UNIVERSITÉ PARIS 1
PANTHÉON SORBONNE

Doctoral Thesis

A Framework for Semi-automated Design and Implementation of Blockchain Applications

Author:
Nicolas Six

Supervisors:
Pr. Camille Salinesi
Dr. Nicolas Herbaut

A thesis submitted in fulfillment of the requirements for the Doctor of Philosophy

Université Paris 1 Panthéon-Sorbonne
Subject: Information Systems

January 3, 2023

Abstract

Context - Blockchain differs from conventional technologies through its unique characteristics, such as decentralization, immutability, or resiliency. However, in spite of the growing interest in blockchain technology from academia and industry, there are still major obstacles to wide blockchain adoption.

Problem - Blockchain qualities come with several drawbacks, such as a low transaction output, data privacy concerns, and application inflexibility due to smart contract immutability once deployed. Failing to handle these drawbacks might lead to blockchain applications misaligned with initial requirements, high operation costs, high maintenance costs, as well as threats to security and privacy. These issues hinder the integration of blockchain technology into existing or new architectures and systems by practitioners.

Method - The construction of this framework and its artifacts has been made possible by following the Design Science Research (DSR) method for information systems from Hevner et al. Following this approach, the framework is constituted with a knowledge base and two code artifacts. Each refinement on the knowledge base helps to refine the code artifacts, and vice-versa.

Results - This thesis proposes a semi-automated end-to-end framework named Harmonica for the design and implementation of blockchain applications. This thesis presents three original contributions. First, a knowledge base to support the recommendation process. To constitute the core of the knowledge base, a systematic literature review was performed to identify, extract, then standardize existing blockchain-based software patterns. The knowledge base is stored as an ontology, that contains the attributes and relations of identified blockchain-based software patterns and blockchains. Second, an automated decision process to recommend a blockchain technology and blockchain-based patterns in a given context. Given a set of requirements, the decision process is able to output a ranking of blockchain technologies to help the user in the selection of an adequate technology. Third, a tool capable of reusing the recommendations to generate a functioning and complete blockchain application. These parts are both related to the implementation and deployment of the blockchain application: a set of smart contracts is generated and augmented with selected blockchain-based software patterns, and deployment scripts are proposed to support the deployment of smart contracts on the target blockchain.

Conclusion - The combination of produced artifacts form a toolkit that facilitates the process of creating blockchain-based applications, from their design to their implementation.

The proposed tools can be used independantly from each other to support a specific activity of blockchain-based software development, or together as they each profit from each other's output. Each part of the framework has been validated independently using case studies and user surveys to ensure they adequately support the different steps of software development, from conception to implementation.

Contents

List of Figures

List of Tables

Acronyms

AHP Analytical Hierarchy Process

BANCO Blockchain ApplicatioN Configurator

BLADE BLockchain Automated DEcision process

BOSE Blockchain-Oriented Software Engineering

BPMN Business Process Model and Notation

CML Contract Modeling Language

CQ Competency Question

DLT Distributed Ledger Technology

DSR Design Science Research

ELECTRE ELimination Et Choix Traduisant la REalité

ERC Ethereum Request for Comment

EVM Ethereum Virtual Machine

FLR Focused Literature Review

GDPR General Data Protection Regulation

HACCP Hazard Analysis and Critical Control Point

Harmonica BlockcHain fRaMewOrk for the desigN and Implementation of deCentralized Application

IPFS InterPlanetary File System

MCDM Multi-Criteria Decision Making

MDE Model-Driven Engineering

NeOn Network of Ontologies

NFR Non-Functional Requirement

ORSD Ontology Requirement Specification Document

PoS Proof-of-Stake

PoW Proof-of-Work

QQ	Quality Question
RO	Research Objective
SCME	Single-Case Mechanism Experiment
SLR	Systematic Literature Review
SPL	Software Product Line
SPLE	Software Product Line Engineering
SWRL	Semantic Web Rule Language
TBCG	Template-Based Code Generation
TOPSIS	Technique for Order Preference by Similarity to the Ideal Solution
UML	Unified Modeling Language
UTAUT	Unified Theory of Acceptance and Use of Technology

Chapter 1

Introduction

1.1 Research Context

A blockchain technology is a distributed ledger constituted of blocks, supported by a network of peers each owning a copy. Every node follows the same protocol and uses a consensus algorithm to keep its copy consistent with others. Users can interact with nodes to append transactions, but their modification and deletion are theoretically impossible. While the first generation of blockchains was only focusing on cryptocurrency transactions between users, such as Bitcoin (Nakamoto, 2008), some of them now support smart contracts, such as Ethereum (Buterin et al., 2013). A smart contract is a decentralized program that can be executed on-chain, through nodes. Users can deploy and interact with smart contracts using transactions.

Blockchain is fully decentralized by nature, where no third party is in charge of the network. Blockchain data are also immutable and tamper-proof, as nobody can alter a block after its creation and addition, into a blockchain. Thanks to these properties, blockchain-based applications can be trusted, as nobody can tamper with the correct execution of a smart contract[1]. Also, it is possible to retrace the state change history of a blockchain. Thus, smart contract state changes can also be replayed for the complete traceability of decentralized applications (dApps).

In recent years, blockchain has been growing rapidly from a niche technology used by a few people as a promising solution for many sectors. According to Gartner, the business value created by blockchain technologies might reach $3.1 trillion (ConsenSys, 2019). This growth is due to its unique properties that empower the design of innovative software architectures and systems (Zeadally and Abdo, 2019). First, due to the native support of cryptocurrencies, blockchain enables the creation or the improvement of use cases in the financial domain that was difficult to leverage using existing technologies. For example, currency exchange through banks can be an expensive process for a consumer, Automated Market Makers (AMM) allow the swap from one cryptocurrency to another without any intermediate using liquidity pools of cryptocurrencies and a smart contract to perform the swap

[1]This is only valid if the smart contract is well designed to prevent execution flaws and security issues.

(Pourpouneh, Nielsen, and Ross, 2020). Regarding insurance, blockchain can be used to automate the claiming process in case of an accident. While such a process takes many days or weeks with traditional insurance systems (Oham et al., 2018), it can be automated using smart contracts.

Blockchain also has many applications in nonfinancial domains, due to its capacity to operate without any third party and enable trust with the usage of decentralized applications. For instance, blockchain can be the platform in an inter-organizational business process, to monitor organizations' actions and data, or to allow the business process execution directly on-chain (Di Ciccio et al., 2019; Herbaut and Negru, 2017; Udokwu et al., 2021). In this context, participants can trust the information stored by the blockchain, and operations performed in smart contracts cannot be tampered with. This layer of decentralized automation and trust is also used in other applications, such as smart grids (Agung and Handayani, 2020), as blockchain can connect thousands of individuals to enable a market for energy exchange between users, or healthcare for medical records sharing.

As blockchain use cases are increasingly considered, many companies start to show interest in blockchain and start building new applications. According to Deloitte's 2020 Global blockchain Survey, 55% of the 1488 surveyed companies across the world considers blockchain as one of their top-five strategic priorities (Deloitte, 2020).

But despite the growing interest from companies towards blockchain, there is no widespread adoption of the technology yet. In an article from Prewett et al., adoption issues related to blockchain technologies are mentioned (Prewett, Prescott, and Phillips, 2020). From a legal standpoint, a concerning aspect is the lack of a regulatory framework. Blockchain growth has been exponential in the last few years, outpacing the development of regulations. Unfortunately, this issue has been exploited by malicious actors in different scams (Zetzsche et al., 2017). The lack of regulation also brings uncertainty when designing a blockchain application. For instance, some applications use smart contracts to encode legally binding data, such as signatures or smart contract obligations between two or more parties. As mentioned in an article from Gilcrest et al., some jurisdictions already recognize these bindings, but there is still no wide recognition (Gilcrest and Carvalho, 2018). Another example of uncertainty is the trust associated with on-chain data. Blockchain applications might also not comply with existing regulations. For example, storing data on the blockchain might conflict with General Data Protection Regulation (GDPR), as it is impossible in most cases for a user to enforce his right to data deletion (Humbeeck, 2019).

From an organizational perspective, the adoption of blockchain is hindered by a lack of blockchain knowledge or skills among practitioners but also stakeholders and users (Prewett, Prescott, and Phillips, 2020). As mentioned by Rupino et al., mass adoption of blockchain technologies requires blockchain-educated people and developers (Cunha, Soja, and Themistocleous, 2021a). This education goes through personal education and training, obtaining certifications, and gaining awareness of the ecosystem. People and organizations will also have to innovate by creating new applications while complying with existing regulations. Finally, new applications will have to be simple to use and user-centered. From a technical standpoint, they will also have to facilitate operability and comply with existing standards. This facilitates the onboarding of new users to blockchain applications, and the interoperability between applications, leading to a composable ecosystem of applications. However, there is a shortage of people with knowledge of blockchain technology, particularly on technical profiles (Ben, 2022). Due to the novelty of the technology, it might be difficult to find talents in this space. The governance of blockchain applications can also be a threat to adoption. For instance, companies interested in a common goal might form a consortium to run and administrate a blockchain network. In this context, they will have to face many questions, such as: who can add data into the blockchain? Who can include new participants in the consortium? Where will the nodes be hosted (e.g. on-premise, cloud service, ...)?

Technical issues are the third type of issue that can occur in blockchain adoption. Indeed, practitioners might face many technical issues and questions along the software engineering process, from the design to deployment in production. This is notably due to the inherent difficulties of the software development process, which are even greater when using a nascent technology such as blockchain. These technical issues will be the focus of concern for this Ph.D. thesis.

During the software design phase, software architects face many choices, such as the selection of an adequate blockchain for their needs (Xu, Weber, and Staples, 2019). This task is far from being straightforward, as many blockchain technologies exist with different specifications and purposes (Belotti et al., 2019). In some cases, using a blockchain technology might be unnecessary or even incompatible with the application to build. Indeed, deciding about using a blockchain technology is not a trivial task: companies might exaggerate the advantages of using blockchain, compared to the actual needs of a specific domain (Ribalta et al., 2021).

Regarding software implementation, developers have to tackle programming paradigms that differ from traditional software engineering. For instance, where off-chain services can easily query data to other services, on-chain smart-contract may have to rely on events to request data from outside the blockchain (so-called oracles (Xu et al., 2018)). Another example is the immutability of blockchain smart contracts once deployed on-chain (Khan et al., 2021). This is not the case for traditional software engineering where an update can be shipped on existing software. Developing a blockchain-based application without prior expertise in the technology might lead to inefficient code or even critical vulnerabilities. This issue is exacerbated as smart contracts may hold and manipulate substantial amounts of cryptocurrencies or act as trusted data storage. To mention an example, a common smart contract vulnerability of *The DAO*, a decentralized autonomous organization on the Ethereum mainnet[2], has led to a loss of $50M USD worth in Ethers at that time (Mehar et al., 2019). Scalability may also be an issue for many applications: as mentioned by Rupino et al., scalability of applications result in high fees and latency problems (Cunha, Soja, and Themistocleous, 2021a). Along that, interoperability can also be mentioned: it may be difficult to design composable

[2]The Ethereum mainnet is the most-known public version of the Ethereum blockchain.

applications. There is an increased need for blockchains to interact with each other (Cunha, Soja, and Themistocleous, 2021a), and although initiatives exist to address this issue (e.g. Polkadot[3], Cosmos[4], etc.), interoperability remains a challenge for developers.

Software patterns are usually one solution in the toolbox of software developers to help them in their work of developing robust and efficient applications. A pattern can be viewed as a possible solution for a recurring problem in a given context (Alexander, 1977). The usage of blockchain-oriented patterns could be a solution for blockchain developers to guide them in the specific issues of implementing a blockchain-based application. However, as blockchain-based software development is a relatively young field, only a few patterns were proposed by practitioners and researchers. Indeed, formalizing a solution into a pattern often requires having already applied the solution successfully in several projects. Another issue in using blockchain-based patterns is the difficulty for developers to find patterns and evaluate their suitability. Patterns are scattered across academic literature or technical repositories and can be hard to understand without prior experience with blockchain technologies.

Finally, launching a blockchain application in production might also be a tedious task. Depending on the requirements, a private blockchain network or a public blockchain node might have to be set up, requiring knowledge to configure it. Ready-to-use solutions can be used instead, but lead to a vendor lock-in (Lu et al., 2019). Along that, a deployment script or a framework is often used to deploy the smart-contracts on-chain (e.g. Truffle framework[5]), also requiring a configuration file to work.

All of the different steps within the software development process are impacted when using blockchain technology. Despite its potential, blockchain adoption is still partly hindered by complex open issues.

1.2 Research Aim and Objectives

To guide this research, a research aim and subsequent Research Objectives (ROs) were defined. They have been defined in the context of the Design Science Research (DSR) method, that consists in iterating over two activities: designing an artifact that improves something for stakeholders and empirically investigating the performance of an artifact in a context (Wieringa, 2014). This method is introduced in detail in Chapter 3.

First, the research aim of this thesis can be defined as follows, using Wieringaś design science research aim template:

Improve the software engineering process for creating blockchain-based applications by designing a semi-automated framework composed of two assisting tools and a knowledge base that assists the practitioner along the tasks of designing and implementing

[3]https://polkadot.network/
[4]https://cosmos.network/
[5]https://trufflesuite.com/

blockchain-based applications so that makes the creation of blockchain-based applications easier, and reduces development costs.

In this research aim, the practitioner is the main stakeholder of this framework and includes all people involved in the design and implementation of blockchain-based applications (e.g. software engineer, software developer, software architect, etc.). The framework proposed as a solution in this research aim attempts to address the aforementioned issues, in the context of already existing solutions. These issues and solutions are further identified and described in Chapter 2 Systematic Literature Review (SLR).

As this research aim defines the broad objective of this research, it can be further broken down into multiple ROs:

RO1 - *Design an assisting tool to guide the practitioner in the selection of blockchain technologies.*

The selection of a blockchain technology is probably the first technical blockchain-related choice practitioners have to make. This choice deeply impacts the final software. For instance, the choice between a public or a private blockchain. The former allows greater transparency of data and decentralization through open access to the blockchain network, the latter can be more suited when participants should be approved prior to any participation and data should be kept confidential. In this context, the following questions can be considered: how to translate the user requirements into actionable knowledge for the selection of a blockchain platform? What features can describe blockchain enough to make accurate comparisons with other blockchain technologies? What tools and algorithms can be leveraged to make a recommendation based on user inputs and blockchain knowledge? And finally, how can the relevance of the recommendation made by such a tool be validated?

RO2 - *Gather and classify existing blockchain-based software patterns in the literature.*

Reusing existing software artifacts is a common practice in software engineering. For instance, developers are used to copying code from online sources (e.g. Stack Overflow[6]). This practice is called "clone-and-own".

Another common practice is the usage of software patterns in the design and implementation of blockchain applications. Software patterns are a great asset to assist practitioners to design robust, efficient, and secure applications. However, it is still difficult to use these assets in the design of a blockchain application, as patterns are still scattered and expressed in non-standard formats. Even identified, patterns can still be difficult to apply as it requires knowledge in blockchain technology in most cases. This research objective raises the following challenge. First, how to collect existing patterns across existing sources? Then, how to uniformize and classify patterns to form a collection that is complete and usable enough to be used by practitioners?

RO3 - *Design a blockchain-based pattern ontology with adequate tooling for the selection of blockchain-based patterns.*

[6]https://stackoverflow.com/

Gathering patterns into a structured collection is the first step to ease their reuse among practitioners. Yet, it may be difficult for non-experts to use the collection, even if the patterns are classified. The goal of this research objective is to refine such collection into an ontology. This approach has multiple advantages. First, it allows the creation of new knowledge through inference using the ontology concepts and a set of rules. For instance, new relations between patterns may be identified using the existing ones. Then, the ontology can be reused to be exposed by a web platform. It allows practitioners to easily explore the patterns stored within the ontology as well as their relations. Finally, the web platform may be able to reuse the ontology to compute pattern recommendations. Instead of only exposing patterns to the practitioners, the platform may be able to recommend patterns depending on the user requirements and knowledge contained in the ontology.

RO4 - *Design a blockchain application configurator and generator that reuses existing code stubs and design decisions.*

Software development process often results in the creation of multiple artifacts: code files that can be compiled or interpreted and then deployed, and configuration files to set up the infrastructure receiving the application or the deployment pipeline. There is no exception for blockchain. Therefore, automating the generation of those files could ease the development and deployment of a blockchain application.

Generating code from existing models is an open research challenge. Different models were used to model then generate blockchain applications, such as Petri nets (Zupan et al., 2020) or Business Process Model and Notation (BPMN) (Lòpez-Pintado et al., 2019). Generating code from models also ensures that the resulting code will not diverge from the underlying models. It also enhances the code quality compared to manual development, portability as the language target can be changed, and maintainability (Hutchinson, Whittle, and Rouncefield, 2014). However, the completeness of generated code often depends on the completeness of the model itself.

Besides Model-Driven Engineering (MDE), another existing approach for the generation of code is Software Product Line (SPL) engineering (Pohl, Böckle, and Van Der Linden, 2005). The main principle of SPL engineering stands in the reuse of existing requirements, models, code, and components created for this purpose. Taking code and components as an example, an application can be assembled by merging multiple core assets. To define the possible combinations, a variability model is also defined. It indicates what combinations are possible, possible dependencies, and conflicts between two or more code artifacts. On the opposite of MDE, SPL can generate complete applications from on-the-shelf components but is tied to the components library already created. There is also an overhead cost of using an SPL approach, as it requires creating the variability model and the components prior to generating any application.

In the context of generating blockchain applications, it raises many questions. Which method is the most suitable for this goal? Choosing an MDE approach, which models can be used, independently or together, to derive code stubs? Regarding the SPL approach, what components are required to build blockchain applications for a specific application domain?

In this thesis, these research objectives were first expressed as technical or knowledge research questions (Chapter 3). The former consists of addressing a design problem while the latter consists in asking for knowledge about the world without calling for an improvement (Wieringa, 2014). Then, each question was answered in a dedicated chapter, following the methods proposed by Wieringa to either answer technical or knowledge research questions.

1.3 Thesis Contribution and Publications

Throughout this thesis, four contributions constituting the different framework parts are made:

(i) A knowledge base of 114 unique blockchain-based software patterns, organized as an ontology. These patterns have been first collected throughout the literature, by performing a SLR following Kitchenham et al. guidelines (Kitchenham and Charters, 2007). The main objective was to identify and describe the existing blockchain-based software patterns in the literature. A taxonomy has also been constructed empirically for its reuse in the ontology to classify the patterns into comprehensive categories, using pattern descriptions. Another objective of the SLR was to identify gaps in the state-of-the-art of blockchain-based patterns research. It has been found that the majority of identified studies were proposing design patterns, exclusively for Solidity[7], a programming language from the Ethereum ecosystem. More research is needed to target other blockchain technologies as well as architectural patterns or idioms.

(ii) A decision-making tool for the selection of an adequate blockchain technology, part of BLockchain Automated DEcision process (BLADE). To get a recommendation, a user has to specify Non-Functional Requirement (NFR) on the platform. NFRs are specified as: (1) a preference level used to weight requirements in the goal to plan their implementation, (2) a boolean indicating if the requirement is mandatory, and (3) a threshold value to satisfy. A multi-criteria decision support algorithm process the inputs to generate the recommendation, named Technique for Order Preference by Similarity to the Ideal Solution (TOPSIS). To facilitate the submission of NFR, some guidance of potentially conflicting requirements is presented to the user at selection time. A dependency model is leveraged to compute conflicting requirements for each selection made on the platform.

(iii) A library and recommender within BLADE for the selection of adequate blockchain-based software patterns, leveraging the aforementioned knowledge base. Using the library, a user can fetch the available blockchain-based patterns and filter on different parameters (e.g. blockchain, pattern type, ...). Any output from the blockchain selection phase in BLADE further guides the user by only indicating compatible patterns with the selected blockchain when fetching patterns. Where the library only allows a manual selection of patterns, the

[7]https://docs.soliditylang.org/en/latest/

recommender automatically proposes a collection of compatible patterns to fulfill user requirements. By answering a set of questions, a user can get a set of blockchain-based patterns recommended fitting its requirements. These questions have been formulated in order to map the answer to one of the taxonomy categories, as each of them groups several patterns.

(iv) A software product line named Blockchain ApplicatioN Configurator (BANCO) for the configuration and the generation of a blockchain product. First, a feature model has been designed to model core features of a chosen domain, that is on-chain traceability, based on the existing literature. Then, a configurator has been implemented to support the feature selection phase. The configurator also handles possible conflicts between features during the selection using a constraint engine. Finally, a generator is able to ingest such configurations to generate on-the-shelf blockchain products. This generator is based on template code generation and performs by assembling templates of functions and smart contracts based on the previously defined configuration.

These contributions have also been published, or in the process of being published in the following peer-reviewed publications:

1. (Six, Herbaut, and Salinesi, 2021b) Six, N., Herbaut, N., & Salinesi, C. "Harmonica: A Framework for Semi-automated Design and Implementation of blockchain Applications." INSIGHT 24.4 (2021): 25-27.
2. (Six, Herbaut, and Salinesi, 2020) Six, N., Herbaut, N., & Salinesi, C. (2020). Quelle blockchain choisir? Un outil d'aide à la décision pour guider le choix de technologie blockchain. In *INFORSID 2020* (pp. 135-150).
3. (Six, 2021) Six, N.. "Decision process for blockchain architectures based on requirements." CAISE Doctoral Consortium (2021).
4. (Six, Herbaut, and Salinesi, 2021a) Six, N., Herbaut, N., & Salinesi, C. (2020) BLADE: Un outil d'aide à la décision automatique pour guider le choix de technologie blockchain. Revue ouverte d'ingénierie des systèmes d'information 2.1 (2021).
5. (Six, Herbaut, and Salinesi, 2022) Six, N., Herbaut, N., & Salinesi, C. (2022). Blockchain software patterns for the design of decentralized applications: A systematic literature review. *blockchain: Research and Applications*.
6. Six, N., Correa-Restrepo C., Herbaut, N., & Salinesi, C. (2022). An ontology for software patterns: application to blockchain-based software development *Accepted for publication at EDOC'22 - Forum*.
7. (Six et al., 2022) Six, N., Herbaut, N., Lopez-Herrejon, R. E., & Salinesi, C. (2022). Using Software Product Lines to Create blockchain Products: Application to Supply Chain Traceability. In *26th ACM International Systems and Software Product Lines Conference*.

In this list, publication n°1 was reused in the writing of Chapter 3, the publication n°2, 3 and 4 in Chapter 4, the publication n°5 in Chapter 5, the publication n°6 in Chapter 6, and the publication n°7 in Chapter 7. In parallel, two contributions were made in the domain of blockchain technologies. The former proposes a blockchain-based design pattern that enables the creation of business processes for legal contract execution based on blockchain smart contracts, whereas the latter proposes an approach for collaborative AI using blockchain as a marketplace.

- (Six et al., 2020) Six, N., Negri-Ribalta C., Herbaut, N., & Salinesi, C. "A blockchain-based pattern for confidential and pseudo-anonymous contract enforcement." 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). IEEE, 2020.
- (Six, Perrichon-Chrétien, and Herbaut, 2021) Six, N., Perrichon-Chrétien, A., & Herbaut, N. "SAIaaS: A blockchain-based solution for secure artificial intelligence as-a-Service." The International Conference on Deep Learning, Big Data and blockchain. Springer, Cham, 2021.

1.4 Thesis Organization

This thesis is organized around the Design Science Research method, that is a research approach that aims to create, validate, and disseminate innovative solutions to practical problems through the development and application of design principles and methods. It notably makes the division between artifacts and knowledge. Using the DSR approach, an artifact is the result of addressing a design problem, where knowledge is the result of answering knowledge questions (Wieringa, 2014) This method is further introduced in Chapter 3.

The framework proposed in this thesis can be divided following the DSR approach. It is composed of two tools: BLADE, the blockchain recommendation engine, and BANCO, the blockchain application configurator and generator. BLADE is further divided into two parts: the blockchain technology recommender, and the blockchain-based software patterns recommender. Together, BLADE and BANCO form a set of three artifacts. Along that, a knowledge base was built to support the recommendation process of BLADE. This knowledge was also obtained following the DSR approach.

As such, the organization of the thesis can be illustrated (Figure 1.1).

First, preliminary content is given in Chapter 2 and 3. In Chapter 2 background on blockchain technologies and software patterns is introduced. Along with that, a literature review is carried out to discover existing issues when designing and implementing blockchain applications. The result of this literature review directly guides the design of the framework proposed in this thesis. Then, Chapter 3 introduces the contribution envisioned as well as the research method, that is DSR. In this chapter, the reader can learn what is built to address the issues mentioned in the literature review, but also how it is built. Along that, a

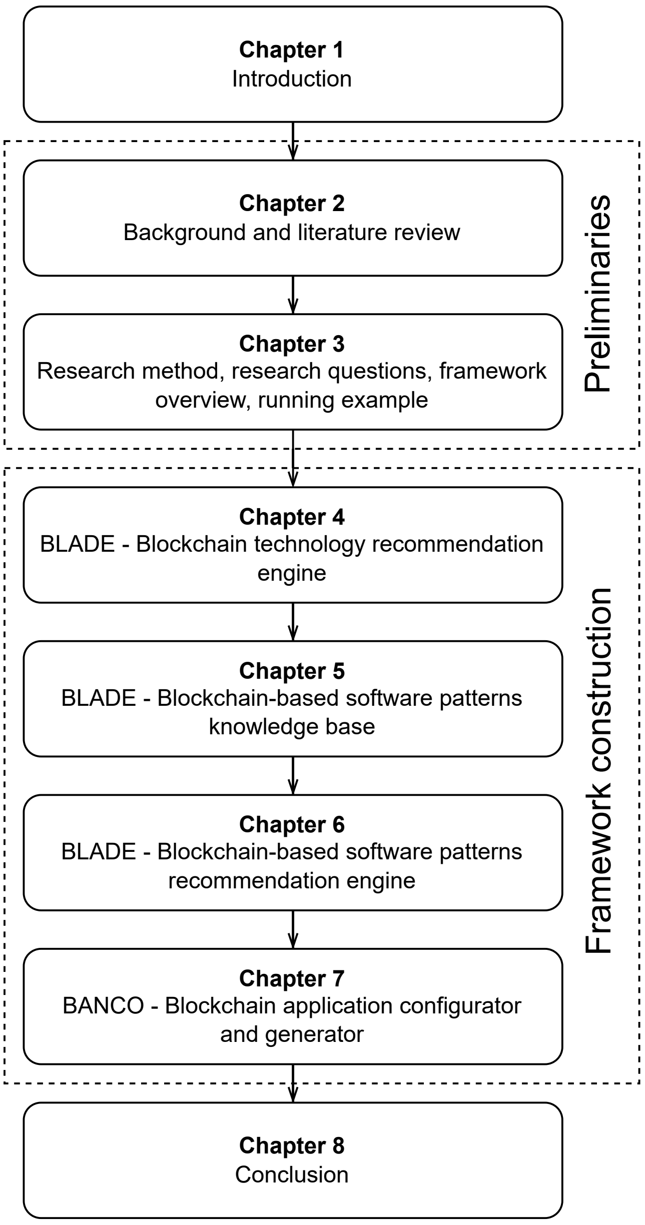

Chapter 1
Introduction
Chapter 2
Background and literature review
Chapter 3
Research method, research questions, framework overview, running example
Preliminaries
Chapter 4
BLADE - Blockchain technology recommendation engine
Chapter 5
BLADE - Blockchain-based software patterns knowledge base
Chapter 6
BLADE - Blockchain-based software patterns recommendation engine
Chapter 7
BANCO - Blockchain application configurator and generator
Framework construction
Chapter 8
Conclusion

Figure 1.1: Thesis organization.

running example is described, that is a case study from the literature on building a blockchain traceability application (Cocco et al., 2021). In each chapter presenting an artifact, a section is dedicated to illustrating the artifact using the running example as a practical case.

Following the preliminaries, Chapter 4, 5, 6, 7 introduces the construction of the different artifacts and the knowledge base. Each chapter individually introduces one contribution (either an artifact or the knowledge base). In the end, the resulting artifacts and the knowledge base sum up as the envisioned framework.

Chapter 4 introduces the first part of BLADE, that allows the recommendation of a blockchain technology depending on non-functional requirements. Then, Chapter 5 describes the process of systematically collecting blockchain-based patterns across the literature, as well as the result, that is a collection of blockchain-based patterns. Chapter 6 reuses this result to form an ontology of blockchain-based software patterns. It also introduces the second part of BLADE, that facilitates the selection of adequate patterns for the practitioner, throughout a web interface and a recommendation system. Chapter 7 introduces BANCO, a tool to configure and then generate a working blockchain application, based on software product line engineering.

Finally, Chapter 8 concludes the thesis and discusses future works. Along with their content, each chapter begins with a series of bullet points that summarized its content. These bullet points act as reminders of the purpose of each chapter and their respective contributions.

Chapter 2

Background

Key takeaways

- A background on blockchain technologies and software patterns is given.
- A focused literature review is presented, that identifies and classifies existing issues met by practitioners during the design and implementation of blockchain applications.
- Main issues that were identified help in defining the goals of the framework presented in this thesis, and allow formulating an answer to the knowledge research question RQ1.

From the exponential growth of blockchain, many developers shifted from traditional software engineering to Blockchain-Oriented Software Engineering (Porru et al., 2017). As they shift, they have to learn a broad range of new concepts, such as distributed systems, cryptography, but also smart contract development and languages (e.g. Solidity), as well as specific blockchain technologies (e.g. Ethereum). This overwhelming amount of knowledge may be difficult to grasp, while the lack of knowledge in the design and implementation of blockchain applications may have severe consequences, such as performance issues, exponential costs, and security threats.

Chapter 1 states that the research aim of this thesis is to improve the software engineering process for creating blockchain-based applications. Yet, it is difficult to propose a solution without knowing precisely what are the issues that might be met by developers along with this process. Although many practitioners and researchers may have some insights on such issues through practice and feedback, it is necessary to gather these issues to understand possible improvements.

This chapter starts with an extensive background on blockchain technologies as well as software patterns. This material will notably be useful for the reader to understand the following chapters. Then, a literature review is carried out to explore the existing literature on designing and implementing blockchain applications. The goal of this literature review is to answer the first research question (RQ1) of this thesis, that is stated as follows: *What are the existing issues faced by developers in the design and implementation of blockchain-based applications?*

The chapter is organized as follows: Section 2.1 introduces background and definitions on blockchain technologies and software patterns, then Section 2.2 presents the literature review: its protocol, its results, and a discussion of potential solutions in light of identified issues. Finally, Section 2.3 concludes the paper.

2.1 Definitions

2.1.1 Blockchain Technology

The first implementation of blockchain technology has been proposed in 2008 when Satoshi Nakamoto released a whitepaper on Bitcoin, a decentralized cryptocurrency (Nakamoto, 2008). He combined several existing technologies, such as asymmetric encryption (Simmons, 1979), Merkle tree structures (Merkle, 1989), consensus methods (Mingxiao et al., 2017), and Hashcash, a cryptographic algorithm where computing the proof is difficult and verifying it is a simple task (Back et al., 2002). This combination has defined the foundation of blockchain technologies.

According to Belotti et al., a possible definition of blockchain can be given as follows (Definition 1):

Definition 1 *A blockchain is an immutable read-only data structure, where new entries (blocks) get appended onto the end of the ledger by linkage to the previous block's hash identifier (Belotti et al., 2019).*

Usually, blocks contain a record of transactions (Definition 2). Their type depends on the blockchain usage: for Bitcoin, transactions represent an exchange of cryptocurrency between users.

Definition 2 *Transactions are individual and indivisible operations that involve the exchange or transfer of digital assets. The latter can be information, goods, services, funds or a set of rules which can trigger another transaction (Belotti et al., 2019).*

The blockchain as a data structure is maintained by a network of peers. Each member of the network owns a copy of the blockchain. They communicate using the same protocol to maintain their copy up to date. To do that, each blockchain protocol comes with its consensus algorithm (Definition 3):

Definition 3 *A consensus algorithm is a protocol or set of rules that allow a collection of machines to work as a coherent group that can survive the failures of some of its members (Ongaro and Ousterhout, 2014).*

As the different nodes of the blockchain network have to synchronize to maintain consistency on their copy of the blockchain, they use a consensus algorithm for coordination. In public blockchains, these algorithms are also responsible for avoiding node misbehaving such as double spending attacks (Chohan, 2021). To mention a few of them, the Proof-of-Work algorithm is based on a mathematical challenge that nodes have to solve for appending a block to the blockchain (Gervais et al., 2016). If so, they can share the block with others and start searching for a solution for the next block. Using the Proof-of-Stake algorithm, participants have to put collateral at stake to be entitled to create and share blocks[1]. The size of the collateral determines the share of the blocks it has the right to create (Saleh, 2021). Misbehaving nodes are punished by taking out their stakes. Another notable algorithm is the Practical Byzantine Fault-Tolerant (PBFT) algorithm, that tolerates Byzantine faults (e.g., dysfunctional or malicious nodes) in the network (Sukhwani et al., 2017). A leader, once elected, is responsible for broadcasting new transactions from clients to backup nodes that verify the transaction, execute the required operations, and then propagate the transaction. If enough backup nodes agree on the same result, the transaction is appended to the blockchain.

For the first two consensus algorithms, participants have to put at stake something with real-world value: either computing power or cryptocurrencies. However, for the third one, there is nothing at stake: the only solution for a secure network is to know the participants and exclude them in case of misbehavior. Depending on the consensus algorithm used, blockchain networks can either allow anybody to join and participate or require approval from others to join, called respectively public and private blockchains. Selecting the right blockchain for a given context is a tough choice. Public blockchains are more decentralized than private ones in general, as anybody can join and participate. However, their consensus algorithms are less efficient than private blockchains ones as there is no implicit trust in network participants, thus they have to prevent participants from misbehaving. On the contrary, private blockchains are more efficient but often controlled by a group of organizations. For example, Bitcoin can only process 6 transactions per second using a Proof-of-Work algorithm, whereas Hyperledger Fabric with PBFT can process hundreds of transactions per second.

Blockchain can be used to transact cryptocurrencies, but also leverage decentralized applications, throughout the usage of smart contracts. The concept of smart contract has been proposed by Wang et al. as the following (4):

Definition 4 *Smart contracts are self-executing contracts with the terms of the agreement between interested parties. The contracts are written in the form of program codes that exist across a distributed, decentralized blockchain network. Smart contracts allow transactions to be conducted between anonymous or untrusted parties without the need for a central authority (Wang et al., 2018).*

Following this definition, blockchain technologies are good candidates to support the execution of smart contracts. Indeed, it is impossible to alter the result of executed functions, unless a vulnerability or a design flaw exists, or if the blockchain network itself is compromised. As such, the first proposal of blockchain smart contracts traced back to 2015 with Ethereum (Buterin et al., 2013), and generalized to many blockchains since then. In the

[1]The term minting is often employed in the context of PoS-based blockchains for this operation.

remaining chapters, the term blockchain smart contracts will be summarized as smart contracts.

Executing a smart contract function follows the same process as adding a transaction into the blockchain: the function is performed by the requested node, and the result is shared with the other nodes that will also verify the correctness of the function execution. When building decentralized applications, the off-chain part can be differentiated from the on-chain part. The on-chain part is usually constituted by smart contracts, and the off-chain part is composed of components that are not part of the network but might interact with it. This distinction is important as it constitutes a separation between patterns in the taxonomy presented in Section 5.2.1.

Through its specific behavior, blockchain technology has many interesting properties (Wust and Gervais, 2018):

- Decentralization - no one is in charge of the whole network. By extension, smart contract-based apps are also decentralized, as no third party is responsible for executing its functions and returning the result to others.
- Transparency - every network participant can dive into the content of the blockchain, either transactions or smart contracts data.
- Tamper-proofing and immutability - it is impossible to modify the content of a block after its addition. It would be detected by others because the hash of the block would change and mismatch the block hash already stored in the next block.

However, blockchain qualities can also be liabilities, depending on the context:

- Data leakage risk - the transparency and immutability of a blockchain can put personal or confidential data at risk. Even if encrypted, it is unsure that data is safe, because of potential advances in data decryption or key leakage.
- Immutability threats - immutability of blockchain also implies the impossibility to reverse transactions, even if they are harmful. As an example, a vulnerability exploited in TheDAO smart contract has led to a loss of 12 million $USD in Ether, the network cryptocurrency (David, 2016).
- Performance issues - poor performance of some blockchains may also be a burden when low latency or high throughput are expected. Performance issues are mostly due to bottlenecks related to peer-to-peer mechanisms within the blockchain network (e.g. consensus, peer discovery, etc.) (Fan et al., 2020).

Thus, any company that wants to use blockchains in their applications should carefully assess the implications, as this is not always the best solution (Wust and Gervais, 2018). Software patterns can help to lower the impact of blockchain liabilities on the final design, to guide the design of blockchain applications through repeatable solutions, or to ensure that blockchain qualities are kept intact in the final design. However, there is still a lack of a wide structured collection of software patterns for blockchain. Chapter 5 and 6 will notably address several research questions aiming towards this goal. A background on software patterns is also given in the next subsection.

2.1.2 Software Patterns

In the software engineering field, patterns are strong assets for engineers and architects to design robust and well-designed applications. The principle of patterns was first proposed by Christopher Alexander in the construction field, as he proposed to document architecture designs in a way that documentation can be reused for other buildings (Alexander et al., 1979). A definition of patterns was given in Alexander's book, commonly reused later by other researchers, that is the following (5):

Definition 5 *Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice (Alexander, 1977).*

In the software engineering field, patterns appeared later in 1987 when Cunningham et al. decided to apply the pattern approach to guide developers using Smalltalk, an object-oriented language (Beck, 1987). Later on, 4 researchers (commonly called the GoF - Gang of Four) released a book that defines a collection of design patterns for the development of object-oriented applications (Gamma et al., 1995). Since then, many researchers have proposed software patterns for many use cases, such as microservices (Taibi, Lenarduzzi, and Pahl, 2018) and Internet-of-Things (IoT) (Qanbari et al., 2016).

Patterns can be grouped into three categories: architectural patterns, design patterns, and idioms:

- Architectural patterns - define, at the highest level of abstraction, the general structure of the application (elements, connections).
- Design patterns - define a way to organize modules, classes, or components to solve a problem.
- Idioms - solutions to language-related problems at the code level.

Using patterns in an application brings many advantages. First, as existing patterns are often extensively tested and applied by others, they can be reused in a new design as the best solution possible for a given case. They also define a common language among developers, as software patterns are defined with a meaningful name. However, their application should not always be systematic: applying the wrong pattern to a certain design can be more harmful than helpful. They might also increase the complexity of software. As an example, the *Proxy* pattern, that helps to control the access to an object is unnecessary if the object in question is not sensitive and only accessed by one other object.

To be easily reused, software patterns are often expressed using a pattern template. The two most commonly used pattern templates are the form proposed by the GoF (GoF pattern format), and the Alexandrian form, by Christopher Alexander (Tešanovic, 2005). In both approaches, a pattern is described by an expressive **Name**, the **Context** it is applicable to, and a recurring **Problem**. The Alexandrian form is also constituted by the following: the **Solution** to describe the pattern, the **Forces** where the pattern has an impact on, **Examples** of application, the **Resulting context**, a **Rationale** on deep or complex aspects of the patterns, **Related pattern** and **Known uses**. The GoF format contains other types of information: an optional **Classification** of the pattern among others, a **Known as** field in case the pattern also exists with different names, the **Motivation** to introduce an example of scenario the pattern can address, **Applicability** to describe situations where the pattern can be applied, **Participants** (eg. classes and objects) and the **Collaboration** that links them to carry out their responsibilities, the **Structure** of the pattern, the **Consequences** of using it on the software, an **Implementation** part to describe code samples and key technical aspects to consider, **Known uses** and **Related pattern**. The process of writing patterns from existing knowledge is also a subject of research in the pattern community. For example, Meszaros et al. propose a pattern language for pattern writing, thus using patterns to address commonly occurring problems when writing patterns (Meszaros and Doble, 1997). Harrison presents advice for shepherding, a method used in the pattern community to improve the quality of patterns by having an experienced pattern writer review patterns from others (Harrison, 1999). The patterns format as well as the methodologies to write patterns are very useful to construct patterns in a comprehensive and informative way, as it can be difficult to formalize a pattern even with expertise in the domain associated with the pattern to write.

2.2 Issues in Blockchain-based Application Development

As the previous section introduces some definitions and background on blockchain technologies, this section aims to explore existing issues encountered by practitioners when creating blockchain-based applications. The section notably explores the issues related to the design and implementation of blockchain-based applications, as this is the core of this thesis. Therefore, this section states the first knowledge question in the context of the Design Science Research (DSR) method followed in this thesis (Chapter 3), that stands as follows: *What are the existing issues faced by developers in the design and implementation of blockchain-based applications?*

To identify the issues, a Focused Literature Review (FLR)[2] is performed on secondary research literature. The goal is to collect and classify these issues, in order to identify the biggest threats to developing blockchain applications by practitioners. Then, possible solutions are discussed regarding the main issues identified during the review according to the existing literature.

[2]Also known as targeted systematic review.

2.2.1 Review Process

To carry this FLR, the Systematic Literature Review (SLR) guidelines from Kitchenham et al. are reused (Kitchenham and Charters, 2007).

The main difference between systematic and focused literature reviews emanates from the sample of collected papers. Whereas systematic literature reviews aim to collect all published research relevant to the research question that guides the review, focused literature reviews aim to only collect key research relevant to the question. Therefore, this focused literature review follows Kitchenham et al. guidelines, but the query employed to get papers from chosen academic libraries is designed to only search for terms in titles. As a result, the query has better precision, but also has a lower recall than queries that search for terms in all of the article. This decision has been taken as this work aims to identify major common issues, and not to collect all existing issues.

Review Planning

The first step of this FLR is planning the review. In this step, the context of the literature review is defined: research question(s), search query, literature database(s), and inclusion/exclusion criteria.

This work is guided by the following research question: *What are the existing issues faced by developers in the design and implementation of blockchain-based applications?*

To extract relevant studies to this question, the Scopus library has been used. Indeed, Scopus aggregates studies from multiple other sources, such as IEEE Xplore, ACM Digital Library, and Springer thus aggregates the majority of existing studies in computer science and information systems. Next, a search query has been designed to retrieve papers from Scopus. As mentioned before, the query search terms were only looked into article titles, to improve precision. This query has been empirically designed using the main terminology of software engineering and blockchain, as well as the terminology often used in secondary research titles. The resulting query can be written as follows:

> *(blockchain OR dapp* OR decentralized application* OR smart contract*) AND (software OR engineer* OR development OR design) AND (challenge* OR issue* OR review)*

To guide the addition of studies into the set of retained papers for the literature review, inclusion and exclusion criteria have been defined. They provide guidelines to include or exclude papers during the literature review, first during the filtering phase (based on title and abstract), then during the reading phase. Table 2.1 provides the chosen inclusion and exclusion criteria.

Finally, a Quality Question (QQ) (Kitchenham and Charters, 2007) has been defined: *Are development issues clearly identified and described in the article?* This question acts as an

Inclusion criteria	Exclusion criteria
• Presents one or more issues related to the design and implementation of blockchain-based applications.	• The paper does not introduces any issues. • The paper introduces issues, but related to the blockchain technology itself. • The paper is not secondary research. • The paper is dedicated to a specific domain (e.g. IoT, supply-chain, etc.). • Full text is not accessible. • The paper is not written in English or French. • The paper has not been peer-reviewed.

Table 2.1: Inclusion and exclusion criteria.

additional filter during the reading phase of the literature review: if an article that was initially retained for reading does not present understandable and clear issues, it was discarded from the final set of papers that constitutes the corpus of the literature review.

Review Execution

Figure 2.1 shows a graphical overview of the review protocol employed. At first, 65 papers were retrieved from Scopus using the aforementioned query. Then, 50 papers were discarded from their title or abstract, as they were not suggesting the presence of any issue for the development of blockchain applications. The remaining 15 articles were read, from which 9 were excluded from the corpus as they were not introducing any issue, or potential issues were not explained clearly (QQ1).

Finally, 6 articles were added from backward snowballing. Backward snowballing consists of analyzing references in the papers read during the literature review, to identify new relevant papers. Of these articles, two were already in the corpus of papers. The other four were discarded as they were not presenting any issues. As a result, 6 papers were kept to form the final corpus of papers, used to extract issues. These papers are listed in Table 2.2.

The first article from Ayman et al. carries a review of two developer social media (Stack Exchange and Medium) to identify the main discussion topics but also technological challenges met by developers, with a focus on tools and security (Ayman et al., 2020). To carry out this study, 30 761 questions on smart contracts, 38 152 answers, and 73 608 comments were collected from Stack Exchange, as well as 4 045 medium articles. In the second article from Worley et al., strengths and weaknesses are reviewed, then design patterns are introduced to tackle existing issues (Worley and Skjellum, 2018). The third article from Bosu et al. proposes a survey among developers that aims to understand the motivations, challenges, and needs of blockchain software developers and analyze the differences between blockchain and non-blockchain software development (Bosu et al., 2019). There were 145

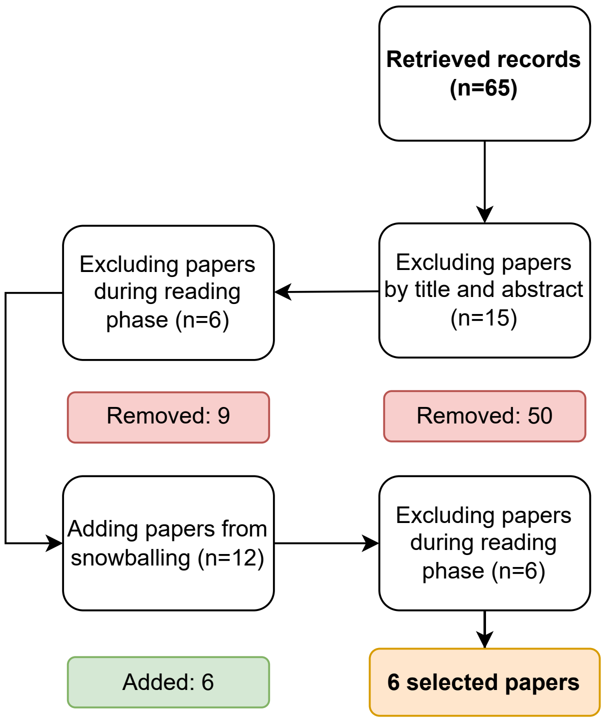

Retrieved records (n=65)
Excluding papers by title and abstract (n=15)
Excluding papers during reading phase (n=6)
Removed: 9
Removed: 50
Adding papers from snowballing (n=12)
Excluding papers during reading phase (n=6)
Added: 6
6 selected papers

Figure 2.1: Review process scheme.

Table 2.2: Literature review corpus of papers.

Ref.	Paper Title	N^b of issues
(Ayman et al., 2020)	Smart contract development from the perspective of developers: Topics and issues discussed on social media	2
(Worley and Skjellum, 2018)	Opportunities, challenges, and future extensions for smart-contract design patterns	3
(Bosu et al., 2019)	Understanding the Motivations, Challenges and Needs of Blockchain Software Developers: A Survey	11
(Zou et al., 2019)	Smart Contract Development: Challenges and Opportunities	29
(Lokshina and Lanting, 2021)	Revisiting State-of-the-Art Applications of the Blockchain Technology: Analysis of Unresolved Issues and Potential Development	4
(Kannengiesser et al., 2021)	Challenges and Common Solutions in Smart Contract Development	29

respondents to the survey, with various backgrounds and levels of experience in the blockchain field. Another survey was carried out in the fourth paper article from Zou et al. among 20 developers to identify differences between smart contract development and traditional software development, as well as existing challenges in smart contract development (Zou et al., 2019). Lokshina et al. propose in the fifth paper a literature review on the state-of-the-art of blockchain applications, but also outstanding issues and potential solutions (Lokshina and Lanting, 2021). Finally, Kannengießer et al. have identified 29 challenges and 60 solutions to smart contract development, as well as 20 design patterns from two literature reviews and expert interviews (Kannengiesser et al., 2021).

From these papers, 78 issues were extracted. For the sake of clarity, the full list of issues in the paper is given in Appendix 8.3. In this collection, an issue is described by two fields: a short name, and a 2-to-4 lines description. Some of the issues were related to a specific set of blockchain technologies: this information was also collected during the literature review. Also, potential solutions to these issues were also collected, if described in the articles mentioning the issues.

As this structure is sufficient to understand the issues described in the articles, it may be difficult to reason on multiple issues at once. Yet, some issues are strongly related: for instance, 7 of them are directly linked to the Solidity language. Also, as issues were extracted from multiple papers, the raw collection contains duplicates, or issues that are sub-issues of others. Thus, the next step in the review execution was to clear duplicates, then classify issues into multiple categories that help understand the major issues for blockchain developers. In this goal, five categories were created:

1. Cost and performance considerations
2. Application security issues
3. Environment and tooling challenges
4. Blockchain-specific challenges
5. Ethereum ecosystem limitations

These categories emanate from the keywords used in descriptions of issues, and the recurrence of specific groups of issues. For instance, many issues were directly targeting the Ethereum ecosystem: a dedicated category was created to classify them. Then, each issue was systematically compared with others to remove duplicates:

- If the issue is identical to another, the two issues were fused into a single one.
- If the issue is a sub-issue of another, the other issue was transformed into a sub-category, and the issue was classified under this category.

As a result, 21 issues were removed: 8 were transformed into sub-categories, and 13 were fused with others. From this filtering, 57 unique issues remain. These issues are described in detail and discussed in the following subsection[3].

[3]Bote that issues are written in italic font.

2.2.2 Development issues in the design and implementation of blockchain applications

Blockchain-specific Challenges

The first category of issues is "Blockchain-specific challenges" (Figure 2.2). It regroups 7 different issues, 3 being in a subcategory named "Immutability", and 3 others in the "Technical Soundness" subcategory.

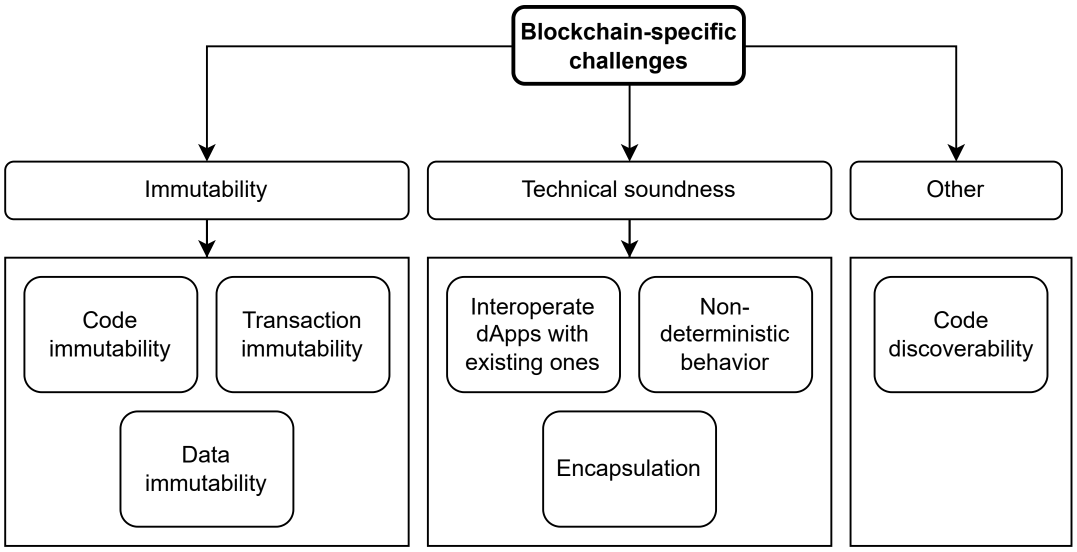


Figure 2.2: Blockchain-specific challenges map.

Immutability is, at the same time, an opportunity and a threat for blockchain-based applications. Immutability makes data accessible and manageable by different entities that do not trust each other (Belotti et al., 2019). But immutability comes with several threats. First, *Code immutability*: once a smart contract is deployed on-chain, it is impossible to delete them afterward. This is a threat, as if a code flaw or vulnerability is discovered in a smart contract, it cannot be repaired natively and can be exploited indefinitely (Worley and Skjellum, 2018). Then, another immutability threat is *Transaction immutability*: unlike traditional software development, the user cannot recover any loss while making transactions on blockchain-based systems using smart contracts (Zou et al., 2019). *Data immutability* is the third immutability threat, as smart contracts may control and manage sensitive digital assets (Zou et al., 2019). Also, the immutability of data may breach data confidentiality, as they are visible on-chain by others (Kannengiesser et al., 2021). Thus, developers must be very careful when implementing blockchain-based applications and think about potential safeguards to prevent any issue with native immutability (e.g. smart contract upgrade mechanisms, checks, extensive testing, etc.).

The second subcategory, "Technical soundness", describes issues that are met by developers when handling the technical capabilities and limitations of a smart contract's execution environment (Kannengiesser et al., 2021). Developers may have to *Interoperate dApps with* *existing ones*. According to Lokshina et al., the literature advises that the number of blockchain-based applications grows rapidly, creating many heterogeneous solutions (Lokshina and Lanting, 2021). This causes interoperability issues, as many of them have different non-standardized interfaces. *Encapsulation* is a similar issue, that concerns the interoperability of smart contracts with external systems. Indeed, smart contracts execute in an environment that is totally determined by the world state of the blockchain and the messages passed to the contract by its caller (Worley and Skjellum, 2018). They have to rely on oracles (Mühlberger et al., 2020) to request external data or move the execution of computation externally (Kannengiesser et al., 2021). Developers may have to carefully design interactions with other smart contracts and external systems if needed. The deterministic environment of blockchain also makes difficult the usage of functions that have a *Non-deterministic behavior* (Kannengiesser et al., 2021). For instance, randomness is difficult to achieve natively in blockchain, and may also require the usage of external services (e.g. oracles) to provide randomness to smart contracts.

The development of blockchain applications also faces other challenges. One of them is *Code discoverability*: it addresses the difficulty of finding deployed smart contracts on a distributed ledger (Kannengiesser et al., 2021). Indeed, smart contracts are often stored in blockchains in a compressed format (e.g. bytecode), referenced by addresses that are hard to read (e.g. hexadecimal public addresses).

Application Security Issues

The second category of issues is "Application security issues" (Figure 2.3). It regroups 11 different issues, 7 being in a subcategory named "Avoiding smart contract vulnerabilities".

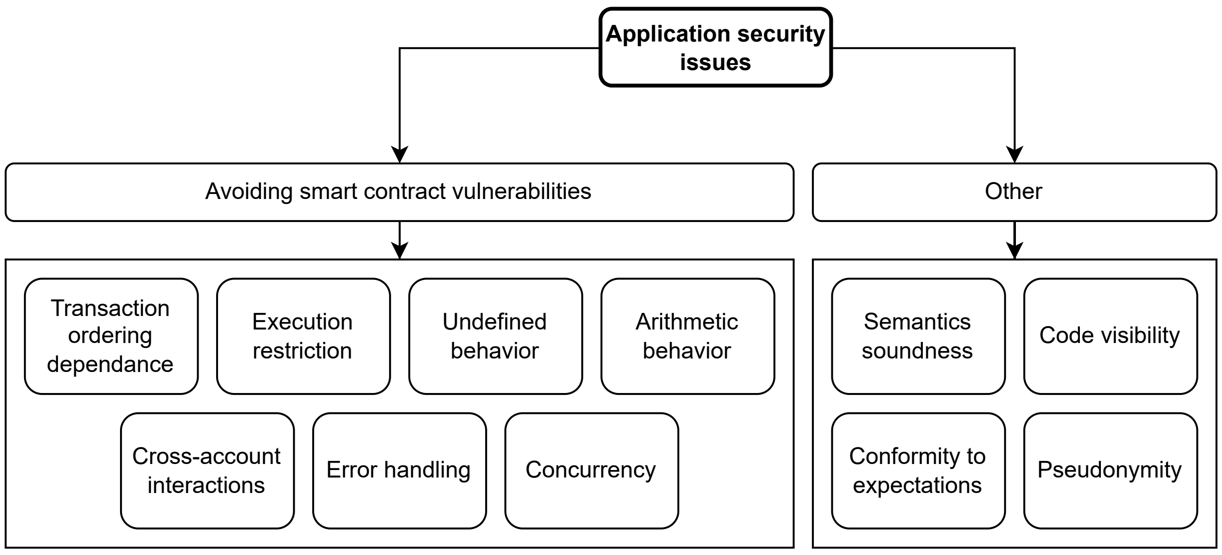

Application security issues
Avoiding smart contract vulnerabilities
Other
Transaction ordering dependance
Execution restriction
Undefined behavior
Arithmetic behavior
Cross-account interactions
Error handling
Concurrency
Semantics soundness
Code visibility
Conformity to expectations
Pseudonymity

Figure 2.3: Application security issues map.

This first subcategory includes issues that directly underline potential vulnerabilities that may occur in smart contracts. These issues should be addressed carefully by developers during the implementation of blockchain applications. The first one is *Execution restriction*. By default, smart contracts functions may be executed by anybody on the blockchain network. This may lead to the undesired executability of functions by malicious entities to exploit smart contracts (Kannengiesser et al., 2021). Similarly, the *Error handling* issue underlines the importance of handling errors well to avoid any exploit. Next, *Undefined behavior* and *Arithmetic behavior* respectively introduce the issues arising when functions rely on under-specified operations or arithmetic operations. Without some knowledge of these operations functioning, a developer may introduce vulnerabilities such as overflows, underflows, or undefined behavior that results in asset loss or denial of service (Kannengiesser et al., 2021). The *Transaction ordering dependance* issue is another example of undefined behavior: as sometimes smart contracts rely on a specific transaction order to perform, poorly designed smart contracts may be threatened by malicious users trying to send a set of transactions in a modified order to exploit this flaw. Finally, the *Cross-account interactions* issue shows potential risks of calling a smart contract by its account: unavailable smart contract, function not found, or unintended function call (Kannengiesser et al., 2021). These interactions must be carefully designed by developers to avoid any flaws that may arise from them.

Along this category of issues, 4 additional issues may threaten the security of the application to design. The *Code visibility* issue emanates from blockchain transparency: the code of smart contracts deployed on-chain is visible by anybody, thus it may be reverse-engineered to design an attack and execute it (Zou et al., 2019), or challenge companies that want to keep their process logic confidential (Kannengiesser et al., 2021). *Pseudonimity* also causes challenges related to accountability and liability as actual entities interacting with blockchain smart contracts remain unknown (Kannengiesser et al., 2021). Finally, it may be difficult for developers to create smart contracts that are both free from logical errors and flaws (*Semantic soundness* issue) but also complies with initial requirements (*Conformity to expectations* issue).

Cost and Performance Considerations

The third category of issues is "Cost and performance considerations" (Figure 2.4). It regroups 9 different issues, 4 being in a subcategory named "Cost of use".

In this category, some issues may cause a higher cost of use, as it may cost money to deploy or interact with on-chain smart contracts. Among these issues, *Data storage* may be the most important: the way data is stored in a blockchain application matters a lot to the application efficiency. An adequate balance should be established between data stored on-chain that benefits blockchain qualities, and data stored off-chain to save storage costs. *Data type complexity* also has a great impact on the application efficiency: selecting appropriate data types may affect the cost of storage and execution of smart contracts (Kannengiesser et al., 2021). Finally, logic also has an impact on the application cost of use. Therefore, *Under-optimized code* may be an issue, as a function that performs poorly may cost more than optimized functions. *Iterations through data structures* is also an issue regarding logic cost of use: developers may have to implement code in order to minimize iterations on data

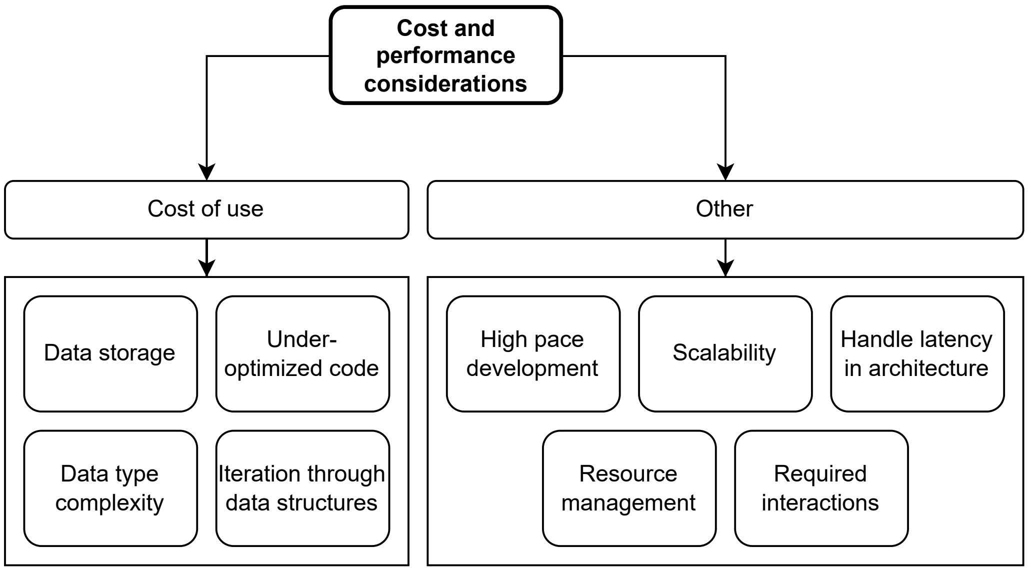


Figure 2.4: Cost and performance considerations map.

structures to guarantee returning data from a data structure in O(1) (Kannengiesser et al., 2021).

Along with that, other issues may impact the cost and the performance of blockchain applications to be designed and implemented. The *High pace development* of the blockchain field threatens blockchain projects, as they may be at risk of losing their market capitalization (Bosu et al., 2019). Developers must move fast and adjust their applications to meet new requirements. *Scalability* is also an area of concern for developers: the peculiar functioning of blockchain-based applications coupled with the aforementioned high pace development makes challenging the creation of scalable applications. Other aspects of blockchain applications have an important role in cost and performance: the issue *Handle latency in architecture* underlines this by stating that blockchain-based applications may face latency when relying on blockchain smart contracts. *Resource management* also threatens the efficiency of blockchain applications, as they may rely on providing a specific resource to guarantee smart contract termination, such as gas for an Ethereum-based application (Kannengiesser et al., 2021). As transactions may be expensive, the *Required Interactions* states that developers may have to reduce as possible interactions with smart contract, for instance using the factory design pattern (Kannengiesser et al., 2021).

Development Environment Challenges

The fourth category of issues is "Development environment challenges" (Figure 2.5). It regroups 15 different issues, 2 being in a subcategory named "Code reviews", 4 in "Lack of guiding knowledge" and 3 in "Tooling issues".

Two code review issues have been identified for blockchain-based application development. The first one is *Longer code reviews*: developers may need more time to carry code reviews in blockchain-based application development (Bosu et al., 2019). They may also need more

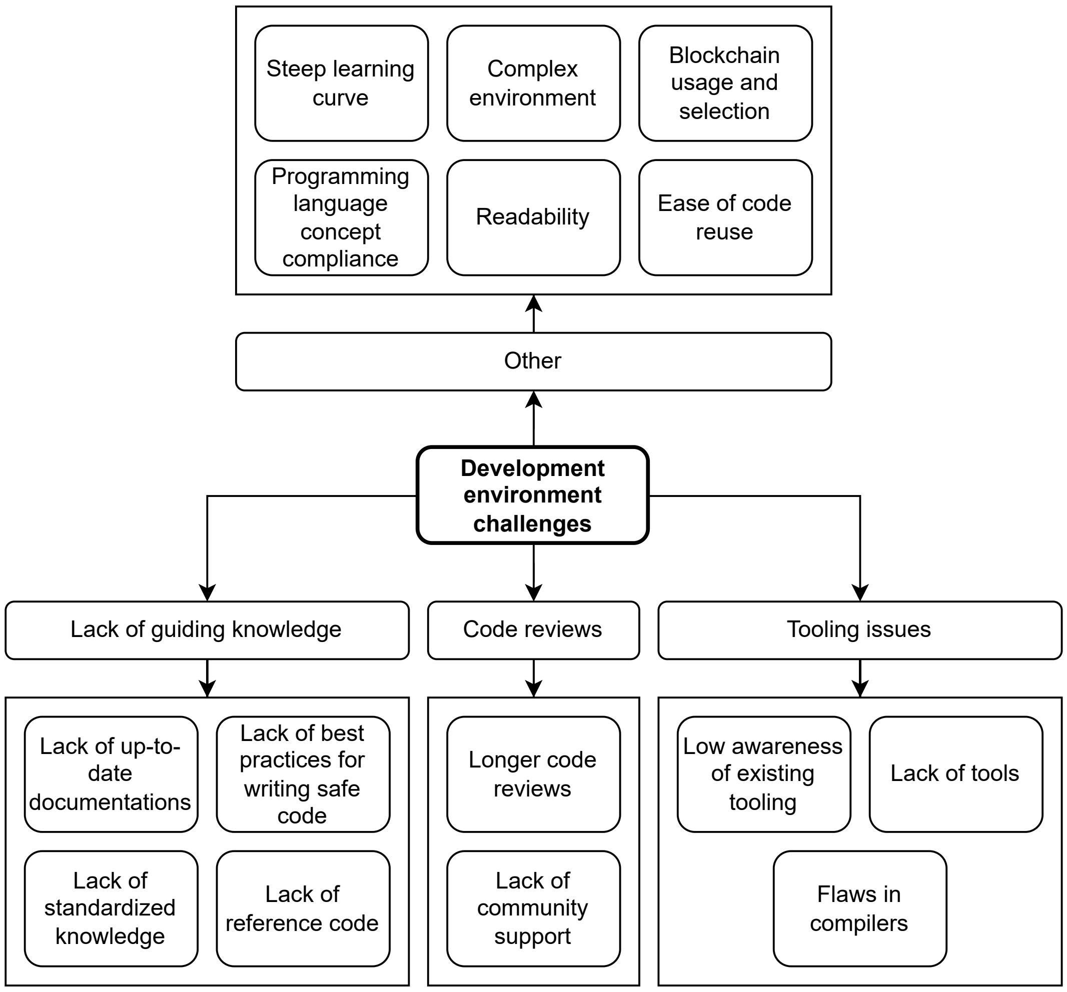

Steep learning curve
Complex environment
Blockchain usage and selection
Programming language concept compliance
Readability
Ease of code reuse
Other
Development environment challenges
Lack of guiding knowledge
Code reviews
Tooling issues
Lack of up-to-date documentations
Lack of best practices for writing safe code
Lack of standardized knowledge
Lack of reference code
Longer code reviews
Lack of community support
Low awareness of existing tooling
Lack of tools
Flaws in compilers

Figure 2.5: Development environment challenges map.

time to find competent people to review their code. This issue is mentioned as being the *Lack of community support* (Bosu et al., 2019).

Another main issue in the developer environment is tooling. First, developers may have *Low awareness of existing tooling*. Ayman et al. have highlighted that only a few posts on the Ethereum StackExchange or Medium discuss or ask questions about existing tooling, such as security tools (Ayman et al., 2020). Then, developers may face the *Lack of tools*, that is both mature and reliable, to assist them in the development of blockchain applications (Bosu et al., 2019). For instance, testing tools, blockchain dedicated IDEs, and additional development extensions (Zou et al., 2019). Also, compilers, that are key components in blockchain-based application development, may still be immature and contain security bugs. This is underlined by the *Flaws in compiler* issue (Zou et al., 2019).

Regarding knowledge, developers may find it difficult to find guiding knowledge for developing blockchain-based applications. Several articles in the literature review corpus mention different gaps: the *Lack of best practices for writing safe code*, the *Lack of reference code*, the *Lack of standardized knowledge*, and the *Lack of up-to-date documentations* (Bosu et al., 2019; Zou et al., 2019).

Other issues related to the development environment of blockchain-based applications can be mentioned. Developers may face a *Steep learning curve*, or a *Complex environment* when trying to grasp knowledge on developing blockchain-based applications (Bosu et al., 2019). Indeed, developers have to ramp up on many domains, such as cryptography, networking, distributed systems, and specific protocols. Once they have the required knowledge on these aspects, they have to tackle the peculiar environment of a blockchain application (e.g. special components such as oracles, on-chain/off-chain distinction, etc.). Another challenge met by developers is *Blockchain usage and selection* (Bosu et al., 2019). Using blockchain in an application is not always relevant: for instance, it may be pointless to use blockchain when there is no need to store data in a decentralized way. If it is relevant, the developer then has to pick the adequate technology regarding its requirements. This may be an issue regarding the rising amount of available technologies, that all have specific characteristics. Regarding coding aspects, developers may struggle with the *Programming language concept compliance* issue. According to Kannengießer et al., programming language concept compliance is the degree to which a programming language conforms to established concepts and the use of terms in related programming languages (Kannengiesser et al., 2021). For instance, the usage of the keyword "protected" may have different significations in smart contract programming languages and established programming languages (e.g. Java, C++, etc.). Along that, developers may face code *Readability* issues, but also difficulties in code reuse (*Ease of code reuse*). Nonetheless, these issues are also present in traditional software development.

Ethereum ecosystem limitations

The fifth and final category is named "Ethereum Ecosystem Limitations" (Figure 2.6). It gathers 15 different issues, 7 of them in the "Solidity" subcategory, 4 of them being in the "Ethereum Virtual Machine (EVM)" subcategory, and 4 of them in the "Gas issues" subcategory. This category was created to address the prevalence of Ethereum-related issues compared to others.

The first subcategory classifies issues that emanate from Solidity[4], the most-used programming language to implement Ethereum smart contracts. As Solidity was released in 2015, the language is still young compared to traditional programming languages such as Java or C++, that cumulate decade of existence. Thus, developers may have to tackle many issues when using this language. Zou et al. underline several of them, met by the developers that participated in their survey (Zou et al., 2019). First, Solidity both *Lack of standards and rules* and *Lack of general purpose libraries*. For the former, although standards exist in the Ethereum ecosystem (so-called Ethereum Request for Comment or ERC), some developers were highlighting their usage scarcity. The latter addresses the need for general-purpose libraries that were extensively tested and applied in applications. As for the EVM, Solidity also struggles with a *Lack of support for error reporting and logging*. Indeed, Solidity does not support basic error printing, making it difficult for developers to debug their applications (Zou et al.,

[4]`https://docs.soliditylang.org/`

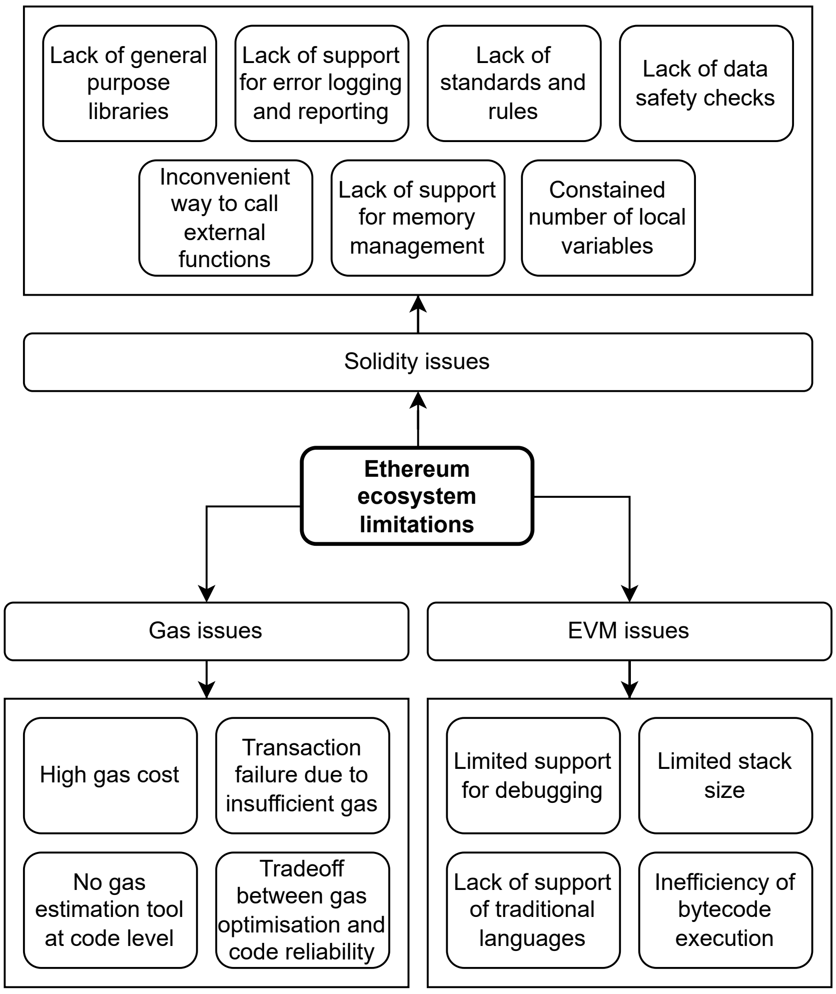

Lack of general purpose libraries
Lack of support for error logging and reporting
Lack of standards and rules
Lack of data safety checks
Inconvenient way to call external functions
Lack of support for memory management
Constained number of local variables
Solidity issues
Ethereum ecosystem limitations
Gas issues
EVM issues
High gas cost
Transaction failure due to insufficient gas
No gas estimation tool at code level
Tradeoff between gas optimisation and code reliability
Limited support for debugging
Limited stack size
Lack of support of traditional languages
Inefficiency of bytecode execution

Figure 2.6: Development environment challenges map.

2019). Regarding security, Solidity also *Lack of data safety checks*[5] (e.g. overflow/underflow), and has an *Inconvenient way of to call external functions*. For the latter, they have to use a low-level operation, thus prone to errors and vulnerabilities. Finally, Solidity has a *Lack of support for memory management*, making difficult memory optimization, and has a *Constrained number of local variables* (i.e. contextual variables) that requires additional efforts from the developer (Zou et al., 2019).

The EVM is the component used by nodes to execute smart contracts when a transaction is sent from a user. Developers may face different issues when executing their code using EVM. They may find a *Limited support for debugging* when trying to identify the source of an error using the EVM debugger (Zou et al., 2019). Error messages may also not be very detailed, making difficult error logging. EVM is also limited regarding programming languages: there is a *Lack of support of traditional languages* that can be compiled and then executed by the EVM. Indeed, two programming languages are mainly used by developers to implement

[5]Note that this issue was valid at the time of the study, but many type checks have recently been added in the latest versions of Solidity.

Ethereum smart contracts, that are Solidity and Vyper[6]. Although their syntax are close to existing languages (resp. Javascript and Python), they are still new languages purely created for developing smart contracts. EVM also suffers from a *Limited stack size*, and *Inefficiency in bytecode execution*, notably as the EVM is single-threaded.

Lastly, 4 other issues related to gas were collected. In Ethereum, gas is a metric used to quantify the cost of performing low-level operations, so-called opcodes (Marchesi et al., 2020). When developers deploy smart contracts, or users interact with them in order to change the blockchain state, they have to pay Ether, that is the network cryptocurrency. On the Ethereum mainnet, as Ether has a cost, it may be expensive to operate smart contracts. Thus, developers may suffer from *High gas cost* (Zou et al., 2019), depending on the type of operation performed by their smart contracts, but also the level of optimization reached by the developer. This causes another issue, that is the *Tradeoff between gas optimization and code readability*. Many developers have reported that it may be tricky to optimize gas without hurting code readability. As gas is used as a resource to run smart contracts, it has to be provided when executing a function that will modify the blockchain state. Sometimes, this execution may run short on gas, as not enough was provided by the user beforehand. This causes the issue of *Transaction failure due to insufficient amount of gas*: developers should implement measures to ensure enough gas is provided, and if not, gracefully handle the transaction failure. Finally, there is *No gas estimation tool at code level*. Where developers often desire to write and optimize source code rather than bytecode as it is more intuitive, no tool exists to provide such information (Zou et al., 2019).

2.2.3 Discussion

As seen in the last section, many issues hinder the development of blockchain-based applications. In this section, these issues will be discussed in light of the driving research question (RQ1), but also potential solutions in the literature. Two major groups can be distinct from these issues: the lack of knowledge from developers, and the lack of tooling suites.

The first group would regroup any issue related to designing/implementing parts of blockchain applications. As underlined in the previous section, developing blockchain applications requires to tackle with blockchain strengths but also weaknesses (e.g. security, efficiency, immutability, scalability, etc.). Even with deep knowledge of traditional software engineering, developers may find challenging the design and implementation of blockchain applications, as they have to be aware of potential security threats, but also of how to design components that are unique to blockchain applications (e.g. oracles). On top of that, they have to perform these tasks while trying to minimize the impact of potential blockchain weaknesses on applications. For instance, although blockchain may help save costs by removing friction and third parties, these savings may be nullified by performance issues or exploited vulnerabilities.

[6]https://vyper.readthedocs.io/

One common solution in traditional software engineering to guide developers in these aspects is the usage of software patterns (Section 5). This solution was mentioned by most of the articles composing this literature review. As an example, Worley et al. introduce several blockchain design patterns for the design of specific blockchain components, such as oracles (Worley and Skjellum, 2018). Using design patterns, developers have clear guidelines on how to design or implement specific components, in a defined context and for a given problem. Another advantage of a design pattern is the potential applicability to any design or implementation issue: collections may be created to address any specific category of issue. For instance, such collections already exist in the literature, such as security blockchain design patterns (Wohrer and Zdun, 2018), or specific blockchain feature design patterns (Xu et al., 2018). Yet, blockchain software patterns are still in their premises: patterns are scattered across academic literature or technical sources, thus developers first have to be aware of pattern sources to reuse them. Also, developers that lack blockchain knowledge may find it difficult to assess the applicability and then apply software patterns, even if the information is given in a structured pattern format (Jalil and Noah, 2007). Going further with design reuse using software patterns, code reuse is also another solution to address the lack of knowledge. Instead of manually creating new features, it may be more interesting to reuse existing ones. Potential benefits of code reuse are the increase in development productivity, improvement of software quality, and better software reliability (Feitosa et al., 2020). In the blockchain field, code reuse is already applied in many cases. For instance, Chen et al. collected and analyzed 146 452 open-source smart contracts to study code reuse in the Ethereum ecosystem and identified that 91,11% of these contracts reuse a set of 20 subcontracts (i.e. code samples) (Chen et al., 2021).

During the literature review, another related solution has also been identified, that is the creation of guidelines, best practices, frameworks, or guides (Bosu et al., 2019). Compared to design patterns that address specific aspects in the design or implementation of blockchain applications, this would help guide the developers across the different steps and parts to implement at the same time. For instance, frameworks already exist to guide the creation of domain-specific blockchain applications, such as insurance (Raikwar et al., 2018), development (Cunha, Soja, and Themistocleous, 2021b), or supply chain (Reddy et al., 2021). Nonetheless, developers may also benefit from domain-agnostic frameworks that focus on the different phases of software engineering.

The second group of issues found in this literature review is the lack of tools and tooling suites in the blockchain field. Although a developer has extensive knowledge of blockchain technology and blockchain application development, it may still be tedious to design and implement applications without any external help. Some tasks, such as finding security issues, implementing specific blockchain features, testing the application, or optimizing the code, may be automated by tools, as it may still be tedious and error-prone to do these tasks manually. This aspect is an open research field, as many researchers propose different tools to address these issues (Vacca et al., 2021). However, as identified by Ayman et al., only a few developers are aware of existing tools (Ayman et al., 2020).

Another range of solutions that address both the lack of knowledge and the need for tools is code generation. Indeed, code generation both allow to reuse existing assets, such as design patterns but also other types of assets such as models (Sebastián, Gallud, and Tesoriero, 2020) or code fragments (i.e. templates) (Syriani, Luhunu, and Sahraoui, 2018). This approach has already been explored in the blockchain field. Different models may be used to generate smart contracts, such as BPMN files (Lu et al., 2020), Petri nets (Zupan et al., 2020), or Unified Modeling Language (UML) diagrams (Jurgelaitis, Butkienė, et al., 2022).

2.2.4 Threats to Validity

Internal threat to validity: as mentioned before, this literature review was not carried out systematically. Some relevant articles may not be present in the final corpus of papers. However, it is important to note that the goal of this study was not to collect all existing issues, but rather to identify the most important ones. This means that the potential impact of this threat is likely to be limited. Additionally, while the analytic process was carried out by a single researcher, the quality question introduced in the review process helped to exclude studies where the identified issues were too vague. This has made the task of collecting issues easier, thus limiting the risk that two researchers would identify different issues or merge non-identical issues.

External threat to validity: one potential external threat to the validity of this literature review is the relatively small sample size of only 6 articles. While these articles are all drawn from secondary research, which means they have already summarized other sources of knowledge, the limited number of articles included in the review may still raise concerns about the generalizability of the findings. However, it is important to note that the results of this literature review are likely to encompass a broader range of knowledge than the small number of articles might initially suggest.

Construction threats to validity: the research query used to collect papers can be mentioned. It has been designed based on titles from the existing secondary literature but also blockchain articles. Yet, articles do not directly mention their belonging to secondary literature but provide clues such as indicating that, for instance, an article is a review of a specific domain. Efforts have been made to design a query capable of getting most of the secondary research, such as including in the research query the terms "challenge" or "issue".

Conclusion threats to validity: as only a few issues were found in different articles, it may be difficult to conclude on the main issues that are met by developers directly based on the subset of issues that were identified during the literature review. To tackle that, 5 categories of issues and subsequent subcategories were created to represent these major issues, where issues within these categories can be considered as instances of these main issues.

2.3 Conclusion

In this chapter, an extended introduction to blockchain technologies and software patterns was given to give the reader the necessary background for the rest of the thesis. Then, a literature review was carried out to explore existing issues met by developers in the design and implementation of blockchain applications. As a result, 6 papers were chosen to form the literature review corpus, and 57 unique issues were identified and then classified into 5 categories and 9 subcategories.

As a result, the answer to RQ1 is that five classes of issues threaten developers when designing and implementing blockchain applications. First, they may find it difficult to tackle with specificities of blockchain applications, such as immutability or decentralization. Then, they have to be very careful with the security of the application to create, to prevent security flaws and vulnerabilities. This also includes potential threats to data confidentiality and privacy. Developers also have to assess the efficiency of their application, as it may be very expensive to deploy and execute non-optimized smart contracts. The lack of tools to assist the developer, as well as the complex development environment also hinders developers in the creation of blockchain applications. Finally, some limitations may also exist depending on the chosen blockchain technology. This is notably the case for Ethereum, as issues were identified with its main language (Solidity) but also its virtual machine, used by nodes to execute smart contracts.

To tackle these issues, potential solutions were also discussed. Software patterns are probably one of the most efficient resources for developers to tackle the difficulty of addressing complex design/implementation blockchain issues (e.g. developing specific features, avoiding vulnerabilities, etc.). Yet, as blockchain is still relatively recent as a technology, some challenges remain, such as creating extensively tested patterns for the pain points of designing or implementing blockchain applications, gathering them into reusable collections, and facilitating their reuse. Guidance throughout frameworks, guidelines, or best practices guides was also identified as a promising solution. The lack of assistance during the creation of blockchain applications may also be solved by creating tools to assist the developer on specific aspects, such as security, testing, code generation, or optimization.

Identifying the different classes of issues, as well as the issues themselves and potential solutions, opens to the creation of approaches that address classes of issues as a whole instead of focusing on specific aspects. In this thesis, such an approach is proposed as a framework named BlockcHain fRaMewOrk for the desigN and Implementation of deCentralized Application (Harmonica) that is made up of tools designed to address the major pain points and issues identified. This framework will be introduced in the next chapter.

Chapter 3

Overview of the Method and Contribution

Key takeaways

- The Design Science Research method is introduced and research questions of the thesis are stated in line with DSR types of questions.
- An overview of the Harmonica framework is presented. Each part of the framework, that contains 3 artifacts and a knowledge base, is presented individually, but also with regards to the DSR method.
- A running example, that is a case study of a blockchain traceability application, is introduced. It is reused throughout the thesis to illustrate the different contributions.

In the last chapter, a literature review was carried out to identify issues met by practitioners in the design and implementation of blockchain applications. The literature review notably highlights the need for developers for guidance throughout these phases, notably with knowledge and tools.

In light of these results, this thesis aims to propose a framework envisioned to provide this guidance, notably in the design and implementation phases of software engineering. First, by guiding developers in the design of their application, by suggesting the blockchain technology and blockchain-based software patterns to use from requirements. Then, by facilitating the implementation of the application throughout code generation.

In this chapter, an overview of this framework is given. This overview synthesizes the different concepts and elements of the framework, through the prism of Design Science Research, the chosen research method to carry the different research works of this thesis. DSR consists in iterating over two activities: designing an artifact that improves something for stakeholders and empirically investigating the performance of an artifact in a context (Wieringa, 2014).

In the context of this method, an presentation of the driving elements of this thesis is given. These elements are the chosen research questions derived from the literature review (Chapter 2), and the proposed framework to answer them, that is the Harmonica framework. Then, the overview of the framework and each of its artifacts is given. The goal of this overview is not to deep dive into the inners of each artifact, but rather to help the reader understand their role within the framework. To further guide the reader in this thesis, a running example is also introduced in order to exemplify the usage of the framework throughout the next chapters.

The finality of this chapter is giving a contextualized overview of the Harmonica framework, giving a big picture for the reader to navigate into the following chapters (Chapter 4, 5, 6, and 7), while understanding the different contributions, how they are addressed with regards to the DSR method, and how they assemble into a framework that addresses formulated research objectives.

This chapter is articulated as follows. First, Section 3.1 introduces the DSR method chosen to carry the thesis research work, then Section 3.2 introduces the framework within the DSR context. Section 3.3 presents the running example that is used to illustrate the framework throughout the thesis, and Section 3.4 concludes the chapter.

3.1 Research Method

3.1.1 Design Science Research Introduction

Whereas research is the process of collecting, analyzing, and interpreting data to understand a phenomenon, the research process is systematic in that defining the objective, managing the data, and communicating the findings occur within established frameworks and in accordance with existing guidelines (Williams et al., 2007). To carry out this research process, it is often required to use a research method. As this thesis makes no exception, the design science research method was used to carry out this research work. More specifically, this research reuses the framework from Wieringa (Wieringa, 2014). Therefore, part of the content in this subsection originates from this book.

Figure 3.1 shows an overview of the aforementioned framework. At its core, design science is about studying an artifact in context. An artifact is something created by people for practical purposes. It can take multiple forms: algorithms, methods, notations, techniques, and frameworks.

The artifact is studied in a twofold context: the social context, and the knowledge context. The social context is about the stakeholders, such as users, operators, maintainers, or sponsors (that provide the research budget). DSR is used to build artifacts that aim to address their goals, with regard to the provided budget. The knowledge context materializes all of the existing knowledge that can serve the design or the investigation of the artifacts: existing theories, specifications of known designs, lessons learned, facts, etc. This knowledge emanates from different sources, such as literature (scientific, technical, professional), or communication with others. During the design science research process, new knowledge may be produced and added to this knowledge context.

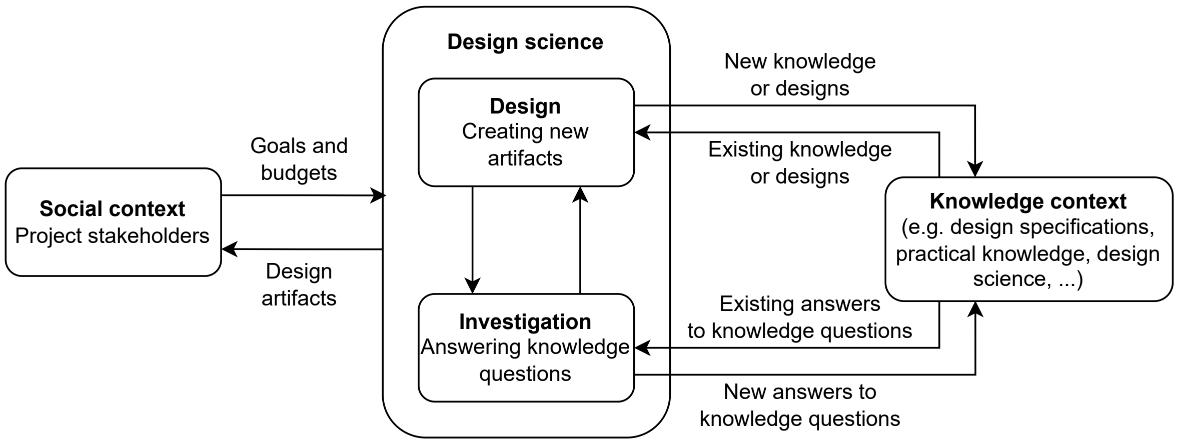

Design science
Design
Creating new artifacts
Investigation
Answering knowledge questions
Social context
Project stakeholders
Goals and budgets
Design artifacts
New knowledge or designs
Existing knowledge or designs
Knowledge context
(e.g. design specifications, practical knowledge, design science, ...)
Existing answers to knowledge questions
New answers to knowledge questions

Figure 3.1: Design Science Research framework from Wieringa.

Two main activities drive the DSR: artifact design, and artifact investigation. Artifact design consists in designing one or more artifacts to improve a problem context. The framework proposed by Wieringa represents this activity using a 3-step process, named design cycle (Wieringa, 2014). This process is composed of three steps:

1. Problem investigation: consists of identifying the problem context: stakeholders, phenomena, causes, effects.
2. Treatment design: in this step, the requirements to address identified problems are defined, available treatments (i.e. solutions) are explored, and new ones may be designed.
3. Treatment validation: produced artifacts are evaluated to assess whether they satisfy requirements. The artifact may be also tested to evaluate if it may be applied in other contexts or possible design trade-offs.

These steps are guided by a design problem. Each design problem should clearly identify the context, the artifact to build, the requirements, and the stakeholder goals. If desired, design problems can also be stated as questions. In this case, design problems become technical research questions. They keep the same elements as design problems but finish with a question mark.

Artifact investigation is about answering knowledge questions about artifacts in context. For instance, knowledge questions on the problem to be solved, or on the artifact to design. Answering these knowledge questions may be required to address the research problems and produce the desired artifacts. These knowledge questions may be addressed using the empirical cycle, that is notably composed of a 5-step process, as well as a checklist of issues that drives these steps. The different steps can be summarized as follows:

1. Research problem analysis: define the research problem to be solved.

2. Research and inference design: design the research setup required to solve the research problem.
3. Validation: consists of defining how the results from the research are going to be validated.
4. Research execution: perform the research according to the defined research design and validation, and take note of the unfolding of the research.
5. Data analysis: analyze the data gathered during the research execution.

Although these activities form a sound research process, the empirical cycle is not meant to be inflexible: researchers may jump from one item to another, or go back to a previous step to improve something. Also, other methods in the literature can be used to answer research questions, even in the context of using the DSR method.

The DSR approach was applied in the context of this Ph.D. thesis. To address the research aim formulated in Chapter 1, 5 research objectives were stated, that has led to 3 technical research questions and 2 knowledge questions. Each question was treated by reusing the two different processes from Wieringa et al. to either address knowledge questions or technical research questions. In the end, the artifacts and the knowledge resulting from answering these questions lead to the framework itself, that addresses the initial research aim. These questions are introduced in the next subchapter.

3.1.2 Research Questions

In Chapter 1, research objectives were stated to introduce the different research works to carry out. This section refines these objectives into proper research questions, in light of discovered issues in the design and implementation of blockchain applications (Chapter 2). From these objectives, 5 research questions can be defined (RQ1-5), further divided into technical research questions and knowledge questions in the context of the DSR method:

- RQ1: What are the existing issues faced by developers in the design and implementation of blockchain-based applications?
- RQ2: How to design a blockchain technology recommendation platform that simplifies the completion of this task for non-experts so that practitioners can choose adequate blockchain recommendations during the design phase of blockchain application development?
- RQ3: How to discover then classify software patterns in a blockchain application?
- RQ4: How to design a blockchain-based software patterns recommendation platform that simplifies the completion of this task for non-experts so that practitioners can choose adequate patterns during the design phase of blockchain application development?

- RQ5: How to design a blockchain application generation platform that reduces the cost and difficulty of implementing blockchain applications as practitioners can configure then generate blockchain applications?

First, two knowledge questions are derived from the research objectives (RQ1, RQ3). They are needed to address technical research questions, as knowledge is lacking to answer them.

RQ1 was the driving question of Chapter Systematic Literature Review. Even if existing issues were mentioned a lot in academic literature, it was needed to collect them in a systematic way, then classify them to identify the most important issues the framework has to address. As a result, it highlights the main challenges faced by developers when designing or implementing applications and discusses existing solutions in the literature that addresses them. Also, existing concepts in these solutions may be reused in the framework.

Regarding RQ3, it is also needed to gather and classify existing blockchain-based software patterns from the literature in an extensive collection. There was no occurrence of such collection in the literature prior to this thesis, while reusing existing blockchain-based software patterns may tackle many design issues identified while answering RQ1. Therefore, this research question is answered in another Systematic Literature Review in Chapter 5.

The other three questions are formulated as technical research questions (RQ2, RQ4, RQ5). Compared to knowledge questions, addressing technical research questions consists in creating a new or improved artifact. In this thesis, each technical research question has led to the design of one artifact.

With RQ2, the answer to the question is materialized with an artifact capable of assisting practitioners when they face the choice of a blockchain technology for their project. This task is being often cumbersome for non-experts, the driving requirement is to simplify it, notably by hiding the complex decision process behind an easy-to-use interface when requirements can be submitted. This technical research question is addressed in Chapter 4. Following the selection of a blockchain technology, the artifact to build in the context of RQ4 is about selecting blockchain-based software patterns with regards to practitioner's requirements. As for RQ4, this task must also be simplified: throughout the platform, practitioners shall be able to explore the existing blockchain-based software patterns, then obtain recommendations on the best patterns to use for given requirements. Finally, throughout RQ5, the goal is to create an artifact for the generation of blockchain applications, directly reusing existing design decisions. Although each artifact that emanates from these research questions can perform independently, they form together the core of the Harmonica framework, introduced in the next subsection.

As a result, the organization of the thesis throughout answering these research questions can be illustrated as shown in Figure 3.2.

This figure notably shows that one research question (RQ1) was already answered in the Systematic Literature Review of Chapter 2. The reason is that the knowledge collected across the literature review, as well as the answer to this research question, was necessary to design the framework proposed to address the research aim of this thesis. As such, Chapter 2

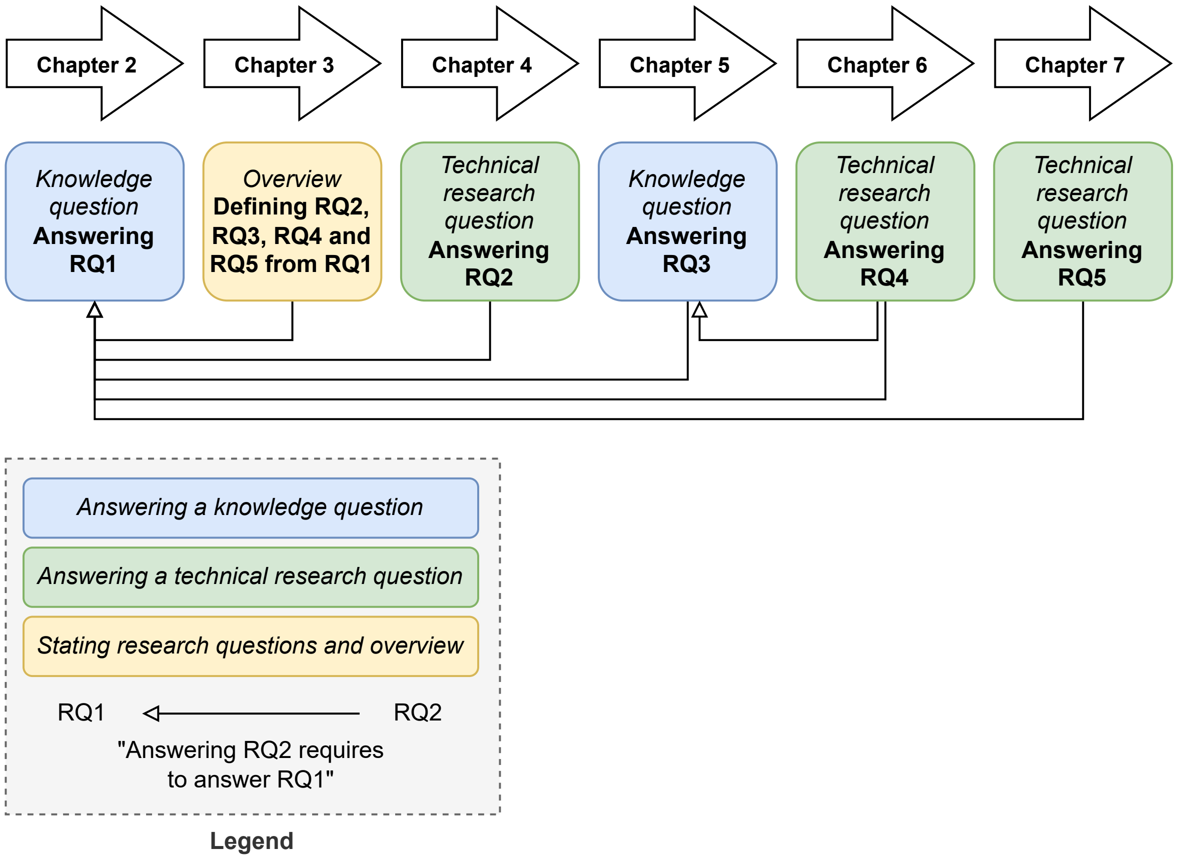

Chapter 2
Chapter 3
Chapter 4
Chapter 5
Chapter 6
Chapter 7
Knowledge question Answering RQ1
Overview Defining RQ2, RQ3, RQ4 and RQ5 from RQ1
Technical research question Answering RQ2
Knowledge question Answering RQ3
Technical research question Answering RQ4
Technical research question Answering RQ5
Answering a knowledge question
Answering a technical research question
Stating research questions and overview
RQ1
RQ2
"Answering RQ2 requires to answer RQ1"
Legend

Figure 3.2: Thesis organization in light of research questions.

addresses RQ1, then Chapter 3 gives an overview of envisioned contribution in light of the DSR method. Then, Chapters 4, 5, 6, and 7 respectively addresses RQ2, 3, 4 and 5.

3.2 Framework Overview

BlockcHain fRaMewOrk for the desigN and Implementation of deCentralized Application (Harmonica) is a semi-automated framework to assist the practitioner in the development process of a blockchain-based application, from its design to its implementation. As Figure 3.3 shows the framework is composed of two tools, BLockchain Automated DEcision process (BLADE) and Blockchain ApplicatioN Configurator (BANCO). BLADE is a recommendation engine of a blockchain technology and associated blockchain-based patterns. BANCO is a tool to generate parts of the blockchain application to build. This toolkit relies on a knowledge base, composed of three parts: (i) blockchain technologies, (ii) software patterns, and (iii) core assets (e.g. code samples, configuration files, etc.).

As shown in Figure 3.3, a practitioner (e.g. software engineer, software architect, etc.) can query BLADE to obtain the recommendation of a blockchain technology and blockchain-based software patterns (Step 1). If the practitioner also needs the generation of a product, the generated recommendations can be forwarded to BANCO for reuse (Step 2). Then, the user can configure the product with the help of the recommendations, and generate the product (Step 3). BANCO can also be used as a standalone tool, but it will not benefit from BLADE recommendations. Indeed, both tools can be used independently, but it is also possible to use them in conjunction to refine produced recommendations and artifacts.

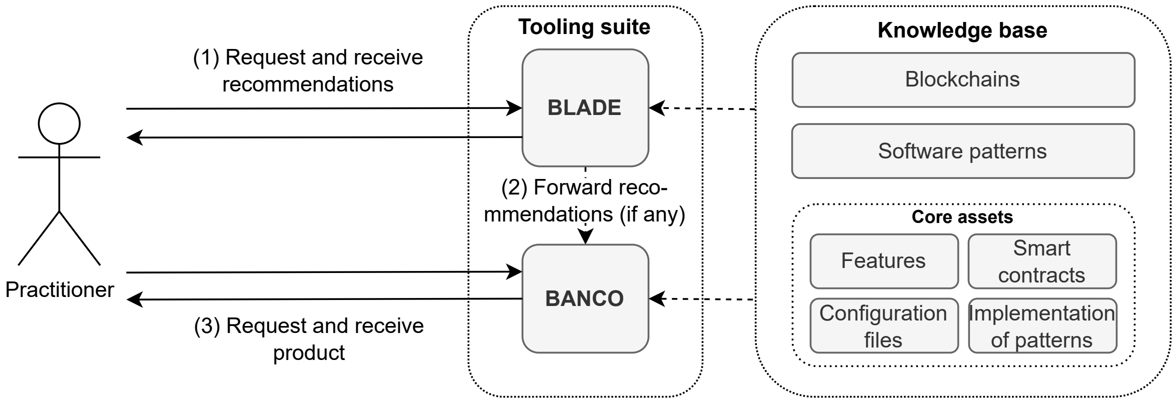

Tooling suite
Knowledge base
(1) Request and receive recommendations
BLADE
Blockchains
Software patterns
Practitioner
(2) Forward recommendations (if any)
Core assets
Features
Smart contracts
Configuration files
Implementation of patterns
BANCO
(3) Request and receive product

Figure 3.3: Harmonica framework overview.

Subsection 3.2.1 introduces the knowledge base of patterns and blockchain technologies. Then, Subsection 3.2.2 describes BLADE, the recommendation engine, and Subsection 3.2.3 presents BANCO, the tool in charge of generating parts of the blockchain application to build. Finally, Subsection 3.2.4 establishes the mapping between the framework itself and its elements to the elements described by the DSR method.

3.2.1 Knowledge Base

The first part of this framework is the knowledge base. As the recommendation process and the generation of code are knowledge-intensive activities, a knowledge base that supports the functioning of the framework's tools was created. The knowledge base is formalized and stored as an ontology. The knowledge base is constituted of two different parts. The first one contains 6 different blockchain technologies, described by 14 different attributes. These blockchains attributes have been collected throughout the construction of BLADE, the recommender tool. More details on its construction are given in Chapter 7.

The second part contains a collection of 120 unique blockchain-based patterns. This knowledge has been gathered by performing a SLR, that is a protocol to answer one or several research questions by systematically collecting and then reading research works. To carry the SLR, the Kitchenham et al. guidelines on completing a SLR in software engineering have been used (Kitchenham and Charters, 2007). In the blockchain-based software pattern ontology, the central element is the *Proposal* class. A proposal is a pattern introduced within a source (e.g. academic paper, technical report, ...). Throughout the SLR, 160 different proposals have been identified. As multiple proposals from different sources could express the same pattern, a class has been created for each identified pattern to regroup proposals. Thus, one pattern is linked to one or more proposal individuals.

Every proposal is described by a name, a context, a problem, and a solution. This format has been chosen as it has been identified in the SLR process that most papers do not follow a standard pattern format, such as the GoF format or the Alexandrian form (Gamma et al., 1995; Alexander, 1977). Proposals are also classified into comprehensive categories to facilitate their filtering and reuse: programming language, blockchain technology, and domain. In addition, they have been further classified using a taxonomy built during the SLR, composed of 4 main categories and 15 subcategories. Finally, every proposal can be linked to others through a series of relations: *Require*, *Benefits from*, *Variant of*, *Related to*, and *Created from*. Identifying the links between software patterns facilitates their appliance in other systems as it identifies potential conflicts with other patterns. It also improves the precision of software patterns recommender systems, as selecting a pattern might indicate to the recommender that other related patterns should be recommended too. Full details on the ontology structure and content is given in Chapter 6.

The resulting knowledge base supports the toolkit proposed in Harmonica: BLADE, the recommendation engine, and BANCO, the code generator. The knowledge base can also be leveraged as such by practitioners in other systems or tools.

3.2.2 BLADE - BLockchain Automated DEcision process

BLADE is a decision-making tool for guiding the selection of adequate blockchain technology and of blockchain-based software patterns. BLADE notably assists the design phase of software development, as it helps design the architecture of the application and its blockchain parts. BLADE also supports the implementation phase by guiding the user, that is a practitioner (e.g. software engineer/architect) into a collection of blockchain-based patterns for its application. The process of selecting blockchain-based patterns gives the user solutions to recurring problems that can be applied during the implementation phase.

The recommendation process of BLADE can be divided into three different parts: (i) Blockchain technology recommendation, (ii) Blockchain-based pattern recommendation, and (iii) Blockchain-based pattern selection.

Blockchain Technology Recommendation

The blockchain recommendation is based on 14 predefined blockchain-related NFRs. These NFRs are mapped within BLADE to specific blockchain attributes. For instance, a requirement for smart contracts might leads to only selecting the blockchains supporting Turing-complete smart contracts in the knowledge base. This requirement is mandatory to build the running example blockchain traceability application specified in Chapter 2.

BLADE users can express their preferences towards any NFR. The expression of preferences is under different labels: *Indifferent*, *Slighly desirable*, *Desirable*,*Highly desirable*, and *Extremely desirable*. These labels form a 5-point Likert scale, often used during surveys to let users express their feelings towards a question (Allen and Seaman, 2007). Requirements are expressed in a different way: the user can express a minimal, maximal, or specific value expected on the recommended blockchain. If a blockchain does not satisfy the requirement, it will be discarded from recommendations independently from its score.

The computation of the recommendation is made possible through the usage of the TOPSIS multi-criteria decision-making algorithm. TOPSIS is able, from a matrix of alternatives and weights, to generate a score between 0 and 1 for each alternative. The score represents the geometric distance between an alternative and the ideal solution, composed of the best scores of each criterion. Starting from the aforementioned inputs, TOPSIS returns a list of blockchain technologies, ordered by score. The user is then free to pick a technology, knowing the adequacy of each technology to its requirements.

By leveraging BLADE, any user can get a recommendation of a suitable blockchain technology for a given case. This recommendation can be used as such, or be forwarded into the next framework tool to improve further recommendation. More details on the construction of BLADE are given in Chapter 4.

Blockchain-based Patterns Recommendation

Besides recommending a blockchain technology, BLADE was designed to guide the selection of blockchain-based patterns. A recommendation process has been implemented for this purpose. The user begins by answering a collection of questions on the application to build. Each question expresses a design problem[1] related to a taxonomy category, as presented in Section 3.2.1. The user can answer using three different options: *Yes*, *No*, and *I don't know*. Thus, answering positively to a question means that the practitioner wants to address the underlying design problem, solved by the patterns classified inside the related category. For instance, the question "I want to store and manage on-chain data in any format (encrypted or clear)" addresses the design problem of storing data on the blockchain. Taking back the running example described in Chapter 2, the user has to answer *Yes*: traceability data are expected to be stored on-chain. Consequently, patterns that facilitate the storage of on-chain data might be recommended.

As the taxonomy forms a tree of categories, and each question is indirectly linked to a taxonomy category, the set of questions also forms a tree. Thus, when the user answers positively to a question, related subquestions might be asked. By extension, these subquestions cover parts of the design problem addressed by the initial question. Blockchain-based software patterns are classified under different subcategories of the taxonomy, that can be seen as tree leaves. To recommend a set of patterns to the user, a score is computed for each pattern based on the answers given beforehand. The result is then ordered and displayed to the user through a web platform for selection.

[1]The terminology "design problem" used in the context of BLADE must not be confused with the one that emanates from the DSR method.

Blockchain-based Patterns Selection

To complete the recommendations given on blockchain-based software patterns beforehand, BLADE allows the user to freely explore and then select blockchain-based patterns. On the web platform, the 120 identified pattern classes are displayed as clickable cards. The practitioner can then click a specific pattern to display every proposal linked to it. If the pattern corresponds to its needs, he can then decide to save it into his selection of patterns, already composed of several patterns selected during the recommendation phase. To guide the practitioner, patterns can be filtered on different aspects: *Blockchain*, *Domain* (e.g. supply-chain patterns), and *Language* (e.g. Solidity patterns). He can also filter the patterns by selecting a specific category of patterns, defined in the pattern taxonomy. Finally, the tool can leverage the blockchain recommendations made by BLADE to automatically filter patterns based on the chosen blockchain. BLADE further helps the practitioner in the design phase but also the implementation phase by (i) making accessible the knowledge on blockchain-based software patterns and (ii) allowing the practitioner to get suitable recommendations on the patterns to use. The construction of this tool is further described in Chapter 6.

3.2.3 BANCO - Blockchain ApplicatioN Configurator

The last part of this framework is BANCO, a web platform based on Software Product Line Engineering (SPLE) to configure and then generate ready-to-use blockchain applications. The main idea behind SPLE relies on the systematic reuse of code and other software artifacts such as design decisions (e.g. software patterns, models), requirements, and tests. Reusable artifacts are created during the domain engineering phase, then reused during the application engineering phase to compose software products. The family of software products that can be created from a Software Product Line (SPL) have common features (i.e. commonality) but also specific features that differentiate them (i.e. variability). These features are often expressed using a feature model, that describes all of the possible features and their constraints. This reuse allows the creation of software-intensive systems (so-called software products) at lower costs, in a shorter time, and with higher quality (Pohl, Böckle, and Van Der Linden, 2005). Reusing existing blockchain artifacts also eases the burden of non-blockchain expert practitioners by providing reusable elements on the shelf.

For the first iteration of BANCO, a feature model of blockchain-based (on-chain) traceability applications has been designed. The features composing this feature model have been identified based on a panel of existing studies on the topic. As such, the feature model expresses the different features that can usually be found in on-chain traceability applications. Nonetheless, BANCO can support other feature models.

In complement of the SPL approach, BANCO automates the application engineering phase with a configurator and a code generator (Krueger, 2009). The configuration is performed by a practitioner using a web platform, that embeds an interface to select desired features. The feature model is reused in the configuration phase of BANCO to guide the user in the possible choices when composing the application. It also prevents the user from selecting two or more conflicting features, thus avoiding the creation of an incorrect configuration.

Once the configuration is complete, a generator is able to create working blockchain products, using a set of templates. These templates are written in Solidity, a language to develop Ethereum smart contracts. Each feature expressed in the feature model corresponds to one or more code blocks within the templates. The code is then generated using a subtractive approach: non-selected features are discarded, and selected features are assembled together to form a suite of smart contracts that fits the configuration. A web application is also generated to set up and deploy the smart contracts, then interact with them. The construction of the feature model and the web platform as well as their validation is further described in Chapter 7.

3.2.4 Framework Mapping to Design Science Research

As the framework contains many elements, each of them is linked to a specific technical research question or a knowledge question, that emanates from the application of the DSR method. Figure 3.4 highlights the link between the framework's elements and research questions.

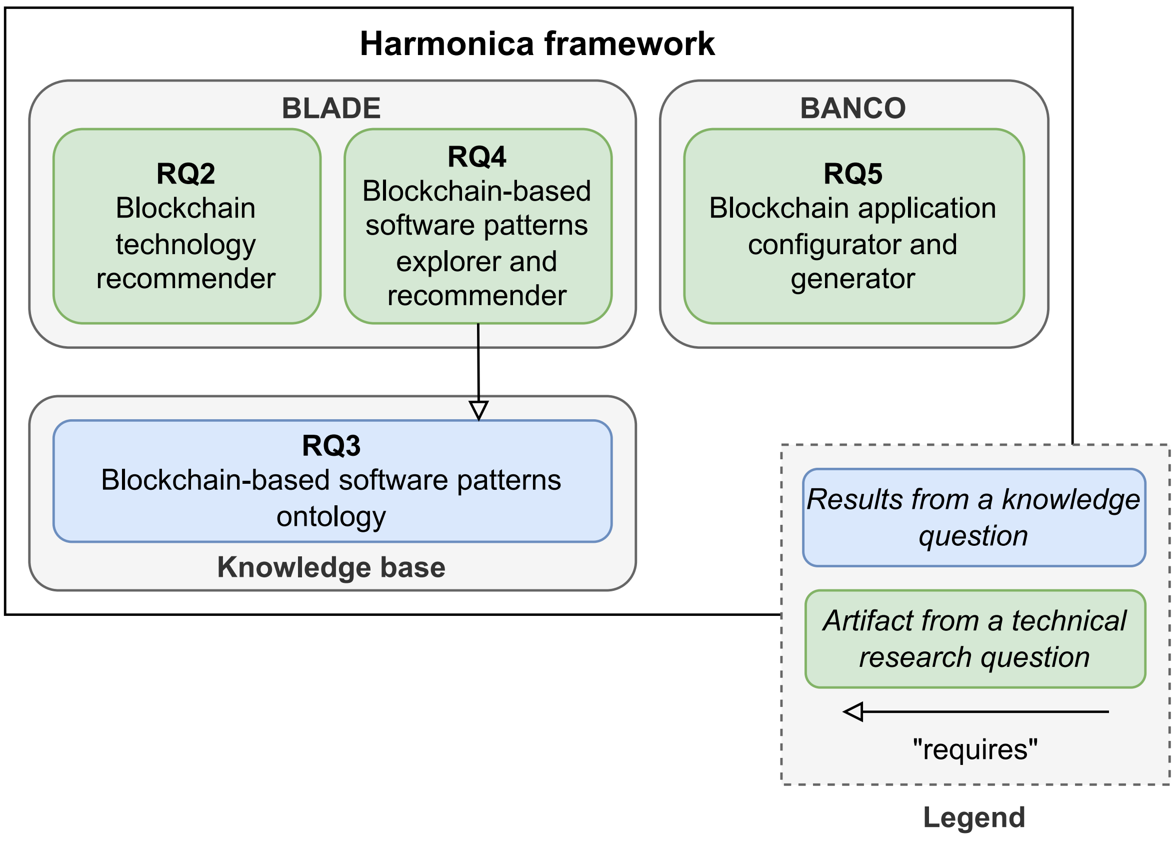

Harmonica framework
BLADE
RQ2 Blockchain technology recommender
RQ4 Blockchain-based software patterns explorer and recommender
BANCO
RQ5 Blockchain application configurator and generator
RQ3 Blockchain-based software patterns ontology
Knowledge base
Results from a knowledge question
Artifact from a technical research question
"requires"
Legend

Figure 3.4: Framework elements with regards to the DSR method.

The first tool of the framework, BLADE, is made of two artifacts: the blockchain technology recommender, and the blockchain-based software pattern explorer and recommender.

These artifacts are designed and implemented when respectively addressing the technical research questions RQ2 and RQ4. The second tool of the framework, BANCO, is an artifact by itself and is designed and implemented throughout the technical research question RQ3. Finally, the knowledge contained within the blockchain-based software patterns ontology emanates from answering the knowledge question RQ2.

As a result, the framework is both the composition of these artifacts and knowledge and an artifact by itself that addresses the research aim stated in Chapter 1, that is assisting the practitioner in the design and the implementation of blockchain applications.

3.3 Running Example

In the following chapters of this thesis, the construction of the artifacts composing the framework is presented. Although a big picture of the framework is given in this chapter, it may be difficult to understand how each tool may be used in practice by developers to design, then implement blockchain applications.

Therefore, this chapter introduces a running example to further guide the reader throughout the different contributions of this thesis. This running example is drawn from a case study of blockchain-based traceability of traditional Italian bread (*Carasau bread*) (Cocco et al., 2021). In each of the following chapters, the running example serves to illustrate how a blockchain application can be built using the Harmonica framework from its design to its implementation. In Chapter 5, the selection of a suitable blockchain is performed using BLADE, according to the running example requirements. Then, adequate patterns are recommended in Chapter 7 to ease the design of the running example application. Finally, the running example is implemented in Chapter 8 using BANCO to generate it.

3.3.1 Description

The *Carasau bread* is a traditional Italian bread from Sardinia, Italia. By being a regional product, the different participants of the bread supply-chain must follow a defined production protocol. First, the bread and its ingredients must be produced in Sardinia in specific conditions. To guarantee the quality of the product, the bread must be produced with a mechanical and physical process, from re-milled semolina of durum wheat and sea salt. The bread must also comply with strict hygienic-sanitary conditions, to guarantee product safety for the final consumer. In this case study, the Hazard Analysis and Critical Control Point (HACCP) system is used to define a systematic framework to identify and analyze the different hazards (e.g., biological, chemical, etc.) that can impact food safety (Cocco et al., 2021).

The *Carasau bread* production begins with the durum wheat production. To ensure that the produced wheat does not contain any toxin or chemical residue, the grain suppliers must first transfer appropriate documentation to discard these threats. If necessary, the milling industry can also run its own grain analysis upstream, to complement the documentation. However, the milling industry has to perform these tests downstream for every batch of wheat produced and periodically test the drinking water used in the production of the wheat. Once the wheat is produced, the bakery industry is able to produce the bread, that will be sold by retailers afterward. In complement to the measures already taken during production, safety measures are also taken during the transportation and storage of the products. The temperature and the humidity of the storage place and the transportation vehicle must stay within a defined range to avoid product deterioration. Thus, these metrics are often monitored to react accordingly if one of them crosses the threshold. The storage, bagging, and transportation of the products (wheat, bread, etc.) must also be done in healthy environments to avoid contamination or degradation of the products.

The blockchain technology is a good candidate to improve the traceability process and ensure full compliance to HACCP and the typical *Carasau bread* production process. Blockchain data transparency and immutability ensure that the audit trail of operations throughout the supply chain can be retraced for verification purposes. Data transparency also helps in enabling trust between the consumer and the producer, as the former can verify the provenance of its product and the fabrication steps. Finally, the decentralization of the traceability process forces all of the third parties to work together, instead of having one-third party in charge of the traceability application.

3.3.2 Requirements and Technical Considerations

To guide the design and the implementation of the traceability application, its requirements have been extracted from the user stories described in the case study paper, then refined using technical choices made by the authors in the description of their solution. These requirements have been specified following the guidelines from Pohl (Pohl, Böckle, and Van Der Linden, 2005).

Eight stakeholders are involved in the traceability application:

- Authority - the administrator of the system and supervisory body, represented by a specific regional Sardinian body.
- Seed producer - provides the seeds of durum wheat;
- Farmer - responsible for the seeding and harvesting of grains.
- Milling industry - produces the re-milled semolina of durum wheat.
- Bakery industry - produces the *Carasau bread*.
- Distributor - responsible for moving the output of the farmer from the farmer's site to the milling industry, the output of the milling industry from the milling industry's site to the bakery, and the output of the bakery from the bakery's site to the retailer.
- Retailer - receives then resells the *Carasau bread*.

- Consumer - final actor of the supply-chain and consumer of the *Carasau bread*.

Each user has a specific interest in the system to build. Where supply-chain participants will be interested in storing traceability data on their production, purchases, or sales, the consumer will be interested in the provenance of the bought bread. In the chosen case study, these interests have been individualized as user stories (Cocco et al., 2021). Table 3.1 lists the different user stories formalized by the case study.

The resulting functional requirements of the application are listed in Table 3.2, and divided into 5 different categories:

1. Traceability document storage - each supply-chain participant has the duty to store traceability-related documents to comply with HACCP.
2. Ownership transfer - these requirements specify the possible ownership transfers in the system between supply-chain participants.
3. IoT data record - to monitor the environment of the different supply-chain products, these requirements specify the participants that will record these data and the concerned places.
4. Traceability checking - these requirements specify who can access the traceability data and in which conditions.
5. Participants management - it should be possible for the authority (that administrates the system) to add new participants and grant or revoke read/write accesses.

In parallel, additional technical considerations coming from the case study were collected. Indeed, knowing the different technical choices made by the authors will help the comparison between the author's solution and the solution created from using Harmonica in Chapter 4, 6, and 7.

Regarding the blockchain used, the authors are mentioning Ethereum without specifying a specific consensus algorithm. As they also mention the willingness to save gas costs and to allow supply-chain transparency for the final consumer, it is assumed that the blockchain network used is the Ethereum mainnet.

Software patterns are also used in the design and implementation of the application. The *Oracle pattern* is mentioned, that is a component designed to push fresh data on the blockchain, as smart contracts cannot query data from outside the blockchain (Xu et al., 2018). The pattern collection for smart contract gas efficiency is also mentioned (Marchesi et al., 2020). Each operation performed on an Ethereum smart contract has an associated gas cost, that should be paid by the user in Ether. Therefore, these patterns are highly beneficial as they introduce good practices to save gas during the deployment or the execution of a smart contract, thus saving costs.

A final technical consideration is the usage of InterPlanetary File System (IPFS). IPFS is a decentralized peer-to-peer network for storing and sharing data[2]. In this case study, IPFS is

[2]https://ipfs.io/

Table 3.1: Case study user stories.

Stakeholder	User story
Seed producer	The seed producer stores technical information of his product (seeds), and data on their sale.
Farmer	The farmer stores data on the purchases of raw materials (seed), amount and technical information of the harvested grain, but also for example data on irrigation, fertilizing, and the sales of the harvested grain.
Farmer	The farmer transfers the ownership of his product, the durum wheat, to the distributor in order to deliver the product to the milling industry, and stores technical documentation on the products transferred.
Milling industry	The milling industry system stores details about the received amount of product (incoming grain/durum wheat batches) from distributors, and data concerning its production of flour (outgoing re-milled semolina/flour batches). The system of this industry also records information about hygienic-sanitary conditions in which it works, and about the temperature and humidity in its storage rooms.
Milling industry	The milling industry transfers the ownership of its product, the flour, to the distributor in order to deliver it to the bakery.
Bakery	The bakery system stores details about the received amount of product (incoming re-milled semolina batches) from distributors, and data concerning its production of bread (outgoing bread batches). In addition, as the milling industry system does, it records information about hygienic-sanitary conditions in which it works, and about the temperature and humidity in its storage rooms.
Bakery	The bakery system transfers the ownership of its outgoing batches, the Carasau bread, to the distributor in order to deliver them to the retailer for the sale.
Distributor	The distributor records information about hygienic-sanitary conditions of the means he works with, and about temperature and humidity in the means used.
Consumer	The consumer via a user-friendly app can retrieve data on the bought bread and can trace and verify each step along the supply chain.
All participants	All actors/systems record information to make available to other actors in the chain by using pdf and jpeg files. So at precise and predefined time intervals, data coming from sensors and optical cameras are automatically elaborated in order to obtain the files idoneous.
Authority	Authority manages and controls the reading and writing accesses in the system by the different actors and devices, and performs inspections to verify the conformity of the products and the work of each actor in the chain. The inspections can be performed by viewing the data documents stored by the nodes of the chain.

Table 3.2: Case study functional requirements.

Req. ID	**Requirement description**
R.1.1	The system shall allow a seed producer to store technical information on seeds.
R.1.2	The system shall allow a seed producer to store seed sales data.
R.1.3	The system shall allow a farmer to store seed purchase data.
R.1.4	The system shall allow a farmer to store crop cultivation data.
R.1.5	The system shall allow a farmer to store the technical documentation of produced durum wheat batches.
R.1.6	The system shall allow a milling industry to store details on received durum wheat and grain batches.
R.1.7	The system shall allow a milling industry to store wheat production data.
R.1.8	The system shall allow a milling industry to transfer the ownership of wheat batches to the distributor.
R.1.9	The system shall allow a bakery to store details on received floor batches.
R.1.10	The system shall allow a bakery to store bread production data.
R.1.11	The system shall allow a milling industry, distributor, or bakery to store hygienic-sanitary working conditions data.
R.2.1	The system shall allow a distributor to transfer the ownership of grains and durum wheat batches to the milling industry.
R.2.2	The system shall allow a distributor to transfer the ownership of flour batches to the bakery.
R.2.3	The system shall allow a distributor to transfer the ownership of Carasau bread batches to the retailer.
R.2.4	The system shall allow a farmer to transfer the ownership of durum wheat and grains batches to the distributor.
R.3.1	The system shall record in real-time the temperature and humidity of transportation vehicles.
R.3.2	The system shall record in real-time the temperature and humidity of storage rooms.
R.4.1	The system shall allow a consumer to retrace the different steps in the production of Carasau bread.
R.4.2	The system shall allow an authority to view the documents stored by supply-chain participants for inspection purposes.
R.5.1	The system shall allow an authority to add a new participant in the supply-chain participant's group.
R.5.2	The system shall allow an authority to grant read/white access to parts of the system to a participant.

used to store large files, as it would be a very expensive operation on-chain. Storing a file on IPFS returns a hash that is stored on-chain and can be used later to retrieve the file.

3.4 Conclusion

Although Chapter 1 states the research aim and objectives, it may be difficult to understand how precisely these objectives are addressed, and what is built as a result. This chapter presents different elements to introduce how research is carried out through this thesis, and what artifacts and knowledge are created to address the research aim.

First, an introduction to the Design Science Research method is given. The driving research questions of this thesis are then defined, reusing the distinction between technical research questions and knowledge questions proposed by the DSR method. Then, an overview of the envisioned solution is given. This overview helps the reader understand the different parts of the framework, giving a big picture of the aspects of the design and implementation of blockchain applications that are eased by the framework. It also introduces a mapping between defined technical research questions and knowledge questions, and the framework's elements. Throughout this mapping, the reader can understand how the different parts of the framework are built, through the different research questions.

Finally, a running example is introduced in-depth. This running example, that is a case study on blockchain-based traceability of traditional italian bread (Cocco et al., 2021), is used as a reference to introduce in the different chapters how the framework can be used in practice to ease the design and implementation of blockchain applications.

Where the goal of Chapter 2 was to give general background on blockchain, software patterns, and issues that threatens the developer in the design and implementation of blockchain applications, Chapter 3 introduces the research questions, the research method, and an overview of the contribution. From these chapters, it is expected that the reader has all the necessary material to understand the different contributions within the next chapters.

Chapter 4

Recommendation Engine for the Selection of an Adequate Blockchain Technology

Key takeaways

- An introduction to Multi-Criteria Decision Making methods is given.
- A conceptualization of the recommendation process is presented, that defines how requirements are submitted and processed to generate blockchain technology recommendations.
- A platform that implements this recommendation process was created. It allows the practitioner to obtain blockchain technology recommendations from requirements.
- This platform is the first part of BLADE, and the first artifact of the framework with regards to the DSR approach. It addresses the technical research question RQ2.

In the design phase of the software development lifecycle, the requirements specified by the user are mapped to a sound architecture to further guide the development during implementation. This architecture is defined by components, and their interfaces and behaviors are often expressed in models to facilitate the comprehension of the components by developers and stakeholders. When developing a blockchain-based application, software developers have to face the selection of a blockchain technology. Answering this question is far from being straightforward. Many blockchains have already been released from the inception of the field in 2008 with Bitcoin, all of them having various attributes and characteristics. For instance, the first major distinction lies in the access-control aspect of a blockchain network: should everybody be entitled to join and participate in the blockchain network (i.e. public blockchains), or only a set of predefined peers (i.e. private blockchains)? It might be tempting for a software architect to select a private blockchain, to contain data confidentiality among network participants and benefit from better performances, but such blockchain suffers from centralization.

This chapter addresses this issue throughout the following technical research question (**RQ2**): *How to design a blockchain technology recommendation platform that simplifies the completion of this task for non-experts so that practitioners can choose adequate blockchain recommendations during the design phase of blockchain application development?*

Also, two requirements for this can be derivated from this technical research question:

1. The platform must ease the task of blockchain technology selection for non-expert users.
2. The platform must recommend blockchain technologies that satisfy the user requirements.

Satisfying these requirements will imply a positive outcome for the stakeholders (i.e. the practitioners) goals, that motivate this research work.

To address this technical research question, a blockchain technology recommendation tool is proposed. This recommendation tool constitutes the first part of BLADE, a platform that aims to allow practitioners to obtain recommendations on blockchain technologies, then blockchain-based software patterns. In this chapter, the terms "BLADE", "platform", and "recommendation tool" are used interchangeably.

The platform is able to produce recommendations on the blockchain to use based on 14 different NFRs. By using the platform, practitioners will be assisted in the choice of a blockchain technology, as executing the recommendation engine will result in a blockchain that satisfies their needs.

The rest of this chapter is organized as follows. Section 4.1 introduces some background in decision-making methods. Section 4.2 presents the recommendation engine model, including its inputs, outputs, and internal logic. The implementation of the recommendation engine is described in detail in Section 4.3. The running example introduced in Chapter 2 is then reused in Section 4.4 to illustrate the functioning of the web platform. Section 4.5 reports a validation of this contribution with regards to DSR, that was performed using a Single-Case Mechanism Experiment (Wieringa, 2014), then Section 4.7 discusses existing threats to validity of the artifact construction. Section 4.8 presents and discussed related works, and Section 4.9 concludes the chapter by introducing future works on the specific topic.

4.1 Introduction to Multi-Criteria Decision-Making

Making decisions is an everyday problem: we all have to face many decisions on a daily basis. Where the majority of them are small and unimportant, we might have to face harder decisions with possible consequences and drawbacks, notably at work. For instance, such a decision could be "should we invest more money to modernize our information system?", or "should we buy this software for our company?". In this context, failing to find the right decision might lead to huge losses.

According to Harris et al., "Decision-making is the study of identifying and choosing alternatives based on the values and preferences of the decision maker. Making a decision implies that there are alternative choices to be considered, and in such a case we want not only to identify as many of these alternatives as possible but to choose the one that best fits with our goals, objectives, desires, values, and so on" (Harris, 1998).

This introduction will notably focus on MCDM. MCDM is one of the most known branch of decision-making methods, that concentrates on decision problems with discrete decision spaces (Triantaphyllou, 2000). A MCDM problem can be expressed as follows (Triantaphyllou, 2000):

Definition 6 *Let $A = A_i\, for\, i = 1, 2, 3, \dots, n$ be a (finite) set of decision alternatives and $G = g_j\, for\, j = 1, 2, 3, \dots, m$ a (finite) set of goals according to which the desirability of an action is judged. Determine the optimal alternative A^* with the highest degree of desirability with respect to all relevant goals g_j.*

To solve such a problem, an MCDM method have to be selected. However, there is no single method to solve all problems. Thus, the first step to solve an MCDM problem is to identify a suitable MCDM method from the ones available in the literature. Nevertheless, the selection of a method is also a decision problem in itself. Thus, many approaches have been proposed in the literature to decide on the best MCDM method to use (Kornyshova and Salinesi, 2007).

Many MCDM methods have been proposed in the literature, such as TOPSIS (Lai, Liu, and Hwang, 1994), Analytical Hierarchy Process (AHP) (Saaty, 1990), and ELimination Et Choix Traduisant la REalité (ELECTRE) (Figueira et al., 2013). In this introduction, TOPSIS will be described in-depth as it is used in this chapter.

TOPSIS

Technique for Order Preference by Similarity to the Ideal Solution (TOPSIS) is an algorithm that allows to rank alternatives that each have attributes of different types and scales, from a weighting matrix given as input (Lai, Liu, and Hwang, 1994). The main assumption of the TOPSIS algorithm is that the most relevant alternative a_m for a given choice set should be as close as possible to the positive ideal solution A^+ and as far away as possible from the negative ideal solution A^-. Several steps are required for the execution of the TOPSIS algorithm:

Matrix construction - Let m a collection of alternatives a. Each alternative a is defined by n attributes c. Those alternatives can be grouped in a matrix $X = \{x_{ij}\}$ for $\{i \in \mathbb{N} \mid 1 \leq i \leq m\}$ and $\{j \in \mathbb{N} \mid 1 \leq j \leq n\}$, x_{ij} representing the attribute c_j of the alternative a_i.

Matrix normalization and weight application - Normalize criteria that have different scales and units with each other. It is necessary to be able to make an accurate comparison. At this step, weights from user preferences ω_j are also applied.

$$v_{ij} = r_{ij} * \omega_j = \frac{x_{ij}}{\sqrt{\sum_{i=1}^{m} x_{ij}^2}} * \omega_j \tag{4.1}$$

Calculating ideal positive and negative solutions then measuring the distance with each alternative - By selecting the best and worst performance of each criterion in the normalized and weighted decision matrix, the ideal positive and negative solutions (resp. A^+ and A^-) can be computed to measure the distance of each alternative with those two solutions (resp. S^+ and S^-).

$$For A^+ = (v_1^+, ..., v_j^+), \tag{4.2}$$

$$Si^+ \triangleq \sqrt{\sum_{j=1}^{m} (v_{ij} - vj^+)^2} \tag{4.3}$$

$$For A^- = (v_1^-, ..., v_j^-), \tag{4.4}$$

$$Si^- \triangleq \sqrt{\sum_{j=1}^{m} (v_{ij} - vj^-)^2} \tag{4.5}$$

Calculating relative distance C_i with the ideal solution - This last step attributes a score to each alternative, that represents its distance from the ideal solution. Ordering the results creates a ranking allowing the selection of the best alternative among given alternatives and user preferences.

$$C_i = \frac{Si^-}{Si^+ + Si^-}$$

4.2 Decision Process Model

This section presents the decision process contained in the recommendation engine to determine adequate blockchains to use based on requirements. Figure 4.1 gives an overview of the recommendation engine. First, the practitioner has to provide requirements to the recommendation engine (Subsection 4.2.1). The recommendation engine also relies on a knowledge base to compute the recommendations, composed of 6 blockchain technologies and described by a collection of 14 attributes. The blockchain recommendation is then processed by forwarding the inputs to the decision process (Subsection 4.2.2).

4.2.1 Inputs

The accuracy of a multi-criteria decision support algorithm depends mostly on the input data. This subsection presents the blockchain alternatives and their attributes chosen to constitute the knowledge base, and the requirements that can be submitted by the user to compute recommendations.

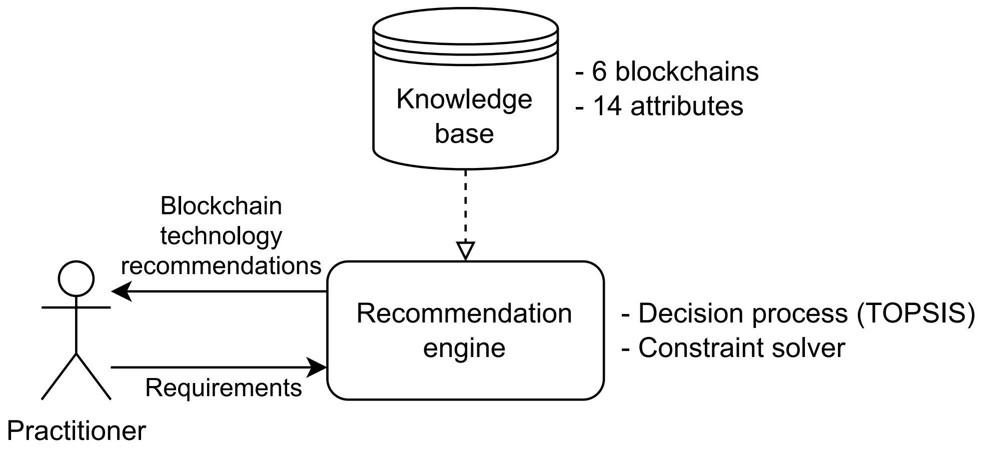


Figure 4.1: Recommendation engine overview.

Alternatives and attributes

To feed the decision support process, a knowledge base has been built, containing a set of blockchain alternatives a_m and their respective attributes c_n. Table 4.1 presents this set of these alternatives and attributes.

A set of six blockchain technologies have been considered to form the collection of alternatives used by the recommendation engine. In this set, three alternatives in this collection are public blockchain:

- Bitcoin (PoW: Proof-of-Work) (Introduced in Chapter 2).
- Ethereum (PoS: Proof-of-Stake) (Introduced in Chapter 2).
- Tezos (PoS: Proof-of-Stake), an open-source, self-upgradeable and energy-efficient blockchain technology[1].

In this context, selecting one of these technologies involves using the public main blockchain network associated with these technologies (e.g. using the Ethereum mainnet, in opposition to a self-deployed private network). They have been chosen as they were, at the time of this work, among the leading blockchain technologies in terms of capitalization (CoinMarketCap, 2020).

The three other blockchain technologies are private:

- Hyperledger Fabric (Raft), the most-known Distributed Ledger Technology (DLT) of the Hyperledger project[2].
- Corda (Practical Byzantine Fault Tolerance), an open-source blockchain project for business carried by R3[3].

[1]https://tezos.foundation/
[2]https://www.hyperledger.org/use/distributed-ledgers
[3]https://www.corda.net/

	Bitcoin	**Ethereum**	**Ethereum**	**H.F.**	**Corda**	**Tezos**
Consensus algorithm	PoW	PoS	PoA	Raft	PBFT	PoS
Public	Yes	Yes	No	No	No	Yes
Permissionned	No	No	No	Yes	Yes	No
Native encryption	No	No	No	Yes	Yes	No
Throughput (tx/s)	3,8	15	∓ 100	∓ 1000	∓ 1000	30
Latency (s)	3600	180	∓ 10	<1	<1	60
Energy efficient	No	Yes	Yes	Yes	Yes	Yes
Byzantine fault-tolerant	50%	50%	33%	0%	33%	33%
Turing-complete smart contracts	No	Yes	Yes	Yes	Yes	Yes
Cryptocurrencies	Yes	Yes	Yes	No	No	Yes
Storage element	Basic	Adv.	Adv.	Adv.	Adv.	Adv.
Computing element	No	Adv.	Adv.	Adv.	Adv.	Adv.
Asset management element	Basic	Adv.	Adv.	Adv.	Adv.	Adv.
Software connector	No	Adv.	Adv.	Adv.	Adv.	Adv.
Learning curve	Low	Medium	Medium	Very high	Very high	Very high

Table 4.1: Chosen alternatives and attributes (Adv.: Advanced, H.F.: Hyperledger Fabric).

- Ethereum, using the PoA (Proof-of-Authority) algorithm that gives the right to create blocks only to a specified subset of participants.

They are among the most-used technologies in companies (Polge, Robert, and Le Traon, 2021). Additionally, this specific subset has been chosen as it contains blockchain technologies that are very different from each other. As this subset is sufficient in the proposition of BLADE, it may be extended by including other blockchain technologies in future works.

Regarding the attributes, a set of criteria was chosen, categorized by the different macro-characteristics[4]:

- **Functional suitability** - Smart contracts, Cryptocurrencies, Storage, Computational element, Software connector, Asset management
- **Performance efficiency** - Throughput, Latency, Energy efficiency
- **Security** - Access management, Permission management, Native encryption
- **Reliability** - Byzantine-fault tolerance
- **Usability** - Learning curve

[4]Here, the term macro-characteristic refers to the 7 categories of software quality (Security, Reliability, ...) under which the software quality attributes are introduced

These macro-characteristics relate to software quality proposed by the ISO 25010, a standard defining the different quality attributes to be considered in order to guarantee the quality of a system or software during its implementation (Haoues et al., 2017). The attributes were chosen for their relevance when selecting a blockchain, but also for the possibility to assign them a numeric value that can be reused by the decision engine. Therefore, these attributes are not only specific to blockchain technology, they apply to the whole system quality.

The values given to each attribute of each blockchain technology in the knowledge base come from different sources: studies (Belotti et al., 2019), white papers (Brown et al., 2016; Nakamoto, 2008; Wood et al., 2014), technical documentation, and scientific literature (Androulaki et al., 2018). Some of these values are fuzzy (marked by the symbol $\mp$), because they are subject to variations in the topology and configuration of the blockchain network as well as in the technical characteristics of the nodes that make it up (CPU, RAM...). Their value is therefore built from known attributes, such as the supported consensus algorithm (a Byzantine fault-tolerant algorithm like Bitcoin's PoW algorithm will have a lower transaction throughput than a fault-tolerant algorithm like Raft used by Hyperledger Fabric).Nevertheless, these values can be fixed when the blockchain parameters are known.

As BLADE has to take into account assets already present in the company (such as technical infrastructure or business process models), a future objective will be to perform performance tests in order to be able to give a fixed value to the variable attributes depending on the given context. This knowledge base will also change over time. The values of the attributes of the different blockchains chosen can be modified (update of one of the elements of a blockchain). As these variations can have an impact on the choice of the best alternative by BLADE, it will be necessary to evaluate the knowledge base's recentness in order to determine if the recommendation is relevant at a given time.

Requirements and preferences

In order to obtain a blockchain recommendation that meets the user's expectations, the decision process within the recommendation engine has to take into account a collection of criteria. Each criterion of this collection corresponds to a specific attribute of the knowledge base. In BLADE, an interface is provided to express requirements as numerical or litteral values that are forwarded as criteria for the decision process. For instance, the requirement "The blockchain network shall be public" can be given to BLADE as a single litteral value ("Public").

To further refine the decision process, the importance of requirements is also taken into account. This importance can be expressed in two ways: the mandatoriness of a requirement, and its preference level. The mandatoriness can be expressed by marking a criterion as *Required* or *Unwanted*. When making a decision, an alternative whose attribute does not meet either of these two requirements would be automatically disqualified from the possible alternatives, regardless of its score obtained by running the multi-criteria decision support algorithm.

The user can also indicate a preference level for a specific criterion c_n, by means of a rating scale taking the form of a sequence of labels, each label is linked to a numerical value (defined in Table 4.2). The choice of a label thus makes it possible to obtain a preference value p_n for each of the c_n criteria. In order to obtain the weights of each criterion ω_n so that the sum of these weights is equal to 1, each preference p_n for a criterion is divided by the sum of the preferences.

Label	**Preference value p_n**
Extremely desirable	4
Greatly desirable	3
Desirable	2
Slightly desirable	1
Indifferent	0

Table 4.2: Ranking scale associating labels and preference values.

4.2.2 Decision Process

The decision process is constituted of two parts: the processing of criteria and the decision-making algorithm. For each criterion modified by the user, recommendations are computed in real-time. Thus, the user can visualize the impact of one criterion on the recommendation results.

Criterion filtering

An overview of criteria processing is given in Figure 4.2.

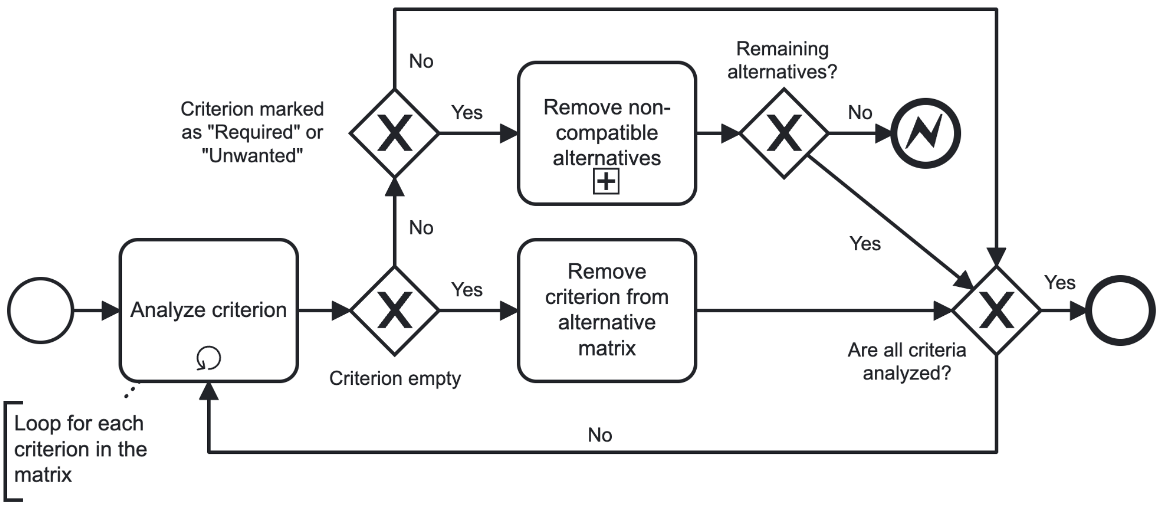

No
Remaining alternatives?
Criterion marked as "Required" or "Unwanted"
Yes
Remove non-compatible alternatives
No
No
Yes
Analyze criterion
Yes
Remove criterion from alternative matrix
Yes
Criterion empty
Are all criteria analyzed?
No
Loop for each criterion in the matrix

Figure 4.2: Criteria processing phase.

In this first phase, criteria and alternatives are retrieved by the recommendation engine. The alternatives are expressed as a 14-by-6 matrix, each column representing a blockchain technology and each line a specific attribute. Each criterion is analyzed for filtering purposes.

A first filtering is performed on the attributes: if a criterion is left unspecified by the user, it is possible to remove the corresponding attribute from the matrix. Indeed, this will have no impact on the final result, as it would be interpreted by the decision engine as a weight of 0.

The second filtering is performed on the alternatives: if a criterion marked as *Required* or *Unwanted* is not met by an alternative, the latter is automatically disqualified for the recommendation, regardless of the score it might have received using the decision algorithm that follows. For a criterion that is not a boolean, the user also has to specify an extremum value. As an example, if a certain number of transactions per second is required, alternatives that do not meet the threshold value will be disqualified. However, if no alternative remains after the selection of a criterion, an error message is displayed, as the user requirement cannot be satisfied by any blockchain technology in the knowledge base. This filtering phase results in a shortened alternative matrix, that facilitates the computation of recommendations by the decision process.

Decision process

The second phase consists of the decision process between the remaining alternatives. The decision engine relies on TOPSIS to compute the recommendations, a decision-making algorithm presented in Section 4.1. The choice of this algorithm was guided by a study presenting state-of-the-art of studies concerning the choice of a multicriteria decision support method (Kornyshova and Salinesi, 2007). The authors propose a decision framework including different properties to focus on when choosing a multicriteria decision support method.

The researcher has chosen the TOPSIS method for the decision process, in particular as it supports the multi-criteria analysis of numerous and varied attributes (it is the case when comparing two blockchains) while being simple to implement and precise in the decision. It also allows taking into account user-defined weights, which is required given the operating mode of the recommendation engine. This is for example not the case of the Condorcet method, or Borda (Zwicker, 2016), where only the attributes would have played a role in the selection of the best alternative. Also, another potential candidate for this tool is Analytical Hierarchy Process (Podvezko et al., 2009). Possessing a large number of similarities with TOPSIS, the latter method was nonetheless selected for its greater ease of entering weights. Indeed, AHP requires comparing the attributes of two to two to express the importance that one attribute has compared to another for the user.

4.3 Implementation

This section details the implementation of the recommendation engine, in a tool deployed online[5] and available as open-source on GitHub[6]. In this subsection, the implementation of each part is presented: (1) the knowledge base, (2) the API containing the different solvers

[5]https://recommender.blade-blockchain.eu/
[6]https://github.com/nicoSix/blade-project

composing BLADE, and (3) the web platform allowing the easy input of requirements. The solvers are also presented in more detail in their respective subsections.

4.3.1 Tool Architecture and Implementation

To implement this tool, a 3-tier architectural model is applied, based on the separation between client, server, and data (Bass, Clements, and Kazman, 2003). Here, the data layer will is the knowledge base. Only accessible by requesting the server, it contains the blockchain technology alternatives and their respective attributes. For this purpose, three collections were created:

1. The first one contains the set of alternatives stored as documents. Each alternative is defined by its name and its consensus algorithm as these two attributes are sufficient to identify a blockchain among the existing ones. The alternatives also contain another document that defines the 14 attributes used in the decision support.
2. A second collection contains all the attributes available for decision. An attribute is defined by its label (e.g. latency), its cost (a variable taking the value 0 or 1 and allowing the solver to know whether to maximize or minimize the goal), and its type (e.g. numeric, boolean, ...). The use of such a structure makes it easier to maintain the knowledge base over time. Indeed, the alternatives introduced in the knowledge base should conform to the existing model to be added.
3. Finally, a third collection allows storing literal values and their numerical equivalence. Indeed, some attributes of the alternatives can be given a literal value (e.g. *Very low*, *Medium*, ...) instead of a numerical value. These literal values are stored in the knowledge base, and associated with a numerical value. This also makes it easier to update the knowledge base, by allowing the numerical value of an attribute expressed as a literal value to be changed for all alternatives at once.

The server part is an API (Application Programming Interface), developed in Python and using the Flask framework[7]. The main advantage of dividing the application into layers, and thus by the independence of the server towards the client, is the possibility of reusing the API in other applications. Therefore, the API can be integrated into future works, presented in Section 4.9. This API allows inbound requests to compute the scores based on inputs, that is the decision process, and another solver allowing the identification of exclusion dependencies while selecting requirements on client-side (introduced in Subsection 4.3.2 and Subsection 4.3.3).

Finally, the client is a web platform written in Javascript and based on the React framework[8]. This framework facilitates the design and implementation of single-paged applications through the definition of components and their associated states. It has been chosen for its compatibility with the 3-tier layer approach, and its capacity to propose a reactive and

[7]https://flask.palletsprojects.com/en/1.1.x/
[8]https://fr.reactjs.org/

efficient platform for users to submit their requirements. Figure 4.3 shows a screenshot of the interface proposed to the user for the selection of requirements and preferences.

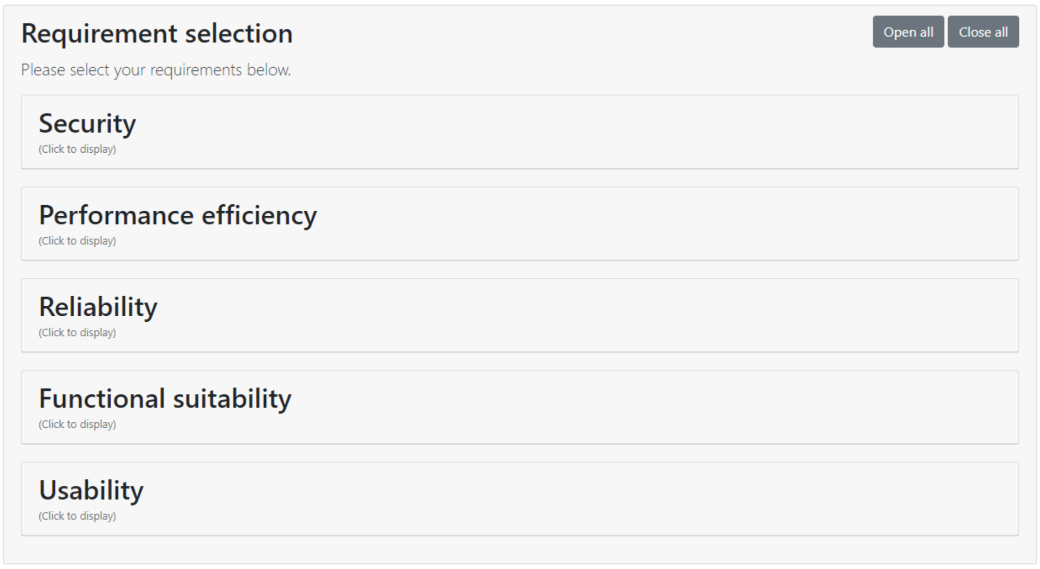


Figure 4.3: Screenshot of requirements selection interface in BLADE.

The selection is made in the left panel, entitled *Requirement selection*. As presented in the Subsection 4.2.1, the attributes are grouped by software quality macro-characteristics from the ISO 25010 standard. The user can unfold the different panels corresponding to these macro-characteristics to display the selection menus for each of the available attributes. The user can then enter her level of preference if the attribute is marked as *Required*, as well as the desired value. The results are displayed in real-time, in the right panel of the interface. A table is displayed, containing the alternatives classified by score. A history of user selections is also displayed, to summarize the attributes taken into account in the recommendations made by the tool. It also allows the user to remove preferences and requirements to alter the recommendations, if necessary.

4.3.2 Score Generation

With a file containing the user's requirements and preferences as input, the score generation engine is responsible for calculating a score for each of the alternatives using their characteristics defined in the knowledge base. When a decision has to be made by the decision support process, a request is made via the API to the server, which transmits it to the engine. As the scores are transmitted in JSON format, it will be responsible for changing the format of the data structure into the format expected by the tool. Then, the engine is in charge of calculating the score of each of the alternatives and disqualifying those that do not meet the input requirements as defined in detail in Section 4.2. The latter returns the results to the server, which in turn returns the results to the client. This allows the updating of the table containing the scores for each of the alternatives in the tool.

4.3.3 Dependency Model Generation Engine

When making a decision, the user might enter conflicting requirements. On the platform, a requirement is conflicting with another when the selection of both requirements disqualifies all alternatives in the decision process. Allowing the user to enter conflicting requirements is not something that is desirable, as the purpose of this tool is to provide decision support from the selection of requirements to the display of results. In order to overcome this problem, an engine has been implemented on the server-side, allowing the generation of a dependency model for the application. Here, a dependency is said to be *exclusionary*, i.e. the model indicates, when selecting a requirement, the ones that are directly in conflict (thus to prevent obtaining at least one valid alternative if both are selected).

From the knowledge base, the engine is in charge of retrieving all the attributes available for the alternatives and generating the corresponding pairs. For each existing attribute, the engine will retrieve all the values it can take, by iterating over the available alternatives. It will then create pairs from all these values so that for a given attribute value, there is a number of pairs equal to the number of existing values. Then, a backtracking algorithm is responsible for applying two constraints to the pairs: (1) a pair cannot contain two identical attributes with the same value, (2) none of the alternatives contained in the knowledge base must be able to satisfy both requirements formulated in the pair.

Formally expressed, supposing an alternative in a set of alternatives $a \in A$, C_n an engine constraint (r_a, r_b) a value pair, the following constraints are posed (Eq. 4.6):

$$\begin{aligned} C_1 \text{ satisfied } &\Leftrightarrow \nexists a \in A, (r_a, r_b) \in a, \\ C_2 \text{ satisfied } &\Leftrightarrow r_a \neq r_b. \end{aligned} \tag{4.6}$$

Failure to satisfy one of the constraints automatically results in the removal of a pair from the initial list of pairs. At the end of the execution of the algorithm, the result is a set of pairs representing all of the conflicting requirements.

This dependency model is stored in the database and then reused at each requirement selection by the user. Indeed, when selecting a requirement, the client will go through this dependency model and automatically grey out the different values in the form corresponding to incompatible choices (because they make it impossible to choose at least one alternative). An indication of conflicting values will also be given to the user so that he can adapt his requirements if necessary.

4.4 Running Example

In this section, the running example given in Chapter 2 is reused to illustrate how BLADE can be applied to a real-life example. First, the requirements introduced in the running example will be reused to determine the values for the different attributes to fill on the web platform. Then, the recommendations will be computed based on these attributes and discussed.

4.4.1 BLADE Requirements and Preferences

Table 4.3 lists the different values entered in BLADE w.r.t. the requirements and the context given in the running example. For the *Security* macro-characteristic, the permissionned attribute has been required to be set to *No*. Indeed, it is mentioned in the running example study that the application should rely on a permissionless blockchain, since the main goal is to render all information around the production of the Carasau bread completely transparent, without possible tampering from privileged parties. Regarding the *Performance* macro-characteristic, the preference towards throughput and latency has been set to *Desirable*. Although there is no explicit mention of a performance need by the running example study, it is appreciable that for two equivalent blockchain technologies, the most-efficient one is picked to support the future scalability of the system. Finally, the *Functionnality* aspect englobes three defined attributes. As the *Carasau bread* application should support the ingestion of IoT data, the creation of records, and other complex features, smart contracts are required elements in the application. Thus, its attribute in BLADE has been set to *Required* and *Yes*. The storage element attribute has been required and set to *Advanced*, as the main purpose of the application is to store data about the production of *Carasau bread* or reference to external data stored in IPFS. The asset management element attribute has also been required and set to *Basic*, as the requirements mention the transfer of asset property from one participant to another.

Attributes	Requirements	Required value	Preferences
Public	None		Indifferent
Permissionned	Required	No	Extremely desirable
Native encryption	None		Indifferent
Throughput (tx/s)	None		Desirable
Latency (s)	None		Desirable
Energy efficiency	None		Indifferent
Byzantine fault tolerance	None		Indifferent
Smart contracts	Required	Yes	Extremely desirable
Cryptocurrencies	None		Indifferent
Storage element	Required	Advanced	Extremely desirable
Computational element	None		Indifferent
Asset management	Required	Basic	Extremely desirable
Software connector	None		Indifferent
Learning curve	None		Indifferent

Table 4.3: Carasau bread application requirements and preferences.

4.4.2 Results

The execution of the decision engine has led to the disqualification of three blockchains (Hyperledger Fabric, Corda, and Bitcoin). Hyperledger Fabric and Corda both are permissioned engine, thus not compatible with the need for a permissionless blockchain. Regarding Bitcoin, it has also been discarded as it does not support Turing-complete smart contracts. As a result, two blockchains (with their consensus algorithm) have been ranked equally: Ethereum (PoS) and Ethereum (PoA). Indeed, there is no defined attribute that discriminates one blockchain/consensus algorithm pair from another. Where the first one represents the mainnet, that is the public version of Ethereum accessible by anybody, the second one has to be set up from scratch in a private infrastructure. Nonetheless, this recommendation concords with the running example study, as Ethereum was also the choice made by the authors.

4.5 Validation

In this section, the third step of addressing a technical research question is carried, that is artifact validation. This step consists in assessing that it contributes to the stakeholder's goals in a problem context, by predicting the effects of the treatment on this context if it is implemented (Wieringa, 2014). In this first artifact, the goal is the selection of a blockchain technology based on requirements. Two requirements drive this artifact: it should ease the selection of a blockchain technology for non-experts, and produce adequate recommendations.

The first requirement is difficult to validate, as it is hard to assess the "easiness" of selecting a blockchain technology. Yet, the usage of the platform to obtain recommendations for the running example and this section has led to two different arguments for validating this requirement. First, the task of reusing existing requirements to fill BLADE's inputs was straightforward in both cases. Without this platform, it may have been harder to find what requirements have a role in the selection of blockchain technologies. Then, the task of selecting a blockchain technology was also easier, as it did not require fetching existing technologies manually to find the most adequate ones. About ten minutes were enough to select BLADE inputs from requirements and obtain recommendations.

For the second requirement, the adequateness of recommendations can be assessed more easily. In order to test and validate the approach in this regard, a Single-Case Mechanism Experiment (SCME) will be carried out. A SCME is a useful approach for validation research, as it exposes the design to a controlled stimuli and analyze in detail the result (Wieringa, 2014). For instance, such an experiment may be carried out by implementing a prototype from the design, building a model of its intended context, and feeding its test scenarios to observe its responses.

In this section, a SCME is carried following the aforementioned example by using an existing study as the context (Longo et al., 2019). Throughout this paper, this study will be called a "reference study". This study proposes to introduce a blockchain system to a supply chain in order to enable data sharing between different actors. First, BLADE is used to obtain a blockchain technology recommendation, that is compared to the decision taken by the authors of the reference study. Then, a benchmark is performed to assess that the performance of the chosen blockchain matches the requirements of the blockchain-based supply chain system to build. These two steps aim to validate that (1) BLADE proposes a blockchain technology in adequation with the one employed by the reference study researchers and (2) that the chosen blockchain technology fulfills the performance requirements of the reference study.

4.5.1 Big-Box Scenario

The supply chain that was modeled in this work consists of a network of Big-Box chain retailers, as well as three wholesalers that supply their stores. Because the Big-Box retailers are grouped together in the same organization, the study considers that there is real-time, transparent, and reliable data sharing among the stores. However, the retailers are still competing with each other, as they operate in the same geographic area and all offer the same product lines. Customers arrive at the store and select products and their respective quantities. If the store's stock is sufficient to satisfy the demand, the product is reserved in the quantity requested; if not, a partial reservation is offered; the unfulfilled demand is used to calculate replenishments. The inventory is taken before the opening of the stores; if an order is needed then the retailer can choose one of the wholesalers to supply, taking into account the supply time, the current demand, and the instantaneous quantity available for the desired products. If the quantity of a product held by a wholesaler is not sufficient for all retailers, then it is shared equally.

In this context, sharing the aggregate demand of different retailers among wholesalers could make it easier to predict the stock to be built up to meet retailers' demands. However, the actors in this system remain competing with each other and therefore do not trust each other. Thus, the study proposes the implementation of a blockchain allowing the recording of data related to the supply chain (notably market demand) in the form of a hash value, as well as the different third parties having access to this data (if they are authorized by the blockchain, they can directly make a request to obtain this data from the third party who recorded it). The storage of this value allows to attest to the veracity of the data transmitted between third parties, they can now trust each other.

4.5.2 Big-Box Client Requirements

In order to be able to select a blockchain using the recommendation engine, the quality attributes, as well as the requirements and preferences for these attributes (Subsection 4.2.1), need to be identified. For this purpose, a textual requirements table is drawn up from the content of the reference study, together with a summary of the actors and systems presented in or derived from the reference study. These requirements have been formulated following Pohl's guidelines (Pohl, Böckle, and Van Der Linden, 2005). This is then used to partially determine the preferences and requirements that the user would have chosen as inputs for BLADE.

Functional Requirements

First and foremost, the textual requirements of the blockchain system to be designed have to be extracted. In these requirements, 6 different actors are introduced:

- BigBox: the company driving the retail network and the blockchain project.
- Wholesaler: buys goods from manufacturers (unspecified) and resells them to BigBox retailers.
- Retailer: manages one or more shops within the BigBox company network.
- Consortium: a group of wholesalers and retailers enabling the use and ensuring the proper functioning of the blockchain.
- Consortium member: a wholesaler or retailer with the right to register data on the blockchain and vote for the acceptance of new members.
- Candidate: a wholesaler or retailer who has applied to join the consortium.

These actors will interact with the system, that contains two main parts: (1) the blockchain, a network composed of nodes, which stores the smart-contract necessary for the proper functioning of the application proposed by the reference study, and (2) the "off-chain" application to build, allowing the different actors to interact with the blockchain and store stock information.

Once these actors and systems are defined, it is possible to analyze the different requirements for the reference study. Table 4.4 details these requirements by category and displays the dependencies between them. These functional requirements will guide the elicitation of the NFRs, used by BLADE during the decision process.

BLADE requirements and preferences

From the textual requirements, it is possible to formalize the preferences and requirements that are submitted to BLADE. This section summarizes each of the BLADE macro-characteristics and makes explicit the choices made in them with respect to the textual requirements (Table 4.5).

Security - As the data stored in the blockchain is simply metadata that does not contain personal information, it is not considered sensitive, nor is the identity of third parties masked by their address. It is therefore possible to use a public blockchain (which is the initial choice of the study), without data encryption. As permissions are managed at the level of the smart contract, it is not necessary to have a blockchain that supports permissions management. By deduction, since these properties are not important in this context, they are all marked as *Indifferent* in the input table.

Category	ID	Requirement	Linked to
Consortium member management	1.1	While an application for addition to the consortium is in progress, the off-chain application shall register on the blockchain the vote of a consortium member for this session if this member has not yet voted and if he is authenticated via his private key.	
	1.2	The blockchain should only accept applications from wholesalers and retailers affiliated with the BigBox company.	[1.1]
	1.3	When an absolute majority of votes in favor of accepting the candidate is obtained, the blockchain shall add the accepted candidate to the list of consortium members.	
Data publication	2.1	When a retailer member of the consortium requests it, the off-chain application shall write in the blockchain the metadata associated with the state of the stock at time T, if the data has been recorded in database beforehand.	[2.2]
	2.2	When a retailer member of the consortium requests it, the off-chain application shall record in its database the data associated with the state of the stock at the moment T.	
	2.3	Each day, the off-chain application shall publish metadata about the inventory information of each retailer in the network.	
Data retrieval	3.1	When requested by a consortium member, the off-chain application shall retrieve from the blockchain the metadata associated with the status of an inventory for a retailer at a given date.	
	3.2	When a retailer member of the consortium requests it, the off-chain application shall retrieve the data associated with the state of a stock for a retailer at a given date, using the metadata retrieved beforehand.	[3.1]
Blockchain properties	4.1	If less than 1/3 of the nodes comprising the blockchain are faulty, the blockchain shall be able to process at least 20 simultaneous transactions without a performance loss of more than 5%.	
	4.2	The blockchain shall support the execution of "Turing-complete" smart contracts.	[1.1] [1.3] [2.1] [2.3]

Table 4.4: Requirements for the Big Box reference study.

Attributes	Requirements	Required value	Preferences
Public	None		Indifferent
Permissionned	None		Indifferent
Native encryption	None		Indifferent
Throughput (tx/s)	None		Indifferent
Latency (s)	None		Weakly desirable
Energy efficiency	None		Extremely desirable
Byzantine fault tolerance	Required	$\geq$33,33 %	Desirable
Smart contracts	Required	Yes	Indifferent
Cryptocurrencies	None		Indifferent
Storage element	Required	Advanced	Indifferent
Computational element	None		Indifferent
Asset management	None		Indifferent
Software connector	None		Indifferent
Learning curve	None		Desirable

Table 4.5: Submitted requirements and preferences.

Performance efficiency - The blockchain system does not need to be capable of handling a high volume of transactions per second (differentiated from the number of transactions per second that can be submitted as input) and low latency. Nevertheless, since a low latency can be beneficial to the user experience, it has been set to *Slighly desirable*. As for energy efficiency, this is a particularly interesting property from a cost reduction perspective, one of the reference study goals. Using public blockchains with heavyweight consensus algorithms (such as PoW) is very energy intensive. Therefore, the preference *Greatly desirable* for this property.

Fiability - Since actors do not trust each other, it is essential to have a Byzantine fault tolerance percentage, that indicates the system is able to function properly for a certain number of nodes that may behave adversely. A percentage of at least 33.3% has been chosen, that guarantees the good continuity of the blockchain network for a number of faulty nodes $f + 1 < \frac{n}{3}$, n being the number of total nodes constituting the network.

Functional suitability - To meet the objectives of the defined topic, the blockchain should be able to take the form of a storage element to hold the retailers' data as well as support the administration of it, de facto through smart contracts. These two attributes are therefore defined as *Advanced* as well as *Required* respectively. The other functionalities are not required, thus they are marked as *Indifferent*.

Usability - Finally, the last chosen attribute is the learning curve: in a context where blockchain should enable to save costs associated with the supply chain operation, as well as support a low-complexity application, using a technology whose mechanics are easy to learn can be an advantage. It has been marked as *Desirable*.

The compilation of selected values leads to Table 4.5, entered as input later to execute the decision process. These values also allow satisfying the constraints defined in the Subsection

4.3.3, thus making their submission into BLADE possible.

4.5.3 Results

Running the automated process eliminates the Bitcoin alternative, as it does not support smart contracts, as well as the Hyperledger Fabric alternative, as it does not tolerate eventual Byzantine faults. It results in two matrices, one containing the weights and the other the possible alternatives (resp. Ethereum-PoS, Ethereum-PoA, Corda and Tezos). Knowing that a weight of 0 for a given attribute makes it insignificant in the calculation of the score of each alternative, it is possible to simplify these matrices for the values defined in eq. 4.7 and eq. 4.8.

$$W = \begin{pmatrix} 0.25 \\ 0.75 \\ 0.5 \\ 0.5 \end{pmatrix} \tag{4.7}$$

$$A = \begin{pmatrix} 180 & 10 & 1 & 60 \\ 0 & 1 & 1 & 1 \\ 0.5 & 0.33 & 0.33 & 0.33 \\ 0.4 & 0.4 & 0.8 & 0.8 \end{pmatrix} \tag{4.8}$$

This is followed by the execution of the decision support algorithm, that proposes the following results (Table 4.6). The decision algorithm considers the Ethereum-PoA alternative as the most suitable alternative in the given selection. Indeed, its score is the closest to 1 (positive ideal solution) of the four alternatives.

Alternative	Score
Ethereum, PoA	0.98054669
Corda, PBFT	0.78586689
Tezos, PoS	0.22030198
Ethereum, PoS	0.21413310
Hyperledger Fabric, Raft	Disqualified
Bitcoin, PoW	Disqualified

Table 4.6: Decision process execution results.

4.5.4 Recommended Solution Validation

The previous subsection showed, according to BLADE, that the most suitable solution for the problem at hand is Ethereum-PoA. To confirm the relevance of the solution, this subsection aims at evaluating the robustness and the performance of an Ethereum PoA-based network through a tool allowing to test of its performance, developed in this sense. This tool, accessible in open-source[9], allows the semi-automatic execution of benchmarks on an Ethereum blockchain deployed for this purpose. In order to perform this performance test, the tool uses machines from the Grid'5000 network, a flexible, large-scale testbed that can be configured at will to support large-scale experiments. Also, the machines are dedicated

[9] https://github.com/harmonica-project/sc-archi-gen

to the task for which they are allocated, not shared with other processes. Using Grid'5000 therefore allows easy reproducibility of the experiment proposed in this subsection.

For the performance test, a smart contract for Ethereum was also implemented. When deployed on the blockchain, it enables the operations defined in the blockchain scenario (saving hashed data, administration of third parties authorized to use the application). The tool is then used to set up the performance test infrastructure, represented by Figure 4.4.

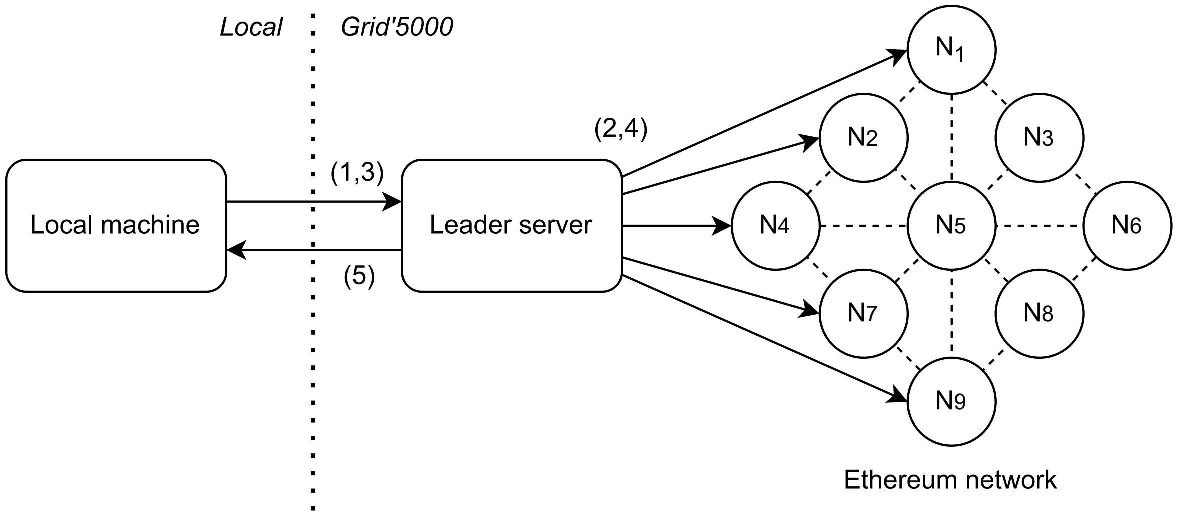


Figure 4.4: Performance test infrastructure typology.

Three different machines can be described:

- Local machine: it is used to start the benchmarking tool, that allocates the machines for the experiment and then retrieves and compiles test results.
- Main server: set up by the tool, this server starts Ethereum PoA nodes by using the parameters provided by the tool, then executes the benchmarking test.
- Ethereum node: these nodes compose the blockchain network and are listening for transactions coming from the main server.

The execution sequence of a benchmark is as follows: the tool requests the allocation of machines for the execution of benchmarks on the Grid'5000 network (1). Once the machines are obtained, it sends the configuration of the desired blockchain network to the main server. This configuration contains, among other things, the inter-block interval, the size of the blocks produced, information about the machines that act as nodes, the type of benchmark (in this case, sending transactions to a single smart contract), and the number of transactions sent per second to the blockchain. The main server initializes the machines as defined by the configuration (2), using Salt[10], a tool that allows to quickly configure several clients from a server. It then waits for the signal from the local machine to start the benchmark (3). The main server starts the benchmark (4), and sends to the blockchain network

[10]https://www.saltstack.com/

a large number of transactions every second, this number being defined in the configuration. These transactions are sent to each node in a fair way: each of them receives the same number of transactions each second, the sum of them being the volume of transactions per second expected for the benchmark. Finally, once the benchmark is completed, the local machine receives the results of the benchmark (5). It can also remain permanently connected to the main server to see in real-time the state of the nodes during the benchmark.

For this performance test, the Ethereum-PoA network is formed of 9 different nodes. Each node is equipped with an Intel Xeon Gold 5220 processor (18 cores), 96 GiB of RAM, two SSDs of 480GB and 960GB respectively, and 2x25 Gbps bandwidth. The server that drives the experiment by sending transactions has the same technical characteristics. Each of the nodes uses the Ethereum client Geth, configured with the Clique[11] PoA algorithm, a block generation interval left at the default value of 5 seconds, and an unbounded block size. The performance test is performed in multiple benchmarks, with each benchmark able to send a different volume of transactions per second to the nodes' inputs. This number is between 380 and 470, with an interval of 10 per measurement point. For each measurement point, 10 benchmarks are executed. The expected value for a benchmark is the ratio of the number of transactions that have been acknowledged to the total number of incoming transactions. It is measured after the execution of a benchmark, which lasts 215 seconds: 200 seconds with sending transactions, then 15 seconds of pause to allow the acknowledgment of the last sent transactions.

Figure 4.5 presents the results of this experiment, through a box plot, that consists of 10 boxes for each input transaction volume per second applied as input to the benchmark tool. In this figure, a box represents a series of 10 benchmarks of the same input transaction volume per second applied as input. The red bar represents the median value of this series.

These results show that the blockchain network can support a load of 380 transactions per second. Such an infrastructure is thus amply capable of supporting a load of 20 transactions per day (for each of the providers) as well as a few one-off administration transactions for the consortium. The choice of Ethereum-PoA is therefore relevant for the given use case from a performance point of view.

This figure also shows a low standard deviation for the lowest values of transactions per second applied as input, but also for the highest values. On the contrary, the series for the benchmarks performed on the values of 430 to 460 transactions per second show a high standard deviation. This can be explained by the capacity of the nodes to hold this load. Indeed, in this range, nodes reach their limit and can quickly go down, unlike the other values where nodes will/will not necessarily go down.

[11] https://github.com/ethereum/EIPs/issues/225

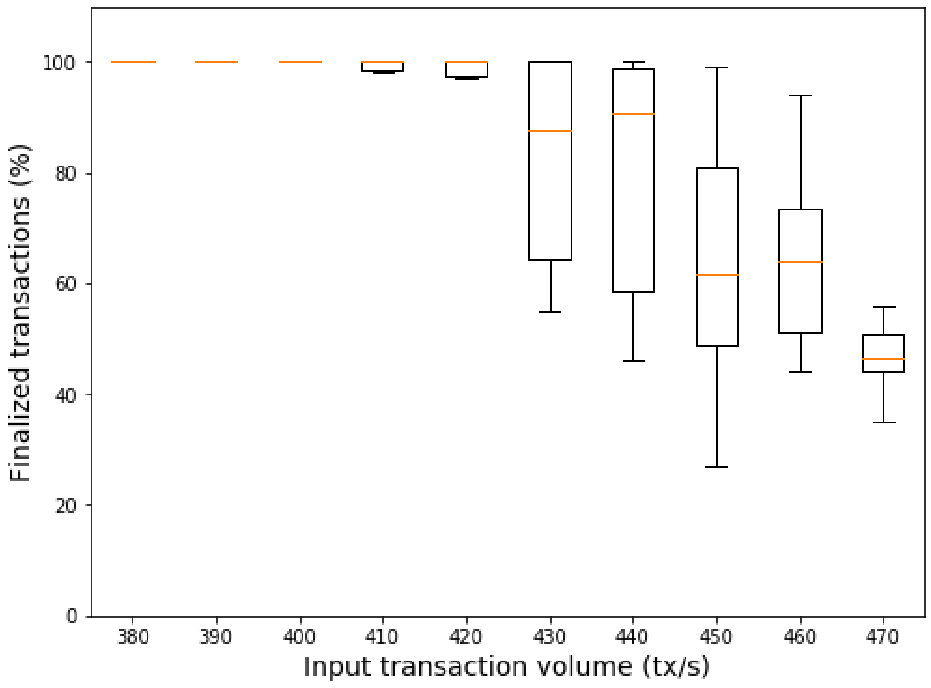


Figure 4.5: Ethereum-PoA performance tests box plot.

4.6 Discussion

The prediction obtained, that is to use Ethereum PoA, is adequate for several reasons. Indeed, all the functionalities that are considered necessary for the good implementation of the chosen case of study are present, while allowing to guarantee an optimal cost of it (low learning difficulty and energy saving). However, some limitations of the decision process have to be taken into account when using the tool.

First, the method remains sensitive to weight variations. If a higher weight for the transaction rate had been chosen, a different output result could have been obtained. Sensitivity studies can be used to establish ranges, indicating how much weights can vary without affecting the final result. There are also methods, such as entropy-based weighting, that can be used to limit the impact of criteria with high entropy by decreasing their weighting (Huang, 2008). Also, the possibility of rank reversal when using TOPSIS must be considered (García-Cascales and Lamata, 2012). This is because, although TOPSIS is suited to the tool's goals of easily adding new alternatives and making decisions from attributes of different units and scales, adding new alternatives can result in a significant change in the rank of each alternative after the decision process is run.

Second, the rating scale chosen for the expression of preferences can lead to a bias depending on the perception of the differences between the different values proposed by the user. Other weighting systems could be considered, such as AHP. Also, the attributes of the alternatives in the knowledge base being static (for example, the number of transactions per second not taking into account the resources of the machine), can lead to uncertainty in the reliability of the results. Further work is needed to propose different values for the attributes depending on the context of the decision.

For the second experiment implementing a performance test of the Ethereum-PoA blockchain, the blockchain was no longer able to process 100% of incoming transactions at 400 transactions per second or more in this experimental context. Monitoring the execution on each of the nodes shows that this inability appears when the CPU of the nodes is no longer able to support the load of transactions received by the Geth client. It is however possible to decrease the inter-block interval in order to increase performance, but a value that is too low could degrade the quality of the network (difficulty in reaching a consensus between authoritative nodes) and increase the required disk space (each block having at least one non-zero size header). Therefore, the default value was kept, but studying the impact of a decrease on stability could be beneficial.

4.7 Threats to validity

Internal threat to validity: changes in the ecosystem can be mentioned. The field of blockchain technology is constantly evolving, with numerous new technologies being introduced on an annual basis. Thus, the current state of the art frequently shifts as a result. This may affect the accuracy of recommendations given by the platform, as the recommendations may be accurate but outdated. To tackle this issue, future works may consist in finding methods for quick and accurate updates of the knowledge base. Nevertheless, the core of this contribution is the design that allows blockchain technology recommendations, given that the knowledge base is up-to-date.

External threat to validity: the platform allows the recommendation between 6 blockchain technologies, using 14 different requirements in this goal. These 6 blockchains were selected as they were, at the time of this work, among the leading blockchain either in terms of capitalization (for public blockchains) or among the most-used blockchains in companies (for private blockchains). Although this sample does not represent the full state-of-the-art of blockchain technologies, it was designed to constitute a representative panel of technologies to illustrate the blockchain recommendation engine. Yet, more technologies may be considered in future works.

Construction threats to validity: an implementation of the recommendation engine was carried out, embedded in a web platform. This implementation is subject may be subject to a divergence between the design and its concrete implementation. Yet, code reviews and tests have been carried out to ensure that the platform implementation conforms to the design.

Conclusion threats to validity: the difficulty to conclude on the satisfaction of formulated requirements can be mentioned. Easiness is notably difficult to assess for its vagueness. As a first qualitative analysis was carried out among the researchers that designed the platform, a survey may be carried out among practitioners to evaluate the usability of the platform through qualitative methods such as Unified Theory of Acceptance and Use of Technology (UTAUT) (Venkatesh et al., 2003) in future works. Also, as the core of the contribution is the decision process, the platform may be refined to improve usability following user feedback.

4.8 Related Works

This work is in line with work done to facilitate the adoption of blockchain through decision support between different types of blockchains, or the decision between using a blockchain or not in a given context.

Wust et al. list the main properties of blockchain (Transparency, integrity, trust ...) and propose a model for deciding whether or not to adopt blockchain based on the answer to certain questions (such as "Are there multiple third parties involved?" or "Are they trusted?") related to the given reference study (Wust and Gervais, 2018). They then apply their model to several example use cases. Although there is a study of the blockchain parameters to define the decision model questions, the result is of a very high level of abstraction (public blockchain, private blockchain, permissioned or no blockchain). Therefore, it does not allow to decide on one specific blockchain technology to be used. In another study from Koens et al., a literature review is performed on studies related to decision models for blockchain in order to build a new model from them (Koens and Poll, 2018). The results of this model are a bit more accurate than the previous one, but still do not give a single recommendation. Labazova et al. also present a literature review work, using a DSR (Design Science Research) approach to build a new model (Labazova, 2019). This model has multiple decision levels and takes into account blockchain properties, allowing a user to make a choice with increased output accuracy compared to previous studies.

Moreover, the study shows the dependencies between some parameters (e.g., confidentiality and transparency). However, the input parameters are mostly specific to blockchain and condition the use of the model by an expert. Another interesting study presents a third decision support approach by proposing a complete detailing of blockchain fundamentals in the first part of their study, as well as a decision model introducing opposing criteria (such as performance/costs), but also a series of questions to refine the choice, such as "When to use blockchain?", "What to use?", or "How to use this blockchain?" (Belotti et al., 2019). All of these studies help guide decision-making for a given blockchain project but do not allow for more detail (blockchain parameters) due to the limitations of the decision models. The lack of automation and manual resolution of the questions do not allow for a large number of input requirements.

Some studies have been conducted to address this issue. As an example, Tang et al. propose to use a multi-criteria decision support method called TOPSIS, that is the same as the one used in this work, to determine the best available public blockchain solution based on a set of input criteria (Tang, Shi, and Dong, 2019). The approach is interesting in this context but does not allow for other (private, permissioned) blockchains to be considered. Moreover, the blockchain technical criteria are grouped under the criteria "basic technology", "applicability" and "transaction per second", the first one being quantified via experts, the recommendations given as a result may therefore lack precision from the point of view of the company wishing to start its project.

In a study from Farshidi et al., a decision-making system for blockchain technologies is implemented, based on previous work for other technologies (Farshidi et al., 2020). A survey was conducted with experts to determine the most relevant selection criteria, then a knowledge base containing the values of these selected attributes for a large set of blockchains (obtained with white papers, studies, performance tests, etc.) is built to give recommendations via an inference engine. The proposed tool allows to give blockchain technology recommendations, but this work aims to go further by proposing a specifically blockchain oriented contribution (taking into account specific business processes and architectural models) that is more accessible for non-experts in blockchain, through a model that links blockchain attributes and software quality. This way, the user can capture more common requirements than those specific to blockchain technology.

4.9 Conclusion and Future Works

In this chapter, a tool named BLADE is presented as an answer to the problem of assisting the practitioner in the selection of a blockchain technology. BLADE provides a recommendation on the blockchain to use given a set of user requirements and preferences. For this purpose, a relevant panel of blockchains, as well as criteria related to the quality of a system (ISO 25010 standard), were selected to create a knowledge base, then a list of terms allowing a user to submit his preferences and requirements regarding the criteria chosen for the decision was chosen. The recommendation engine is then presented in detail, from the submission of entries in it to the recommendation through TOPSIS. An implementation of BLADE including this recommendation engine is presented. It allows the easy input of requirements through a web platform. Finally, the decision process is validated through a supply chain management reference study and shown that BLADE is able to recommend a blockchain aligned with the user's needs. An implementation of the tool and a link to a working demo is available on GitHub[12].

The recommendation of a blockchain technology using BLADE is the first artefact of the Harmonica framework, proposed in this thesis. This is an important step in the creation of a blockchain application: the selection of a blockchain technology has a huge impact on the final design. In Chapter 6, the second part of BLADE is introduced, to further guide the practitioner in the design of a blockchain application by proposing software patterns that are compatible with the blockchain recommended by this artifact.

[12]https://github.com/harmonica-project/BLADE

Chapter 5

Collecting Blockchain-based Software Patterns from the Literature

Key takeaways

- A literature review is carried out to identify existing blockchain-based software patterns in the literature, build a taxonomy, and classify them. This knowledge will be part of the knowledge base of the framework and serve the different DSR artifacts.
- Along with collecting these patterns, further research questions are studied to identify research gaps in the blockchain-based software pattern field, links between these patterns and traditional design patterns, and the most important patterns.
- The creation of this collection of patterns allows formulating an answer to the knowledge question RQ3.

In the previous chapter, the selection of a blockchain technology has been addressed. As the blockchain technology used is the ground of the architecture to build, a second phase in the design of a blockchain application is the selection of complementary blockchain patterns. However, identifying these patterns and ordering them to form a usable collection is not a straightforward task. Many patterns have been proposed in the academic literature, yet there is no systematic literature study to collect and classify them.

This chapter addresses the discovery and classification of patterns throughout the second research question of this thesis (**RQ3**): *How to discover then reuse software patterns in a blockchain application?* In this goal, a systematic literature review (SLR) is performed to address 6 sub-research questions that answer the main question aforementioned. First, a corpus of publications about blockchain-based software patterns is identified. Then, the knowledge about identified patterns is extracted using a generic pattern format that allows to store in a uniform way collected patterns. A taxonomy has also been built using this knowledge to classify this state-of-the-art of existing patterns into comprehensive categories.

This chapter is organized as the following. First, Section 5.1 describes the SLR in-depth, from the description of the SLR process to the obtained results. The method to build the taxonomy is also discussed in this Then, Section 5.2 discusses obtained results and addresses every sub-research question presented in the previous section. It notably introduces the identified patterns w.r.t. their respective taxonomy category. Finally, Section 5.5 concludes the chapter.

5.1 Review Process

As it is important to follow a robust methodology to perform a high-quality literature review, this work follows Kitchenham et al. guidelines to conduct a Systematic Literature Review (SLR) (Kitchenham and Charters, 2007). This task was divided into three main stages as follows:

1. Planning: during this phase, the research questions, as well as the goals of the SLR, are elicited. Also, the literature databases that will serve for the retrieval of papers are selected, and inclusion/exclusion criteria are given.
2. Conducting: the SLR is conducted, following the plan designed during the planning phase. Studies are extracted then filtered, and the remaining papers are read. An analytic framework is used to extract the necessary data to answer the research questions.
3. Reporting: results of the SLR are factually given, as well as a quality assessment of the extracted studies. Then, they are discussed in their own section.

5.1.1 Review Planning

The first step in planning the systematic literature review is the formalization of sound research questions. Those questions have to be designed considering that the answers should address the research goals of this work. The main purpose of this work is the design of a comprehensive and uniform collection of blockchain software patterns extracted from the existing literature. However, collecting the patterns in bulk is not enough to allow their reusability and usability; thus a classification scheme have to be proposed along. To further refine the quality of extracted patterns, consider the context of those patterns can also be considered: their relation with existing non-blockchain patterns, such as the ones in Gamma et al. design pattern book (Gamma et al., 1995), or their links with specific technologies or domains. Indeed, several patterns that cannot be separated from their domain or their technology have been found. As an example, the *Limit modifiers* pattern is directly bound to the modifier keyword in the Solidity language, thus non-applicable to blockchains that do not support it. These aspects should be addressed in the research questions. Finally, the results of the systematic literature review can be used to highlight several research gaps in the blockchain software pattern literature for further exploration.

From the different considerations of this work, six research questions were formulated and addressed during the SLR. The first three of them actively contribute to the construction of the blockchain-based software patterns knowledge base:

- **RQ3.1**: What taxonomy can be built from existing literature on blockchain-based patterns?
- **RQ3.2**: What are the existing blockchain-based patterns and their different categories?
- **RQ3.3**: What are the applications of the literature patterns?

Along these questions, three additional questions were stated that aim to explore other relevant aspects of blockchain-based software patterns:

- **RQ3.4**: Are some of the patterns equivalent to existing software patterns?
- **RQ3.5**: What are the most frequently mentioned patterns and their variants across the patterns identified?
- **RQ3.6**: What are the current gaps in research on blockchain-based patterns?

To avoid any confusion between the core material of the thesis and additional considerations on blockchain-based software patterns, the first set of research questions is labeled "Core research questions", and the second set is "Additional research questions". In order to extract relevant studies, three library databases have been selected: IEEE Xplore, ACM Digital Library, and Scopus. These databases have been chosen as they cover the majority of peer-reviewed research in information systems. Snowballing from selected papers is also considered as a data source, as it might help to include other relevant papers. To query the databases of papers, a search query has to be designed. The words composing the query have been chosen using the Quasi-Gold Standard (QGS) technique (Zhang, Babar, and Tell, 2011). The QGS method consists in selecting a set of studies that should appear in the results of the query, then designing the query around the terms employed in those papers. Thus, 5 studies have been selected to compose this corpus of studies (Xu et al., 2018; Bartoletti and Pompianu, 2017; Wöhrer and Zdun, 2018; Wohrer and Zdun, 2018; Liu et al., 2020). From that, the following query has been constituted:

> *(blockchain OR blockchain-based OR "smart contract*") AND ("idiom*" OR ("architectural pattern*" OR "design pattern*" OR "blockchain pattern*" OR "blockchain-based pattern*"))*

In this chapter, the query is written in a literal format with wildcards (*). During the literature review, the syntax of this query was modified for each chosen library database, to conform with the different search engines. Nevertheless, it was assessed that the same query was executed for each database, independently of their different syntaxes. The SLR only includes the studies that have those terms in their title, abstract, or keywords, to improve the precision of the query. To prepare for the filtering phase of the SLR, inclusion, and exclusion criteria have been defined. They provide systematic guidelines to include or exclude papers during the filtering phase, where papers are selected for further reading. Table 5.1 provides the chosen inclusion and exclusion criteria.

Inclusion criteria	Exclusion criteria
• Presents one or more blockchain-based software patterns. • Presents a collection of blockchain-based software patterns.	• The paper is described as presenting blockchain-based patterns from other accepted studies. • The paper lies outside the software engineering and blockchain domains. • Full text is not accessible. • The paper is a duplicate of another. • The paper is not written in English. • The paper has not been peer-reviewed.

Table 5.1: Inclusion and exclusion criteria.

Two inclusion criteria were defined. A paper is considered for the literature review if it either introduces a collection of blockchain-based software patterns or standalone patterns. Regarding the exclusion criteria, papers were discarded if they introduce patterns where their description lies in other papers, as it would lead to the creation of duplicates. Also, they were discarded if they were duplicates of other studies. In the case of a short paper being extended to a long paper, the long paper was kept and the short paper was marked as duplicate. Some papers were also discarded as they present patterns that are unrelated either to blockchain technologies, or software engineering. Also, papers were discarded if they were not accessible, not peer-reviewed, or written in a non-English language.

Finally, a set of questions, named Quality Questions (QQs), have been prepared to assess the quality of the extracted patterns:

- **QQ1**: Does the paper clearly present the pattern solutions, problems, and contexts?
- **QQ2**: Does the paper reference existing solutions using explicit patterns?
- **QQ3**: Does the paper use a standard pattern presentation form, such as GoF or Alexandrian templates (Tešanovic, 2005), described in Subsection 2.1.2?

For each question, an answer can be given among the following options: "Yes", "Partially", and "No". Knowing the answer to a paper can help to assess the quality of the patterns introduced in it, whereas knowing the answer to all the papers assesses the quality of the collection derived from this literature study. To guarantee the quality of the extracted patterns, papers were kept only where the answer to the first question is at least "Partially". Indeed, it is difficult to extract a clear pattern where there is no description of the solution and the problem it addresses in a specific context.

5.1.2 Review Execution

Figure 5.1 gives a graphical overview of the review protocol, where for each step the number of remaining or excluded papers is displayed. At first, 98 papers have been retrieved using the query over the three selected databases. As Scopus indexes papers from many other libraries, 17 duplicates were found and removed. Then, papers have been filtered on their title, abstract, and keywords based on the inclusion/exclusion criteria defined in the review planning (5.1.1). After filtering, 32 papers were kept from this initial filtering, from which 18 additional papers were filtered out during the reading phase, for several reasons. First, some of them were not fitting the inclusion/exclusion criteria, as they were not presenting any design patterns in their studies. Also, the presentation of software patterns in several papers was not clear enough for data extraction (QQ1). Lastly, some papers were excluded as they were merely presenting patterns without proposing any enhancement. During the reading phase, papers that were mentioned by others to introduce blockchain-based patterns were added to the corpus of papers.

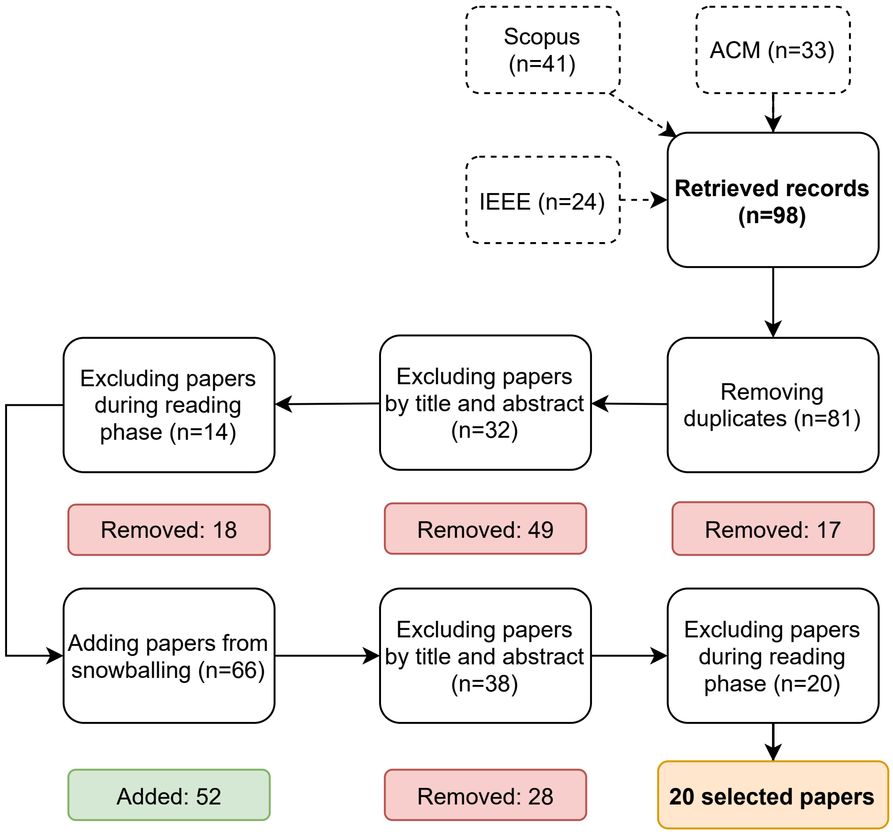

Scopus (n=41)
ACM (n=33)
IEEE (n=24)
Retrieved records (n=98)
Removing duplicates (n=81)
Removed: 17
Excluding papers by title and abstract (n=32)
Removed: 49
Excluding papers during reading phase (n=14)
Removed: 18
Adding papers from snowballing (n=66)
Added: 52
Excluding papers by title and abstract (n=38)
Removed: 28
Excluding papers during reading phase (n=20)
20 selected papers

Figure 5.1: Review process scheme.

In addition, backward and forward snowballing was done for each paper to complete the corpus of studies. Regularly performed during systematic literature reviews, backward and forward snowballing respectively aims to analyze the citations of selected papers and other papers that cited selected papers to find new relevant papers. This has led to the addition of 52 papers from snowballing, where 46 of them were filtered out. The result is the addition of 6 new studies into the final corpus, that were exclusively found during backward snowballing. Forward snowballing hasn't yielded any new study into the corpus. Note that, contrary to forward snowballing and regular inclusion of papers through performed queries, non-peer-reviewed papers were not excluded during backward snowballing. This decision has been made as they can be considered relevant, as selected papers citing them were peer-reviewed themselves.

5.1.3 Taxonomy Construction

In parallel with the review process, the taxonomy was built using newly acquired knowledge. To achieve such a task, a taxonomy development methodology was used (Nickerson, Varshney, and Muntermann, 2013). The methodology proposed by Nickerson et al. first describes what a taxonomy is and the associated problems for taxonomy development. Then, it gives a method for taxonomy development that satisfies the problems mentioned before, adaptable for many contexts.

According to Nickerson et al. (Nickerson, Varshney, and Muntermann, 2013), a possible definition of a taxonomy is the following (7):

Definition 7 *A taxonomy is a set of dimensions each consisting of mutually exclusive and collectively exhaustive characteristics such as each object under consideration has one and only one characteristic for each dimension.*

An important attribute, as stated by the definition, is that no object can have two different characteristics in a dimension. Also, a taxonomy is not meant to be perfect and can change over time, but they have to fulfill qualitative attributes to be usable:

- *Conciseness* - too many dimensions can lead to difficulties in applying the taxonomy.
- *Robustness* - containing enough clear dimensions and characteristics to differentiate objects contained inside, and comprehensive, that is the capability to classify all known objects within the domain.
- *Extensibility* - to adapt to the needs and enable the inclusion of new objects, and explanatory to provide information on the nature of the objects under study.

These qualities are particularly important for the construction of the taxonomy: the conciseness and the robustness of the taxonomy will help the reader to navigate in the different categories available to pick relevant patterns (i.e., the knowledge domain), and the extensibility will allow the taxonomy to grow over future studies on blockchain patterns.

In Nickerson et al., two methods for taxonomy construction are presented: empirical-to-conceptual, and conceptual-to-empirical. For this work, the first one was used, as existing content on patterns is empirically reused. An overview of this method is given in Figure 5.2, as presented by Nickerson et al. (Nickerson, Varshney, and Muntermann, 2013).

The first step of the taxonomy construction is to define meta-characteristics. This gives a basis for identifying the other characteristics of the taxonomy. In this taxonomy, the two meta-characteristic "On-chain pattern" and "On/off-chain interaction pattern" have been

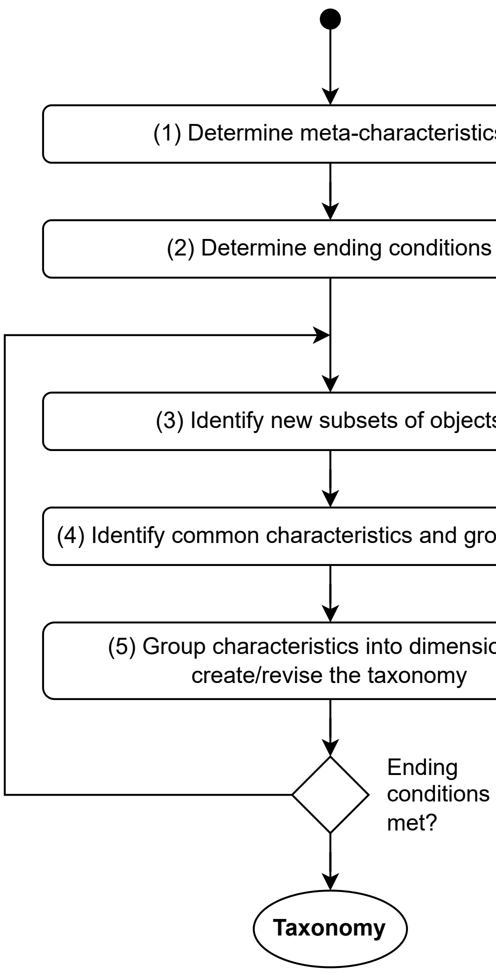

(1) Determine meta-characteristics
(2) Determine ending conditions
(3) Identify new subsets of objects
(4) Identify common characteristics and group objects
(5) Group characteristics into dimensions to create/revise the taxonomy
Ending conditions met?
Taxonomy

Figure 5.2: Empirical-to-conceptual taxonomy development method.

chosen. As this work focuses on design aspects, I found it relevant to order patterns depending on their position regarding the blockchain: in the blockchain (smart contracts, transaction data), or out of the blockchain (services that interact with the blockchain, wallets, ...). Then, as building a taxonomy is an iterative process, ending conditions have to be determined. Indeed, as indicated earlier, a taxonomy is never perfect; thus the process stops when the taxonomy is "good enough" (i.e. when all the qualities of a well-built taxonomy are present).

Additional ending conditions can also be added. For instance, I chose to examine all objects of a representative sample of objects. As the patterns are the cornerstone of this work, it is important to examine all of them to construct an accurate taxonomy. Therefore, this taxonomy construction is empirical-to-conceptual rather than the opposite: from the patterns, categories are drafted and then refined to return an accurate taxonomy. If categories are significantly unbalanced (e.g. 50 patterns in one category, 2 in another one), or one pattern cannot be classified into a single category, another iteration on the taxonomy is performed to create new subcategories or rename existing categories.

The next three steps are the construction of the taxonomy itself. As they are incremental, they have to be repeated until ending conditions are met. To begin, the identification of a subset of objects should be done. In this case, the subset is constituted of all the identified patterns. The next step is identifying common characteristics and group objects. To do that, a Natural Language-based algorithm was designed to ingest all the pattern descriptions, lemmatize them, and identify a recurrent suite of words (n-grams). For bigrams, the most recurrent combination of words was "Smart contract(s)" (54 times), "Data storage" (11 times), "Proxy contract" (6 times), and "Factory object" (6 times). Other interesting combinations were found: "Outside blockchain" (5 times), "Restrict execution" (3 times), and "Critical operation" (3 times). From those combinations and others, three assumptions can be made: (1) smart contract is a crucial topic in blockchain-based patterns, (2) many traditional software design patterns were found in pattern summaries. Thus links might exist between the existing knowledge of software patterns and newly designed patterns, (3) some important design aspects are recurrent in pattern summaries. Existing collection names were also exploited to generate categories. For example, (Marchesi et al., 2020) proposes patterns exclusively dedicated to smart contract gas efficiency. Such a collection gives hints of potential types of categories.

Using these assumptions and the researcher's personal knowledge, a first taxonomy has been built empirically. Several categories were created from the aforementioned subset of concepts, such as "smart contract patterns". Then, each pattern collected during the literature review was assigned a category by a researcher, based on the pattern title, context, and problem. In the end, the number of patterns in each category was analyzed, as well as the list of patterns where it was challenging to assign them to a single category. This may highlight the lack of balance between categories, as too many patterns may be present in a single category, or categories that overlap themselves. As this was the case for the first taxonomy, two other iterations were performed to construct the version of the taxonomy introduced in this chapter. During the literature review, it was also found that the majority of the patterns identified are design patterns, thus the taxonomy has been recentered from all software patterns to design patterns. The final version of this taxonomy is presented in the Subsection 5.2.1 associated with the RQ1.

5.1.4 Results

This section factually presents the results of the systematic literature study. More details are given when discussing each research question in Section 5.2.

The final corpus of papers is composed of 20 studies, out of which 6 were added through reference snowballing. Within the corpus, 19 papers propose design patterns, whereas only one proposes architectural patterns. No study that introduced idioms was found. However, some of the patterns found were more related to idioms than design patterns and categorized as such. From these 20 studies, 160 patterns were found, including duplicates. At first, patterns that were said to come from other studies were also added but filtered out afterward to ensure no pattern is missing from the extracting phase. After duplicate removal, 114 unique patterns have been found: 104 of them have been classified as design patterns, 3 of them as architectural patterns, and 14 as idioms. As the links between patterns across papers were collected during the SLR, they have been used to filter a large number of duplicates. Then, pattern names and summaries/solutions were used to filter out additional patterns. Precautions have been taken when removing patterns using those fields: close patterns that diverge on tiny aspects were kept as separate patterns.

Regarding the quality assessment performed on accepted papers, Figure 5.3 shows the distribution of the answers to each question.

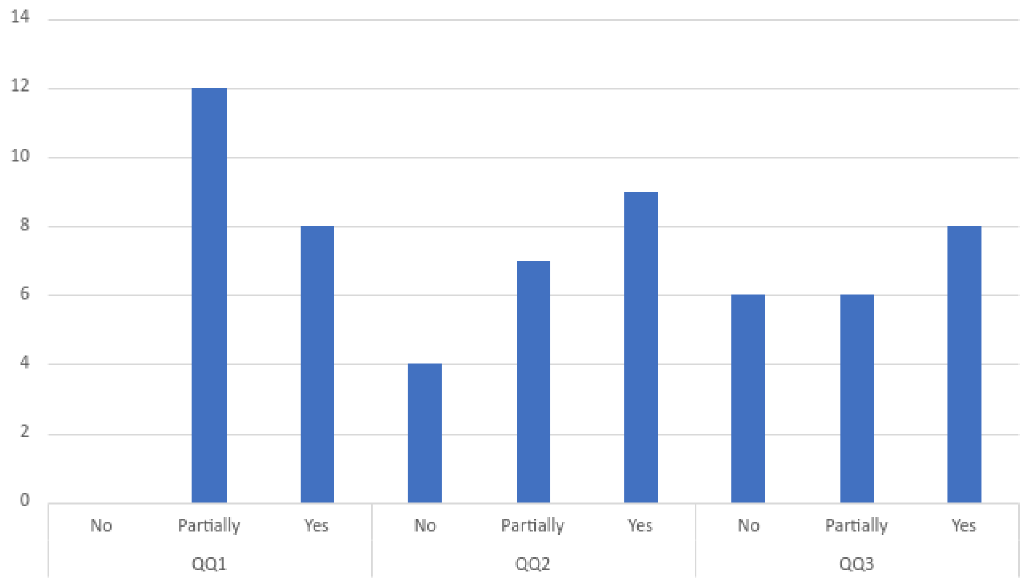


Figure 5.3: Quality assessment answers distribution (labels detailed in Subsection 5.1.1)

For the first quality question QQ1, 8 papers out of 20 introduce patterns that are easily understandable and detailed, whereas 12 papers might lack details in the pattern detailing. The second quality question QQ2 shows that 4 papers do not mention any example of implementation, 7 references one example on average per pattern, and 9 studies reference more than 2 implementation examples. Finally, the third quality question QQ3 indicates that 8 papers are using a pattern format to describe their patterns, 6 papers are using a form but lack important sections usually found in pattern formats, and 6 studies do not use any format.

5.2 Discussion

In this section, each research question is addressed using the results collected throughout the completion of the systematic literature review. For each question, a set of data tailored to answer the research question has been collected. The synthesis of these results allows to formulate a detailed answer to each research question and discuss them. As mentioned in Subsection 5.1.1, these research questions are divided into two sets, that are the core content of the thesis and additional considerations.

5.2.1 Core Research Questions

RQ3.1: What taxonomy can be built from existing literature on blockchain-based patterns?

A taxonomy of blockchain-based patterns is presented to classify the design patterns in comprehensive categories that help to decide on what patterns to use for a specific aspect of blockchain-based application development. This taxonomy has been built using Nickerson's methodology (Nickerson, Varshney, and Muntermann, 2013), and its construction is detailed in Subsection 5.1.3. The patterns collected during the systematic literature review were reused as a knowledge source to build the categories of the taxonomy. They were regrouped into categories based on their commonalities: for instance, the different types of oracles identified (2018; 2018; 2020; 2018; 2020) form the *Oracle patterns* subcategory.

Figure 5.4 shows a graphical representation of the proposed design pattern taxonomy.

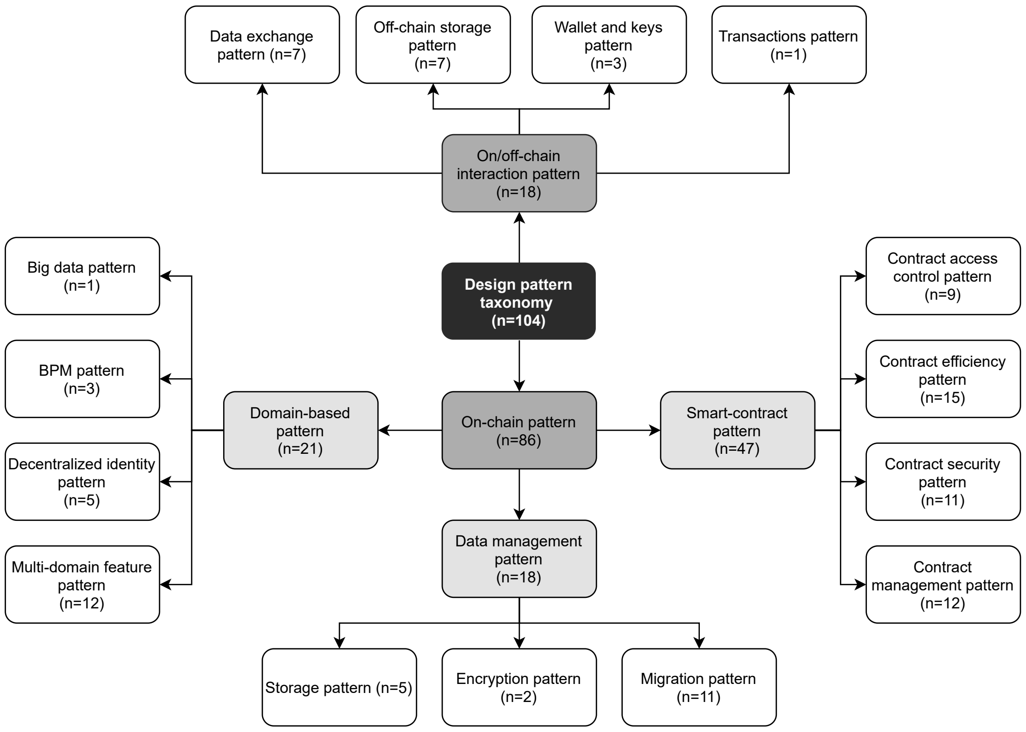

Data exchange pattern (n=7)
Off-chain storage pattern (n=7)
Wallet and keys pattern (n=3)
Transactions pattern (n=1)
On/off-chain interaction pattern (n=18)
Big data pattern (n=1)
Design pattern taxonomy (n=104)
Contract access control pattern (n=9)
BPM pattern (n=3)
Contract efficiency pattern (n=15)
Domain-based pattern (n=21)
On-chain pattern (n=86)
Smart-contract pattern (n=47)
Decentralized identity pattern (n=5)
Contract security pattern (n=11)
Data management pattern (n=18)
Multi-domain feature pattern (n=12)
Contract management pattern (n=12)
Storage pattern (n=5)
Encryption pattern (n=2)
Migration pattern (n=11)

Figure 5.4: Design pattern taxonomy.

The taxonomy consists of 2 main categories (i.e. meta-characteristics), "On-chain pattern" and "On/off-chain interaction pattern", and 15 different categories. Intermediate categories were also created to group categories together in the "On-chain pattern" meta-category: "Smart contract pattern", "Data management pattern", and "Domain-based pattern".

The "On/off-chain interaction pattern" category aims to regroup design patterns constituted of off-chain elements that interact with a blockchain. This is key for the development of decentralized applications, as proposed design patterns might help bridge off-chain systems and software with on-chain data or smart contracts.

This category is composed of four subcategories. The first one, the "Data exchange pattern" subcategory, groups patterns that enable communication between on-chain smart contracts and off-chain components. Indeed, blockchain cannot request data from outside, thus requiring an external service (i.e. Oracle) to push fresh data inside smart contracts.

The "Data management pattern" subcategory is comprised of design patterns that leverage off-chain data but use blockchain to guarantee tamper-proofing or trustability of those data. For instance, hashing a dataset, then storing the hash on-chain to attest later the integrity of the dataset.

The "Wallet and keys pattern" subcategory tackles the management of wallets and keys in the context of a decentralized application. Finally, the "Transactions pattern" subcategory deals with the transaction aspects between off-chain components and the blockchain, such as transaction confirmation or block inclusion.

In the "Domain-based pattern" intermediate category, on-chain patterns that deal with domain features are regrouped. Note that this category is meant to be extended with the advances in blockchain-based patterns for specific domains. Therefore, three domain-specific categories were created from the knowledge of existing domain-based patterns: the "Business Process Management (BPM) pattern" subcategory concerns on-chain business process management (e.g., on-chain activities, ...), the "Big data pattern" subcategory proposes applications of blockchain for big data, and the "Decentralized identity pattern" subcategory leverage blockchain to create and manage decentralized identities. A fourth subcategory, "Multi-domain feature pattern", contains features that do not belong to a single domain but rather can be used by multiple domains.

The "Smart contract pattern" intermediate category classifies patterns that concern smart contract implementation and management. As ensuring the security of smart contracts is primordial, the "Contract security pattern" subcategory regroups smart contract patterns that deal with security issues such as reentrancy attacks, overflow attacks, or flawed behavior of smart contracts.

The "Contract efficiency pattern" subcategory essentially deals with patterns that reduce the price of leveraging smart contracts, especially on public blockchains. It also contains patterns on other efficiency aspects such as data refreshing, a difficult task with smart contracts as they cannot perform requests on other smart contracts by themselves.

The "Contract access control pattern" subcategory regroups patterns for permission and authorization management for the execution of smart contract functions. Finally, the "Contract management pattern" subcategory helps with designing the organization of smart contracts together. For example, having a proxy smart contract that relays the function calls to other contracts.

The last intermediate category, "Data management pattern" deals with patterns for efficient on-chain data management. It is different from the "Data management pattern" subcategory of the "On/off-chain interaction pattern" subcategory as it only concerns data on-chain, located in smart contracts or directly in transactions. The "Migration pattern" subcategory groups patterns that help with migrating data from one blockchain to another. Under "Encryption pattern" are classified patterns for on-chain data encryption, and "Storage pattern" regroups patterns that deal with on-chain data storage.

Through the systematic literature review, the taxonomy has been applied to classify patterns with success, as the researcher was able to classify every pattern in a single category. However, it is meant to be extensible; thus categories might be changed depending on the evolution of the state of the art in blockchain-based patterns notably with the appearance of new architectural patterns or idioms, not present in this taxonomy due to their scarcity.

This taxonomy is important for the adequate usage of patterns identified in the systematic literature review. For example, a user willing to implement smart contract security measures in his application to protect it against threats or vulnerabilities will be tempted to search in the "Contract security pattern" subcategory instead of directly searching in the corpus of patterns. They are also complementary: as each category covers a specific aspect of the design of a blockchain-based application, they can be combined depending on the user requirements. For instance, "Contract security" patterns can be used along the "Contract efficiency" patterns to improve at the same time the cost efficiency and the security of designed smart contracts. However, possible conflicts between individual patterns are left outside the scope of this chapter, as this information is not present in retrieved papers.

RQ3.2: What are the existing blockchain-based patterns and their different categories?

The systematic literature review yielded 160 descriptions of blockchain-based software patterns, from which 116 unique patterns were identified. These patterns have then been classified using the taxonomy into 15 different categories. This subsection will introduce these different categories along with a short description of their respective patterns. The focus will notably be made on patterns observed in multiple studies. The complete description of each pattern, its category, and its links with others are available on GitHub[1].

On/off-chain Interaction Patterns

This first category regroups all of the patterns with their components both on and off-chain. It is divided into four subcategories. Table 5.2 lists all the patterns contained in this category.

[1]https://github.com/harmonica-project/blockchain-patterns-collection

On/off-chain interaction patterns	
Subcategory	**Patterns**
Data exchange pattern	• Ticker tape (Worley and Skjellum, 2018) • Oracle (Rajasekar et al., 2020; Xu et al., 2018; Wöhrer and Zdun, 2018; Bartoletti and Pompianu, 2017) • Reverse Oracle (Rajasekar et al., 2020; Xu et al., 2018; Worley and Skjellum, 2018) • Pull-based inbound oracle (Mühlberger et al., 2020) • Push-based inbound oracle (Mühlberger et al., 2020) • Pull-based outbound oracle (Mühlberger et al., 2020) • Push-based outbound oracle (Mühlberger et al., 2020)
Data management pattern	• State Channel (Rajasekar et al., 2020; Xu et al., 2018) • (Off-chain) Contract Registry (Rajasekar et al., 2020) • Legal and smart contract pair (Xu et al., 2018) • Off-chain data storage (Müller, Ostern, and Rosemann, 2020; Rajasekar et al., 2020; Xu et al., 2018; Liu et al., 2020; Lemieux, 2017; Eberhardt and Tai, 2017) • Confidential and pseudo-anonymous contract enforcement (Six et al., 2020) • Off-chain Signatures (Eberhardt and Tai, 2017) • Delegated Computation (Eberhardt and Tai, 2017)
Wallet and keys pattern	• Master & Sub Key (Liu et al., 2020) • Hot & Cold Wallet Storage (Liu et al., 2020) • Key Sharding (Liu et al., 2020)
Transactions patterns	• X-confirmation (Xu et al., 2018)

Table 5.2: On/off-chain interaction patterns.

The first subcategory is named "Data exchange pattern", to group patterns that enable communication between on-chain smart contracts and off-chain components. This subcategory contains 7 patterns. The most frequent pattern is the *Oracle* pattern, introduced or mentioned by 5 different papers (Rajasekar et al., 2020; Xu et al., 2018; Wöhrer and Zdun, 2018; Bartoletti and Pompianu, 2017; Worley and Skjellum, 2018). As blockchain cannot request the external world to retrieve up-to-date information, components named oracles have been designed to listen for blockchain requests or statuses that indicate some information is needed, then send a transaction to the blockchain to inject them.

Its opposite has also been proposed: the *Reverse oracle* pattern is applied when off-chain components need blockchain data to work, so they listen for specific state changes and react accordingly (Worley and Skjellum, 2018; Xu et al., 2018; Rajasekar et al., 2020; Marchesi et al., 2020). Another study proposed more detailed variants of those patterns, as they differentiate the data flow direction (as the *Oracle* and *Reverse Oracle*), as well as if data are pushed out of the data source or pulled from an active component.

The second subcategory groups 7 patterns that manage and store data off-chain while using blockchain as an additional layer of trust. A commonly proposed pattern under many names is the *Off-chain data storage* pattern (Müller, Ostern, and Rosemann, 2020; Rajasekar et al., 2020; Xu et al., 2018; Liu et al., 2020; Lemieux, 2017; Eberhardt and Tai, 2017). It consists of storing large amounts of data off-chain, then producing a hash of the data and saving it on-chain. Therefore, it is far cheaper to leverage while having the possibility to check the integrity of stored data using the hash on-chain. This pattern is presented in detail in a dedicated part of Subsection 5.2.2.

The same concept has been applied to variants. For example, the *State channel* pattern involves letting two or more users perform micro-transactions off-chain and regularly storing a hash on-chain to prove the existence of such transactions later on. Other studies propose the binding between an off-chain legal contract and an on-chain smart contract, to ensure sensitive data are kept off-chain while only important signatures and states are stored on-chain (Six et al., 2020; Xu et al., 2018).

Finally, the third and fourth subcategories are respectively "Wallet and keys pattern" and "Transaction pattern". They only contain three and one patterns respectively: *Key sharding*, *Hot & Cold wallet storage*, and *Master & Sub keys* patterns (Xu et al., 2018; Liu et al., 2020) for healthy management of blockchain wallets and keys, as well as the *X-confirmation* pattern (Xu et al., 2018). The latter consists in waiting for a predefined number of blocks to ensure that the transaction added is probabilistically immutable. Although there are only a few patterns in those categories, they have been added as they might contain more patterns later in future studies.

On-chain patterns - domain-based Patterns

The "Domain-based pattern" intermediate category is part of the "On-chain pattern", and contains patterns that propose a feature to address a domain-based problem, either for a specific domain or applicable to many. A list of all the patterns contained in this category is presented in Table 5.3.

On-chain patterns - domain-based patterns	
Subcategory	**Patterns**
BPM pattern	• Blockchain BP Engine (Müller, Ostern, and Rosemann, 2020) • Smart Contract Activities (Müller, Ostern, and Rosemann, 2020) • Decentralize business process (Müller, Ostern, and Rosemann, 2020)
Decentralized identity pattern	• Identifier Registry (Liu et al., 2020) • Multiple Registration (Liu et al., 2020) • Bound with Social Media (Liu et al., 2020) • Dual Resolution (Liu et al., 2020) • Delegate List (Liu et al., 2020)
Big data pattern	• Blockchain Security Pattern for Big Data Ecosystems (Moreno et al., 2019)
Multi-domain feature pattern	• Blockchain-based reputation system (Müller, Ostern, and Rosemann, 2020) • Blocklist (Worley and Skjellum, 2018) • Vote (Worley and Skjellum, 2018) • Announcement (Worley and Skjellum, 2018) • Bulletin Board (Worley and Skjellum, 2018) • Randomness (Bartoletti and Pompianu, 2017) • Poll (Bartoletti and Pompianu, 2017) • Selective Content Generation (Liu et al., 2020) • Time-Constrained Access (Liu et al., 2020) • One-Off Access (Liu et al., 2020) • Digital Record (Lemieux, 2017) • State machine (Wöhrer and Zdun, 2018)

Table 5.3: On-chain patterns - domain-based patterns.

For BPM, 3 patterns have been identified, all proposed by Müller et al. (Müller, Ostern, and Rosemann, 2020): the *Blockchain BP Engine* pattern, that enables collaborative business processes by storing and executing a business process through a smart contract, the *Smart contract activities* pattern where business logic activities are stored in a single smart contract for execution, and the *Decentralize business process* pattern that uses blockchain as a software connector for collaborative business process execution.

Regarding decentralized identity patterns, 5 design patterns have been extracted (Liu et al., 2020). The first one, *Identifier registry* pattern, proposes the usage of smart contracts to establish a mapping between a DID (Decentralized Identifier), a unique identifier for a human within a domain, and the location of off-chain storage attributes. Here, the DID is managed using a private key used to prove the ownership of an identifier. If the key is lost, the *Delegates list* pattern can be used to retrieve this ownership. To protect user privacy, multiple identifiers can be created using the *Multiple identifiers* pattern. An identifier can also be mapped to a social media account through the *Blockchain & Social Media Account Pair* pattern, to improve the trustworthiness of both social media accounts and identifiers. Finally, the *Dual resolution* pattern helps to use a DID to enable communication with another entity through its own DID.

One pattern has been identified for the "Big data pattern" category: the *Blockchain Security Pattern for Big Data Ecosystems* pattern leverages blockchain to register operations performed on a data store (Moreno et al., 2019).

The "Multi-domain feature pattern" subcategory groups 12 patterns that propose on-chain features to address problems found in multiple domains. For example, the *Poll* and the *Vote* patterns (Bartoletti and Pompianu, 2017; Worley and Skjellum, 2018) can be used to take collaborative decisions on-chain, the *Time-constrained access* or the *One-Off Access* patterns (Liu et al., 2020) let users give access to off-chain resources from an on-chain authorization smart contract, and the *Randomness* pattern (Bartoletti and Pompianu, 2017) can be used to generate random numbers on-chain, a difficult task.

On-chain patterns - smart contract patterns

The second intermediate category of "On-chain patterns" is the "Smart contract pattern". In a decentralized application, smart contracts are often the most important pieces. Many sensitive operations can be performed on them, such as storing and transferring cryptocurrencies. Therefore, maximal security in smart contract operations is paramount, and well-designed access control functions should be implemented to support it. Managing them is also difficult, as a smart contract code is immutable once deployed. Thus, the on-chain smart contract architecture has to be adequately designed to tackle the inflexibility of smart contracts and ensure they fill their initial goals while being easily upgradeable if needed. Finally, they often have to be efficient, as for public blockchains developers and users have to pay for deploying and executing smart contract functions. Each of those topics is important for the development of smart contracts and has its own subcategory, presented below.

Table 5.4 and 5.5 respectively introduce design patterns related to the management and the security of smart contracts, and design patterns related to the efficiency and access control of smart contracts.

On-chain patterns - smart contracts patterns	
Subcategory	**Patterns**
Contract management pattern	• Migration (Worley and Skjellum, 2018) • Inter-family communication (Owens et al., 2019) • Data Contract (Rajasekar et al., 2020; Xu et al., 2018; Marchesi et al., 2020; Wöhrer and Zdun, 2018) • Factory Contract (Rajasekar et al., 2020; Xu et al., 2018; Zhang et al., 2018; Zhang et al., 2017; Liu et al., 2018) • Proxy Contract (Rajasekar et al., 2020; Wöhrer and Zdun, 2018; Zhang et al., 2017; Liu et al., 2018; Marchesi et al., 2020; Zhang et al., 2018) • Flyweight (Rajasekar et al., 2020; Zhang et al., 2018; Zhang et al., 2017) • Satellite (Wöhrer and Zdun, 2018) • Contract Registry (Xu et al., 2018; Wöhrer and Zdun, 2018) • Contract Composer (Liu et al., 2018) • Contract Decorator (Liu et al., 2018) • Contract Mediator (Liu et al., 2018) • Contract Observer (Liu et al., 2018)
Contract security pattern	• Fork check (Bartoletti and Pompianu, 2017) • Emergency Stop (Rajasekar et al., 2020; Wohrer and Zdun, 2018) • Mutex (Rajasekar et al., 2020; Wohrer and Zdun, 2018) • Contract Balance Limit (Rajasekar et al., 2020; Wohrer and Zdun, 2018) • Automatic Deprecation (Wöhrer and Zdun, 2018) • Speed Bump (Wohrer and Zdun, 2018) • Rate Limit (Wohrer and Zdun, 2018) • Check Effect Interaction (Rajasekar et al., 2020; Wohrer and Zdun, 2018) • Time Constraint (Bartoletti and Pompianu, 2017) • Termination (Bartoletti and Pompianu, 2017) • Math (Bartoletti and Pompianu, 2017)

Table 5.4: On-chain patterns - smart contracts patterns (management and security).

The first subcategory "Contract management pattern" is about properly organizing and managing the lifecycle of smart contracts in the decentralized application architecture. This subcategory contains 12 different patterns. Some of them address the separation of concerns, between the dApp entry point, features, and data. The most frequently mentioned pattern is the *Proxy* pattern (Rajasekar et al., 2020; Marchesi et al., 2020; Wöhrer and Zdun, 2018; Zhang et al., 2017; Liu et al., 2018; Zhang et al., 2018). Usually implemented in traditional software engineering to wrap an object only accessible by it, this pattern is used in blockchain to wrap a smart contract (the object) into another one (the proxy). A full description of this pattern is given in a dedicated part of Subsection 5.2.2. Another pattern for separation of concerns is the *Data contract* that decouples data from functions in two separate contracts (Rajasekar et al., 2020; Xu et al., 2018; Wöhrer and Zdun, 2018). The *Flyweight* pattern is similar in functioning but consists in storing data used by multiple contracts in one place (Rajasekar et al., 2020; Zhang et al., 2017; Zhang et al., 2018). Finally, a mentionable pattern is the *Satellite* that can be used to decouple features that are more likely to change from features that will not change over time (Wöhrer and Zdun, 2018).

The second subcategory, "Contract security pattern", is filled with 11 patterns. Most of them target Solidity-based contracts. Solidity is a programming language for smart contracts deployed on Ethereum blockchain networks. One usage of such patterns is the restriction of access to smart contracts functions when it is needed. To cite a few of them, the *Termination* pattern consists in locking the contract to prevent any further function call (Bartoletti and Pompianu, 2017). It is also possible to use the *Emergency Stop* pattern to simply halt its functioning until reactivated. This can be used for instance to protect the contract against the abusive withdrawal of funds (Rajasekar et al., 2020; Wohrer and Zdun, 2018). The *Speed bump* (Wohrer and Zdun, 2018), *Rate Limit* (Wohrer and Zdun, 2018), and *Time constraint* (Bartoletti and Pompianu, 2017) patterns are used to implement time limitations when executing smart contract functions. Some other patterns aim to protect the correct execution of a function. The *Check-Effects-Interaction* pattern (Wohrer and Zdun, 2018) guarantees safe execution of the function by first, checking the satisfaction of preconditions, then applying the modifications on the contract, and finally applying modifications on other external contracts, if needed. Also, some other interesting patterns are the *Mutex* pattern (Rajasekar et al., 2020; Wohrer and Zdun, 2018) that protects the access to a used resource, or the *Contract Balance Limit* pattern (Rajasekar et al., 2020; Wohrer and Zdun, 2018) to ensure that the smart contract does not hold too many funds, to mitigate the risk of losing all the funds if compromised.

On-chain patterns - smart contracts patterns	
Subcategory	**Patterns**
Contract efficiency pattern	• Incentive Execution (Rajasekar et al., 2020; Xu et al., 2018) • Tight Variable Packing (Rajasekar et al., 2020) • Limit storage (Marchesi et al., 2020) • Minimize on-chain data (Marchesi et al., 2020) • Limit external calls (Marchesi et al., 2020) • Fewer functions (Marchesi et al., 2020) • Use libraries (Marchesi et al., 2020) • Short constant strings (Marchesi et al., 2020) • Limit modifiers (Marchesi et al., 2020) • Avoid redundant operations (Marchesi et al., 2020) • Write values (Marchesi et al., 2020) • Pull payment (Wöhrer and Zdun, 2018) • Publisher-Subscriber (Zhang et al., 2017; Zhang et al., 2018) • Challenge Response (Eberhardt and Tai, 2017) • Low Contract Footprint (Eberhardt and Tai, 2017)
Contract access-control pattern	• Judge (Worley and Skjellum, 2018) • Embedded Permission (Worley and Skjellum, 2018; Rajasekar et al., 2020; Xu et al., 2018; Bartoletti and Pompianu, 2017) • Dynamic Binding (Rajasekar et al., 2020) • Multiple authorization (Xu et al., 2018; Liu et al., 2018) • Off-chain secret enabled dynamic authentication (Xu et al., 2018) • Access Restriction (Wöhrer and Zdun, 2018) • Ownership (Wöhrer and Zdun, 2018) • Hash Secret (Liu et al., 2018)

Table 5.5: On-chain patterns - smart contracts patterns (efficiency and access-control).

The third subcategory, "Contract efficiency pattern", contains 15 patterns. It mainly targets Ethereum smart contracts. Indeed, a user has to pay a defined amount of Ether, the native cryptocurrency of Ethereum, to deploy and interact with a contract on a public Ethereum network. The more the function stores data or perform complex operations, the more it will cost the user. Design patterns in this section help to reduce the fees associated with the deployment, storage, or execution of smart contract functions. For on-chain storage reduction, many patterns have been proposed (Marchesi et al., 2020): the *Limit storage* or *Minimize storage data* patterns in general, or *Fewer functions* and *Limit modifiers* to reduce function overhead and code size. The *Short constant string* pattern can also be used to limit on-chain storage by limiting the size of strings to prevent a high consumption of storage size. *Tight variable packing* pattern, as proposed by Rajasekar et al. (Rajasekar et al., 2020), can also be a solution to reduce storage size by storing data in the smallest unit possible (e.g., Uint8 instead of default Uint256 to store a number below 256). At computation, the *Avoid redundant operations* and the *Low contract footprint* patterns can help reduce the complexity of operations, thus saving costs (Rajasekar et al., 2020; Eberhardt and Tai, 2017). This taxonomy also places in the *Contract efficiency pattern* subcategory patterns that help to keep on-chain data accurate. For example, the *Incentive execution* pattern (Rajasekar et al., 2020; Xu et al., 2018) refunds or rewards users that call a specific function to update contract data, as no update can be done by the contract itself without external intervention.

The last subcategory is the "Contract access control pattern" and concerns the permission management of contracts. This category is constituted of 9 patterns. The most important one is the *Embedded permission* pattern (also called *Access Control* or *Authorization*), mentioned by 4 papers (Mavridou and Laszka, 2018; Bartoletti and Pompianu, 2017; Rajasekar et al., 2020; Xu et al., 2018), that consists of encoding permission in a smart contract for sensitive functions. Only authorized addresses will be able to call those functions. One variant is the *Owner* pattern (Wöhrer and Zdun, 2018), that defines a contract owner as the solely entitled person to execute specific functions. Authorization to execute a function can also require multiple signatures at the same time. A pattern named *Multiple Authorization* (Xu et al., 2018; Liu et al., 2020; Liu et al., 2018) consists in defining a set of addresses in the contract, where a fraction of them is required to execute a function. Another noteworthy pattern is the *Judge* pattern (Worley and Skjellum, 2018), that lets users vote to elect a trusted third party. The winner is given the authorization to update the smart contract with fresh information, as an *Oracle* could do.

On-chain patterns - Data management pattern

The last intermediate category of "On-chain patterns", "Data management pattern", proposes patterns related to the storage, migration, and encryption of on-chain data. The complete list of patterns contained in this category is given in Table 5.6.

On-chain patterns - data management patterns	
Subcategory	**Patterns**
Storage pattern	• Transparent Event Log (Müller, Ostern, and Rosemann, 2020) • Key-value store (Worley and Skjellum, 2018) • Address mapping (Worley and Skjellum, 2018) • Event log (Marchesi et al., 2020) • Tokenisation (Xu et al., 2018; Bartoletti and Pompianu, 2017; Lemieux, 2017; Worley and Skjellum, 2018)
Migration pattern	• Token burning (Bandara, Xu, and Weber, 2020) • Snapshotting (Bandara, Xu, and Weber, 2020) • State Aggregation (Bandara, Xu, and Weber, 2020) • Node Sync (Bandara, Xu, and Weber, 2020) • Establish Genesis (Bandara, Xu, and Weber, 2020) • Hard Fork (Bandara, Xu, and Weber, 2020) • State Initialization (Bandara, Xu, and Weber, 2020) • Exchange Transfer (Bandara, Xu, and Weber, 2020) • Transaction Replay (Bandara, Xu, and Weber, 2020) • Virtual Machine Emulation (Bandara, Xu, and Weber, 2020) • Smart Contract Translation (Bandara, Xu, and Weber, 2020)
Encryption pattern	• Commit and Reveal (Rajasekar et al., 2020; Wöhrer and Zdun, 2018) • On-chain encryption (Xu et al., 2018)

Table 5.6: On-chain patterns - data management patterns.

Regarding the "Storage pattern" subcategory, 5 have been identified. The most-proposed one is the *Tokenization* pattern (Worley and Skjellum, 2018; Bartoletti and Pompianu, 2017; Xu et al., 2018; Lemieux, 2017). Through this design pattern, real-life or complex assets can be encapsulated into a token and exchanged on-chain. A dedicated part of Subsection 5.2.2 gives a detailed presentation of this pattern. Other forms of data storage can be mentioned: the *Key-value store* pattern to organize data into a resizable store, accessible with keys, or the *Address mapping* pattern where mapping is established between an address and its associated data (Worley and Skjellum, 2018). Finally, some patterns propose to store logs of data into event logs, either in a native blockchain event log (proposed by some blockchains, such as Ethereum) (Marchesi et al., 2020) or in a smart contract (Müller, Ostern, and Rosemann, 2020).

Two patterns have been added to the "Encryption pattern" subcategory. Despite the lack of patterns for this subcategory, it still has been added as many patterns will probably be added to this subcategory in the future, following the advances in on-chain encryption strategies such as homomorphic encryption (Liang et al., 2020) or zero-knowledge proofs (Yang and

Li, 2020). The *On-chain encryption* pattern (Xu et al., 2018) helps in protecting sensitive on-chain data through symmetric encryption. Data can then be stored on-chain and be non-readable by anybody who does not have the encryption key. The main drawback of this pattern is the key leakage threat because data will remain on-chain forever, even in case of a leak. The *Commit and Reveal* pattern works differently: some values are kept secret during the commit phase and revealed when needed (Rajasekar et al., 2020; Wöhrer and Zdun, 2018). It is possible to attest that the revealed value was the same as the one committed in secret. Through this pattern, it is possible to commit some data without revealing its content.

In the last subcategory, "Migration pattern", 11 design patterns for data migration are included. All of those patterns were found in a paper that proposes a pattern collection for data migration (Bandara, Xu, and Weber, 2020). To mention a few of them, the *Snapshotting* pattern consists in saving a copy of states, smart contracts, and transactions on the source blockchain to transfer them to the target blockchain later. This operation can be done using the *State initialization* or the *Establish genesis* patterns to respectively transfer states from source to target blockchain or set states in the first block of target blockchain (i.e., genesis block). Besides existing data, the code of useful smart contracts may also be changed to fit the target blockchain; this can be done using the *Smart contract translation* pattern.

Architectural Patterns and Idioms

To conclude this question, other patterns that do not belong to the design pattern taxonomy are introduced. This sample of patterns contains 14 idioms (Rajasekar et al., 2020; Marchesi et al., 2020). They all concern Solidity, a smart contract programming language for the Ethereum blockchain, and address smart contract efficiency. As presented before, users have to pay for smart contract function execution on a public blockchain. Proposed idioms help to reduce execution fees in various ways: for example, *Packing variables* or *Packing booleans* patterns can be used to reduce variable required storage with a smart ordering of variables in the code, as background variables are grouped by the compiler in 32-bytes slots. More efficient structures can be selected to save space, thus costs, using *Uint* vs Uint256* and *Mapping vs Array* patterns. Ether can also be returned to the user when using the *Freeing storage* pattern, that consists in deleting unused variables or smart contracts.

Additionally, 3 architectural patterns were identified by Wessling et al. (Wessling and Gruhn, 2018).

The *Self-generated transactions* pattern lets the responsibility to the user of creating and signing transactions to interact with blockchain smart contracts. It ensures maximal security, as they keep control of their keys at all times and can verify the code to ensure correct behavior, but it leads to poor user experience and expertise is required. To facilitate this task, they can use a browser wallet (e.g., Metamask[2]) to generate and sign transactions.

[2]https://metamask.io/

The *Self-Confirmed Transactions* pattern is a tradeoff between security and usability as the website is in charge of generating transactions and the user is given the choice of signing them or not, using a browser wallet.

The *Delegated Transactions* pattern offers the most convenient experience for users, as the website handles all the blockchain-related operations. However, trust in the website is mandatory, as they have full control of keys and wallets.

RQ3.3: What are the applications of identified patterns?

Looking at the domain applications, 7 papers out of 20 targeted a specific domain, such as healthcare, big data, decentralized identity, record management, financial services, and BPM. The proximity between some of the patterns and their application domain is the reason they have been classified in the *Domain-based pattern* subsection of the taxonomy. Also, specific patterns for BPM have been proposed (Müller, Ostern, and Rosemann, 2020). They might be applied in other solutions, but their main purpose is bound to business process management. In other cases, some patterns are presented as a domain-agnostic solution coupled with implementation details in a specific application domain. For instance, Zhang et al. propose an adaptation of GoF patterns to serve healthcare solutions, using blockchain (Zhang et al., 2017).

From a technological standpoint, 6 of the 20 selected papers propose patterns for specific blockchain technology. Of those papers, 5 are focusing on Ethereum, and more specifically Solidity smart contracts. Indeed, a growing interest is shown by academics and businesses for Ethereum since its release in 2016, as its mainnet is currently the most-adopted public blockchain network for smart contract development. In this context, software patterns support many aspects of Solidity-based smart contracts. As seen before, found patterns mainly address the efficiency and the security of smart contracts, two major aspects to consider when developing Solidity-based decentralized applications. Another paper introduced a pattern for the Hyperledger ecosystem, more specifically for Sawtooth, a modular blockchain technology[3]. Looking at patterns themselves, over the 160 non-unique patterns retrieved, 28 of them were not mentioning the usage of smart contracts, 79 of them mentions the usage of smart contracts without any precision on used technology in the pattern **Solution**, and 53 patterns are proposed in the context of using a specific technology (e.g., Ethereum). However, some of the patterns might be proposed in a more generic form, thus allowing its application to other technologies. This might be the ground for future research in this domain.

5.2.2 Additional Research Questions

RQ3.4: Are some of the patterns equivalent to existing software patterns?

Since the first collection of design patterns released by the GoF (Gamma et al., 1995), many patterns have been proposed that can be applied in many contexts. As dApps have many

[3]https://www.hyperledger.org/use/distributed-ledgers

similarities with traditional applications, one aspect this work investigates is the links between existing software patterns and proposed blockchain-based patterns, either through the creation of variants or the direct usage of existing patterns in blockchain applications.

Table 5.7 introduces the list of all identified software patterns. It has been found that 22 extracted patterns mention references to existing software patterns, where 16 of them directly arise from the GoF design pattern collection. A possible explanation is that smart contracts have many similarities with objects, thus many GoF design patterns can be applied to them. For example, the *Factory* pattern is used to create instances of smart contracts from a factory contract, as it can be used in OOP (Object-Oriented Programming) to create objects from one.

On top of that, using GoF patterns with smart contracts can help to tackle their lack of flexibility, a difficult aspect to manage in dApp development. To illustrate, where the *Proxy* pattern is a good practice for protecting the access of sensitive objects in Object-oriented Programming, it is even stronger with smart contracts. As detailed in Subsection 5.2.2, the *Proxy contract* pattern can relay a function call to another smart contract. As the proxy contract allows changing the relay target, this mechanism allows to upgrade an existing smart contract by simply redeploying a new version, then updating the proxy contract relay target. Where the logic behind the proxy changed, the interface remains unchanged.

Existing pattern	Mentionned in
GoF patterns	
Proxy	• Proxy Contract (Rajasekar et al., 2020) • (Off-chain) Contract Registry (Rajasekar et al., 2020) • Proxy (Zhang et al., 2017) • Proxy (Zhang et al., 2018)
Factory	• Factory Contract (Rajasekar et al., 2020) • Abstract Factory (Zhang et al., 2017) • Abstract Factory (Zhang et al., 2018)
Flyweight	• Flyweight (Rajasekar et al., 2020) • Flyweight (Zhang et al., 2017) • Flyweight (Zhang et al., 2018)
Chain of responsibility	• Checks-Effect-Interactions (Rajasekar et al., 2020) • Dynamic Binding (Rajasekar et al., 2020)
Observer	• Reverse verifier (Rajasekar et al., 2020)
Facade	• Embedded permission (Rajasekar et al., 2020)
Memento	• Emergency Stop (Rajasekar et al., 2020)
Composite	• Incentive Execution (Rajasekar et al., 2020)
Other patterns	
Publisher-subscriber	• Publisher-subscriber (Zhang et al., 2017) • Publisher-subscriber (Zhang et al., 2018)
Mutex	• Mutex (Rajasekar et al., 2020)
Snapshot	• Snapshotting (Rajasekar et al., 2020)
Layered design	• Data Segregation (Wöhrer and Zdun, 2018)

Table 5.7: Existing software patterns reused by blockchain-based software patterns.

RQ3.5: What are the most frequently mentioned patterns and their variants across the patterns identified?

In this subsection, four patterns are introduced in detail, using the Alexandrian form, a pattern format described in the subsection 2.1.2. Exploiting the taxonomy, the researcher only selected the most representative patterns in every subcategory (On/off-chain interaction patterns, Data management patterns, Domain-based patterns, and Smart contract patterns), based on the number of references in the corpus of papers. Whenever possible, the formalization synthesizes each using the description of the mentioned academic work. Each pattern was completed by the researcher's own analysis of the pattern, whenever specific information required by the pattern format was found missing.

Off-chain Data Storage pattern

The *Off-chain data storage pattern* consists in storing a hash of off-chain data in a smart contract, to be able to verify the off-chain data integrity later. This pattern belongs to the

"On/off-chain interaction pattern" category and has been found 6 times in the corpus of papers.

Context - As the blockchain is replicated among nodes, some applications might consider storing data within the blockchain, ensuring their integrity (Rajasekar et al., 2020; Xu et al., 2018).

Problem - Allowing users to store on-chain data without any limit of storage could hamper the network's functioning. Therefore, many blockchain networks enforce a block size limit to tackle the size growth issue of blockchain over time. Even if the size limit suits the needs of the user, storing data on-chain is prohibitively expensive. Thus, how can the user store data on-chain while taking advantage of blockchain immutability and integrity (Xu et al., 2018)?

Forces - Using this pattern implies balance forces. The first one is cost, as storing data on-chain is expensive and even more if using a smart contract to keep the possibility to perform operations on them directly on-chain. Then, scalability, because storing large files on a blockchain is difficult as they are replicated across all nodes (Xu et al., 2018). Finally, the immutability level has to be considered: storing a hash on-chain does not offer the same protection as storing the file itself. Indeed, it can still be modified or deleted off-chain.

Solution - Store the data off-chain, then calculate a hash of those data. Store the result on-chain in a smart contract, possibly associated with metadata (e.g., resource location, description, ...) (Rajasekar et al., 2020; Eberhardt and Tai, 2017). As hashing data is a one-way function, data confidentiality is preserved, and users can check the integrity of their data using the immutable hash stored on-chain (Xu et al., 2018).

Example - A company that wants to store proof that a legal contract is signed can hash the contract after its signature and store the result on-chain. Thus, if another company denies the authenticity of a contract, it is possible to prove the existence of the document as well as its metadata (e.g., signature time).

Resulting context - Data are kept off-chain, and stay confidential, but their integrity can still be accessed using the on-chain hash. It is inexpensive to store the hash on-chain compared to the file itself, considering the size of such a file is large. However, the file is still vulnerable to deletion or tampering, as the hash itself cannot help retrieve a lost file or deleted content. Adequate measures should be taken to preserve off-chain data.

Related patterns - According to Liu et al., this pattern is directly related to the *Low contract footprint pattern* in (Eberhardt and Tai, 2017), as the latter propose to minimize the number and size of on-chain transactions to save costs, notably with optimizing write operations (Liu et al., 2020). As the *Off-chain data storage pattern* only stores a hash on-chain, this cost is kept low.

Known uses - The Government of Estonia's e-health solution utilizes blockchain as a "fingerprint" registry to ensure the integrity of e-health records (Lemieux, 2017). Factom[4], a

[4]https://www.factom.com/

blockchain for building records systems, implements this pattern by systematically hashing files sent to the blockchain. Only the hash is kept on-chain after the operation.

State Machine pattern

The *State machine pattern* proposes to manage smart contract state transitions through state machines, to break the problem of state changes into simple state transitions. It belongs to the "Domain-based pattern" intermediate category. As each of the patterns included in this category has only been found one time in selected papers, the researcher decided to select the *State machine pattern* for a thorough introduction as this pattern can be used in many different scenarios, including basic implementations of smart contracts.

Context - When leveraging smart contracts, state changes are often performed. Depending on the purpose of the smart contract, many state changes might occur during its lifecycle.

Problem - A smart contract might be difficult to design if many state changes occur, as complex logic may have to be implemented.

Forces - Some forces are bound to the usage of this pattern: the complexity of the smart contract to design and its efficiency, as depending on the implementation of the state changes part, the contract might be efficient or cumbersome to use.

Solution - Apply a state machine to model and represent different contract stages and their transitions in the smart contract (Wöhrer and Zdun, 2018).

Example - A company that wants to leverage a business process on-chain with multiple steps that might trigger automatic operations might be tempted to use the *State machine pattern* in order to model and perform the state changes within the contract.

Resulting context - The state machine breaks complex problems into simple states and state transitions (Wöhrer and Zdun, 2018), resulting in a more efficient smart contract.

Related patterns - In the *Confidential and pseudo-anonymous contract enforcement pattern* (Six et al., 2020), a state machine can be employed in the smart contract used by the pattern to handle state changes of the associated legal contract on-chain.

Known uses - The DutchMachine smart contract implements a state machine for handling auctions (Wöhrer and Zdun, 2018).

Tokenization pattern

The third presented pattern is the *Tokenization pattern*. Classified in the "Data management pattern" intermediate category and mentioned 4 times, this pattern consists in representing an asset by a token, to facilitate its exchange on blockchain networks.

Context - Through a blockchain network, it is possible to send transactions and interact with smart contracts without any third party as an intermediate. Such a network enables the exchange of value directly between one user to another, notably with the exchange of native cryptocurrency.

Problem - Native fungible blockchain tokens (e.g., Bitcoin, Ether) often serve as the native cryptocurrency of the associated blockchain network. In some cases, they can also be used as token support to track assets, but their capabilities are limited. Indeed, extending the concept of value exchange for other types of assets (e.g., other currencies, art, houses, ...) is not a straightforward process due to the dissimilarity between those assets.

Forces - Some forces are bound to this pattern: authority, as it should be ensured that the on-chain asset is the authority source of the correlated asset (Xu et al., 2018), and liquidity, as blockchain can enable a frictionless exchange of value.

Solution - Model many types of assets on blockchain using tokens. Two types of tokens can be differentiated: fungible tokens that are indistinguishable from each other, and non-fungible tokens (NFTs), representing a unique asset with its own properties. Smart contracts can thus be used as a data structure to handle the tokens and associated operations (transfer, deletion, ...) (Xu et al., 2018), but also enhance their capabilities.

To illustrate, Ethereum proposes two different standards to create fungible and non-fungible tokens using smart contracts, that are respectively ERC20 and ERC721 tokens (Di Angelo and Salzer, 2020). Using these standards simplifies the usage of tokens, as on-chain applications and users can rely on standard interfaces to interact with all of the smart contracts that implement tokens for their usage. Other standards exist in the Ethereum ecosystem to improve their usability in different contexts. For instance, the ERC1155 can also be mentioned as it allows the usage of both fungible and non-fungible tokens (ERC20 and ERC721) in the same smart contract. ERC998-based tokens go even further by regrouping multiple tokens under a single token (commonly called a basket). This simplifies their exchange between users and enables other use cases (e.g. a service proposing users to invest in a specific basket of tokens all at once). A variant, the ERC3664, allows the combination of multiple NFTs into a single one. This composability of NFTs is notably useful in the gaming industry (e.g. a set of items merged into a better one).

Where tokens can be used to represent different types of assets, they can also be used for other purposes. One of the most popular uses in this context is token governance: depending on the amount of owned tokens, users could vote on important decisions. For instance, by owning governance tokens that represent a share of an on-chain fund, users could vote about the usage of those funds, such as their investment in other protocols. Another similar concept is staking, notably for Proof-of-Stake blockchains: by locking a defined amount of their tokens at stake, users could be entitled by the consensus algorithm to create new blocks.

Example - A real estate company can use non-fungible tokens to represent the ownership of houses directly into the blockchain. Ownership of a house can then be directly exchanged on-chain, and a complete history of transactions can be retraced for a house.

Resulting context - Assets are tokenized on-chain and can be easily sent between users. Using smart contracts, many features can be implemented along with the tokens, such as royalties, sales, or burns (i.e., destroying tokens).

Related patterns - The *Address mapping pattern* can be used as a complement to map blockchain accounts (e.g., public addresses) with owned tokens. The *Poll* pattern might use the *Token* pattern to materialize votes as tokens and keep track of them.

Known uses - The *Tokenization* pattern has already been applied in a tremendous number of domains. For instance, stablecoins (e.g., Tether[5]), consist in emitting fungible tokens on-chain that keep the same value as an underlying asset (e.g., US Dollar) using different strategies. This enables many other use cases relying on the usage of fiat currencies, such as frictionless currency swaps. Another use case is the usage of NFTs in art. Many artists have digitalized their art as NFTs to sell it on on-chain marketplaces, such as OpenSea[6].

Proxy contract pattern

The fourth and last presented pattern is the *Proxy contract pattern*. It belongs to the "Smart contract pattern" intermediate category and appeared 6 times in found patterns.

Context - In a blockchain, data becomes immutable after addition. This concept is also applied to smart contracts, that cannot be modified after their deployment on-chain (Marchesi et al., 2020).

Problem - If a smart contract has to be changed (e.g. for upgrades, bug correction, ...), the developer has to deploy another version of the contract and manually change the other contracts that reference the old contract (Marchesi et al., 2020). In the best case, this is a cumbersome task, and it might even not be possible in certain cases.

Forces - The problem requires balancing the following forces: first, immutability, as deployed smart contracts are designed to be immutable, and upgradeable, as proposing features to allow upgradeability enhances designed smart contracts.

Solution - Using a proxy contract, a user can query the latest version of a target contract. The proxy contract will relay the request to the target contract (Wöhrer and Zdun, 2018). By replacing the reference of the target contract with a new one, it is possible to easily upgrade parts of the decentralized application (Marchesi et al., 2020).

Example - A user can request a proxy contract as the bridge for a decentralized application, such as the latest version of a decentralized cryptocurrency exchange.

Resulting context - Proxy contracts can be used to easily access the latest version of a contract, without requiring storing the latest contract addresses off-chain. Reference updates can easily be performed by requesting the proxy contract with the latest contract address.

[5]https://tether.to/
[6]https://opensea.io/

Related patterns - The *Data contract pattern* can be implemented along the *Proxy contract pattern* as the proxy will allow updating the logic used to access the data contract without updating the data contract itself. The *Contract registry pattern* is related to the *Proxy contract pattern*, as the contract registry has a reference to all the latest versions of the contracts, whereas the proxy only references one contract.

Known uses - A security company named OpenZeppelin proposes a generic implementation of the *Proxy contract pattern* for Solidity-based smart contracts (OpenZeppelin, 2019). Uniswap, a decentralized exchange on Ethereum, uses proxy contracts to forward user transactions to the exchange smart contract[7].

RQ3.6: What are the current gaps in research on blockchain-based patterns?

Regarding current gaps in research on blockchain-based patterns, the lack of non-design patterns can be mentioned. Among the 114 patterns retrieved, only 3 of them are architectural patterns and 14 of them are idioms. Although design patterns are a very compelling solution for the design of robust and efficient applications, exploring new forms of blockchain architectures, then formalizing them as architectural patterns could benefit a lot to blockchain dApp design. Taking back the examples mentioned in Subsection 5.2.1, Mavridou et al. show the strong impact on software quality using architectural patterns (Mavridou and Laszka, 2018). On one side, applying the *Self-Generated Transactions* pattern means letting the task of signing transactions to users on the client-side, thus ensuring no one aside from the client has access to the keys. On the other side, using the *Delegated Transactions* pattern lets full control of the funds to the application. This can be convenient for users without knowledge of using a blockchain wallet but adds a potentially vulnerable third party into the balance. Such research could be conducted by exploring the existing literature or applications to find innovative ways of organizing decentralized application components. For example, Tonelli et al. propose a microservices system where smart contracts are services themselves (Tonelli et al., 2019). As the *Microservices* architectural pattern already exists, adapting it for blockchain could lead to a new way of designing a loosely coupled smart contract system with its own advantages and liabilities.

Regarding the idioms, and the other smart contract patterns found, all of them deal with Solidity, except one (Hyperledger Sawtooth). Although Ethereum is the most used public blockchain for decentralized applications as of today, other languages could have been considered. Rust, a high-level compiled language, is used for smart contract development by many blockchain technologies, such as ink! from Polkadot[8], or Rust for Solana[9]. Formalizing new idioms and patterns in this context could help improve code quality and security. In addition, existing patterns in Solidity could also be translated for other blockchains. As an example, the *Freeing storage* idiom could also be applied to other public blockchains where freeing the storage refunds a defined amount of money (Marchesi et al., 2020).

[7]https://etherscan.io/address/0x09cabec1ead1c0ba254b09efb3ee13841712be14
[8]https://github.com/paritytech/ink
[9]https://github.com/solana-labs/solana

5.3 Threats to Validity

Internal threats to validity: in this literature review, the Kitchenham et al. methodology has been applied to systematically conduct the study, from the selection of papers to the collection of data (Kitchenham and Charters, 2007). Several steps in this method consisted of filtering papers based on title and abstract, or quality (assessed by QQs). This phase is subject to researchers' bias, as the process of discarding papers based on these pieces of information is subjective. To tackle this threat, multiple researchers performed this filtering, and disagreements were discussed, then settled. Also, the query applied to retrieve papers can be mentioned. The terms in the query were only searched in titles, abstracts, and keywords to improve the precision of the request, yet some papers might have been missed. To overcome this, backward and forward snowballing has been used to retrieve papers that cited or have cited studies found while performing the systematic literature review.

External threats to validity: as this SLR focuses on academic literature, blockchain-based software patterns may have been missed from other grey literature sources, such as blog posts or technical articles. Nevertheless, this threat is limited, as many papers identified during the literature review as containing papers was citing sources from the grey literature. Yet, future works may consist in carrying a multivocal literature review that includes grey literature (Garousi, Felderer, and Mäntylä, 2019).

Construction threats to validity: the exclusion criteria can be mentioned. Although only studies that contain patterns were included in the final corpus of papers, there were no exclusion criteria for patterns that are correctly formulated but lack prior application in different contexts. Indeed, one requirement to adapt a solution into a pattern is that the solution should have extensively been tried and tested. Nevertheless, from the data collected throughout the literature review (e.g. number of citations, application examples for each pattern, etc.) a degree of reliability for each pattern can be computed.

Conclusion threats to validity: the collection, categorization, and grouping of patterns may lead to errors, as this is a manual process. Yet, collected patterns from one researcher were double-checked by others, and the resulting collection was also reviewed during a meeting to identify errors. In parallel, great attention has been paid to not merging patterns that are not strictly identical, to avoid missing variants of patterns that serve in different contexts.

The resulting taxonomy was also subject to bias. For instance, even if different methods were used to generate category names, the final decision is up to the taxonomy builders. Selecting the high-level dimensions (meta-characteristics) is also a subjective task that has a high impact on the construction of the taxonomy. To limit such bias, the methodology from Nickerson et al. was applied (Nickerson, Varshney, and Muntermann, 2013). Also, identified patterns were easily classified into the final version of the taxonomy, hence, the produced version of the taxonomy satisfies the goals initially described before its construction.

5.4 Related Works

Using a systematic literature review to collect patterns is a strategy that has already been used in other fields. For instance, Juziuk et al. have gathered 206 design patterns on multi-agent systems (MAS) from the literature (Juziuk, Weyns, and Holvoet, 2014). Authors have also identified the links between found patterns and proposed classification to group patterns under different categories and subcategories. The study also mentions several research gaps in the literature for MAS, such as the lack of standardization when describing a pattern despite the existence of several pattern formats, the lack of links between patterns that do not belong to the same category, and the lack of mentioned applications of presented patterns. This chapter also shares the same conclusions.

In another literature review, 44 architectural patterns are extracted from a corpus of 8 papers about microservices (Osses, Márquez, and Astudillo, 2018). A taxonomy is also provided to classify the different patterns. It has been found that identified patterns are mostly bound to five quality attributes: scalability, flexibility, testability, performance, and elasticity.

Washizaki et al. have also performed a systematic literature review of IoT software patterns and have collected 143 architecture and design patterns from a corpus of 32 papers (Washizaki et al., 2020). They have also identified that 57% of all found patterns are non-IoT patterns, thus meaning IoT systems are designed through a conventional architecture perspective, something that has also been identified in this work through the "On/off-chain interaction pattern" category as well as GoF-based design patterns. This chapter follows the same path as others by proposing a taxonomy and a collection of patterns. To the best of the researcher's knowledge, this is the first attempt in the blockchain-based pattern literature to propose such work.

5.5 Conclusion and Future Works

Ensuring the high quality and efficiency of newly built decentralized applications is a challenge of uttermost importance for the future of blockchain. Software patterns are a promising solution to address this challenge, as they ensure commonly occurring problems in a given context are addressed with extensively tested solutions. In this chapter, a systematic literature review is performed on the available blockchain pattern literature to identify existing software patterns and classify them into a comprehensive taxonomy.

During this systematic literature review, 20 studies were selected out of which 160 patterns were extracted. After duplicate removal, 114 unique patterns were found and regrouped in a taxonomy. The taxonomy consists of 4 main categories and 15 subcategories and has been built using a construction taxonomy methodology (Nickerson, Varshney, and Muntermann, 2013). This chapter also discusses the links between blockchain-based software patterns, but also their relation with existing software patterns such as the GoF design pattern collection. One finding is that many patterns from this collection are translated into blockchain patterns, such as the *Proxy* or *Factory* pattern. Application domains of patterns have also been discussed: among the corpus of papers, 3 papers directly linked to domain-based patterns were found, respectively healthcare (Zhang et al., 2018), collaborative business processes (Müller, Ostern, and Rosemann, 2020), and decentralized identity (Liu et al., 2020). Finally, research gaps are addressed, by enlighting the scarcity of architectural patterns and idioms for the design of blockchain-based architectures, and the concentration of patterns on one blockchain protocol (Ethereum).

This chapter, in conjunction with Chapter 6, aims to propose a comprehensive and reusable knowledge base of blockchain-based software patterns. In this chapter, the knowledge has been extracted in a semi-structured format, using a systematic method. In Chapter 6, this knowledge is reused to build an ontology of blockchain-based software patterns. Finally, this work also contributes to the state of the art of blockchain-based patterns, through a taxonomy that will help to classify newly created patterns in comprehensive categories, a systematic literature review to map and describe the existing literature on blockchain-based patterns within the taxonomy, and highlight research gaps that could be addressed in further studies.

Chapter 6

Recommendation Engine for the Selection of Adequate Blockchain-based Software Patterns

Key takeaways

- An ontology of blockchain-based software patterns is designed, reusing the pattern collection created in the last chapter.
- A platform is created to allow the practitioner to explore the ontology of patterns, then obtain recommendations on blockchain-based software patterns.
- This platform is the second part of BLADE, and the second artifact of the framework with regards to the DSR approach. It addresses the technical research question RQ4.

The previous chapter contributed to forming a collection of blockchain-based software patterns, classified into comprehensive categories. However, their usage in recommendation tools to assist the design of a blockchain application is still a challenge. Although these patterns have been stored in a public GitHub repository, they are still stored as a plain Excel file. Thus, it requires finding adequate support to store these patterns in order to ease their reuse. Ontologies are a good candidate to address this issue. By defining a set of ideas and categories that reflect the subject, an ontology can show the qualities of a subject area and how they are related. Here, the subject area is both the organization of software patterns and the blockchain-based software patterns themselves.

This chapter addresses the reuse of patterns throughout the following technical research question (**RQ4**): *How to design a blockchain-based software patterns recommendation platform that simplifies the completion of this task for non-experts so that practitioners can choose adequate patterns during the design phase of blockchain application development?*

As in Chapter 4, two requirements for this can be derivated from this technical research question:

1. The platform must ease the task of blockchain-based software pattern selection for non-expert users.
2. The platform must recommend blockchain-based software patterns that satisfy the user requirements.

Satisfying these requirements will imply a positive outcome for the stakeholders (i.e. the practitioners) goals, that motivate this research work.

First, an ontology[1], named blockchain-based software patterns ontology, is proposed. It is composed of two distinct subspaces:

1. Blockchain-based patterns subspace: a set of classes and individuals related to blockchain-based patterns.
2. Pattern proposal subspace: a subspace for organizing the knowledge related to software patterns with regard to the academic literature.

In this goal, the Network of Ontologies (NeOn) method is used to guide the construction of these two subspaces (Suárez-Figueroa, Gómez-Pérez, and Fernández-López, 2012). NeOn eases the reuse of existing non-ontological knowledge, that is the collection of patterns, and eases the reusability of the ontology in a network of other related ontologies. Then, a web platform is proposed to navigate into the ontology of patterns but also to perform recommendations based on a questionnaire presented to the user. By answering several design questions, the recommender is able to output a sample of adequate patterns that suits the needs of the user. This web platform is the second part of BLADE, and reuses the blockchain recommendations to output results related to the chosen blockchain technology.

This chapter is organized as the following. Section 6.1 introduces the NeOn method used to build the ontology from the knowledge acquired in Chapter 5. It also introduces the competency questions, that lead the design of the ontology. Section 6.2 describes the resulting ontology, as well as a web platform built to query the ontology and leverage its content without deep knowledge in the field. The running example introduced in Chapter 2 is then reused in Section 4.4 to illustrate the functioning of the web platform. The validity of the artifact (i.e. the ontology and the platform) to answer the technical research question RQ4 is assessed in Section 6.4, then the possible threats to validity are discussed in Section 6.5. Some related works are discussed in Section 6.6, then Section 6.7 concludes the chapter and introduces envisioned future works.

[1]For the sake of conciseness, the terms "the ontology" and "the blockchain-based software patterns ontology" may be used interchangeably.

6.1 Ontology Construction

The first research work introduced in this chapter is the construction of the ontology. Building an ontology is not a straightforward task, notably as this work has to reuse existing non-ontological resources, that are the collection of blockchain-based software patterns and the related taxonomy of patterns. To carry out this work, a method for ontology construction has been chosen, that is NeOn (Suárez-Figueroa, Gómez-Pérez, and Fernández-López, 2012). This method is introduced in the following subsection.

6.1.1 Construction Method

For the construction of the ontology, the NeOn method has been chosen due to its inherent flexibility and focus on the reuse of both ontological and non-ontological resources in a structured manner (Suárez-Figueroa, Gómez-Pérez, and Fernández-López, 2012). The goal of this method is not to replace the DSR method used in the context of the thesis, but rather complement it: NeOn was used during the treatment design step of the design cycle used to address technical research questions.

NeOn does not force strict guidelines that must be followed sequentially upon the ontology design: a set of scenarios is given and the designer is free to select, and if needed, adapt any scenario that suits their needs. In this work, two of the scenarios envisaged within NeOn have been chosen. The first scenario mainly concerns ontology construction from the ground up, to produce a new, standalone, ontology. The principal motivation for this choice is the absence of literature on existing ontologies covering blockchain-based patterns. There is also the inability of existing software pattern ontologies to adequately capture the results of the literature review upon which the pattern proposal subspace is based, hence the need to produce a new ontology to also cover this domain of interest. The second addresses the specific aspects of reusing non-ontological resources in the construction of ontologies. This is key since the blockchain-based software pattern ontology will be primarily based on the reuse of previous results obtained through a systematic literature review (Six, Herbaut, and Salinesi, 2022). The NeOn methodology proposes a set of closely related life cycle models linked to the different scenarios it incorporates. The six-phase waterfall life cycle has been chosen (Suárez-Figueroa, Gómez-Pérez, and Fernández-López, 2012), given the need to reuse non-ontological resources (Figure 6.1).

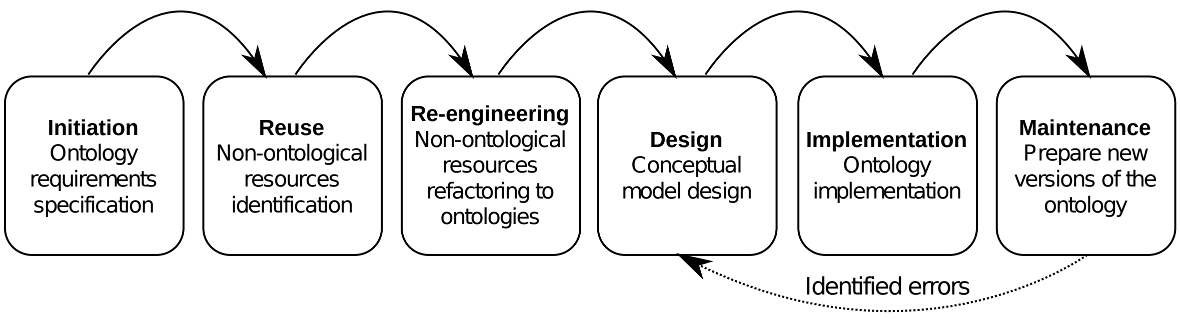

Initiation
Ontology requirements specification
Reuse
Non-ontological resources identification
Re-engineering
Non-ontological resources refactoring to ontologies
Design
Conceptual model design
Implementation
Ontology implementation
Maintenance
Prepare new versions of the ontology
Identified errors

Figure 6.1: NeOn framework workflow.

6.1.2 Initiation

In the initiation phase, one important step in the construction of an ontology is the specification of requirements through an Ontology Requirement Specification Document (ORSD) (Suárez-Figueroa, Gómez-Pérez, and Fernández-López, 2012) that serves as an agreement on which requirements the ontology should cover, its scope, implementation language, intended uses and end users. The ORSD facilitates the reuse of existing knowledge-aware resources in the creation of new ontologies (Suárez-Figueroa, Gómez-Pérez, and Fernández-López, 2012). Competency Questions (CQs) is a way to introduce the functional requirements of an ontology; their coverage, ideally in a generalizable manner, allows one to consider the ontology functionally complete. The CQs are not formulated as functional requirements, but rather as questions that can be translated to requirements afterward (e.g. for CQ4, the ontology shall allow the user to retrieve the possible relations between two patterns). For the sake of brevity, only the CQs are detailed in this chapter, listed in Table 6.1. However, more information about the ontology's purpose can be found in the introduction or in the full ORSD, available on GitHub[2].

Table 6.1: Ontology competency questions.

CQ1	What are the classes of patterns in the blockchain area and how can they be differentiated and characterized?
CQ2	What are the different propositions of patterns in the academic literature and how can their acceptance by others be quantified?
CQ3	How can the concept of pattern from their possible descriptions in different sources be differentiated?
CQ4	What are the types of relations or constraints that can connect two patterns together?
CQ5	What are the different problems bound to the design and implementation of blockchain applications?

These competency questions define the two main purposes of the ontology. The first purpose is the definition of a sound structure to store software patterns, especially patterns proposed in the academic literature (CQ3). As these patterns might not have been applied enough in real use cases, one objective is to quantify the acceptance by others (CQ2) of a proposed pattern in a study. One possible solution to this problem is the usage of paper citations, described in Section 6.2.1. The relations between these patterns are also an important topic, as patterns are often used together to address larger-scale problems (CQ4).

The second purpose is the storage of blockchain-based software patterns, taking their specificities into account. It notably includes their classification into comprehensive categories (CQ1) to guide the reader in the space of blockchain-based software patterns, as well as the problems they address (CQ5).

[2] https://github.com/harmonica-project/blockchain-patterns-ontology

The process outlined by Suarez et al. was followed to validate the requirements specification, within the larger framework of the NeOn methodology (Suárez-Figueroa, Gómez-Pérez, and Fernández-López, 2012). Since the ontology was to be built with extensibility in mind, should new requirements arise, the queries that correspond to the competency questions act as a test suite that ensures the ontology remains conformant as it evolves.

6.1.3 Reuse and Re-engineering of Non-Ontological Resources

The construction of the blockchain-based software pattern ontology formalizes the knowledge gained from a previous systematic literature review. The ontology incorporates knowledge from two different non-ontological resources that can both be found on GitHub[3]:

1. A collection of 160 patterns that were identified during the literature review (Chapter 5) within 20 different papers; out of which 114 unique patterns have been derived.
2. A taxonomy that emerges from the categorization of the results in the literature review, and is comprised of 4 main categories and 14 subcategories.

More details are given in the introduction of the ontology conceptual model in the results presented in Section 6.2.

Each of the collected patterns is described by a set of attributes, e.g., a *Name*, a *Context and Problem*, and a *Solution*. The citations count for each paper that proposes one or more patterns have also been collected. Thus, the reliability of a pattern can be assessed more easily: a pattern proposed in a paper cited multiple times can be considered, to some extent, to be more trustable than a pattern from a non-cited study. More rationale on the usage of these citations to assess pattern reliability is given in Subsection 6.2.2.

The domain, programming language, implementation examples, and blockchain technology associated with the pattern are also collected if available. Indeed, some patterns may be proposed by paper for a specific programming language (the Solidity smart contract language[4]), or in the context of a specific domain (e.g., patterns to enable decentralized identity on blockchain). Also, different types of relations between patterns were identified throughout the study: *Created from*, *Variant of*, *Requires*, *Benefits from*, and *Related to*. As the application of a specific pattern might require considering other patterns, its relations to others should be made explicit. Further details about these relations are given in Subsection 6.2.1. Patterns are classified in one of three categories depending on their general purpose: *Architectural patterns* that regroup patterns impacting the general structure of the application (elements, connections); *Design patterns* that are a way to organize modules, classes, or components to solve a problem; and *Idioms*, solutions to a programming language-related problems.

[3]https://github.com/harmonica-project/blockchain-patterns-collection
[4]https://docs.soliditylang.org/

6.2 Blockchain-based Software Pattern Ontology

The application of the NeOn method resulted in a blockchain-based software pattern ontology, and a querying tool that can be used to leverage the ontology through different ways of retrieving and then selecting blockchain-based patterns.

6.2.1 Ontology Overview

The primary result of the NeOn blockchain-based software pattern ontology construction process is depicted in the conceptual model[5] shown in Figure 6.2. This ontology proposes an approach based on the existing literature (Chapter 5) to store, classify, and link blockchain-based software patterns but also to infer new knowledge using inference rules, such as new relations between patterns. As the figure shows, the driving idea of the ontology is: (a) the explicit distinctions between patterns, variants, and pattern proposals, (b) proposals and their relations with others, and (c) design problems outside the scope of patterns.

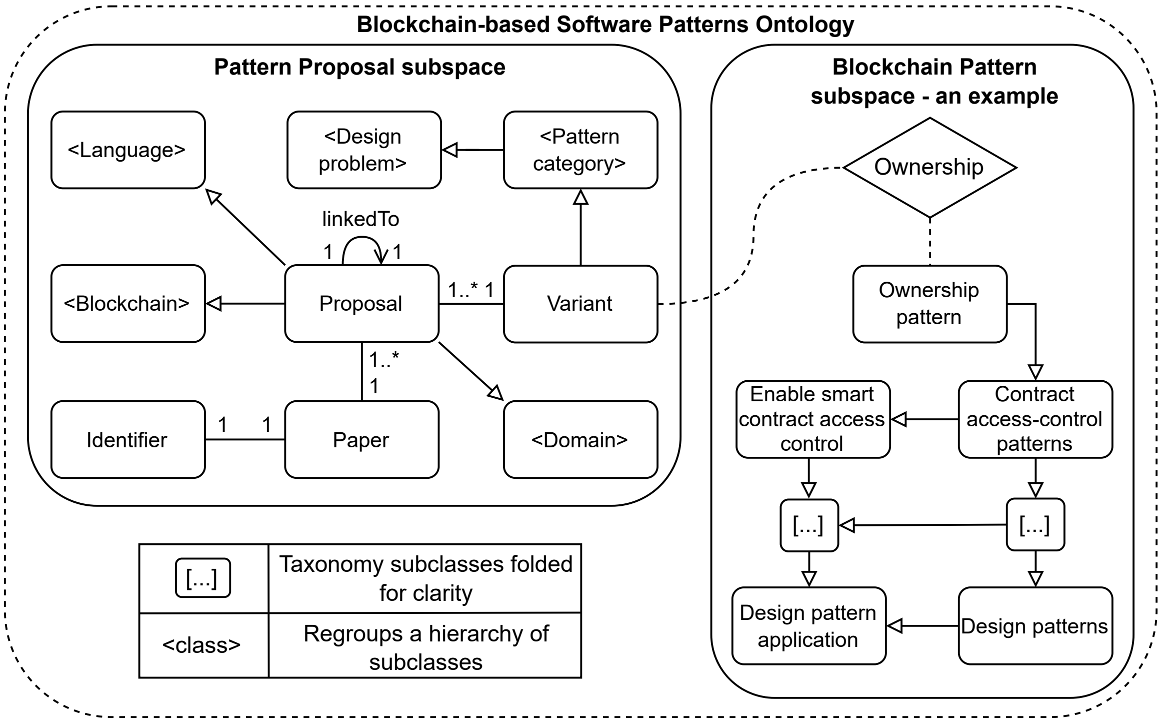

Blockchain-based Software Patterns Ontology
Pattern Proposal subspace
<Language>
<Design problem>
<Pattern category>
linkedTo
1
1
<Blockchain>
Proposal
1..* 1
Variant
1..*
1
Identifier
1
1
Paper
<Domain>
Blockchain Pattern subspace - an example
Ownership
Ownership pattern
Enable smart contract access control
Contract access-control patterns
[...]
[...]
Design pattern application
Design patterns
[...]
Taxonomy subclasses folded for clarity
<class>
Regroups a hierarchy of subclasses

Figure 6.2: Blockchain-based software pattern ontology with an exemplified section.

Using ontologies as a support to store gathered knowledge about patterns also enables the usage of inference engines to find new relations between entities. This aspect is described along with the ontology content in this subsection. Using an ontology also allows connecting it with other ontologies. Although it is beyond the scope of this work, it is possible to connect patterns with blockchain technologies expressed in other blockchain ontologies.

[5]For the sake of clarity, some subclasses of the ontology are hidden.

The central element of this model is the *Proposal* class. A proposal is a pattern introduced within an academic paper. In the current form of the pattern proposal subspace, all sources of patterns are academic papers, thus this class is not implemented yet. Nonetheless, a class named *Source* will merge other types of sources such as technical documentation (e.g., OpenZeppelin proxy pattern: OpenZeppelin, n.d.) in the future. This improves the extensibility of the model, as patterns might be proposed in many different sources.

Each paper is linked to a *Identifier*, which can take the form of a DOI (Digital Object Identifier) or an ArXiv ID. For each paper present in the pattern proposal subspace, its citations have been included as identifiers individuals and linked to the paper using a *references* object relation. This system allows inferring the citations of a specific paper, then makes pattern recommendations where the number of citations of a paper is taken into account to evaluate the score of a specific proposal (and by extension, a specific pattern). More rationale on pattern recommendation is given in Subsection 6.2.2.

In this model, a proposal is linked to a variant. A variant inherits from a specific `Pattern`, and represents one of its possible forms. Indeed, variants are used to express the variability of a pattern: two variants of a pattern might be close enough to address the same problem and solution, but may vary in some aspects (e.g., implementation).

Figure 6.3 shows an example of this concept. In this example, purple arrowed plain lines represent the relation between classes and instances, blue arrowed plain lines the class inheritances, and dashed arrowed lines the relations between a variant and a proposal. The *Oracle* pattern proposed by Xu et al. (Xu et al., 2018) is an individual instance of *Proposal* and attached to the *Oracle* variant, an individual of the *Variant* and *Oracle* class. The distinction between proposals and resulting patterns is important as in some cases multiple papers proposed the same pattern using different words, templates, and for different domains or blockchains. As such, a *Proposal* inherits from a specific blockchain, domain, or language[6]

In this conceptual model, a *Proposal* is described by a *Context and Problem*, that gives a rationale for the purpose of the pattern and addressed problems, and a *Solution* field to introduces the different elements composing the pattern solution. This structure for pattern description is derived from the two main pattern formats (GoF pattern format and Alexandrian form, Tešanovic, 2005), usually used by researchers and practitioners to express software patterns. Because of the lack of standardization across the literature on the description of patterns, only the context, problem, and solution have been kept to describe a pattern in this pattern proposal subspace.

Proposals can also be linked together, using 5 different relation types that were identified from the systematic literature review:

- *Created from* - when a pattern directly takes its sources in another.
- *Variant of* - when a pattern is a variant of another.

[6]For the sake of clarity, subclasses of blockchain system, domains, and language (e.g., respectively Ethereum, IoT, or Solidity) are hidden in the provided conceptual model.

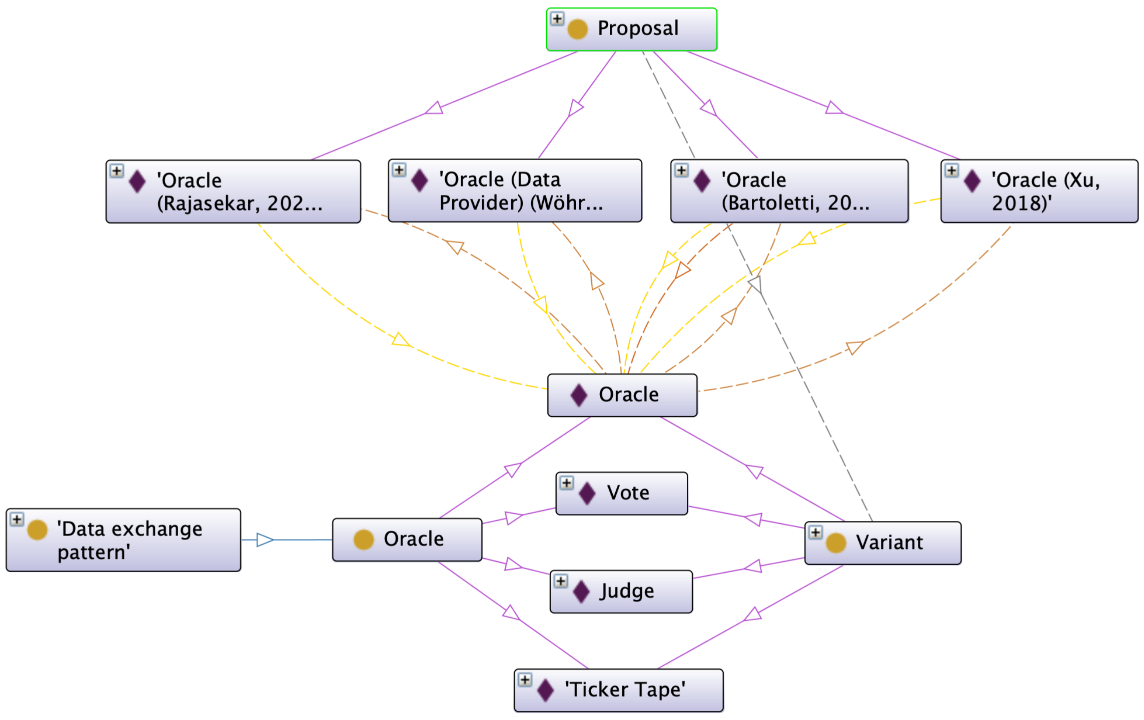


Figure 6.3: Oracle pattern ontology example.

- *Requires* - when a pattern has to use another to perform well once implemented.
- *Benefits from* - when a pattern might use another to perform well once implemented.
- *Related to* - to identify a weak relation between a pattern and another (e.g., "see also").

By using inference, it is possible to translate these relations from proposals to variants, creating new knowledge about possible relations between patterns. Semantic Web Rule Language (SWRL) rules have been written for the inference engine to generate such relations. As an example, the following rule translates a *benefitsFrom* object relation from two proposals to their corresponding variant (Equation 6.1).

$$\begin{aligned} \forall\ (p_1, p_2) \in P\ and\ (v_1, v_2) \in V, \\ p_1\ benefitsFrom\ p_2\ \cdot\ p_1\ hasVariant\ v_1\ \cdot\ p_2\ hasVariant\ v_2 \\ \implies v_1\ benefitsFrom\ v_2 \end{aligned} \tag{6.1}$$

The subclasses of the *Pattern* class emanate from the reused taxonomy for blockchain-based patterns, built in its related systematic literature review (Chapter 5). For instance, the *Oracle* variant from Xu et al. (Xu et al., 2018) is linked to the *Oracle* pattern class, that inherits from the *Data exchange pattern*, then *On-chain pattern*, *Design pattern*, and finally *Pattern*.

Although this hierarchy exists in the blockchain-based software pattern ontology, it is not shown in Figure 6.2 for clarity.

To further refine this part of the ontology, each *Pattern* addresses a specific *Design problem*. By extension, each subclass of *Pattern* addresses a design *Design problem* subclass. Figure 6.4 illustrates this aspect with the classes that directly inheritate from *Pattern*. The orange dashed line represents a *addressProblem* relation between a *Pattern* and a *Design problem*.

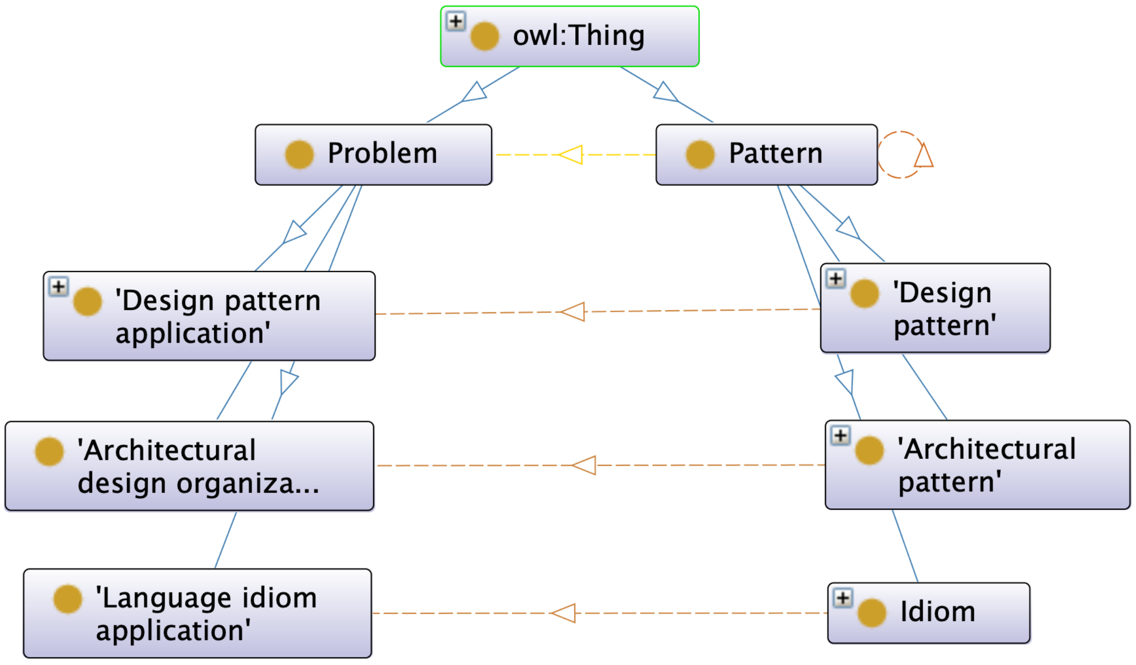
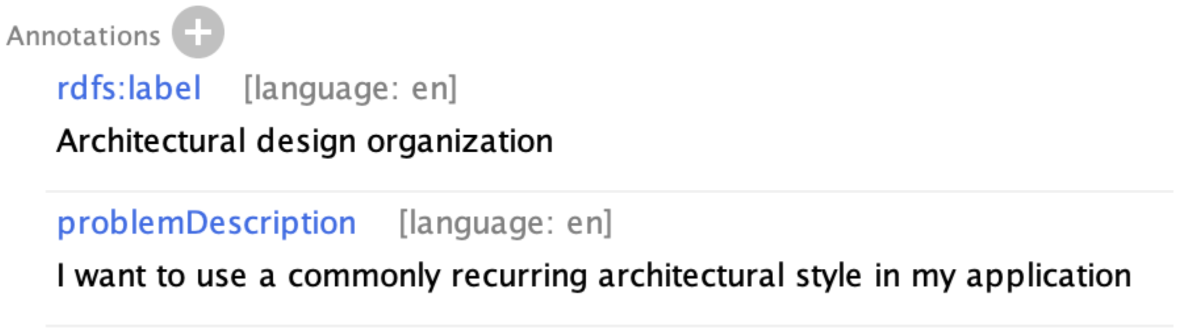

owl:Thing
Problem
Pattern
'Design pattern application'
'Design pattern'
'Architectural design organiza...
'Architectural pattern'
'Language idiom application'
Idiom

Figure 6.4: Example of relations between Patterns and Design problems.

Also, each problem has been assigned an associated literal question, notably used for recommendations (Figure 6.5).


Annotations

rdfs:label [language: en]

Architectural design organization

problemDescription [language: en]

I want to use a commonly recurring architectural style in my application

Figure 6.5: Example of question associated to the *Architectural design organization* subclass.

These questions have been designed along the construction of the design problem taxonomy to give a literal sentence of the problem. The question is presented as an affirmation (here, a user story sentence), that can be answered by yes or no. For instance, the question associated with the *Smart contract usage* design problem, solved by the *Smart contract patterns* is “I want to execute part of my application on-chain”. Such an affirmation can be thus presented as a question to the user and answered positively or negatively, to guide pattern recommendations.

6.2.2 Ontology Querying Tool

In parallel with the ontology, a tool was designed to leverage it without having to use specific tools such as Protégé[7]. Using this tool facilitates access to ontology content for non-experts, but querying the ontology directly through SPARQL[8] queries is also a possibility. This tool has two main features: the explorer and the recommender, described in the following sections.

Explorer

The first one is the explorer feature, which allows diving into ontology knowledge through the presentation of all available patterns in a grid. The purpose of this section is to link the solution domain (the list of patterns) to the problem domain (user requirements and goals). Indeed, any user reading available patterns descriptions might find some that suit their goals. The application shows each pattern’s name but also the number of linked proposals. By clicking on the pattern card, the user can consult the context, problem, and solution for each pattern variant and proposal. She also has access to a list of linked patterns following the same notation as defined in the pattern proposal subspace. The tool allows filtering patterns out, using the proposal’s respective domains, blockchains, and languages. For instance, a user can select Ethereum as the desired blockchain and filter out every non-corresponding pattern.

Recommender

The second part of the tool is the recommender feature. Contrary to the explorer feature, any user can leverage the recommender to navigate from the problem domain (a set of questions asked by the user), to the solution domain (a set of patterns matching given answers). To personalize pattern recommendations, the user answers a set of questions linked to design problems, as presented in Subsection 6.2.1. An illustrative scheme of this process is shown in Figure 6.6.

Questions are organized in a tree structure, traversed by a conditional depth-first algorithm. The questionnaire starts with a high-level question (e.g., “I want to use design patterns in my application”) Depending on the user’s answers, each node of the tree is assigned a score: 1 for “Yes”, -1 for “No” and 0 for “‘I don’t know”, children of nodes with the negative score being skipped. The tool generates the recommendation once the questionnaire is filled up. Patterns constitute the leaf node of the tree. To compute the score S_q for each pattern, the

[7]https://protege.stanford.edu/
[8]https://www.w3.org/TR/sparql11-overview/

tool sums the score of every parent node and then normalizes the score using the length of the branch, accounting for branch length differences.

Next, three different algorithms were used to compute pattern rankings based on the scores S_q: `NoCitationsAndQS`, `WeightedCitationAndQS` and `UnWeightedCitationAndQS`. `NoCitationsAndQS` simply orders the patterns based on their score, while the other two also take into account an inferred number of citations. For each pattern, this number of citations is computed by summing the number of citations of all papers that proposes the pattern. As an example, if a pattern is proposed by two papers that respectively have 50 and 150 citations, the pattern is given a number of citations of 200. In `UnWeightedCitationAndQS`, the rank is obtained by multiplying S_q with the ratio between the number of citations of a pattern and the number of citations of the most-cited pattern. For `WeightedCitationAndQS`, instead of using the number of citations directly, its logarithm is used. The rationale for this is the extreme ranking skewness in favor of highly cited patterns in `UnWeightedCitationAndQS`.

Note that both `UnWeightedCitationAndQS` and `WeightedCitationAndQS` might discriminate negatively against newly proposed patterns that do not have many citations yet.

The user can select one of the three algorithms. When `NoCitationsAndQS` outputs a ranking only impacted by the answers, `WeightedCitationAndQS` and `UnWeightedCitationAndQS` also take into account citations, which serves as an indicator of the

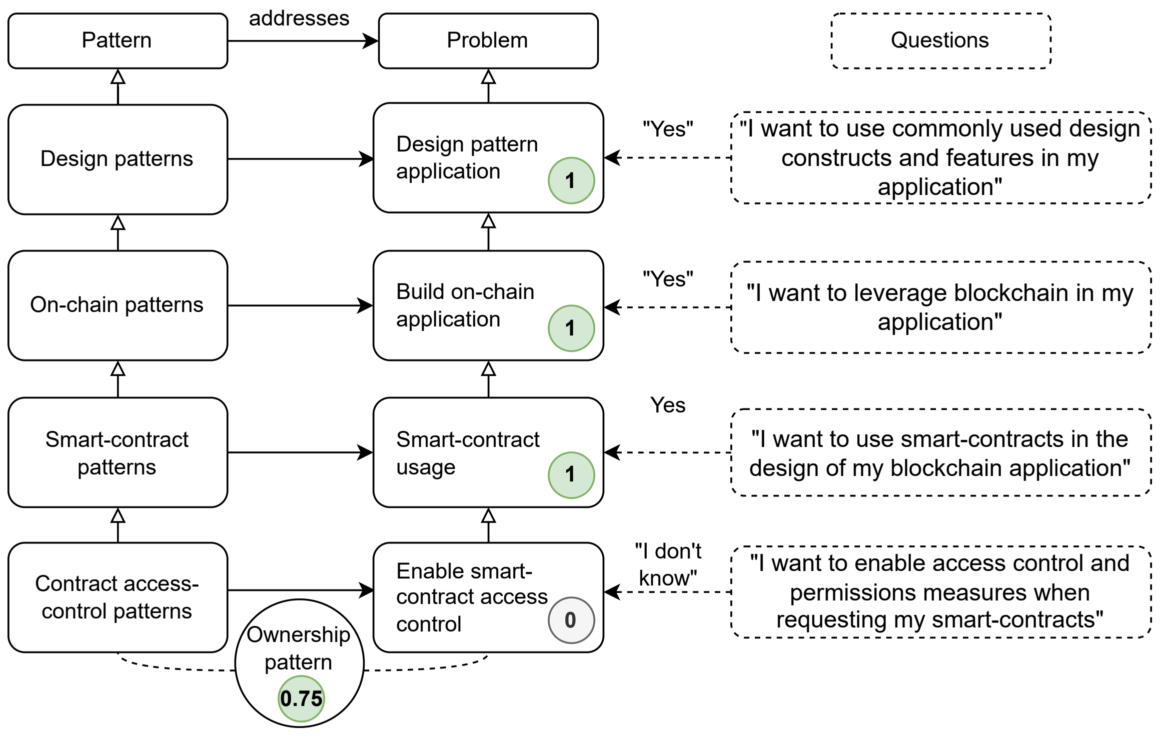

Pattern
addresses
Problem
Questions
Design patterns
Design pattern application
1
"Yes"
"I want to use commonly used design constructs and features in my application"
On-chain patterns
Build on-chain application
1
"Yes"
"I want to leverage blockchain in my application"
Smart-contract patterns
Smart-contract usage
1
Yes
"I want to use smart-contracts in the design of my blockchain application"
Contract access-control patterns
Enable smart-contract access control
0
"I don't know"
"I want to enable access control and permissions measures when requesting my smart-contracts"
Ownership pattern
0.75

Figure 6.6: Pattern scoring based on patterns/problem categories.

pattern adoption in the literature. Also, `UnWeightedCitationAndQS` tends to recommend highly cited patterns, that may be considered as more recognized by other researchers, whereas `NoCitationsAndQS` provides a ranking closer to the questionnaire's answers, to the risk of having newly proposed patterns that lack prior reuse. `WeightedCitationAndQS` is compromising between the two others, as it reduces the impact of citations without discarding them. In conclusion, the decision of using one algorithm instead of another is up to the user, depending on its goals: either using recognized patterns or newly proposed patterns that don't have this recognition yet. Nonetheless, the ranking differences between all of those algorithms are evaluated in Section 6.4.

6.3 Running Example

As in Chapter 4, the running example given in Chapter 2 is reused to guide to illustrate the usage of the blockchain-based software patterns recommendation part of BLADE. First, the running example requirements will be reused to answer the design questions given by the recommendation engine and needed to compute the recommendations. Then, the patterns returned at the top of the ranking will be discussed to see if they are applicable to the running example application, and to what degree.

6.3.1 Recommendation Engine Answers

To get a set of recommended patterns, the user has to answer a series of questions about design problems. In this goal, a statement about a design problem is displayed, and the user has to answer "Yes", "I don't know", or "No". By answering the question, the user can specify if a design problem affects the construction of its application. Table 6.2 lists the questions where the given answer was "Yes". In this list, some design problems are preeminent. First, the *On/off-chain data exchange* design problem: it notably englobes the impossibility of smart contracts to query external off-chain data by themselves. One common answer to this design problem is the *Oracle* design pattern, that pushes external data into smart contracts when needed. Then, the *Smart contract efficiency* and *Smart contract security* are two major design problems for the design of the *Carasau bread* traceability application. As mentioned in the running example paper, deploying and executing smart contracts can involve high costs, as it modifies the state of the blockchain. Also, the security of smart contracts is paramount as they might hold valuable assets. Thus, these two design problems should be addressed carefully. Finally, the *Enable smart contract access control* design problem is important in this context, as the access to the *Carasau bread* traceability application should be restricted to involved participants (e.g. baker, milling company, etc.).

6.3.2 Results

After answering the questions from the recommender engine, a set of recommended patterns has been returned. On the first page of the recommender results, 18 pattern proposals

Design problem	**Statement**
Design pattern application	I want to use commonly used design constructs and features in my application
Build on-chain application	I want to leverage blockchain in my application
Interacting with blockchain	I want to enable communication between my application and a blockchain
Domain-oriented application design	I want my blockchain application to handle features that serve the purpose of a specific domain
Manage on-chain data	I want to store and manage on-chain data in any format (encrypted or clear)
Smart contract usage	I want to use smart-contracts in the design of my blockchain application
Multi-domain feature application	I want to reuse multi-domain on-chain features in my blockchain application
On-chain storage	I want to use the best practices to store on-chain data in my blockchain application
Enable smart contract access control	I want to enable access control and permissions measures when requesting my smart-contracts
Smart contract efficiency	I want to improve the efficiency of my blockchain application by optimizing the costs of deploying and executing smart-contracts
Smart contract security	I want to improve the security of my smart contract against vulnerabilities and abuses
Blockchain-enhanced off-chain storage	I want to store data off-chain while taking profit from blockchain capabilities to trace and attest off-chain data
On/off-chain data exchange	I want to push up-to-date data on-chain or pull data and events from on-chain smart-contracts

Table 6.2: Relevant design problems for the Carasau bread application.

are displayed. In this list, some patterns are extremely relevant for the *Carasau bread* traceability application. For instance, the *Authorization pattern* proposal, ranked 4th, ensures that only allowed participants can fire restricted functions of a smart contract. This feature is mentioned as mandatory by the authors of the running example study: for instance, the milling company is the sole participant that can store floor production data on-chain. The *Oracle pattern* proposal, ranked in 9th position, is also very relevant. Indeed, the application includes the usage of IoT devices to submit production data on-chain, thus these devices act as oracles ingesting external data into the blockchain. Some security patterns are also very applicable to the *Carasau bread* traceability application, such as the *Check-Effects-Interaction pattern* proposal (12th) that consists in following a recommended functional code order to prevent reentrancy attacks[9], or the *Emergency stop pattern* proposal (14th) to freeze the execution of smart contracts in case of exceptional events. Finally, the *Tokenization pattern* proposal (3rd) can be mentioned: as the supply chain participants might transfer the property of real-world assets (e.g. floor, bread), tokenizing these assets is very relevant to allow more liquid and traceable exchange of assets between participants.

6.4 Validation

In this section, the artifact (that is the ontology, but also the platform as a whole) is validated to show that it allows answering to RQ4. This phase of artifact validation is two-fold. At first, the validity of the ontology is shown using both the ORSD mentioned above and the more general ontology evaluation methods outlined in the work of Raad and Cruz (Raad and Cruz, 2015).

Then, it is shown that this ontology, although valid, is also relevant to bring an answer to the RQ4 as its usage throughout the platform addresses the two requirements formulated at the beginning of the chapter. Summarized, these requirements are artifact usability and accuracy. For the former, it is expected that practitioners, including non-experts, are able to use the platform to explore the existing patterns and find adequate ones. The latter concerns the recommendation system: it is expected that patterns recommended by the platform are in adequation with the user requirements.

In this goal, these two aspects were evaluated separately in this section. For the usability aspect, a survey has been conducted with experts to assess the capability of using the explorer to understand the pattern proposals, and by extension to assess the relevancy of knowledge within the blockchain-based software pattern ontology. Within DSR, this survey is a form of expert opinion (Wieringa, 2014). From a given problem context, experts have to understand the artifact and use it to address the problem. Although the problem is artificial, it helps predict how the artifact, that is the platform, would be used to solve real-world problems.

[9] A reentrancy attack consists in using external smart contracts to maliciously re-execute a vulnerable function (for instance, multiple Ether transfers instead of one) using recursive calls.

For the accuracy aspect, a protocol has been designed to evaluate the recommendations produced by the recommender. Indeed, if the recommendation system is able to suggest adequate patterns, it illustrates the capability of using the ontology to find adequate patterns for specific requirements. Several papers were chosen from the literature, and each of them presents a blockchain application. Then, requirements were extracted for each application. Finally, the recommender was used for each paper following extracted requirements and the precision/recall of retrieved papers was assessed (Croft, Metzler, and Strohman, 2010). This is a form of Single-Case Mechanism Experiment (SCME), a DSR validation method already used in Chapter 4.

6.4.1 Ontology Validity

The first method of validation draws from both the ORSD mentioned above and the more general ontology evaluation methods outlined in the work of Raad and Cruz (Raad and Cruz, 2015). In particular, their task-based approach was followed, linking the evaluation of the ontology and the tool itself. The ability of the ontology to cover its requirements is demonstrated by, on the one hand, using SPARQL queries in isolation to answer the CQs, and, on the other, by showing its ability to be used as the central knowledge representation mechanism of the tool, through the validation methods to be covered below.

Some of the main evaluation criteria mentioned by Raad and Cruz are briefly touched (Raad and Cruz, 2015): **accuracy, completeness, clarity, and conciseness**, though difficult to demonstrate in an absolute sense, are nevertheless covered by the fact that the ontology has been constructed on the basis of an extensive literature review of the field, where care has been taken to isolate only the most relevant aspects; **adaptability** is a consequence of the use of the NeOn methodology and the use of SHACL shapes for automated verification of the ontology and the inferences made thereof, rendering the addition of new patterns to the ontology straightforward; **computational efficiency** is ensured by the compactness of the ontology and the avoidance of recomputing rule-based inferences for every query through pre-compilation of the inferred ontology triples; and, finally, **consistency** is ensured through the use of the Pellet OWL Reasoner and the aforementioned SHACL shapes for every main class in the knowledge base.

6.4.2 Ontology Relevancy

Usability

To validate the usability of the artifact, the researcher has surveyed a panel of 7 experts from different backgrounds (academia, industry) and positions (engineers, manager) as shown in Table 6.3.

A custom scenario has been designed on a blockchain use case. This scenario was short enough (2 standard pages) to ensure participants had the time to assimilate it in the survey within the allocated 30" timeframe. Organizers proposed 5 patterns $P^j_{H_1}$ $(0 < j < 5)$ for each expert n, the objective was to assess if the expert was able to find and understand the

Table 6.3: Panel Description.

ID	Role	Blockchain experience (in years)	Software design experience (in years)
E1	Lead tech	4	5+
E2	Ph.D. student	4	1
E3	Software engineer	4	5
E4	Blockchain engineer	4	5
E5	Ph.D. student	2	2
E6	Software engineer	1	2
E7	Ph.D. student	2	5+

patterns well enough to decide if they were applicable to the given scenario. This applicability of pattern j was rated by each participant n from 0 (non-applicable) to 4 (must-have) $R_n(P^j_{H_1})$. Then, the survey organizers performed the same exercise. As they worked on the construction of the knowledge base and the ontology, they know in detail the patterns presented in the tool and their related papers $\tilde{R}(P^j_{H_1})$.

Finally, participants' answers were compared to the organizers' own responses and a normalized score for each participant was calculated $S_n^{H_1}$, the average absolute difference between his score and the organizer's score.

Accuracy

The second validation step aims at evaluating the performance of the various recommender engines, especially those including citation metrics in the ontology. In this goal, the precision and the recall of the recommender engine at cutoff k were evaluated (Croft, Metzler, and Strohman, 2010). This method allows computing the precision and the recall regarding the k^{th} first items recommended by the engine, instead of all of the recommendations.

To compute these values, the following protocol was undertaken:

- Select a paper p from the literature (n=13), which propose a blockchain-based application
- From this paper, an expert manually extracts the requirements R_p and the emerging patterns $\hat{I}_p$, from the pattern list, which represents the golden standard of patterns for p
- Answer the questions of the recommender tool using only the requirements R_p, and retrieve a set of I_p recommended patterns, their position, and their score $S_{p,i}, i \in I_p$.
- Compute the precision at cutoff k, that is the ratio between the number of found emerging patterns in the recommended patterns $\hat{I}_p \in I_p$ and the cutoff number k.
- Compute the recall at cutoff k, that is the ratio between the number of found emerging patterns in the recommended patterns $\hat{I}_p \in I_p$ at cutoff k and the total number of found emerging patterns $\hat{I}_p$.

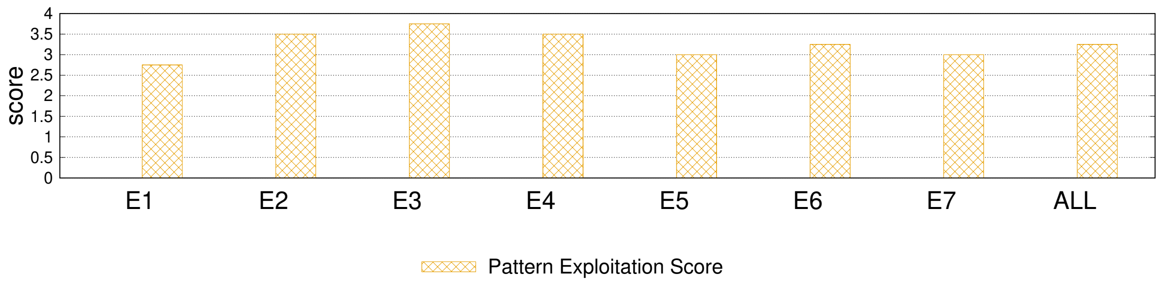


Figure 6.7: Panel Usecase Score $S_n^{H_i}$

6.4.3 Results and Analysis

Usability - Figure 6.7 shows the descriptive statistics for the score for each panel participant. The mean values for all the questions range from 2.75 to 3.75 with an average of $3.25/4$, which indicates that the participants have successfully navigated the solution space and provided adequate options on the relevance of the proposed pattern. Strong prior blockchain experience is not necessarily a good predictor for successfully judging patterns, since the most experienced participant has the lowest score. The most junior profiles having a score of 3, have used the tool effectively despite their lack of proficiency in blockchain application design.

The expert panel results show positive mean scores for all metrics. Thus, the usability of the platform (and by extension the ontology) can be considered satisfactory, despite having room for improvements, essentially in its perceived added value. The small sample size, should also prompt further large-scale surveys, including a pre-flight questionnaire to better quantify prior blockchain background for the respondents, and question its impact on the tool usability.

Accuracy - Figure 6.8 and 6.9 respectively show the precision and the recall at cutoff k for the three recommender systems considered. To interpret the results, it is required to select an adequate k w.r.t. the usage of the recommendation system. As the web platform displays the first 18 patterns on the first page when executing the recommendation system, a value of $k = 18$ has been chosen. Nonetheless, the selection of a suitable k is a difficult issue, discussed in Section 6.5.

Regarding the precision, the three algorithms are producing similar results except for a cutoff of $k < 20$, where the inclusion of citations increases the precision. For a cutoff of $k = 18$, the precision is 0.2, meaning that on average 20% of the first 18 recommended are relevant for the considered paper. Regarding the recall, the curves are similar for the three different algorithms, with a small advantage to the *NoCitationsAndQS* algorithm. For a cutoff of $k = 18$, on average 57% of the identified relevant patterns in the papers $\hat{I}_p$ are recommended. This number goes up to 80% for a cutoff of $k = 40$. By extension, it indicates that the majority of the most suitable patterns are ranked at the top by the recommender system.

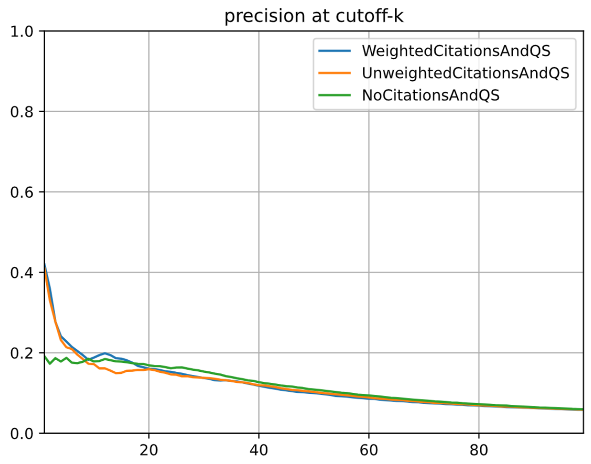


Figure 6.8: Average precision at cutoff-k.

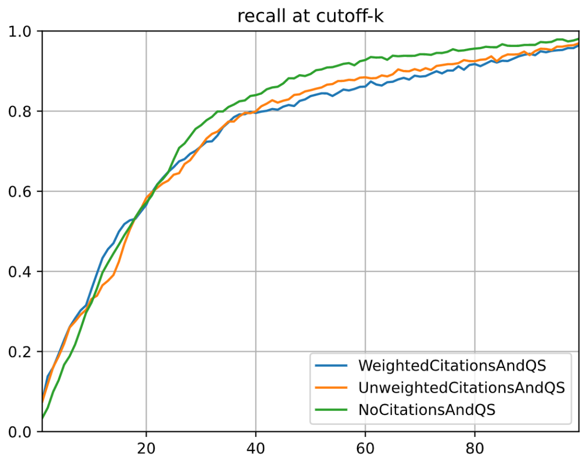


Figure 6.9: Average recall at cutoff-k.

6.5 Threats to Validity

Internal threat to validity: some aspects on the method used to validate $H2$ can be mentioned. Indeed, the selection of adequate patterns for a given paper has been carried by two researchers in 13 papers. Although these researchers are experts in blockchain technologies and decentralized applications, it still leaves some space for subjectivity. Several measures have been taken to limit the impact on the results: restricting the selection to the most important patterns, and comparing the results between the two researchers to evaluate possible discrepancies.

The method used to build the ontology can also be a threat to validity. Ontologies, by their very nature, have a degree of subjectivity; nevertheless, by using a structured methodology (NeOn), and building upon a peer-review literature review, this risk is mitigated, in conjunction with the formal specification of the ontology requirements and with the validation methodology outlined above.

Finally, the selection and retrieval of pattern proposals from the literature, that constitutes the core of the ontology, is also subject to be a threat to validity. However, it is mitigated by the strict method followed to perform the literature review (SLR). More details about possible threats and mitigations are given in another study (Six, Herbaut, and Salinesi, 2022).

External threat to validity: the main threat is the generalizability of the ontology. Even if the main purpose of the ontology was its reusability in a tool, careful attention has been made to maximize the ontology reusability. Part of the blockchain-based software pattern ontology is inspired by the Design Pattern Intent ontology (Kampffmeyer and Zschaler, 2007), to bind design patterns (by extension, software patterns) with blockchain design problems. Patterns are also expressed using a shortened pattern format, similar to the GoF pattern format or the Alexandrian form (Tešanovic, 2005). Future works will refine those patterns to fully comply with one of those two formats. Finally, the ontology has been designed with extensibility in mind. For example, the blockchain class can easily be a connection point between this ontology and other blockchain-related ontologies, such as (De Kruijff and Weigand, 2017), a blockchain domain ontology.

Construction threats to validity: the tool built to leverage the ontology can be mentioned. This tool eases the usage of the ontology by non-experts but forces users to indirectly leverage the ontology, as intended by the tool. Nevertheless, the tool was built with regard to the validation of H_1 and H_2. Where the *Explorer* helps to navigate freely in the ontology by displaying all patterns and links, the *Recommender* allows fetching across design problems and their related questions to generate a recommendation.

Conclusion threats to validity: the difficulty to conclude on the recommendation engine accuracy w.r.t. adequate cutoffs k can be mentioned, as its selection mainly depends on the user behavior. Indeed, some users might only read the first 5 patterns, whereas others might fetch all of the recommendations. In the web platform, the first page of the recommender results displays 18 patterns; thus this number might be a good candidate for k. Nonetheless, this number might change depending on the usage of the recommendation engine and the ontology in the future.

6.6 Related Works

The literature shows that the idea of using ontologies to describe software patterns has already been explored. Kampffmeyer and Zschaler propose an ontology (Kampffmeyer and Zschaler, 2007) derived from GoF (Gang-of-Four) design patterns (Gamma et al., 1995)[10] Each pattern is linked to a set of design problems it solves, along with a tool to help practitioners select patterns without having to write semantic queries. However, their ontology does not bring out any dependency link between the patterns themselves. The contribution reuses the concept of problem ontology and extends it, as shown in Section 6.2.

Another ontology for software patterns is proposed by Girardi and Lindoso (Girardi and Lindoso, 2006). This ontology encompasses not only design patterns but also architectural patterns and idioms. A pattern is described using different attributes (such as *Problem*, *Context*, *Solution*, etc.), and can be linked to other patterns through a pattern system and specific relations (e.g., require, use).

Henninger and Ashokkumar propose a similar metamodel for software patterns (Henninger and Ashokkumar, 2006). Some differences can be mentioned, such as the possibility to specify that two patterns conflict with each other and cannot be applied at the same time, or the *seeAlso* relationship to indicate other patterns related to a specific pattern. In addition, Pavlic et al. propose a design pattern repository taking the form of an ontology (Pavlic, Hericko, and Podgorelec, 2008). The contribution enlights tedious knowledge management and

[10]The authors of this book, Gamma et al., are often referred to as the Gang-of-Four.

sharing with traditional pattern collections and argues for a structured ontology format. The proposed ontology groups patterns into pattern containers, where one pattern can belong to many containers. Patterns can also be linked to a set of questions and answers, elicited from expert knowledge, through an *answer relevance* attribute. It indicates how relevant a pattern is in addressing a specific question. The contribution follows a similar path to that taken by the blockchain-based software pattern ontology by structuring a set of patterns of a specific domain, in this case, blockchain-based patterns.

Some ontologies have been proposed for modeling the blockchain domain, such as that proposed by De Kruijff and Weigand (De Kruijff and Weigand, 2017), that of Ugarte-Rojas and Chullo-Llave (Hector and Boris, 2020), and that of Glaser (Glaser, 2017) that models the technology itself and its components. Another work by Seebacher and Maleshkova (Seebacher and Maleshkova, 2018) focuses on modeling the characteristics of blockchains within corporate networks and their use.

The ontology proposed in this chapter complements the state-of-the-art of existing ontologies on blockchain technologies. This may allow connecting the blockchain-based software patterns ontology to these other ontologies in future works. For instance, the aforementioned ontology that contains blockchain components (Glaser, 2017) may be mapped to the *Blockchain* class in the blockchain-based software patterns ontology. The blockchain-based software patterns ontology also contributes to the state-of-the-art of software patterns ontologies, by extending existing ontologies to support new concepts such as blockchain-based software patterns, the pattern/variant/proposal distinction, and citations.

6.7 Conclusion and Future Work

This chapter proposes a blockchain-based software pattern ontology to store, classify, and reason about blockchain-based patterns. The ontology has been built over previous results obtained by performing a systematic literature review of the state-of-the-art of blockchain-based patterns. It is composed of proposals that are patterns formalized in the context of an academic paper. These patterns have been stored in the blockchain-based software pattern ontology. They were created out of 160 proposals found in the literature, showing that about a quarter of patterns in literature are redundant. Also, those patterns have been classified using a taxonomy reused from the systematic literature review mentioned above. This ontology is leveraged in the second part of BLADE: practitioners can explore the ontology and its collection of patterns, but also use a recommender to get adequate patterns fulfilling their needs. This tool is also meant to be extendable following ontology evolution and support future works. The ontology can also be leveraged as standalone, using SPARQL queries.

The ability of the artifact to answer RQ4 while also satisfying the two requirements (usability and accuracy) has been validated in two separate parts. During the first part, a survey was conducted among 7 practitioners in the blockchain software engineering field. Participants were asked to rate the applicability of a list of patterns for a specific scenario, both proposed in the context of the survey. Results showed that participants were able to successfully perform this task using the tool. In the second part, the recommendation system were evaluated by manually picking suitable patterns for given use cases, then using the recommender system to assess the ranking of manually picked patterns compared to the others. As a result, the majority of manually picked patterns are ranked at the top by the recommender system.

The web platform and the ontology presented in this chapter constitute the second part of BLADE, the recommendation engine for the design of blockchain applications. It further extends the recommendation of a blockchain platform by adding relevant blockchain-based software patterns to it. Using BLADE, the practitioner is guided into complex aspects of the design of a blockchain application. In the next chapter, the last artifact of the framework, BANCO, is introduced. Positioned next after BLADE in the framework, BANCO is able to produce a blockchain-based application based on BLADE's recommendations.

Chapter 7

Generating a Blockchain-Based Application Reusing Previous Recommendations

Key takeaways

- A feature model was designed to represent the variability between products of the same family, that are blockchain-based traceability products (i.e. applications).
- A platform was designed to allow the practitioner to configure its desired product, reuse the feature model, then generate it using template-based code generation.
- BANCO is the third artifact of the framework with regards to the DSR approach. It addresses the technical research question RQ5.

As solutions to ease the design of a blockchain application was investigated in Chapter 4, 5 and 6 through the design of BLADE, this chapter addresses the implementation phase of the software engineering process. This is a tedious task in the blockchain field, where there is a lack of blockchain education among practitioners (Cunha, Soja, and Themistocleous, 2021a). They might struggle with the complexity of designing specific blockchain features, such as tokens or oracles. Reusing existing code is one solution to solve this issue (so-called "clone-and-own") and is a common practice in the blockchain field (Chen et al., 2021). Some of these solutions have even been formalized as design patterns to ease their reuse. For instance, as smart contracts cannot query data from outside the blockchain, developers have to apply the *Oracle pattern* (Xu et al., 2018). An oracle includes two components: a smart contract capable of emitting an event when new data is required, and an off-chain service listening to these events to inject fresh data when needed.

This reuse of existing code is a first step in addressing the difficulties of implementing a blockchain application, but it could be further systematized throughout code generation. In this chapter, the following technical research question mentioned is addressed (**RQ5**): *How to design a blockchain application generation platform that reduces the cost and difficulty* *of implementing blockchain applications as practitioners can configure then generate blockchain applications?*

From this research question, it is notably expected that is a reduction of blockchain application development costs when using SPLs compared to traditional software engineering.

To tackle this question, multiple approaches can be envisioned, such as Model-Driven Engineering (MDE) and Software Product Line Engineering (SPLE). MDE encompasses several aspects of software engineering assisted with models, such as domain-specific modeling languages and transformation engines and generators (Schmidt, 2006). Several approaches for blockchain software engineering based on MDE have already been proposed in the literature, such as reusing BPMNs (Lòpez-Pintado et al., 2019) or Petri Nets (Zupan et al., 2020). Yet, one MDE approach for software blockchain engineering remains unexplored: the combination of SPLE and blockchain. A comparison between using Software Product Lines (SPLs) and other methods to generate blockchain applications is given in Subsection 7.6.4.

SPLE is based on the reuse of various software artifacts (e.g., requirements, models, code, and tests) designed for this purpose, to create (software) products that have common elements (Pohl, Böckle, and Van Der Linden, 2005). By leveraging a SPL approach, developers could easily configure and generate blockchain applications based on efficient and extensively tested components and patterns. As a result, new research sub-questions should be investigated to assess the relevancy of applying SPLs to blockchain:

- **RQ5.1** - Is SPLE applicable to the blockchain field?
- **RQ5.2** - Do blockchain applications created following a standard software development engineering differs from applications derivated from a software product line?

To address these questions, a software product line for blockchain applications has been created from scratch. It results in a web platform that allows the configuration and the generation of a blockchain product. The generation is performed by assembling code templates (e.g., smart contracts), based on the configuration given by the user. A feature model guides the configuration process, by describing existing features and their constraints with others. This feature model has been designed by extracting features found in studies of a specific domain, that is blockchain-based traceability. The capacity to generalize the approach was evaluated by reproducing existing blockchain-based traceability applications using exclusively the web platform. Also, the source code of the web platform and the templates is available on Github[1]

The chapter is organized as follows: Section 7.6 discusses related works on applying software product lines for nascent technologies. Section 7.1 and 7.2 introduce the platform, first by describing the construction of the feature model and then its usage through the web platform. The running example introduced in Chapter 2 is then reused in Section 4.4 to illustrate the functioning of the web platform. The ability of the artifact to constitute an answer to RQ5 is validated in Section 7.4, and Section 7.5 discusses those results along with lessons learned

[1]`https://github.com/harmonica-project/BANCO`

in the implementation of the software product line as well as possible research challenges. Finally, Section 7.7 concludes the chapter.

7.1 Feature Model Design

The first step in the software product line engineering process is the domain analysis (Czarnecki and Ulrich, 2000), where the result is often a feature model. A feature model is a widely adopted notation to describe allowed variability between products of the same family, and feature dependencies (Schobbens et al., 2007). The main advantage of using a feature model is the increased ease of reusing existing features, as it models accurate cartography of them that can be shared between stakeholders.

7.1.1 Construction Method

The feature model has been created using the standard feature model and FeatureIDE, an open-source framework [2]. It is composed of different notation elements (Thüm et al., 2014). It allows the definition of concrete/abstract features, that can be optional or mandatory. It also supports *and*- and *xor*- decomposition of features, to either select multiple subfeatures (but at least one) among a given set linked to a feature, or select only one subfeature in the selection. Finally, feature models include constraints between features, preventing for instance the selection of two conflicting features. The standard feature model has been chosen as it satisfies the researcher's needs for the construction of an on-chain traceability feature model.

The construction of a feature model requires extensive knowledge of its associated domain. In this study, this knowledge has been extracted from 5 different works that propose on-chain traceability solutions, called foundational set, shown in Table 7.1 (Baralla et al., 2021; Wei, 2020; Caro et al., 2018; Figorilli et al., 2018; Kuhn et al., 2021).

These papers were selected as they propose blockchain-based traceability applications for various domains, and as the features composing the application were easily identifiable to be extracted and then included in a feature model. Also, each of these papers proposes a concrete implementation of the Ethereum blockchain. Indeed, the Solidity language was chosen to implement the templates and reused to generate the blockchain products.

From these papers, the features that were at least present twice (2-of-5) were considered for the feature model. Other features were discarded, as they were too specific to the proposed application. The remaining collection of features was then generalized to suit any domain implementing a blockchain-based traceability application. For instance, the identified features `WoodBatchTraceability` and `SparePartsTraceability` were merged and generalized as `AssetTracking`. In some cases, they have been refined manually by adding subfeatures (e.g., adding CRUD methods to manage application participants). This results in a feature model, presented in the following subsections. Note that this feature

[2] https://featureide.github.io/

Table 7.1: Blockchain traceability research used to design and test the feature model.

Ref.	Authors	Traced item	N^b of features	Part of
(Baralla et al., 2021)	Baralla et al.	Food	12	Foundational set
(Caro et al., 2018)	Caro et al.	Food	14	
(Figorilli et al., 2018)	Figorilli et al.	Wood	15	
(Kuhn et al., 2021)	Kuhn et al.	Manufactured items	12	
(Wei, 2020)	Wei et al.	Goods	3	
(Hasan et al., 2020)	Hasan et al.	Spare parts	N/A	Test set
(Casino et al., 2021)	Casino et al.	Food	N/A	

model is not meant to be a complete representation of existing on-chain traceability features, but provide the most salient features of an on-chain traceability solution. A complete analysis through a systematic literature review is left for future work.

The resulting feature model is composed of 53 different features, distributed across three different main features:

- `SmartContracts` feature - gathers all features included in smart contracts. The selection of the subfeatures of `SmartContracts` represents the configuration of the on-chain part of the application.
- `Storage` feature - regroups the features that address how and where traceability data is stored.
- `Frontend` feature - represents the off-chain part of the application.

Each of these features represents a specific aspect of a blockchain application. They are described along with their subfeatures in detail in the following subsections.

7.1.2 Smart Contracts Feature

The first feature of the model is the `SmartContracts` feature (Figure 7.1[3]). It represents the on-chain part of the traceability application, composed of a collection of smart contract instances. This part of the feature model also involves three different constraints, expressed in Table 7.2.

This feature is divided into two subfeatures: the management of participants, and the traceability methods used. The `Participants` feature distinguish two important aspects: individuals, that will interact with the traceability smart contracts and are identified by a public address, and roles, that can be assigned to individuals. Using roles is optional in the model,

[3]For the sake of readability, CRUD methods are folded. For each feature in the figures introducing the feature model, a label with a number indicates the number of related CRUD subfeatures.

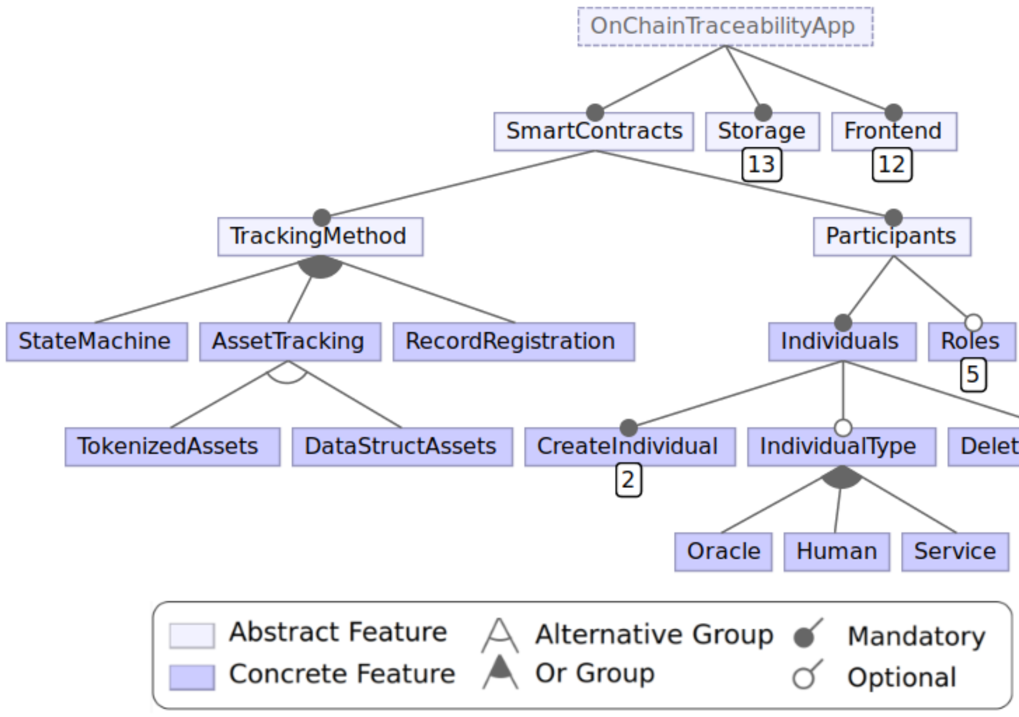


Figure 7.1: Focused view of the SmartContract FM

as access control can be done using only public addresses (e.g., only a given set of individuals can add records in a given record collection). However, they can be useful to regroup individuals based on their role in the process (e.g., in a supply chain, identify the suppliers, carriers, and buyers). Besides roles, individuals can be classified through types: they can either be human, service, or oracle (from the *Oracle pattern* (Xu et al., 2018)).

Three traceability methods can be selected in conjunction or standalone in the model.

The `StateMachine` subfeature allows tracking state changes on-chain. A state machine is defined by a set of state variables and commands, that transform its state (Schneider, 1990). For each transition, it is possible to define a set of individuals and roles that are entitled to trigger the transition between two states. The current implementation behind the `StateMachine` subfeature only allows the creation of linear state machines. The term linear is used here to qualify state machines where each state has only one preceding and one following state, and where it is impossible to make a transition more than one time in a specific state. Nonetheless, it will be improved in future works by handling all of the state machine features (e.g., multiple transitions from one state, etc.).

`AssetTracking` consists of storing data on real-world assets, such as a batch of products. Each asset has a set of owners and a set of entitled individuals and roles that can modify it. A state machine can be attached to an asset: for instance, a batch can be stored, shipped, or delivered. Assets can either simply be stored as a simple data structure, or as tokens (as proposed by Kuhn et al. (Kuhn et al., 2021)). Storing assets as tokens allow their transfer between individuals. For instance, a batch can be sent from the supplier to the carrier. Tokenization is a common blockchain-based design pattern (Xu et al., 2018), standardized for

Table 7.2: Smart contracts and frontend feature constraints.

Range	Operator	Target
DeleteIndividualByRole	Implies	Roles
IndividualsSetup	If and only if	CreateIndividualAtSetup
RolesSetup	If and only if	CreateRoleAtSetup

many blockchains such as the ERC721 standard for Ethereum[4].

Finally, `RecordCollections` allows bulk storage of records in arrays. These records are stored as described in the `Storage` feature. As with others, a collection has a set of entitled individuals and roles that can append new records.

7.1.3 Feature Storage

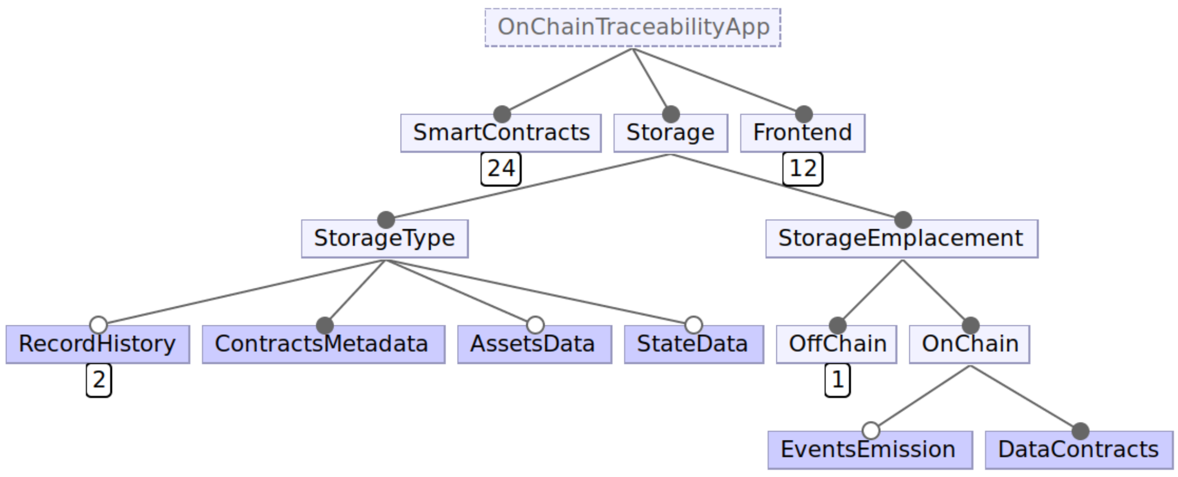


Figure 7.2: Focused view of the Storage FM.

The second feature of this model is `Storage` (Figure 7.2), divided into two aspects.

For the first aspect, data can be stored in multiple formats. In some applications, it is a suite of timestamped records. These records can either be data on a specific event that occurred in the traceability process or regularly pushed traceability data (e.g., real-time temperature). The feature model further refines the subfeature `RecordHistory` in two subfeatures: `StructuredRecords` and `HashedRecords`. Where the first one can contain any type of data, the second one is a timestamped hash of a `StructuredRecord`. These records can be used when it is not desirable to store data on-chain for confidentiality reasons or storage limitations. In this case, each structured record is stored off-chain in a database, then hashed and stored on-chain as it. This storage strategy is a common blockchain-based design pattern named *Off-chain data storage Pattern* (Xu et al., 2018). Traceability data can also be stored as objects representing `AssetsData`, or as a set of states and the transition history between them when using a `StateMachine`. These dependencies between storage type

[4] `https://eips.ethereum.org/EIPS/eip-721`

Table 7.3: Storage feature constraints.

Range	Operator	Target
RecordRegistration	If and only if	RecordHistory
RecordHistory	If and only if	RecordsCollectionSetup
AssetTracking	If and only if	AssetsData
AssetsData	If and only if	AssetsSetup
StateMachine	If and only if	StateMachineData
StateMachineData	If and only if	StateMachineSetup

and traceability methods imply a set of constraints (Table 7.3). Indeed, the selection of a specific traceability method should automatically select the related storage type and setup form feature. Finally, a mandatory feature named `ContractMetadata` is in charge of storing the address of every smart contract deployed for a traceability process. This feature includes the usage of the *Factory Pattern* (Xu et al., 2018), as the factory deploys and keeps track of existing contract instances.

Regarding the storage emplacement, data can either be stored on-chain or off-chain. On-chain data is stored in smart contracts following the *Data Contract Pattern*, that separates data storage from logic contracts (e.g., controllers) (Xu et al., 2018). Events can also be emitted when something occurs (e.g., storing a new record, firing a transition, etc.). Traceability data can also be stored off-chain, in databases. The `Database` feature is mandatory in the feature model, as smart contract metadata have to be at least stored off-chain to allow retrieving the address of existing contracts. However, traceability data can either be stored off-chain, on-chain, or both.

7.1.4 Frontend Feature

The last feature is `Frontend` (Figure 7.3). The frontend application can be used to set up the traceability process through the `DeploymentView` feature.

Indeed, individuals, roles, and traceability assets/states/collections are not defined statically in the code but dynamically as parameters passed when instantiating the smart contracts. Thus, the user has to specify these data in order to set up the traceability process.

The `ParticipantSetup` helps to define individuals and roles that will be granted the right to create, modify or own traceability items. For instance, the supplier of a spare part manufactory shall be able to create spare parts in the traceability contract to represent his real-life assets. Where the setup of individuals is mandatory (the application must have at least one administrator), the setup of roles before deployment can be optional, as they can be created later.

For the `TraceabilitySetup`, three subfeatures can be selected: `RecordsCollec-tionsSetup`, `AssetSetup`, and `StateMachineSetup`. They respectively implement forms to define the collections of records, the assets, and the state machines. Each form also

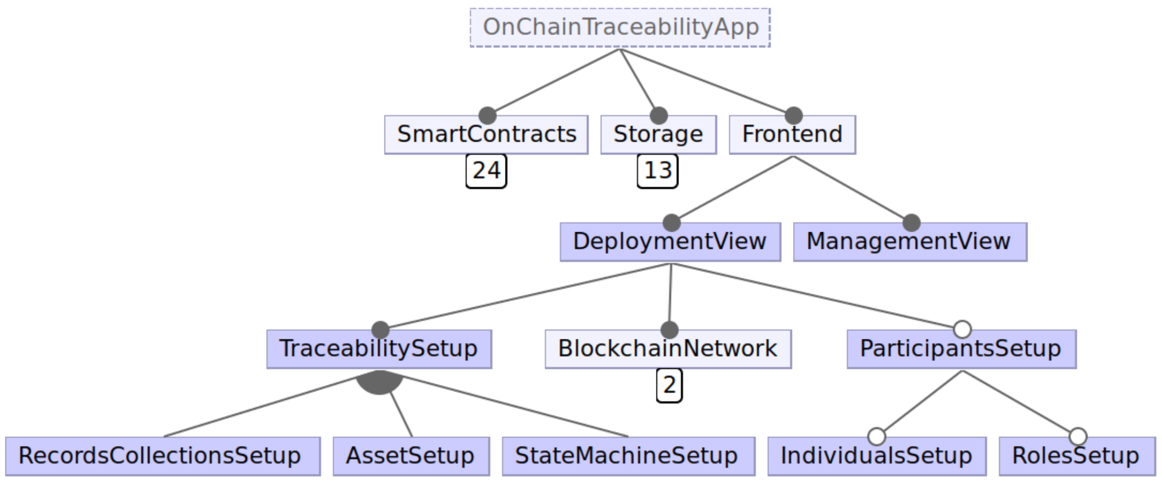


Figure 7.3: Frontend feature model.

allows binding participants to traceability items, and roles if the corresponding subfeature has been selected.

One feature that is `BlockchainNetwork`, specifies the targeted network: in this model, either the Ethereum testnet (for testing purposes, free to use) or mainnet (in production). Users can then interact with deployed smart contracts through the application to leverage the aforementioned features.

7.2 BANCO construction

A feature model usually guides the selection of features by the user when composing products. However, this task is burdensome when performed manually. In this work, a web platform under the name of Blockchain ApplicatioN Configurator (BANCO) has been built, using the concept of SPL configurator to tackle this issue. A SPL configurator is a class of technology that enables software mass customization, based on automated product instantiation rather than manual application engineering (Krueger, 2009). A configurator is able to reuse existing core assets (e.g. software artifacts) to allow the composition of software products. It leverages a variability model, that expresses features commonalities, and variability, to guide the user in the range of possible combinations.

According to Krueger, this automation has several advantages:

- *Reuse* - all software exists within a consolidated collection of core assets, thus it is possible to refactor existing software artifacts for reuse purposes.
- *Scalability* - as the development is focused on core assets (domain engineering) rather than application engineering, the organizational structure behind is more efficient.
- *Upgradability* - it is possible to re-instantiate existing products when a core asset is changed.

Using a configurator, there is very little overhead to adding a new product to a product line, as the developers simply have to focus on missing core assets rather than developing a complete application (Krueger, 2009).

Figure 7.4 shows an overview of the BANCO configurator. The process of configuration is the following:

1. Practitioners (e.g. software engineers/architects) can use the web platform exposed by BANCO to configure the product, according to their needs.
2. The configurator then verifies the configuration provided to assess its completeness and its validity[5].
3. The product is created by the generator, reusing the configuration created by the user.

Also, two types of assets are provided to the configurator: core assets, that are the common and varying software assets such as requirements, code, tests, and design decisions, and the feature model defined in Section 7.1. The following subsections respectively discuss the construction of the configurator and the generator.

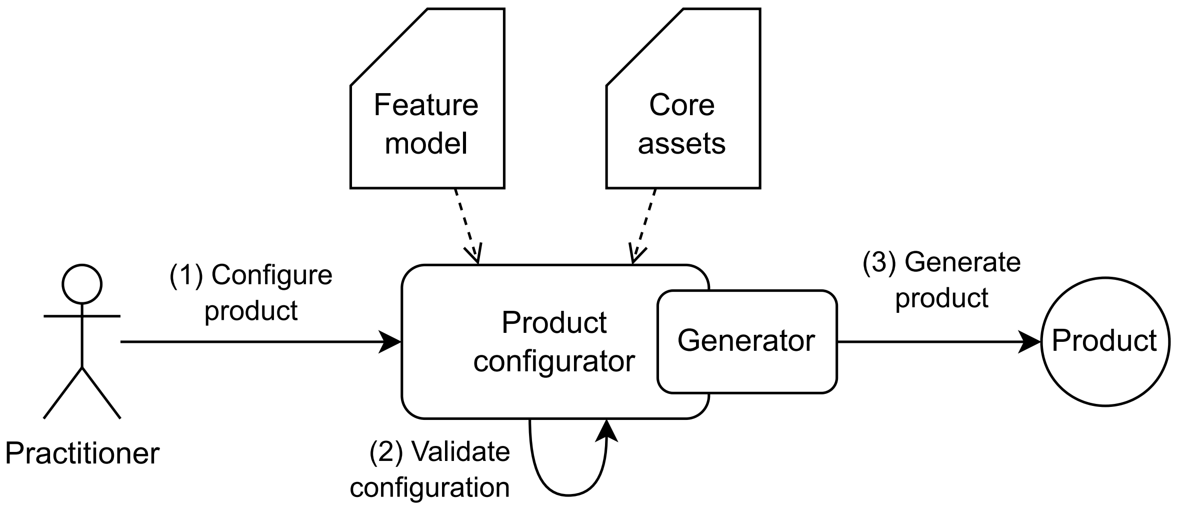


Figure 7.4: Overview of BANCO.

7.2.1 Product Configuration

The configurator is the first part of BANCO. In this work, the SPL configurator is implemented as a web platform. This platform displays a configuration panel and a feature model visualizer that were adapted from Kuiter et al. work (Kuiter et al., 2018). The configuration panel displays a tree of features, generated using as input the on-chain traceability feature model. Some of the features are already pre-selected, as the feature model contains mandatory features. For the rest, the user can either select the inclusion or the exclusion of a feature by selecting the corresponding box. Each selection will trigger the constraint engine, which

[5]Completeness and validity are defined in Subsection 7.2.1.

will automatically include or exclude features based on the constraints formulated along the feature model.

The configuration also has two different states: its validity and its completeness. The first one indicates if a configuration is valid, i.e., if constraints are satisfied. As the configurator prevents selecting two conflictual features, the user cannot make a selection that results in an invalid configuration. The second one indicates if the configuration is complete, i.e., all the features are either selected or deselected. Therefore, the user can rely on these indicators to know if the configuration step is complete or not.

The feature model visualizer helps the user visualize the on-chain traceability domain and its available features. This visualizer also guides the user during the configuration by changing the color of selected or deselected features respectively in green or red. It allows to quickly visualize the impact of selecting one feature on others, and the features that remain to be selected.

7.2.2 Product Generation

From a valid and complete product configuration, the web platform is capable of generating a working product. A generator has been implemented to perform this operation, based on Template-Based Code Generation (TBCG). TBCG is a technique from the model-driven engineering field that consists of generating code based on templates. A template is constituted of static text with embedded dynamic portions that are evaluated by a template engine to output functioning code (Jörges, 2013). Such evaluation also requires providing data in order to fill the dynamic portions of the text.

In this work, the task of evaluating templates is performed by Mustache, a logic-less web template system[6]. Mustache is capable of evaluating any provided text input that contains a series of tags (i.e., dynamic portions), providing it with a suitable JSON object to compute the tags. This template system handles features such as optional code blocks, text completion, and loops. It is also possible to modify the default opening and closing tags of Mustache to adapt them to the language used in the templates.

From the configuration made by the user on the web platform, a JSON object is generated containing all of its choices. This object will be ingested by Mustache to process the templates. For the on-chain part, the smart contract templates are written in Solidity[7], a language to implement Ethereum smart contracts. The default Mustache tag has been modified from the default notation ({{ }}) to the block comment symbols used in Solidity (/* */), to allow writing Mustache instructions in Solidity comments. This allows for developing and testing smart contract templates without raising any errors because of Mustache notation. The approach taken to develop the templates is based on subtractive code generation: all of the features are included in the templates, and Mustache removes or modifies them according to the configuration. For instance, the following code block (Listing 7.1) will be conditionally

[6]`https://mustache.github.io/`
[7]`https://docs.soliditylang.org/en/v0.8.13/`

rendered in the final product only if the feature *AddRoleDynamically* has been selected by the user.

```
/* #AddRoleDynamically */
function addRoleToParticipant(
    address _participant,
    string memory _roleName
)
    public
    verifyRolePermission(_roleName)
{
    participantsContract.addRoleToParticipant(_participant, _roleName)
        ;
}
/* /AddRoleDynamically */
```

Listing 7.1: Solidity template code sample

Figure 7.5 describes the chosen architecture for the on-chain part of the application. At first, the user deploys a single factory contract (1). A factory contract, designed following the *Factory pattern* (Xu et al., 2018), is in charge of creating other contract instances at instantiation (e.g., participant contracts) (2). The factory contract also acts as a contract registry, as it stores the addresses of created contracts. Once this deployment is completed, the user can interact with controllers (3). Each on-chain feature (e.g., participant management, state machine, etc.) is implemented as a pair of two contracts: a data contract in charge of holding data collections and getters/setters to manipulate them, and a controller contract to interact with data contracts. These controllers also enforce specific conditions to modify the data (e.g., verifying asset ownership before updating it). The separation between logic and data is a common blockchain design pattern that also increases upgradeability: controllers can be changed without having to migrate the data from one contract to another (Xu et al., 2018). Otherwise, this operation would be very expensive in terms of storage and costs and tedious to perform.

The different features defined in the feature model can be traced to this architecture. One controller/data smart contract pair is in charge of participants and roles, whereas three controllers/data contract smart pairs are responsible for the different traceability methods defined in the feature model. As the user can select one to three different traceability methods, it is possible that some of these contracts are not present in the final product. However, the participant data/controller smart contract pair will always be present, although the role features might not depend on the configuration.

For the off-chain part, a web application has been developed, where pages are conditionally included in the final product depending on the configuration. For instance, if the user does not select the `Roles` feature, the web page to configure or to allocate roles to users will not be included in the generated application.

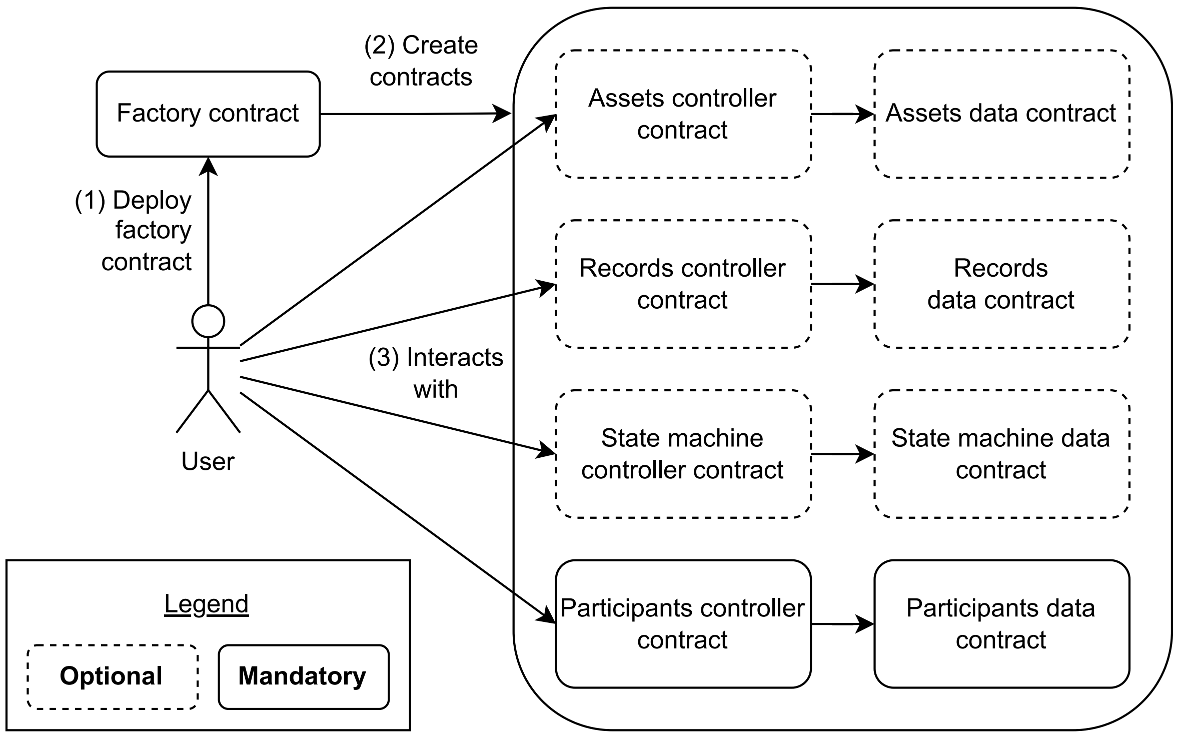

(2) Create contracts
Factory contract
(1) Deploy factory contract
User
(3) Interacts with
Assets controller contract
Assets data contract
Records controller contract
Records data contract
State machine controller contract
State machine data contract
Participants controller contract
Participants data contract
Legend
Optional
Mandatory

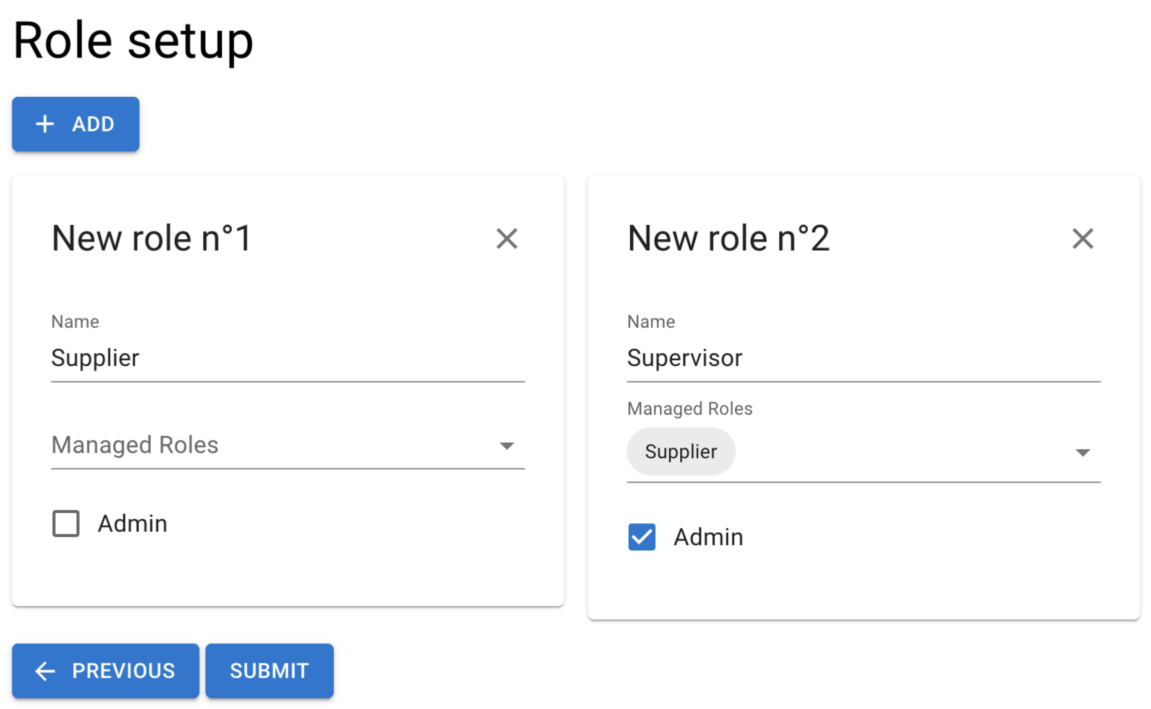
Figure 7.5: Smart contract architecture.

7.2.3 Product Deployment

The web application generated by BANCO acts as the frontend of the blockchain traceability application. However, the smart contracts have to also be deployed on the target blockchain (here, Ethereum), as they act as the backend of the application. In this goal, the user can leverage the deployment view, issued from the `DeploymentView` feature, to deploy the application providing a set of parameters. These parameters are provided using several forms, that correspond to the various aspects of the blockchain traceability application.

For instance, the available roles in the blockchain traceability application can be specified using the role setup panel (issued from the `RoleSetup` feature). Figure 7.6 shows a filled role form. Two roles are defined in the application: the supplier, and the supervisor. In this example, the supervisor is an admin of the blockchain traceability application, thus capable of creating new individuals and roles in the application. Also, the supervisor is in charge of suppliers: any supervisor can attach or remove the supplier role to an individual.

At deployment, the values filled in the form are translated to contract parameters. Figure 7.2 shows the result of this translation for the role form. These values are then passed as parameters to the constructor of the role smart contracts. In this way, roles are dynamically created at deployment instead of being directly written in the code. This allows SPL-issued products to be more flexible with regard to possible use-cases of a blockchain traceability application.


Role setup

New role n°1

Name: Supplier

Managed Roles

☐ Admin

New role n°2

Name: Supervisor

Managed Roles: Supplier

☑ Admin

Figure 7.6: Role form example.

```
{
  "roles": [
    {
      "name": "Supplier",
      "isAdmin": false,
      "managedRoles": []
    },
    {
      "name": "Supervisor",
      "isAdmin": true,
      "managedRoles": [
        "Supplier"
      ]
    }
  ]
  ...
}
```
Listing 7.2: Participants contract parameters example.

7.3 Running Example

As in Chapter 4 and 6, the running example given in Chapter 2 is reused to guide to illustrate the usage of BANCO, the configurator and code generator introduced in this chapter. Therefore, this section presents the configuration of the product corresponding to the *Carasau bread* traceability application.

Listing 7.3 displays the entire configuration of the product corresponding to the *Carasau bread* traceability application in an XML format. For each feature, it is indicated if the feature is *selected* or *unselected* by two attributes: *manual*, and *automatic*. The presence of one attribute or another depends if the selection was made by the user or automatically done due to existing constraints (Table 7.4).

Table 7.4: Signification of configuration XML file attributes.

	Selected	**Unselected**
Manual	The user manually selected the feature	The user manually unselected the feature
Automatic	The feature was automatically selected due to constraints	The feature was automatically unselected due to constraints

The most important part of the configuration is the tracking method. In the running example study, there is a need to track batches from one node (e.g. supply-chain participant) to another. Also, some documents and information might be recorded by participants or IoT sensors. Thus, two of the three tracking features have been selected: *AssetTracking* and *RecordRegistration*. As the *AssetTracking* feature requires additional configuration to specify how assets are tracked, the *DataStructAssets* subfeature has been selected. It allows to store additional information regarding an asset, as they are stored as a Solidity struct. Nonetheless, tokenized assets could also be used in the future.

Regarding participants, the *Roles* feature has been selected, as well as the capability to add/remove participants and roles at any moment. Indeed, the requirements specify the need for an authority shall be able to manage supply-chain participants, from their access to the application to their rights towards existing assets. The *IndividualType* feature has also been selected, as well as its subfeatures *Human* and *Oracle*. As IoT sensors might be used to push information on-chain, the distinction is important.

For the storage configuration, most of the options have been prefilled as they are constrained in their selection depending on the chosen tracking methods. As the *RecordHistory* feature requires the selection of subfeatures (e.g. what type of records are stored), the *HashRecords* subfeature has been chosen along with the *StructuredRecords*. Indeed, both records are important: where the former allows storing hashes of data stored on IPFS, as specified in the running example study, the latter allows storing detailed records directly in the smart contract state. The *EventsEmission* feature has also been selected, as they are required to inform participants about updates in the traceability process. They are also used in the study's proposed application.

As for the storage configuration, the frontend features were automatically preselected using defined constraints with other features. The only unselected feature left is the *Blockchain-Network* feature. As it defines the blockchain network used by the generated product, it is a mandatory feature that leads to two choices (subfeatures): *EthereumMainnet* to directly deploy the product in production, and *EthereumTestnet* for testing purposes. For this running example, *EthereumMainnet* has been selected, yet this decision does not have a great impact on the final product.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<configuration>
  <feature automatic="selected" name="OnChainTraceabilityApp"/>
  <feature automatic="selected" name="SmartContracts"/>
  <feature automatic="selected" name="TrackingMethod"/>
  <feature automatic="unselected" name="StateMachine"/>
  <feature automatic="selected" manual="selected" name="AssetTracking"/>
  <feature automatic="unselected" name="TokenizedAssets"/>
  <feature manual="selected" name="DataStructAssets"/>
  <feature automatic="selected" manual="selected" name="RecordRegistration"/>
  <feature automatic="selected" name="Participants"/>
  <feature automatic="selected" name="Individuals"/>
  <feature automatic="selected" name="CreateIndividual"/>
  <feature automatic="selected" name="CreateIndividualAtSetup"/>
  <feature manual="unselected" name="CreateIndividualDynamically"/>
  <feature automatic="selected" name="IndividualType"/>
  <feature manual="selected" name="Oracle"/>
  <feature manual="selected" name="Human"/>
  <feature manual="unselected" name="Service"/>
  <feature automatic="selected" name="DeleteIndividual"/>
  <feature manual="unselected" name="DeleteIndividualByIndividual"/>
  <feature manual="selected" name="DeleteIndividualByRole"/>
  <feature automatic="selected" name="Roles"/>
  <feature automatic="selected" name="CreateRoleAtSetup"/>
  <feature manual="selected" name="RemoveRole"/>
  <feature automatic="selected" manual="selected" name="AddRole"/>
  <feature manual="selected" name="AddRoleAtSetup"/>
  <feature manual="selected" name="AddRoleDynamically"/>
  <feature automatic="selected" name="Storage"/>
  <feature automatic="selected" name="StorageType"/>
  <feature automatic="selected" name="RecordHistory"/>
  <feature automatic="selected" name="StructuredRecords"/>
  <feature manual="selected" name="HashRecords"/>
  <feature automatic="selected" name="ContractsMetadata"/>
  <feature automatic="selected" name="AssetsData"/>
  <feature automatic="unselected" name="StateData"/>
  <feature automatic="selected" name="StorageEmplacement"/>
  <feature automatic="selected" name="OffChain"/>
  <feature automatic="selected" name="Database"/>
  <feature automatic="selected" name="OnChain"/>
  <feature manual="selected" name="EventsEmission"/>
  <feature automatic="selected" name="DataContracts"/>
  <feature automatic="selected" name="Frontend"/>
  <feature automatic="selected" name="DeploymentView"/>
  <feature automatic="selected" name="TraceabilitySetup"/>
  <feature automatic="selected" name="RecordsCollectionsSetup"/>
  <feature automatic="selected" name="AssetSetup"/>
  <feature manual="unselected" name="StateMachineSetup"/>
  <feature automatic="selected" name="BlockchainNetwork"/>
  <feature manual="selected" name="EthereumMainnet"/>
  <feature automatic="unselected" name="EthereumTestnet"/>
  <feature automatic="selected" name="ParticipantsSetup"/>
  <feature automatic="selected" name="IndividualsSetup"/>
  <feature automatic="selected" name="RolesSetup"/>
  <feature automatic="selected" name="ManagementView"/>
</configuration>
```

Listing 7.3: Fulfilled configuration of the Carasau bread traceability application in FeatureIDE.

7.4 Validation

This research work is driven by the main motivations of using software products compared to traditional software engineering, that stands as follows (Pohl, Böckle, and Van Der Linden, 2005):

- Reduction of development costs.
- Reduction of the time needed to create an application.
- Increased code quality.

In the end, these aspects can be summarized as a reduced cost of creating new applications. It is also expected that SPLs may assist non-blockchain experts in the design and implement blockchain applications, thus reducing the difficulty of creating new blockchain applications. This section aims to validate that the artifact proposed to address the technical research question satisfies both the reduction of costs and the decreased difficulty when developing blockchain applications.

7.4.1 Experiment

To assess that the SPL induces the reduction of cost and simplifies the creation of blockchain-based applications, an experiment has been designed for this purpose. It takes the form of Single-Case Mechanism Experiment (SCME), a DSR validation method already used in Chapter 4 and 6. In this experiment, a comparison is made between using SPLs or traditional software engineering to create blockchain-based applications. The goal is to show that SPL induces fewer development costs, but also less operational costs.

The experiment was carried out as follows. First, a sample of two studies has been chosen, called the Test Set in Table 7.1. This selection has been performed following two criteria: (1) the source code is available online and (2) the functional requirements of the application proposed in the study can be easily identified and extracted.

Then, for each study, the following steps have been conducted:

- *Requirements extraction* - extract main functional requirements[8] formulated by the authors for their on-chain traceability application.
- *Feature selection* - configure and generate a product using the web platform from these requirements.
- *Requirements satisfaction* - assess the satisfaction of formulated requirements towards the produced blockchain application.
- *Performance assessment* - compare the operating cost of deploying the on-chain part of the product and the implementation proposed by the authors.

[8]For the sake of brevity, the subset extracted was restricted to functional requirements that involve writing/modifying data.

During the feature selection step, the configuration of a product is guided by the formulated functional requirements. However, some features are left to be configured at the end, as selecting/deselecting these features does not have any impact on the satisfaction of these requirements. To arbitrate on these features, the source code of the reference paper implementation has been used to extend these requirements. As an example, although the first reference paper does not require the usage of events, the proposed implementation is using events in all of the functions. Thus, the `Events` feature has been selected in this configuration, following this information. Finally, if the reference implementation along the requirements does not allow finishing the configuration, the features left to be configured are automatically deselected. Indeed, as implementing additional features increases the operation cost, this measure allows for saving gas.

Regarding the performance assessment step, the operating cost will be measured in gas, a unit that represents the cost of performing an operation on an EVM-compatible[9] blockchain. It is computed by summing all the low-level operations performed during the operation (so-called opcodes). As the templates of the software product line have been written using Solidity, this metric is very relevant to assess and compare the performance of blockchain applications. However, other metrics might be considered for other technologies. This aspect is discussed in Subsection 7.5.1.

The cost reduction of developing blockchain applications involved by using the proposed software product line will be considered satisfying if the two following conditions are matched:

- The products generated from the web platform sufficiently match the requirements formulated by the authors of reproduced applications.
- The gas cost for the deployment and the execution of the generated smart contracts is satisfactory compared to the reference paper's implementations.

A graph compiles these costs for each reference implementation and generated products (Figure 7.7).

7.4.2 Spare Part Study Comparison

[9]The EVM (Ethereum Virtual Machine) is used by nodes to execute smart contracts.

Table 7.5: Spare parts study functional requirements (SR: satisfied in reference paper, SP: satisfied in a generated product).

Category	ID	Requirement	SR	SP
Purchase request	R.1.1	The engineer shall be able to submit a purchase request.	Yes	Yes
	R.1.2	The line manager shall be able to approve a purchase request.	Yes	Yes
	R.1.3	The procurement manager shall be able to approve a purchase request if the requested spare part in the request is missing from the inventory.	Yes	Yes
Purchase quotation	R.1.4	The procurement manager shall be able to submit purchase quotations for a requested spare part.	Yes	Yes
	R.1.5	The engineer shall be able to select a purchase quotation for a requested spare part.	Yes	Yes
	R.1.6	The procurement manager shall be able to confirm the availability of the requested spare part.	Yes	Partially
Purchase order	R.1.7	The engineer shall be able to submit a purchase order for a requested spare part.	Yes	Yes
	R.1.8	The line manager shall be able to approve a purchase order.	Yes	Yes
	R.1.9	The purchase manager shall be able to purchase the spare part specified by the approved purchase order.	Yes	Yes
	R.1.10	The engineer shall be able to request a spare part from the inventory.	Yes	Yes
	R.1.11	The engineer shall be able to submit a purchase order for a requested spare part.	Yes	Yes
Spare part transfer	R.1.12	An OEM (Original Equipment Manufacturer) shall be able to create a spare part entry.	No	Yes
	R.1.13	Any participant shall be able to transfer the spare part ownership to another.	No	Yes

The first study chosen for the validation discusses a blockchain-based traceability system for spare parts purchasing in manufacturing (Hasan et al., 2020). The main motivation for this study is the lack of spare parts tracing and tracking solutions, especially when they are employed in sensible domains, such as aeronautics. From this study, a set of 13 functional requirements have been identified and classified (Table 7.5). Then, a configuration has been created based on these requirements, and the corresponding product has been generated and deployed to assess its performance.

Feature Selection

In this configuration, two traceability features have been selected. The first feature is `AssetTracking`, as a representation of a spare part should be created by an OEM (Original Equipment Manufacturer) for ownership traceability purposes. As there is no need for modeling tokenized assets, the `StructuredAssets` subfeature is used. Then, the second chosen feature is `StateMachine`, as it is required to trace the current state of purchasing new spare parts. Regarding the `Participants` feature, only individuals have been included in the configuration. Indeed, there is no need to create groups of individuals (e.g., roles) in this scenario. The configuration does not include individual types either, as there are no oracle or external services specified.

For storage concerns, the spare parts study does not specify any off-chain storage, however, events are emitted along the process of refilling spare parts. Thus, the `EventsEmission` feature has been included. Also, the `StateData` and the `AssetsData` storage type subfeatures have also been included, due to the specified constraints between features.

Requirements Satisfaction

After the generation of a product based on this configuration, the satisfaction of requirements can then be assessed (Table 7.5). As a result, 12 of the 13 specific requirements are marked as satisfied. Indeed, the generated product is able to support these requirements by leveraging a state machine to track the state of spare parts refilling, and the ownership of spare parts through assets. However, one requirement has been marked as partially filled. The requirement R6 is difficult to satisfy with the current implementation of the product, as it requires establishing a communication system between the OEM and the procurement manager to ask for spare part availability.

As only the on-chain part of both applications (i.e., smart contracts) is evaluated, R.1.4 and R.1.12 have been marked as satisfied. These requirements demand storing some documents on IPFS (Inter-Planetary File System), a decentralized storage system (Benet, 2014), then storing the document reference (so-called tag) in the smart contract. Both the spare part study implementation and the generated product can do that, however, they do not propose a frontend feature to store a document on IPFS for the moment.

Note that requirements R.1.12 and R.1.13 have been marked as unsatisfactory in the spare part study implementation. Indeed, only one hardcoded spare part has been found in the

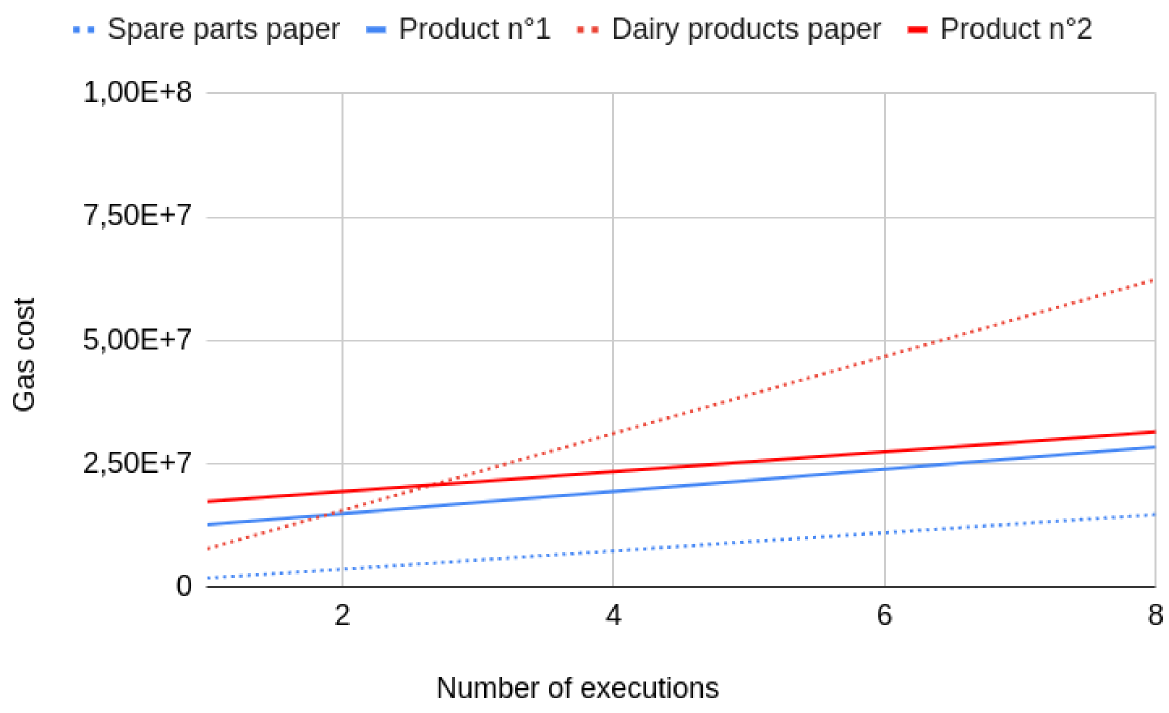


Figure 7.7: Gas cost of executing several times the reference implementations and generated products.

spare part study implementation code, and no function allows the transfer of a spare part from one participant to another.

Performance Assessment

To evaluate the performance ratio between the smart contract proposed in the spare part study and the generated product, a test scenario for spare part refilling has been designed, from the request to the purchase. This scenario covers the functional requirements specified in Table 7.5. Figure 7.7 compiles the differences from 1 to 8 executions.

The process to compute these metrics is the following. At first, the cost of deploying the smart contracts is assessed. This cost is separated from others as usually it is paid only one time by the user, at deployment. However, this is not the case for the spare part study architecture. Then, each function was executed, both in the smart contract proposed in the spare part study and the generated product. For the latter, the scenario involved the creation of an on-chain state machine using the same states as the spare part study, then transitioning from one state to another providing the same parameters as the first spare part study.

The cost of deploying the generated product is up to 10 431 963 gas, whereas the smart contract proposed in the spare part study costs 1 513 078 gas to be deployed. However, the generated product allows the creation of a new traceability process using already deployed contracts, where the smart contract proposed in the spare part study has to be redeployed to be used when starting a new traceability process. Thus, the deployment of smart contracts is not a one-time cost in the spare part study and has to be paid for each traceability process created. Also, the implementation cost of the generated product includes features for asset management, specified in the spare part study. However, these features are missing from the spare part study implementation. Regarding the cost of executing the scenario once the deployment is performed, the spare part study cumulates a gas cost of 329 840, where the generated product adds up to 2 248 064 gas. Note that two features specified in the requirements are missing from the spare part study implementation, thus the cost of the generated product for the 11 first requirements can be adjusted to 1 970 268 gas.

Figure 7.7 diSPLays a tendency of the execution cost of both spare parts study implementation and generated product. To extend these results, the cost of executing the scenario a high number of times was also computed. As a result, the cost of executing 100 times the scenario is $1.85 * 10^8$ gas for the spare parts study implementation, and $2.35 * 10^8$ gas for the generated product. For the spare part study implementation, the cost is obtained by summing 100 times the deployment and the function execution cost. In the generated product, the cost is obtained by summing 100 times the function execution cost and then adding the deployment cost. This difference in calculation method is explained as the smart contracts from the generated product do not have to be redeployed in order to create a new process. More rationale on identified cost differences between these two implementations is given in Section 7.5.

7.4.3 Dairy Products Study Comparison

For the second chosen study, a blockchain-based food supply chain traceability for dairy products is introduced (Casino et al., 2021). As safety is a critical aspect of food supply chains, blockchain and smart contracts can be used to build a secure and trustworthy architecture for food supply chain traceability. In their work, Casino et al. propose such architecture through a concrete use case for the traceability of dairy products. In this work, eleven functional requirements have been identified and classified (Table 7.6). From these requirements, a configuration has been made on the web platform, and the corresponding product has been generated for its performance evaluation.

Feature Selection

This case study both involves the tracking of asset ownership, records, and state changes in a process. The three different tracking methods have been selected to address these requirements: `AssetTracking`, `RecordsHistory`, and `StateMachine`. Also, as there is no need for modeling tokenized assets, the `StructuredAssets` subfeature is used.

For the `Participants` main feature, the paper describes the need to define two roles. The first one is the *Stakeholders* role: stakeholders are involved in the milk transformation process. The second one is the *Administrators* role. Members of this role group are employees from the dairy company and oversee the blockchain traceability application. They are able to perform administration operations, such as adding new stakeholders. The presence of roles in this application justifies the selection of the `Role` feature. According to the dairy products study, it is also possible to create new stakeholders or delete them at any moment. This involves the following features and their subfeatures: `CreateIndividual`, `DeleteIndividual`, and `AddRole`. However, `IndividualTypes` have not been added to the configuration, as there is no explicitly mentioned oracles or services.

Regarding the `Storage` feature, the study does not mention the emission of events. As this is an expensive feature in terms of gas cost, `EventEmission` has been excluded from the configuration. The other storage subfeatures, notably the ones related to the traceability data, were automatically selected.

Requirements Satisfaction

Once the configuration step was finished, the satisfaction of extracted requirements was assessed (Table 7.6). As a result, 8 out of 11 requirements have been marked as satisfied, 2 requirements marked as partially satisfied, and one requirement marked as unsatisfied. In the proposed software product line, an asset can only be weakly attached to a process (here, state machine instances), using the additional data field to reference the instance. Thus, the requirements R8 and R9 have also both been marked as partially satisfied. Regarding requirement 7, it is not satisfied as it demands a feature to stale an ongoing process, an aspect not handled by the generated product. Also, as only smart contracts are evaluated, the requirement R5 is satisfied. Indeed, as in the first study, it is possible to attach an IPFS tag to an asset in the generated product, in order to link it to a document. However, this requires uploading the document beforehand, a feature not handled by the web platform at the moment.

Table 7.6: Dairy products study functional requirements (SR: satisfied in reference paper, SP: satisfied in the generated product).

Category	**ID**	**Requirement**	**SR**	**SP**
Product management	R.1.1	An administrator or a stakeholder shall be able to create a product.	Yes	Yes
	R.1.2	An administrator shall be able to change the stakeholder of a product.	Yes	Yes
	R.1.3	A stakeholder shall be able to change the stakeholder of a product if owned.	Yes	Yes
	R.1.4	An administrator or a stakeholder shall be able to push a new temperature or location record for a given product.	Yes	Yes
	R.1.5	An administrator or a stakeholder shall be able to create and attach a chemical test to a product.	Yes	Yes
Milk transformation process management	R.1.6	An administrator or a stakeholder shall be able to create a new milk transformation process.	Yes	Yes
	R.1.7	An administrator or a stakeholder shall be able to disable a milk transformation process.	Yes	No
	R.1.8	An administrator shall be able to link a product to a milk transformation process.	Yes	Partially
	R.1.9	A stakeholder shall be able to link a product to a milk transformation process if owned.	Yes	Partially
Stakeholder management	R.1.10	An administrator shall be able to disable a stakeholder.	Yes	Yes
	R.1.11	An administrator shall be able to create a new stakeholder.	Yes	Yes

Performance Assessment

The cost of deploying the smart contracts is assessed, then each function was executed, both in the smart contract proposed in the first reference paper and the generated product. Figure 7.7 compiles the differences from 1 to 8 executions.

The cost of deploying the generated product is up to 15 400 174 gas, whereas the smart contract proposed in the dairy products study costs 6 748 484 gas to be deployed. As in the first paper, this is not a one-time cost for the dairy products study implementation: smart contracts have to be redeployed for each legal agreement signed between stakeholders and the dairy company in charge of the application. For the functions-related costs, the dairy products study implementation sums up a gas cost of 1 044 928, and the generated product a gas cost of 2 004 322.

As in the spare parts study comparison, the cost of executing a high number of times the scenario was also computed. The cost of executing 100 times this scenario is $7.79 * 10^8$ gas for the dairy products study implementation, and $2.17 * 10^8$ gas for the generated product. As explained in the first paper, the implementation cost has been added only one time to the generated product gas cost sum, whereas it has been added 100 times for the dairy products study implementation.

7.5 Discussion

7.5.1 Research Sub-Questions

In the previous section, it was assessed that the artifact proposed in this chapter, that is the SPL, is able to constitute an answer to the technical research question RQ5. This section further discusses the results obtained during the experiment, in light of formulated research sub-questions related to this chapter (Section 7). In particular, the ability to generalize the usage of SPLs in various application domains of blockchain technology is discussed, as well as potential divergences between traditional software engineering and SPLE.

RQ5.1: Is SPLE applicable to the blockchain field?

To address the first research sub-question, the satisfaction rate of requirements between the generated products and the reference papers implementations is assessed. Indeed, if it is possible to replicate most of the existing blockchain-based traceability applications by only using the web platform, the software product line approach is relevant. It has been shown that the web platform was able to produce blockchain applications that satisfy most of the requirements expressed by the studies that were used as references. Yet, some of the requirements were not fully satisfied. One reason is the genericity of the products that can be generated by the web platform. Indeed, the product line architecture and the templates have been designed to be very flexible rather than implementing specific domain-oriented features. An illustration of this flexibility is the management of roles: rather than using data structures tailored after the possible roles in a traceability application, a generic data structure named Role is implemented. Also, domain-specific features might be missing from the generated product. This has been faced during the artifact validation (Section 7.4), where some requirements need to verify a specific condition or execute a defined operation before changing the state of the traceability process. Nevertheless, the design of the product line architecture facilitates the integration of new domain-oriented features. In this case, the generated product is solid ground to start implementing more complex features on it.

RQ5.2: Do blockchain applications created following a standard software development engineering differs from applications derivated from a software product line?

The second research sub-question consists in evaluating if differences between applications generated from a software product line or implemented using a traditional software engineering approach exist. For these applications, the gas cost of deploying and then executing 100 times a defined scenario has been measured. Then, the divergence of design and code between these applications has been studied to explain the measured gas costs. For the spare parts study, the generated product was 27% more expensive to deploy and execute 100 times, and for the dairy products study, 72% less expensive. This difference is mainly due to two architectural aspects: the redeployment of smart contracts when willing to relaunch a new traceability process, and the deployment of numerous contracts to facilitate contract upgradeability. Indeed, redeploying a contract requires reallocating a large amount of storage to initialize state variables and store the source code, an expensive operation. The products generated by the web platform are designed to avoid this issue: a new state machine (by extension, a traceability process) can be created with a dedicated function rather than another deployment. The separation of concerns between data and logic also addresses this issue, as a new controller can be deployed to upgrade some features in generated products, rather than redeploying everything. However, this approach has a drawback: the logic required for the separation of concerns and easier upgradeability requires the deployment of bigger smart contracts. This results in a more expensive deployment for generated products.

The implementation of the different features of the generated products and reference study implementations also differs. For the latter, many hardcoded values were found, in particular for the definition of participants and roles. This leads to decreased gas costs, as there is no additional feature for dynamic management of them (e.g., getters and setters functions). On the opposite, the generated products derivated from the software product line are very flexible and foster maintainability and upgradeability. Consequently, the flexibility of this approach increases the operating costs of the application. Nevertheless, the high gas costs observed during the software product line validation might be reduced in future works by implementing features to reduce the code and needed storage size, to the detriment of upgradeability. Also, the deployment of smart contracts and the execution of functions is free on private blockchains networks, such as Proof-of-Authority-based Ethereum networks. In this context, it is not necessary to optimize the application in order to decrease its gas costs.

It should also be noted that although the gas cost is an accurate metric to describe Ethereum-based smart contracts performance, it is not systematically generalizable to any blockchain technology. Indeed, there is no gas cost at all on other non-EVM-based blockchains, such as Hyperledger Fabric. Other metrics might be considered to assess the performance of blockchain applications in future works, using these technologies. For instance, the resource usage of an application (e.g., CPU, RAM, storage size, etc.) could be monitored. The cost of executing the features themselves could also vary depending on the blockchain used. As an example, a feature for data confidentiality requires implementing a function to encrypt data on Ethereum-based blockchains. On Hyperledger Fabric, this is unnecessary as it is possible to restrict the read access of a contract to a defined set of participants, using channels (Androulaki et al., 2018).

7.5.2 Lessons Learned

The main advantage identified during the completion of the study was the time saved compared to manually developing traceability applications. Indeed, after the identification of desired requirements in these works, the configuration and the generation of blockchain applications can be done in minutes. Also, the quality of generated products benefits from the integration of good practices, design patterns, and standards in core assets. However, the main drawback to this approach is the time overhead needed and the difficulty to set up the software product line (feature analysis, feature model development, template development). For the latter, blockchain experts are still needed to design and implement core assets in the domain engineering phase. Nevertheless, a working SPL can easily be used by non-experts during the application engineering phase.

These lessons learned are in line with the advantage of SPL in general: reduced time-to-market, reduced costs, and enhanced product quality. However, using SPLs is a decision that should be carefully assessed by a company willing to follow this approach, as it implies significant costs and risks for the company (Rincón, Mazo, and Salinesi, 2018). It requires spending more time upfront to design core assets during the domain engineering phase. Thus, this approach might not be tailored for a company aiming to develop a single blockchain application. On the contrary, large companies willing to propose wide ranges of blockchain-based applications might benefit from the usage of SPLs.

The templating engine used in this contribution was enough to illustrate the capability of generating blockchain products from configurations. However, a domain engineer may feel limited by the templating engine when implementing many templates for large-scale software product lines. The implementation of the different features within templates might also be tedious, as it has to take into account all the possible combinations of features and possible nestings.

Nonetheless, this issue can be mitigated in the blockchain field by different means. First, smart contracts can be designed in a way that the resulting architecture is a set of loosely coupled smart contracts. This approach eases the addition of new features to the software product line. Such architecture is notably introduced by Tonelli et al. (Tonelli et al., 2019), as they implement a microservice-based system with blockchain smart contracts. Consequently, the architecture proposed in this work was designed with modularity as a main concern. Second, many design patterns, standards, and commonly reused code blocks already exist. As identified by Chen et al., 26% of Ethereum smart contracts code blocks are from reused sources, notably ERC20-related contracts (Chen et al., 2021). Indeed, ERC20[10] is a standard for the creation of fungible tokens on Ethereum. This existing code can be easily bundled into a feature, reusable in many software product lines.

7.5.3 Research Challenges

Using software product lines to create blockchain applications raises new research challenges to address. In this work, the Solidity language has been chosen to develop smart contracts. However, a wider range of languages exist to develop smart contracts for one or other blockchain technologies (e.g., Solidity, Go, Rust, etc.). Future feature models of blockchain products could contain a feature for the selection of a specific smart contract language. This feature could yield software product lines that are able to produce the same application for multiple blockchain technologies. It would allow developers to focus on the application to build rather than the blockchain target behind and its technical specificities. Still, there is an issue with the implementation of such features: the programming model might differ between different blockchains. For instance, Ethereum is account-based, whereas other blockchains such as Bitcoin, rely on a UXTO (unspent transaction output) model (Brünjes and Gabbay, 2020). A consequence of these different programming paradigms could be the impossibility to design some features with specific blockchain technologies.

Also, this chapter proposes a domain-oriented feature model (on-chain traceability), yet another type of feature model could be created around existing blockchain features. The resulting SPL could allow the creation of generic blockchain applications, that provide a solid ground for developers to start implementing the domain features above. However, a developer willing to use this SPL would still partially face the issue of writing blockchain-related code. Nonetheless, the most difficult aspects of blockchain software engineering could be handled by the SPL itself, while the developer could focus on designing and implementing domain-oriented code. For instance, a generic blockchain SPL could contain an `Oracle` feature. If selected by the developer during the configuration, the oracle would be included in the generated product with an adequate interface that eases its reuse.

The evolution of software product lines, when core assets (e.g., templates, feature models) evolve over time to address newer requirements or changes in the technology used (Marques et al., 2019), is also a challenge for blockchain software product lines. This issue is very relevant to blockchain: due to the novelty of the field, many existing standards, patterns, and commonly reused code blocks might change in the future, impacting existing features. Future research on blockchain-based software product lines should consider this issue and include mechanisms to handle the evolution of blockchain core assets.

[10] `https://ethereum.org/en/developers/docs/standards/tokens/erc-20`

7.6 Related Works

The application of SPLs to blockchain technologies remains unexplored in the existing literature. However, several works in the literature have already been proposed to assist practitioners in designing, generating, and deploying blockchain-based solutions, starting from low-level code generation tools to MDE approaches and proposals that take blockchain variability into account.

7.6.1 Smart Contract Code Generation

The most recent blockchain solution supports general-purpose programming languages, such as JVM-based languages Java/Kotlin for Corda (Hearn and Brown, 2016), or Go, Node.js, and Java for Hyperledger Fabric (Androulaki et al., 2018). Yet, the vast majority of the literature presenting blockchain-based solutions still rely on Ethereum and its Ethereum-specific languages (Solidity, Viper) to demonstrate the feasibility of their proposal. For this reason, several papers focus on helping developers write smart contracts with Ethereum.

Wöhrer and Zdun proposed a Contract Modeling Language (CML) to simplify the writing of smart contracts (Wohrer and Zdun, 2020). CML defines contract-specific concepts such as Party, Asset, or Event, and decorators to indicate the usage of blockchain-based design patterns in specific functions. A parser is also proposed to convert a CML file into Solidity code. However, this approach requires developers to become proficient in CML in addition to Solidity, which is not an easy undertaking. Indeed, only learning CML might limit developers in the development of smart contracts, as they would be restricted to CML existing elements.

Other approaches in the literature focus on reusing existing models to generate code. For instance, Zupan et al. propose a framework to generate smart contracts based on Petri nets (Zupan et al., 2020). These Petri nets model places that are linked by transitions that can be crossed under specific conditions. The generation of code is made through their translation engine, which is able to convert Petri nets into Solidity smart contracts. López-Pintado et al. use BPMN to generate a suite of Solidity smart contracts, able to run the corresponding business process on the blockchain with a solution called Caterpilar(Lòpez-Pintado et al., 2019). Generated smart contracts are used to start business process instances, manage business process activities, and handle the business process workflow. Choudhury et al. use a different model for smart contract generation composed of an ontology with classes linked together, and constraints expressed as a set of rules (Choudhury et al., 2018).

7.6.2 Blockchain and Model-Driven Engineering

Smart-contract code generation is useful for supported use cases where all the processed data happens to be on the blockchain. However, these approaches fall short when dealing with the integration of other domain-specific components into the blockchain solution at different architectural levels. Several authors propose relying on Model-Driven Engineering to help grasp the complexity integrating of blockchain-based solutions within the Information Systems,

Lu et al. propose a tool called Lorikeet that extends the BPMN modeling capabilities already proposed in Caterpillar with the support of asset registry management and interconnects them (Lu et al., 2020). Both Business Process modeling and Asset Registry modeling are used to generate smart contracts making the developers more productive, the operators able to monitor the execution of generated smart contracts, and the domain experts capable of understanding how their ideas are represented in the system. De Sousa and Burnay present MDE4BBIS, a framework to incorporate MDE in the development of blockchain-based IS (Sousa and Burnay, 2021). They demonstrate their solution to support cross-organizational business processes. Górski and Bednarski propose new UML stereotypes in a UML profile for distributed ledger deployment and incorporated their solution in a modeling tool to automate the deployment to Corda (Gorski and Bednarski, 2020).

7.6.3 Blockchain and Software Product Lines

Finally, a few proposals have been made to use software product lines for blockchain. Kim et al. present a feature model to allow organizations to build their own blockchain platform by selecting its features (e.g., smart contract language, consensus algorithm, etc.) (Kim et al., 2018). They present a feature model for blockchain platforms allowing the selection of the desired features, without however supporting feature binding or code generation. Liaskos et al. introduce a meta-model for the derivation of specialized blockchain network simulators, emphasizing the importance of SPLE and MDE (Liaskos, Anand, and Alimohammadi, 2020).

7.6.4 Comparison with the SPL Approach

The aforementioned MDE-based methods allow the partial or full generation of blockchain applications. However, several differences with the SPL approach can be underlined.

1. These methods can be applied to a wide range of use cases, yet their genericity often requires writing additional domain-oriented code. As such, non-experts might struggle with the modification of the produced applications to fit their needs.

2. The SPL approach provides a clear separation between the implementation of reusable artifacts (domain engineering) and the actual reuse in new products (application engineering), as well as methods and guidelines to handle SPL evolution (Marques et al., 2019). These aspects are important to maintain the generation of relevant blockchain applications, knowing the rapid pace of development in the blockchain field.

3. Compared to existing approaches, the usage of a feature model to define the variability of generated applications allows a fine-grained and clear selection of features by the developer.

These differences led to the usage of SPLE for BANCO, as the main purpose of the Harmonica framework is to assist non-expert practitioners in the design and implementation of blockchain applications.

7.7 Conclusion

As the development of blockchain applications is still tedious and error-prone, the usage of a software product line can help in the systematic reuse of existing code, good practices, and standards (e.g., Ethereum ERCs) to build robust and efficient applications. This chapter denotes the relevance of leveraging software product lines for the design and implementation of blockchain-based applications with an exemplified approach. First, a feature model for on-chain traceability applications is introduced, built by extracting features from 5 different works in this field. Along that, a web platform is proposed to allow the configuration of an on-chain application based on this feature model. The web platform also includes a code generator that reuses this configuration to feed a templating engine that produces a working blockchain application, without any coding. By specifying its desired features, the user is capable of generating an application for on-chain traceability that suits its needs. Also, the produced code is designed to be highly modular, thus easing the addition of new features, either through adding extra features in the feature model or manually. This approach is validated by using the web platform to recreate existing on-chain traceability applications proposed in the literature. In particular, it was assessed that using SPLs implies the reduction of costs, whether development or operational costs. It was also shown that it is easier for non-experts to create a blockchain application using a SPL rather than implementing everything from scratch. Many research challenges still have to be addressed, such as the management of the software product line evolution considering the rapid pace of blockchain development. Yet, this work paves the way for blockchain-backed solutions created with the software product line method.

This chapter is also the final part of the current version of the Harmonica framework, BANCO being its third artifact. BANCO proposes a method and a tool to semi-automate the implementation of a blockchain application, using the recommendations produced by BLADE (Chapter 4 and 6).

Chapter 8

Conclusion and Perspectives

Many issues and challenges may be faced by developers when designing and implementing blockchain applications (Chapter 2). First, developers may struggle with security issues that threaten their applications. For instance, they may introduce by mistake vulnerabilities in their smart contracts or neglect aspects such as data privacy. This may induce important consequences: for instance, smart contracts may hold high amounts of cryptocurrencies. Then, they may face blockchain-specific challenges. As an example, using a blockchain in an application involves dealing with data immutability or interoperability with other applications. Additionally, developers may have to consider cost and performance issues, as it may be very expensive to deploy and interact with smart contracts. Finally, some issues were identified related to the development environment when implementing blockchain applications, such as the lack of assisting tools, or the lack of guidelines.

The main goal of this thesis was to address these different classes of issues faced by developers. As such, the research aim of the thesis is to improve the software engineering process for creating blockchain-based applications, by designing a semi-automated framework composed of two assisting tools and a knowledge base that assists the practitioner along the tasks of designing and implementing blockchain-based applications to make the creation of blockchain-based applications easier, and reduces development cost (Chapter 1). By reusing existing concepts such as recommendation systems, software product lines, or design patterns, Harmonica aims to propose a solution to assist the practitioner in the design and implementation of blockchain-based applications.

The chapter is organized as follows. First, the thesis novel contributions are recapitulated in Section 8.1. The limitations related to the different artifacts as well as the framework as a whole are discussed in Section 8.2. Finally, future works are introduced in Section 8.3.

8.1 Novel Contribution

The driving idea of the thesis was to develop a framework, named Harmonica, to assist the practitioner in the design and the implementation of blockchain-based applications. The framework is constituted of two tools: BLADE, a recommender of blockchain technologies and patterns for a given set of requirements and preferences, and BANCO, a configurator and a generator of blockchain applications (Chapter 3). These tools can either be used as standalone by practitioners depending on their needs or used as a suite to navigate from the design to the implementation of an application. To support the execution of these tools, the framework also includes an ontological knowledge base of blockchain-based software patterns was also proposed.

To build this framework and its tools, 3 design artifacts were created:

- A web platform for blockchain technology recommendation in a given context. In order to compute an adequate ranking, the practitioner has to express their preferences and requirements towards 14 different blockchain attributes, that covers the different aspects of software quality defined by the ISO25010 standard. Where the requirements are used to discriminate blockchain if they dissatisfy one, preferences are used to attribute a weight for each of the 14 different attributes. A multi-criteria decision-making method, named TOPSIS, is then able to compute a matrix containing all of the blockchain attributes and the weights defined by the user preferences to output a ranking of blockchain technologies (Chapter 4).
- An ontology-based web platform for the selection of blockchain-based software patterns. The driving idea of ontology lies in the distinction between the concept of pattern and the patterns proposed (so-called proposals) in academic papers. Indeed, many papers might propose the same pattern or closely related variants using different wordings and descriptions. This ontology also extensively reuses the knowledge gathered in the aforementioned systematic literature review and extends it, notably with the inference of relations between patterns from the relations between proposals. The web platform reuses this ontology by exposing the patterns as a catalog, but also by implementing a recommendation system of patterns using the ontology concepts and knowledge (Chapter 6).
- A SPL-based web platform for the configuration and the generation of a blockchain application. To guide the configuration, a feature model has been created to model possible combinations of features for a chosen domain, that is on-chain traceability. The feature model also describes constraints between one or more patterns, to prevent invalid configurations. The practitioner is then able to configure a blockchain application using the web platform w.r.t. the feature model. The generator is then able to ingest the configuration to create on-the-shelf blockchain products that can be used as-is (Chapter 7).

These artifacts were created in the context of the Design Science Research (DSR) method, where the creation of these artifacts was driven by three technical research questions. In this context, each design artifact constitutes an answer to its associated technical research question as long as it satisfies the stated requirements. As a result, it was assessed that the framework can successfully assist the user in the selection of a blockchain technology (RQ2), recommend a set of blockchain-based software patterns for given requirements (RQ4), and generate on-the-shelf blockchain applications based on previous recommendations, design decisions, and an adequate product configuration (RQ5).

Along with the construction of these artifacts, this thesis has also proposed several contributions to the state of the art of blockchain technologies:

- A focused literature review of the state of the art of issues met by developers when designing or implementing blockchain applications. In total, 67 studies were queried and then filtered to create a corpus of 6 papers. It has led to the identification of 57 unique issues, that were classified into 5 categories and 9 sub-categories.
- A systematic literature review of the state of the art of blockchain-based software patterns. In total, 98 studies were retrieved, from which 20 studies have been kept for reading. The completion of this review has led to the construction of the blockchain-based software pattern collection aforementioned, but also resulted in actionable knowledge in this field (Chapter 5):
  - A collection of 114 unique blockchain-based software patterns. Each pattern is described in a simplified pattern format, derived from the GoF or Alexandrian pattern format, to address the lack of uniformity in the description of patterns in each paper.
  - A taxonomy for blockchain-based design patterns constituted of 20 different categories and distributed over 3 layers of classification. It has been proposed to classify these patterns, empirically created from the patterns extracted during the systematic literature review.
  - An overview of key blockchain-based software patterns that are often present in existing blockchain applications.
  - The relations between existing software patterns and blockchain-based software patterns were identified, showing the applicability of some existing software patterns to the blockchain field.
  - The application domains of identified blockchain-based software patterns, highlighting the current domain agnosticism of blockchain-based software patterns.
  - Research gaps in blockchain-based software patterns, such as the lack of blockchain technology support and the lack of architectural patterns/idioms proposals.
- An illustration of the applicability of software product lines to blockchain applications, with the end-to-end construction of a blockchain software product line (BANCO). This applicability was carefully evaluated by assessing that blockchain products benefit from the same advantages as other software product lines (cost reduction, reduced time-to-market, and enhanced quality). Also, several research challenges and lessons learned were identified to further guide future researchers on this topic (Chapter 7).

These contributions were created by answering two knowledge questions, in the context of the DSR method. Knowledge questions are different from technical research questions as they do not aim to produce new artifacts, but the knowledge that will serve the artifacts or contribute to the research context. In this thesis, the issues related to the design and implementation of blockchain-based applications met by developers have been identified (RQ1), and a collection of blockchain-based patterns has been constituted from the existing state-of-the-art (RQ3).

The different artifacts that were proposed in this thesis fill multiple research gaps in state-of-the-art of blockchain application development. First, the holistic approach taken to build the framework can be mentioned. Many approaches, tools, methods, and solutions exist in the literature to tackle multiple design and implementation issues met by practitioners, as mentioned in the different related works sections in Chapter 4, 5, 6, and 7. Yet, these solutions were often addressing specific aspects of the design and implementation of blockchain applications, instead of assisting the practitioner with these steps. The Harmonica framework proposes a different approach, composed of 3 artifacts that may be used in order to address the major issues when they occur during the creation of blockchain-based applications.

This contribution also adapts existing approaches in software engineering to blockchain technologies. Three approaches can be mentioned: the usage of software patterns, SPLs, and ontologies. The concept of software pattern is a well-known solution to formalize existing knowledge on recurring problems and solutions. Although this solution was already proposed multiple times in the literature for blockchain technologies, throughout pattern collections or applications (Chapter 5 and 6), this thesis has led to the creation of a large pattern collection that aggregates existing literature, and a novel way of structuring this knowledge that is the blockchain-based software pattern ontology. Regarding SPL, this thesis was among the first to investigate the application of SPLE to blockchain technologies (Chapter 7). Although many approaches were already explored in the Model-Driven Engineering (MDE) field (e.g. BPMN and Petri Nets code generation), this thesis adapts an existing approach, that is SPL and configurators, as a new way to generate blockchain applications.

8.2 Limitations

However, multiple limitations were identified in the completion of this work. The upgradeability of the framework knowledge is the first limitation of this work. Indeed, the knowledge base of the framework might become more and more obsolete over time, regarding the rapid pace of evolution in the blockchain field. For instance, the description of blockchain technologies using multiple attributes may become outdated. During the completion of this thesis, the mainnet of the Ethereum blockchain changed its consensus algorithm from Proof-of-Work (PoW) to Proof-of-Stake (PoS). Also, blockchain technologies that quickly gained traction from users may be absent from the knowledge base, as they were not added yet. This limitation also affects the recommendation of blockchain-based software patterns, as developer practices may evolve fast and new patterns may be identified, or existing patterns may be adapted to new blockchains. Finally, the templates that were implemented to serve the generation process of BANCO may also be threatened by obsolescence: for instance, programming languages and libraries may evolve, modifying their syntax.

Another limitation of the framework is its applicability to only two phases of the software engineering process: design and implementation. In order to achieve the construction of the Harmonica framework within the thesis timeframe, this focus was needed. Although these steps might be the most tedious to handle as a non-expert, including all phases of software engineering in the framework (requirements elicitation, deployment, maintenance) would be highly beneficial to assist the practitioner in all the steps of creating blockchain-based applications. Yet, efforts have been made to allow composability between the framework and future works in the field. Indeed, every tool within the framework takes a specific input and produces a specific output. This allows designing new tools for other software engineering process phases that produce outputs reusable by the Harmonica framework tools, or that reuse outputs from the Harmonica framework tools. Thus, the framework can evolve along with the research works in this field, but also with the evolution in blockchain application development.

The validity of the framework can also be discussed. Although every artifact as well as the knowledge that supports their execution were validated independently using the methods suggested by Wieringa (Wieringa, 2014), there was no validation of the framework as a whole. Nevertheless, as each research question formulated to address the research aim was answered and produced artifacts were validated, the framework can be considered valid. Yet, future works may evaluate the usability of the framework by practitioners, to further demonstrate that it constitutes a solution to the formulated research aim but also identify if using every tool in conjunction yields additional benefits.

Finally, although each part of the framework is able to reuse the results from the previous one, the coupling could be improved. For instance, the recommendations of blockchain-based software patterns in BLADE should impact the configuration of the blockchain product in BANCO.

8.3 Future Works

The different research works carried out in this thesis pave the way for future works. In this section, these possible future works for each artifact as well as the framework are introduced.

BLADE - Blockchain Technology Recommendation

Regarding BLADE, its produced blockchain technology recommendations might be improved through different research works. First, by adding more attributes and alternatives into the knowledge base to better consider the state of the art of blockchain technologies. By adding more attributes, the precision of recommendations would increase, as the recommendation engine has more parameters to rank the blockchain technologies. In parallel, adding more blockchain technologies would offer more options for the practitioner to design its application.

Then, another types of inputs may also be considered to further refine the recommendations made by the system (system architecture topology, infrastructure, business processes, etc.). For instance, this information could be used to run a customized performance test (such as the one presented in the Subsection 4.5.4) for each user before even running the decision algorithm, the goal being to set the values of the varying criteria (transaction throughput, latency ...) extremely precisely. Another way to improve is the use of approaches based on fuzzy logic or Bayesian models that would allow taking into account the subjective aspect of the decision criteria.

One of the most important challenges that will also have to be addressed in the future will be the updating of the knowledge base. As mentioned earlier, the precision of BLADE is bonded to the correctness of the knowledge base: outdated knowledge would yield outdated recommendations. Indeed, the result provided by BLADE will be relevant if the attributes remain up to date, through the evolution of the different blockchains proposed, the addition of attributes used in the tool, and the benchmarks carried out which will allow refining certain values of the knowledge base. To address this issue, close collaboration with companies as well as blockchain experts and architects might be envisioned.

BLADE - Blockchain-based Software Patterns Recommendation

Multiple future works are envisioned to improve the collection of patterns and ontology. Regarding the collection of patterns, research work could be carried out on the translation of GoF patterns that have not been translated yet as blockchain-based patterns. During the systematic literature review (Chapter 5), it has also been found that while some patterns are described in a very generic form, some variants propose specific forms of patterns based on them, such as the *Oracle* pattern that can be derived into 4 different variants (Mühlberger et al., 2020). Further research in creating architectural patterns and idioms for various blockchain protocols could benefit the development of robust blockchain-based applications.

Some extensions could also be envisioned for the ontology, in the software pattern domain. Although the pattern proposal subspace is introduced within the scope of blockchain patterns, it could be generalized to all software patterns, such as Internet-of-Things (IoT) or microservices. Finally, existing software patterns in the ontology might be extended to include a formal description using existing pattern formats.

BANCO

Regarding BANCO, the tool currently supports only one domain, that is on-chain traceability, but also one blockchain technology (Ethereum). More domains and blockchain technologies might be added in the future to extend their applicability in other fields. Another envisioned extension of BANCO could be the support of domain-agnostic configuration. Rather than selecting domain-related features (for instance, record management for on-chain traceability), blockchain-based features might be designed and then coupled together to generate a robust ground of a blockchain application, where domain-specific features might be added later. This would improve the versatility of BANCO, allowing its usage in many domains rather than specific ones.

Harmonica Framework

To conclude with future works, more phases of the software engineering process might be considered in Harmonica, to further ease the development of blockchain-based applications without prior extensive knowledge. This notably includes the requirements specifications, where blockchain-specific requirements might emerge, but also the deployment and maintenance phases, that require the practitioner to have extensive knowledge in upgradeability methods of blockchain applications.

As mentioned before, the coupling of the different parts of the framework may be improved. Future works will target better reuse of existing tool outputs to improve others, such as reusing the requirements provided to BLADE for blockchain technology recommendations in the existing or upcoming tools.

While Harmonica still has room for improvement, this work opens the way for future improvements in the guidance of practitioners in the creation of blockchain applications, thus greatly contributing to the adoption of the technology in many sectors.

Appendix A

In this appendix, the full table of issues met by practitioners in the design of blockchain-based applications, for each paper from the literature review in Chapter 2, is given.

Table 1: Issues met by practitioners in the design of blockchain-based applications, for each literature review paper.

Ref.	**Issue**
(Ayman et al., 2020)	• Avoiding smart contract vulnerabilities • Low awarness of existing tooling
(Worley and Skjellum, 2018)	• Code immutability • Chain-boundedness • Cost of use
(Bosu et al., 2019)	• Cost of defects • Steep learning curve • Complex environment • Data immutability • Technological complexity • High pace development • Scalability • Application security • Code reviews • Lack of documentation • Lack of tools
(Zou et al., 2019)	• Handling sensitive data • Irreversible transactions • Code unmodifiable • Public code access • Flaws in compiler • Lack of best practices for writing safe code • Lack of tools and techniques to verify code correctness • Longer code reviews • Lack of powerful debuggers • Non-informative error messages • Solidity lack of general purpose library • Solidity lack of support for error logging and reporting

Continued on next page

Table 1 – *Continued from previous page*

Ref.	**Issue**
	• Solidity lack of standards and rules • Solidity lack of data safety checks • Solidity inconvenient way to call external functions • Solidity lack of support for memory management • Solidity constrained number of local variables • EVM limited support for debugging • EVM lack of support of traditional language • EVM inefficiency of bytecode execution • EVM limited stack size • High gas cost • Transaction failure due to insufficient amount of gas • No gas estimation tool at code level • Tradeoff between gas optimisation and code reliability • Lack of reference code • Lack of standardized knowledge • Lack of up-to-date documentations • Lack of community support
(Lokshina and Lanting, 2021)	• Blockchain usage and selection • Handle latency in architecture • Interoperate dApps with already existing ones • Transactional privacy
(Kannengiesser et al., 2021)	• Code Visibility • Data Visibility • Pseudonimity • Randomness • Transaction Ordering Dependance • Code Discoverability • Code Updatability • Execution Restriction • Resource Management • Undefined Behavior • Arithmetic Behavior • Concurrency • Non-deterministic Behavior • Conformity to Expectations • Cross Account Interactions • Encapsulation • Error Handling • Programming Language Concept Compliance • Iteration Through Data Structures

Continued on next page

Table 1 – *Continued from previous page*

Ref.	**Issue**
	• Data Storage • Data Type Complexity • Under-optimized code • Required Interactions • Readability • Ease of Code reuse • Appropriate Data Type use • Semantics Soundness • Technical Soundness • Smart Contract API Conformity

References

Agung, Anak Agung Gde and Rini Handayani (2020). "Blockchain for smart grid". In: *Journal of King Saud University-Computer and Information Sciences*.

Alexander, Christopher (1977). *A pattern language: towns, buildings, construction*. Oxford university press.

Alexander, Christopher et al. (1979). *The timeless way of building*. Vol. 1. New york: Oxford university press.

Allen, I Elaine and Christopher A Seaman (2007). "Likert scales and data analyses". In: *Quality progress* 40.7, pp. 64–65.

Androulaki, Elli et al. (2018). "Hyperledger fabric: a distributed operating system for permissioned blockchains". In: *Proceedings of the Thirteenth EuroSys Conference*, pp. 1–15.

Ayman, Afiya et al. (2020). "Smart contract development from the perspective of developers: Topics and issues discussed on social media". In: *International Conference on Financial Cryptography and Data Security*. Springer, pp. 405–422.

Back, Adam et al. (2002). *Hashcash-a denial of service counter-measure*.

Bandara, HMN Dilum, Xiwei Xu, and Ingo Weber (2020). "Patterns for blockchain data migration". In: *Proceedings of the European Conference on Pattern Languages of Programs 2020*, pp. 1–19.

Baralla, Gavina et al. (2021). "Ensuring transparency and traceability of food local products: A blockchain application to a Smart Tourism Region". In: *Concurrency and Computation: Practice and Experience* 33.1, e5857.

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

Bass, Len, Paul Clements, and Rick Kazman (2003). *Software architecture in practice*. Addison-Wesley Professional.

Beck, Kent (1987). "Using Pattern Languages for Object-Oriented Programs". In: *Proceedings of OOPSLA (Object-Oriented Programming, Systems, Languages & Applications)*. url: http://c2.com/doc/oopsla87.html.

Belotti, Marianna et al. (2019). "A vademecum on blockchain technologies: When, which, and how". In: *IEEE Communications Surveys & Tutorials* 21.4, pp. 3796–3838.

Ben, Strack (2022). *There's a Shortage of Tech (And Female) Crypto Talent: Report*. https://blockworks.co/news/theres-a-shortage-of-tech-and-female-crypto-talent-report. [Accessed 20-Dec-2022].

Benet, Juan (2014). *IPFS - Content Addressed, Versioned, P2P File System*. eprint: arXiv:1407.3561.

Bosu, Amiangshu et al. (2019). “Understanding the motivations, challenges and needs of blockchain software developers: A survey”. In: *Empirical Software Engineering* 24.4, pp. 2636–2673.

Brown, Richard Gendal et al. (2016). “Corda: an introduction”. In: *R3 CEV, August* 1, pp. 1–15.

Brünjes, Lars and Murdoch J Gabbay (2020). “UTxO-vs account-based smart contract blockchain programming paradigms”. In: *International Symposium on Leveraging Applications of Formal Methods*. Springer, pp. 73–88.

Buterin, Vitalik et al. (2013). “Ethereum white paper”. In: *GitHub repository* 1, pp. 22–23.

Caro, Miguel Pincheira et al. (2018). “Blockchain-based traceability in Agri-Food supply chain management: A practical implementation”. In: *2018 IoT Vertical and Topical Summit on Agriculture-Tuscany (IOT Tuscany)*. IEEE, pp. 1–4.

Casino, Fran et al. (2021). “Blockchain-based food supply chain traceability: a case study in the dairy sector”. In: *International Journal of Production Research* 59.19, pp. 5758–5770.

Chen, Xiangping et al. (2021). “Understanding code reuse in smart contracts”. In: *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, pp. 470–479.

Chohan, Usman W (2021). “The double spending problem and cryptocurrencies”. In: *Available at SSRN 3090174*.

Choudhury, Olivia et al. (2018). “Auto-generation of smart contracts from domain-specific ontologies and semantic rules”. In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, pp. 963–970.

Cocco, Luisanna et al. (2021). “A blockchain-based traceability system in agri-food SME: Case study of a traditional bakery”. In: *IEEE Access* 9, pp. 62899–62915.

CoinMarketCap (2020). *Historical Snapshot - 02 February 2020 - CoinMarketCap*. https://coinmarketcap.com/historical/20200202/. [Accessed 20-Dec-2022].

ConsenSys (2019). *Gartner: Blockchain Will Deliver $3.1 Trillion Dollars in Value by 2030*. https://media.consensys.net/gartner-blockchain-will-deliver-3-1-trillion-dollars-in-value-by-2030-d32b79c4c560. [Accessed 20-Dec-2022].

Croft, W Bruce, Donald Metzler, and Trevor Strohman (2010). *Search engines: Information retrieval in practice*. Vol. 520. Addison-Wesley Reading, pp. 308–322.

Cunha, Paulo Rupino da, Piotr Soja, and Marinos Themistocleous (2021a). “Blockchain for development: a guiding framework”. In: *Information Technology for Development* 27.3, pp. 417–438. doi: 10.1080/02681102.2021.1935453. eprint: https://doi.org/10.1080/02681102.2021.1935453. url: https://doi.org/10.1080/02681102.2021.1935453.

Cunha, Paulo Rupino da, Piotr Soja, and Marinos Themistocleous (2021b). *Blockchain for development: a guiding framework*.

Czarnecki, Krzysztof and Eisenecker Ulrich (2000). *Generative Programming: Methods, Tools, and Applications*. Reading, MA, USA: Addison-Wesley, p. 864. isbn: 0201309777.

David, Siegel (2016). *CoinDesk: Bitcoin, Ethereum, Crypto News and Price Data*. https://www.coindesk.com/understanding-dao-hack-journalists. [Accessed 20-Dec-2022].

De Kruijff, Joost and Hans Weigand (2017). “Understanding the blockchain using enterprise ontology”. In: *International Conference on Advanced Information Systems Engineering*. Springer, pp. 29–43.

Deloitte (2020). *Deloitte’s 2020 Global Blockchain Survey*. https://www2.deloitte.com/content/dam/Deloitte/tw/Documents/financial-services/2020-global-blockchain-survey.pdf. [Accessed 20-Dec-2022].

Di Angelo, Monika and Gernot Salzer (2020). “Tokens, types, and standards: identification and utilization in Ethereum”. In: *2020 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*. IEEE, pp. 1–10.

Di Ciccio, Claudio et al. (2019). “Blockchain support for collaborative business processes”. In: *Informatik Spektrum* 42.3, pp. 182–190.

Eberhardt, Jacob and Stefan Tai (2017). “On or off the blockchain? Insights on off-chaining computation and data”. In: *European Conference on Service-Oriented and Cloud Computing*. Springer, pp. 3–15.

Fan, Caixiang et al. (2020). “Performance evaluation of blockchain systems: A systematic survey”. In: *IEEE Access* 8, pp. 126927–126950.

Farshidi, S. et al. (2020). “Decision Support for blockchain Platform Selection: Three Industry Case Studies”. In: *IEEE Transactions on Engineering Management*, pp. 1–20. issn: 1558-0040. doi: 10.1109/TEM.2019.2956897.

Feitosa, Daniel et al. (2020). “Code reuse in practice: Benefiting or harming technical debt”. In: *Journal of Systems and Software* 167, p. 110618.

Figorilli, Simone et al. (2018). “A blockchain implementation prototype for the electronic open source traceability of wood along the whole supply chain”. In: *Sensors* 18.9, p. 3133.

Figueira, José Rui et al. (2013). “An overview of ELECTRE methods and their recent extensions”. In: *Journal of Multi-Criteria Decision Analysis* 20.1-2, pp. 61–85.

Gamma, Erich et al. (1995). *Elements of reusable object-oriented software*. Vol. 99. Addison-Wesley Reading, Massachusetts.

García-Cascales, M Socorro and M Teresa Lamata (2012). “On rank reversal and TOPSIS method”. In: *Mathematical and Computer Modelling* 56.5-6, pp. 123–132.

Garousi, Vahid, Michael Felderer, and Mika V Mäntylä (2019). “Guidelines for including grey literature and conducting multivocal literature reviews in software engineering”. In: *Information and Software Technology* 106, pp. 101–121.

Gervais, Arthur et al. (2016). “On the security and performance of proof of work blockchains”. In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pp. 3–16.

Gilcrest, Jack and Arthur Carvalho (2018). “Smart contracts: Legal considerations”. In: *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, pp. 3277–3281.

Girardi, Rosario and Alisson Neres Lindoso (2006). “An ontology-based knowledge base for the representation and reuse of software patterns”. In: *ACM SIGSOFT Software Engineering Notes* 31.1, pp. 1–6.

Glaser, Florian (2017). “Pervasive Decentralisation of Digital Infrastructures: A Framework for Blockchain enabled System and Use Case Analysis”. In: *Proceedings of the 50th Hawaii International Conference on System Sciences (2017)*. Hawaii International Conference on System Sciences. doi: 10.24251/hicss.2017.186. url: https://doi.org/10.24251/hicss.2017.186.

Gorski, Tomasz and Jakub Bednarski (2020). “Applying Model-Driven Engineering to Distributed Ledger Deployment”. In: *IEEE Access* 8, pp. 118245–118261. doi: 10.1109/access.2020.3005519. url: https://doi.org/10.1109%2Faccess.2020.3005519.

Haoues, Mariem et al. (2017). “A guideline for software architecture selection based on ISO 25010 quality related characteristics”. In: *International Journal of System Assurance Engineering and Management* 8.2, pp. 886–909.

Harris, Robert (1998). *Introduction to Decision Making, Part 1*. http://www.virtualsalt.com/introduction-to-decision-making-part-1/. [Accessed 21-Dec-2022].

Harrison, Neil B (1999). “The language of shepherding”. In: *Pattern languages of program design* 5, pp. 507–530.

Hasan, Haya R. et al. (2020). “Blockchain-Based Solution for the Traceability of Spare Parts in Manufacturing”. In: *IEEE Access* 8, pp. 100308–100322. doi: 10.1109/access.2020.2998159. url: https://doi.org/10.1109/access.2020.2998159.

Hearn, Mike and Richard Gendal Brown (2016). “Corda: A distributed ledger”. In: *Corda Technical White Paper* 2016.

Hector, Ugarte-Rojas and Chullo-Llave Boris (2020). *BLONDiE: blockchain Ontology with Dynamic Extensibility*. eprint: arXivpreprintarXiv:2008.09518.

Henninger, Scott and Padmapriya Ashokkumar (2006). “An ontology-based metamodel for software patterns”. In: *CSE Technical reports*, p. 55.

Herbaut, Nicolas and Daniel Negru (2017). “A model for collaborative blockchain-based video delivery relying on advanced network services chains”. In: *IEEE Communications Magazine* 55.9, pp. 70–76.

Huang, Jingwen (2008). “Combining entropy weight and TOPSIS method for information system selection”. In: *2008 IEEE Conference on Cybernetics and Intelligent Systems*, pp. 1281–1284.

Humbeeck, Andries Van (2019). “The blockchain-GDPR paradox”. In: *Journal of Data Protection and Privacy*.

Hutchinson, John, Jon Whittle, and Mark Rouncefield (2014). “Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure”. In: *Science of Computer Programming* 89, pp. 144–161.

Jalil, Masita Abdul and Shahrul Azman Mohd Noah (2007). “The difficulties of using design patterns among novices: An exploratory study”. In: *2007 International Conference on Computational Science and its Applications (ICCSA 2007)*. IEEE, pp. 97–103.

Jörges, Sven (2013). *Construction and evolution of code generators: A model-driven and service-oriented approach*. Vol. 7747. Springer, pp. 29–31.

Jurgelaitis, Mantas, Rita Butkienė, et al. (2022). “Solidity Code Generation From UML State Machines in Model-Driven Smart Contract Development”. In: *IEEE Access* 10, pp. 33465–33481.

Juziuk, Joanna, Danny Weyns, and Tom Holvoet (2014). “Design patterns for multi-agent systems: A systematic literature review”. In: *Agent-Oriented Software Engineering*. Springer, pp. 79–99.

Kampffmeyer, Holger and Steffen Zschaler (2007). “Finding the pattern you need: The design pattern intent ontology”. In: *International Conference on Model Driven Engineering Languages and Systems*. Springer, pp. 211–225.

Kannengiesser, Niclas et al. (2021). “Challenges and common solutions in smart contract development”. In: *IEEE Transactions on Software Engineering*.

Khan, Shafaq Naheed et al. (2021). “Blockchain smart contracts: Applications, challenges, and future trends”. In: *Peer-to-peer Networking and Applications* 14.5, pp. 2901–2925.

Kim, Suntae et al. (2018). “A Feature based Content Analysis of blockchain Platforms”. In: *2018 Tenth International Conference on Ubiquitous and Future Networks (ICUFN)*. IEEE. doi: `10.1109/icufn.2018.8436843`. url: `https://doi.org/10.1109%2Ficufn.2018.8436843`.

Kitchenham, B. and S Charters (2007). *Guidelines for performing Systematic Literature Reviews in Software Engineering*.

Koens, Tommy and Erik Poll (2018). “What blockchain alternative do you need?” In: *Data Privacy Management, Cryptocurrencies and blockchain Technology*. Springer, pp. 113–129.

Kornyshova, Elena and Camille Salinesi (2007). “MCDM techniques selection approaches: state of the art”. In: *2007 IEEE Symposium on Computational Intelligence in Multi-Criteria Decision-Making*, pp. 22–29.

Krueger, Charles W (2009). “New methods behind a new generation of software product line successes”. In: *Applied software product line engineering*. Auerbach Publications, pp. 61–82.

Kuhn, Marlene et al. (2021). “Blockchain-based application for the traceability of complex assembly structures”. In: *Journal of Manufacturing Systems* 59, pp. 617–630.

Kuiter, Elias et al. (2018). “Getting rid of clone-and-own: moving to a software product line for temperature monitoring”. In: *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 1*, pp. 179–189.

Labazova, Olga (Dec. 2019). “Towards a Framework for Evaluation of blockchain Implementations”. In: *ICIS 2019 Proceedings*, pp. 1–10.

Lai, Young-Jou, Ting-Yun Liu, and Ching-Lai Hwang (1994). “Topsis for MODM”. In: *European journal of operational research* 76.3, pp. 486–500.

Lemieux, Victoria L (2017). “A typology of blockchain recordkeeping solutions and some reflections on their implications for the future of archival preservation”. In: *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, pp. 2271–2278.

Liang, Wei et al. (2020). “Circuit copyright blockchain: Blockchain-based homomorphic encryption for IP circuit protection”. In: *IEEE Transactions on Emerging Topics in Computing*.

Liaskos, Sotirios, Tarun Anand, and Nahid Alimohammadi (2020). “Architecting blockchain network simulators: a model-driven perspective”. In: *2020 IEEE International Conference* *on blockchain and Cryptocurrency (ICBC)*. IEEE. doi: 10.1109/icbc48266.2020.9169413. url: https://doi.org/10.1109%2Ficbc48266.2020.9169413.

Liu, Yue et al. (2018). "Applying design patterns in smart contracts". In: *International Conference on Blockchain*. Springer, pp. 92–106.

Liu, Yue et al. (2020). "Design patterns for blockchain-based self-sovereign identity". In: *Proceedings of the European Conference on Pattern Languages of Programs 2020*, pp. 1–14.

Lokshina, Izabella V and Cees JM Lanting (2021). "Revisiting state-of-the-art applications of the blockchain technology: analysis of unresolved issues and potential development". In: *Developments in Information & Knowledge Management for Business Applications*. Springer, pp. 403–439.

Longo, Francesco et al. (2019). "Blockchain-enabled supply chain: An experimental study". In: *Computers & Industrial Engineering* 136, pp. 57–69.

Lu, Qinghua et al. (2019). "uBaaS: A unified blockchain as a service platform". In: *Future Generation Computer Systems* 101, pp. 564–575.

Lu, Qinghua et al. (2020). "Integrated model-driven engineering of blockchain applications for business processes and asset management". In: *Software: Practice and Experience* 51.5, pp. 1059–1079. doi: 10.1002/spe.2931. url: https://doi.org/10.1002%2Fspe.2931.

Lòpez-Pintado, Orlenys et al. (2019). "Caterpillar: a business process execution engine on the Ethereum blockchain". In: *Software: Practice and Experience* 49.7, pp. 1162–1193. doi: 10.1002/spe.2702. url: https://doi.org/10.1002%2Fspe.2702.

Marchesi, Lodovica et al. (2020). "Design patterns for gas optimization in ethereum". In: *2020 IEEE International Workshop on blockchain Oriented Software Engineering (IWBOSE)*. IEEE, pp. 9–15.

Marques, Maíra et al. (2019). "Software product line evolution: A systematic literature review". In: *Information and Software Technology* 105, pp. 190–208.

Mavridou, Anastasia and Aron Laszka (2018). "Designing secure ethereum smart contracts: A finite state machine based approach". In: *International Conference on Financial Cryptography and Data Security*. Springer, pp. 523–540.

Mehar, Muhammad Izhar et al. (2019). "Understanding a revolutionary and flawed grand experiment in blockchain: the DAO attack". In: *Journal of Cases on Information Technology (JCIT)* 21.1, pp. 19–32.

Merkle, Ralph C (1989). "A certified digital signature". In: *Conference on the Theory and Application of Cryptology*. Springer, pp. 218–238.

Meszaros, Doble J and Jim Doble (1997). "G. A pattern language for pattern writing". In: *Proceedings of International Conference on Pattern languages of program design (1997)*. Vol. 131, p. 164.

Mingxiao, Du et al. (2017). "A review on consensus algorithm of blockchain". In: *2017 IEEE international conference on systems, man, and cybernetics (SMC)*. IEEE, pp. 2567–2572.

Moreno, Julio et al. (2019). "BlockBD: a security pattern to incorporate blockchain in big data ecosystems". In: *Proceedings of the 24th European Conference on Pattern Languages of Programs*, pp. 1–8.

Mühlberger, Roman et al. (2020). “Foundational oracle patterns: Connecting blockchain to the off-chain world”. In: *International Conference on Business Process Management*. Springer, pp. 35–51.

Müller, Marcel, Nadine Ostern, and Michael Rosemann (2020). “Silver bullet for all trust issues? Blockchain-based trust patterns for collaborative business processes”. In: *International Conference on Business Process Management*. Springer, pp. 3–18.

Nakamoto, Satoshi (2008). *Bitcoin: A peer-to-peer electronic cash system*.

Nickerson, Robert C, Upkar Varshney, and Jan Muntermann (2013). “A method for taxonomy development and its application in information systems”. In: *European Journal of Information Systems* 22.3, pp. 336–359.

Oham, Chuka et al. (2018). “B-fica: blockchain based framework for auto-insurance claim and adjudication”. In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, pp. 1171–1180.

Ongaro, Diego and John Ousterhout (2014). “In Search of an Understandable Consensus Algorithm”. In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC’14. Philadelphia, PA: USENIX Association, 305–320. isbn: 9781931971102.

OpenZeppelin (n.d.). *Proxy Upgrade Pattern - OpenZeppelin Docs*. `https://docs.openzeppelin.com/upgrades-plugins/1.x/proxies`. [Accessed 20-Dec-2022].

– (2019). *Proxy Patterns - OpenZeppelin blog*. `https://blog.openzeppelin.com/proxy-patterns/`. [Accessed 20-Dec-2022].

Osses, Felipe, Gastón Márquez, and Hernán Astudillo (2018). “An exploratory study of academic architectural tactics and patterns in microservices: A systematic literature review”. In: *Avances en Ingenieria de Software a Nivel Iberoamericano, CIbSE 2018*.

Owens, Luke et al. (2019). “Inter-family communication in hyperledger sawtooth and its application to a crypto-asset framework”. In: *International Conference on Distributed Computing and Internet Technology*. Springer, pp. 389–401.

Pavlic, Luka, Marjan Hericko, and Vili Podgorelec (2008). “Improving design pattern adoption with ontology-based design pattern repository”. In: *ITI 2008-30th International Conference on Information Technology Interfaces*. IEEE, pp. 649–654.

Podvezko, Valentinas et al. (2009). “Application of AHP technique”. In: *Journal of Business Economics and Management* 2, pp. 181–189.

Pohl, Klaus, Günter Böckle, and Frank Van Der Linden (2005). *Software product line engineering: foundations, principles, and techniques*. Vol. 1. Springer.

Polge, Julien, Jérémy Robert, and Yves Le Traon (2021). “Permissioned blockchain frameworks in the industry: A comparison”. In: *Ict Express* 7.2, pp. 229–233.

Porru, Simone et al. (2017). “Blockchain-oriented software engineering: challenges and new directions”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, pp. 169–171.

Pourpouneh, Mohsen, Kurt Nielsen, and Omri Ross (2020). *Automated Market Makers*. Tech. rep. IFRO Working Paper.

Prewett, Kyleen W, Gregory L Prescott, and Kirk Phillips (2020). “Blockchain adoption is inevitable—Barriers and risks remain”. In: *Journal of Corporate Accounting & Finance* 31.2, pp. 21–28.

Qanbari, Soheil et al. (2016). “IoT design patterns: computational constructs to design, build and engineer edge applications”. In: *2016 IEEE First International Conference on Internet-of-Things Design and Implementation (IoTDI)*. IEEE, pp. 277–282.

Raad, Joe and Christophe Cruz (2015). “A survey on ontology evaluation methods”. In: *Proceedings of the International Conference on Knowledge Engineering and Ontology Development, part of the 7th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*.

Raikwar, Mayank et al. (2018). “A blockchain framework for insurance processes”. In: *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, pp. 1–4.

Rajasekar, Vijay et al. (2020). “Emerging Design Patterns for blockchain Applications.” In: *ICSOFT*, pp. 242–249.

Reddy, Kotha Raj Kumar et al. (2021). “Developing a blockchain framework for the automotive supply chain: A systematic review”. In: *Computers & Industrial Engineering* 157, p. 107334.

Ribalta, Claudia Negri et al. (2021). “Blockchain Mirage or Silver Bullet? A Requirements-driven Comparative Analysis of Business and Developers’ Perceptions in the Accountancy Domain.” In: *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)* 12.1, pp. 85–110.

Rincón, Luisa, Raúl Mazo, and Camille Salinesi (2018). “APPLIES: A framework for evaluAting organization’s motivation and preparation for adopting product lines”. In: *2018 12th International Conference on Research Challenges in Information Science (RCIS)*. IEEE, pp. 1–12.

Saaty, Thomas L (1990). “How to make a decision: the analytic hierarchy process”. In: *European journal of operational research* 48.1, pp. 9–26.

Saleh, Fahad (2021). “Blockchain without waste: Proof-of-stake”. In: *The Review of financial studies* 34.3, pp. 1156–1190.

Schmidt, Douglas C (2006). “Model-driven engineering”. In: *Computer-IEEE Computer Society-* 39.2, p. 25.

Schneider, Fred B (1990). “Implementing fault-tolerant services using the state machine approach: A tutorial”. In: *ACM Computing Surveys (CSUR)* 22.4, pp. 299–319.

Schobbens, Pierre-Yves et al. (2007). “Generic semantics of feature diagrams”. In: *Computer networks* 51.2, pp. 456–479.

Sebastián, Gabriel, Jose A Gallud, and Ricardo Tesoriero (2020). “Code generation using model driven architecture: A systematic mapping study”. In: *Journal of Computer Languages* 56, p. 100935.

Seebacher, Stefan and Maria Maleshkova (2018). “A model-driven approach for the description of blockchain business networks”. In: *Proceedings of the 51st Hawaii International Conference on System Sciences*.

Simmons, Gustavus J (1979). “Symmetric and asymmetric encryption”. In: *ACM Computing Surveys (CSUR)* 11.4, pp. 305–330.

Six, Nicolas (2021). “Decision process for blockchain architectures based on requirements”. In: *CAiSE (Doctoral Consortium)*, pp. 53–61.

Six, Nicolas, Nicolas Herbaut, and Camille Salinesi (2020). “Quelle blockchain choisir? Un outil d’aide à la décision pour guider le choix de technologie Blockchain”. In: *INFORSID 2020*, pp. 135–150.

– (2021a). “BLADE: Un outil d’aide à la décision automatique pour guider le choix de technologie Blockchain”. In: *Revue ouverte d’ingénierie des systèmes d’information* 2.1.

– (2021b). “Harmonica: A Framework for Semi-automated Design and Implementation of blockchain Applications”. In: *INSIGHT* 24.4, pp. 25–27.

– (2022). “Blockchain software patterns for the design of decentralized applications: A systematic literature review”. In: *Blockchain: Research and Applications*, p. 100061.

Six, Nicolas, Andrea Perrichon-Chrétien, and Nicolas Herbaut (2021). “SAIaaS: A Blockchain-based solution for secure artificial intelligence as-a-Service”. In: *The International Conference on Deep Learning, Big Data and Blockchain*. Springer, pp. 67–74.

Six, Nicolas et al. (2020). “A blockchain-based pattern for confidential and pseudo-anonymous contract enforcement”. In: *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, pp. 1965–1970.

Six, Nicolas et al. (2022). “Using Software Product Lines to Create blockchain Products: Application to Supply Chain Traceability”. In: *26th ACM International Systems and Software Product Lines Conference*.

Sousa, Victor Amaral de and Corentin Burnay (2021). “MDE4BBIS: A Framework to Incorporate Model-Driven Engineering in the Development of Blockchain-Based Information Systems”. In: *2021 Third International Conference on blockchain Computing and Applications (BCCA)*. IEEE. doi: `10.1109/bcca53669.2021.9657015`. url: `https://doi.org/10.1109%2Fbcca53669.2021.9657015`.

Suárez-Figueroa, Mari Carmen, Asunción Gómez-Pérez, and Mariano Fernández-López (2012). “The NeOn methodology for ontology engineering”. In: *Ontology engineering in a networked world*. Springer, pp. 9–34.

Sukhwani, Harish et al. (2017). “Performance modeling of PBFT consensus process for permissioned blockchain network (hyperledger fabric)”. In: *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, pp. 253–255.

Syriani, Eugene, Lechanceux Luhunu, and Houari Sahraoui (2018). “Systematic mapping study of template-based code generation”. In: *Computer Languages, Systems & Structures* 52, pp. 43–62.

Taibi, Davide, Valentina Lenarduzzi, and Claus Pahl (2018). “Architectural Patterns for Microservices: A Systematic Mapping Study.” In: *CLOSER*, pp. 221–232.

Tang, Huimin, Yong Shi, and Peiwu Dong (2019). “Public blockchain evaluation using entropy and TOPSIS”. In: *Expert Systems with Applications* 117, pp. 204–210. issn: 09574174. doi: `10.1016/j.eswa.2018.09.048`.

Tešanovic, Aleksandra (2005). “What is a pattern”. In: *Dr. ing. course DT8100 (prev. 78901/45942/DIF8901) Object-oriented Systems*.

Thüm, Thomas et al. (2014). “FeatureIDE: An extensible framework for feature-oriented software development”. In: *Science of Computer Programming* 79, pp. 70–85.

Tonelli, Roberto et al. (2019). “Implementing a microservices system with blockchain smart contracts”. In: *2019 IEEE International Workshop on blockchain Oriented Software Engineering (IWBOSE)*. IEEE, pp. 22–31.

Triantaphyllou, Evangelos (2000). “Multi-criteria decision making methods”. In: *Multi-criteria decision making methods: A comparative study*. Springer, pp. 5–21.

Udokwu, Chibuzor et al. (2021). “Implementation and evaluation of the DAOM framework and support tool for designing blockchain decentralized applications”. In: *International Journal of Information Technology* 13.6, pp. 2245–2263.

Vacca, Anna et al. (2021). “A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges”. In: *Journal of Systems and Software* 174, p. 110891.

Venkatesh, Viswanath et al. (2003). “User acceptance of information technology: Toward a unified view”. In: *MIS quarterly*, pp. 425–478.

Wang, Shuai et al. (2018). “An Overview of Smart Contract: Architecture, Applications, and Future Trends”. In: *2018 IEEE Intelligent Vehicles Symposium (IV)*, pp. 108–113. doi: 10.1109/IVS.2018.8500488.

Washizaki, Hironori et al. (2020). “Landscape of architecture and design patterns for iot systems”. In: *IEEE Internet of Things Journal* 7.10, pp. 10091–10101.

Wei, Yihang (2020). “Blockchain-based Data Traceability Platform Architecture for Supply Chain Management”. In: *2020 IEEE 6th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing,(HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*. IEEE, pp. 77–85.

Wessling, Florian and Volker Gruhn (2018). “Engineering software architectures of blockchain-oriented applications”. In: *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, pp. 45–46.

Wieringa, Roel J (2014). *Design science methodology for information systems and software engineering*. Springer.

Williams, Carrie et al. (2007). “Research methods”. In: *Journal of Business & Economics Research (JBER)* 5.3.

Wöhrer, Maximilian and Uwe Zdun (2018). “Design patterns for smart contracts in the ethereum ecosystem”. In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, pp. 1513–1520.

Wohrer, Maximilian and Uwe Zdun (2018). “Smart contracts: security patterns in the ethereum ecosystem and solidity”. In: *2018 International Workshop on blockchain Oriented Software Engineering (IWBOSE)*. IEEE, pp. 2–8.

Wohrer, Maximilian and Uwe Zdun (2020). “Domain Specific Language for Smart Contract Development”. In: *2020 IEEE International Conference on blockchain and Cryptocurrency (ICBC)*. IEEE. doi: 10.1109/icbc48266.2020.9169399. url: https://doi.org/10.1109%2Ficbc48266.2020.9169399.

Wood, Gavin et al. (2014). “Ethereum: A secure decentralised generalised transaction ledger”. In: *Ethereum project yellow paper*, pp. 1–32.

Worley, Carl R and Anthony Skjellum (2018). “Opportunities, Challenges, and Future Extensions for Smart-Contract Design Patterns”. In: *International Conference on Business Information Systems*. Springer, pp. 264–276.

Wust, Karl and Arthur Gervais (2018). “Do you need a blockchain?” In: *Proceedings - 2018 Crypto Valley Conference on blockchain Technology, CVCBT 2018*, pp. 45–54. doi: 10.1109/CVCBT.2018.00011.

Xu, Xiwei, Ingo Weber, and Mark Staples (2019). *Architecture for blockchain applications*. Springer.

Xu, Xiwei et al. (2018). “A pattern collection for blockchain-based applications”. In: *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, pp. 1–20.

Yang, Xiaohui and Wenjie Li (2020). “A zero-knowledge-proof-based digital identity management scheme in blockchain”. In: *Computers & Security* 99, p. 102050.

Zeadally, Sherali and Jacques Bou Abdo (2019). “Blockchain: Trends and future opportunities”. In: *Internet Technology Letters* 2.6, e130.

Zetzsche, Dirk A et al. (2017). “The ICO Gold Rush: It’s a scam, it’s a bubble, it’s a super challenge for regulators”. In: *University of Luxembourg Law Working Paper* 11, pp. 17–83.

Zhang, He, Muhammad Ali Babar, and Paolo Tell (2011). “Identifying relevant studies in software engineering”. In: *Information and Software Technology* 53.6, pp. 625–637.

Zhang, Peng et al. (2017). *Applying software patterns to address interoperability in blockchain-based healthcare apps*. eprint: arXivpreprintarXiv:1706.03700.

Zhang, Peng et al. (2018). “Blockchain technology use cases in healthcare”. In: *Advances in computers*. Vol. 111. Elsevier, pp. 1–41.

Zou, Weiqin et al. (2019). “Smart contract development: Challenges and opportunities”. In: *IEEE Transactions on Software Engineering* 47.10, pp. 2084–2106.

Zupan, Nejc et al. (2020). “Secure smart contract generation based on petri nets”. In: *Blockchain Technology for Industry 4.0*. Springer, pp. 73–98.

Zwicker, William S (2016). *Introduction to the Theory of Voting*.