



University of
Zurich^{UZH}

Integration and Deployment of a Blockchain-based Supply Chain Tracing Application

Valentin Meyer
Schwerzenbach, Switzerland
Student ID: 20-715-728

Supervisor: Jan von der Assen, Daria Schumm
Date of Submission: August 4, 2024

Zusammenfassung

Lieferketten sind komplexe Prozesse, an denen viele verschiedene Akteure beteiligt sind. Die Globalisierung hat die Komplexität dieser Prozesse weiter erhöht. Um die Transparenz, das Vertrauen und die Sicherheit in der Lieferkette des Schweizer Käses Tête-de-Moine AOP zu erhöhen, wurde das Projekt CheeseChain ins Leben gerufen. Ziel des Projekts ist die Entwicklung einer neuen Technologiekombination, bei der DNA-basierte Authentizitätstests mit der Blockchain-Technologie kombiniert werden. Die Blockchain in Verbindung mit Smart Contracts bietet eine ideale Basis, um dieses Ziel zu erreichen. Im Rahmen von zwei Bachelorarbeiten wurden Prototypen entwickelt: System 1 fokussiert auf den privaten Aspekt der Anwendung und verbindet das digitale Qualitätsmanagementsystem mit dem Smart Contract auf einer privaten Blockchain. System 2 ermöglicht es Akteuren der Lieferkette und Käseliebhabern, Daten über die Käseproduktion in die Blockchain hochzuladen und abzurufen. In dieser Bachelorarbeit wurde ein privates Ethereum Netzwerk implementiert, Fehler in beiden Systemen identifiziert und behoben. Die Systeme wurden mit Docker containerisiert und auf eine virtuelle Maschine deployt, um produktionsreif zu sein. Abschliessend wurde die implementierte Lösung hinsichtlich Laufzeit, Webperformance und Usability evaluiert.

Abstract

Supply chains are complex processes in which many different players are involved. Globalization has further increased the complexity of these processes. The CheeseChain project was launched to increase transparency, trust and security in the supply chain of Swiss Tête-de-Moine AOP cheese. The aim of the project is to develop a new combination of technologies in which DNA-based authenticity tests are combined with blockchain technology. The blockchain in conjunction with smart contracts offers an ideal basis for achieving this goal. Prototypes were developed as part of two bachelor theses: System 1 focuses on the private aspect of the application and combines the digital quality management system with the smart contract on a private blockchain. System 2 enables actors in the supply chain and cheese lovers to upload and retrieve data on cheese production in the blockchain. In this bachelor thesis, a private Ethereum network was implemented, bugs in both systems were identified and fixed. The systems were containerized with Docker and deployed on a virtual machine to be ready for production. Finally, the implemented solution was evaluated in terms of runtime, web performance, and usability.

Acknowledgments

This thesis is more than the work of one person. First and foremost, I would like to thank my primary supervisor, Jan von der Assen, for his valuable advice, quick responses to email inquiries, willingness to share his vast knowledge, and overall support. I would also like to thank my secondary supervisor, Daria Schumm, for her support.

Second, I extend my gratitude to Prof. Dr. Burkhard Stiller and the Communication Systems Group from the University of Zurich for their confidence and the opportunity to work on this project.

Finally, I would like to thank my father for his support and proofreading throughout the process, my girlfriend for her proofreading and unyielding support, my mother for her valuable proofreading, and my friends for their help and input.

Contents

Abstract	i
Acknowledgments	v
1 Introduction	1
1.1 Motivation	1
1.2 Description of the Work	2
1.3 Thesis Outline	3
2 Background	5
2.1 Supply Chain Management	5
2.2 Blockchain Technology	6
2.2.1 Architecture	6
2.2.2 Security Aspects	7
2.2.3 Consensus Mechanisms	8
2.3 Blockchain Types	8
2.4 Smart Contracts	9
2.4.1 Origin	9
2.4.2 SCs and BC	10
2.5 Ethereum	10
2.5.1 Pros and Cons	11
2.6 Containerization	11
2.7 The CheeseChain Project	12

3	Related Work	13
3.1	Methodology	13
3.2	Blockchain Technology in Supply Chain Tracking	14
3.2.1	Dairy Supply Chain Tracking	14
3.2.2	Cheese Supply Chain Traceability	15
3.2.3	Previous Theses	16
3.3	Summary	18
4	Architecture and Design	21
4.1	Application Scenarios	21
4.1.1	Application Scenario of System 1	21
4.1.2	Application Scenario of System 2	23
4.2	Technical View	24
4.2.1	Technical View of System 1	24
4.2.2	Technical View of System 2	26
4.3	Deployment Scenario	28
4.3.1	Deployment of System 1	28
4.3.2	Deployment of System 2	29
5	Implementation	31
5.1	Implementation of a Private Ethereum Network	31
5.2	Implementation of System 1	32
5.2.1	Phase 1: Local Execution and Debugging	32
5.2.2	Phase 2: Containerization	37
5.2.3	Phase 3: Migration to Production	39
5.3	Implementation of System 2	39
5.3.1	Phase 1: Local Execution and Debugging	40
5.3.2	Phase 2: Containerization	41

<i>CONTENTS</i>	ix
6 Evaluation	45
6.1 Runtime Evaluation	45
6.1.1 Test Setup	45
6.1.2 System 1	46
6.1.3 System 2	47
6.2 System 2 - Frontend	51
6.2.1 Lighthouse	51
6.2.2 Usability	52
6.3 Comparison with Related Work	54
7 Summary and Future Work	55
7.1 Summary	55
7.2 Future Work	56
Bibliography	57
Abbreviations	61
List of Figures	61
List of Tables	63
A Installation Guidelines	67
A.1 Private Network	67
A.2 System 1	68
A.3 System 2	69
B Answers Usability Evaluation	71
C Contents of the CD	75

Chapter 1

Introduction

This thesis deals with the integration and deployment of an application responsible for the traceability of the supply chain of Tête-de-Moine Appellation d'Origine Protégée (AOP) cheeses, using Blockchain (BC) technology. This application is the result of a collaborative project between FROMARTE, Agroscope, Tête-de-Moine AOP producers and the University of Zurich. It consists of two parts. System 1 is the private part of the application that acts as a connector between the Digital Quality Management System (DigitalQM) hosted by FROMARTE and a Smart Contract (SC) on a private Ethereum network. System 2 is the public part of the application that provides a **Frontend** and a **Server** to store and retrieve data from another SC on a public BC. The prototypes were developed as part of two previous bachelor theses.

1.1 Motivation

The application provides more trust and transparency along the value chain of Tête-de-Moine AOP cheese. Several technologies have been used to achieve this goal, with blockchain technology being the most notable for its immutability, decentralization and transparency. These characteristics ensure that the data cannot be tampered with, thus creating trust among the actors in the value chain and between them and their consumers.

Another important issue that this project aims to address is counterfeit cheese. The project uses a DNA-based intrinsic product authentication system to address this issue. The CheeseChain project combines this technology with BC technology for the first time. The Interprofession Tête-de-Moine has been a pioneer in fraud detection since 2013 [1], using a proof-of-origin culture developed by Agroscope.

The application integrated and deployed in this thesis plays an integral role in the CheeseChain project, providing the infrastructure for the BC component. The existing prototypes for the application, System 1 and System 2, have not been deployed, which is the goal of this thesis.

1.2 Description of the Work

Several steps were taken to achieve the goal of making System 1 and System 2 deployable.

1. **Getting Familiar with the Systems:**

The first step was to become familiar with the two systems, including understanding their design, the goals each system was intended to achieve, and potential application scenarios. This phase also included a detailed examination of the code to understand how it worked and to identify any initial problems.

2. **Local Setup and Debugging:**

Once the systems were understood, the next step was to make them work locally. This involved setting up the necessary environments on a local machine and debugging the prototypes to make sure they worked correctly. This debugging process was critical to identifying and fixing bugs that prevented the systems from working as intended.

3. **Containerization:**

The systems were then containerized using Docker, Docker Compose, and Docker Secrets. Containerization was necessary to ensure that the application could run consistently across different environments. This step also required modifying the existing code to be compatible with the containerized deployment.

4. **Private Ethereum Network Deployment:**

A private Ethereum network, based on an existing setup, was slightly modified and deployed. This network was essential for the operation of System 1, as it connects FROMARTE's DigitalQM to a SC on the private Ethereum network.

5. **VM Deployment:**

The final step in the implementation was to deploy the systems on a virtual machine (VM) hosted by the University of Zurich (UZH). This deployment was critical for testing the systems in an environment that closely resembled a production environment.

6. **Evaluation:**

Following the implementation, a thorough evaluation was conducted. This included:

- **Runtime:** Testing and evaluating the runtime of the systems to determine the impact of containerization and VM deployment on performance.
- **Performance Testing:** Using Lighthouse to test the performance of System 2's **Frontend**, focusing on key metrics such as speed and responsiveness.
- **Usability Evaluation:** Conducted a usability evaluation with three potential consumers to gather feedback on the user experience and identify areas for improvement.

1.3 Thesis Outline

The first chapter focuses on describing the motivation behind the thesis, the work done, and the outline of the thesis. Chapter 2 provides the reader with the necessary knowledge to understand the rest of the thesis. This includes, but is not limited to, a description of the origin, evolution, and technical aspects of BC technology and SCs, the motivation behind containerization, and a detailed description of the CheeseChain project.

Chapter 3 focuses on describing the work that has already been done in this area and identifies a research gap that this thesis aims to fill. Chapter 4 describes the systems in more technical detail, provides application scenarios for each, and concludes with a design proposal for implementation. Chapter 5 outlines the steps taken to implement the systems, illustrates the code modifications, and explains how the containerization and migration to the VM were performed.

Chapter 6 describes the evaluation of the final solution. It begins by examining whether and how the containerization and VM affected the runtime of System 1 and the **Frontend**, as well as the performance of the **Frontend**. It includes a usability test and concludes with a comparison of the results with the research gap identified in Chapter 3. Chapter 7 completes this thesis by providing a summary and suggestions for improvements to further enhance the application.

Chapter 2

Background

The purpose of this chapter is to provide the reader with the knowledge necessary to understand the technical aspects of the thesis. BC is a disruptive new technology that has emerged over the last two decades and is already impacting and even changing entire industries [2]. The foundation for BC technology was laid in 2008 with the release of the white paper for the bitcoin cryptocurrency. The idea was to create a decentralized digital currency that was both secure and independent of a third party [3]. In the years that followed, other use cases for BC technology emerged, such as supply chain tracking, financial trading, banking, healthcare, smart property, and others [2], [4].

2.1 Supply Chain Management

Supply chain management is a complex and challenging task that aims to "maximize customer value and achieve sustainable competitive advantage". There are three key flows that are integrated in supply chain management, the flow of information, the flow of products and materials, and the flow of funds. Two different types of supply chains can be identified, basic and extended supply chains. Basic supply chains involve at least three companies directly linked by one or more upstream or downstream flows of products/-materials, funds or information. Extended supply chains also include suppliers of direct suppliers and customers of direct customers [5].

The concept of Supply Chain Tracking can be further illustrated with an example. Figure 2.1 is a simplification of the Tête-de-Moine AOP cheese value chain. It starts with milk production in the Jura mountain region of Switzerland. Milk producers must follow strict guidelines that prohibit the use of animal meal, hormones and genetically modified products. After production, the milk is transported to the production facilities. At these facilities, the milk is first aged to achieve the desired acidity. After aging, the milk is heated and then shaped and pressed into cylindrical wheels. The wheels are then immersed in a salt bath and placed on spruce boards where they mature for at least 75 days. In the final stage of production, experts perform tests on the quality of the cheese and its compliance with the AOP specifications [7].

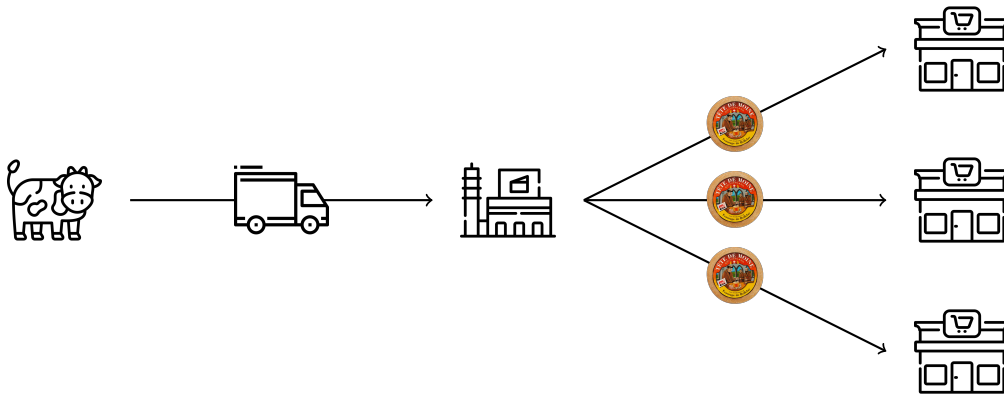


Figure 2.1: Simplified value chain of Tête-de-Moine AOP cheese [6].

2.2 Blockchain Technology

BC is a peer-to-peer (P2P) network of nodes that validates the chronological order of transactions or events. This data is stored in a decentralized, transparent, and immutable manner [2], [4]. All of these characteristics make BC a powerful technology for supply chain traceability.

2.2.1 Architecture

A BC, simply put, is, as the name suggests, a chain of blocks. Each block consists of two main parts: the header and the body. The header contains several parts, which will be discussed in more detail below. The body contains all the transactions of the block and a transaction counter (see Figure 2.2). Blocks are linked together by their hash values, with each block containing the hash of its parent block (see Figure 2.3), the only exception being the first block (or genesis block) [4], [8].

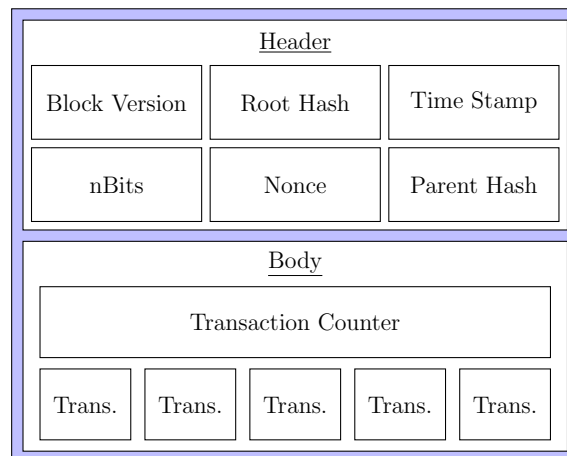


Figure 2.2: Single Block Architecture [4].

The following is a brief description of the different parts that make up the header [2], [4]:

- **Block Version** → comprises the rules for validating new blocks.
- **Merkle Tree Root Hash** → is the hash of the transactions in the block. Transactions are paired until only a single hash, *i.e.*, the root hash, remains (see Figure 2.4).
- **Time Stamp** → is the current time in seconds according to Universal Time since January 1, 1970.
- **nBits** → represents the target threshold of a valid root hash.
- **Nonce** → is a unique random number used as an authentication protocol.
- **parent hash** → is a 256-bit hash value pointing to the parent block.

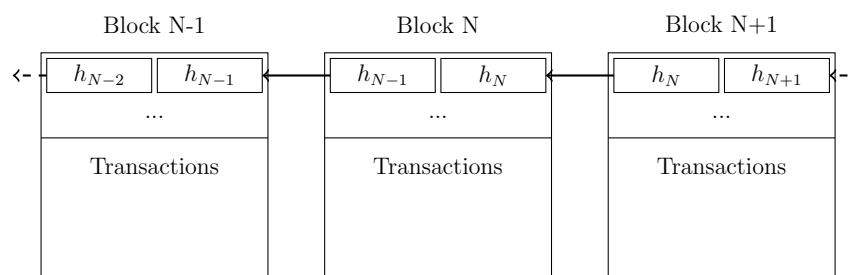


Figure 2.3: Block Chain Data Structure [2], [8].

Figure 2.3 illustrates how the different blocks within a BC are connected. The letter h stands for the root hash and the number (*i.e.*, $N + 1, N, N - 1, N - 2$) indicates which block this root hash corresponds to. For example, in block N , h_N corresponds to the current block, *i.e.*, block N , while h_{N-1} corresponds to its parent block, *i.e.*, block $N-1$. In block N , h_{N-1} is a link to the parent block, *i.e.*, block $N-1$, while in block $N-1$ it is the root hash.

2.2.2 Security Aspects

The generally accepted order of blocks is based on distributed consensus, *i.e.*, whatever the majority of nodes agree on is accepted [2]. There are many security aspects to BC. Below are brief descriptions of the most important ones [8].

Hashes are one-way mathematical functions used to ensure data integrity. This means that you cannot derive the original data from a hash value.

Hash chains build on this concept and use previous hash values to calculate new ones, *i.e.*, h_1 (hash value one) is used to calculate h_2 , h_2 is used for h_3 , and so on. The nature of one-way functions ensures that previous hash values cannot be inferred from newer ones.

Merkle trees are used in BCs to form the root hash of each block. All transactions in the block are hashed, and the resulting hash values are recursively grouped in pairs until only a single hash (the root hash) remains. Figure 2.4 illustrates how transactions are hashed and grouped in a Merkle tree to form the root hash.

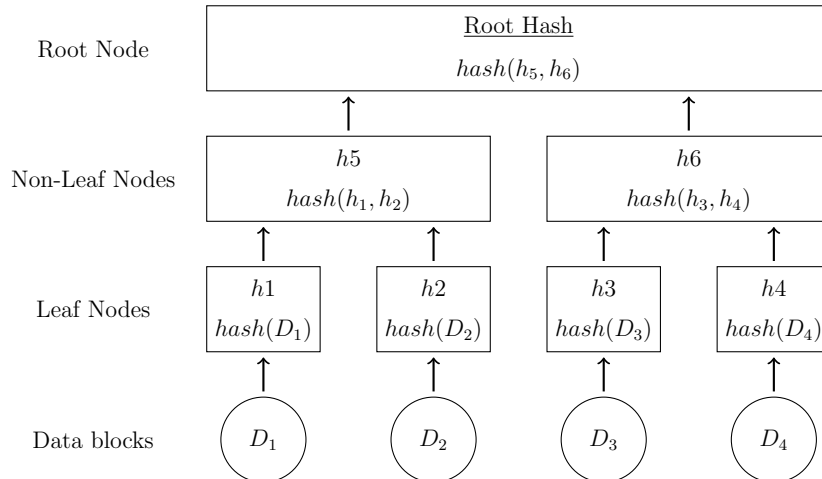


Figure 2.4: Merkle Tree Example [8], [9].

2.2.3 Consensus Mechanisms

There are a number of different consensus mechanisms in BC technologies. The following is a brief description of two important ones.

Proof of Work (PoW) is a consensus mechanism based on a verifying party and a proving party, where the proving party attempts to prove to the verifying party that it has expended some computational effort in a specified time interval [10]. This mechanism is most commonly used in bitcoin to calculate new blocks, *i.e.*, transactions [3]. Miners have decision-making power in direct proportion to the computing power they possess [11].

Proof of Stake (PoS) requires much less or no computing power, relying instead on the stakes of the network's own coins [8]. Validators are selected based on different mechanisms depending on the BC. The idea is that a potential attacker must have a stake in the system and would undermine the value of their own coins in order to carry out an attack. Similar to PoW, the stakeholders' decision-making power is directly proportional to the number of coins they own [11].

Proof of Authority (PoA) is a consensus mechanism that is popular in private BC networks because it offers several advantages such as performance, efficiency, and verified participants. There are several implementations of the algorithm, including the Clique algorithm by Geth [12].

51% Attacks occur when a malicious party owns more than 50% of the network's computing power (PoW) or more than 50% of the coins (PoS) and thus controls the consensus [8].

2.3 Blockchain Types

The three main types of BCs are public (or "permissionless"), private (or "permissioned"), and consortium [9]. The following is a brief overview of the different types and their respective advantages and disadvantages.

Public Blockchains are accessible to everyone, both in terms of visibility and usability. Anyone can join the network and see all the information stored within the chain. Public BCs are decentralized and don't require third-party participation. The more nodes that participate in a peer-to-peer network, the more secure and decentralized it naturally becomes, but this can lead to higher energy consumption, slower speed, and scalability issues [8], [13].

Private blockchains operate on an invitation basis with respect to write permissions; only a single organization decides who can join the network, and the network itself is still decentralized. Read permissions can be public or reserved for participants in the network. Since only known nodes can join the network, security is increased in exchange for privacy within the network. Private BCs, consisting of smaller networks, are less decentralized by nature, but typically offer lower or no costs, greater efficiency through faster transactions, and privacy outside the network [2], [8].

Consortium blockchains are in the middle. Like private BCs, only "permissioned" nodes have access to the network, but the network can span multiple organizations [8].

2.4 Smart Contracts

Another technology well suited and widely used for supply chain traceability is SCs. Today's SCs are distributed, self-executing, and autonomous contracts that are immutable and irreversible due to their association with blockchain technology [14].

2.4.1 Origin

SCs were first proposed in 1996 in [15]. With the rise of multinational corporations, the need to conduct international business across many jurisdictions became more prevalent. SCs are digitized contracts with specified terms and conditions that the parties agree to abide by. Four characteristics of contracts have been highlighted:

- Observability → Each party knows the performance of the other party/parties.
- Verifiability → Each party can confirm compliance or noncompliance.
- Privity → Each party knows only as much information about the other party/parties as is necessary to perform the contract.
- Enforceability → the contract is enforceable due to "[r]eputation, built-in incentives, 'self-enforcing' protocols, and verifiability".

In particular, the last feature of enforceability was thought to often require the involvement of third parties, even when using SCs.

2.4.2 SCs and BC

With the emergence of BC technologies in the late 2000s, SCs have received increasing attention and are often at the core of newer BC technologies such as Ethereum. Today's SCs can be established between anonymous or untrusted parties without the need for a third party. The parties specify conditions that, if met, result in pre-defined actions. Such conditions include the fulfillment of the contract itself, which can be action- or time-bound, or contract violations, among others. The three main characteristics of SCs are [14]:

- **Autonomy** → Once deployed on the BC, no further action is required by the parties involved.
- **Self-sufficiency** → SCs can access resources as needed.
- **Decentralized** → SCs are distributed among the nodes of the network.

Today's SCs, as implemented in and used with BC technologies, include all the key characteristics of contracts as described in [15]. However, SCs still face challenges because they have certain vulnerabilities that can be exploited by attackers. One such vulnerability concerns timestamp-dependent contracts, where the miner could change the timestamp and thus change the intended outcome of the contract. This example highlights the dependence of SCs on accurate information from external sources. SCs typically lack mechanisms for verifying the accuracy of this information [14]. Other challenges to the development of SCs were identified by [16]. The top challenge identified was security concerns, followed by debugging difficulties, lack of general purpose libraries, and more.

2.5 Ethereum

In 2014, Vitalik Buterin published [17], the white paper for Ethereum. The idea was to provide an alternative BC to bitcoin with improved capabilities. One major improvement was that Ethereum offered its own Turing-complete programming language called Solidity. This opened the door to writing SCs and storing them on a BC. The Ethereum BC comes with its own cryptocurrency called Ether, which is used to cover transaction fees. Initially, three types of applications were envisioned to run on top of Ethereum: financial applications, semi-financial applications, and non-financial applications such as online voting and decentralized governance [17]. Ethereum uses PoS as its consensus mechanism after a transition from PoW in September 2022 [18]. Similar to bitcoin, block mining rewards were continuously lowered when PoW was still in place [18]. With PoS, the rewards are now proportional to the number of validators and their respective stake [19].

2.5.1 Pros and Cons

Apart from the inherent advantages of BC technology, there are some specific advantages of Ethereum. A wide range of decentralized applications (dApps) are hosted on the Ethereum chain, covering topics from finance to NFTs to gaming [20]. Therefore, the Ethereum ecosystem is already tested, used and documented. There are many test networks like Truffle, Ganache or Hardhat that allow developers to develop and test their applications before deploying them on the main network.

Unfortunately, there are also challenges. One such challenge is the scalability of the ecosystem, which is a barrier to widespread adoption. As noted in [21], transaction speed depends much more on block size than on the number of miners. As noted in [17], larger block sizes would compromise decentralization, as only validators with huge amounts of storage capacity could store the entire BC and participate as a full node. Another challenge lies in the nature of SCs, once deployed on the main network they are immutable and defects cannot be fixed [22].

For companies and individuals who prefer to host their own private network, Ethereum offers the use of its technology to create private networks [23].

2.6 Containerization

In the past, developers installed applications and systems locally or in VMs to run them. This introduced a number of difficulties because the underlying operating system (OS), such as Linux, Mac OS, or Windows, could behave differently. Sharing software between different environments often resulted in bugs and errors. Enter containerization. Containerization is the process of packaging software applications, or code in general, into containers. This process makes it easier to distribute software across environments. Containers package the code, along with its configuration files and dependencies, into a single, self-contained software package. The templates for the containers are called images [24].

Figure 2.5 shows two different scenarios where software is shared between two developers. Developer A is programming in Mac OS, while Developer B is using Windows. On the left side we see a traditional approach where Developer A programs on his system and later tries to share his code with Developer B who needs to implement a new feature for the software. However, when Developer B tries to run the code, she discovers that it is not compatible with her OS. As a result, she must debug the code before she can start coding. Contrast this with the second scenario on the right. Here, Developer A creates a container of the software before sharing it. The container contains all the necessary dependencies, code, and configuration files, making it a complete system. This allows Developer B to simply install and run the container on her machine and immediately start coding the new feature.

One widely used technology for containerization is Docker. Written in the Go programming language, Docker was originally developed for the Linux kernel, but was later adapted to be compatible with Mac OS and Windows. Two important objects to mention

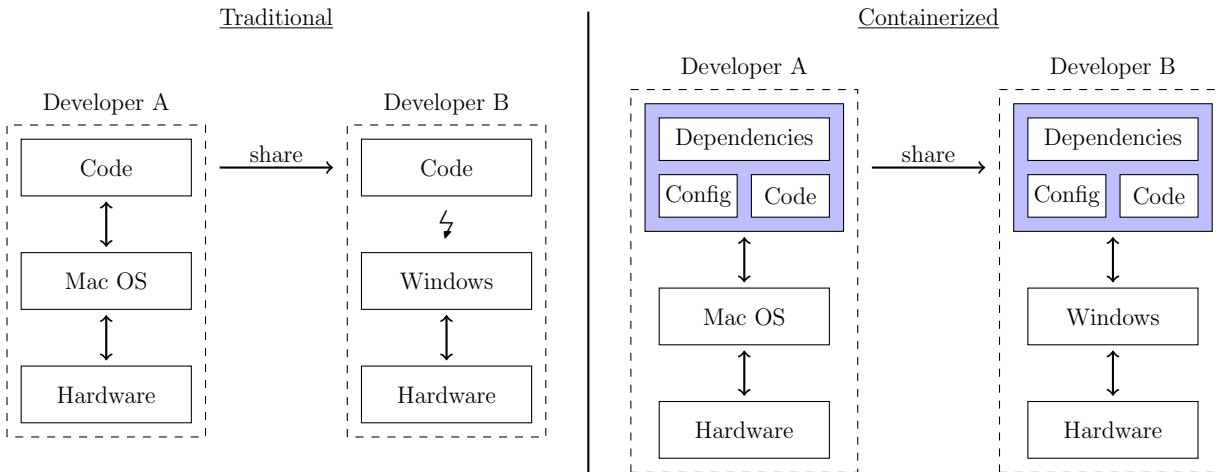


Figure 2.5: Traditional vs. Containerized Applications.

are images and containers. Images are templates that contain instructions for creating containers. A container is a runnable instance of an image, largely isolated from the host machine and other containers. For larger projects, it makes sense to use containers, one for each component, which is where Docker Compose comes in. Docker Compose allows you to manage multiple containers from a single file [25].

2.7 The CheeseChain Project

The CheeseChain project aims to promote trust and transparency along the value chain of Tête-de-Moine AOP cheese, through DNA-based intrinsic product authentication combined with automated fraud detection. To achieve this goal, BC technology will be combined with biological data. The use of BC will provide the basis for a tamper-proof and immutable audit trail. This approach will facilitate sampling by brand owners and simplify access for authorities, distributors and even consumers. Partners in this project are FROMARTE, Agroscope, Tête-de-Moine AOP producers and the University of Zurich [1], [26].

Chapter 3

Related Work

The purpose of this chapter is to summarize and analyze work focused on the use of BC technology in the area of dairy supply chain tracking. It is divided into three parts. The first part is a brief description of the methodology used to identify relevant works. This is followed by the summaries and analyses of the identified works, which make up the second part. The chapter ends with a summary of the findings. The goal is to provide an overview of different solutions to related problems.

3.1 Methodology

The primary search engine used to find the papers summarized above was IEEE Xplore. The initial queries are listed below, along with the number of results they yielded. After these initial prompts, the resulting papers were briefly reviewed for eligibility. The primary criteria for suitability were the use of BC technology and the tracking of the supply chain of a product (ideally dairy-based, or at least food-based). For each paper identified, we elicited information about its purpose, implementation, scope of testing, and deployment status.

Initial prompts for the paper search:

1. "supply chain tracking with blockchain" → 489 results
2. "food supply chain tracking with blockchain" → 136 results
3. "cheese supply chain tracking with blockchain" → 1 results
4. "dairy product supply chain tracking with blockchain" → 5 results

Searches three and four returned few enough results to analyze the papers for eligibility. The following papers were either found based on other but unrecorded prompts, or were referenced in the papers already analyzed.

3.2 Blockchain Technology in Supply Chain Tracking

This section aims to provide an overview of the work that has already been done on supply chain tracking using BC technology in the dairy sector. This includes research, implementation and, in most cases, deployment of solutions to this problem. The first part is an analysis of the work that deals primarily with the supply chain of milk itself. The second part consists of the works that focused on the supply chain of different types of cheese. The last part provides a brief description of the two theses that preceded this one.

3.2.1 Dairy Supply Chain Tracking

The first paper examined, [27], combined an industrial analysis with a social study to deliver a distributed application (dApp) called NUTRIA. Using BC technology, NUTRIA aims to track dairy products in Switzerland from farm to consumer. To test the usability of their application, they conducted a survey with nearly 200 participants who reflected potential end users. In addition, they interviewed 20 people involved in different stages of the value chain, such as feed and milk producers, transport companies, and wholesalers. They used Ethereum as their layer one BC and issued a unique QR code and a new SC for each product produced and step taken along the value chain, resulting in multiple QR codes and SCs per product. The different actors along the same value chain generated new QR codes on top of older ones, thus integrating them. For example, the feed producer (actor 1) generates QR code 1, and the milk producer (actor 2) generates QR code 2 on top of QR code 1. Consumers were able to track each step of the product along the entire value chain. The authors present evidence for the successful application of BC technology in food supply chains.

A second paper dealing with tracking issues in the dairy supply chain is [28]. The focus was on tracking the dairy supply chain in Sri Lanka. One of the issues was to enable temperature monitoring of milk during transport. This is a critical part of the supply chain because milk can go bad quickly if it is at the wrong temperature. They used an Internet of Things (IoT) device to measure the temperature of the milk. In addition, they deployed a SC on the Ethereum network that alerts the driver when the temperature reaches a worrisome level. An additional functionality of their application was to help identify cow diseases using image processing.

The authors of [29] used Hyperledger Fabric as their BC technology to tackle the problem of shipping dairy products, especially milk, in a unique way. The idea was to publish a SC to the BC that contains an upper and lower limit for the temperature of the milk being transported. The link between the SC and the milk was a sensor that measured the temperature of the milk. If the temperature violated one of the conditions in the contract, a penalty would be issued. For each item, a new SC was issued with details of the penalty and payment. The service was provided through a mobile application.

The first three papers discussed have all deployed their systems. However, although the use case of milk production is related, cheese production may have additional details and

trust assumptions that differ from this scenario. Therefore, it is necessary to explore the deployment of BC-based supply chain tracking in the cheese production scenario.

Using an Ethereum-based service, [30] proposed a system to increase the traceability and transparency of supply chains for customers. This addresses the immense problem of traceability in the food supply chain, particularly that of milk and other dairy products. Dairy products, especially milk, are very susceptible to going bad and failure to meet safety standards. The motivation behind their proposed solution was to provide publicly available information about the product and reduce the search time from days to seconds. This would increase both transparency and accessibility for customers, allowing them to make better purchasing decisions. Because they had not deployed their application at the time of publication, the authors left some questions unanswered, such as how their prototype would handle real-world data and whether the demand for their proposed solution actually existed.

While the previous paper focused on making information publicly available, [31] proposed using BC technology in combination with SCs to automate processes such as quality control, product tracking, and reverse logistics, all aimed at tracking dairy products along their supply chain. Each user of the SC is assigned one of three roles (supplier, retailer, or customer), and depending on that role, the user has access to specific functionality. This ensures, for example, that only a user with the supplier role can add new products to the supply chain. The authors intended to deploy their contract on Ethereum. They proposed a proof of concept and have not deployed their application for real-world use. At the time of publication, they have only deployed it on the Ganache and Rinkeby test environments. It is difficult to evaluate the true cost, performance, and benefits of their application without deploying it in a real-world environment.

Similar to the first three papers, [30], [31] also proposed systems focused on tracking dairy products along their supply chain. However, as their solutions have not been deployed, they highlight a research gap in the need to deploy and test such solutions in a real-world scenario.

3.2.2 Cheese Supply Chain Traceability

To improve the supply chain of Fontina PDO cheese, an Italian dairy product, the authors of [32] proposed a solution to standardize the recording of information by operators and provide consistent information to customers. The value chain includes five stages: milk production, processing, seasoning, packaging, and distribution. Each operator records data in a relational database management system (RDBMS), and key fields for identifying transactions are then stored in the BC. They used MySQL for their RDBMS and Algorand as their BC technology. Each phase generates a new table with an ID that is stored in the table of the next phase, thus linking them. All the data is stored in the BC when the packaging phase is completed. The information is made available to customers through a QR code.

The authors of [33] aimed to improve on previous food supply chain tracking solutions by developing a system that would be interoperable, autonomous, and functional. They

also integrated a backend data sharing model to improve efficiency. To achieve their goal, they used BC technology and developed three different SCs. The first SC managed the traceability of raw materials and the resulting products. The second SC managed the list of stakeholders and their interactions with the products. The third SC helps to facilitate management tasks related to processes. Participants could modify the content according to their role. Public users, *e.g.*, a customer, have access to read-only functions such as basic qualitative information, which should promote trust. The authors used a private BC, but recognized the value of a hybrid approach.

Another paper dealing with the supply chain of Greek cheese is [34]. The authors looked at the value chain of feta cheese from milk production to the finished product. The goal was to provide better traceability along the value chain and greater transparency regarding the quality and origin of ingredients and the final product through the use of QR codes, BC technology, and SCs. The authors conducted a study to gain insight into what consumers ultimately consider to be the most relevant information. The four stages of cheese production are milk collection, start of production, ripening, and packaging. Different actors along the value chain perform checks on the product and register this information in the BC. The consumer can then access the registered information through a QR code printed on the product's packaging. Each QR code corresponds to a specific production batch. The complete data was stored on a private BC (Quorum), and then a hash based on the complete data was stored on a public BC (Ethereum). SCs were implemented to verify compliance with the established rules and to register the information on the public and private BCs. With this approach they tried to get the best out of both technologies. Consumers would get access to the data on the private BC (through the QR code) and could then verify the integrity of the data through the hash on the public BC.

The last three papers focused on traceability in the cheese supply chain. [32], [33] both focused on private BCs, thus highlighting the research gap for using a public BC or a hybrid approach. [34], although using a hybrid approach, solved the problem with a different architecture, the distinction is discussed further in section 3.3.

3.2.3 Previous Theses

The following is a review of the previous two undergraduate theses that laid the foundation for this thesis. [35] focused on the private aspect of the CheeseChain project with a database-to-BC data collection system. FROMARTE hosts a DigitalQM that allows value chain actors to collect their data in forms. The [35] solution retrieves new data from the DigitalQM as it is stored and updates or creates forms on the BC. Each update or change to the form on the DigitalQM results in a new file on the BC. The different versions of the file on the BC are unidirectionally linked by the name (a string) of the previous version. To ensure data integrity, hashes of the data are stored in the files, similar to the file names. The solution of [35] uses a SC to allow communication with the BC.

While [35] dealt with the private part of the system, [36] was concerned with implementing a SC to track the supply chain of Tête-de-Moin cheese. The system was designed to track the production history by storing different information along the value chain. While everyone has read access, each participant was additionally granted a certain level of write

access depending on the role they played in the value chain, *e.g.*, a milk producer could add information about a batch of milk. Roles were assigned only by an administrator. The system consisted of three parts: an on-chain, an off-chain, and an external part. The thesis was mainly concerned with the three parts that made up the off-chain part of the system. The three parts were a front-end, an Application Programming Interface (API) layer to connect the private BC to the public, and a SC connector to enable this connection. Through the front-end, consumers can retrieve data about products. Since this system has not yet been deployed, only a proposal for a public BC exists. The thesis evaluates whether using Ethereum alone or in combination with Polygon is better and concludes that the combination with Polygon is more cost and time efficient.

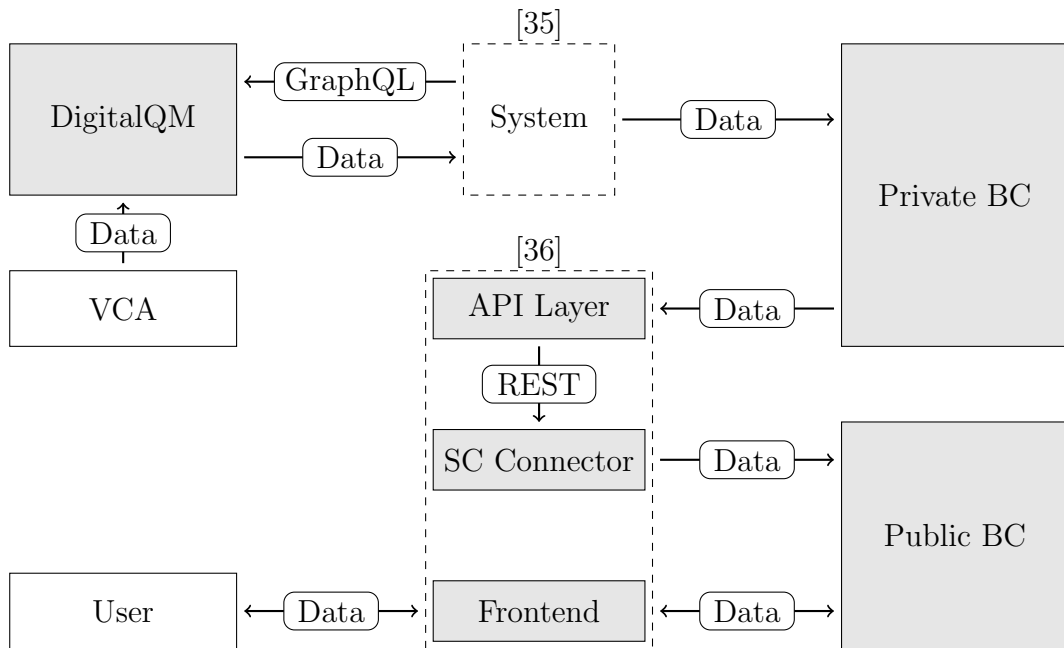


Figure 3.1: Overview of the CheeseChain project architecture [35], [36]

Figure 3.1 is a high-level illustration of how [35] and [36] will work. The DigitalQM acts as a database to which the various value chain actors (VCA) can upload their data. The system from [35] will then automatically upload selected data to the private BC via a SC. From there, the data reaches the public BC via the API layer through the SC connector of the system from [36]. Users¹ can access the data stored in the public BC through the **Frontend**, and certain users can also upload data to the public BC through the **Frontend**. Although not shown in the figure, they can also use the API Layer to access the data, which is a secondary use case.

Although these two theses provided solutions for the same project as this thesis, they did not deploy their systems, highlighting the need to deploy their systems and investigate how their solutions perform in a real-world scenario.

¹Users in this context refers to both consumers and participants in the CheeseChain project, such as milk producers and lab workers.

3.3 Summary

Considerable work has been done in the area of supply chain tracking using BC technology. Various solutions and approaches have been proposed, implemented, and in most cases deployed. This section briefly summarizes the analyses performed in section 3.2. Additionally, this section identifies a research gap in the previous work according to five criteria. The goal of this thesis is to fill this gap.

Examining the first criterion, the use of SCs, one work can be identified that has managed to create a supply chain tracking application using BC technology without SCs [32]. This work highlights the need to investigate the use of SCs for such a system. Looking at the second column, the works that have not implemented their system can be identified [30], [31], [35], [36]. However, the deployment of the system is a critical part, as real-world scenarios very often provide new insights into the potential shortcomings of such a system. Different BC technologies have different kinds of strengths and weaknesses, so the BC technology used is another differentiating criterion. There are two papers that don't use Ethereum, [29], [32]. Using different types of BC networks, such as private, public, consortium, or hybrid, again offers different trade-offs. Looking at the fourth criterion, exactly one other work can be singled out that has used a hybrid approach, using both a private and a public network, which is [34].

Having identified the work most similar to the current thesis (*cf.*, [34]), a closer look at the underlying use case of both works is necessary. [34] uses the private BC to access the data, while this thesis will provide data retrieval from the public BC via the **Frontend**. Second, [34] stores all the data in its private BC, while this thesis will store only parts of the data in the BC, the rest remaining exclusively in the DigitalQM. Finally, [34] creates a hash of all the data stored in the private BC and stores it in the public BC. This thesis, on the other hand, allows the actual supply chain-related data to be stored in the public BC.

Table 3.1 provides an overview of the works and the characteristics of their respective solutions according to the five most important criteria regarding this thesis. The five criteria are whether SCs were used, whether the system was deployed, what BC technology was used, what type of network was used, and last but not least, what the underlying use case was.

Table 3.1: Overview of Related Work

Work	SC	Deployed	Blockchain	Type	Use Case
[27]	✓	✓	Ethereum	Public	dairy
[28]	✓	✓	Ethereum	Public	dairy
[29]	✓	✓	Hyperledger	Consortium	dairy
[30]	✓	×	Ethereum	Private	dairy
[31]	✓	×	Ethereum	Public	dairy
[32]	×	✓	Algorand	Private	cheese
[33]	✓	✓	Ethereum	Private	cheese
[34]	✓	✓	Ethereum & Quorum	Hybrid	cheese
[35]	✓	×	Ethereum	Private	cheese
[36]	✓	×	Ethereum	Public	cheese
<i>This thesis</i>	✓	✓	Ethereum	Hybrid	cheese

Chapter 4

Architecture and Design

The goal of this chapter is first to highlight the use case of this work by describing the application scenario. The second goal is to summarize the current state of the art by describing the functionalities and potential shortcomings of the systems. The chapter concludes with a proposed scenario for the use of both systems. For simplicity, the system from [35] will be referred to as System 1, and the system from [36] will be referred to as System 2. System 2. The first part focuses on System 1, the second on System 2, and the third on the interplay between the two.

4.1 Application Scenarios

This section presents and explains an application scenario for each of the two systems to provide a better understanding of their respective use cases.

4.1.1 Application Scenario of System 1

FROMARTE, the umbrella organization of Swiss cheese specialists, aims to preserve Switzerland's unique cheese culture and diversity. FROMARTE achieves its goal by creating optimal conditions for its members and representing their interests in politics and on the market. In line with this, they offer their DigitalQM to store production and quality data. With regard to System 1, the stored data focuses on the supply chain of Tête-de-Moine cheese. Organizations involved in the production of Tête-de-Moine cheese, that are also members of the DigitalQM, upload relevant data. System 1 then periodically and automatically makes queries to the DigitalQM. When new forms are created or existing forms are updated, System 1 either stores the new form in the BC or updates the data of an existing form.

Figure 4.1 shows a high-level data flow in System 1. For the sake of illustration, assume that the example is about the supply chain of Tête-de-Moine cheese, but in a simplified format. First, a VCA uploads data to the DigitalQM (step 1); in this scenario, let's

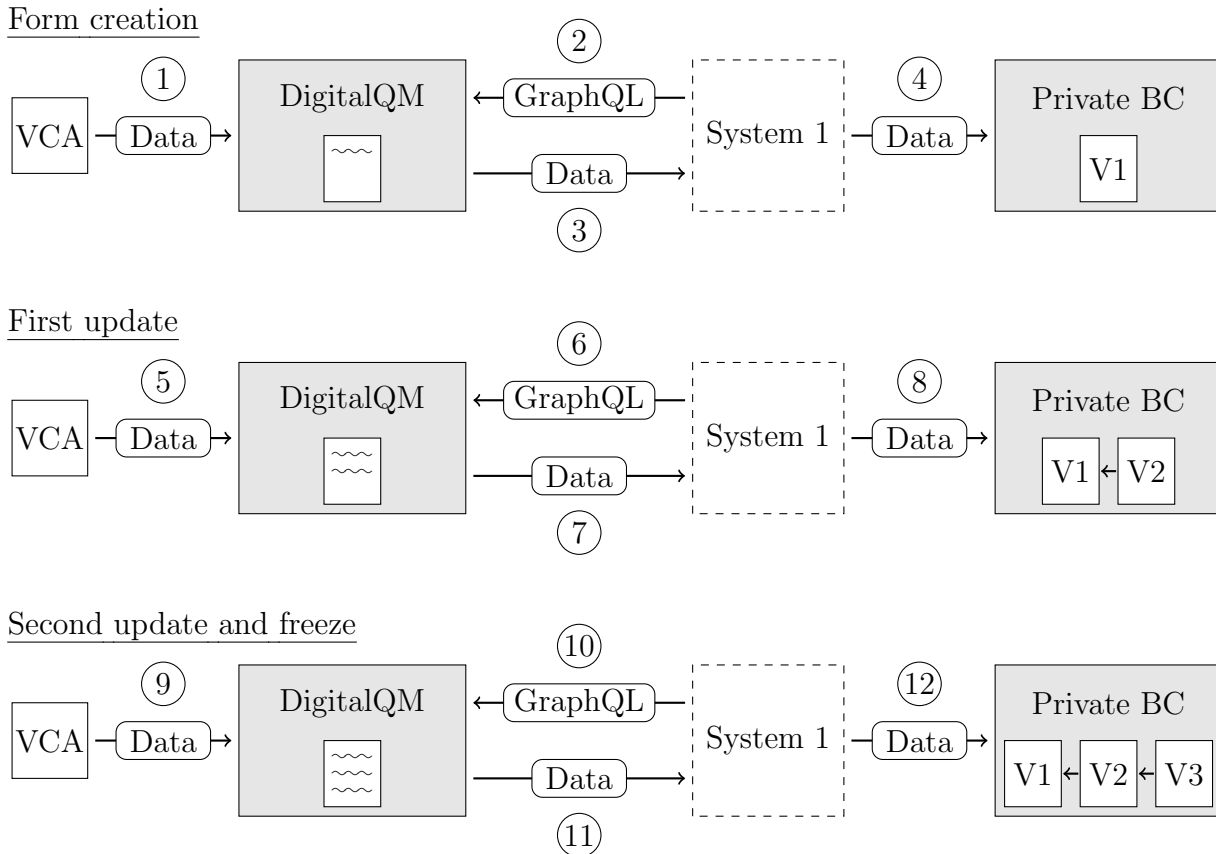


Figure 4.1: Application Scenario System 1

assume the VCA is a milk producer from the Swiss Jura mountain region. A new form is created in the DigitalQM. System 1 detects the new form and retrieves the data using a GraphQL query (steps 2 and 3). The fourth step is to upload the data to the private BC by creating a new form on the SC (shown as V1).

After production, the milk must be transported to a cheese producer. The transporter records relevant data in the DigitalQM (step 5). New data is added to the existing form in the DigitalQM. System 1 notices the change and retrieves the new data via another GraphQL query (steps 6 and 7). In the final step, the new data is automatically stored in the private BC by creating a new form on the SC (V2). The new form is linked to its parent by storing the name of the parent. This creates a unidirectional linked list, linking new forms to the second newest version. This allows the file history to be traced backwards. So the two forms, V1 and V2, are linked by V2 storing the name of V1. The fact that V1 does not store a name indicates that it is the genesis form or the beginning of that particular file. Linking the files allows for version control and provides insight into the change history of the files. An additional benefit is that the file name is a hash based on its data, so tampering with the data will result in a new hash, making it easier to detect fraud.

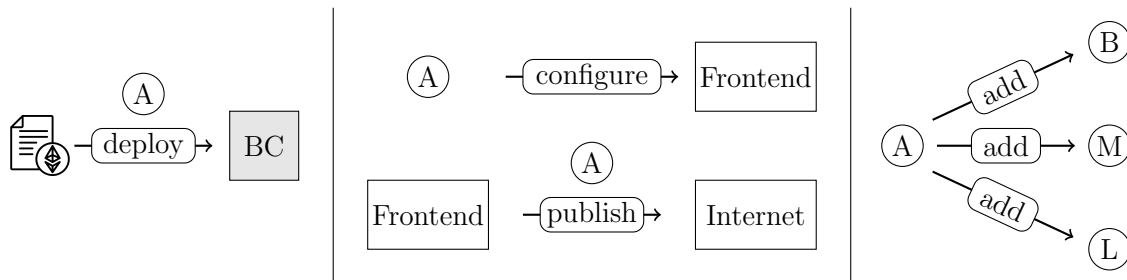
The milk has now arrived at a cheese factory and is ready for further processing. The cheese producer adds production-related data to the Digital QMS. For simplicity, let's

assume that the cheese is simultaneously inspected for compliance. Since this is the last step, the form is frozen after the update (step 9). System 1 notices the change and freezing of the form and retrieves the data (steps 10 and 11). The final step is to create a new form on the SC (V3), associate it with the now second newest version, *i.e.*, V2, and mark it as frozen.

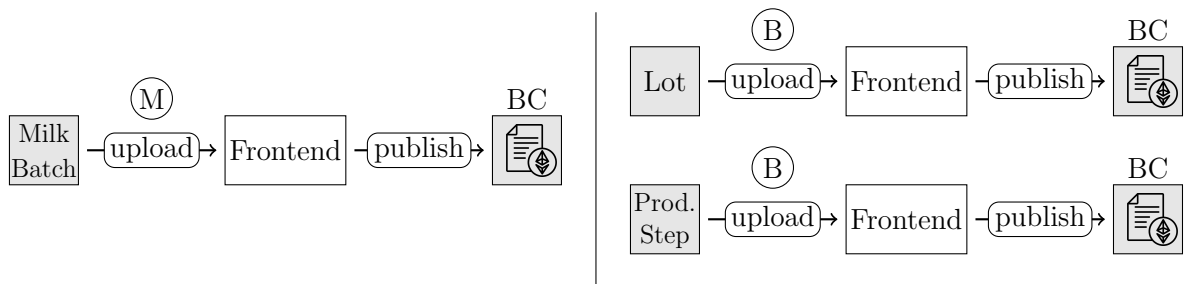
4.1.2 Application Scenario of System 2

System 2 can be used through both the **Server** and through the **Frontend**, but in this application scenario, the focus is on the **Frontend** part of System 2. The author acknowledges the importance of the **Server**, but since its primary use case is to automatically store data from the private BC into the public BC, and its secondary use case is only suitable for technical users, he has decided not to describe a detailed application scenario.

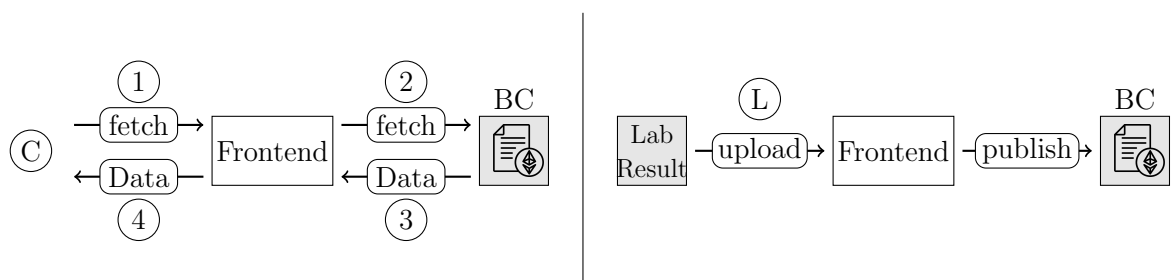
System Setup



Production



Data Retrieval and Testing



A = Administrator, B = Basic, C = Consumer, L = Laboratory, M = Milk Producer

Figure 4.2: Application Scenario System 2

Figure 4.2 shows a scenario using System 2, which is meant as an illustration and simplifies the whole process, leaving out many important details and steps. The scenario consists of 3 distinct phases. The first phase focuses on system setup, the second phase focuses on cheese production, and the last phase focuses on a customer and a lab worker.

The first phase focuses on system setup and consists of three steps. The first step is to deploy the **SC** to a **BC**. The person who deploys the **SC** automatically becomes the sole administrator (shown as a circle with an **A**). After deploying the contract, the administrator has to modify the deployment environment of the **Frontend** so that it connects to the correct **SC**. With the configuration in place, the administrator deploys the **Frontend**. The final step in the first phase is to provide access to the participants. The administrator can add new participants to the system and assign roles to them. In the scenario, three participants are added with different roles. One participant has the role **Basic**, another has the role **MilkProducer**, and the third has the role **Laboratory** role. This allows a milk producer, a cheese producer, and a lab worker to join the project and store their data on the **BC**.

A milk producer with the **MilkProducer** role now has access to the running system and can add production-related data to the **SC**, such as where the milk was produced and in what quantities. After production, the milk needs to be transported to the cheese producer. The producer can get information about the milk through the **Frontend**. At the beginning of the production, the cheese producer, who is given the **Basic** role, adds a new lot, which will later be used to identify the cheese. During production, the cheese producer can continuously add new data for each step. When the cheese is finished, it is shipped to stores around the world.

In the final phase, a curious consumer who buys one of the newly produced cheeses can view the production data by accessing the **Frontend**. By default, everyone has the **ViewOnly** role and can access the production data using the lot number. The consumer then has access to the production history and can see where the milk came from and how the cheese was made. A lab worker also buys a cheese and performs a routine test to determine whether the cheese is an original or a counterfeit. After the test, the lab worker can upload the result to the **SC** using the **Frontend**.

4.2 Technical View

The purpose of this section is to provide an in-depth technical analysis of the current state of the two systems by describing their main functions and components.

4.2.1 Technical View of System 1

The following is a technical analysis of how System 1 actually works. The four **MAIN** components are three Python scripts and one **SC**. They provide the desired functionality through a complicated interplay of functions.

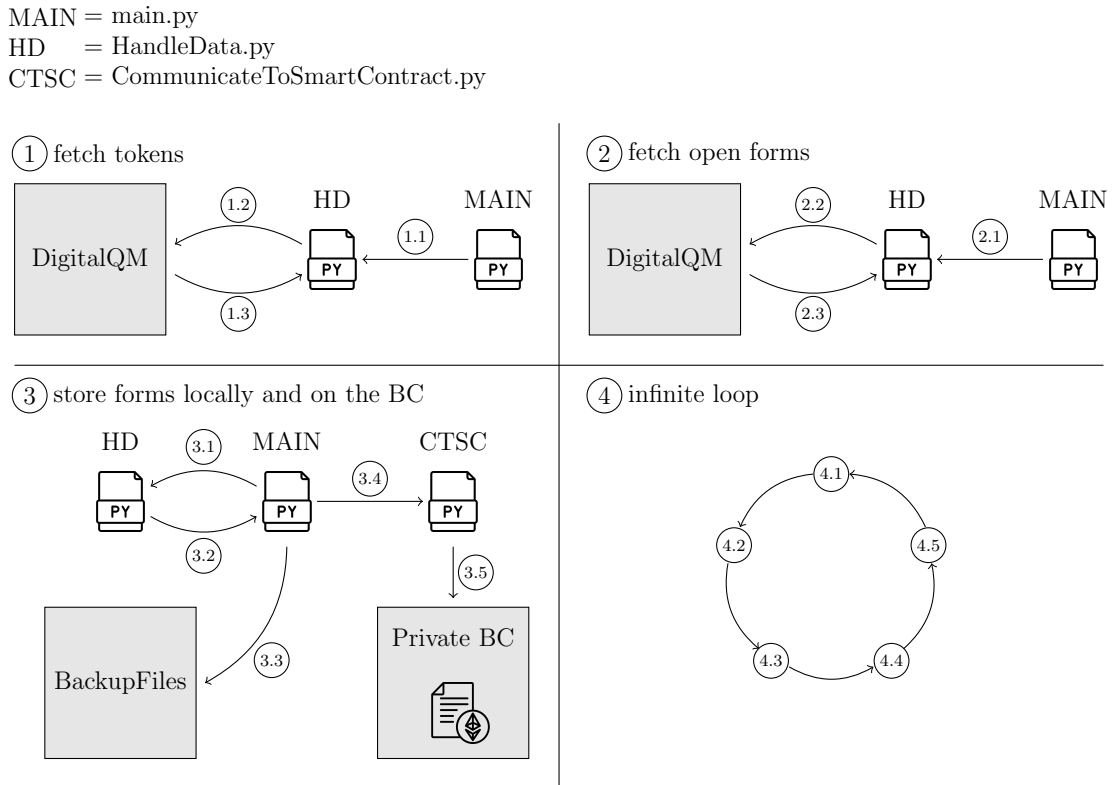


Figure 4.3: Technical illustration of System 1

At the start, it is necessary to manually fetch a refresh token from the FROMARTE website, store it in the config.json file, and deploy the SC to the BC. Figure 4.3 illustrates what happens next. System 1 starts with the main.py (MAIN) file. In the first step, it retrieves the authentication token along with a new refresh token from the DigitalQM, the original refresh token is needed for this step. It uses the HandleData.py (HD) file to communicate with DigitalQM. HD is a class, and at the beginning it creates an instance of the class, which now stores both tokens. So technically there is now communication from the HD file back to the MAIN file, and the MAIN file gives the order for HD to fetch and store the two tokens.

At a high level, the second step looks identical to the first, as the components involved are the MAIN file, the HD file, and the DigitalQM. However, there are different functions involved in HD. In this step, MAIN tells HD to fetch all open forms from DigitalQM and store them locally as JavaScript Object Notation (JSON) files in the instance of the class.

The third step involves additional components. First, MAIN retrieves the stored JSON files from the instance of HD (3.1 and 3.2). Next, MAIN stores the JSON files in a BackupFiles folder and simultaneously stores the file contents on the deployed SC using the CommunicateToSmartContract.py (CTSC) file functionality (steps 3.3 through 3.5). Steps 3.3 through 3.5 look sequential in the sense that once step 3.4 is started, step 3.3 is not executed again, as are steps 3.1 and 3.2. The difference is that they are sequential only within a larger sequence. That is, for each file retrieved from the disk instance, steps 3.3 through 3.5 are performed.

This completes the process of updating the system and storing the open forms on the BC. The fourth and final step is to keep the information stored on the private BC up to date with the information stored in the DigitalQM. To accomplish this, step four is an infinite loop. In this infinite loop, the previous three steps are repeated indefinitely with minor adjustments. First, a new refresh authentication token pair must be fetched from the DigitalQM via the HD file (step 4.1), which corresponds to step 1. Second, the forms are re-fetched and checked for any updates or form freezes. In addition, any changes are automatically stored in the BC. This step (4.2) is equivalent to steps 2 and 3. In step 4.3, the DigitalQM is checked for new files, which corresponds to step 2. The next step is 4.4, which stores any new form on the BC, which is the same as step 3. Finally, there is a ten second time buffer before the cycle starts again.

4.2.2 Technical View of System 2

The following is a technical description of System 2. On a high level, the system consists of three parts, the **SC**, the **Server** and the **Frontend**, each with its own functionality.

The **SC** is written in Solidity and is responsible for storing and retrieving data on the BC. There is one administrator, the person who deployed the contract, the other roles are **ViewOnly**, **Basic**, **Laboratory**, and **MilkProducer**, each with its own set of permissions. The roles are stored as Enums and represented internally as numbers from 0 to 3. Each participant has to be manually added and removed by the administrator using the `addParticipant` and `removeParticipant` functions. A participant's role can also only be changed by the administrator using the `changeParticipant` function. Function modifiers such as `onlyAdministrator`, `onlyBasicParticipant`, `onlyLaboratory` or `onlyMilkProducer` ensure that only participants with that particular role are granted access. The following is a description of additional functions of the SC and the role required to execute these functions. The administrator is not mentioned again as he or she can execute all these functions. The `addMilkBatch` function can be used by participants with the role **MilkProducer** to add information about milk production. Participants with the role **Basic** have access to the functions `addLot` and `addStep`. Together, these functions provide the functionality to store information about the production of a production lot of cheese. The `addLabResult` function is used by the **Laboratory** participants to store data about the laboratory results of a lot. The `addLabResult` function stores only the lot number and a boolean that is true if the bacterial colonies were a match and false if they weren't.

The **Server** consists of two parts, the **API Layer** and the **Smart Contract Connector (SCC)**. The principal task of the **Server** is to provide an interface between the private part (System 1) and the public part (System 2) of the CheeseChain project. Additionally, the **API Layer** exposes all the endpoints of the SC, allowing for simplified interaction with the SC. The **SCC** is the heart of the **Server** and transforms incoming requests into transactions that are passed to and stored on the BC. To do this, the `requireContract` function checks whether a contract address is present in the execution environment. If not, the user is informed of the absence of said contract address with a Hypertext Transfer Protocol (HTTP) response status code of value 428 and an appropriate error message

about the absence of said contract address. The `attachContract` function is executed when a contract address is present. It creates an Ethers.js contract instance and connects a wallet to it. This instance allows signing transactions that write to the contract and facilitates reading its contents.

The **Frontend** is a single page application built with TypeScript and React. Its main use case is to interact with the data on the BC in a user-friendly way. This is especially aimed at non-technical users¹ who are interested in the production data. It also facilitates data storage for **MilkProducer** and **Laboratory** participants. The **Frontend** is detached from the **Server** and both parts of the system can run independently. There are two ways to interact with an already deployed SC. The first option is to store the contract address in the `.env` file by assigning it to the variable `REACT_APP_CONTRACT`. The second option is to use the **Frontend**. A new contract can be deployed directly through the **Frontend** or through traditional methods using technologies such as Remix or Hardhat. Another important part of the **Frontend** is the **SCC**, which is similar to the **Servers SCC**. However, it uses the React Context API instead of middleware functions. The heart of the **Frontend's SCC** are the **ContractProvider** and **FunctionsProvider** functions. The former passes an Ether contract to the SC on the BC, providing connectivity between the **Frontend** and the SC. The **FunctionsProvider** function is crucial in exposing all the functions of the SC that are used in the **Frontend**.

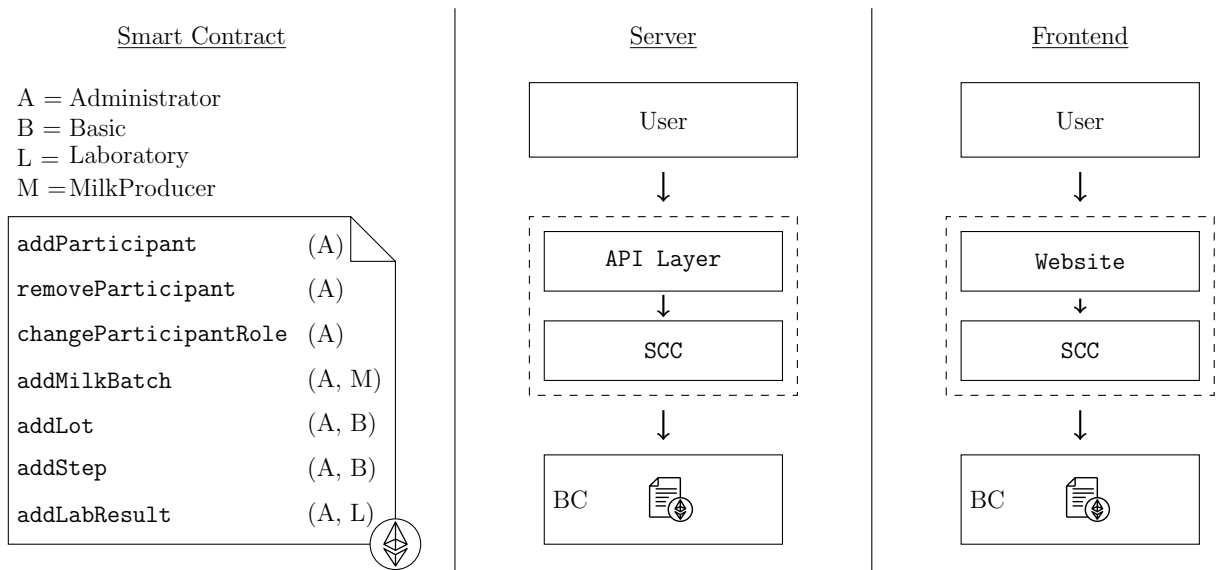


Figure 4.4: Technical illustration of System 2

Figure 4.4 shows the three main components that make up System 2. On the left is the SC with its main functions described above. Next to the functions are the roles with the permissions required to access those functions. The **Server** and the **Frontend** are two independent systems that provide similar functionality. The **Server** provides more functionality than the **Frontend** and is intended for more technical users as the requests are more complex. The **Frontend**, on the other hand, focuses solely on user interaction

¹User in this context refers to consumers as well as participants in the CheeseChain project, such as milk producers and lab workers.

with the SC through a User Interface (UI). Both the **Server** and the **Frontend** use an SCC to interact effectively with the SC.

The **Server** is designed to provide access to the data stored on the SC and allows participants to modify the stored data according to their roles. Users of the **Server** must be technically savvy to take full advantage of the API Layer. In contrast, the **Frontend** provides a non-technical and generally more user-friendly way to interact with the SC.

4.3 Deployment Scenario

The two works are largely independent, nevertheless [36] foresaw to use some of the data stored in the private BC by [35] for storage and further use in the public BC. Therefore, it is obvious to start the deployment process with System 1.

4.3.1 Deployment of System 1

The implementation of System 1 will be done in three phases that are illustrated in Figure 4.5. Prior to deploying System 1, it is necessary to containerize the required scripts that contain the code, configuration files, and dependencies needed to make the system work. The private BC nodes are also containerized. Containerization allows the system to be installed and run on different machines. This approach will ensure that other participants in the CheeseChain project can quickly and easily deploy their own nodes and participate in the network. In the second phase, the system will be deployed and run on a local UZH server. Additionally, three nodes representing the private Ethereum network will be launched and the SC will be deployed on this system. In the second phase, the goal is to have a fully operational System 1. The third phase will be to move from the DigitalQM beta to the one that is in the production environment. This change will likely require changes to the SC. As a result, the modified SC will need to be deployed again. This final phase also includes debugging the system. This approach ensures that the system will run on other machines. Consequently, another instance of the system can be deployed on a different server, and new nodes can be efficiently integrated into the network.

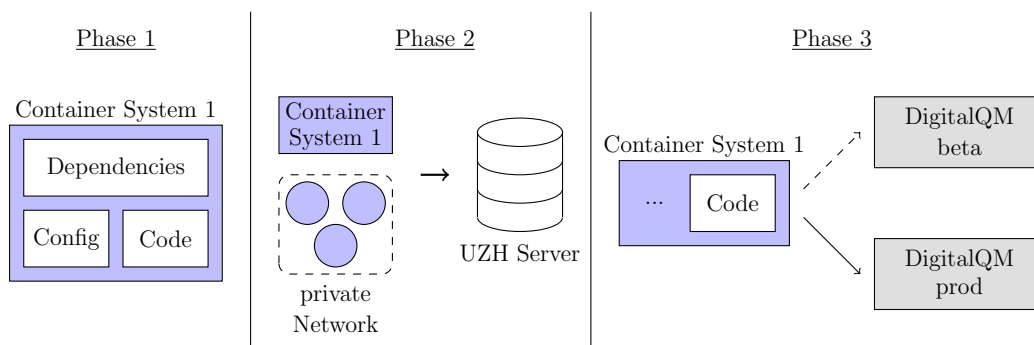


Figure 4.5: Deployment of System 1

4.3.2 Deployment of System 2

The deployment of System 2 is simpler. Since it does not depend on an external system like a DigitalQM, it is a more independent system. Nevertheless, the system still needs to be tested and adjusted if necessary. The deployment scenario of System 2 is illustrated in Figure 4.6. The first phase is analogous to that of System 1, as it also involves containerizing the system. In the second phase, System 2 will also be deployed on a server hosted by UZH, and the SC will be deployed on an Ethereum test network. The third phase will focus on debugging.

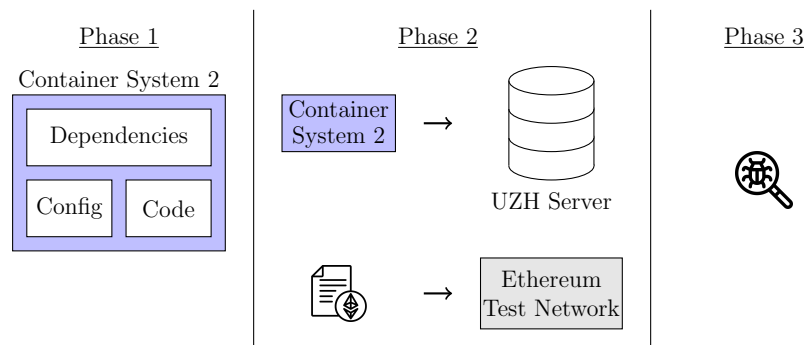


Figure 4.6: Deployment of System 2

Chapter 5

Implementation

This chapter focuses on the implementation of the deployment scenarios proposed in Chapter 4. First, the implementation of a private Ethereum network is described. This network was used to deploy the SCs of both systems. Then the implementation of System 1 is described, followed by the implementation of System 2. Both systems were deployed using an Ubuntu Linux VM, Docker, and Docker Compose as the containerization technology. This thesis is only possible because of the previous work done on System 1 and System 2 as part of two bachelor theses. In order to respect the previous work, the author was careful to change, delete, or add code only where absolutely necessary.

5.1 Implementation of a Private Ethereum Network

To implement a private Ethereum network, three types of nodes are required: a bootnode, a JSON Remote Procedure Call (RPC) endpoint, and a miner node. Below is a more detailed explanation of the three types of nodes:

- The **bootnode** is used for peer discovery. The default configuration is to listen on port 30303. Before becoming part of the Ethereum network, a new client must connect to the bootnode.
- The **JSON-RPC endpoint** exposes a JSON-RPC API over HTTP. The default configuration exposes port 8545. This API is needed by wallets like MetaMask or applications to connect to the Ethereum network.
- The **miner** is responsible for the process of mining, which creates new blocks for the BC. It is typical for there to be multiple miners in a network, but only one is required to establish a network.

The private Ethereum network was implemented using an existing project that was slightly modified to better fit the needs of this thesis [37]. Firstly, the network ID and account password were defined in the `.env` file. To ensure connectivity to the network from outside

the VM, the option `--rpcvhosts=*` had to be added to the RPC endpoint configurations in the `docker-compose.yml` file. In the `genesis.json` file, the MetaMask account used was added to have Ether available when the network is initialized. The private Ethereum network used was already containerized and used the PoW consensus algorithm.

5.2 Implementation of System 1

This section focuses on the implementation of System 1 according to the deployment scenario described in Subsection 4.3.1, with minor modifications. The first phase of the implementation focused on getting the system to run locally on the author's host machine. This phase included some debugging of the system, as there were some problems running it locally. The second phase involved containerizing the system and running the container on the VM. The third and final phase focused on migrating from the DigitalQM beta to the DigitalQM in production.

5.2.1 Phase 1: Local Execution and Debugging

The following is a description of the current state of System 1, an account of the difficulties encountered during the initial run, and the steps taken to resolve those issues. The discussion is in alphabetical order of the modified files, from top to bottom of each file, not in order of implementation.

CheeseChainPrivateSmartContract.sol and CommunicateToSmartContract.py

Originally, System 1's SC had a public access modifier for the constructor. This access modifier has been removed to allow successful compilation with Remix.

In `CommunicateToSmartContract.py` a persistent error was identified as `ContractLogicError`. The error occurred when trying to save a form that had already been saved on the SC or when trying to freeze a form that had already been frozen. To address these issues, the errors have been caught and appropriate error messages have been implemented in the `createNewFormSmartContract` and `freezeForm` functions without changing the call to the SC. Listing 5.1 illustrates the try exception block implemented in the `createNewFormSmartContract` function.

Listing 5.1: Updated `createNewFormSmartContract` Function

```

1 try:
2     self.contract_instance.functions.createForm(id, data["name"], 1391,
        name_of_file, self.createHash(data), "").transact({'from': self.
        __mySCAddress})
3 except ContractLogicError:
4     print("ContractLogicError occurred, the form seems to have been
        created already.")
5 except Exception as e:
6     print("Unexpected error occurred:", e)

```

The modification to `freezeFrom` is similar, but with a slightly different error message for better logging (illustrated in Listing 5.2).

Listing 5.2: Updated `freezeForm` Function

```
1 try:
2     self.contract_instance.functions.freezeForm(id).transact({'from':
3     self.__mySCAddress})
4 except ContractLogicError:
5     print("ContractLogicError occurred, the form seems to be frozen
6     already.")
7 except Exception as e:
8     print("Unexpected error occurred:", e)
```

HandleData.py

The largest number of changes were made to the `HandleData.py` script. Two identical while loops in the script were modified. The first occurrence of the loop is in the `refreshToken` function, which is responsible for automatically retrieving new refresh tokens to maintain the connection to the DigitalQM. The second occurrence is in the `getDocument` function, which is responsible for retrieving the forms from the DigitalQM. In the original while loop, the issue was that the while loop was always executed if no new token was fetched or if the variable `tries` was equal to 5. It can be reasonably assumed that the intention was either to successfully fetch a new token or to try to fetch the token a maximum of five times. Listing 5.3 shows both loops, the original loop on line 1 and the modified loop on line 3.

Listing 5.3: While Loop Comparison

```
1 while got_new_token == False or tries == 5:
2
3 while got_new_token == False and tries < 5:
```

In addition, exception blocks for specific errors and appropriate error messages have been added to properly handle errors and prevent the system from crashing. The modifications to the `refreshToken` function are illustrated in Listing 5.4. The first two exception blocks are designed to handle specific errors, while the third exception block is designed to handle unexpected errors. The function attempts to fetch the token a maximum of five times, waiting a set amount of time (10 seconds at the time of this writing) between retries. Refresh tokens are valid for a limited time. Therefore, retrying is only a viable option up to a certain time, after which a refresh token must be manually fetched and re-entered, and the system must be restarted. The `exit` command on the last line ensures that the system will exit safely.

Listing 5.4: Modified refreshToken Function

```

1 while got_new_token == False and tries < 5:
2     try:
3         response = requests.post('https://beta.qs.fromarte.ch/openid/
4 token', headers=self.headers, cookies=self.cookies, data=self.data)
5         got_new_token = True
6     except requests.exceptions.ConnectionError:
7         print("Connection refused: Failed to fetch refresh token.")
8     except urllib3.exceptions.ConnectTimeoutError:
9         print("Connection timed out: Failed to fetch refresh token.")
10    except Exception:
11        print("Unexpected error occured: Failed to fetch refresh token."
12)
13    finally:
14        tries += 1
15        if not got_new_token:
16            if tries < 5:
17                print("\nRetrying...")
18                time.sleep(config["config"]["retry_interval"])
19            else:
20                print("System will exit as five unsuccessful attempts
21 were made. Please add it manually and restart the system.")
22                exit

```

Listing 5.5: Modified getDocument Function

```

1 response = None
2 while got_connection == False and tries < 5:
3     try:
4         response = client.execute(query, variable_values=params)
5         got_connection = True
6     except requests.exceptions.ConnectionError:
7         print("Connection refused, failed to fetch forms.")
8     except asyncio.exceptions.TimeoutError:
9         print("Timeout error occurred, failed to fetch forms.")
10    except Exception as e:
11        print("An unexpected error occurred, failed to fetch forms.",
12 type(e))
13    finally:
14        tries += 1
15        if not got_connection:
16            if tries < 5:
17                print("Retrying...")
18                time.sleep(config["config"]["retry_interval"])
19            else:
20                print("Failed to establish a connection and to fetch
21 forms. Exiting after 5 tries.")
22 return response

```

As shown in Listing 5.5, the changes in the `getDocument` function are nearly identical to those implemented in the `refreshToken` function. The main difference between the two is the omission of the `exit` command. This is due to the fact that repeating the process after the five unsuccessful attempts may be successful at a later time. This is because the issue

could be caused either by a person accessing the DigitalQM with identical credentials, or by a system overload in general.

In some cases, the modifications in the `getDocument` function resulted in the return of `None` values, which subsequently caused errors in the `updateFiles` function. Implementing a check whether the `getDocument` function returns `None` fixed this issue.

The system could not handle the case where new forms were added to the DigitalQM while the system was running. The system continued to register the same files as new. To address this issue, a check was added to the `checkForNewFiles` function (shown in Listing 5.6). The goal is to iterate through all files that are assumed to be new and compare their creation date and time with the `lastChecked` variable. Files created before the date and time stored in the `lastChecked` variable are deleted from the list as they are not new.

Listing 5.6: New-File-Check in the `checkForNewFiles` Function

```

1 new_forms_keys = list(new_forms.keys())
2 for new_form in new_forms_keys:
3     date_str = new_forms[new_form]['createdAt']
4     if "+" in date_str:
5         date_str = date_str.split("+")[0]
6         date = datetime.fromisoformat(date_str)
7         last_checked = datetime.fromisoformat(self.lastChecked)
8         if date < last_checked:
9             del new_forms[new_form]

```

The last discussed modification to the `HandleData.py` file involved removing a snippet of code from the `getRelevantInfoFromJsonAnswers` function. This code snippet was responsible for throwing a `FileNotFoundError`. Listing 5.7 illustrates the code snippet that caused the error. Deleting this code fixed the problem.

Listing 5.7: Deleted Code Snippet

```

1 if type(json_name) == str:
2     with open(json_name, encoding='utf-8') as f:
3         loaded = json.load(f)
4 else:
5     loaded = json_name

```

main.py

Using a private Ethereum network instead of a local test network caused the RPC endpoint node to be unaware of the account used for publishing the SC and subsequent communication. After each reboot, the account remained registered on the node, but had to be unlocked. Therefore, the newly added function (illustrated in Listing 5.8) is called and executed at system startup. The flow of the function is as follows: first a connection to the node is established, then the account is added if necessary, and finally the account is unlocked. Errors are handled accordingly, and the entire process is logged

using print statements. To deploy and communicate with the SC, the author used Remix and MetaMask.

Listing 5.8: Adding and Unlocking an Account on a Node

```

1 def add_and_unlock_account():
2     load_dotenv()
3     http_provider = os.getenv('HTTP_PROVIDER')
4     private_key = os.getenv('PRIVATE_KEY')
5     metamask_pw = os.getenv('METAMASK_PASSWORD')
6     web3 = Web3(Web3.HTTPProvider(http_provider))
7     if web3.isConnected():
8         print("Connected to Ethereum node")
9     else:
10        print("Failed to connect to Ethereum node")
11        accounts = web3.eth.accounts
12        print("Accounts on the node:", accounts)
13        try:
14            account = web3.eth.account.privateKeyToAccount(private_key)
15            if account.address not in accounts:
16                web3.geth.personal.importRawKey(private_key, metamask_pw)
17                web3.geth.personal.unlockAccount(account.address, metamask_pw,
18                300)
19                print(f"Account {account.address} unlocked")
20            except Exception as e:
21                print("Failed to import/unlock account:", e)
22                accounts = web3.eth.accounts
23                print("Accounts on the node:", accounts)

```

Adding and unlocking an account on the RPC endpoint node requires three pieces of information: the HTTP provider, the private key of the MetaMask account being used, and that account's password. The last two pieces of information are confidential credentials and must not be publicly available. Consequently, the data is stored in a `.env` file that is not part of the repository, and the example file in Listing 5.9 illustrates what the `.env` file looks like. However, this still poses a security risk, as anyone with access to the folder can see the credentials. Therefore, this configuration was only used during debugging and was replaced in the containerization phase.

Listing 5.9: Example File for Confidential Credentials

```

1 HTTP_PROVIDER=your-http-provider
2 PRIVATE_KEY=your-metamask-private-key
3 METAMASK_PASSWORD=your-metamask-password

```

Another minor modification was to move the `print("ran for the first time")` statement to the end of the `firstTimeRun` function when creating forms on the SC was complete.

5.2.2 Phase 2: Containerization

The second phase of deploying System 1 involved containerization, which consisted of two steps. First, a Dockerfile was created to specify the build process for System 1's Docker containers. Next, a Docker Compose file was developed to facilitate the management of the containers, and Docker secrets were included to enhance security.

Dockerfile

The Dockerfile is designed to be simple. The first line installs Python. Line 2 creates an application directory and sets it as the working directory. Then line 3 copies all the necessary files into the container. Line 4 installs the dependencies, and finally line 5 starts the system with the `-u` flag to ensure that output is unbuffered and logged in real time.

Listing 5.10: Illustration of the Dockerfile

```
1 FROM python:3.10
2 WORKDIR /app
3 COPY . .
4 RUN pip install -r requirements.txt
5 CMD ["python3", "-u", "main.py"]
```

Docker Compose File

To manage the containers, called services, you must specify them in the Docker Compose file. The Docker Compose file uses an image from Docker Hub, which means an image must be built and pushed before it can be used effectively. The rest of the file, specifically lines 6 through 21, ensures that Docker secrets are properly accessed and defined. Listing 5.11 illustrates the complete file.

Listing 5.11: Illustration of the Dockerfile Compose File

```

1 version: '3.8'
2 services:
3   system_1:
4     image: <DOCKER-HUB-USERNAME>/system_1:tag
5     secrets:
6       - http_provider
7       - private_key
8       - metamask_password
9     environment:
10      HTTP_PROVIDER_FILE: /run/secrets/http_provider
11      PRIVATE_KEY_FILE: /run/secrets/private_key
12      METAMASK_PASSWORD_FILE: /run/secrets/metamask_password
13 secrets:
14   http_provider:
15     external: true
16   private_key:
17     external: true
18   metamask_password:
19     external: true

```

Docker Secrets

The advantages of Docker secrets over environment variables include encryption and limited scope, ensuring that secrets are securely stored and only accessible by specific containers. To set and use Docker secrets, the project must be part of a Docker swarm and connected to the swarm node. One way to ensure that the project is running in a swarm is to initialize it with `docker swarm init` and then set the Docker secrets before the application boots. Listing 5.12 illustrates how to set the Docker secrets once the swarm is initialized.

Listing 5.12: Example on how to set Docker Secrets

```

1 echo "https://example.com" | docker secret create http_provider - && \
2 echo "your_private_key" | docker secret create private_key - && \
3 echo "your_password" | docker secret create metamask_password -

```

The final step in migrating from environment variables to Docker secrets was to access the secrets in the Python code. To accomplish this, the `get_secret` helper function was implemented in the `main.py` file to load the secrets. Listing 5.13 illustrates the helper function. Additionally, lines 3 to 5 in Listing 5.8 have been modified to use the `get_secret` function instead of loading the environment variables directly.

Listing 5.13: Helper Function for accessing Docker Secrets

```

1 def get_secret(secret_name):
2     secret_file_path = os.getenv(f"{secret_name.upper()}_FILE")
3     if secret_file_path:
4         with open(secret_file_path, 'r') as file:
5             secret_value = file.read().strip()
6             return secret_value
7     else:
8         raise FileNotFoundError(f"{secret_name.upper()}_FILE environment
    variable not set or file not found")

```

5.2.3 Phase 3: Migration to Production

The migration to the production version of the DigitalQM was straightforward. The only change required was to change the Uniform Resource Locator (URL) from the beta version, *i.e.*, <https://beta.qs.fromarte.ch/>, to the production version, *i.e.*, <https://qs.fromarte.ch/>. The production system had only four files that were published at long intervals. As a result, the fetch interval for new files was reduced to twice a day instead of every 10 seconds. In addition, since the refresh token expires quickly, the `refreshToken` function needed to be run more frequently. To address this, the `refreshToken` function call was left in place, while the rest of the while loop in the `main` function was moved to an if-else statement. The modified `main` function is shown in Listing 5.14.

Listing 5.14: Modified Main Function

```

1 remaining_time = config["config"]["form_fetch_interval"]
2 token_fetch_interval = config["config"]["token_fetch_interval"]
3 while True:
4     d.refreshToken()
5     if remaining_time <= 0:
6         refetchingFreezingAndUpdating()
7         new_files, local_names = d.checkForNewFiles()
8         createNewFormOnSC(new_files, local_names)
9         remaining_time = config["config"]["form_fetch_interval"]
10    remaining_time -= token_fetch_interval
11    time.sleep(token_fetch_interval)

```

5.3 Implementation of System 2

This section describes the implementation of System 2, following the scenario described in Subsection 4.3.2, with minor modifications. As with System 1, the first phase involved running and debugging the application on a local machine. The second phase was also the last, since the system is self-contained and focused on containerizing the system.

5.3.1 Phase 1: Local Execution and Debugging

The original system was developed using a local Hardhat network, configured and stored in the `chain` folder. For the new implementation, a different approach was taken by setting up a private Ethereum network, similar to System 1, instead of the local Hardhat network. The other two components of the system were the `Server` and the `Frontend`. Because of the user-friendly nature of the `Frontend`, the author focused on its implementation first.

Frontend

Once the private network was up and running, the next step was to deploy the SC and store its address in the `.env` file, then a Google Maps API key had to be created and also stored in the `.env` file, the last modification to the `.env` was to add the URL of the RPC endpoint. Second, the private network ID, *i.e.*, 1297 in the default configuration, was added to the `ChainDisplay.tsx` and `connectors.ts` files. Third, there was a type mismatch error in the `FormSkeleton.tsx` file. Listing 5.15 illustrates the change that was made, which was to make the variable `T` extend `FormikValues`. This modification ensures that `T` is always compatible with `FormikValues`. These three modifications allowed the `Frontend` to work as intended.

Listing 5.15: Modified FormSkeleton.tsx file

```
1 export const FormSkeleton = <T,>({
2
3 export const FormSkeleton = <T extends FormikValues,>({
```

Server

The `Server` is used to interact with the system and provides the same core functionality as the `Frontend`, but also provides additional interactions such as retrieving the number of milk batches stored on the SC. Both the `Server` and the `Frontend` can be used simultaneously. This paragraph assumes that the SC is already deployed on a network. To start the `Server`, a `.env` file must be created. This file must specify the port, store the private key of the interacting account, specify the RPC endpoint URL, and store the SC address.

GET requests can be made directly in the browser by specifying the port and desired action. Listing 5.16 illustrates an example of a request to get the number of milk batches stored on the BC. Line 1 shows the request, and line 3 shows the response. To perform the same GET request from a terminal, the `curl` command must be used before the request.

Listing 5.16: Example of a GET Request in the Browser

```
1 http://localhost:8080/total-milk-batches
2
3 {"totalMilkBatches":10}
```

Adding a milk batch requires more detailed information and is only done using a terminal. Listing 5.17 illustrates a POST request to add a new milk batch. This request requires the coordinates to be specified explicitly. This approach requires detailed knowledge of and about the API, which in this case means looking at the `CheeseChain.json` file that defines the Application Binary Interface (ABI) for interacting with the SC.

Listing 5.17: Example of a POST Request from the Terminal

```
1 curl -X POST http://localhost:8080/add-milk-batch -H "Content-Type:
  application/json" -d '{"coordinates": {"latitude":47.38,"longitude
  ":8.55}}'
```

When accessing the URL for the `Server`, the private key of the wallet used for SC interactions was displayed. This was a security risk and did not provide any useful information for interacting with the SC through the `Server`. To address this, the private key display has been removed and replaced with instructions on how to perform GET requests through the browser to simplify interaction with the SC.

5.3.2 Phase 2: Containerization

This phase focused on containerizing System 2. The first step was to create `Dockerfiles` for both the `Frontend` and the `Server`. The second step was to create a `docker-compose.yml` file that integrates the `Frontend` and `Server` into a unified system.

Frontend

Listing 5.18 illustrates the Dockerfile for the `Frontend`. The workflow starts with specifying and installing the node version. Next, an app directory is created and set as the working directory. Then all the files from the `Frontend` are copied to the app directory. Line 4 ensures that the `.env` file is included in the Docker container, which became necessary for the `Frontend` once it was implemented on the VM. Line 5 ensures that the dependencies are installed. Line 6 exposes the port, and finally line 7 launches the application.

Listing 5.18: The Dockerfile of the Frontend

```
1 FROM node:20
2 WORKDIR /app
3 COPY . .
4 COPY .env .env
5 RUN yarn install
6 EXPOSE 3000
7 CMD ["yarn", "start"]
```

Server

The Dockerfile illustrated in Listing 5.19 is virtually identical to the one in **Frontend**. The only difference is in line 5, where the exposed port is defined. The **Server** exposes port 3001 instead of 3000 to avoid conflicts.

Listing 5.19: The Dockerfile of the Server

```
1 FROM node:20
2 WORKDIR /app
3 COPY . .
4 RUN yarn install
5 EXPOSE 3001
6 CMD ["yarn", "start"]
```

Docker Compose

The integration of both the **Frontend** and the **Server** into System 2's final application serves a specific purpose. Although the core functionality is the same, the **Frontend** is more user-friendly due to its UI, which frees the user from having to type cumbersome commands in the terminal. The **Server**, on the other hand, does not have a UI, but provides additional functionality such as retrieving the total number of milk batches or lots.

The final step in containerizing System 2 is to create the `docker-compose.yml` file to combine the **Frontend** and **Server** into a single application. Listing 5.20 shows the `docker-compose.yml` file, which is in the parent folder of the **Frontend** and **Server** folders. Therefore, the context must be specified in the build command. Lines 6 to 7 and 11 to 12 specify the ports exposed by the **Frontend** and **Server** respectively. Since the services are independent, they can be built at the same time.

One problem that only affected the **Frontend** and became apparent once it was implemented on the VM was access to the environment variables. Part of the solution was already described in 5.3.2. The second part was to specify the location of the file that stores the environment variables, which is done in lines 8 to 9 of Listing 5.20.

Listing 5.20: The Docker Compose File of System 2

```
1 version: '3.8'
2 services:
3   frontend:
4     build:
5       context: ./frontend
6     ports:
7       - "3000:3000"
8     env_file:
9       - ./frontend/.env
10  server:
11    build:
12      context: ./server
13    ports:
14      - "3001:3001"
```


Chapter 6

Evaluation

This chapter focuses on evaluating the prototypes implemented in Chapter 5. Successful deployment requires a comprehensive evaluation of the application’s performance and usability. Since the systems are largely independent, they were evaluated separately. Both System 1 and System 2 were evaluated in terms of their runtime when run locally versus on the VM, and when run non-containerized versus containerized. Since this thesis deals with the implementation and deployment of two existing systems, the focus was on evaluating the deployment solution rather than the systems themselves. Finally, both System 1 and System 2 were evaluated based on their usability.

6.1 Runtime Evaluation

The evaluation is performed as follows: First, the runtime of the system is measured on the host machine, both when it is containerized and when it is not. Second, the same evaluation is performed on the VM. Environment consistency is critical for the evaluation. Therefore, it is necessary to ensure that the configurations - such as the same Python or Node version and identical dependencies - of the Docker container, the host machine, and the VM are as similar as possible. The first step is to update the code and run all four configurations. The second step is to summarize and compare the results.

The host machine was a MacBook Pro with an Apple M2 Pro chip, 10 cores, and 32 GB of Random-Access Memory (RAM). The VM had an Intel Xeon E312xx (Sandy Bridge) chip, 4 Central Processing Units (CPUs), 1 CPU core, and 8 GB of RAM.

6.1.1 Test Setup

The first step is to update the code and run the system in four different configurations. To measure elapsed time, the `time.perf_counter()` function was used because it provides “the highest available resolution to measure a short duration”[38]. For consistency, a dedicated private Ethereum network was instantiated for each system and used for all

runs of that system. Listing 6.1 illustrates the code modifications to capture runtime. Line 1 captures the time before each code snippet is executed, line 3 captures the time after execution, and line 4 returns the delta of the two.

Listing 6.1: Measuring the Runtime

```

1 start_time = time.perf_counter()
2 # tested code
3 end_time = time.perf_counter()
4 print("\nelapsed time:", end_time - start_time, "\n")
5 # rest of the code

```

6.1.2 System 1

Since there is one main function in System 1, the testing focused on it. A total of 120 runs were performed, 30 for each configuration. Second, a new SC was deployed and used for each run to ensure that no data was already stored on it, as the behavior is different when data is present. The following section presents the results of the tests performed on System 1, followed by a discussion of these results.

Results

The results of the runs are shown in Figure 6.2. The X-axis shows the four different environments: the non-dockerized and dockerized versions on the host and VM, respectively. The Y-axis shows the runtime results, including the mean, median, maximum, and minimum execution times, as well as the standard deviation.

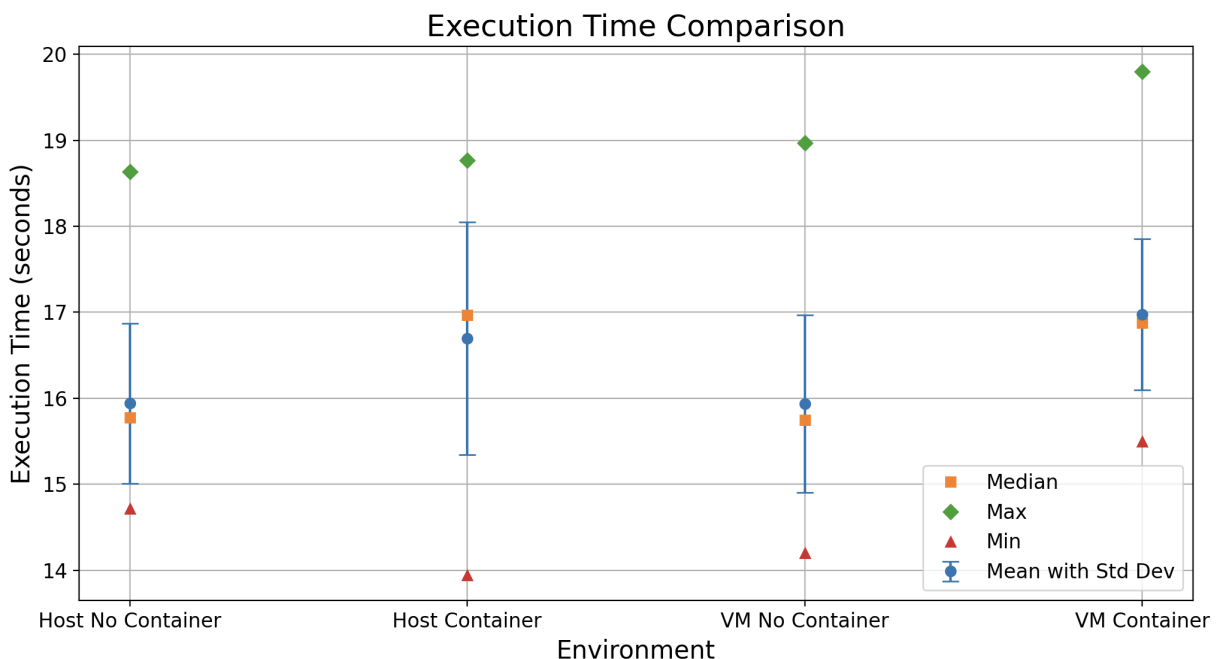


Figure 6.2: Plotted Runtimes of System 1 in different Environments.

Additionally, Table 6.1 provides a numerical summary of the execution times for each environment:

Table 6.1: Summary of System 1 Execution Times for Different Environments

Environment	Mean (s)	Median (s)	Std Dev (s)	Max (s)	Min (s)
Host Non-Docker	15.940	15.776	0.930	18.637	14.718
Host Docker	16.695	16.970	1.352	18.766	13.941
VM Non-Docker	15.932	15.748	1.032	18.968	14.199
VM Docker	16.976	16.880	0.880	19.799	15.493

Discussion

Looking at the mean and median execution times in each environment, we see that the non-dockerized versions produced similar results for both the host machine and the VM. Similarly, the dockerized versions produced similar results on both machines. The small differences recorded are likely due to the number of runs performed. This suggests that the performance of the system is not significantly affected by running on a VM compared to a local machine. On average, the dockerized version was 0.899 seconds slower over the 30 runs performed.

The standard deviation, maximum and minimum execution times paint a slightly different picture, as the results from the dockerized and non-dockerized configurations do not closely resemble each other. For example, the highest standard deviation was recorded on the dockerized version on the host machine, while the lowest standard deviation was recorded on the dockerized version on the VM.

In summary, containerizing System 1 most likely had a slightly negative impact on its runtime. However, it cannot be conclusively determined how containerization affects the stability of the runtime over time.

6.1.3 System 2

This subsection describes the runtime evaluation of System 2. Due to the multiple functionalities of the system, it was not possible to adequately test the system as a whole. Therefore, the individual functionalities of the system, especially those of the `Server`, were tested separately. This approach was taken because testing the runtime when user interaction is required, such as clicking buttons and confirming SC transactions, is not reliably reproducible. The functionality tested included adding a milk batch, adding a lot, and adding a step. When adding a lot or adding a step, the system relies on data already stored on the SC. Therefore, the same SC was used for the entire test of each configuration, resulting in four SCs being deployed and used. Several Python scripts were written to facilitate the process of generating, evaluating, and merging runtime results.

A total of 1200 test runs were performed: 300 per configuration, with 100 for each of the three functions tested, i.e., adding a milk batch, adding a lot, and adding a step.

Results

The results are illustrated in Figure 6.3. The first row of the X-axis lists the different functions that were executed, and the second row lists the different environments. The gray dashed lines serve as visual separators between the environments. The Y-axis shows the mean, median, maximum, and minimum runtime in seconds, as well as the standard deviation.

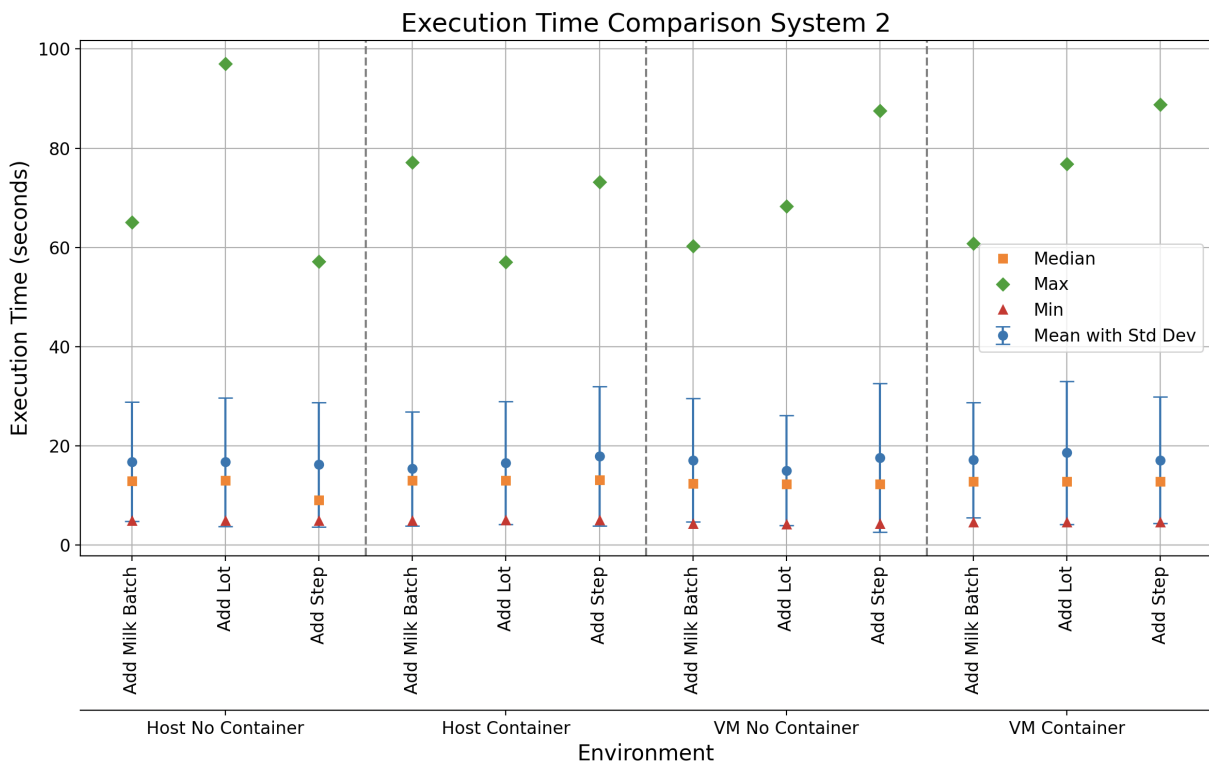


Figure 6.3: Plotted Runtimes of System 2 Functions in different Environments.

To gain a better understanding, the data from each function was combined, resulting in a focus on the four environments alone. The result is illustrated in Figure 6.4.

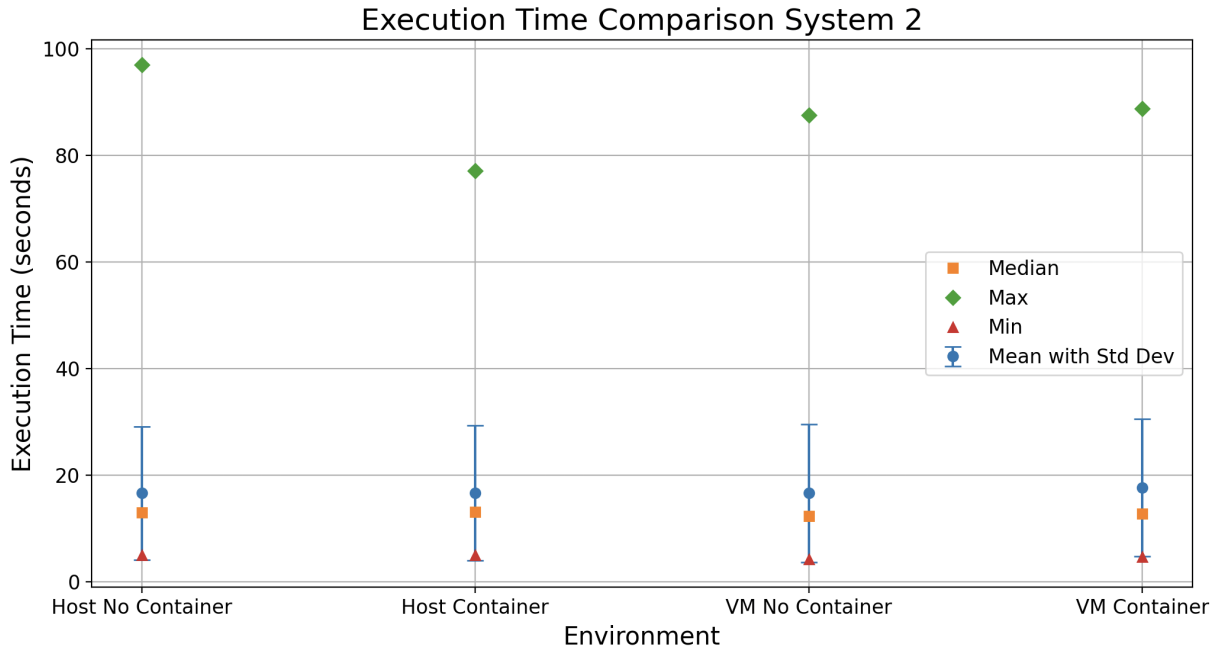


Figure 6.4: Merged Plotted Runtimes of System 2 in different Environments.

A numerical summary of the complete data is provided in Table 6.2. The results are grouped similarly to Figure 6.3: first the four environments are listed, followed by the functions tested and their respective runtime.

Table 6.2: Execution Time Summary of System 2 Functions in different Environments

Env.	Function	Mean (s)	Median (s)	Std Dev (s)	Max (s)	Min (s)
Host No Container	Add Milk Batch	16.838	12.985	12.013	65.027	4.925
	Add Lot	16.757	13.016	12.982	97.021	4.947
	Add Step	16.240	9.122	12.546	57.167	4.948
Host Container	Add Milk Batch	15.398	13.023	11.509	77.141	4.951
	Add Lot	16.615	13.079	12.385	57.072	4.990
	Add Step	17.950	13.170	14.026	73.161	5.010
VM No Container	Add Milk Batch	17.145	12.391	12.466	60.315	4.300
	Add Lot	15.042	12.305	11.086	68.332	4.242
	Add Step	17.602	12.336	15.012	87.516	4.274
VM Container	Add Milk Batch	17.190	12.800	11.589	60.826	4.664
	Add Lot	18.644	12.802	14.416	76.801	4.606
	Add Step	17.107	12.799	12.745	88.787	4.607

The test runs lasted approximately 90 minutes per environment. This length of time allowed for an analysis of the performance of the functions over time, and provided an

opportunity to investigate whether stored data had a noticeable effect on speed. Figure 6.5 illustrates the different environments and their performance over time.

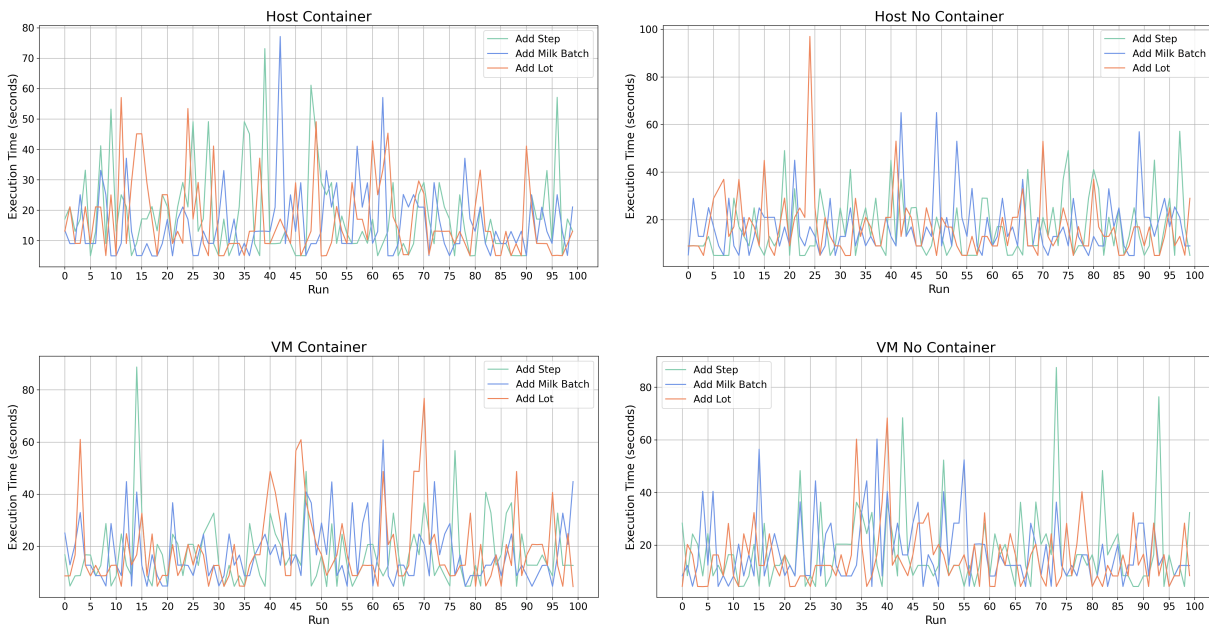


Figure 6.5: Plot of the Runtimes for each Configuration over Time.

Discussion

The results were less consistent than for System 1. The extremes, especially the maximum times required, were in some cases much more pronounced compared to the average runtime measured. For example, for the `add-lot` function in the non-containerized version on the host, the average runtime was 16.757 seconds. The maximum runtime of 97.021 seconds was almost six times the average, while the minimum runtime of 4.947 seconds was less than a third of the average. In addition, there was no clear trend, as the functions did not consistently run faster or slower in a given scenario.

To illustrate this point, let's compare the `add-milk-batch` and `add-step` functions on the host, both in a container and without a container. The `add-milk-batch` function ran almost a second and a half slower on average in the non-dockerized environment, while the opposite was true for the `add-step` function, which ran just over a second and a half faster in the non-dockerized environment.

Despite the large spikes in runtime, the system appears to be fairly consistent over time, with no apparent loss in performance. The spikes were also of similar intensity across time and functions.

In summary, the expected runtime of the functions averages between 15 and 18 seconds, but can be as fast as four seconds and as slow as 97 seconds.

6.2 System 2 - Frontend

The evaluation of the Frontend was done using Lighthouse and usability tests with two potential users.

6.2.1 Lighthouse

Lighthouse analyzes websites according to four criteria: performance, accessibility, best practices, and Search Engine Optimization (SEO). The results contain a degree of variability [39]. Therefore, the command shown in Listing 6.6 was used to perform 30 analyses per environment and generate the corresponding reports containing the results.

Listing 6.6: Generating Lighthouse Reports

```
1 npx -p @lhci/cli lhci collect --url http://localhost:3000/connect -n 30
2 npx -p @lhci/cli lhci upload --target filesystem --outputDir ~/path/to/
  report/folder
```

Results

Table 6.3 illustrates that the results differ only for the performance benchmark, while the results for the accessibility, best practices, and SEO benchmarks remained consistent across all environments.

Table 6.3: Benchmarks Medians from the Lighthouse Reports.

Environment	Performance (%)	Accessibility (%)	Best Practices (%)	SEO (%)
Host Non-Docker	64.2	88.0	93.1	100
Host Docker	64.2	88.0	93.1	100
VM Non-Docker	63.5	88.0	93.1	100
VM Docker	63.2	88.0	93.1	100

Table 6.4 shows the mean, median, maximum and minimum execution times for each environment, as well as the standard deviation.

Table 6.4: Performance Medians from the Lighthouse Reports.

Environment	Mean (%)	Median (%)	Std Dev (%)	Max (%)	Min (%)
Host Docker	64.3	64.0	0.8	65.0	61.0
Host Non-Docker	64.2	64.0	0.8	65.0	61.0
VM Non-Docker	63.5	64.0	1.0	65.0	59.0
VM Docker	63.2	63.0	1.0	64.0	59.0

Discussion

This subsection discusses the results recorded in the previous subsection, focusing on the differences found in the respective environments and the results themselves.

Table 6.3 shows that the accessibility, best practices, and SEO benchmarks remained unchanged across all environments. Performance was slightly worse on the VM compared to the host machine, as indicated by the lowest recorded score and the mean score. Additionally, the dockerized version experienced a slight loss in performance compared to the non-dockerized version on the VM. Table 6.4 further illustrates that the performance measured on the host machine was slightly more consistent compared to the VM, as indicated by the lower standard deviation.

A more detailed discussion of the performance benchmark is necessary because it was the only benchmark of the four that was unsatisfactory. Lighthouse measures key performance metrics and assigns weights to them, resulting in a final score. This discussion focuses on two metrics in particular: the highest weighted metric, at 30%, called Largest Contentful Paint (LCP), and one of the two second highest weighted metrics, at 25%, called Total Blocking Time (TBT) [40].

LCP measures the time it takes to load the main content of the website. An LCP under 2.5 seconds is considered fast, an LCP between 2.5 and 4 seconds is considered moderate, and an LCP over 4 seconds is considered slow [41]. The reported LCP metrics were routinely over 8 seconds. The LCP resource in the `Frontend` is the Tête-de-Moine logo. TBT measures the time the website is blocked after the user has interacted with it through mouse clicks, screen taps, or keyboard presses. Response times between 0 and 200 milliseconds are considered fast, between 200 and 600 milliseconds are considered moderate, and times over 600 milliseconds are considered slow [42]. The recorded TBT times averaged between 400 and 500 milliseconds.

The LCP can be improved by ensuring that the LCP resource is loaded at the same time as the first resource loaded by the website. Another improvement is to ensure that the LCP resource is rendered immediately after it is loaded. A third measure would be to reduce the size of the LCP resource. Finally, reducing the time it takes to deliver the initial Hypertext Markup Language (HTML) can improve the LCP score; this can be achieved, for example, by reducing the number of redirects the website requires [43].

Improving the TBT boils down to two measures: reducing or eliminating unnecessary JavaScript loading, parsing, or execution where possible, and improving inefficient JavaScript code [42].

6.2.2 Usability

To evaluate the usability of the `Frontend`, three potential users were given short instructions. The goal was to determine if the interface was designed to be intuitive.

Setup and Questions

A private Ethereum network was set up, the SC was deployed, and the **Frontend** was initialized. The entire system was run inside a VM to closely mimic a real-world scenario. The following instructions were given:

Scenario: You are a customer who has just purchased a cheese and would like to learn more about the product. On the back of the packaging, you find the following information:

- URL to access the data: <https://frontend-system2.comsyslab.xyz/>
- Smart contract (SC) address: 0xaF3D61a8308496E6aD8B937e7601209B38B5813E
- Lot number of the cheese: 1

Please execute the following tasks and document your actions and the reasons behind them. After completing each task, please rate the intuitiveness of the process on a scale of 1 to 5, with 1 being "not intuitive at all" and 5 being "extremely intuitive".

1. Access the website to retrieve production data.
2. Retrieve the production data related to the specified lot of the cheese.
3. Identify the location where the milk was produced.
4. Specify the date and time of the second production step.
5. Report the results of the laboratory authentication test.

Results

The answers themselves are not included in the main text, but can be found in Appendix B. Table 6.5 shows a summary of the ratings for each task and the overall rating, which is calculated as the average rating for each task.

Table 6.5: Summary of the Ratings.

Task 1	Task 2	Task 3	Task 4	Task 5	Overall
5	5	4.67	5	4	4.79

Discussion

The fact that all reviewers were able to solve almost all of the tasks and the overall rating of 4.79 out of 5 indicates that the tasks were adequately solvable. The only task that was not solved was task 5 by reviewer 1, which may have been due to unclear task formulation or wording.

Two tasks did not receive a unanimous rating of 5: Tasks 3 and 5. In Task 3, all three reviewers appreciated the use of Google Maps to show the production site because they were familiar with it. Reviewer 3 noted that the inclusion of timestamps could improve the user experience by eliminating ambiguity in timelines for production steps performed on the same day. In addition, zooming in further from the beginning to make the location clearer and providing a direct link to Google Maps were suggested improvements.

In Task 5, Reviewer 2 highlighted that the laboratory test results were presented in an ambiguous manner, leading to uncertainty about what exactly was being tested - individual production steps or the overall quality of the cheese - and how well the test was passed. As the test indicates whether the cheese is a genuine Tête-de-Moine, it would be beneficial to clarify what is tested and how the results are determined.

6.3 Comparison with Related Work

In Chapter 3, the goal was to identify a research gap that this thesis aimed to fill. The identified research gap was to implement a supply chain tracing application and deploy it into production. Chapter 5 and the current chapter have described in detail how this thesis provided a deployed solution to this very problem. Additionally, it presents a novel approach to combating fraud in the cheese industry by combining a DNA-based intrinsic product authentication system with BC technology.

Chapter 7

Summary and Future Work

This chapter summarizes the work that has been done and briefly describes the future work that remains to be done.

7.1 Summary

This thesis focused on the integration and deployment of a supply chain tracing application consisting of two separate systems. System 1 retrieves data from FROMARTE's DigitalQM and stores selected data in a private BC. System 2 provides a **Frontend** and a **Server** to interact with the SC on the BC. These interactions include managing participants and storing and retrieving production-related data on the BC.

Chapter 2 provides the necessary background to better understand this work, focusing on BC technology, supply chain management, and containerization. Chapter 3 analyzes previous approaches to this problem and identifies a research gap, primarily related to the types of data stored and deployed.

Chapter 4, describes both systems, using an application scenario for each, followed by a technical analysis of the code functionality and deployment scenario. In Chapter 5, an existing private BC setup was slightly modified to meet the needs of the application. Both systems were implemented by first running and debugging them on the author's local machine, followed by containerization using Docker, Docker Compose, and Docker secrets. After containerization, the systems were deployed to the VM, where further debugging was required due to differences in behavior compared to the local machine. System 1 also had to be switched from the DigitalQM beta to the production version.

Chapter 6 focused on evaluating the containerization of the systems. Several methods were used to evaluate different aspects of the application. The runtime of System 1 and the **Server** of System 2 were analyzed to evaluate the impact of containerization. The runtime of System 1 was minimally increased by containerization, both on the host machine and in the VM. The results indicated that the **Server** was not significantly affected by containerization, either negatively or positively. System 2's **Frontend** was

analyzed using Lighthouse, which showed no significant effect of containerization and slightly lower performance on the VM compared to the host machine. The usability test for the `Frontend` was generally positive, but highlighted the need for better error handling, especially regarding geolocation.

Containerizing both systems decoupled the application from a local development environment, allowing it to be moved to and run on other machines, improving flexibility. A project like the Cheesechain project would benefit from implementing a CI/CD pipeline in the future, starting with the creation and deployment of smaller applications. This approach would make a working application available sooner and minimize the loss of information between the people developing the application and those implementing and deploying it.

7.2 Future Work

Both systems need to be evaluated over time, ideally under production conditions. As more real users interact with the system, additional areas for improvement will be identified.

Once the SC for System 2 is deployed on the main network, the private BC currently used for both systems will be used for System 1 only. It is important to switch from PoW to PoA in the private BC because PoA is the preferred consensus mechanism in private BCs. It would also be beneficial for other Cheesechain members to host their own nodes. Another useful feature, if feasible, would be a script that automatically fetches the first refresh token to further automate the startup process.

The SC intended for production use in System 2 needs to be deployed on the main network. Currently, System 2 can continue to run despite geolocation error messages. However, the user experience would be improved if these error messages were sent only when absolutely necessary. A possible solution is to stop requesting geolocation for `ViewOnly` users, which is the default role for all users.

Immediately after System 2's SC is deployed on the main network, costs will automatically increase significantly. As pointed out in [36], these costs could potentially be significant, suggesting that a different approach may be needed instead of storing all data on the public BC. A hybrid approach like that of [34] is worth exploring. In this approach, all data is stored on a private network, while only parts of the data, or metadata, are stored on the main network.

The Lighthouse evaluation performed in section 6.2 showed that there are ways to improve `Frontend` performance, such as loading the LCP faster and improving the TBT by optimizing JavaScript. The usability evaluation suggested some minor design improvements, such as placing the test result and including a timestamp for the test results.

Finally, the link between System 1 and System 2 needs further investigation. It is in the `/private/fears` folder, but is not considered fully functional.

Bibliography

- [1] [Accessed 26-04-2024]. [Online]. Available: <https://www.csg.uzh.ch/csg/en/research/CheeseChain.html>.
- [2] R. Sapra and P. Dhaliwal, “Blockchain: The new era of technology”, in *2018 Fifth International Conference on Parallel, Distributed and Grid Computing (PDGC)*, 2018, pp. 495–499. DOI: 10.1109/PDGC.2018.8745811.
- [3] Satoshi Nakamoto, *Bitcoin: A peer-to-peer electronic cash system*, Decentralized business review, 2008.
- [4] M. Niranjnamurthy, B. N. Nithya, and S. J. C. C. Jagannatha, “Analysis of blockchain technology: pros, cons and SWOT”, in *Cluster Computing*, 22, 2019, pp. 14 743–14 757.
- [5] N. Tikoo, *Logistics and supply chain management DMGT523 Edited*. New Delhi: Lovely Professional University Phagwara, 2023.
- [6] [Accessed 01-06-2024]. [Online]. Available: <https://www.migros.ch/de/product/210685103000>.
- [7] [Accessed 01-06-2024]. [Online]. Available: <https://www.tetedemoine.ch/en/infos/production>.
- [8] M. N. M. Bhutta, A. A. Khwaja, A. Nadeem, *et al.*, “A survey on blockchain technology: Evolution, architecture and security”, *IEEE Access*, vol. 9, pp. 61 048–61 073, 2021. DOI: 10.1109/ACCESS.2021.3072849.
- [9] H. Sheth and J. Dattani, “Overview of blockchain technology”, *Asian Journal For Convergence In Technology (AJCT) ISSN -2350-1146*, 2019. [Online]. Available: <http://asianssr.org/index.php/ajct/article/view/728>.
- [10] M. Jakobsson and A. Juels, “Proofs of work and bread pudding protocols(extended abstract)”, in *Secure Information Networks: Communications and Multimedia Security IFIP TC6/TC11 Joint Working Conference on Communications and Multimedia Security (CMS’99) September 20–21, 1999, Leuven, Belgium*, B. Preneel, Ed. Boston, MA: Springer US, 1999, pp. 258–272, ISBN: 978-0-387-35568-9. DOI: 10.1007/978-0-387-35568-9_18. [Online]. Available: https://doi.org/10.1007/978-0-387-35568-9_18.
- [11] I. Bentov, A. Gabizon, and A. Mizrahi, “Cryptocurrencies without proof of work”, *CoRR*, vol. abs/1406.5694, 2014. arXiv: 1406.5694. [Online]. Available: <http://arxiv.org/abs/1406.5694>.

- [12] C. N. Samuel, S. Glock, F. Verdier, and P. Guitton-Ouhamou, “Choice of ethereum clients for private blockchain: Assessment from proof of authority perspective”, in *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2021, pp. 1–5. DOI: 10.1109/ICBC51069.2021.9461085.
- [13] P. Paul, P. Aithal, R. Saavedra, and S. Ghosh, “Blockchain technology and its types—a short review”, *International Journal of Applied Science and Engineering (IJASE)*, vol. 9, no. 2, pp. 189–200, 2021.
- [14] S. Wang, Y. Yuan, X. Wang, J. Li, R. Qin, and F.-Y. Wang, “An overview of smart contract: Architecture, applications, and future trends”, in *2018 IEEE Intelligent Vehicles Symposium (IV)*, 2018, pp. 108–113. DOI: 10.1109/IVS.2018.8500488.
- [15] N. Szabo, “Smart contracts: Building blocks for digital markets”, 1996. [Online]. Available: <https://api.semanticscholar.org/CorpusID:198956172>.
- [16] W. Zou, D. Lo, P. S. Kochhar, *et al.*, “Smart contract development: Challenges and opportunities”, *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2084–2106, 2021. DOI: 10.1109/TSE.2019.2942301.
- [17] V. Buterin *et al.*, “Ethereum white paper”, 2014.
- [18] *History and forks of ethereum*, [Accessed 26-04-2024]. [Online]. Available: <https://www.ethereum.org/en/history>.
- [19] [Accessed 26-04-2024]. [Online]. Available: <https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/rewards-and-penalties>.
- [20] [Accessed 26-04-2024]. [Online]. Available: <https://ethereum.org/en/dapps>.
- [21] M. Bez, G. Fornari, and T. Vardanega, “The scalability challenge of ethereum: An initial quantitative analysis”, in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2019, pp. 167–176. DOI: 10.1109/SOSE.2019.00031.
- [22] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, “Defining smart contract defects on ethereum”, *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 327–345, 2022. DOI: 10.1109/TSE.2020.2989002.
- [23] [Accessed 22-06-2024]. [Online]. Available: <https://ethereum.org/en/enterprise/>.
- [24] [Accessed 22-06-2024]. [Online]. Available: <https://www.ibm.com/topics/containerization>.
- [25] [Accessed 22-06-2024]. [Online]. Available: <https://docs.docker.com/guides/docker-overview/>.
- [26] [Accessed 26-04-2024]. [Online]. Available: <https://www.blockchain.uzh.ch/projects/application-of-blockchain-technology-in-the-swiss-cheese-supply-chain/>.
- [27] S. R. Niya, D. Dordevic, M. Hurschler, S. Grossenbacher, and B. Stiller, “A blockchain-based supply chain tracing for the swiss dairy use case”, in *2020 2nd International Conference on Societal Automation (SA)*, 2021, pp. 1–8. DOI: 10.1109/SA51175.2021.9507182.

- [28] I. Liyanage, N. Madhuwantha, M. Perera, S. Ruhunage, M. M. Hansika, and L. Rupasinghe, “Idairy: Intelligence and secure e-commerce platform for dairy production and distribution using block chain and machine learning”, in *2022 IEEE 7th International conference for Convergence in Technology (I2CT)*, 2022, pp. 1–6. DOI: 10.1109/I2CT54291.2022.9824112.
- [29] S. Bhalerao, S. Agarwal, S. Borkar, S. Anekar, N. Kulkarni, and S. Bhagwat, “Supply chain management using blockchain”, in *2019 International Conference on Intelligent Sustainable Systems (ICISS)*, 2019, pp. 456–459. DOI: 10.1109/ISS1.2019.8908031.
- [30] C. Fang and W. Zhu-Stone, “An ecosystem for the dairy logistics supply chain with blockchain technology”, in *2021 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*, 2021, pp. 1–6. DOI: 10.1109/ICECCME52200.2021.9591146.
- [31] K. L. Nadime, R. Benabbou, S. Mouatassim, and J. Benhra, “Blockchain-enabled two-echelon supply chains for perishable products using just in time inventory management: A case study of the dairy industry”, in *2023 3rd International Conference on Innovative Research in Applied Science, Engineering and Technology (IRASET)*, 2023, pp. 01–08. DOI: 10.1109/IRASET57153.2023.10152897.
- [32] G. Varavallo, G. Caragnano, F. Bertone, L. Verneti-Prot, and O. Terzo, “Traceability platform based on green blockchain: An application case study in dairy supply chain”, *Sustainability*, vol. 14, no. 6, 2022, ISSN: 2071-1050. DOI: 10.3390/su14063321. [Online]. Available: <https://www.mdpi.com/2071-1050/14/6/3321>.
- [33] F. Casino, V. Kanakaris, T. K. Dasaklis, *et al.*, “Blockchain-based food supply chain traceability: A case study in the dairy sector”, *International Journal of Production Research*, vol. 59, no. 19, pp. 5758–5770, 2021. DOI: 10.1080/00207543.2020.1789238. eprint: <https://doi.org/10.1080/00207543.2020.1789238>. [Online]. Available: <https://doi.org/10.1080/00207543.2020.1789238>.
- [34] F. Melissari, A. Papadakis, D. Chatzitheodorou, *et al.*, “Experiences using ethereum and quorum blockchain smart contracts in dairy production”, *Journal of Sensor and Actuator Networks*, vol. 13, no. 1, 2024, ISSN: 2224-2708. DOI: 10.3390/jsan13010006. [Online]. Available: <https://www.mdpi.com/2224-2708/13/1/6>.
- [35] D. Diener, *Design and implementation of a database-to-blockchain data gathering solution for cheese tracking*, Bachelor’s thesis, Available at <https://files.ifi.uzh.ch/CSG/staff/scheid/extern/theses/BA-D-Diener.pdf>, Binzmühlestrasse 14, CH-8050 Zürich, Switzerland, 2022.
- [36] M. Gamba, *Design and implementation of a sc-based system for the tracking within a cheese supply chain*, Bachelor’s thesis, Available at <https://files.ifi.uzh.ch/CSG/staff/scheid/extern/theses/BA-M-Gamba.pdf>, Binzmühlestrasse 14, CH-8050 Zürich, Switzerland, 2022.
- [37] nhapentor, [Accessed 27-06-2024]. [Online]. Available: <https://gist.github.com/nhapentor/47a0aac5b32e9bb30c36c13adfd2490c>.
- [38] [Accessed 17-07-2024]. [Online]. Available: https://docs.python.org/3/library/time.html#time.perf_counter.

- [39] [Accessed 23-07-2024]. [Online]. Available: <https://developer.chrome.com/docs/lighthouse/performance/performance-scoring>.
- [40] [Accessed 23-07-2024]. [Online]. Available: https://developer.chrome.com/docs/lighthouse/performance/performance-scoring/?utm_source=lighthouse&utm_medium=cli.
- [41] [Accessed 23-07-2024]. [Online]. Available: <https://developer.chrome.com/docs/lighthouse/performance/lighthouse-largest-contentful-paint>.
- [42] [Accessed 23-07-2024]. [Online]. Available: <https://developer.chrome.com/docs/lighthouse/performance/lighthouse-total-blocking-time>.
- [43] [Accessed 23-07-2024]. [Online]. Available: <https://web.dev/articles/optimize-lcp>.

Abbreviations

ABI	Application Binary Interface
AOP	Appellation d'Origine Protégée
API	Application Programming Interface
CPU	Central Processing Units
CTSC	CommunicateToSmartContract.py
DigitalQM	Digital Quality Management System
FCP	First Contentful Paint
HD	HandleData.py
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
JSON	JavaScript Object Notation
LCP	Largest Contentful Paint
MAIN	main.py
OS	Operating System
PDO	Protected Designation of Origin
PoA	Proof of Authority
PoS	Proof of Stake
PoW	Proof of Work
QR	Quick Response
RAM	Random-Access Memory
RDBMS	Relational Database Management System
RPC	Remote Procedure Call
SC	Smart Contract
SCC	Smart Contract Connector
SEO	Search Engine Optimization
TBT	Total Blocking Time
UI	User Interface
URL	Uniform Resource Locator
VCA	Value Chain Actor
VM	Virtual Machine

List of Figures

2.1	Simplified value chain of Tête-de-Moine AOP cheese [6].	6
2.2	Single Block Architecture [4].	6
2.3	Block Chain Data Structure [2], [8].	7
2.4	Merkle Tree Example [8], [9].	8
2.5	Traditional vs. Containerized Applications.	12
3.1	Overview of the CheeseChain project architecture [35], [36]	17
4.1	Application Scenario System 1	22
4.2	Application Scenario System 2	23
4.3	Technical illustration of System 1	25
4.4	Technical illustration of System 2	27
4.5	Deployment of System 1	28
4.6	Deployment of System 2	29
6.2	Plotted Runtimes of System 1 in different Environments.	46
6.3	Plotted Runtimes of System 2 Functions in different Environments.	48
6.4	Merged Plotted Runtimes of System 2 in different Environments.	49
6.5	Plot of the Runtimes for each Configuration over Time.	50
A.1	Network Configuration in MetaMask.	67

List of Tables

3.1	Overview of Related Work	19
6.1	Summary of System 1 Execution Times for Different Environments	47
6.2	Execution Time Summary of System 2 Functions in different Environments	49
6.3	Benchmarks Medians from the Lighthouse Reports.	51
6.4	Performance Medians from the Lighthouse Reports.	51
6.5	Summary of the Ratings.	53
B.1	Answers of Reviewer 1.	71
B.2	Answers of Reviewer 2.	72
B.3	Answers of Reviewer 3.	73

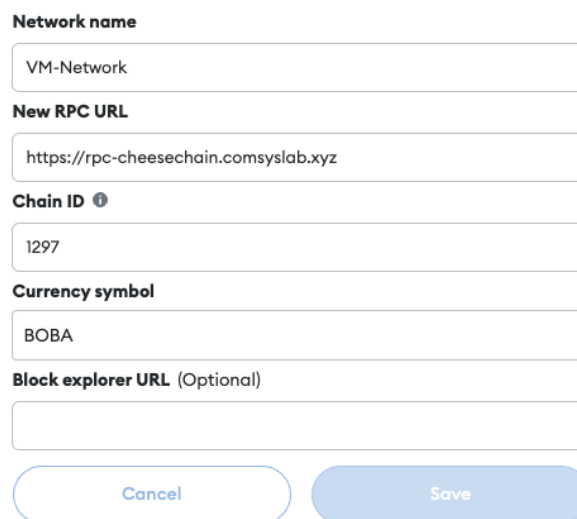
Appendix A

Installation Guidelines

A.1 Private Network

1. Clone the source code
2. Set a custom network ID and Password in the `.env` file.
3. In the `genesis.json` file, add the custom network ID as well and your wallet address under "alloc".
4. Spin the chain up:
`sudo docker compose up --build -d`

After initializing the network, it is ready to be added to the MetaMask account (illustrated in Figure A.1).



The image shows a network configuration form in MetaMask. It contains the following fields and labels:

- Network name**: Input field containing "VM-Network".
- New RPC URL**: Input field containing "https://rpc-cheesechain.comsyslab.xyz".
- Chain ID** (with an information icon): Input field containing "1297".
- Currency symbol**: Input field containing "BOBA".
- Block explorer URL (Optional)**: Empty input field.
- At the bottom, there are two buttons: "Cancel" and "Save".

Figure A.1: Network Configuration in MetaMask.

A.2 System 1

1. Clone the source code
2. Create a Docker Hub account, if necessary, and login:
`sudo docker login`
3. Add your Docker Hub username in the `docker-compose.yml` file
4. Deploy the SC ¹ and fetch a new refresh token ². Then, add these two pieces of information, along with your wallet and the RPC endpoint addresses, to the `Config.json` file.
5. Initialize a Docker swarm:
`sudo docker swarm init`
6. Register the Docker secrets in the swarm:
`echo "<RPC-NODE-ADDRESS>" | sudo docker secret create http_provider -`
`echo "<PRIVATE-KEY>" | sudo docker secret create private_key -`
`echo "<WALLET-PASSWORD>" | sudo docker secret create metamask_password -`
7. Build the image:
`sudo docker build -t <DOCKER-HUB-USERNAME>/system_1:tag .`
8. Publish the image on Docker Hub:
`sudo docker push <DOCKER-HUB-USERNAME>/system_1:tag`
9. Deploy the stack:
`sudo docker stack deploy -c docker-compose.yml mystack`

Following is a list of useful but not required commands:

- `sudo docker secret ls` (list all the created secrets)
- `sudo docker stack services mystack` (see all the services of a stack)
- `sudo docker service logs mystack_system_1` (see the logs of a specific service)
- `sudo docker stack rm mystack` (stop and remove a specific stack)

¹If Remix is used for the deployment, it is recommended to use compiler **0.8.8+commit.dddeac2f**.

²Log in to <https://qs.fromarte.ch> with the developer tools open and copy the refresh token from the network tab.

A.3 System 2

Requirements: NodeJs and MetaMask browser extension installed.

The **Frontend** and **Server** can be spun up at the same time using Docker Compose. The **Chain** part is only needed for local development, in which case it must be spun up first. The installation Guidelines for the **Chain** can be found in the `/public/README.md` file of the repository.

1. Clone the source code
2. Deploy the SC
3. Create `.env` files inside the `/frontend` and `/server` folders, according to the respective `.env.example` files.
4. Make sure the ID of the network that was used to deploy the SC is included in the following files:
 - `/frontend/src/components/ConnectWallet/ChainDisplayer.tsx`
 - `/frontend/src/utils/connectors.ts`
5. Start System 2:
`sudo docker compose up --build -d`

That there are two routes implemented for the **Frontend**:

- **BASE-URL/**: Used to retrieve the lot history via the frontend and is designed for consumers.
- **BASE-URL/connect**: Used registered participants to interact with the smart contract and requires users to connect a Metamask wallet to the **Frontend**. Participants can be added by the system administrator.

Opening the URL of the **Server** in the browser will display some useful GET-commands.

Appendix B

Answers Usability Evaluation

Table B.1: Answers of Reviewer 1.

Task		Answer	Grade
Task 1	Actions	I bought your product and wanted to learn about the production and safety	5
	Remarks	Short and easy information site	
Task 2	Actions	The milk was produced on July 24th, 2024	5
	Remarks	Instant visible	
Task 3	Actions	Binzmühlestr. 14 in Zürich Oerlikon	5
	Remarks	Including Maps is helpful	
Task 4	Actions	July 24th, 2024 Time: 15:56	5
	Remarks	Simple – one click only	
Task 5	Actions	Thanks for the site and its information about the production process	-
	Remarks	-	

Table B.2: Answers of Reviewer 2.

Task		Answer	Grade
Task 1	Actions	I wanted to know more about the product I bought and found the access code on the backside of the cheese. So I accessed the website and found all the required information.	5
	Remarks	-	
Task 2	Actions	After typing in the number of my lot, all the data was perfectly available for me to look at which was what I wanted to achieve. The exact production time and date were visible and transparent for the customer.	5
	Remarks	Since you could only look at one lot, you cannot make a differentiated statement about it	
Task 3	Actions	At every step of the production, a map popped up and showed the exact location of production which I really valued. It was transparent and convenient for me to get such specific information all at once.	5
	Remarks	-	
Task 4	Actions	After opening the production data, you could get more detailed information on each step, which I then did for the second production step. Not only the time and place, but also the role and name of the producer was visible.	5
	Remarks	-	
Task 5	Actions	Lastly, I wanted to know if all the production steps were clean and if my cheese had any issues in quality. The result of the laboratory test is displayed at the top right of the production data. The information I got from it was clearly positive. So I was glad to have that information.	3
	Remarks	It's not so clear if every step of the production passed the laboratory test or only the first one since it's written at the very top. Secondly, no grading is given as to how good or bad the product performed in the test.	

Table B.3: Answers of Reviewer 3.

Task		Answer	Grade
Task 1	Actions	Opened the link provided in the introduction of the questionnaire.	5
	Remarks	Very intuitive as link was provided and the well-known approach for opening a website in a browser can be applied.	
Task 2	Actions	Entered the provided lot number into the input field and pressed on the button labeled “get lot details”.	5
	Remarks	Very intuitive as the input field is labeled and the button which triggers the action is placed nearby. Minor improvement suggestion: use different text in the label above the input field and the button.	
Task 3	Actions	Opened the collapse menu containing the word milk. Looked on the map where the location is. Answer: Universität Zürich-Nord	4
	Remarks	1st step (Opening the collapse menu): Very intuitive as only one entry exists containing the keyword milk. Some thoughts for minor (UX-) improvements: make the modal wider and include a timestamp as all of the entries are from the same date so it is not clear how the ordering is done. 2nd step (finding the exact location): Intuitive as google maps is a widely known and used product. Finding the exact location requires additional actions like zooming and making the map fullscreen. Could be improved by being zoomed in more when opening and providing a direct link to open the location in the google maps application. (Both are improvement suggestions and should be checked against requirements and technical feasibility).	
Task 4	Actions	Open collapse menu titled “second step” Answer: 27.07.2024 15:56	5
	Remarks	Intuitive as required steps have been learned in previous steps. Here again: having the timestamp in the overview would save a click (minor improvement).	
Task 5	Actions	Read the title of the modal. Answer: passed	5
	Remarks	Very intuitive. Maybe it would be a good idea to highlight the title or make it differ from the other entries since it has a different meaning semantically and is also different function-wise (can be opened and collapsed vs. Information display)	

Appendix C

Contents of the CD

As agreed with the supervisor the thesis was submitted as a zip file containing the following:

- `BA-V-Meyer.pdf`, a PDF of the final version of the thesis.
- `Thesis_Source_Code.zip`, the Latex source code of the final version of the thesis in zip format.

Links to the GitHub repositories were also provided.

Attributions

Icons from <https://www.flaticon.com/> were used for some of the figures created and used in this work.