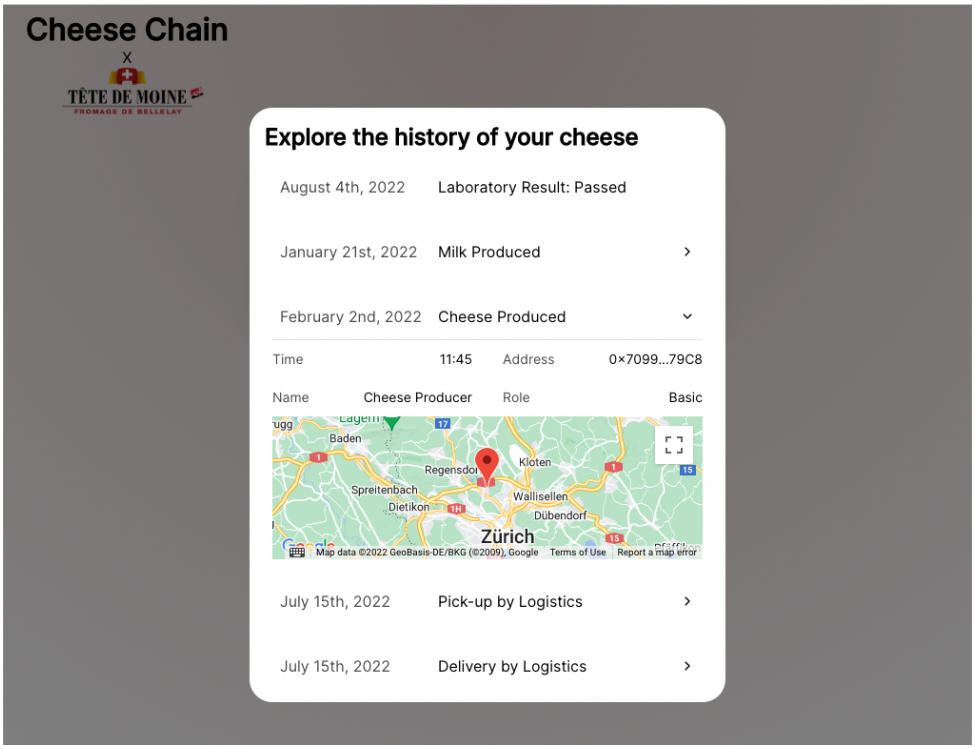


Figure 4.7: The Production History of a Cheese Lot

Chapter 5

Evaluation and Use Case

This chapter evaluates different aspects of the developed SCT system. First, a cost analysis is conducted in Section 5.1, followed by a performance analysis in Section 5.2 and a security analysis in Section 5.3. Finally, Section 5.5 concludes the chapter by discussing the system's usability in the previously evaluated areas. Both cost and performance analyses were conducted on the Ethereum Ropsten Testnet, the Polygon Mumbai Testnet, and on a local Hardhat network.

5.1 Cost Analysis

With the SC being the primary cost driver of operating the system, this section focuses on the cost dimension of the SC to estimate the cost of a sample use case within the CheeseChain project.

Each function implemented in the SC was called multiple times (n=10) from the frontend on the three networks. Afterward, the corresponding transactions were inspected on *Etherscan.io*, *Polygonscan.com* and the *local Hardhat logs* to determine the amount of gas that was used. The gas used for a specific function appeared not to be deterministic and sometimes varied by a small amount. Varying the input parameters for each function call suggests that the variance in the gas used does not only depend on the input parameters. To counteract this observation, the mean of all records for each function on each network was used to determine the values in Table 5.1. Exceptions to this rule are `Add Milk Batch`, `Add Lot` and `Add Step`, because adding the first milk batch or the first lot in the SC, as well as adding the first step on a specific lot uses significantly more gas. Therefore, these cases are shown separately from the rest and are not included in the mean. The same case applies to `Add Lab Result` but is ignored since the laboratory test result will only be registered once on every lot. Figure 5.1 visualizes this phenomenon of higher execution costs for the first function call for the three functions mentioned above and shows the cost relation between all functions on the SC.

$$Total\ fee = Gas\ units * Gas\ price\ per\ unit \tag{5.1}$$

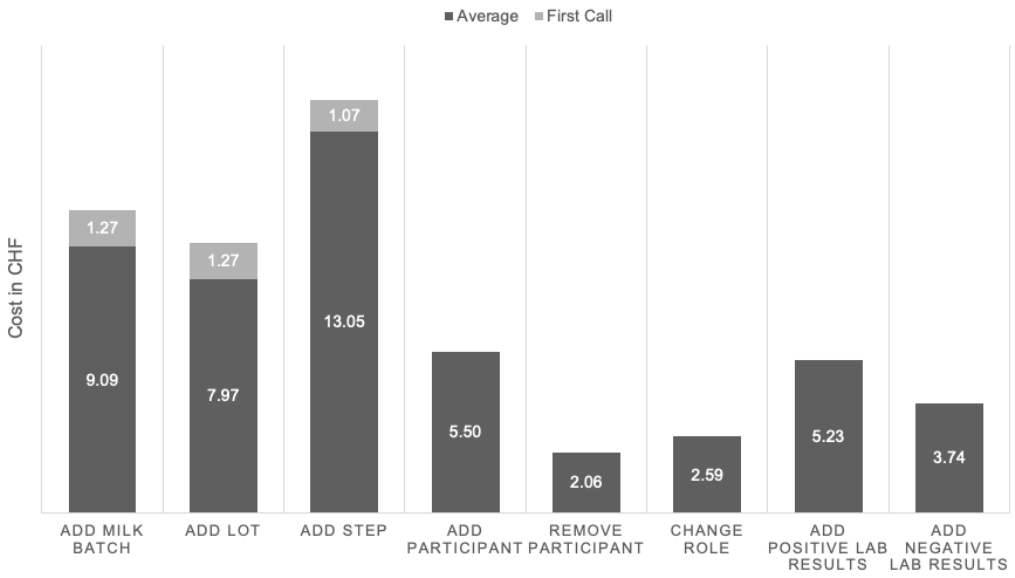


Figure 5.1: Cost Relation of SC Functions

Table 5.1 shows the result of the analysis and describes the cost of the function executions as well as SC deployment in terms of Gas (rounded to the closest integer), Ether (ETH), MATIC, Polygon's native token (rounded to 7 decimal figures, calculated according to Equation 5.1) as well as CHF (rounded to two, respectively four decimal figures). For the *gas price* the six month Ethereum (70 Gwei) [35] and Polygon (147 Gwei) [56] averages were used. The token prices used to represent the current price of ETH (1,064.49 CHF [13]) and MATIC (0.38 CHF [14]) at the time of writing (June 21, 2022).

Analyzing the cost of each function provides insight into how much executing an individual function costs but provides no information on how much a CheeseChain scenario would cost using the SC. For that reason, a representative scenario was constructed in Section 5.4 and evaluated by determining how much its operation would represent in costs on the Ethereum BC and on the Polygon side chain. The results in Table 5.1 show that a SC interaction on Polygon costs 0.075% of one performed on Ethereum. In other words, using the Polygon side chain is currently 1334 times cheaper than using the Ethereum BC for the same application.

5.2 Performance Analysis

When using an application in production, it is essential to know what performance characteristics to expect. For this reason, this section presents the results of a performance analysis regarding execution time.

Due to the decentralized nature of the system and the intention to be deployed on a public BC network, performance depends on one main factor, *BC performance*. Similar to the cost analysis, this analysis was conducted from within the Hardhat framework with the help of a custom script. The script records a timestamp immediately before sending

Table 5.1: Cost of SC Function Calls as of June 20, 2022

Function	**Gas Used**	**Gas Cost (ETH)**	**Gas Cost (MATIC)**	**Cost on Ethereum (CHF)**	**Cost on Polygon (CHF)**
Deploy Contract	1'750'851	0.1225596	0.2573751	130.46	0.0978
Add Milk Batch	121'970 139'070*	0.0085379 0.0097349*	0.0179296 0.0204433*	9.09 10.36*	0.0068 0.0078*
Add Lot	107'005 124'105*	0.0074904 0.0086874*	0.0157297 0.0182434*	7.97 9.25*	0.0060 0.0069*
Add Step	175'095 189'395*	0.0122567 0.0132577*	0.0257390 0.0278411*	13.05 14.11*	0.0098 0.0106*
Add Participant	73'846	0.0051692	0.0108554	5.50	0.0041
Remove Participant	27'611	0.0019328	0.0040588	2.06	0.0015
Change Role	34'696	0.00242872	0.0051003	2.59	0.0019
Add Positive Lab Results	70'162	0.00491134	0.0103138	5.23	0.0039
Add Negative Lab Results	50'250	0.0035175	0.0073868	3.74	0.0028

* First function call gas

a transaction to the Ethereum node provider or the local BC and records a timestamp as soon as the transaction was mined. Since SCs are executed during block validation, the execution time depends on the block time rather than the function execution time. Therefore, all measured data points for all functions executed on the network (n=180) are aggregated to calculate some statistics. In this analysis, the Hardhat's "auto" feature was used to determine the gas price and gas limit. This prevents the user from over- or underpaying and estimates the transaction parameters reasonably.

Table 5.2 shows the minimum, maximum, mean and median time it took a transaction to be included in a mined/validated block. The local Hardhat network is the fastest, using its *automine* configuration that mines a block as soon as a transaction is received. On both public test networks, Ethereum Ropsten and Polygon Mumbai, the transactions were significantly slower, which resembles the execution on a public main network. Over the last two years (June 25, 2020 - June 25, 2022), the Ethereum main network experienced an average block time of 13.26 seconds and a median of 13.19 seconds. Polygon costs less to build on (as shown above) and reports a much shorter block time averaging 2.19 seconds, with a median of 2.16 seconds. Block times are good reference points for performance during low congestion since transactions are included relatively quickly. During congested times, the time it takes for a transaction to be included in a block can increase by a multiple if the priority fees are not adjusted accordingly.

Predicting execution times of decentralized applications in an absolute manner is impossible since it depends on many factors, such *e.g.*, as block time, congestion, and priority fees. Therefore the best thing one can do is to come up with estimations based on the

Table 5.2: Recorded SC Performance n=180

Blockchain	Min	Max	Mean	Median
Ethereum Ropsten	9.5s	230.4s	22.8s	22.0s
Polygon Mumbai	6.0s	335.7s	16.9s	10.9s
Local Hardhat	0.024s	0.186s	0.038s	0.032s

current state of the BC. Choosing the BC that best suits the application is an important decision to make and is one of the most significant decisions that can be taken regarding the execution times of a SC-based application.

5.3 Security Analysis

Once deployed, SCs perform immutable business logic and can store large amounts of tokens, which often have a monetary value. This makes them targets for attackers looking to profit by exploiting vulnerabilities in SC and unexpected behavior of the BC. Patching issues in a SC is usually impossible, which is also true for recovering the stolen tokens [16]. Well-performed security analysis and auditing of the SC by an experienced party helps identify issues before deployment and prevent them in production. This chapter analyses and evaluates the developed SC regarding security.

There are plenty of tools that help developers identify bugs in their SC. One example is *Slither*, a static analysis framework written in Python. It runs a suite of vulnerability detectors against a specific SC and prints out information about bugs or optimization potential it has detected [28]. Running Slither against our SC did not identify any security issues but only proposed to change some *public* functions to *external*.

Since the SC does not store or interact with any tokens, it does not expose itself to the risk or the incentive of stealing tokens from it. Further, there is no general monetary incentive around the functionality of the SC that could motivate people to exploit the behavior of the BC in their favor, such as *frontrunning*.

The SC exposes an apparent vulnerability regarding the timestamps used throughout the system. The first problem is caused when a supply chain entity calls a function of the SC at the wrong point in time, injecting erroneous information into the SC. A second problem arises because the miner can slightly manipulate the block time, which would also lead to inaccurate information. The latter is not a big issue since the range within which the miner can manipulate the timestamp is relatively tiny. A new block can not be timestamped earlier than its parent and not too far into the future, as no miner would build on that block [15].

The former can be fixed by establishing an incentive structure within the supply chain, which motivates the entities to write the data to the BC at the right time.

Overall, the SC hardly exposes any incentive structure to perform malicious acts against it, and modifying the state of the SC is sharply restricted only to trusted entities granted

Table 5.3: Stakeholders of the CheeseChain [41]

Category	Description	Stakeholder
Invested	Funded the project and supervise the project's development towards its conclusion.	Innosuisse
Primary	Their participation substantially influences the success or failure of the project.	Agroscope, interprofession Tête-de-Moine, Fromarte, Fromages Spielberger, Federal Food Safety and Veterinary Office
Secondary	Their acceptance and consent have limited influence on the success or failure of the project	Cheese dairies, retailers, cantonal laboratories, Organisme Intercantonal de Certification (OIC), milk producers, regional office for dairy consulting
Ternary	Use the products or services provided by the project	Consumers

access by the administrator of the SC. Regarding the visibility of the data stored within the SC, everything written to it is publicly visible, which matches the system's requirements. Therefore, the security risk of the CheeseChain SC solution is evaluated to be low.

5.4 Use Case

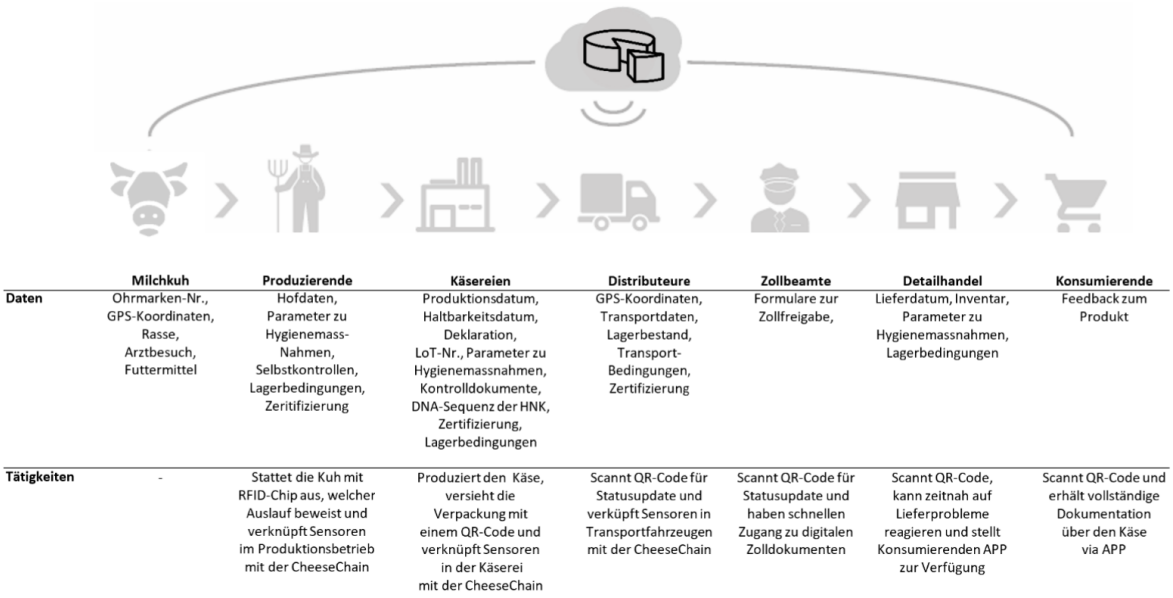
To this point, this thesis focused on presenting the developed system and its general use case. This section goes a step further and applies it to fit the specific use case it was created for in the first place, the CheeseChain.

First, the stakeholders of the CheeseChain are presented. Then, an evaluation is conducted on a representative use case, including said stakeholders.

5.4.1 Stakeholders

The CheeseChain project is funded by *Innosuisse*, the Swiss Agency for advances in innovation, and is comprised of a consortium consisting of the following four organizational entities: Agroscope, University of Zurich, Tête-de-Moine varietal organisation and Fromarte [17]. In addition to this consortium, additional stakeholders are defined for the CheeseChain and categorised into four categories by [41].

Table 5.3 gives an overview of all identifiable stakeholders and sorts them into categories according to the significance of their involvement in the CheeseChain. The following paragraphs introduce the primary stakeholders and explain their involvement in the CheeseChain to understand their role and interplay in the project.



	Milchkuh	Produzierende	Käsereien	Distributeure	Zollbeamte	Detailhandel	Konsumierende
Daten	Ohrmarken-Nr., GPS-Koordinaten, Rasse, Arztbesuch, Futtermittel	Hofdaten, Parameter zu Hygienemass-Nahmen, Selbstkontrollen, Lagerbedingungen, Zeritifizierung	Produktionsdatum, Haltbarkeitsdatum, Deklaration, LoT-Nr., Parameter zu Hygienemassnahmen, Kontrolldokumente, DNA-Sequenz der HNK, Zertifizierung, Lagerbedingungen	GPS-Koordinaten, Transportdaten, Lagerbestand, Transport-Bedingungen, Zertifizierung	Formulare zur Zollfreigabe,	Lieferdatum, Inventar, Parameter zu Hygienemassnahmen, Lagerbedingungen	Feedback zum Produkt
Tätigkeiten	-	Stattet die Kuh mit RFID-Chip aus, welcher Auslauf beweist und verknüpft Sensoren im Produktionsbetrieb mit der CheeseChain	Produziert den Käse, versieht die Verpackung mit einem QR-Code und verknüpft Sensoren in der Käserei mit der CheeseChain	Scannt QR-Code für Statusupdate und verküpft Sensoren in Transportfahrzeugen mit der CheeseChain	Scannt QR-Code für Statusupdate und haben schnellen Zugang zu digitalen Zolldokumenten	Scannt QR-Code, kann zeitnah auf Lieferprobleme reagieren und stellt Konsumierenden APP zur Verfügung	Scannt QR-Code und erhält vollständige Dokumentation über den Käse via APP

Figure 5.2: A Simplified Cheese Supply-Chain [41]

Agroscope is the federal competence center for agricultural research in Switzerland. They supply Tête-de-Moine dairies with bacterial cultures needed to achieve proof of origin of the produced cheese. The provided bacterial cultures are compared to the effective cultures in a sold cheese. This action is taken within a laboratory by conducting a PCR test and helps determine a cheese's authenticity [41].

The **Interprofession Tête-de-Moine** was founded as a supporting organization for the registration of the Tête-de-Moine PDO. The association includes milk producers, cheese dairies and affineurs, and aims to represent and market the Tête-de-Moine cheeses to authorities and the public at home and abroad. Further, they pursue the fight against counterfeits and provide the specifications for obtaining the PDO seal [44].

It is assumed that future audits of the cheese dairies can be carried out within the infrastructure of the CheeseChain, and thus digitalised certification will be possible. Therefore, the CheeseChain heavily depends on its informatics infrastructure [41].

Fromarte is the umbrella organization of Swiss cheese artisans, committed to strengthening the commercial structures within milk processing. Fromarte unites 500 commercial cheese dairies in Switzerland and offers them various services and training. In addition, they are involved in public relations and politics [39]. Their quality management tool QS Fromarte Digital is highly relevant for the CheeseChain. The application supports recording information on fabrication controls, cleaning, maintenance schedules and analysis results on compliance [41]. The CheeseChain bridges the data stored from the application to the BC.

Fromages Spielhofer is a milk processing company that produces and refines Tête-de-Moine PDO. They are also responsible for the trade of the cheese produced by themselves and that produced by neighboring dairies. Further, the company records that 80% of the cheese produced is sold abroad [38]. Within the CheeseChain, they aim to strengthen the

consumers' trust abroad and are mainly responsible for handling product and export data [41].

One of the tasks performed by **Federal Food Safety and Veterinary Office** is food inspection along the supply chain, which tests for compliance with hygiene and ingredient regulations and is conducted by random sampling. Farm managers must report a wide range of data to the authorities. This data is fed into the federal government's Agricultural Policy Information System (AGIS). The CheeseChain could be linked to AGIS to provide increased data exchange and food safety. Further, it is assumed that coordinated interaction of fraud detection and food inspections will allow food counterfeiting to be detected more efficiently with the help of statistical indicators [41].

5.4.2 Scenario

This subsection represents a scenario analysis of the CheeseChain. It discusses how the developed system could be applied to the specific use-case of a cheese supply chain and evaluates the costs of one cycle. [41] presents a simplified use case of the cheese supply-chain in Figure 5.2.

Figure 5.2 shows a use case with seven entities. Notably lacking is a laboratory entity conducting the proof-of-origin tests. The system at its current state neither supports tracking of individual cows during milk production nor does it support access to customs documents to facilitate customs clearance. Removing these two entities and adding the laboratory, the scenario is shrunk to six entities:

- **Milk Producer:** The milk producer registers every new batch of milk in the SC. The milk producer does not have to know in which dairy or lot the milk will eventually end up.
- **Cheese Dairy:** The cheese dairy registers a new lot and specifies from which batches of milk the cheese will be produced. Once the cheese is produced, the first step is recorded to complete the production. As the Tête-de-Moine usually ripens for four months, another step is recorded once the cheese is ready to be sold.
- **Distributors:** Distributors pick up the cheese in the dairy and deliver it to the retailer who purchased it. It is important to know the location where the cheese was picked up and where it was delivered, as well as the time delta between these two locations. For this reason, the distributor records a step at pick-up.
- **Retailers:** The retailer then confirms the receipt of the goods by adding another step.
- **Laboratory:** Representatives of the laboratory entity buy a random sample of cheese from the shelves of the retailer and conduct proof-of-origin tests on it. The results are added to the SC.

Table 5.4: Transactions in the Sample Scenario

Transaction	**Count**	**Description**	**Total Cost in CHF on Ethereum (Polygon)**
Add Milk Batch	1	In this scenario all the milk used stems from one batch	10.36 (0.0078)
Add Lot	1	Registering a lot is necessary to ingest information about the milk used and add further steps.	7.97 (0.0060)
Add Step	4	Cheese dairy (2), Distributor (1), Retailer (1)	53.25 (0.0399)
Add Lab Result	1	The Laboratory adds test results	5.23 (0.0039)
Total	**7**		**76.82 (0.0576)**

Table 5.5: One Time (Fixed) Costs

Transaction	**Count**	**Description**	**Total Cost in CHF on Ethereum (Polygon)**
Contract Deployment	1	Deployment of the SC is necessary to use its functions	130.46 (0.0978)
Add Participants	5	Registering each entity is necessary to grant write permissions on the SC	66.30 (0.0497)
Add Milk Batch	1	The additional cost of the first milk batch added to the SC	1.27 (0.0010)
Add Lot	1	The additional cost of the first lot added to the SC	1.27 (0.0010)
Add Step	1	The additional cost of the first step added to the SC	1.07 (0.0008)
Total	**8**		**200.38 (0.1502)**

- **Customers:** Once the customer holds the cheese, he can retrieve the cheese's history from the website with the frontend application by entering the lot identifier or scanning a QR code. The customer only reads the SC; therefore, no transaction is performed.

Table 5.4 summarizes all transactions performed on the SC to facilitate the described scenario and the cost associated with each. The several one-time costs of operating the CheeseChain SC are displayed separately in Table 5.5, including the higher gas cost for first call of `addMilkBatch` and `addLot`, the registration of all participants and the contract deployment. These costs do not belong to the variable costs of a normal scenario but can be considered fixed costs of the system. It is assumed that most laboratory results will be positive. A negative test result would decrease the costs by 1.48 CHF on Ethereum and by 0.0011 CHF on Polygon.

5.5 Discussion

The performance and cost of operating a dApp depend on the system's BC used as infrastructure. Therefore, selecting the best suitable BC has to be done with great diligence [62]. The Ethereum Mainnet and Ropsten Testnet operating on the PoW consensus mechanism, are significantly more expensive and less performant than their Polygon alternatives which run on a PoS consensus mechanism.

Analyzing the performance of public BCs has shown that the performance of a system is highly influenced by block time. Further, other factors, *e.g.*, network congestion and priority fees influence the time needed for a transaction to be included in a block. While influenced by several factors, the performance of dApp functionalities relying on the underlying BC is not deterministic. This is emphasized by the outliers recorded in a relatively uncongested BC environment while performing tests with similar parameters.

The performance of the system does not need to be real-time. A rough time estimate of each step in the supply chain is enough to construct a valuable and understandable history of the cheese. Even if it were to be decided that exact production timestamps have to be used, an additional property providing this information could be added to the `Step` struct. Adding this property would reduce the importance of the performance, as it would only be necessary that the steps are recorded in the correct order. The exact time the transaction was included in a block would not provide any valuable additional information.

The execution of a particular transaction on the EVM requires the same amount of gas, independent of the BC used, assuming that the EVM version is the same. This is due to the deterministic nature of the Ethereum EVM. Therefore, the resulting price for a transaction is determined by the gas and the native token price of each BC. A total of 14 transactions, respectively nine transactions when excluding the registration of participants, can add up to a substantial monetary cost on the Ethereum network. This operational cost can be reduced by 1334 times using Polygon alternatively to the Ethereum Mainnet.

Regarding SC security, the system has a shallow risk since the lack of monetary valuable tokens in the contract as well as a lack of an obvious incentive to exploit potential vulnerabilities in the SC. However, bad private key management and dishonest participants can put the integrity of the SC at risk. If the administrator's keys were to be leaked, anybody knowing the key could make changes to the system *e.g.*, changing the laboratory results or adding new lots. Should any other participant leak their key, the risk of incorrect information entering the system persists. However, the administrator could remove the affected participant from the system and prevent other malicious behavior.

With the combined lower transaction cost and improved throughput offered by Polygon BC, in tandem with the low-security risk of the SC, the author prefers the Polygon BC for the system's operation and thinks the cost and performance are acceptable for a real-world use case. Since technology evolves and the requirements differ for each system, one must evaluate each case separately and can not rely on a universally correct answer.

Chapter 6

Summary, Conclusions, and Future Work

This chapter concludes the work with a summary and some conclusions, listing potential areas of future work based on insights gained during development.

6.1 Summary

This thesis aimed to create a Smart Contract (SC) based system for tracking within a Swiss cheese supply chain. The system comprises a SC, a frontend and a server, for communication with the SC through a website or API calls. In order to achieve this, the basics of BCs and SCs have been studied. Thereafter, existing solutions in the area of supply chain tracking and anti-counterfeiting with the help of BC have been researched. As such, the developed SC presents a novel append-only solution for the general supply chain tracking use case.

This was achieved by abstracting a supply chain into a product batch and steps registered on the batch during the production process. The product batch is implemented and recorded as a custom data type, a struct, that stores information about the product and keeps track of the last production step performed. Each step is a struct containing a timestamp, coordinates, the BC address of the entity that added the step and a pointer to the previous step in production, forming an immutable backward traceable linked list of steps. Further, an access control mechanism was integrated into the SC and is managed by the system administrator and allows only authorized and registered entities within the supply chain to make changes to the SC. This is necessary since the SC is intended to be deployed on a public permissionless BC, where, if not restricted, any user could write to the SC.

Such a SC was evaluated in terms of cost, performance and security using the Ethereum Ropsten Testnet, the Polygon Mumbai Testnet and a local Ethereum Network. Finally, a representative use case scenario of the system within the Tête-de-Moine supply chain was constructed, and the costs for its operation were presented.

6.2 Conclusions

This work has presented a new approach to supply chain tracking, leveraging BC technology and SCs for the EVM. When designing a dApp, choosing the best suitable BC is important and will impact a project's performance, cost and success. The cost of operating the system is volatile and dependent on technological advances of the chosen BC, as well as on the current native token and gas price. Performance is another variable factor of the system, which is influenced by multiple factors *e.g.*, network congestion, block time and priority fees.

The developed SC is tailored to the use case of tracking the Tête-de-Moine supply chain and helps to enhance transparency and trust along the entire production and supply chain. Due to the lower transaction costs on the Polygon Network, deploying the system on Polygon is much more cost-efficient than deploying it on the Ethereum Mainnet. In order to fulfill its purpose, it is vital that every step within the supply chain is recorded, or the history of an item would not be complete. Hence, the developed system is agnostic to the choice of BC as long as it executes SCs on the EVM. The SC architecture proposed is flexible by design and allows for adaption and extension to fit any SCT use case.

6.3 Future Work

The system is usable for traceability along a cheese supply chain, especially the Tête-de-Moine supply chain. It demonstrates a novel approach to BC-based supply chain traceability, which allows customers to validate their products on-chain. However, there is still great potential for improvements and future work, which is outside this thesis's scope.

As of now, a step allows participants to record a specific set of data points, which is equal for all production steps. Considering that most steps in production are different from one another, as well as not equally important and regulated, the most insightful and relevant information can vary. An improved version of the SC could consider extending the `Step` struct to store more information or even create a custom data type for each step that produces extra valuable information, *i.e.*, the temperature at which the cheese was transported. Further, steps could be added more frequently and with greater granularity producing a greater insight into the supply chain.

There is further room for improvement regarding steps, as the current implementation assumes that at any point within the supply chain, only one step is performed and that the output from that step is being used for the next step. This might be accurate for cheese production; in more complex supply chains, it is common that two or more raw materials or intermediary products are built simultaneously and then used together in the following step. In order to allow the system to handle more complex cases, the SC should support one step to follow two parallel steps and two parallel steps to follow one single step.

As it is crucial that entities along the supply chain record their performed steps, it is important to make interaction with the SC as easy as possible. A mobile or web application could facilitate that process, allowing participants to scan a QR code or Near-Field Communication (NFC) tag available on the product. The use of sensors allows for recording more detailed data and makes the system less prone to human error or injection of faulty information. Instead of relying on the separate supply chain entities to provide the coordinates where a step was performed, adding a GPS tracker could provide more complete and insightful information.

Since more information means higher storage costs when using BCs, data could be stored on an off-chain DB or on IPFS to either decrease the storage costs of the current implementation or decrease the storage costs of an adapted implementation that uses more data. This solution requires the hash of the data stored off-chain to be stored inside the SC, which makes the system more cost-efficient while still assuring data immutability.

Lastly, a tighter designed access control mechanism would increase the security of the SC and allow for operation in lower trust supply chain environments. Currently, the CheeseChain relies on a degree of trust, which allows every `Basic` participant to deploy a lot and add any step to any given lot at any time. A possible solution is to allow the system administrator to assign participants to a set of lots they are allowed to manipulate together with more specific roles.

Bibliography

[1] American Institute of CPAs and Chartered Professional Accountants of Canada (AICPA and CPA Canada). Blockchain Technology and Its Potential Impact on the Audit and Assurance Profession, 2017. `https://bit.ly/3IzTxFg`, Last visit March 21, 2022.

[2] Andreas M Antonopoulos. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*. O'Reilly Media, Inc., 2014.

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

[4] Imran Bashir. *Mastering Blockchain*. Packt Publishing Ltd., Birmingham, UK, 2017.

[5] B. M. A. L. Basnayake and C. Rajapakse. A Blockchain-based decentralized system to ensure the transparency of organic food supply chain. In *2019 International Research Conference on Smart Computing and Systems Engineering (SCSE)*, pages 103–107, Colombo, Sri Lanka, March 2019.

[6] Kamanashis Biswas, Vallipuram Muthukkumarasamy, and Wee Lum Tan. Blockchain based wine supply chain traceability system. In *Future Technologies Conference (FTC) 2017*, pages 56–62, Vancouver, Canada, November 2017. The Science and Information Organization.

[7] Marshall Blair. How are transactions validated?, 2018. `https://medium.com/@blairlmarshall/how-do-miners-validate-transactions-c01b05f36231`, Last visit April 22, 2022.

[8] Thomas Bocek, Bruno Rodrigues, Tim Strasser, and Burkhard Stiller. Blockchains Everywhere - a Use-Case of Blockchains in the Pharma Supply-Chain. In *IFIP/IEEE Symposium on Integrated Network and Service Management (IM 2017)*, pages 772–777, Lisbon, Portugal, May 2017.

[9] Dimitar Bogdanov. Proof of Authority Explained, 2021. `https://limechain.tech/blog/proof-of-authority-explained/`.

[10] Bolsey Catherine. Switzerland Combats Counterfeit Cheese With DNA Fingerprints, 2014. `https://www.swissinfo.ch/eng/bloomberg/switzerland-combats-counterfeit-cheese-with-dna-fingerprints/40574480`, Last visit March 15, 2022.

[11] Etherscan Information Center. What is an Ethereum Address?, 2021. https://info.etherscan.com/what-is-an-ethereum-address/, Last visit April 18, 2022.

[12] Glad Chinda. Mocha.js, the JavaScript test framework: A tutorial, 2020. https://blog.logrocket.com/a-quick-and-complete-guide-to-mocha-testing-d0e0ea09f09d/, Last visit June 7, 2022.

[13] CoinMarketCap. Ether Price, 2022. https://coinmarketcap.com/currencies/ethereum/, Last visit June 20, 2022.

[14] CoinMarketCap. Matic Price, 2022. https://coinmarketcap.com/currencies/polygon/, Last visit June 20, 2022.

[15] Jeff Coleman. Can a contract safely rely on block.timestamp?, 2016. https://ethereum.stackexchange.com/questions/413/can-a-contract-safely-rely-on-block-timestamp, Last visit June 25, 2022.

[16] Ethereum Commnity. Smart Contract Security, 2022. https://ethereum.org/en/developers/docs/smart-contracts/security/, Last visit June 25, 2022.

[17] Communication Systems Group (CSG). Application of Blockchain Technology in the Swiss Cheese Supply Chain (CheeseChain), 2022. https://www.csg.uzh.ch/csg/en/research/CheeseChain.html, Last visit March 8, 2022.

[18] Ethereum Community. Set up Web3.js to use the Ethereum Blockchain in Javascript, 2020. https://ethereum.org/en/developers/tutorials/set-up-web3js-to-use-ethereum-in-javascript/#main-content, Last visit March 26, 2022.

[19] Ethereum Community. Ethereum Accounts, 2022. https://ethereum.org/en/developers/docs/accounts/.

[20] Ethereum Community. Gas and Fees, 2022. https://ethereum.org/en/developers/docs/gas/.

[21] Ethereum Community. Introduction to dapps, 2022. https://ethereum.org/en/developers/docs/dapps/, Last visit March 26, 2022.

[22] Ethereum Community. Proof-of-Stake (POS), 2022. https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/.

[23] Ethereum Community. PROOF-OF-WORK (POW), 2022. https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/, Last visit April 22, 2022.

[24] Ethereum Solidity Community. Contract ABI Specification, 2022. https://docs.soliditylang.org/en/v0.8.12/abi-spec.html,, Last visit June 13, 2022.

[25] Ethereum Solidity Community. Introduction to Smart Contracts, 2022. https://docs.soliditylang.org/en/v0.8.13/introduction-to-smart-contracts.html, Last visit April 22, 2022.

[26] EthersJs Community. Providers, 2022. `https://docs.ethers.io/v5/api/providers/`, Last visit June 13, 2022.

[27] Michael Crosby, Pradan Pattanayak, Sanjeev Verma, Vignesh Kalyanaraman, et al. Blockchain technology: Beyond bitcoin. *Applied Innovation*, 2(6-10):71, 2016.

[28] Crytic. Slither, the Solidity source analyzer, 2022. `https://github.com/crytic/slither`, Last visit June 25, 2022.

[29] Jean Cvllr. Solidity Tutorial: all about modifiers, 2021. `https://medium.com/coinmonks/solidity-tutorial-all-about-modifiers-a86cf81c14cb`, Last visit June 7, 2022.

[30] Burkhard Stiller Danijel Dordevic, Sina Rafati. Application of Blockchain Technology in the Swiss Food Value Chain (Foodchains), 2019. Research and Industrial Project, Communication Systems Group, Department of Informatics, Universität Zürich.

[31] Chris Dannen. *Introducing Ethereum and solidity*, volume 1. Springer, 2017.

[32] Ruma Das. Proof of Authority, 2020. `https://medium.com/coinmonks/proof-of-authority-ac34f1b3a2c2`.

[33] Stefano De Angelis, Leonardo Aniello, Federico Lombardi, Andrea Margheri, and V. Sassone. PBFT vs proof-of-authority: applying the CAP theorem to permissioned blockchain. 01 2017.

[34] Ethereum.Stackexchange - user 'q9f. What is an ABI and why is it needed to interact with contracts?, 2016. `https://ethereum.stackexchange.com/questions/234`, Last visit June 13, 2022.

[35] Etherscan. Ethereum Average Gas Price Chart, 2022. `https://etherscan.io/chart/gasprice`, Last visit June 20, 2022.

[36] Tiago M. Fernández-Caramès and Paula Fraga-Lamas. Towards Post-Quantum Blockchain: A Review on Blockchain Cryptography Resistant to Quantum Computing Attacks. *IEEE Access*, 8:21091–21116, 2020.

[37] Nomic Foundation. Hardhat Documentation, 2022. `https://hardhat.org/getting-started`, Last visit June 6, 2022.

[38] Fromages Spielhofer. Kompetenzen, 2022. `https://www.fromagesspielhofer.ch/uber-uns/kompetenzen/`, Last visit July 1, 2022.

[39] Fromarte. über fromarte, 2022. `https://www.fromarte.ch/de/ueber-uns/fromarte`, Last visit July 1, 2022.

[40] Arthur Gervais, Ghassan O. Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the Security and Performance of Proof of Work Blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS 16, pages 3–16, Vienna, Austria, October 2016. Association for Computing Machinery.

[41] Luca Girardi. Applikation der Blockchain-Technologie zur Authentifizierung von TTête de Moine AOP, December 2021. Bachelor Thesis, College of Agricultural, Forest and Food Sciences HAFL.

[42] Dominique Guegan. Public blockchain versus private blockhain. 2017.

[43] Parikshit Hooda. Proof of Work (PoW) Consensus, 2019. `https://www.geeksforgeeks.org/proof-of-work-pow-consensus/`, Last visit April 22, 2022.

[44] Interprofession Tête-de-Moine. Die organisation hinder dem Tête-de-Moine AOP, 2022. `https://www.tetedemoine.ch/de/infos/verwertungskette`, Last visit July 1, 2022.

[45] Gwyneth Iredale. Public Vs Private Blockchain: How Do They Differ?, 2021. `https://101blockchains.com/public-vs-private-blockchain/`, Last visit April 12, 2022.

[46] Hussam Juma, Khaled Shaalan, and Ibrahim Kamel. A Survey on Using Blockchain in Trade Supply Chain Solutions. *IEEE Access*, 7:184115–184132, 2019.

[47] Zachary C Kennedy, David E Stephenson, Josef F Christ, Timothy R Pope, Bruce W Arey, Christopher A Barrett, and Marvin G Warner. Enhanced anti-counterfeiting measures for additive manufacturing: coupling lanthanide nanomaterial chemical signatures with blockchain technology. *Journal of Materials Chemistry C*, 5(37):9570–9578, 2017.

[48] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual international cryptology conference*, pages 357–388, California, USA, August 2017. Springer.

[49] Manlu Liu, Kean Wu, and Jennifer Jie Xu. How will blockchain technology impact auditing and accounting: Permissionless versus permissioned blockchain. *Current Issues in auditing*, 13(2):A19–A29, 2019.

[50] Petra Lüdin, Ueli von Ah, Deborah Rollier, Alexandra Roetschi, and Elisabeth Eugster. Lactic Acid Bacteria as Markers for the Authenticitation of Swiss Cheeses. *CHIMIA International Journal for Chemistry*, 70(5):349–353, 2016.

[51] Juri Mattila. The blockchain phenomenon. *Berkeley Roundtable of the International Economy*, 16, 2016.

[52] MetaMask. A crypto wallet & gateway to blockchain apps, 2022. `https://metamask.io`, Last visit June 13, 2022.

[53] Elena Nadolinski and Facu Spagnuolo. Proxy Patterns, 2018. `https://blog.openzeppelin.com/proxy-patterns/`, Last visit June 7, 2022.

[54] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. `https://bitcoin.org/bitcoin.pdf`.

[55] Michael Nofer, Peter Gomber, Oliver Hinz, and Dirk Schiereck. Blockchain. *Business & Information Systems Engineering*, 59(3):183–187, 2017.

[56] Polygonscan.com. Polygon PoS Chain Average Gas Price Chart, 2022. `https://polygonscan.com/chart/gasprice`, Last visit June 20, 2022.

[57] Shaan Ray. The Difference Between Traditional and Delegated Proof of Stake, 2018. `https://hackernoon.com/the-difference-between-traditional-and-delegated-proof-of-stake-36a3e3f25f7d`.

[58] D Sathya, S Nithyaroopa, D Jagadeesan, and I Jeena Jacob. Block-chain technology for food supply chains. In *2021 Third International Conference on Intelligent Communication Technologies and Virtual Mobile Networks (ICICV)*.

[59] Eder J. Scheid, Daniel Lakic, Bruno B. Rodrigues, and Burkhard Stiller. PleBeuS: a Policy-based Blockchain Selection Framework. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–8, Budapest, Hungary, April 2020.

[60] Eder J. Scheid, Bruno Rodrigues, Christian Killer, Muriel Franco, Sina Rafati, and Burkhard Stiller. Blockchains and Distributed Ledgers Uncovered: Clarifications, Achievements, and Open Issues. In *Advancing Research in Information and Communication Technology*, IFIP AICT Festschrifts, pages 1–29. Springer, Cham, Switzerland, August 2021.

[61] Eder J. Scheid, Bruno Rodrigues, and Burkhard Stiller. Toward a Policy-based Blockchain Agnostic Framework. In *IFIP/IEEE Symposium on Integrated Network and Service Management (IM 2019)*, pages 609–613, Washington D.C., USA, April 2019.

[62] Eder J. Scheid, Bruno Rodrigues, and Burkhard Stiller. Policy-based Blockchain Selection. *IEEE Communications Magazine*, 59(10):48–54, 2021.

[63] Jason Shah. How to Send Ethereum Transactions Using Web3, 2021. `https://betterprogramming.pub/how-to-send-ethereum-transactions-using-web3-d05e0c95f820`, Last visit June 13, 2022.

[64] Affaf Shahid, Ahmad Almogren, Nadeem Javaid, Fahad Ahmad Al-Zahrani, Mansour Zuair, and Masoom Alam. Blockchain-Based Agri-Food Supply Chain: A Complete Solution. *IEEE Access*, 8:69230–69243, 2020.

[65] Moralis Web3 Technology. Web3.js vs Ethers.js - Guide to ETH JavaScript Libraries, 2022. `https://moralis.io/web3-js-vs-ethers-js-guide-to-eth-javascript-libraries/`, Last visit March 26, 2022.

[66] Lee Ting Ting. Comparison of The Top 10 Smart Contract Programming Languages in 2021, 2021. `https://pontem.network/posts/comparison-of-the-top-10-smart-contract-programming-languages-in-2021`, Last visit March 26, 2022.

[67] Karl Wust and Arthus Gervais. Do you Need a Blockchain? In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 45–54, Los Alamitos, CA, USA, jun 2018. IEEE Computer Society.

[68] Dylan Yaga, Peter Mell, Nik Roby, and Karen Scrfone. Blockchain Technology Overview. *arXiv preprint arXiv:1906.11078*, 2019.

[69] Jesse Yli-Huumo, Deokyoon Ko, Sujin Choi, Sooyong Park, and Kari Smolander. Where is current research on blockchain technology? A systematic review. *PloS one*, 11(10):e0163477, 2016.

[70] Changqiang Zhang, Cangshuai Wu, and Xinyi Wang. Overview of Blockchain Consensus Mechanism. In *Proceedings of the 2020 2nd International Conference on Big Data Engineering*, pages 7–12, Shanghai, China, May 2020. Association for Computing Machinery.

[71] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. Blockchain challenges and opportunities: A survey. *International Journal of Web and Grid Services*, 14(4):352–375, 2018.

[72] Peng Zhu, Jian Hu, Yue Zhang, and Xiaotong Li. A Blockchain Based Solution for Medication Anti-Counterfeiting and Traceability. *IEEE Access*, 8:184256–184272, 2020.

[73] Maarten Zuidhoorn. The Magic of Digital Signatures on Ethereum, 2020. `https://medium.com/mycrypto/the-magic-of-digital-signatures-on-ethereum-98fe184dc9c7`, Last visit April 18, 2022.

Abbreviations

ABI	Application Binary Interface
ACM	Access Control Mechanism
API	Application Programming Interface
BC	Blockchain
CM	Consensus Mechanism
dApp	Decentralized Application
DB	Database
DPoS	Delegated Proof of Stake
EOA	Externally Owned Accounts
ETH	Ether
EVM	Ethereum Virtual Machine
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
IPFS	Interplanetary File System
JSON	JavaScript Object Notation
PCR	Polymerase Chain Reaction
PDO	Protected Designation of Origin
PoA	Proof of Authority
PoS	Proof of Stake
PoW	Proof of Work
PUC	Public CheeseChain Solution
QR	Quick Response
SC	Smart Contract
SCC	Smart Contract Connector
SCL	Smart Contract Language
SCT	Supply Chain Tracking
UI	User Interface
Uint	Unsigned Integer
URI	Unique Resource Identifier
UX	User Xperience

List of Figures

List of Tables

Appendix A

Installation Guidelines

The CheeseChain is a project developed by Matteo Gamba within the scope of his Bachelor thesis and introduces **supply chain tracking to the Swiss cheese industry**.

A.1 Overview

The project consists of three main parts:

- Chain
- Frontend
- Server

Each of the parts has to be spun up, respectively deployed separately and configured in order to work properly. The following sections explain how to get each part up and running.

A.1.1 Chain

Requirements: NodeJs installed

The chain consists of all Smart Contract specific source code (i.e. Smart Contracts, tests, hardhat development environment).

1. **Install Node modules:** `yarn install`
2. **Create .env file:** Follow instructions in `.env.example` file.
3. **Compile Smart Contracts:** This will compile your contracts and copy the artifacts into the frontend folder, where it is necessary in order to deploy the Smart Contract and communicate with it `npx hardhat compile`
4. **Start a local Hardhat network:** `npx hardhat node`

Congratulations, you have a local Ethereum network running, ready to develop and deploy your Smart Contracts on.

If you want to connect a frontend or server to the smart contract run the following commands to generate TypeScript types from the contract ABI and copy them to the frontend/server directory:

1. **Generate types:** `npx hardhat typechain`
2. **Copy types to frontend & server:** `yarn copy:types`

The server additinally needs to know the contract's ABI. For that run `yarn copy:abi`.

Deploy the contract via the Hardhat framework:

1. **Add the network:** In the `hardhat.config.ts` file add the network configurations of the netowork you wish to deploy to. Here[1] is a guide.
2. **(Optional) Create .env file:** Add sensitive information needed in step 1 by to a .env file, otherwise leave it inside the `hardhat.config.ts`.
3. **(Optional) Add Etherscan information in `hardhat.config.ts`:** If you want to verify the Smart Contract via Hardhat, uncomment the *Etherscan* section and add your information or follow the instructions[2].
4. **Deploy the Smart Contract:** `npx hardhat deploy -network <your-network>`. The network names must be equal to those specified in the `hardhat.config.ts` file.
5. **(Optional) Verify the Smart Contract:** `npx hardhat verify -network <your-network> DEPLOYED_CONTRACT_ADDRESS`. Find the instructions here[3].

To run the test suite:

1. **(Optional) Add GasReporter:** If you want to get a gas report on all Smart Contract functions uncomment *gasReporter* inside the `hardhat.config.ts` and add you information or follow the guide[4].
2. **Run the Tests:** `npx hardhat test`

*Note: All the environment variables inside the **chain** directory are optional and depend on the configuration of the `hardhat.config.ts` file*

A.1.2 Frontend

Requirements: NodeJs and Metamask browser extension installed

1. **Install Node modules:** `yarn install`

[1] <https://hardhat.org/hardhat-runner/docs/config>
[2] <https://hardhat.org/hardhat-runner/plugins/nomiclabs-hardhat-etherscan>
[3] <https://hardhat.org/hardhat-runner/plugins/nomiclabs-hardhat-etherscan>
[4] <https://www.npmjs.com/package/hardhat-gas-reporter>

2. **Create .env file:** Follow instructions in `.env.example` file.
3. **Serve the App locally:** `yarn start`

You now have a website running locally that is ready to be connected to the Ethereum network. Note that there are two routes implemented: The base route `BASE-URL/` only allows to retrieve the lot history via the frontend and is designed for the end customer. In order to connect a Metamask wallet to the frontend and interact with the smart contract the `BASE-URL/connect` url exists which allows all registered participants to call the smart contract functions they were authorized to by the system administrator If you want to connect to the local Hardhat network and interact with it, you must add the network to Metamask first. A guide of how to do this can be found at https://support.chainstack.com/hc/en-us/articles/4408642503449-Using-MetaMask-with-a-Hardhat-node.

A.1.3 Server

Requirements: NodeJs installed

The system does not require the server to be running in order to function. The server exposes a REST API interface and serves as a middleman to communicate with the Blockchain via HTTP requests.

1. **Install Node modules:** `yarn install`
2. **Create .env file:** Follow instructions in `.env.example` file.
3. **Run the server locally:** `yarn start`

Appendix B

Contents of the CD

The thesis was submitted as a *.zip-file* instead of a physical CD as agreed with the supervisor. The content of the ZIP includes:

- `thesis.zip` containing the Latex source code.
- `BA_Matteo_Gamba_Cheese_Chain_Final.pdf`, the final version of the thesis in PDF format.
- `code.zip` containing the source code of the SC, the frontend and the server as well as the figures created in draw.io in their .drawio format.
- `presentations.zip` containing the slides for the midterm presentation held on May 19, 2022 and the slides for the final presentation held on August 30, 2022.