



universität
wien

DISSERTATION / DOCTORAL THESIS

Titel der Disseratation / Title of the Doctoral Thesis

„Engineering Blockchain-Based Applications
in the Context of the Ethereum Ecosystem“

verfasst von / submitted by

Dipl.-Ing. Alex Maximilian Wöhrer, Bakk. (FH)

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of
Doktor der technischen Wissenschaften (Dr. techn.)

Wien, 2022 / Vienna, 2022

Studienkennzahl lt. Studienblatt /
degree programme code as it appears on
the student record sheet:

A 786 880

Studienrichtung lt. Studienblatt /
degree programme as it appears on
the student record sheet:

Doktoratsstudium Informatik

Betreut von / Supervisor:

Univ.-Prof. Dr. Uwe Zdun

Vita

Maximilian Wöhrer is a research associate in the Software Architecture Research Group at the Faculty of Computer Science, University of Vienna, Austria. Before that, he worked as a software engineer in different areas. He received a master's degree in computer science in 2009 from the University of Vienna. His research interests include blockchain technology, smart contracts, software architecture, software engineering, and the application of design patterns in the afore mentioned domains.

Acknowledgements

First and foremost, I would like to thank my doctorate supervisor, Professor Uwe Zdun, for his valuable suggestions, expert advice, and continuous support in all matters during my work on this thesis. Through his guidance and support, it was possible for me to successfully navigate the rocky road from a rough vision to the completion of this thesis. Likewise, I would like to thank all co-authors for their help and contributions as well as the anonymous reviewers of my scientific contributions who helped me to revise and improve my work. Furthermore, I would like to express my gratitude to the entire Software Architecture Group, especially my long-time room and research colleague Philipp Paulweber, the secretaries Edisa Redzic and Sylvia Ennsberger, as well as our technician Gerhard Pernecker for all their support and for providing such a warm, productive and pleasant working environment. Last but not least, a big thank you goes to my parents, my twin sister, and my girlfriend who have supported me in every aspect of my life, not only during my studies.

Abstract

Blockchain is more than just the technology behind the cryptocurrency Bitcoin. It can be seen as a driving force that has the potential to transform many domains. The technology is based on various computational and economic concepts to create a fraud-free intermediation platform to efficiently settle transactions between mutually distrusting parties. More specifically, the blockchain provides an infrastructure based on a Peer-to-Peer network to record transactions in a decentralized, transparent, and immutable manner to enable the storage of data or the autonomous execution of programs known as smart contracts.

Driven by these possibilities, numerous companies and organizations have begun to explore blockchain in recent years to find meaningful application areas and develop applications based on the technology. In this context, new ground is being broken in many areas, and accordingly, recommendations for efficient approaches are needed.

In this regard, this thesis addresses technical considerations and methods for building blockchain-based applications. In general, overarching designs and approaches are elaborated that have established themselves as best practices for blockchain-based software solutions from an architectural, development, and implementation perspective. The analysis is done in the scope of different topics in the context of the well-known blockchain platform Ethereum. However, many of the insights gained can also be generalized and applied to other platforms.

First, fundamental architectural decisions and approaches are studied, from which follows that a hybrid architecture consisting of on- and off-chain components offers a good compromise between decentralization and quality attributes such as scalability, privacy, and usability. Furthermore, design patterns for data exchange across blockchain boundaries by means of so-called oracles are elaborated, resulting in four basic oracle patterns related to data flow dimensions (i.e., inbound/outbound and pull/push). Moreover, design patterns for smart contracts are devised and presented by means of 18 concrete patterns that solve problems related to operation, access control, management, and security. Subsequently, an abstract domain-specific language for smart contracts is developed and studied to enable the automatic application of these design patterns via code generation. Along the way, it is shown that abstraction and code generation can be a viable way to formulate smart contracts in order to increase the efficiency, clarity, and flexibility of code while reducing the susceptibility to errors. Last, typical DevOps activities are explored which show that core DevOps

Abstract

concepts and activities are similar to those in other domains, with the difference that more rigorous testing and differentiated deployment practices are required due to the inherent immutability of the blockchain. In summary, this thesis contributes to a better understanding of various topics that are important for the development of blockchain-based applications.

Kurzfassung

Blockchain ist mehr als nur die Technologie hinter der Kryptowährung Bitcoin. Die Technologie kann als treibende Kraft angesehen werden, die das Potenzial besitzt, viele Bereiche zu verändern. Die Blockchain selbst basiert auf verschiedenen rechnerischen und wirtschaftlichen Konzepten, um eine betrugsfreie Vermittlungsplattform zur effizienten Abwicklung von Transaktionen zwischen sich gegenseitig misstrauenden Parteien zu schaffen. Genauer gesagt, bietet die Blockchain eine auf einem Peer-to-Peer-Netzwerk basierende Infrastruktur zur dezentralen, transparenten und unveränderlichen Aufzeichnung von Transaktionen, um die Speicherung von Daten oder die autonome Ausführung von Programmen, sogenannten Smart Contracts, zu ermöglichen.

Angetrieben von diesen Möglichkeiten haben in den letzten Jahren zahlreiche Unternehmen und Organisationen begonnen, sich mit der Blockchain zu beschäftigen, um sinnvolle Anwendungsbereiche zu erkunden und Anwendungen auf Basis der Technologie zu entwickeln. In diesem Zusammenhang wird allerdings in vielen Bereichen Neuland betreten, dementsprechend sind Empfehlungen für effiziente Herangehensweisen gefragt.

Diese Arbeit behandelt in dieser Hinsicht technische Überlegungen und Methoden zur Erstellung von Blockchain-basierten Anwendungen. Dabei werden im Allgemeinen übergreifende Designs und Ansätze elaboriert, die sich als Best Practices für Blockchain-basierte Softwarelösungen aus Architektur-, Entwicklungs- und Implementierungssicht etabliert haben. Die Aufarbeitung geschieht im Rahmen unterschiedlicher Themenkomplexe im Kontext der bekannten Blockchain Plattform Ethereum. Viele der gewonnen Erkenntnisse lassen sich aber auch auf andere Plattformen verallgemeinern und anwenden.

Zunächst werden grundlegende Architekturentscheidungen und -ansätze erarbeitet, aus denen folgt das eine hybride Architektur bestehend aus on- und off-chain Komponenten einen guten Kompromiss zwischen Dezentralisierung und Qualitätsmerkmalen wie Skalierbarkeit, Datenschutz und Benutzerfreundlichkeit bietet. Des Weiteren werden Entwurfsmuster für den Informationsaustausch über Blockchain-Grenzen hinweg mittels sogenannter Orakel ausgearbeitet, woraus sich vier grundlegende Orakelmuster in Bezug auf Datenflussdimensionen (d. h. inbound/outbound und pull/push) ergeben. Außerdem werden Entwurfsmuster für Smart Contracts erarbeitet und anhand von 18 konkreten Mustern präsentiert, die Probleme im Zusammenhang mit dem Betrieb, der Zugangskontrolle, der Verwaltung und der Sicherheit lösen. In weiterer Folge wird eine abstrakte

Kurzfassung

domänenspezifischen Sprache für Smart Contracts entwickelt und studiert um die automatische Anwendung dieser Entwurfsmuster mittels Codegenerierung zu ermöglichen. Dabei wird gezeigt, dass Abstraktion und Codegenerierung ein gangbarer Weg zur Formulierung von Smart Contracts sein können um die Effizienz, Klarheit und Flexibilität von Code zu erhöhen und gleichzeitig die Fehleranfälligkeit zu verringern. Zuletzt werden typische DevOps Aktivitäten untersucht die zeigen, dass die zentralen DevOps-Konzepte und -Aktivitäten denen in anderen Bereichen ähneln, mit dem Unterschied, dass aufgrund der inhärenten Unveränderlichkeit der Blockchain strengere Tests und differenzierte Bereitstellungspraktiken erforderlich sind. Zusammenfassend trägt diese Arbeit zu einem besseren Verständnis verschiedener Themen bei, die für die Entwicklung von Blockchain-basierten Anwendungen von Bedeutung sind.

Contents

Vita	i
Acknowledgements	iii
Abstract	v
Kurzfassung	vii
List of Tables	xiii
List of Figures	xv
Listings	xvii
1. Introduction	1
1.1. Motivation	1
1.2. Problem Statement	2
1.3. Research Methodology	6
1.4. Publications	7
1.5. Thesis Structure	8
2. Background	11
2.1. Blockchains and Cryptocurrencies	11
2.2. Smart Contracts	14
2.3. Ethereum Platform	15
2.3.1. Ethereum Virtual Machine (EVM)	15
2.3.2. Ethereum Smart Contracts	16
2.3.3. Ethereum Programming Languages	16
3. Architecture of Blockchain-Based Applications	19
3.1. Introduction	19
3.2. Related Work	20

Contents

3.3. Research Study Design	21
3.4. Architectural Design of Blockchain-Based Applications	21
3.4.1. Event-Driven Architecture	22
3.4.2. Blockchain as a Multi-Faceted Architectural Component	23
3.4.3. Degrees of Decentralization	23
3.4.4. Transaction Handling	25
3.4.5. Practices for Scalability and Privacy	27
3.4.6. Conceptual Components and Their Interaction	28
3.4.7. Feature Model	36
3.4.8. Smart Contracts and Microservices	36
3.4.9. Blockchain as a Service (BaaS)	38
3.5. Discussion and Threads to Validity	39
3.6. Conclusion	40
4. Oracle Patterns	43
4.1. Introduction	43
4.2. Background	45
4.3. Related Work	46
4.4. Patterns	47
4.4.1. Inbound Oracle Patterns	48
4.4.2. Outbound Oracle Patterns	51
4.5. Use Cases	53
4.6. Analysis of Performance and Transaction Fees	56
4.7. Discussion and Threats to Validity	59
4.8. Conclusion	60
5. Smart Contract Patterns	63
5.1. Introduction	63
5.2. Related Work	64
5.3. Research Study Design	65
5.4. Patterns	66
5.4.1. Action and Control Patterns	67
5.4.2. Authorization Patterns	77
5.4.3. Lifecycle Patterns	80
5.4.4. Maintenance Patterns	83
5.4.5. Security Patterns	90
5.5. Discussion	96

5.6. Conclusion	98
6. Domain Specific Language for Smart Contract Development	101
6.1. Introduction	101
6.2. Background	103
6.2.1. Contract Stages	103
6.2.2. Contract Building Blocks	103
6.3. Research Study Design	106
6.4. Contract Modeling Language (CML)	106
6.4.1. Language Characteristics	106
6.4.2. Type System	107
6.4.3. Clause Structure	108
6.4.4. CML by Example: Simple Open Auction	109
6.5. Solidity Code Generation	110
6.5.1. CML to Solidity Mapping	110
6.5.2. Code Generation Idioms	113
6.6. Evaluation	121
6.7. Discussion	122
6.8. Related Work	123
6.9. Conclusion	124
7. Blockchain DevOps	127
7.1. Introduction	127
7.2. Related Work	128
7.3. Research Study Design	129
7.4. DevOps for Blockchain Smart Contracts	130
7.4.1. Preliminary Considerations	130
7.4.2. Continuous Integration (CI)	132
7.4.3. Continuous Delivery (CD)	140
7.4.4. CI/CD Overview	144
7.5. Discussion and Threats to Validity	145
7.6. Conclusion	146
8. Conclusions and Future Work	149
8.1. Research Questions Revisited	149
8.2. Limitations and Threats to Validity	153
8.3. Future Work	154

Contents

Bibliography	157
A. Appendix	183
A.1. DSL for Smart Contracts - Contract Modeling Language (CML)	183
A.2. DevOps for Ethereum Smart Contracts	188

List of Tables

4.1.	An overview of the four oracle types.	44
4.2.	A summary of statistics on time and costs for oracle invocations.	59
5.1.	An overview of pattern usage examples.	96
5.2.	An overview of smart contract design patterns.	97
6.1.	An overview of basic contractual building blocks.	104
6.2.	An overview of construct mappings between CML and Solidity.	111
6.3.	A complexity comparison between CML and generated Solidity representation. .	122

List of Figures

2.1.	A high-level overview of the basic principles and operation of a blockchain. . . .	12
3.1.	A comparative overview of a traditional, hybrid, and DApp architecture.	24
3.2.	An overview of transaction handling options.	26
3.3.	A typical component structure pattern of a DApp utilizing a backend.	28
3.4.	A typical component structure pattern of an enterprise grade blockchain integration.	29
3.5.	A sequence diagram showing key components for processing a transaction. . . .	34
3.6.	A feature model for a blockchain-based application.	37
4.1.	A conceptual overview of the oracle data flow partitioning.	47
4.2.	An overview of the oracle types and conceptual structural components.	48
4.3.	A sequence diagram showing the Pull-Based Inbound Oracle.	49
4.4.	A sequence diagram showing the Push-Based Inbound Oracle.	51
4.5.	A sequence diagram showing the Pull-Based Outbound Oracle.	52
4.6.	A sequence diagram showing the Push-Based Outbound Oracle.	53
4.7.	A supply chain process employing oracles.	54
4.8.	The procedure for oracle-based creditworthiness verification.	55
4.9.	The procedure for oracle-based tracking of goods.	56
4.10.	The schematic process for measuring latency.	58
4.11.	The performance plots for the four oracle implementations.	59
5.1.	An overview of the conducted MLR process.	66
6.1.	An overview of contract stages.	103
6.2.	An exemplary contract for the sale of goods.	105
6.3.	A conceptual breakdown of covenant clause components.	108
6.4.	The structure of a CML clause declaration.	109
6.5.	The CML code generation process.	111
7.1.	An overview of test types for smart contracts and blockchain-based software. . .	135
7.2.	An illustration of the test structure for Solidity and JavaScript tests.	138

List of Figures

7.3. An overview of DevOps stages for smart contracts.	144
A.1. An overview of the Xtext DSL Framework.	184

Listings

2.1. A simple deposit contract.	17
5.1. Application of the Checks-Effects-Interaction pattern within a function.	68
5.2. An example of an insecure withdrawal function.	68
5.3. An auction contract with a push payment.	70
5.4. An auction contract with a pull payment.	71
5.5. A contract implementing a state machine.	72
5.6. A contract implementing a commit and reveal scheme.	75
5.7. An oracle contract that allows to request data from outside the blockchain.	77
5.8. An oracle consumer contract with a callback.	77
5.9. A simple contract to track the ownership of a contract.	79
5.10. A contract implementing various function access restrictions.	80
5.11. A contract that provides its creator with the ability to destroy it.	81
5.12. A contract interface with time period based deprecation.	83
5.13. A contract interface with time block number based deprecation.	83
5.14. A contract to separate data storage.	85
5.15. A contract to separate the logic.	85
5.16. A satellite contract encapsulates certain contract functionalities.	86
5.17. A base contract referring to a satellite contract.	87
5.18. A register contract to store the latest version of a contract.	88
5.19. A relay contract to forward data and calls.	89
5.20. A contract implementing an emergency stop.	91
5.21. A contract that delays the withdrawal of funds deliberately.	92
5.22. A contract with a rate limit that avoids repetitive function execution.	93
5.23. A contract implementing a mutex pattern.	94
5.24. A contract implementing a balance limit.	96
6.1. A CML contract for a simple open auction.	110
6.2. An excerpt of the generated Solidity contract from Listing 6.1	113
6.3. A CML contract with “Ownership” and “Pullpayment” annotations.	114

Listings

6.4.	An excerpt of the generated Solidity contract from Listing 6.3.	115
6.5.	A CML contract with “SafeMath” annotation.	116
6.6.	An excerpt of the generated Solidity contract from Listing 6.5.	116
6.7.	A CML contract with “FixedPointArithmetic” annotation.	117
6.8.	An excerpt of the generated Solidity contract from Listing 6.7.	118
6.9.	A CML contract using a collection.	119
6.10.	An excerpt of the generated Solidity contract from Listing 6.9.	120
6.11.	An excerpt of the generated Solidity collection library code.	121
A.1.	DSL specification in Xtext for the Contract Modeling Language (CML).	188
A.2.	GitLab CI/CD configuration for Ethereum smart contracts.	191

1. Introduction

Advances in the development of information and communication technologies have enabled a number of innovations in recent decades that have had a holistic impact on society. In particular, developments related to the Internet have spawned a large number of disruptive innovations. That is, innovations with the potential to fundamentally challenge traditional ways of doing business. Social media, sharing economy, and cryptocurrencies can be cited as examples in this context, with cryptocurrencies attracting considerable attention in various contexts in recent years.

Cryptocurrencies have now reached a market capitalization of almost 3 trillion US dollars at their peak in 2021 [1] and are increasingly accepted as a means of payment in different places. The distinctive feature of cryptocurrencies is that transactions as well as the general management of the currencies can be carried out securely via distributed computer networks and without the influence of central institutions. Traditional intermediaries, such as banks, are thus obsolete. For this very reason, the underlying technology is now often considered the real innovative breakthrough: the blockchain.

Blockchain offers a promising platform technology for implementing new decentralized software architectures in which distributed components can make agreements on shared system states without relying on a central point of trust. It provides an infrastructure based on a Peer-to-Peer (P2P) network for the decentralized, transparent and immutable recording of transactions to enable the storage of data or the autonomous execution of programs, so-called smart contracts.

Due to these characteristics, blockchain has the potential to change a whole range of areas far beyond the field of digital currencies and to be a disruptive innovation. Driven by this outlook, numerous companies and organizations have started to engage with blockchain in recent years to explore meaningful application areas and develop applications based on blockchain technology.

1.1. Motivation

The introduction of new technologies is fraught with challenges as they break new ground in many areas. This uncharted territory extends to many different areas with regard to blockchain. The goal of this thesis is to provide guidance for the realization of blockchain-based applications from a technical perspective, covering several relevant topics in this context. This includes the identification of overarching designs and approaches that have been established as best practices for

1. Introduction

blockchain-based software solutions from an architecture, development and implementation perspective. Our efforts in this area are mainly related to the very established and well-known platform Ethereum, but most findings can also be applied to other platforms. Based on the collected insights, a compendium of guiding principles for common problems is formed and archetypes or tools for specific problem domains are developed to foster the creation of blockchain-based software.

1.2. Problem Statement

In this thesis, a holistic approach is taken to address some of the issues that exist in the development of blockchain-based applications. In the following, particular research problems and corresponding research questions are discussed in more detail. Regarding the research questions, we have formulated a general research question for each problem domain that roughly outlines the scope of the research, which is then supplemented by further detailed questions in that context.

Lack of Architectural Design Guidance for Blockchain Applications

Looking at the maturity of blockchain technologies in practical applications, it is fair to say that the adoption and diffusion of the technology is arguably still evolving. Having gone through the hype cycle for blockchains as an emerging technology, we are currently in a phase of consolidation where practical applications provide insights into the pros and cons of using blockchain technology in real-world scenarios. Accordingly, the optimal integration and design of software solutions that integrate blockchains is not entirely clear. From a software architecture perspective, there is only a limited systematic and holistic approach to the development of blockchain-based software systems. In this context, it would be helpful to have a set of basic architectural design decisions for blockchain integration that could be derived from innovative and already realized applications. Looking at blockchain as part of a larger system, it is likely that certain application scenarios and architectural patterns will be more common and prove more beneficial than others.

To explore this architectural design space and possible solution strategies, we are interested in fundamental architectural design decisions, design options for these decisions, and involved conceptual components in the design of blockchain-based software architectures. In this context, we ask the following questions:

Research Question 1 (RQ 1)

What design decisions, design options, and conceptual components need to be considered for blockchain-based applications?

RQ 1.1 What are the (key) architectural design decisions for blockchain-based

applications?

RQ 1.2 *What are possible design options regarding these decisions and the associated (best) practices?*

RQ 1.3 *Which conceptual components are relevant in the architectural design and what are their relations?*

Lack of Design Guidance Connecting Blockchain to the Off-Chain World

Blockchain technologies provide a secure process execution infrastructure that can be used to formulate and execute business logic in the form of smart contracts. However, smart contracts cannot invoke software components that reside outside the on-chain execution environment. Rather, they can only perform operations within the blockchain environment and be accessed via blockchain transactions. This isolates them from the physical world and other applications. But many process and information flows require communication across blockchain boundaries with external applications and real-world objects. This requires the use of so-called oracles, i.e., software elements that form a bridge between the on-chain execution environment and the off-chain world. Yet, despite their high relevance, a thorough investigation of oracle designs and their characteristics is still lacking in the literature.

This research gap leads us to pose the following research questions in order to elaborate basic blockchain oracle patterns:

Research Question 2 (RQ 2)

What (fundamental) patterns exist to implement blockchain oracles and how do they differ regarding cost and performance?

RQ 2.1 *What are the fundamental design patterns for implementing blockchain oracles?*

RQ 2.2 *What are the characteristics of these regarding cost and performance?*

Lack of Smart Contract Design Patterns

Smart contract development for blockchains is a challenging task so far. It requires the use of unconventional programming paradigms due to the inherent design and characteristics of blockchain-based program execution. Or to put it another way, the development process requires a different technical approach than what most web and mobile developers are used to. Unlike modern program-

1. Introduction

ming languages, which are based on a long evolution and thus contain many helpful abstractions and data types, blockchain-based programming languages are still relatively young. This means that smart contract developers lack these aids and are therefore often responsible for internally organizing and manipulating data at a deeper level which is inherently more error-prone. Complicating matters further is the fact that errors in implemented contracts can have serious consequences due to the direct coupling of contract code and financial assets. Fortunately, many implementation issues and vulnerabilities associated with smart contract development can be avoided by following best practices. However, much practical knowledge in this area is scattered across development-specific literature and practitioner reports, so the information is often not very structured or comprehensive. This means there is a lack of an informative compendium that contains a solid foundation of established and proven design and code patterns that facilitate writing functional and error-free code.

In view of these problems, it is beneficial to have a foundation of established design and coding guidelines. This leads us to consider the following research questions:

Research Question 3 (RQ 3)

What are common design patterns and Solidity coding practices for Ethereum smart contracts?

RQ 3.1 Which design patterns commonly appear in the Ethereum ecosystem and what problems do they solve?

RQ 3.2 How do these design patterns map to Solidity coding practices?

Lack of Tools for Secure-By-Design Smart Contracts

In view of the problems mentioned in the previous point, a methodology is desirable which prevents implementation errors from occurring when transferring business processes to smart contracts. This first requires suitable abstractions and descriptive means for contract representation, which can then be transferred to an implementation through code generation while adhering to common design patterns. Such an approach can reduce the complexity of the design, resulting in increased understandability and reduced susceptibility to errors. However, such abstract descriptions and their implementation including the generation of executable code are anything but easy to realize and are still the subject of research.

Following the above approach, leads us to consider the following research questions:

Research Question 4 (RQ 4)

What might a secure-by-design approach to smart contracts look like that starts from a higher level of abstraction and generates an implementation leveraging design patterns?

RQ 4.1 How and in how far is it possible to bring the abstraction level of smart contracts closer to the contract domain?

RQ 4.2 Can higher abstraction levels in combination with code generation (considering platform-specific programming idioms) reduce the risk of smart contract errors?

Lack of DevOps Guidance for Smart Contracts

As blockchain continues to evolve and spread, the technology is increasingly finding its way into enterprise software development. It is in this area that many companies are already embracing established practices such as DevOps. DevOps is an operational philosophy that combines practices and tools that increase an organization's ability to deliver applications and services faster and with better quality than would be possible with traditional software development and infrastructure management processes. Accordingly, it makes sense to extend the DevOps approach to the blockchain space to accelerate the overall pace of software development and delivery while improving software quality. This translates into improved overall productivity, resulting in lower total cost of ownership for enterprises. However, there is currently a lack of guidance for a structured DevOps approach and a breakdown of the specifics in the context of blockchain-based software development.

This problem leads us to consider the following research questions:

Research Question 5 (RQ 5)

What does a typical DevOps approach for blockchain-based applications look like and what are the differences compared to a DevOps approach for traditional software projects?

RQ 5.1 What are typical stages and activities in a DevOps approach for blockchain-based applications?

RQ 5.2 What are the particularities of using DevOps in blockchain-based software development?

1.3. Research Methodology

The research conducted in this thesis to answer the above research questions is based on various research methods or principles. A brief insight into relevant research methods is provided below, a more detailed description as well as the modalities of application will follow in due course later in the various chapters of this thesis.

(Systematic) Literature Review

Literature review and survey are methods that establish the basis for scientific inquiries and enable research synthesis [2]. Research synthesis refers to the combination of results from multiple research studies. The general goal is to make the results on a particular topic from several different studies generalizable and applicable. Furthermore, new insights can be gained by comparing these results.

One type of literature review that relies on repeatable analysis methods is the Systematic Literature Review (SLR). It is an independent scientific method for identifying and evaluating all relevant literature on a topic in order to draw conclusions about a research question under investigation [3]. In this context, a methodological formal approach is used to reduce bias and increase the reliability of the selected literature.

Multivocal Literature Review

A Multivocal Literature Review (MLR) is a form of SLR which includes “gray” literature (e.g., blogs, videos, and web pages) in addition to published “white” literature (e.g., academic journals, and conference papers) [4]. The approach allows us to take into account knowledge from practitioners outside the academic front in order to gain valuable application-oriented insights.

Grounded Theory

Grounded Theory (GT) is a qualitative strategy of inquiry in which the researcher derives a general, abstract theory of a process, action, or interaction grounded in the views of participants in a study [5]. GT methods consist of systematic, yet flexible guidelines for collecting and analyzing qualitative data to construct theories from the data themselves [6]. Although the research conducted in the context of this thesis is not directly based on GT, we have used principles of the GT method in order to observe and analyze collected data.

Design Science Research

Our approach to the development of software tools in the scope of this thesis is based on the methodology of design science research, where “knowledge and understanding of a problem

domain and its solution are achieved in the building and application of the designed artifact” [7]. In other words, a possible solution for a given problem domain is envisioned and implemented as an innovative prototype. Then the implementation is evaluated to see if the problem has been solved. If the problem is only partially solved or not solved at all, new concepts have to be developed and implemented again until an adequate solution is found. Accordingly, the approach to generating progress or new knowledge is driven by practical implementation where designed artifacts are both useful and fundamental to understanding the problem [8]. This approach enables the exploration of new technologies and the progression of accepted practices, in the absence of a solid theoretical foundation, through the design and evaluation of new systems and their components [9].

1.4. Publications

This thesis consists of papers previously published in scientific conferences, workshops and journals. It should be noted that given the constant and rapid development in many of the areas covered, some of the content needs to be considered in a new light and may no longer represent the state of the art. In this respect, time related statements in the texts are to be understood in relation to the publication date of the respective underlying papers and may not reflect the current state of affairs.

The following is a detailed list of the publications included in this thesis:

- Paper A:** M. Wohrer and U. Zdun, “Smart contracts: Security Patterns in the Ethereum Ecosystem and Solidity,” in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, vol. 2018-Janua, IEEE, Mar. 2018, pp. 2–8, ISBN: 978-1-5386-5986-1. DOI: 10.1109/IWBOSE.2018.8327565. [Online]. Available: <https://ieeexplore.ieee.org/document/8327565/>
- Paper B:** M. Wohrer and U. Zdun, “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, IEEE, Jul. 2018, pp. 1513–1520, ISBN: 978-1-5386-7975-3. DOI: 10.1109/Cybermatics_2018.2018.00255. [Online]. Available: <https://ieeexplore.ieee.org/document/8726782/>
- Paper C:** M. Wohrer and U. Zdun, “Domain Specific Language for Smart Contract Development,” in *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, IEEE, May 2020, pp. 1–9, ISBN: 978-1-7281-6680-3. DOI: 10.1109/ICBC48266.2020.9169399. [Online]. Available: <https://ieeexplore.ieee.org/document/9169399/>

1. Introduction

- Paper D:** M. Wohrer and U. Zdun, “From Domain-Specific Language to Code: Smart Contracts and the Application of Design Patterns,” *IEEE Software*, vol. 37, no. 5, pp. 37–42, Sep. 2020, ISSN: 0740-7459. DOI: 10.1109/MS.2020.2993470. [Online]. Available: <https://ieeexplore.ieee.org/document/9089272>
- Paper E:** R. Mühlberger, S. Bachhofner, E. Castelló Ferrer, C. Di Ciccio, I. Weber, M. Wöhler, and U. Zdun, “Foundational Oracle Patterns: Connecting Blockchain to the Off-Chain World,” in *Lecture Notes in Business Information Processing*, vol. 393 LNBIP, 2020, pp. 35–51, ISBN: 9783030587789. DOI: 10.1007/978-3-030-58779-6_3. arXiv: 2007.14946. [Online]. Available: https://link.springer.com/10.1007/978-3-030-58779-6_3
- Paper F:** M. Wohrer and U. Zdun, “Architectural Design Decisions for Blockchain-Based Applications,” in *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, IEEE, May 2021, pp. 1–5, ISBN: 978-1-6654-3578-9. DOI: 10.1109/ICBC51069.2021.9461109. [Online]. Available: <https://ieeexplore.ieee.org/document/9461109/>
- Paper G:** M. Wohrer, U. Zdun, and S. Rinderle-Ma, “Architecture Design of Blockchain-Based Applications,” in *2021 3rd Conference on Blockchain Research and Applications for Innovative Networks and Services (BRAINS)*, IEEE, Sep. 2021, pp. 173–180, ISBN: 978-1-6654-3924-4. DOI: 10.1109/BRAINS52497.2021.9569813. [Online]. Available: <https://ieeexplore.ieee.org/document/9569813/>
- Paper H:** M. Wohrer and U. Zdun, “DevOps for Ethereum Blockchain Smart Contracts,” in *2021 IEEE International Conference on Blockchain (Blockchain)*, IEEE, Dec. 2021, pp. 244–251, ISBN: 978-1-6654-1760-0. DOI: 10.1109/Blockchain53845.2021.00040. [Online]. Available: <https://ieeexplore.ieee.org/document/9680513/>

1.5. Thesis Structure

Chapter 2, *Background*, provides an overview of the terms and topics relevant to this thesis. In particular, basic concepts of blockchain technology and smart contracts are explained and an introduction to the Ethereum ecosystem is given.

This chapter contains contents of the following paper: Paper A

Chapter 3, *Architecture of Blockchain-Based Applications*, systematically explores the architectural design space and possible solution strategies for designing blockchain-based software solutions.

More specifically, architectural design decisions and decision options are described in terms of patterns and practices for applications with different degrees of decentralization, including conceptual components, and possible relationships between them.

This chapter is based on the contents on the following papers: Paper F, Paper G

Chapter 4, *Oracle Patterns*, discusses foundational blockchain oracle patterns. In this context, oracle patterns are characterized and categorized based on fundamental dimensions, described in a structured manner, and discussed in the context of an implementation based on use cases.

This chapter is based on the contents of the following paper: Paper E,

Chapter 5, *Smart Contract Patterns*, presents design patterns that address common issues and vulnerabilities related to smart contract implementation. More specifically, design patterns for the Ethereum ecosystem, organized by operational scope, are explained along with their application context and sample code.

This chapter is based on the contents of the following papers: Paper A, Paper B

Chapter 6, *Domain Specific Language for Smart Contract Development*, focuses on the design and study of a domain-specific smart contract language. Specifically, a language based on a higher level of abstraction is described that can be automatically transformed into an implementation through code generation, following common idioms and design patterns.

This chapter is based on the contents of the following papers: Paper C, Paper D

Chapter 7, *Blockchain DevOps*, explains current practices and solution approaches for an efficient blockchain-oriented DevOps approach. In this context, procedural steps and related activities are elaborated according to the main phases of Continuous Integration (CI) and Continuous Delivery/Deployment (CD).

This chapter is based on the contents of the following paper: Paper H

Chapter 8, *Conclusions and Future Work*, gives a summary of the main contributions of this thesis, discusses the limitations, and presents open challenges to be addressed in future work.

2. Background

This chapter provides an overview of the terms and topics relevant to this thesis. First, the basic operating principles of a blockchain are explained, followed by an explanation of smart contracts and finally a description of the Ethereum platform.

2.1. Blockchains and Cryptocurrencies

A universal definition of blockchain is not easy because there are many ways to describe a blockchain, depending on which perspective one takes. Blockchain can be seen as a novel architecture built on existing technologies, an immutable database and distributed shared ledger, a trustless secure transaction system, a cryptocurrency, possibly even as a precursor to the next generation of the Internet and the Web. All these declarations are valid views of one and the same thing. In the abstract, blockchain can be understood as a new architectural paradigm and trust protocol. At its core, blockchain offers a P2P transaction model that allows parties to conduct tamper-proof and cryptographically secured transactions. A concept that forms the foundation of most decentralized applications and cryptocurrencies. Looking at the latter in more detail, the importance of the concept can be understood particularly well. Any exchange for assets relies on some kind of record-keeping to track the balances of individuals and to verify that transactions are covered by sufficient funds. This principle also applies to cryptocurrencies, which rely on maintaining a shared global ledger between nodes of a P2P network. The ledger is organized as a hash-chain of blocks, the so-called blockchain, wherein each block contains a set of facts (transactions). Network nodes replicate the blockchain and listen for new transaction requests on the network, which they collect and verify, to form a new block. This process is executed in relative synchrony, as each node proposes its own block of new transactions, to append and update the chain. In order to choose a globally accepted successor block, the nodes agree to follow a distributed consensus protocol. The protocol defines a set of rules to select a winner among all nodes that propose a new block. The most widely known and used consensus protocol is called Proof-of-Work (PoW) consensus. It works by probabilistically determining a winner through the solution of a difficult cryptographic puzzle. Once a winning node is determined, that node broadcasts its block to all other nodes, who check that the proposed block fulfills predefined validity constraints. If the check is passed, the nodes update their blockchain to

2. Background

include the newly proposed block, and move on to work on the next block. This line of action makes it possible to create a distributed trustless consensus system, which is the major breakthrough of the blockchain technology. Bitcoin, the first real-world implementation of a blockchain-based cryptocurrency system, uses a blockchain as an open, distributed ledger that resides on a large, decentralized, publicly accessible network to record and validate the exchange of its tokens (Bitcoins).

Following Bitcoin, we will now take a closer look at the general operating principle of a blockchain. Figure 2.1 serves to illustrate the high-level process of transaction processing, which can be described step-by-step as follows:

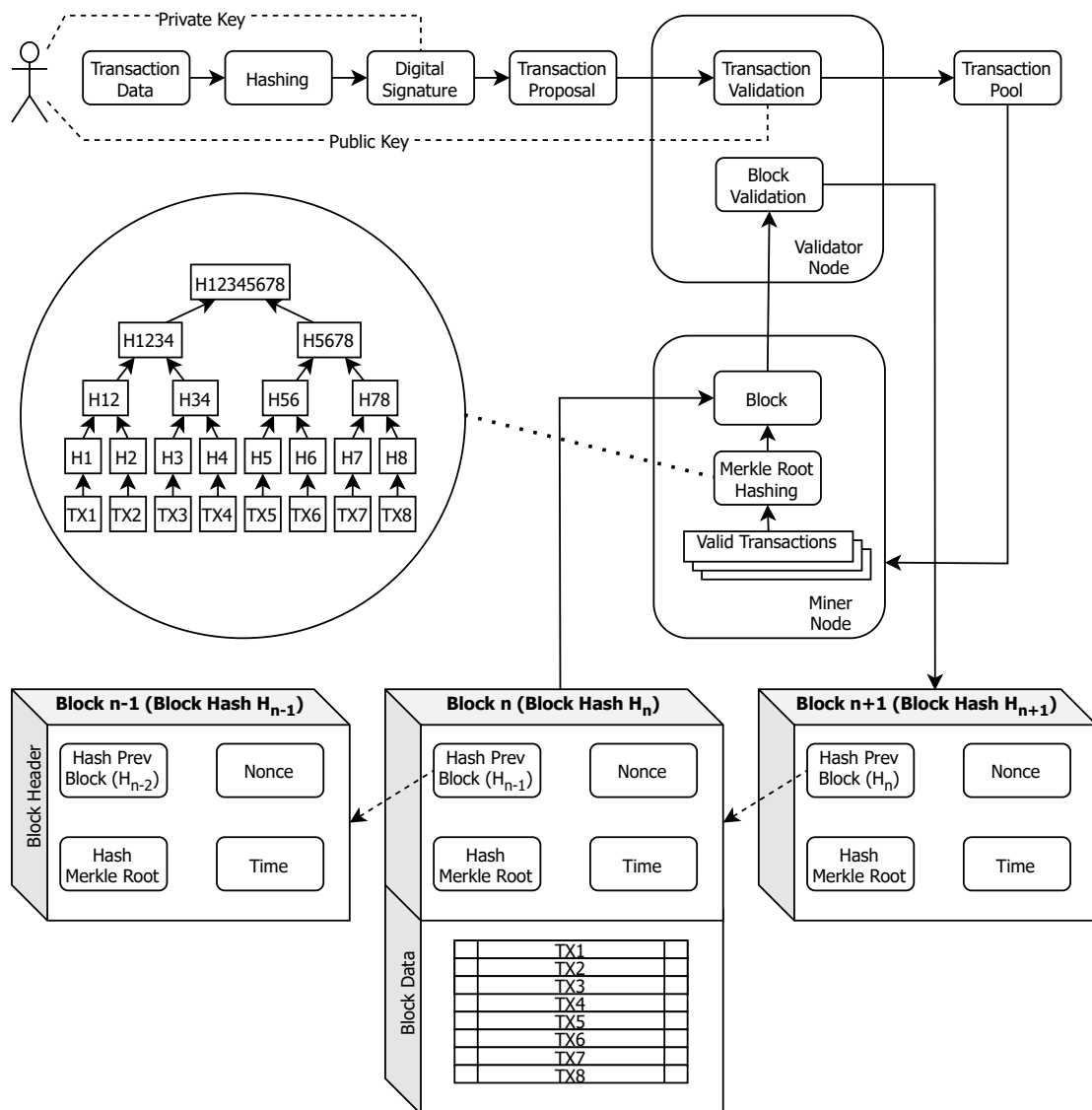


Figure 2.1.: A high-level overview of the basic principles and operation of a blockchain.

2.1. Blockchains and Cryptocurrencies

- a) *Transaction proposal*: A blockchain user creates a transaction (e.g., to transfer financial value to another party or to interact with a smart contract). The transaction data is first hashed for later data integrity verification, then digitally signed with the users's private key, and finally sent to the blockchain P2P network.
- b) *Transaction validation*: Once the transaction has been submitted to the blockchain network, typically a full node of the P2P network, as the recipient, validates the transaction according to the business and technical rules set by the blockchain network. Validation of the transaction includes verifying user authentication by decrypting the digital signature with the user's public key and verifying data integrity by hashing the transaction data and comparing it with the previously decrypted signature. In case the transaction is valid, the node adds the transaction to the transaction pool and forwards it to peers on the network.
- c) *Block creation*: Transactions in the transaction pool are combined into a block at regular intervals (e.g., every 10 minutes for Bitcoin). Network nodes, called mining nodes or miners, collect all valid transactions from the transaction pool and create candidate blocks. A candidate block bundles the most recent valid transactions into a block structure based on block specifications. For each transaction in the bundle, a cryptographic hash of the transaction data is created. These hash values are then combined in pairs and hashed again until a single hash value is obtained. This value is known as the Merkle (tree) root hash value. In addition, the miner node looks for the most recent block in the blockchain and, as a reference to that block, inserts its hash in the block header of the candidate block.
- d) *Block mining*: Once a block candidate is created, mining nodes compete against each other to add new blocks and receive a reward for their efforts. This process is called mining. The winner is determined by a consensus mechanism. In blockchain systems such as Bitcoin or Ethereum, a PoW consensus mechanism is used for mining. Here, the blockchain has an additional field per block, the "nonce" field (abbreviation for "number used once"), which holds a number that is only used once in the respective context. The nodes in the competition, known as miners, must find a nonce number that, in combination with the data in the block, produces a hash that is subject to certain conditions that are difficult to fulfill. For example, one possible condition is that the resulting block hash is less than a target number, or that the hash contains a fixed number of leading zeros. So, to find the right number, miners must try different numbers for the nonce field and compute the hash until a number is found that meets the requirements for the target hash. In practice, each random number has an equal chance of winning the race, so one can start by iterating through a loop in a brute-force manner (e.g., for Bitcoin from 1 to 2^{32}) until one finds such a nonce. However, it requires a tremendous hashing power to find such a nonce in a timely manner. The challenging

2. Background

condition, called difficulty, can be dynamically adjusted over time to account for mining hardware advancements in order to maintain constant block generation time intervals. The effort of finding a correct nonce increases exponentially the smaller the target number.

- e) *Block chaining*: The first node to find a matching nonce notifies the rest of the network to verify the new block. Validation nodes then verify the validity of the block by checking the correctness of the block's hash value, its timestamp, its height (i.e. position within the blockchain) and size, the hash value of the previous block, and the validity of all the block's transactions. Here, validation only requires the computation of a single hash, namely the number specified by the miner in the nonce field and the block's data. So solving the PoW task is challenging, but validation takes only a fraction of a second. If the verification is successful, the peers add the new block to their copy of the blockchain, stop the mining work at hand, and move on to the next block. Furthermore, the winning miner can claim his reward. Once the new block is verified and accepted by the majority of miners in the network, it is accepted and forms the new head of the chain. Should someone want to tamper with the blockchain, for example to change a confirmed transaction in a specific block, they would need to determine a nonce not only for that block, but for all chained blocks following the changed transaction. Thus, a fraudulent miner would be much slower than a miner that only confirms the current block, therefore such an attempt would not be fruitful.

In summary, the fundamental concept of Bitcoin is to create a shared public ledger (longest PoW chain), which verifies and immutably records all transactions through a decentralized P2P computer network and a consensus mechanism with computational proof. However, a blockchain may not only record financial transactions, but also allow specially designed computer programs to be executed and stored as part of transactions. This is based on the idea that a blockchain may be thought of as a shared database, which might allow any kind of data to be recorded. In particular, the data recorded for a transaction may be the source code of a computer program and the validity check for an asset transaction may be replaced by executing that program. Taken this concept one step further implies, that a blockchain may be used to formulate mutual agreements through the creation of programmed applications or so-called smart contracts.

2.2. Smart Contracts

Smart contracts are computer programs that can facilitate, verify, and enforce the negotiation and execution of legal contracts. They are executed through blockchain transactions, interact with cryptocurrencies, and have interfaces to handle input from contract participants. When run on the blockchain, a smart contract becomes an autonomous entity that automatically executes specific

actions when certain conditions are met. Because smart contracts run on the blockchain, they run exactly as programmed, without any possibility of censorship, downtime, fraud or third party interference [18]. Smart contracts are designed to provide more contractual certainty compared to traditional contract law while reducing transaction costs. Their use enables new business models and promises improved contract processing, as manual contract processes with possible scope for interpretation and documentation effort can be avoided. This allows to increase the speed and quality of basic business processes. There are now a number of platforms that enable the operation of smart contracts, such as Ethereum [18], Solana [19], Polkatod [20], Ergo [21], Algorand [22], Cardano [23], to name the more important ones. Of these, Ethereum, as the world's first smart contract platform, is the most well-known and widely used platform today.

2.3. Ethereum Platform

Ethereum is a public blockchain based distributed computing platform, that offers smart contract functionality. It provides a decentralized virtual machine as runtime environment to execute smart contracts, known as Ethereum Virtual Machine (EVM).

2.3.1. Ethereum Virtual Machine (EVM)

The EVM handles the computation and state of contracts and is build on a stack-based language with a predefined set of instructions (opcodes) and corresponding arguments [24]. So, in essence, a contract is simply a series of opcode statements, which are sequentially executed by the EVM. The EVM can be thought of as a global decentralized computer on which all smart contracts run. Although it behaves like one giant computer, it is rather a network of smaller discrete machines in constant communication. All transactions, handling the execution of smart contracts, are local on each node of the network and processed in relative synchrony. Each node validates and groups the transactions sent from users into blocks, and tries to append them to the blockchain in order to collect an associated reward. This process, as explained earlier, is called mining, and the participating nodes are called miners. To ensure a proper resource handling of the EVM, every instruction the EVM executes has a cost associated with it, measured in units of gas. Operations that require more computational resources cost more gas, than operations that require fewer computational resources. This ensures that the system is not jammed up by denial-of-service attacks, where users try to overwhelm the network with time-consuming computations. Therefore, the purpose of gas is twofold. It encourages developers to write quality applications by avoiding wasteful code, and ensures at the same time that miners, executing the requested operations, are compensated for their contributed resources. When it comes to paying for gas, a transaction fee is charged in small amounts of Ether, the built-in digital currency of the Ethereum network, and the token with which

2. Background

miners are rewarded for executing transactions and producing blocks. Ultimately, Ether is the fuel for operating the Ethereum platform.

2.3.2. Ethereum Smart Contracts

Smart contracts are applications which are deployed on the blockchain ledger and execute autonomously as part of transaction validation. To deploy a smart contract in Ethereum, a special creation transaction is executed, which introduces a contract to the blockchain. During this procedure the contract is assigned an unique address, in form of a 160-bit identifier, and its code is uploaded to the blockchain. Once successfully created, a smart contract consists of a contract address, a contract balance, predefined executable code, and a state. Different parties can then interact with a specific contract by sending contract-invoking transactions to a known contract address. These may trigger any number of actions as a result, such as reading and updating the contract state, interacting and executing other contracts, or transferring value to others. A contract-invoking transaction must include the execution fee and may also include a transfer of Ether from the caller to the contract. Additionally, it may also define input data for the invocation of a function. Once a transaction is accepted, all network participants execute the contract code, taking into account the current state of the blockchain and the transaction data as input. The network then agrees on the output and the next state of the contract by participating in the consensus protocol. Thus, on a conceptual level, Ethereum can be viewed as a transaction-based state machine, where its state is updated after every transaction.

2.3.3. Ethereum Programming Languages

Smart contracts in Ethereum are usually written in higher level languages and are then compiled to EVM bytecode. Such higher level languages are LLL (Low-level Lisp-like Language) [25], Serpent (a Python-like language) [26], Viper (a Python-like language) [27], and Solidity (a JavaScript-like language) [28]. LLL and Serpent were developed in the early stages of the platform, while Viper is currently under development, and is intended to replace Serpent. The most prominent and widely adopted language is Solidity.

Solidity

Solidity is a high-level Turing-complete programming language with a JavaScript similar syntax, being statically typed, supporting inheritance and polymorphism, as well as libraries and complex user-defined types.

When using Solidity for contract development, contracts are structured similar to classes in object oriented programming languages. Contract code consists of variables and functions which read and modify these, like in traditional imperative programming.

Solidity

```

1 pragma solidity ^0.4.17;
2
3 contract SimpleDeposit {
4     mapping (address => uint) balances;
5
6     event LogDepositMade(address from, uint amount);
7
8     modifier minAmount(uint amount) {
9         require(msg.value >= amount);
10        _;
11    }
12
13    function SimpleDeposit() public payable {
14        balances[msg.sender] = msg.value;
15    }
16
17    function deposit() public payable minAmount(1 ether) {
18        balances[msg.sender] += msg.value;
19        LogDepositMade(msg.sender, msg.value);
20    }
21
22    function getBalance() public view returns (uint balance) {
23        return balances[msg.sender];
24    }
25
26    function withdraw(uint amount) public {
27        if (balances[msg.sender] >= amount) {
28            balances[msg.sender] -= amount;
29            msg.sender.transfer(amount);
30        }
31    }
32 }

```

Listing 2.1: A simple contract where users can deposit some value and check their balance.

Listing 2.1 shows a simple contract written in Solidity in which users can deposit some value and check their balance. Before describing the code in more detail, it is helpful to give some insights about Solidity features like global variables, modifiers, and events.

Solidity defines special variables (`msg`, `block`, `tx`) that always exist in the global namespace and contain properties to access information about an invocation-transaction and the blockchain. For example, these variables allow the retrieval of the origin address, the amount of Ether, and the data

2. Background

sent alongside an invocation-transaction.

A particular convenient feature in Solidity are so-called modifiers. Modifiers can be described as enclosed code units that enrich functions in order to modify their flow of code execution. This approach follows a condition-orientated programming paradigm, with the main goal to remove conditional paths in function bodies. Modifiers can be used to easily change the behavior of functions and are applied by specifying them in a whitespace-separated list after the function name. The new function body is the modifiers body where `'_'` is replaced by the original function body. A typical use case for modifiers is to check certain conditions prior to executing the function.

An additionally important and neat feature of Solidity are events. Events are dispatched signals that smart contracts can fire. User interfaces and applications can listen for those events on the blockchain without much cost and act accordingly. Other than that, events may also serve logging purposes. When called, they store their arguments in a transaction's log, a special data structure in the blockchain that maps all the way up to the block level. These logs are associated with the address of the contract and can be efficiently accessed from outside the blockchain.

Given this short feature description, we can now return and analyze the code example. First, the compiler version is defined (line 1), then the contract is defined in which a state variable is declared (line 3), followed by an event definition (line 5), a modifier definition (line 7), the constructor (line 12), and the actual contract functions (line 16 onward). The state of the contract is stored in a mapping called `balances` (which stores an association between a users address and a balance). The special function `SimpleDeposit` is the constructor, which is run during the creation of the contract and cannot be called afterwards. It sets the balance of the individual creating the contract (`msg.sender`) to the amount of Ether sent along the contract creation transaction (`msg.value`). The remaining functions actually serve for interaction and are called by users and contracts alike. The `deposit()` function (line 16) manipulates the `balances` mapping by adding the amount sent along the transaction-invocation to the senders balance, while utilizing a modifier to preliminary ensure that at least 1 Ether is sent. The `withdraw()` function (line 25) manipulates the `balances` mapping by subtracting the requested amount to be withdrawn from the senders balance and the `getBalance()` function (line 21) returns the actual balance of the sender by querying the `balances` mapping.

In summary, this simple example shows the basic concepts of a smart contract coded in Solidity. Moreover, it illustrates the most powerful feature of smart contracts, which is the ability to manipulate a globally verifiable and universally consistent contract state (the `balances` mapping).

3. Architecture of Blockchain-Based Applications

Designing blockchain-based applications is a challenging task and requires a number of coordinated architecture decisions, including how to connect and orchestrate centralized elements, such as backend logic, with decentralized elements, such as the blockchain ledger and smart contracts. To guide decision making in this regard, we systematically explore this architectural design space and possible solution strategies in this chapter. More precisely, we provide architectural design decisions and decision options in terms of patterns and practices for applications with different degrees of decentralization, describe conceptual components, as well as possible relations between them.

Based on an analysis of collected data with Grounded Theory techniques, our research indicates that blockchain interactions can be abstracted as two kinds of blockchain gateway services, one of which sends state-changing operations and the other of which collects state information. By applying an event-driven architectural pattern, blockchain integration boils down to choreographing blockchain-dependent business logic with these gateway services.

3.1. Introduction

Blockchains are distributed peer-to-peer systems which implement a trustless shared public append-only transaction ledger [29]. They are being recognized as a useful technology in a wide variety of business applications to increase operational efficiency and enable new business models. However, considering the degree of maturity of blockchain technologies in practical applications, the acceptance and adoption of the technology is still in an early stage. Having gone through the hype-cycle for blockchains as an emerging technology, the industry is currently in a phase of consolidation where first practical applications provide insights into the advantages and disadvantages of using blockchain technology. Accordingly, there is currently a lack of a systematic and holistic approaches to system design of blockchain-based applications [30]. To close this gap, we investigate architectural design options for the integration of blockchains in software solutions by gathering data from different sources and applying GT techniques to extract and identify common practices. If one considers the blockchain as a part of a larger system, it can be assumed that certain practices and architectures occur more frequently and thus prove to be more advantageous than

3. Architecture of Blockchain-Based Applications

others. Furthermore, we investigate existing, well-proven software design patterns and assess their applicability to blockchain based applications.

In order to concretize the research objectives, we ask the following research questions: What are the key architectural design decisions for blockchain-based applications? What are possible design options regarding these decisions and the associated (best) practices? Which conceptual components are relevant in the architectural design and what are their relations? For illustrative purposes this chapter refers to the Ethereum blockchain, today's most popular ecosystem. Please note that the presented concepts are independent from a particular blockchain implementation.

This chapter is structured as follows: First, we discuss related work in Section 3.2 and our research methodology in Section 3.3. Then, we elaborate the architectural design of blockchain-based solutions as our main contribution in Section 3.4. Finally, we discuss our findings in Section 3.5, and draw conclusions in Section 3.6.

3.2. Related Work

Blockchain-Oriented Software Engineering (BOSE) is a growing discipline that focuses on the application and definition of software engineering principles for blockchain-based system design, development, and deployment. Porru, Pinna, Marchesi, *et al.* [30] present one of the first works to identify issues, challenges, and peculiarities in this field. They advocate the need for new research directions and novel specialized blockchain software engineering practices. Wessling, Ehmke, Meyer, *et al.* [31] argue that a blockchain-oriented view is required for the architectural design process and propose the idea of blockchain tactics as a means to support the process of integrating decentralized elements in software architecture. In this work, however, the authors focus on the effects of design patterns at the implementation level and do not provide architectural guidance. Marchesi, Marchesi, and Tonelli [32] [33] propose a holistic agile software development process to gather and analyze requirements as well as design, develop, test, and deploy blockchain applications. Nonetheless, the approach is not specific enough to derive decisions on the architectural level. Udokwu, Anyanka, and Norta [34] explore and evaluate several high-level design approaches for developing blockchain-based applications, including the former two works. They also propose another model-driven design framework with an automatic architecture model derivation, which unfortunately is not very specific. Bodkhe, Tanwar, Parekh, *et al.* [35] present various blockchain-based solutions and their applicability in various Industry 4.0-based applications. In the aforesaid work, a blockchain-based reference architecture is described, but rather on a high level. Viswanathan, Dasgupta, and Govindaswamy [36] present a Blockchain Solution Reference Architecture (BSRA) that guides architects in creating end-to-end solutions based on Hyperledger Fabric. Architectural components are mentioned, but only described within a layered structure, so that the interaction of the components is not apparent.

So far none of these works provide systematic architecture guidance in the field of BOSE. It is the aim of our work to close this gap.

3.3. Research Study Design

In the search for (best) practices (hereinafter conceptually equivalent to software design patterns and other similar best practices), we apply a research methodology that is guided by the pattern derivation approach of Riehle, Harutyunyan, and Barcomb [37]. The approach describes the application of established scientific research methods for the purpose of pattern discovery and validation. In accordance with this approach patterns are discovered (“mined”) and codified (“written”) using GT [38], [39] techniques. Driven by our research questions and known practices from our own experience, we defined initial search terms that were used to query major search engines (e.g., Google, Bing) in order to compile a number of well-fitting, technically detailed sources from the so-called “gray” literature [40] (e.g., practitioner reports, practitioner blogs, system documentations etc.). The resulting sources pool [41] was then examined in a later analysis with GT techniques. This included a thorough study and the annotation of the materials with labels (“codes” established with so-called “open coding”) along with optional memos explaining important aspects of codes. Further, conceptual relations between codes (so-called “axial coding”) were established to identify candidate categories for patterns. While this may indicate a simple linear execution of the work, pattern discovery and validation proceeded incrementally in several iterative stages, in which new sources (inspired from previous iterations) were exploited to constantly compare, revise, and contrast patterns until a theoretical saturation was reached. Theoretical saturation [38], [39] refers to a state in which adding new sources no longer yields new findings, and is commonly used as a stop criterion in GT-based studies.

3.4. Architectural Design of Blockchain-Based Applications

Today, the design and development of applications based on blockchain technologies is a difficult undertaking and the degree to which the technology is used is also significantly influenced by characteristics such as performance, usability, and user experience. A well thought-out architectural design helps to balance these criteria. To this end, this section discusses design guidance for blockchain integration that we found and coded in our study. Architectural decisions, decision options (aka practices or patterns), and typical conceptual components and their relationships are discussed. In a final analysis these decisions, their options, and the associated conceptual components are summarized in a feature model along with the relationships between them. To round things off, we also briefly touch on relevant aspects related to microservices and Blockchain as a Service (BaaS) at the end of this section.

3. Architecture of Blockchain-Based Applications

3.4.1. Event-Driven Architecture

As a software component, the blockchain has an asynchronous and event-driven character. This is due to the latency in the execution and confirmation of transactions and the fact that significant changes or operations that occur on the blockchain are usually propagated as events. Examples are events resulting from the execution of a smart contract or the creation of a new block. Given these characteristics, blockchains are not suited for real-time based systems and likewise scenarios where end-users expect an immediate impact of an operation. As with other systems that do not rely on synchronous execution or communication (i.e., no strict arrival times of messages or signals), message coordination can be achieved by using Event Driven Architecture (EDA). EDA is an architectural style in which there is no centralized controller to manage a workflow. Instead, different components interact with each other much more dynamically when certain events that affect their respective domains occur.

Event Sourcing

Event Sourcing is a persistence concept used in event-driven architectures. It refers to storing application state as a sequence of immutable events. With it, a complete replay of the events that have happened since the beginning of the event recording can be achieved. This stands in contrast to the traditional Create, Read, Update and Delete (CRUD) approach, where only the current state of an object is stored and iteratively mutated. Event Sourcing has several benefits. It allows for the creation of any number of user-defined data stores as materialized views of persisted events and knowledge about the state of domain objects at any given time by examining retroactive events. Blockchain and Event Sourcing share characteristics which suggest a natural affinity. Both share the concept of an “immutable append only log” which is considered as the single source of truth containing all events that have happened. Therefore, it seems natural to map, combine, and extend blockchains by Event Sourcing within application scenarios.

Command-Query Responsibility Segregation

The Command-Query Responsibility Segregation (CQRS) pattern [42] is quite often mentioned alongside with Event Sourcing, because when using Event Sourcing some form of CQRS emerges almost naturally. CQRS is a design solution that segregates operations that read data from operations that write data by using separate interfaces and persistence models. This approach promotes separation of concerns, as the distinction between write and read aspects can result in persistence models that are more aligned, maintainable, and flexible. Most of the complex business logic can go into the write model, while the read model can be kept relatively simple. Further, problems such as scaling read and write operations, using optimized data schemata, and securing authorized writes are

3.4. Architectural Design of Blockchain-Based Applications

easier to solve. The pattern can be utilized for blockchain integration, for example to account for the general discrepancy of write and read operations. The write model is represented by both transaction execution and the blockchain itself. The read model, as antagonist, is a locally synchronized replica of the blockchain to achieve fast read performance and rich querying capabilities.

3.4.2. Blockchain as a Multi-Faceted Architectural Component

When using the blockchain as a component in enterprise software, an important question is how to categorize it from an architectural point of view. If one starts out from classifying the blockchain within traditional boundaries such as a 3-tier architecture (presentation, application, data), the blockchain can be both, a data-tier that can store data on its ledger, and an application-tier, that can process data by means of smart contracts. When choosing the perspective of providing and consuming software as a service, in short Everything as a Service (XaaS), no clear picture emerges either, since the blockchain contains aspects of an application (Software as a Service [SaaS]), middleware (Platform as a Service [PaaS]), or infrastructure (Infrastructure as a Service [IaaS]) component. Consequently, the blockchain can be understood as a multi-faceted architectural software component that can be used in different degrees and settings.

3.4.3. Degrees of Decentralization

One can use the blockchain as a stand-alone platform capable of implementing a complete application logic (on top of smart contracts) or as an auxiliary tool in larger enterprise solutions to meaningfully complement business aspects (e.g., auditable history, asset tracking, etc.). Figure 3.1 illustrates this aspect, comparing a traditional 3-tier application design with a fully decentralized and hybrid blockchain-driven architecture. The latter two are essentially the major decentralization styles which will be discussed in the following.

Fully Decentralized Applications

A Decentralized Application (DApp) is a software solution built on top of a distributed peer-to-peer network. A DApp typically consists of a Web frontend that makes direct calls to a decentralized backend infrastructure (here, the blockchain executing smart contracts incorporating the core application logic). This structure is similar to a two-tier client-server architecture, with no intermediate support required for operation. The front-end code, which can be written in any language (just like a traditional Web application), can be hosted on a central server or on a decentralized storage (e.g., InterPlanetary File System [IPFS]). Through the latter a complete decentralization of the application is achieved. Benefits of DApps include an increased trust level and resistance to censorship, as the execution is not relying on a central provider which makes computation more

3. Architecture of Blockchain-Based Applications

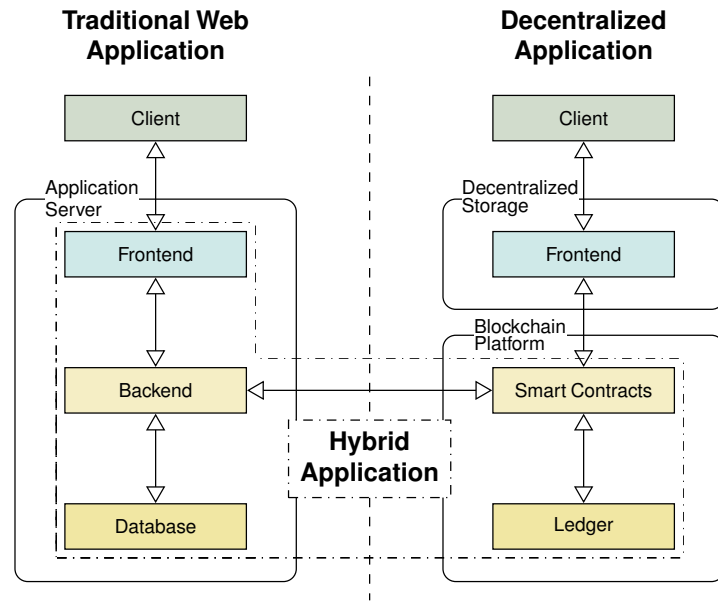


Figure 3.1.: A comparative overview of a traditional, hybrid, and decentralized application architecture.

transparent and further lowers the risk for a single point of failure. The disadvantages include low transaction throughput, high response times, difficult updating, incurring transaction costs (to be paid by the user), fluctuating transaction costs, and in general an immature technology stack accompanied by a vendor lock-in. For a comprehensive empirical study of blockchain-based DApps we refer to [43] and for an intra-architectural performance comparison to [44].

Hybrid (Semi-Decentralized) Applications

Building fully decentralized applications is a difficult undertaking. DApps based solely on distributed components quickly reach their limits due to current technical limitations and usability challenges. As a result, the current approach in building such applications is more nuanced. Instead of relying exclusively on decentralized components, often a hybrid architecture is realized and centralized components are added where appropriate. In this context, a traditional backend is still relevant and several reasons speak for its use, although it lowers the trust compared to purely decentralized applications.

First of all, the blockchain is a closed ecosystem, thus smart contracts cannot directly interact with off-chain services to fetch information or trigger actions. On the contrary, they depend on the outside world to push information into the network or triggering actions by monitoring the network. This means a backend server is needed whenever a reliance on third-party services exists, such as

3.4. Architectural Design of Blockchain-Based Applications

ingesting external data, or performing mundane operations like sending emails. Another use for a backend is to act as cache or indexing engine for the blockchain, which also helps to provide a more responsive user interface and smoother perceived user experience. While the ultimate source of truth is the blockchain, clients can rely on the backend for search functions and validate the returned data on-chain. Next, large data storage is impractical on the blockchain due to the high costs associated with on-chain storage. Therefore, an application may need to rely on a backend to store large amounts of data, while only a hash is stored on the blockchain for validation. In the same way, complex calculations that would exceed the block gas limit of the blockchain can be moved to a backend. Another case where a backend can be useful is batching multiple transactions. The user can be relieved of repetitive transactions that take a lot of time by collecting user signed transactions in the backend and issuing them all at once. As long as these transactions are not time sensitive, batching them is a valid use case. A backend is also handy for automation, when a smart contract is designed to be called at a future time. Since smart contracts are passive entities, i.e. they do nothing until a participant explicitly interacts with them, there is no built-in ability to schedule events in smart contracts. Thus, a backend system can be used to reliably initiate periodic calls for smart contracts.

Overall, to build a secure and partially decentralized application there is a strong need for a dedicated running backend.

3.4.4. Transaction Handling

Transactions provide the means to interact with a blockchain. Essentially, a transaction is a cryptographically signed instruction that is generated by an account, serialized and then transmitted to the blockchain for processing. There are three options how transactions can be initiated [43][45] which are discussed in the remainder of this section.

User Signed Transaction

The traditional way to initiate a transaction is from the user. In this case, the user interacts directly with a smart contract by signing a transaction with his private key and then sending it to the blockchain network along with a payment to cover the execution costs. This procedure requires that the user has software that supports client-side interactions (e.g., wallet) and tokens to pay for the transaction.

Meta Transaction

Transaction costs and the acquisition of tokens to pay for them are a major hindrance to a mainstream adoption of DApps. Meta transactions aim to solve this onboarding issue so that first time users can execute decentralized transactions without a crypto wallet. The simple idea is that a third

3. Architecture of Blockchain-Based Applications

party sends another user's transaction and pays for the execution. In this arrangement, the user signs a message containing information about a transaction the user wants to execute. This message is then sent free of charge to the off-chain third party, who subsequently wraps this information in a transaction and sends it to the blockchain network. This transaction is usually sent to an intermediate contract that verifies the user's signature of the attached transaction payload, before forwarding a subsequent transaction that executes a user intended method on the target contract. The advantage of this method is that the user is in control of his private key and does not need to bother with transaction fees. On the downside, the total costs are higher due to the additional costs for the relay contract and the fact that more transactions are required. Another issues is that the third party is centralized and could turn rogue, censoring transactions.

Backend Signed Transaction

Another approach to solving problems related to the accessibility of decentralized applications is to take the entire transaction signing and payment process away from the user and handle these matters in the backend. Although this method offers a high degree of comfort for the user and is relatively easy to implement, it destroys the fundamental concept of sovereignty and decentralized trusted execution, which is a basic principle of blockchains. Another downside is lacking transparency as the application may take unknown or unauthorized actions on behalf of the user and there is no way to challenge or reverse misaligned transactions. Further, since users do not own private keys they cannot own tokens or perform operations directly with other smart contracts. The only way to achieve this is to manage user keys in the backend which requires a sophisticated security concept to avoid any attacks.

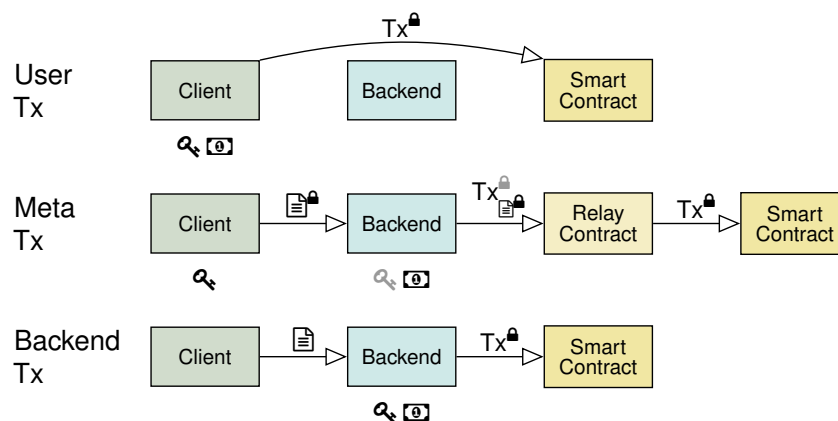


Figure 3.2.: An overview of transaction handling options.

3.4.5. Practices for Scalability and Privacy

This section briefly describes some of the techniques as options that can be selected or not in an architecture that are already in use to meet the challenges of scalability (see [46], [47]) and privacy (see [48]) that blockchains face.

Zero-Knowledge Proofs

Zero-knowledge proofs (ZKPs) are cryptographic protocols that allow one party to prove to another party that a statement is true without disclosing information beyond the fact that the statement is true. The algorithm that accomplishes this repeatedly tests a statement for true/false until the probability that the statement is false becomes incredibly low. For example, a company stores instead of the actual count (n) of a product inventory the hashed count on the blockchain and publishes the statement that $n > 10$ is true. With ZKPs others can rely on the fact that the published statement of this calculation is correct, without knowing the exact input value. This proof of correctness on the basis of “zero knowledge” gives ZKP its name. When used correctly, this technique can increase privacy on the blockchain. For a survey on ZKPs for blockchains see [49], [50].

Homomorphic Encryption

Homomorphic Encryption (HE) makes it possible to perform certain arithmetic operations on encrypted values; when the result is decrypted, the result is the same as it would have been if the same calculation had been performed with the unencrypted inputs. One can use HE to store encrypted data on the blockchain and allow a third party to analyze it and return an encrypted result that later can only be accessed by trusted parties in possession of the decryption key. Although this approach is very promising, it requires lengthy computations and is therefore not generally applicable. It is currently more practical to use ordinary encryption or an off-chain storage. For an illustration of potential practical application scenarios we refer to [51], [52].

State Channels

State channels refer to an approach in which users transact with one another directly outside the blockchain to minimize the use of blockchain operations. Instead of using the blockchain as the primary processing layer, it is used as a settlement layer. State updates occur outside and are only propagated to the blockchain when necessary. This approach enables a faster transaction flow and increased privacy because participants interact directly with each other. We refer the interested reader to [53], [54] for more information on this topic.

3.4.6. Conceptual Components and Their Interaction

The anatomy of blockchain-based applications in terms of essential components is to some degree similar across different use cases. In the following, we list several components that every blockchain-like solution may wish to consider in its architectural design. Commonly found layout patterns of components are outlined using the example of a typical DApp incorporating a backend in Figure 3.3 and an enterprise-oriented application containing several loosely coupled services (e.g., microservices [55]) in Figure 3.4. While there are many degrees of freedom in such architecture designs, such rough blueprint patterns can help in initial designs and for architecture classification. In the following, we discuss the typical conceptual components in those broader architecture patterns.

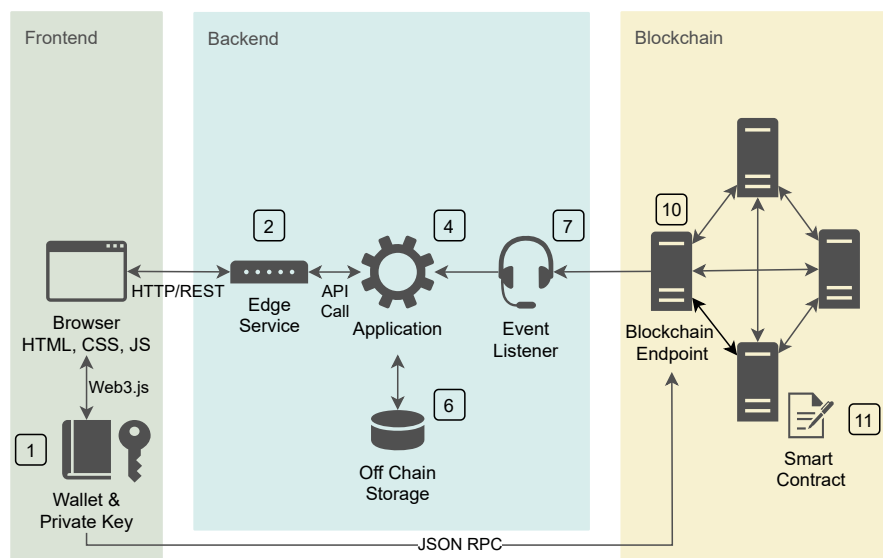


Figure 3.3.: A typical component structure pattern of a DApp utilizing a backend.

1) Wallet and Private Key

Instead of a password-protected centralized account, blockchain users have a decentralized identity that is based on asymmetric encryption, also known as public-key cryptography. The mechanism relies on pairs of keys: public keys, which may be distributed openly, and private keys, known only to the owner. A key pair is the identity on the blockchain. The public key (in a shorter representation) serves as account identification (address) and is derived from the private key that grants ownership of that account. Therefore the private key is the most crucial information for identification and its safekeeping is essential. A wallet is either a device, physical medium, program or service that stores a user's private keys. In addition to this basic purpose, wallets often provide the functionality of encrypting, signing, and forwarding information (transactions) to the blockchain.

3.4. Architectural Design of Blockchain-Based Applications

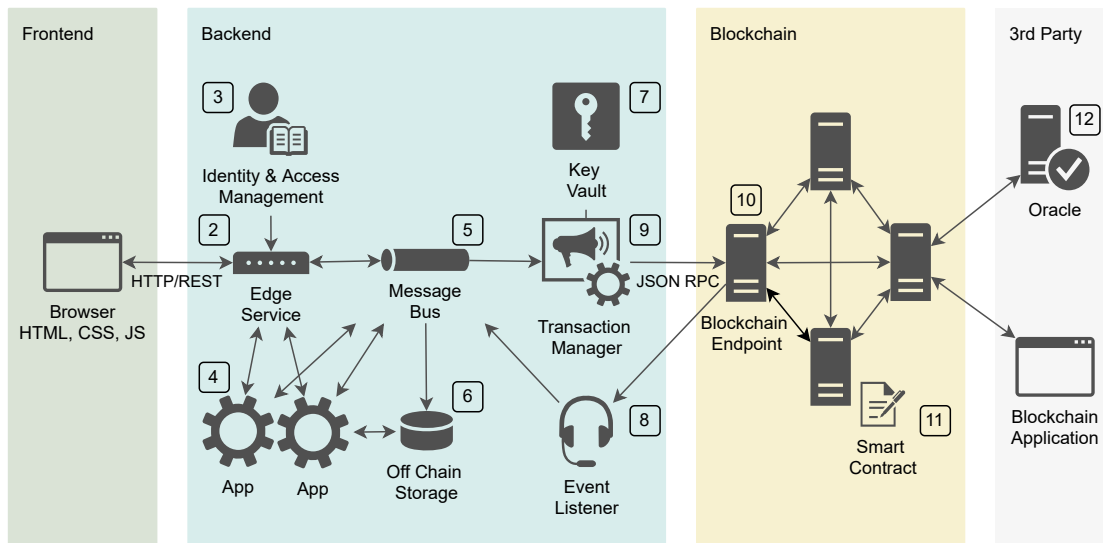


Figure 3.4.: A typical component structure pattern of an enterprise grade blockchain integration.

2) Edge Service

Edge services are components that are exposed to the public Internet and provide the capabilities required to deliver functionality and content to users over the Internet. They can refer to a multitude of components such as Content Delivery Networks, firewalls, load balancers, Application Programming Interface (API) gateways, reverse proxies, etc. They typically allow a shielded data flow from the Internet into the provider's infrastructure and into the enterprise. Edge services can also support backend applications by performing common tasks such as authentication, authorization, logging or monitoring to simplify underlying services.

3) Identity and Access Management

The identity and access management component stores user information to support user authentication and authorization as well as the provision of user data. Edge services can use this to control user-specific access to resources, services, and applications. For blockchain-based applications, it is necessary to align identity and access management holistically also with the blockchain inherent identity concept. It needs to be clarified to what extent users need sovereignty over their own blockchain identity and how this maps into an application-wide identity management perspective. For example, it is possible to leave the blockchain identity handling to the user or to the application as a custodian responsibility; in addition, an application-tailored blockchain identity concept without user binding is also conceivable. All of this must also be reconciled with transaction handling and secret key custody. In this context, it is possible to handle key management and transaction sub-

3. *Architecture of Blockchain-Based Applications*

mission either entirely by the user or in the backend, or as a middle ground have meta-transactions where user-signed transaction requests are sent via a backend that pays for their execution.

4) Backend Application Logic

An application implements the logic required to achieve business goals, typically by building either a monolithic solution or a set of small services organized by business capabilities that can be deployed independently. One of the most important considerations when integrating blockchain is what data and computations to put on-chain respectively off-chain. This decision is largely influenced by the business case and the intended benefit of using a blockchain (trust building, traceability, etc.) as well as current technological limitations. Generally, one should follow the basic design philosophy of using blockchains sparingly as they are slow and expensive. In terms of the communication flow between the application logic and the blockchain, this aspect is typically split into two separate components, one performing read and the other write operations. These components then serve as an interface to interact with the blockchain. When following a service-based design, it is also possible to further encapsulate both components through a dedicated “blockchain service” to have a central hub for blockchain interaction (e.g., Hyperledger Fabric Gateway [56]).

5) Message Bus

In a service-oriented application, the services not only process requests from users, but interact with each other to handle these requests. Therefore, they must engage in an appropriate communication protocol. In such a situation, asynchronous communication by exchanging messages via a message framework has many advantages. The messaging framework takes the role of a message broker allowing to validate, store (buffer), transform, and route (one-to-one/many, content/topic-based) messages between services.

This approach offers the advantage of loose runtime coupling, because it decouples the message sender from the consumer. It also improves availability, given that the message framework buffers messages when a consumer is temporarily unavailable. However, this aspect also reveals a disadvantage, namely that the message framework must be highly available. Looking at our gray literature sources, the choice for a messaging framework basically comes down to message processing or stream processing. In message processing, messages are written to a queue and a broker takes care of delivering the published messages addressed to specific endpoints. Once the processing of a message is confirmed, the message is removed. This form of message delivery is suitable for environments with complex pre-definable and stable routing logic or where there is a need for guaranteed one-to-one delivery of messages. Common platforms for message processing are ActiveMQ [57] or RabbitMQ [58]. In stream processing, messages are written to a log that

3.4. Architectural Design of Blockchain-Based Applications

is persistent (limited to a retention period/size) and any endpoint may listen to these events and react accordingly. Messages are not removed once they are consumed, instead they can be replayed or consumed multiple times. This implies that the endpoints keep track of which messages to read (next). Popular stream processing platforms are Kafka [59] or Pulsar [60]. When using a streaming platform one can filter, aggregate, analyze or transform any blockchain events (e.g., mined transactions or blocks, emitted event logs) and also combine this information with non-blockchain events. Hence, one could build a streaming analytics process that performs state checks by monitoring contextually relevant events over time (e.g., [61]). To sum up, message processing is all about smart pipes, dumb endpoints; while data stream processing is the opposite: dumb pipes, smart endpoints. In principle, both messaging framework types can be used for blockchain integration, but this decision also depends on the surrounding application components and the integration effort regarding the software stack used. Overall, regardless of the taken approach, the logic implemented in endpoints should be idempotent, so that receiving the same event twice has no side effects.

6) Off-Chain Storage

Any data store outside the blockchain that holds data related to the blockchain can be considered as off-chain storage. Off-chain storage serves two main purposes. On the one hand, it should enable faster access to on-chain data through local replication, and on the other hand, it should decouple business data from the blockchain, be it for reasons of confidentiality or data size. The former is a read-only store and since it reflects the on-chain state, it should only be updated according to received blockchain events and not by business logic. Its purpose is to support caching and indexing to enable search, filter, sort, and pagination capabilities for on-chain data. There are several ways to realize this type of storage. For example, it is conceivable to use the messaging framework (e.g., Kafka [59]) as event store and utilize its sink connectors (with Kafka Connect [62]) to provide data to databases, key-value stores, and search indexes (e.g., [63]). For data provisioning from the blockchain, source connectors exist that allow to ingest blockchain data tapped via web3 into Kafka (e.g., [64]). Another approach is to create a separate data store and synchronization service that subscribes to various blockchain events on the message bus and pushes data to a storage, which later on is consumed by application services for queries. As a side note, there is also a decentralized solution for querying in which a frontend database is used. For this purpose a browser database (e.g., PouchDB [65], GunDB [66]) syncs all relevant events, but this approach is not suitable for applications with a high data respectively event load. Now for the second purpose, namely a separate off-chain storage to detach business data from the blockchain. This type of storage can be used by business logic for a more controlled management of confidential data and may also serve as an exchange channel, if a shared storage is used. Its realization can take many forms depending on the type of data, such as a database (e.g., SQL, NoSQL) for metadata or a decentral-

3. *Architecture of Blockchain-Based Applications*

ized Content Adressable Storage (CAS) (e.g., IPFS [67], Swarm [68]) for Binary Large Objects (BLOBs), whereby the integrity of the data is guaranteed by storing hashes on-chain. All things considered, it is advisable to treat both storage concepts separately, although it is technically possible to unify them. For completeness, if data is to be held only on-chain, techniques such as ordinary encryption, homomorphic encryption, or zero-knowledge proofs can be used to ensure data confidentiality.

7) Key Vault

A key vault is a component used to maintain control of encryption keys and other secrets. It is essential for providing the private key that is always required when publishing transactions. There are several complex strategies and different software solutions that allow storing private keys quite securely on the backend (e.g., HashiCorp Vault [69]). Some solutions build on geographically distributed databases, while others built on specially designed hardware. In any case, it is not possible to sign transactions on the backend without revealing private keys somewhere in the system. Hence, it must be assumed that there is no 100% protection against the compromise of stored private keys. Through a neat construction of smart contracts, the effects of private key leakage can be minimized. The basic principle is to limit the functionality for accounts used on a backend, so that an adversary is unable to cause damage other than stealing a limited amount of funds. For this purpose, smart contracts use purely operational accounts for operative tasks and integrate the possibility to manage these accounts via a master account. The master account, whose private key is highly secured, finances the operational accounts and is able to override them, i.e. replace compromised accounts with newly created ones.

8) Event Listener

The event listener is a service component in the backend infrastructure and listens for and reacts to events emanating from a blockchain system or application. It contributes to a clear separation of concerns by avoiding the need for services to subscribe directly to a blockchain endpoint for events. It handles (dynamically) registered event subscriptions, and broadcasts these events in a consumable manner (over a message bus) to downstream services running on the backend. Blockchain is a closed system, so for event retrieval, inevitably repeated polling against blockchain endpoints is required. Notably, there are two possibilities: Either make explicit endpoint protocol requests for certain events, e.g. to check whether a transaction has been mined based on a transaction hash, or follow a “crawler” based approach, where a bulk invocation retrieves all transactions at once given a (new) block number for examination. In both cases, in order to avoid undetected events, the event listener has to gracefully handle various errors that invariably occur in production systems: nodes out of sync, crashing nodes, congested nodes, network disconnects, stale data returned in

3.4. Architectural Design of Blockchain-Based Applications

requests, etc. This suggests that it is advantageous to use multiple (own) blockchain endpoints for redundancy. In this constellation an aggregator pattern can combine (and de-duplicate) multiple endpoint events to propagate the data in a reliable, at-least-once manner. In the same way, the event listener may consider the immutability of the event stream (depending on the consensus mechanism). For a consensus algorithm that allows multiple chain heads, there may be multiple competing event streams at any one time. It is either possible to wait for guaranteed event finality (sufficient succinct block confirmations), or propagate events as soon as they arrive and assume that downstream services handle ramifications of prematurely published events. The latter approach has been mentioned in a few gray literature sources and adopts an eventual consistency guided way of thinking for transactions, whereby the application assumes that any blockchain transaction waiting on, will eventually confirm and continues on as usual. This approach leaves the application in a state which is ahead of the blockchain, allowing for example an improved user experience. However, having two instances of state (i.e. blockchain and application) can be problematic if state management is not handled carefully including rollback scenarios; namely, in case a transaction fails.

9) Transaction Manager

The transaction manager is a service within the blockchain application that receives messages (from the message bus) and issues state-changing transactions (invoking smart contracts). It is an abstraction that controls how transactions are signed and broadcast to the blockchain network, via a connected blockchain endpoint. The component performs various tasks associated with the publication of transactions. First, it takes care of estimating adequate transaction costs, to ensure transactions are equipped with enough funds for a timely execution. Second, it takes care of nonce management. A nonce is an arbitrary (mostly sequential), unique number that is used to prevent replay attacks. Third, it deals with signing the entire transaction (including the nonce). This step usually integrates a key vault as a private key assembly solution. Some blockchain libraries embed the mentioned tasks behind the scenes, nevertheless it is important to know the background. In addition, the transaction manager has to handle various errors that may occur: network congestion, dropping peers, dropping transactions due to a sudden price increase, etc. In order to ensure reliable and stable transaction processing, the transaction manager can join forces with the event listener to verify that transactions are mined within a specified time. If this is not the case, a certain transaction can be republished with different parameters (e.g., corrected nonce, a higher tx fee) and monitoring starts again. The basic process flow of this approach is depicted in Figure 3.5.

3. Architecture of Blockchain-Based Applications

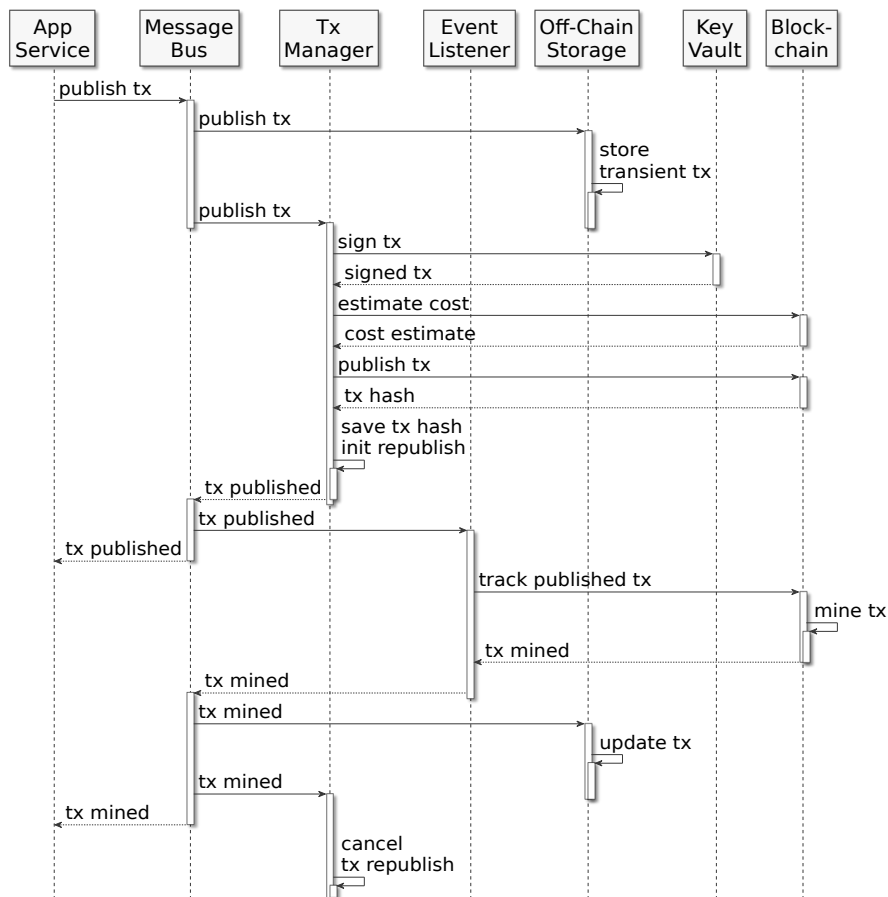


Figure 3.5.: A sequence diagram showing the interplay of key components during the processing of a transaction.

10) Blockchain Endpoint

A blockchain endpoint is a device or data point running a piece of software that implements the blockchain protocol to participate in the blockchain network. A node verifies all transactions in each block, keeping the network secure and the data accurate. Often there are different node implementations in different programming languages (e.g., Geth [70], Parity [71]), which follow a formal specification. Furthermore, a distinction is made between different node types. A full node has the entire blockchain downloaded and available. Hence, it can verify transactions and execute smart contracts independently. A blockchain network is maintained and operated by full nodes and the total number of full nodes indicates the degree of decentralization. A light node holds block headers instead of the entire blockchain and can validate whether a transaction belongs to the blockchain with the support of a full node. Accordingly, a lightweight node cannot operate

3.4. Architectural Design of Blockchain-Based Applications

without a full node. In addition, there are also service providers that operate node clusters (e.g., Infura [72], QuikNode [73]) which allow users to interact with the blockchain without having to set up their own node. All in all, operating an own node requires no trust in the network since the data can be verified in the node itself. If the blockchain is to be used in a truly private, self-sufficient and trustless manner, the operation of an own node is required.

11) Smart Contract

A smart contract implements the business logic on the blockchain and can be seen as a self-executing autonomous entity. It needs to be deterministic as otherwise peers could not agree on the results of a valid execution. As a design principle, contracts should be constructed to minimize the number and size of on-chain transactions and write operations to reduce costs. Smart contracts require a rather unconventional programming paradigm, because of the inherent characteristics of blockchain-based program execution. To handle those challenges, design patterns emerged to capture best practices and common solutions in a structured way [74]–[76]. We examine this topic in more detail in Chapter 5. In terms of data processing, smart contracts have their own state, but mostly they operate in relation to a common data model within the domain of a system, thus modeling and handling state transfer is a main concern. Complex aggregate or inferred state computations are typically kept off-chain and pushed on-chain with trusted oracles as needed. Another topic worth mentioning is the structural design and layout of smart contracts. In this context, there are different design options: A single smart contract acts as an interface (facade) that orchestrates interaction with other downstream smart contracts, or multiple smart contracts act independently with equal priority, and their functionality is being combined within a client/backend. In certain scenarios it is also common to use a template (factory) contract on-chain to instantiate contracts with the same structure and flow, but for a different context.

12) Oracle

Smart contracts often require access to data that is external to the blockchain ledger. An oracle is a trusted system designed to supply external data to the blockchain. Oracles require a level of trust that contradicts the trustless and decentralized nature of blockchains. This oracle problem, basically boils down to the issue of verifying the reliability of extrinsic information. The use of oracles should therefore be well thought out and accompanied by measures to mitigate this problem (e.g., sourcing data from multiple oracles). We will deal with oracles separately in Chapter 4.

3.4.7. Feature Model

To visually summarize the content presented so far, i.e., architectural design decisions, their options in the form of patterns and practices, and associated conceptual components, we present a feature model in Figure 3.6. Here, a feature is a distinctive aspect or characteristic of the blockchain-based software system. The feature model models possible relations between design options (modeled as features) via affiliated tags (mandatory/optional), relations (or/alternative), and relations to conceptual components (requires).

Where appropriate, the given design options are evaluated according to their positive impact on privacy, usability, and scalability. By means of drawn-in relations, the required conceptual architectural components for certain scenarios as well as necessary design options for a fully decentralized solution can be derived from the model.

3.4.8. Smart Contracts and Microservices

Microservices are an application architectural style in which a complex application is composed of many smaller, discrete, decoupled, and network-connected services that communicate with each other using standardized interfaces. Although smart contracts and microservices are fundamentally different in terms of the native environments they serve (decentralized vs. centralized platforms) and the challenges they seek to address, they are both a response to the rise of distributed architecture. While smart contracts are more about enabling transactions in low-trust environments, microservices are about enabling modularity and scale. However, smart contracts and microservices also have commonalities from a service-oriented architecture perspective (see [77]–[79]). Both are designed for focused functionality, autonomy, composability, and communication via standardized and well-defined interfaces. Hence, smart contracts can to some extent be interpreted as services of a blockchain-based computing paradigm. In this light, it makes sense to combine both concepts and design blockchain-based applications with microservices architecture principles. Following this approach brings not only the benefits associated with microservices (e.g., loose coupling, scalability, polyglot development, etc.) but also facilitates blockchain integration. When blockchain is treated as a service component in a microservices paradigm, it becomes easier to deal with its asynchronous and event-based nature. Proven concepts in microservices architecture, such as EDA, event sourcing, and CQRS, provide useful tools in this context. With EDA blockchain transactions can emanate as events and the flow of information within a system can be organized asynchronously in coordination with these events. Event sourcing and CQRS complement this approach, as blockchain state changes can be naturally stored as a continuum of immutable events within an event store, from which any view or structural model can be derived.

3.4. Architectural Design of Blockchain-Based Applications

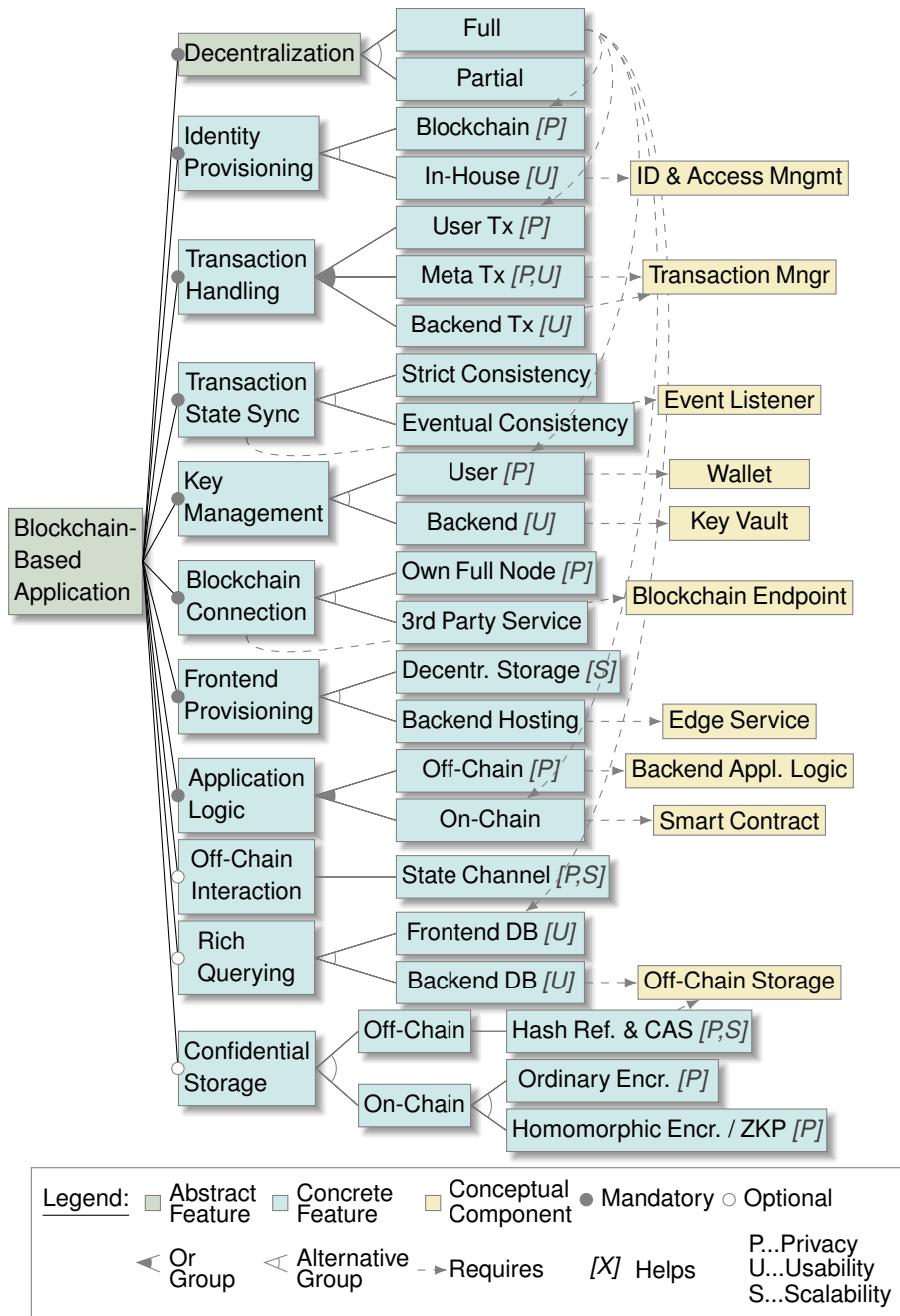


Figure 3.6.: A feature model for a blockchain-based application.

3.4.9. Blockchain as a Service (BaaS)

Looking at current trends in software development, one can speak of a new cloud-native application era where IT systems and applications are increasingly being outsourced to cloud service providers, for reasons of cost savings and improved management and maintenance. This trend has also caught up with the blockchain sector under the term Blockchain as a Service (BaaS). BaaS enables businesses to rely on a service provider to provision and manage aspects of a blockchain infrastructure in order to facilitate the development, testing, deployment, and ongoing management of blockchain applications. This approach allows development resources to be better focused on a specific goal by utilizing a wide range of readily available components while avoiding infrastructure and platform configuration overhead. A cloud environment can also be attractive in order to keep access and cooperation hurdles low for other participants (e.g., in a consortium). However, there are also disadvantages. Relying on a single service provider to run a decentralized blockchain network can be contradictory as it introduces a form of (re)centralization (trusting those who manage infrastructure). There is also the risk of a vendor lock-in, as it is difficult or very expensive to switch to a different service provider (due to lack of standardization especially for BaaS); the same also applies to the selection of a blockchain platform.

Many well-known IT companies such as Microsoft, Amazon, IBM, SAP, or Oracle now offer BaaS solutions (for a comparison see [80]–[82]) and enable the operation of blockchain nodes on their respective platforms, some even allow the use of third-party infrastructure. Various blockchain ecosystems and consensus mechanisms are offered, mostly geared towards permissioned blockchains (e.g., Hyperledger Fabric [83], ConsenSys Quorum [84]) that focus on performance and speed with strict privacy and access controls, which are preferred for business collaboration. In addition, many platforms offer a variety of different applications for operations, node, and smart contract management. Some even offer ready-to-use templates to provide predefined blockchain network configurations or generic applications (e.g., supply chain, financial services). Pricing models vary (e.g., number of nodes/transactions, storage space required, CPU utilization), also among providers, and often tenants can choose among several different options. This fits well to the prevailing on-demand and pay-as-you-go model of cloud service providers and various other services tenants can consume. As for the degree of built-in blockchain integration features, these vary between service providers. Some providers allow a deep integration with other built-in platform services out-of-the-box (e.g., through adapters, connectors, triggers, purpose fitted SDKs, etc.) using different integration approaches (e.g., serverless functions, workflow orchestration), while others focus only on infrastructure provision and basic interaction. With service providers in the former group, a Function as a Service (FaaS) approach is feasible, allowing the use of serverless computation options based on event-driven models where a piece of code (aka “function”) calling a smart contract is invoked by an event-based trigger such as a HTTP request (or

any other event). This enables, for example, the implementation of a uniform interface for smart contract interaction in a REST-API manner.

3.5. Discussion and Threads to Validity

A typical software architecture design requires various trade-off decisions to balance desired quality attributes. In the case of blockchain integration, this boils down to striking a balance between decentralization on the one hand and performance, privacy, and usability on the other. It can be said that the more decentralized a solution is, the more difficult it is to ensure the above quality attributes. To tackle this challenge, a hybrid architecture approach currently offers a good compromise. Decentralized and centralized components are combined, allowing the advantages of both to be used. The engineering challenge is then comprised of ensuring a smooth and timely communication between the two. In this context, it can be advantageous to treat events as first-class citizens of an application and use a messaging framework to coordinate communication between different components in an event-driven manner.

As far as communication is concerned, interaction with the blockchain can be narrowed down to two aspects. The first is listening to blockchain events and reading the blockchain network state. The second is publishing transactions to invoke state-changing operations. This is basically a two-way communication where the transaction manager is the write channel and the event listener is the read channel. By applying a service-oriented paradigm to these components and employing asynchronous communication, blockchain integration boils down to choreograph blockchain-dependent business logic with these gateway services.

Today, the computational and storage capabilities of blockchains are limited. If blockchains were scalable, they could host the whole software stack of an application, including its graphical user interface, business logic, and data. In that case, an application could truly be called decentralized respectively distributed. Since blockchain is not scalable today, various approaches (such as a dedicated backend, off-chain storage, etc.) are combined in order to achieve the highest possible degree of decentralization while taking into account basic requirements and software quality attributes.

Setting up application and blockchain infrastructure components can be a time-consuming and laborious task. Instead of developing an in-house solution with self-managed software, it can be efficient to outsource the infrastructure challenges to API gateway services or to cloud service providers. While gateway services abstract and encapsulate blockchain interaction behind HTTP API calls, cloud service providers offer a comprehensive implementation platform with a set of architectural artifacts that can be leveraged to accelerate the development. Various compute, orchestration, storage, messaging, logging, and monitoring services can be combined. Service providers with built-in integration options also have the advantage of being able to abstract direct

3. *Architecture of Blockchain-Based Applications*

interactions with the blockchain, facilitating the flow of communication and eliminating concerns about the reliability of transaction handling and event notification. In addition, available integration options can offer a scaffolding, that allows to wrap blockchain interaction into easier consumable and integratable blockchain services (e.g., as FaaS).

Overall, BaaS solutions can ease blockchain and application infrastructure management and speed up development, but are not on par with self-managed solutions in terms of provided trust. Nevertheless, the degree of trust can be indirectly influenced by how much is managed by a service provider, respectively whether BaaS is consumed as SaaS, PaaS, or IaaS.

The presented work is subject to a number of limitations and threats to validity. The practices are mined using a qualitative research method, thus possible biases of individual researchers cannot be fully excluded and might have influenced the results. This includes the pattern discovery and codifying procedure, as other researchers may have different interpretations and may code differently. Due to the fact that basic functional principles are the same across different blockchain implementations, it is likely that our results can be generalized. However, given that the majority of available sources only deal with a handful of more prominent blockchain solutions, there is a threat that our results can only be applied to similar blockchain architectures and the integration of different blockchain systems might not be possible without adaptation.

3.6. Conclusion

Blockchain is considered a disruptive technology that enables new business models and technological solutions. Consequently, new types of architectures and designs are required to utilize the technology at its best while addressing currently associated inefficiencies. To this end, we describe architectural design solutions for creating blockchain-based applications with different degrees of decentralization. In this context, we studied architectural decisions (see Figure 3.6) along with decision options (patterns and practices) and their connections to conceptual components. Based on our findings, we identified high-level architectural blueprints or patterns (see Figure 3.3 and Figure 3.4) in which we describe key components along with their purpose and interaction.

While it may seem straightforward to use blockchain for a specific business aspect, using the technology often comes at the expense of scalability, privacy, and usability. As a result, most design decisions are influenced by the need to compensate for these drawbacks. For example, to allow complex calculations, store large amounts of data, keep data confidential, or query blockchain-related data. To meet these requirements, skillfully coordinated design decisions are needed to work around the current drawbacks of blockchain. As for the integration of blockchain with its asynchronous and event-driven character, it is natural to adopt architectural styles and programming paradigms that are focused on these very characteristics. In this regard, an event-driven architec-

3.6. Conclusion

ture consisting of reactive components such as microservices or, in the cloud context, serverless functions, and dedicated blockchain read/write gateway services provides a good fit.

In the future, ongoing developments in the area of blockchains could lead to blockchains becoming more powerful and mainstream. Architectures embedding the technology will likely evolve and provide a promising foundation for diverse applications. In this context, future research could investigate (architectural) migration patterns to transfer functionality or architectural components to blockchain technology. With this outlook, it may one day be taken for granted that standard applications based on blockchain technology will be executed in an absolutely decentralized manner.

4. Oracle Patterns

Blockchain provides a platform for the development of decentralized applications which have beneficial characteristics such as high integrity, transparency, and resistance to censorship and manipulation. However, a blockchain is an isolated system due to its underlying operating principle and structure. This means that access to external information is not readily available. To address this limitation, oracles are used in practice. However, the best practices related to the use of such oracles have not yet been broken down, classified, and studied in their fundamental aspects. In this chapter, we fill this gap by examining basic blockchain oracle patterns based on fundamental dimensions related to data flow. From the blockchain perspective, this refers first to the direction of data flow, i.e., inbound or outbound, and second to the initiation of data flow, i.e., push- or pull-based. We provide a thorough and systematic description of the four patterns that emerge from these dimensions and discuss an implementation of the patterns based on use cases. On this basis, we perform a quantitative analysis that leads to the conclusion that the four distinct patterns are characterized by different performance and cost profiles.

4.1. Introduction

From a conceptual point of view, the blockchain represents an append-only transaction store distributed across many machines and structured into a linked list of blocks [85]. Based on this rationale and additions for distributed code execution, blockchain technology today offers a platform for decentralized applications with properties such as high integrity, transparency, and resistance to censorship and manipulation. These properties are particularly handy in use cases where data integrity and traceability are critical, such as clinical trials [86], [87], food safety [88], or in handling financial arrangements [85, Ch.12]. In addition, these appealing properties also promise new possibilities for handling inter-organizational business processes [89]. Specifically, the use of blockchain technology promises efficiency and effectiveness gains, for example, through the automated execution of business processes [90] or the exchange of information between mutually distrusting parties on the blockchain. In this context, the automation of business processes is achieved through the use of smart contracts, i.e., programs that are stored on the blockchain ledger and executed in the course of transaction processing. The blockchain thus offers a decentralized,

4. Oracle Patterns

	Pull	Push
Inbound	The on-chain component requests the off-chain state from an off-chain component	The off-chain component sends the off-chain state to the on-chain component
Outbound	The off-chain component retrieves the on-chain state from an on-chain component	The on-chain component sends the off-chain state to an off-chain component

Table 4.1.: An overview of the four oracle types.

neutral execution platform for coded business processes in the form of smart contracts.

However, blockchain systems have one limitation and that is that they form self-contained systems. In this respect, only data that is already on the blockchain can be accessed within the blockchain. To mitigate this limitation, so-called oracles are used in practice as intermediaries. An oracle is a component that can transmit data between the outside world and the blockchain. But implementing oracles presents significant conceptual challenges, as they can be considered a major point of failure and also introduce security and trust issues [89]. Consequently, much of the research on oracles focuses on how to address these security and trust issues. For example, by using multiple independent oracle entities to form a decentralized oracle [74], extending trust properties to off-chain computations [91], or strengthening trust in incoming data [92]. Nevertheless, fundamental aspects of blockchain oracles that enable their categorization and abstraction have not yet been explored in detail.

In this chapter, we fill this gap by abstracting from the way oracles are implemented and focusing on the basic patterns by which they are realized. In this context we ask the following research questions: What are the fundamental design patterns for implementing blockchain oracles? What are the characteristics of these regarding cost and performance? To this end, we examine two key dimensions regarding the realization of oracles: (i) the *direction*, i.e., whether the data flow is *inbound* or *outbound* from the blockchain’s perspective; and (ii) the *initiation* of the data flow, i.e., whether it is a *push*- or *pull*-based interaction from the blockchain’s perspective. These options result in four possible combinations, which are shown in an overview in Table 4.1. We describe each of these options as a pattern and examine the associated implications and characteristics. It is worth noting at this point that these four patterns can be implemented without relying on smart contracts, meaning their application is possible even for first-generation blockchains like Bitcoin. Additionally, each of the patterns can be appropriately combined with other higher-level patterns from the literature, e.g., to incorporate decentralization or provable computation. In order to characterize the different patterns, we implemented them in the context of two use cases on

the Ethereum platform. These implementations were then used to perform measurements. This involved sending more than 2,500 transactions to the Ethereum test network to obtain concrete data that allowed us to quantitatively investigate the characteristic differences between the four oracle patterns, with a focus on evaluating time (latency) and costs.

This chapter is organized in the following way: First, we provide a short background on oracles in Section 4.2 and discuss related work in Section 4.3. Then, we present and compare elaborated oracle patterns in Section 4.4. After that, we describe use cases for their implementation in Section 4.5 before analyzing the patterns in terms of time and cost based on these use cases in Section 4.6. Finally, we discuss our results and threats to validity in Section 4.7 and draw conclusions in Section 4.8.

4.2. Background

Blockchain Oracles

In many cases, applications running on the blockchain require data from off-chain states and events. Examples include sports results, weather forecasts, stock prices, randomness sources, or any other arbitrary data from services or devices outside the blockchain, where the required data is mostly accessible via web services and APIs. Blockchain oracles provide the ability to communicate with entities outside the blockchain in such cases. In general, they can be designed and implemented in various ways, e.g., as software (interacting with online sources) or hardware (interacting with the physical world), human (interacting with individuals) or computer (performing computations outside the chain), single-source (centralized, with one source) or consensus-based (decentralized, with a variety of sources) oracles [93].

There are a number of commercial and open source tools that implement inbound oracles, such as Orisi [94], Provable Things [95], TinyOracle [96], ChainLink [97], and Witnet [98], [99]. Orisi is a solution for Bitcoin that enables a distributed system of oracle nodes operated by independent and trusted parties. From the set of oracles, the majority must agree on the outcome of a particular condition. This process is handled by a temporary multi-signature address, which also serves as a vault for backers' funds. In terms of our categorization scheme, Orisi represents a pull-based inbound oracle. Provable Things, previously known as Oraclize, is a popular inbound oracle service that supports various smart-contract blockchain platforms. The service acts as a trusted intermediary between blockchains and a variety of independent data sources, and supports mechanisms (e.g., returning a result median) to minimize the risk of corrupt oracles [100]. It is powered by the Provable Engine which executes a set of instructions once certain conditions are met. This allows the service to be classified as both a push and pull-based inbound oracle. TinyOracle is an Ethereum-specific toolkit that is used to request data asynchronously via an intermediary contract. These data requests are being caught by a server side listener in the form of an Remote Procedure Call (RPC) client. After

4. Oracle Patterns

processing, the response will be sent back to a specific method of the querying contract. In regards to our categorization scheme, TinyOracle represents a pull-based inbound oracle. Chainlink is a decentralized blockchain oracle network built on Ethereum. It is an open-source technology infrastructure that allows any blockchain to securely connect to off-chain data and computation resources. The Chainlink network nodes retrieve, verify, and transmit data (even without prior explicit request) to the blockchain and executing smart contracts. Node operators are compensated with the network's native cryptocurrency, LINK. The service can be classified as both a push and pull-based inbound oracle. Witnet provides a protocol that creates an overlay decentralized oracle network (DON) connecting smart contracts to any online data source. The network runs a native customized blockchain and its own protocol token namely WIT. Witnet peer nodes are colloquially referred to as witnesses, these earn WIT tokens as a reward for retrieving web data and reporting it directly to the smart contracts. The protocol applies a common consensus algorithm that resolves inconsistencies. In our categorization scheme, the service can be classified as a pull-based inbound oracle.

4.3. Related Work

Blockchain oracles have been considered in a number of research papers, specifically examining inbound oracles. Xu, Pautasso, Zhu, *et al.* [101] introduce the concept of validation oracles for evaluating conditions that cannot be expressed within blockchains. In this context, they refer to trusted third parties that act as either automated or human intermediaries. The authors distinguish between two types here. First, internal validation oracles that regularly submit externally verified data to the blockchain, and external validation oracles that act as trusted external validators of transactions based on information outside the blockchain. According to our scheme, the first type corresponds to push-based and the second type to pull-based inbound oracles. Adler, Berryhill, Veneris, *et al.* [102] present a decentralized, trustless, and permissionless blockchain oracle system, called ASTRAEA. Submitters enter propositions into the system, while voters and certifiers play a game to determine the truth value of each proposition. In the process, the game-theoretic incentive structure is analyzed to show that a desirable Nash equilibrium exists in which, under a set of simple assumptions, all rational players behave honestly. The proposed oracle implementation resembles a pull-based inbound oracle. Zhang, Cecchetti, Croman, *et al.* [103] present Town Crier, an oracle system to provide authenticated data feeds. The system addresses trust issues in regards to oracles by using trusted hardware, namely the Intel Software Guard Extensions (SGX) instruction set, a capability in certain Intel CPUs. It allows to scrape HTTPS-enabled websites and serve source-authenticated data to relying smart contracts. This oracle implementation resembles a push-based inbound oracle.

Overall, it can be noted that a majority of the effort is dedicated to the design and implementation of inbound oracles. In fact, a recent ISO/TC 307 technical report characterizes oracles solely for

providing off-chain information to the blockchain [104]. However, in this chapter, we also examine and specify the patterns behind the opposite information flow, that of outbound oracles, also known as *reverse* oracles [74].

4.4. Patterns

This section describes basic oracle patterns resulting from the division of direction (inbound/outbound) and initiation of data flow (pull/push) between on-chain and off-chain components. Figure 4.1 shows the data flow along the basic dimensions outlined above. In applying this breakdown, a basic distinction can be made between inbound oracles and outbound oracles, each of which can be further refined according to data pull and push strategies.

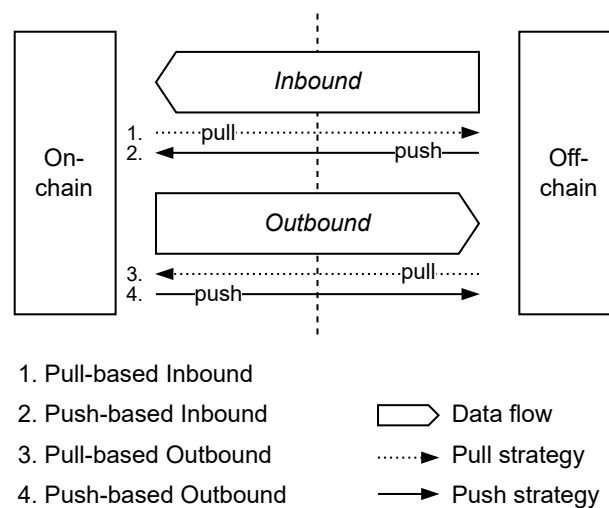


Figure 4.1.: A conceptual overview of the oracle data flow partitioning.

Before discussing the individual patterns in more detail, we first provide a general overview of the patterns and their respective conceptual structural components (also called “pattern participants”) in Figure 4.2. The blockchain is viewed as part of a larger software system, with software components located on- and off-chain. In such an environment, it is often necessary to be able to communicate across system boundaries in both directions to exchange information. For example, components on the blockchain (such as smart contracts) may need knowledge from software components outside the blockchain and vice versa, meaning the outside world also needs knowledge from the blockchain. Regarding the terminology used in this chapter, it should be noted that the term “event” in relation to the blockchain refers to any activity that can take place on the blockchain (e.g., data is persisted, a transaction takes place, a block is added, etc.).

4. Oracle Patterns

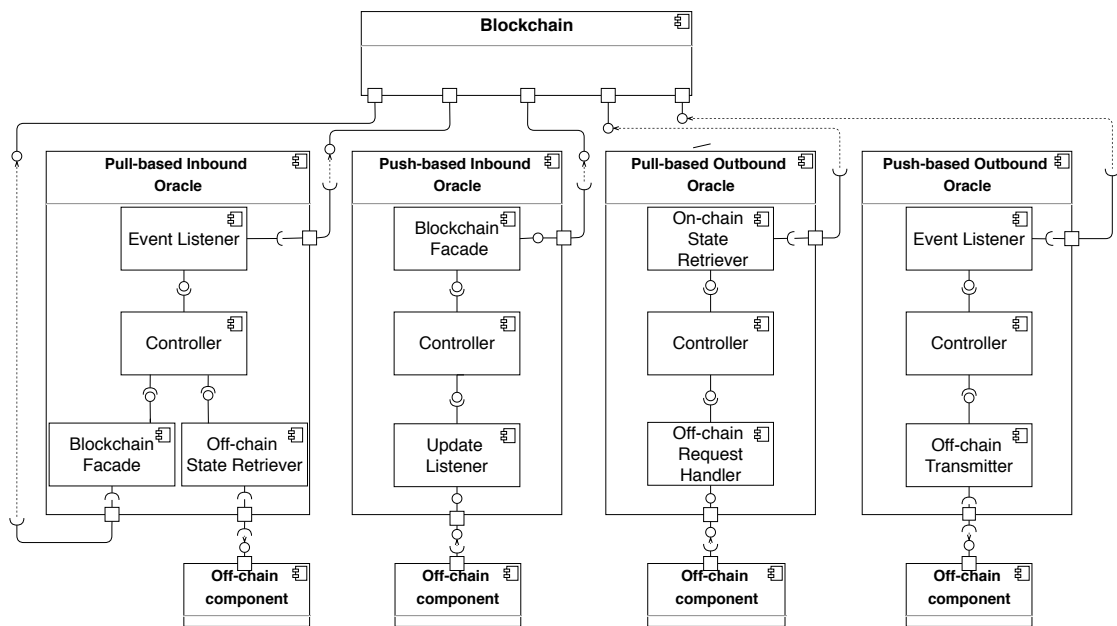


Figure 4.2.: An overview of the oracle types and conceptual structural components.

4.4.1. Inbound Oracle Patterns

In the following, we present the patterns for inbound data flow towards the blockchain, i.e., the Pull-Based Inbound Oracle and Push-Based Inbound Oracle.

Pull-Based Inbound Oracle

PATTERN Pull-Based Inbound Oracle

Problem A blockchain application requires knowledge contained outside the blockchain. However, because blockchains are closed systems, applications cannot directly obtain information from the outside world.

Solution A Pull-Based Inbound Oracle allows blockchain applications to request state information from off-chain components. When a blockchain application requests an off-chain state, the Pull-Based Inbound Oracle receives this request, collects the state from off-chain components, and sends the result (via a transaction) back to the blockchain.

Benefits	State requests are initiated on the blockchain. Thus, the entire process is transparent. It can be traced whether off-chain data was successfully provided (in time) or not.
Drawbacks	State requests are initiated on the blockchain but cannot be fulfilled without outside intervention, resulting in a passive nature of information retrieval. Furthermore, the response time depends on the speed of the blockchain network, which can lead to a bottleneck. Network congestion may result in delayed or missed off-chain state inquiries, as the oracle starts working only after it has registered requests from the blockchain.

An inbound oracle transmits information from the outside world to the blockchain. Since a blockchain cannot obtain information directly from the outside world, it relies on the outside world to push information into the network. Given this fact, the most obvious approach for obtaining external information on the blockchain is to inform the outside world of the need to push the required information to the network. This approach is described in the Pull-Based Inbound Oracle pattern and is characterized by initiating the exchange of information on-chain.

The conceptual interaction of the pattern participants is shown in Figure 4.3: An *Event Listener* subscribes to relevant events on the blockchain, which forwards the event data to a *Controller*. The *Controller* collects the required data from an off-chain component via an *Off-Chain State Retriever*. The collected data can be further processed by the *Controller* before being returned to the blockchain via a *Blockchain Facade*.

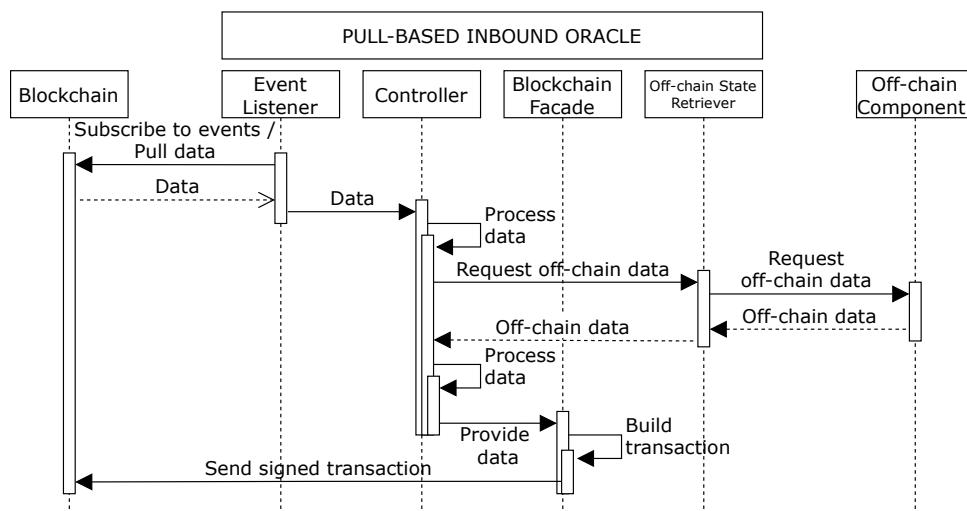


Figure 4.3.: A sequence diagram showing the component interactions for the Pull-Based Inbound Oracle.

Push-Based Inbound Oracle

PATTERN	Push-Based Inbound Oracle
Problem	A blockchain application needs to be supplied with knowledge outside the blockchain, but since blockchains are closed systems, this knowledge cannot be communicated directly.
Solution	A Push-Based Inbound Oracle enables off-chain information to be propagated to the blockchain by monitoring off-chain state changes and passing them to the blockchain.
Benefits	Scattered or irregularly updated data outside the blockchain is proactively forwarded to the blockchain application. Therefore, the blockchain application does not need any capabilities to search and query data outside the blockchain. In addition, given the limited functionality of blockchain environments, data outside the blockchain can be more easily checked and verified by the Push-Based Inbound Oracle.
Drawbacks	The Push-Based Inbound Oracle is not deployed or triggered on the blockchain, so data provisioning depends entirely on (non-distributed) applications running off-chain. To manipulate blockchains with false information, an attacker only needs to compromise the off-chain component(s) from which the oracle receives the data.

Another approach to transferring external knowledge to the blockchain is to monitor changes in the off-chain world that are relevant to the blockchain and transfer these changes to the network. Data can be sent from a specific data source to the blockchain when a state change occurs or as soon as data needs to be transferred to the blockchain. This approach is described by the Push-Based Inbound Oracle pattern and is characterized by the fact that the information exchange is initiated off-chain.

The Push-Based Inbound Oracle, as conceptualized in Figure 4.4, listens for relevant off-chain component updates via an *Update Listener* and forwards the data to the *Controller*. The *Controller* can process (e.g., filter, verify, etc.) the data before sending it to the blockchain via a *Blockchain Facade*.

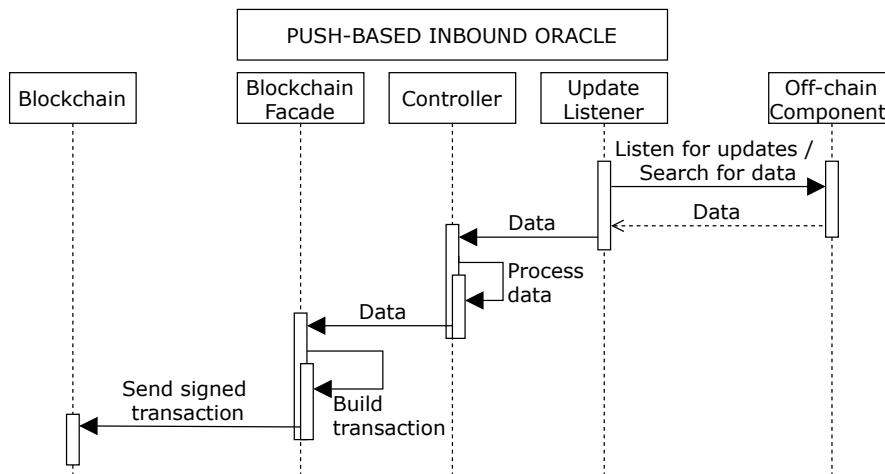


Figure 4.4.: A sequence diagram showing the component interactions for the Push-Based Inbound Oracle.

4.4.2. Outbound Oracle Patterns

In the following, we present the patterns for outbound data flow from the blockchain, i.e., the Pull-Based Outbound Oracle and Push-Based Outbound Oracle.

Pull-Based Outbound Oracle

PATTERN	Pull-Based Outbound Oracle
Problem	Information contained on the blockchain is needed outside the blockchain, but since blockchains are closed systems, the outside world cannot request this information directly.
Solution	A Pull-Based Outbound Oracle can be used to query and filter blockchain data to make it available to the outside world. It can be called by (off-chain) components to pull (all) blockchain data and query relevant information.
Benefits	The Pull-Based Outbound Oracle can be used to decouple state requests from the actual state query. Thus, the pattern offers the possibility to access and query relevant information on the blockchain in a consistent way.
Drawbacks	Depending on the size of the blockchain and knowledge about the location of the requested information, it may take some time to provide the data.

4. Oracle Patterns

An outbound oracle transmits information from the blockchain to the outside world. Because of its underlying properties, a blockchain can store state information in the form of transactions, but it cannot actively communicate that state to the world outside the blockchain. Therefore, the most obvious way to obtain data from the blockchain is to retrieve it. This approach is described by the Pull-Based Outbound Oracle pattern and is characterized by initiating the exchange of information off-chain.

The Pull-Based Outbound Oracle, as conceptually outlined in Figure 4.5, receives off-chain data requests via an *Off-chain Request Handler* and forwards the requests to the *Controller*, which processes the requests before forwarding them to the *State Retriever*, which is responsible for retrieving data from the blockchain. The result is returned to the *Controller*, which can process the data before sending it to the off-chain requestor via the *Off-chain Request Handler*.

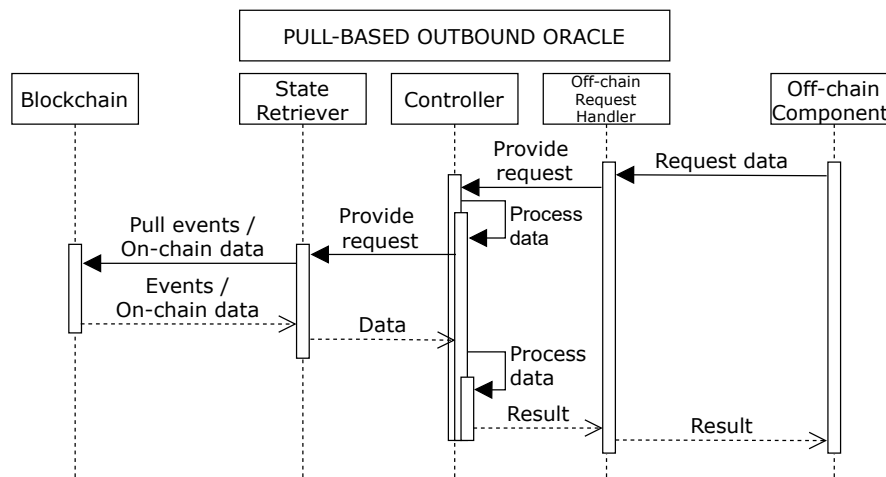


Figure 4.5.: A sequence diagram showing the component interactions for the Pull-Based Outbound Oracle.

Push-Based Outbound Oracle

PATTERN	Push-Based Outbound Oracle
Problem	Information contained on the blockchain must also be available outside the blockchain. However, because blockchains are closed systems, blockchain applications cannot share this information directly with the outside world.
Solution	A Push-Based Outbound Oracle monitors the blockchain for any relevant changes and then initiates or performs activities outside the blockchain.

Benefits	The Push-Based Outbound Oracle constantly monitors the blockchain. This allows to (partially) automate blockchain-related tasks by taking action when a blockchain state is updated.
Drawbacks	The oracle must run continuously to monitor all events (in a timely manner) on the blockchain. If the oracle stops unexpectedly, state updates may be missed (depending on the implementation). In addition, depending on the speed of the blockchain network, delays can occur, which can lead to undesired delays in time-critical scenarios.

Another approach to transfer internal information from the blockchain is to observe changes on the blockchain that are relevant to the outside world and transfer these changes off-chain. This approach is described by the Push-Based Outbound Oracle pattern and is characterized by the fact that the information exchange is initiated on-chain.

The Push-Based Outbound Oracle, as shown in Figure 4.6, subscribes to relevant events on the blockchain via an *Event Listener* and forwards event data to the *Controller*, which can process the data before sending it to an off-chain component via the *Off-chain Transmitter*.

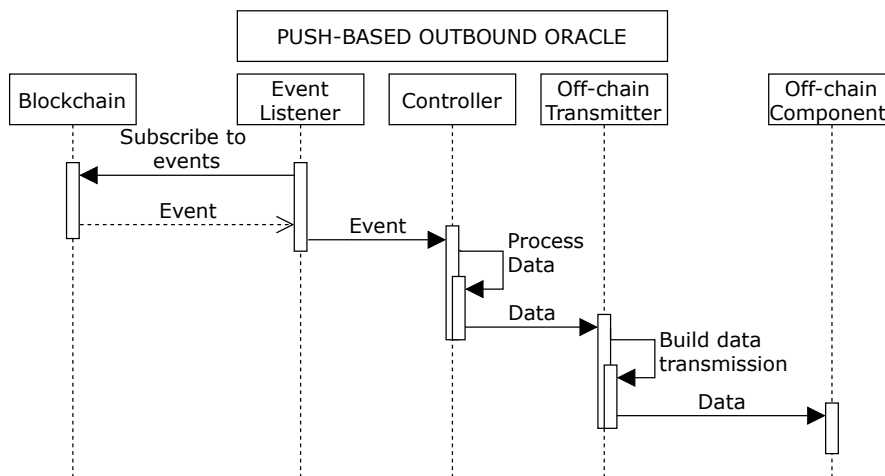


Figure 4.6.: A sequence diagram showing the component interactions for the Push-Based Outbound Oracle.

4.5. Use Cases

There are several uses for blockchain, one of which is the execution or orchestration/choreography of business processes between multiple parties (see, e.g., [105]). In the following, a business

4. Oracle Patterns

process as well as two use cases within this business process are described in more detail, which we referred to when implementing the oracle patterns.

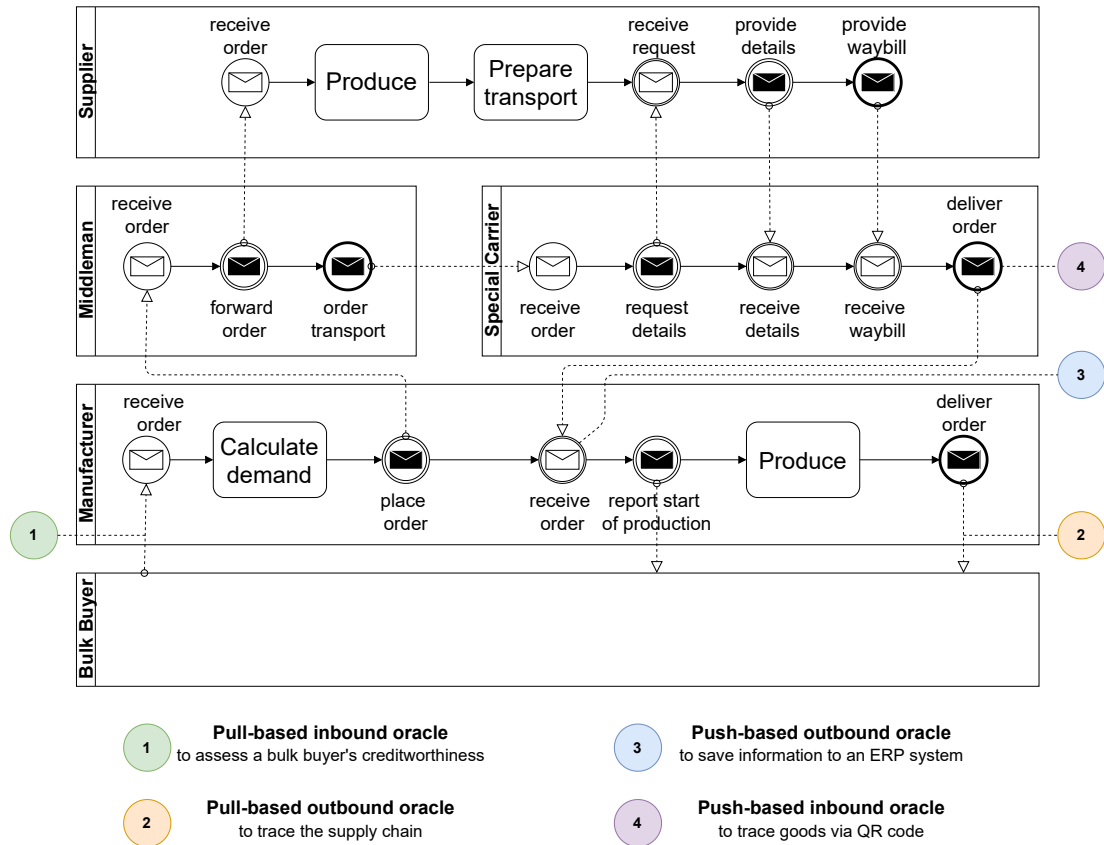


Figure 4.7.: A supply chain process in Business Process Model and Notation (BPMN) adapted from [90] including the use of oracles.

As a business process, we take a supply chain scenario represented in Figure 4.7, which is derived from a scenario in [90]. The illustrated business process is about the flow of goods between a wholesale buyer, a manufacturer, a middleman, a special carrier, and a supplier. The process begins with a bulk buyer placing an order with a manufacturer. The manufacturer, in turn, calculates the materials needed and instructs an middleman to forward the order to a supplier and book transportation through a dedicated special carrier. Once the required materials are ready, the special carrier takes care of the transport from the supplier to the manufacturer. Last but not least, the manufacturer produces the goods and delivers them to the bulk buyer.

When this business process is mapped on a blockchain, there is inevitably a flow of information between the blockchain and the real world. Here, oracles ensure the transfer of information from the off-chain to the on-chain world and vice versa. In our implementation, we use four oracles -

one for each pattern. Their use is highlighted in Figure 4.7 and is explained in more detail below using two specific use cases.

Use Case 1 The first use case in Figure 4.8 concerns a bulk buyer's order request to the manufacturer. It involves a single Pull-Based Inbound Oracle pattern to verify the buyer's creditworthiness. The following steps are executed:

1. The bulk buyer places an order via a web application.
2. The order data, including the order ID and information about the bulk buyer, is forwarded to a smart contract via a transaction.
3. The smart contract publishes an event containing information about the bulk buyer such as name and tax registration number. The *Event Listener* of the oracle as a subscriber of such events receives this information.
4. To retrieve information about the creditworthiness of the buyer, the oracle makes a request to an external credit check service via the *Off-chain State Retriever*.
5. The oracle processes the response with the *Controller* and returns this information as transaction data to the smart contract with its *Blockchain Facade*.
6. Finally, the manufacturer can retrieve the order with the knowledge that a credit check has been performed on the buyer.

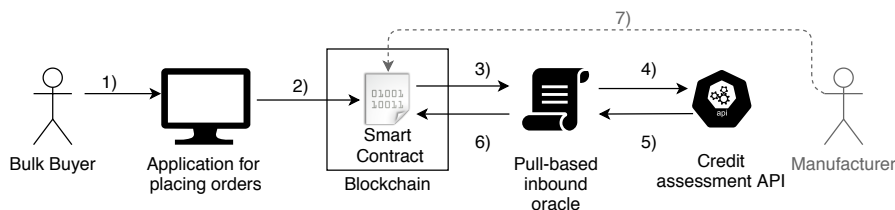


Figure 4.8.: Oracle-based creditworthiness verification of actors in the supply chain process shown in Figure 4.7.

Use Case 2 The second use case in Figure 4.9 concerns blockchain-based tracking of goods in a supply chain via QR code scanning. The process involves three different oracle patterns, namely the Push-Based Inbound Oracle, Push-Based Outbound Oracle, and Pull-Based Outbound Oracle. The basic process flow is as follows:

1. An employee of the manufacturer receives a package containing goods from the supplier and uses a device with a QR code scanning application to confirm the receipt.
2. The QR code information is decoded to reveal the order ID, name, and quantity of the items.
3. The Push-Based Inbound Oracle receives the decoded data through its *Update Listener* and its *Controller* enriches the data with the scan location and current timestamp.

4. Oracle Patterns

4. The oracle's *Blockchain Facade* encodes the data into a blockchain transaction and transmits it to a smart contract.
- 5a. The smart contract in turn publishes an event which is subsequently registered by the *Event Listener* of a Push-Based Outbound Oracle.
- 6a. The oracle's *Controller* decodes the event data and forwards it to an Enterprise Resource Planning (ERP) system via its *Off-chain Transmitter*.
- 5b. The bulk buyer tracks the production status of items identified by the order ID via the blockchain using a web application.
- 6b. Upon an order ID information request, the web application calls the *Off-chain Request Handler* of a Pull-Based Outbound Oracle.
- 7b. The oracle's *Controller* transforms that request into a query that is fulfilled by the oracle's *On-chain State Retriever*.
- 8b. The query result is processed by the oracle and the information is passed through its components in reverse order.
- 9b. The processed result is returned to the web application and the entire record about the order and its products can be retrieved.

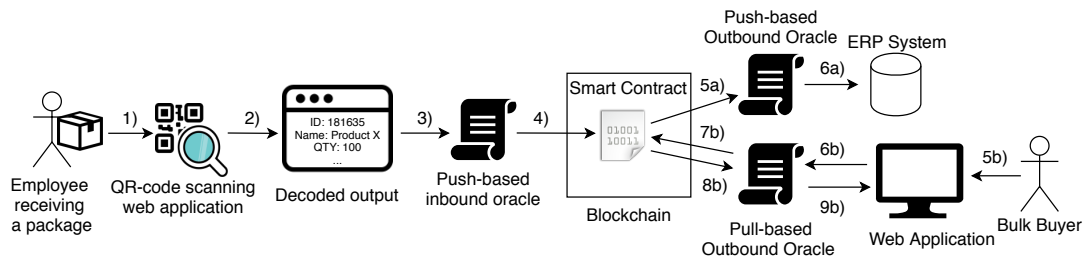


Figure 4.9.: Oracle-based tracking of goods via QR Code scanning in the supply chain process shown in Figure 4.7.

We have implemented both of the above use cases for the Ethereum platform using Python, web3.py [106], and QR code scanner [107]. The full source code is available on GitHub [108].

4.6. Analysis of Performance and Transaction Fees

In this section, we explain our results regarding the quantitative analysis of proof-of-concept implementations of the four oracles based on the use cases described above.

Setup

In our analysis, we are mainly interested in two dimensions: time and cost. Regarding time, more precisely latency, we are focusing on two questions. The first question aims to determine whether there are time differences that can be detected between different oracle implementations. This can provide insights into the potential uses of each of the proposed oracle patterns and further support their breakdown from a software perspective provided in this chapter. The second question aims to explore whether our experimental settings affect the time delays.

To measure latency (see also Figure 4.10), we record the time between sending a transaction to the blockchain node (t_1) and receiving the transaction hash (t_2). We report the difference as $dt_{\text{tx-hash}}$. For the Push-Based Outbound Oracle, we measure the time between the timestamp of the block that contained the transaction (i.e., the timestamp when the miner started mining that block, t_3) and the reception of the event (t_4). We refer to the difference as $dt_{\text{tx-mined}}$. If it is clear from the context, we refer to both measures as dt . In regards to measuring latency, it is debatable whether mining time should be part of the latency measurement. This is due to the fact that the time between the submission of a transaction and its recording or confirmation in the ledger can vary drastically between different blockchain platforms. In this context, it is also important to consider other influencing factors, such as network congestion and, for commit time on PoW blockchains, the number of confirmation blocks, which is a user-defined parameter (see, e.g., [109] for details and measurements). In our analysis, we measured the simple inclusion time without additional confirmation blocks, as a placeholder and to highlight the underlying problem.

To measure the costs of inbound oracles, we measure gas consumption. It should be noted that the gas costs also capture the computation and storage overhead. We convert Ether to Euros using the average exchange rate for Ether during the study period (144.86 €/Ether), and gas consumption is converted to Ether using the gas price of the transactions (on average 7.45×10^{-10} Ether / gas).

The measurements were conducted on Ethereum's Ropsten test network, which is recognized in the scientific literature for testing purposes, see [110]–[112]. The test code as well as the code used for the quantitative analysis are available on GitHub [108]. The first use case of Figure 4.8 is mimicked by the smart contract *customer.sol*, which is used to evaluate the Pull-Based Inbound Oracle. It is deployed under the address `0x9c2306ecc5afa6ee0c1eca6deab66cc336c3b3d`. The second use case of Figure 4.9 is mimicked by the smart contract *arrival.sol*, which is used to evaluate the Push-Based Inbound Oracle, Pull-Based Outbound Oracle, and Push-Based Outbound Oracle. It is deployed under the address `0x1186aEDAb8f37C08CC00a887dBb119787cfE6AAf`. Regarding the above implementations, the following should be noted: The retrieval state of the Pull-Based Outbound Oracle is kept constant to exclude it as a varying factor. Furthermore, in the implementation of the Pull-Based Inbound Oracle, we do not store any states in the receiving smart contract, since the transaction invokes the client smart contract directly and we exclude its handling

4. Oracle Patterns

of the data in the experiment. In contrast, the Push-Based Inbound Oracle stores state and emits an event; this is necessary for the client smart contract to retrieve the state.

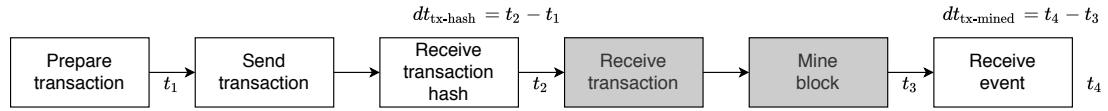


Figure 4.10.: The schematic process for measuring latency, with off-chain (white) and on-chain (gray) tasks.

Results

Figure 4.11 and Table 4.2 show the results of our measurements. The Pull-Based Outbound Oracle is the oracle with the lowest latency with a mean dt of 0.13 ± 0.03 seconds, while the Push-Based Outbound Oracle has the highest latency with a mean dt of 16.20 ± 15.95 seconds. The difference can be explained by the fact that the Pull-Based Outbound Oracle reads existing state from the blockchain, while the Push-Based Outbound Oracle relies on the inclusion of a transaction that is subject to high variance and an average delay of about 1.5 inter-block times [85]. Said transaction triggers the event that is picked up by the Push-Based Outbound Oracle. For the Push-Based Outbound Oracle, 75% (i.e., the third quartile) of transactions are within 0.12 seconds. For the Push-Based Outbound Oracle, on the other hand, the corresponding value (i.e., the third quartile) is 21.44 seconds. One can see from the box plots in Figure 4.11 that the dt measurements of the Pull-Based Outbound Oracle and the Push-Based Inbound Oracle are characterized by a significant number of outliers that follow a long-tail distribution. This fact is less pronounced for the other two oracles. Leaving aside these outliers, the dt distribution for the Pull-Based Inbound Oracle is similar to that for the Push-Based Inbound Oracle, with a mean dt of 0.52 ± 0.05 and 0.53 ± 0.08 , respectively, and the same minimum (0.46) and median (0.50) values. Both differ marginally in their 25th (0.48 vs. 0.49) and 75th (0.52 vs. 0.54) percentiles.

For the Push-Based Inbound Oracle and Pull-Based Inbound Oracle, we measured transaction costs in Ether and converted them to Euros using the previously mentioned exchange rate. Table 4.2 summarizes the obtained results. The determination of the gas price in our setup was based on the current market price, which proved to be highly variable on Ropsten and not representative of the Ethereum mainnet. To get an indication of the cost we would have incurred on the mainnet, we obtained the approximate median gas price from Google's public BigQuery database for Ethereum for the period in question, which was 8.5 Gwei (averaged over 3.15 million transactions). Multiplying this by the average gas consumption and the exchange rate, we get a median transaction cost of 0.028 € for the Push-Based Inbound Oracle and 0.056 € for the Pull-Based Inbound Oracle.

Table 4.2.: A summary of statistics on time and costs for oracle invocations (on the Ropsten Ethereum testnet).

	<i>n</i>	mean	std	min	$x_{0.25}$	$x_{0.50}$	$x_{0.75}$	max
<i>Push-based inbound oracle</i> 2437								
$dt_{tx-hash}$ [seconds]		0.53	0.08	0.46	0.49	0.50	0.54	2.14
Transaction cost [Gas]		44,827	1,265	36,739	45,139	45,235	45,259	45,319
Transaction cost [€]		4.96×10^{-3}	5.78×10^{-3}	2.96×10^{-11}	6.55×10^{-5}	6.53×10^{-3}	6.55×10^{-3}	1.37×10^{-1}
<i>Push-based outbound oracle</i> 438								
$dt_{tx-mined}$ [seconds]		16.20	15.95	0.53	5.41	10.71	21.44	129.95
<i>Pull-based inbound oracle</i> 126								
$dt_{tx-hash}$ [seconds]		0.52	0.05	0.46	0.48	0.50	0.52	0.78
Transaction cost [Gas]		22,770	0	22,770	22,770	22,770	22,770	22,770
Transaction cost [€]		8.91×10^{-5}	3.96×10^{-4}	7.91×10^{-7}	7.91×10^{-7}	7.91×10^{-7}	7.91×10^{-7}	1.85×10^{-3}
<i>Pull-based outbound oracle</i> 2611								
$dt_{tx-hash}$ [seconds]		0.13	0.03	0.11	0.11	0.12	0.12	0.45

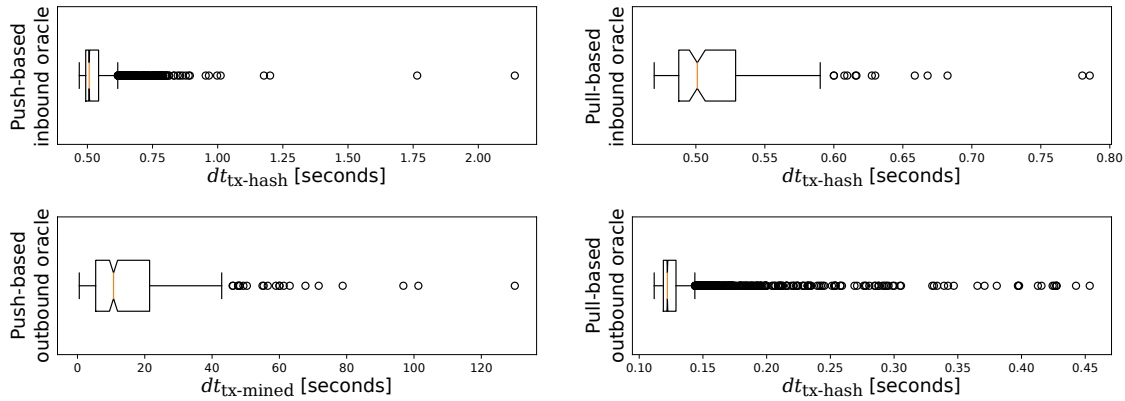


Figure 4.11.: The performance plots for the four oracle implementations.

4.7. Discussion and Threats to Validity

In the following, we discuss advantages as well as disadvantages of our work, our experiences from the implementation process, the results analysis above, and finally the limitations and threats to validity of our research.

One advantage of the classification of oracles into specific patterns, described in this chapter and the structuring based on sub-components therein, is the clear separation and aggregation of concerns that we can achieve. For example, our implementation, guided by the patterns in this chapter, allows us to implement logic for different levels of abstraction. This makes it possible to implement logic for basic concerns that apply to different oracles in a reusable way. Another advantage in this context, for example, is that adding or changing information sources for an oracle only requires modifying sub-components of the individual oracle without changing the

4. Oracle Patterns

implementation logic on the blockchain.

Regarding our analysis on time, our results show that the latency for the individual oracles is not particularly high. If we take for comparison the results of [109], where the mean commit time of transactions was about 200 seconds, we can say that the latency of less than one second (where no transaction lock-in time is part of the latency) measured in almost all cases is relatively low.

Regarding our analysis on costs, our results indicate that the execution costs are not particularly high either. According to our observations, a single interaction with any of the two inbound oracles does not generate high fees. In order to make an objective assessment, particular attention should be paid to gas consumption, which can be used as an absolute measure detached from current market prices. For this purpose, the results from [113] (a cost-optimized version of [90]) can be used, which suggest a typical gas consumption of 24,000 to 27,000 gas. The average gas consumption for the Pull-Based Inbound Oracle reflects this expected range, for the Push-Based Inbound Oracle the recorded higher gas consumption can be explained by the additional storage costs. Specific implementations of this pattern can be optimized in this regard, in particular by storing data only when needed in the chain. This may be particularly important if many oracle calls are expected in a given environment where the cost and time delays would add up in such a scenario.

The research presented in this chapter includes a number of limitations and threats to validity. The elaboration of patterns is based on a qualitative research approach, as is common in the field. Therefore, it is possible that misinterpretations and biases of individual researchers or the entire team of authors may have influenced the results. Regarding the generalizability of our research, this can only be claimed for the technologies studied (see Section 4.2). Through our efforts to define fundamental patterns, we have tried to minimize this risk as much as possible. Accordingly, we assume that our findings are applicable to other blockchain platforms. Nevertheless, we cannot claim any viability or form of completeness. For the time being, our conducted analysis can only provide a rough estimate regarding time performance and incurred costs; further measurements would be required to claim generalizability beyond the scope of the cases we studied. In this context, it should be noted that the use of a test network such as Ropsten may reduce the significance of the analysis results for practical applications. We have attempted to mitigate this by not relying on time and cost measurements from the testnet in our discussion and instead basing relevant cost analyses on data from the Ethereum mainnet.

4.8. Conclusion

In this chapter, we explored how blockchain oracles can be classified and characterized to connect the on-chain to the off-chain world. To this end, we abstracted individual technical solutions of existing implementations into four basic oracle patterns. In this context, we have elaborated

4.8. Conclusion

pros and cons of the individual patterns and conceptual structural building blocks that can be used to implement the patterns in a modular fashion. On this basis, we implemented the four patterns and analyzed them quantitatively in terms of temporal performance (latency) and cost. Our evaluations show that neither the latency nor the costs for a single invocation of any of the patterns are particularly high. Furthermore, the observed distributions in terms of latency suggest they can be narrowed down to a certain probable range in most cases, but can also be left by outliers and dominated by the transaction inclusion time.

In future work, we plan to explore the application of the patterns in the context of other blockchain platforms. Further, we plan to explore different strategies for data structures and communication flow to evaluate the impact of different approaches on the overall execution costs. In addition, it would be interesting to explore the combination of oracle patterns as well as the use of patterns for information exchange between blockchains.

5. Smart Contract Patterns

Smart contracts that build up on blockchain technologies are receiving great attention in new business applications and the scientific community, because they allow distrusted parties to manifest contract terms in program code and thus eliminate the need for a trusted third party. However, writing well performing and secure smart contracts in today's most prominent smart contract ecosystem Ethereum is a difficult task, due to the inherent nature of blockchain based contract execution, missing low level programming abstractions, and the constant evolution of platform features and security considerations. In order to provide design guidelines in this regard, this chapter presents a set of design patterns that have been mined by means of a MLR and an analysis of collected data based on qualitative research methods. The elicited patterns are described in detail with sample code for better illustration and can be applied by developers to address typical implementation issues.

5.1. Introduction

Ethereum as major blockchain-based ecosystem provides an environment to code and run smart contracts. However, writing smart contracts in Solidity, the predominant programming language on the platform, has so far been a difficult task associated with a number of problems. First, due to the inherent characteristics of blockchain-based program execution, rather unconventional programming paradigms are required. For example, programmers have to consider the lack of execution control and the immutable character of smart contracts once they are deployed. Second, the lack of low-level programming abstractions makes the developer responsible for the internal organization and manipulation of data at a deeper level. Third, the rapid change of platform features and security considerations requires continuous awareness of platform capabilities and potential security risks. On top of all this, smart contracts may handle considerable financial values, therefore it is crucial that their implementation is correct and secure against attacks. Given these points, it is beneficial to have a solid foundation of established design and coding guidelines that promote the creation of correct and secure smart contracts, for example in the form of design patterns. A design pattern describes an abstraction or conceptualization of a concrete, complex, and reoccurring problem that software designers have faced in the context of real software development projects and a successful solution they have implemented multiple times to resolve this problem [14]. This

5. Smart Contract Patterns

means that design patterns describe solutions to frequently occurring problems in a formalized way. Another, more explicit explanation for a *pattern* is that it is a proven *solution* to a *problem* in a *context*, resolving a set of *forces*. Here, the context refers to a recurring set of situations in which the pattern applies, while the problem refers to a set of goals associated with constraints, referred to as forces of the pattern, that typically occur in that context and influence the pattern's solution.

So far, design patterns have not received a lot of attention in Ethereum research and information on Solidity design and coding guidelines is scattered among different sources. To fill this gap, we address general design patterns for smart contracts in Ethereum in this chapter. Our research aims to answer the following two research questions: Which design patterns commonly appear in the Ethereum ecosystem? How do these design patterns map to Solidity coding practices? In order to answer these questions, we followed the MLR method by Garousi, Felderer, and Mäntylä [115] to incorporate practitioners' experience and applied an analysis of the gathered data based on qualitative research methods (namely GT [116] techniques to synthesize the patterns). Our research identified several patterns that highlight common problems in smart contract implementation and provide guidance on how to resolve them.

The chapter is organized in the following way: First, we present related work in Section 5.2, before we discuss the research study design in Section 5.3. Then we present design patterns for Solidity in Section Section 5.4, and discuss our findings in Section Section 5.5. Finally, we draw a conclusion in Section 5.6.

5.2. Related Work

According to Alharby and Moorsel [117] current research on smart contracts is mainly focused on identifying and tackling smart contract issues and can be divided into four categories, namely coding, security, privacy and performance issues. The technology behind writing smart contracts in Ethereum is still in its infancy, which is why coding and security are among the most discussed issues. Unfortunately, a lot of research and practical knowledge in this field is scattered throughout blog articles and gray literature, therefore information is often not very structured. Here, only few efforts have been made with the intention of collecting and categorizing patterns in a structured manner (see, e.g., [118], [119]). Relatively little work addresses software patterns in blockchain technology respectively design patterns in the Solidity language for the Ethereum ecosystem. A work with general scope on blockchain software development written by Xu, Weber, Staples, *et al.* [120] proposes a taxonomy of blockchain-based systems on architecture design. The elaborated taxonomy assists the evaluation and design of software architectures using blockchain technology and captures major architectural characteristics of blockchains to assess the impact of different design decisions. Another work by Bartoletti and Pompianu [121] conducted an empirical analysis

of Solidity contracts and identified a list of nine common design patterns that are shared by studied contracts. These patterns summarize the most frequent solutions to handle common usage scenarios and are named token, authorization, oracle, randomness, poll, time constraint, termination, math, and fork check. Yet another paper by Zhang, White, Schmidt, *et al.* [122] describes how the application of familiar software patterns can help to resolve design specific challenges. In particular, commonly known design patterns such as the Abstract Factory, Flyweight, Proxy, and Publisher-Subscriber pattern are applied to implement a blockchain-based healthcare application. Finally, a paper by Mavridou and Laszka [123] describes a framework for designing contracts as Finite-State Machine (FSM) utilizing design patterns for code generation.

5.3. Research Study Design

The process of identifying recurring design patterns can be a rather informal process, where an author defines a pattern based on his own experience and then tries to find similar or related pattern applications to confirm the initial pattern statement. However, to follow a systematic research methodology, we pursued a different path. That is to say, we applied a GT approach which is based on the paradigm of deploying a theory from the reality, that the theory is meant to explain. This means that our research is based on an inductive analysis of collected data which we systematically explore and investigate to extract a grounded pattern compilation. Due to a lack of academic literature regarding design patterns for Ethereum and Solidity to collect that data, we decided to carry out a MLR. A MLR is a form of SLR which includes “gray” literature (e.g., blogs, videos, and web pages) in addition to published “white” literature (e.g., academic journals, and conference papers) [4]. Figure 5.1 depicts the general process of our conducted MLR incorporating guidelines elaborated by Garousi, Felderer, and Mäntylä [4]. Starting from our research questions we defined initial search keywords as “ethereum”, “solidity”, “(smart) contract”, and “(software OR design) pattern”. These keyword combinations were then used to query different data sources for “white” and “gray” literature. The results were examined, i.e., citations and links were followed and reference lists were studied during a process called snowballing [124]. At the same time, initial search keywords were iteratively extended until theoretical saturation was reached. Next, the subsequent pool of sources was filtered according to predefined inclusion and exclusion criteria which encompassed to accept sources of any type that relate to Ethereum design patterns and exclude non-English works or works that seem unbalanced in presentation. Further, as Ethereum and Solidity have significantly evolved in recent years, we prioritized recent works. The accumulated final source pool contained among others the following important major sources. First, academic literature related to Ethereum and Solidity patterns [121]–[123]. Second, the official Solidity development documentation [28] and smart contract best practices [125]. Third, Internet blogs

5. Smart Contract Patterns

and discussion forums about Ethereum, such as the Ethereum community on Reddit [126], and the Ethereum QA section on StackExchange [127]. Forth, Ethereum conference talks [118], [128]. Fifth, existing GitHub repositories related to smart contract coding patterns in Solidity [119], [129], [130]. As next step the final source pool was reviewed and the extracted relevant information was analyzed with GT techniques, following recommendations by Stol, Ralph, and Fitzgerald [116]. In general, we took an iterative and pragmatic approach and recorded the concepts of our observations and insights in theoretical memos. These memos represented the actual pattern synthesis process and happened in several iterative stages, in which the patterns were constantly compared, revised, and contrasted until all the gathered information was accounted for.

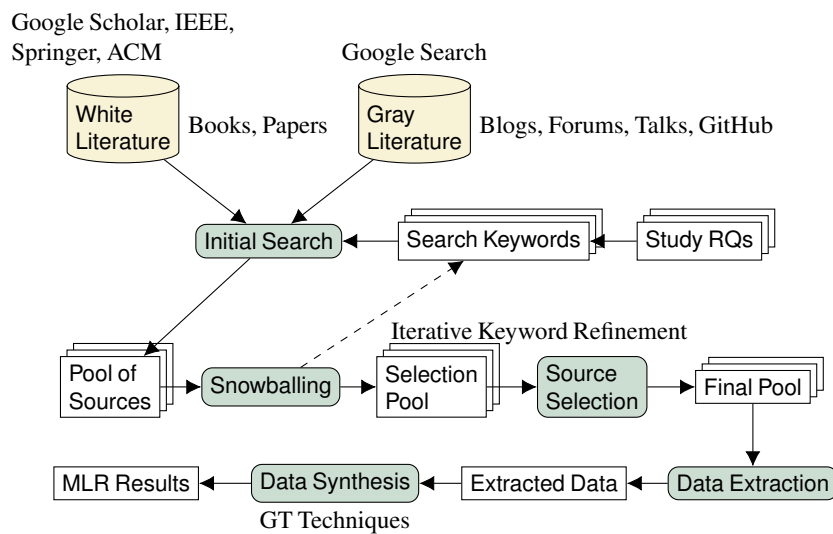


Figure 5.1.: An overview of the conducted MLR process.

5.4. Patterns

This section gives an overview of typical design patterns that are inherently frequent or practical in the context of smart contract coding in Ethereum. The presented patterns help to solve commonly recurring application requirements, or help to address typical problems and vulnerabilities related to smart contract implementation and execution. The gathered patterns are logically divided according to their operational scope into five categories: *a) Action and Control*, *b) Authorization*, *c) Lifecycle*, *d) Maintenance*, and *e) Security*. To better illustrate the context and applicability of the patterns, each pattern is explained in a problem- and solution-based approach with benefits and drawbacks along with sample code in Solidity, which is also available on GitHub [131]. To provide a concise overview of all patterns at the end of this section, Table 5.2 lists the pattern categories

and their associated patterns, including a brief description of the underlying pattern problem and its solution. To illustrate the patterns in practice, Table 5.1 also lists an example contract with published source code used on the Ethereum mainnet for each pattern.

5.4.1. Action and Control Patterns

Action and Control is a group of patterns that provide mechanisms for typical operational tasks.

Checks-Effects-Interaction

PATTERN	Checks-Effects-Interaction
Problem	When a contract calls another contract, it hands over control to that other contract. The called contract can then, in turn, re-enter the contract by which it was called and try to manipulate its state or hijack the control flow through malicious code.
Solution	Follow a recommended functional code order, in which calls to external contracts are always the last step.
Benefits	The attack surface of a contract relying on external calls is reduced, especially against re-entrant attacks, since multiple nested function calls are no longer possible.
Drawbacks	Since it is common practice in procedural programming languages to wait for feedback from a function execution (before making further changes based on its results), the use of the pattern is counter intuitive compared to common programming paradigms.

The Checks-Effects-Interaction pattern is fundamental for coding functions and describes how function code should be structured to avoid side effects and unwanted execution behavior. It defines a certain order of actions: First, check all the preconditions, then make changes to the contract's state, and finally interact with other contracts. Hence its name is "Checks-Effects-Interactions Pattern". According to this principle, interactions with other contracts should be, whenever possible, the very last step in any function, as seen in Listing 5.1. The reason being, that as soon as a contract interacts with another contract, including a transfer of Ether, it hands over the control to that other contract. This allows the called contract to execute potentially harmful actions. For example, a so-called re-entrancy attack, where the called contract calls back the current contract, before the first invocation of the function containing the call, was finished. This can lead to an

5. Smart Contract Patterns

unwanted execution behavior of functions, modifying the state variables to unexpected values or causing operations (e.g. sending of funds) to be performed multiple times. An example for a contract function, prone to the described attack scenario, is shown in Listing 5.2. The re-entrancy attack is especially harmful when using low level `address.call`, which forwards all remaining gas by default, giving the called contract more room for potentially malicious actions. Therefore, the use of low level `address.call` should be avoided whenever possible. For sending funds `address.send()` and `address.transfer()` should be preferred, these functions minimize the risk of re-entrancy through limited gas forwarding. While these methods still trigger code execution, the called contract is only given a stipend of 2,300 gas, which is currently only enough to log an event.

```
Solidity
```

```
function auctionEnd() public {
    // 1. Checks
    require(now >= auctionEnd);
    require(!ended);
    // 2. Effects
    ended = true;
    // 3. Interaction
    beneficiary.transfer(highestBid);
}
```

Listing 5.1: Application of the Checks-Effects-Interaction pattern within a function.

```
Solidity
```

```
contract SimpleDeposit {
    ...
}

function withdrawBalance() public {
    uint amount = balances[msg.sender];
    require(msg.sender.call.value
        (amount)()); // caller's code is executed and can re-enter withdrawBalance again
    balances[msg
        .sender] = 0; // INSECURE - user's balance must be reset before the external call
}
```

Listing 5.2: An example of an insecure withdrawal function prone to a re-entrancy attack.

Pull Payment

PATTERN	Pull Payment
Problem	When a contract sends funds to another party, the send operation can fail.
Solution	Let the receiver of a payment withdraw the funds.
Benefits	Problems that can arise in connection with sending funds are mitigated, especially when multiple transfers are performed at once since a failed transfer no longer causes all successful operations to be undone.
Drawbacks	It is the responsibility of the payee to receive payments which leads to an additional step, namely requesting (withdrawing) the payment. This not only leads to higher transaction costs, but also negatively impacts the user experience by making it less convenient for the payee.

A very common task when coding smart contracts is to send funds to another party. Unfortunately there are several circumstances under which a transfer of funds can fail. This is due to the fact that the implementation to send funds involves an external call, thus the same security considerations regarding external calls and re-entrancy attacks, mentioned afore, apply.

Currently, there are three different methods to transfer funds in Solidity. These are `address.send()`, `address.transfer()`, and `address.call.value()`. If the payment recipient is a contract, calling these methods triggers the execution of a so-called fallback function in the receiver contract. Per definition, the fallback function is a name- and parameterless function, that is called when the function signature does not match any of the available functions in a Solidity contract. Since `send()` specifies a blank function signature, it will always trigger the fallback function if it exists. `x.transfer(y)` is equivalent to `require(x.send(y))`; and defines a maximum stipend of 2,300 gas, given to the receiver contract for execution, which is currently only enough to log an event. `address.call.value()` gives all available gas to the receiving contract for execution, which makes this type of value transfer unsafe against re-entrancy. So, the difference between `send()` and `address.call.value()` is how much gas is made available to the fallback function in the receiving contract.

Due to the possibility of deliberately sabotaging the transfer of funds by executing expensive operations in the fallback method, causing an “out of gas” error, or manipulations involving re-entrancy attacks, a more favorable approach is to reverse the payment process. Namely, let users withdraw their funds themselves, rather than sending funds directly. Listing 5.3 shows a problematic reliance on a successful transfer of funds, whereas Listing 5.4 mitigates this problem

5. Smart Contract Patterns

by isolating the external call into its own transaction that can be initiated by the recipient of the call. Overall, it is advisable to favor pull over push payments when coding smart contracts.

```
Solidity
```

```
pragma solidity ^0.4.17;

contract Auction {
    address public highestBidder;
    uint highestBid;

    function bid() public payable {
        require(msg.value >= highestBid);
        if (highestBidder != 0) {
            // if call fails causing a rollback,
            // no one else can bid
            highestBidder.transfer(highestBid);
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
    }
}
```

Listing 5.3: An intuitive solution in an auction contract would be to push a payment to a defeated bidder once a higher bid has been received.

```
Solidity
```

```
pragma solidity ^0.4.17;

contract Auction {
    address public highestBidder;
    uint highestBid;
    mapping(address => uint) refunds;

    function bid() public payable {
        require(msg.value >= highestBid);
        if (highestBidder != 0) {
            // record the underlying bid to be refund
            refunds[highestBidder] += highestBid;
        }
    }
}
```

```

    highestBidder = msg.sender;
    highestBid = msg.value;
}

function withdrawRefund() public {
    uint refund = refunds[msg.sender];
    refunds[msg.sender] = 0;
    msg.sender.transfer(refund);
}
}

```

Listing 5.4: Introducing a refunds mapping, which stores the claimable defeated bids, to be withdrawn by participants in a pull payment process.

State Machine

PATTERN	State Machine
Problem	An application scenario implicates different behavioral stages and transitions.
Solution	Apply a state machine to model and represent different behavioral contract stages and their transitions.
Benefits	Mapping contract behavior using a state machine helps to reduce and simplify the logical complexity of the implementation.
Drawbacks	Implementing a large number of behavioral phases and transitions is difficult to handle without a clear implementation respectively design concept.

A state machine models the behavior of a system based on its history and current inputs. Developers use this construct to break complex problems into simple states and state transitions. These states and state transitions are then used to represent and control the execution flow of a program. A state machine is often a very compact way to represent a set of complex rules and conditions and usually helps to reduce and simplify the logical complexity of the implementation. State machines can also be applied in smart contracts, exemplified in Listing 5.5. Many usage scenarios require a contract to have different behavioral stages, in which different functions can be called. When interacting with such a contract, a function call might end the current contract stage and initiate a transition into a consecutive stage.

```

pragma solidity ^0.4.17;

contract DepositLock {
    enum Stages {
        AcceptingDeposits,
        FreezingDeposits,
        ReleasingDeposits
    }
    Stages public stage = Stages.AcceptingDeposits;
    uint public creationTime = now;
    mapping (address => uint) balances;

    modifier atStage(Stages _stage) {
        require(stage == _stage);
        _;
    }

    modifier timedTransitions() {
        if (stage == Stages.AcceptingDeposits && now >= creationTime + 1 days)
            nextStage();
        if (stage == Stages.FreezingDeposits && now >= creationTime + 8 days)
            nextStage();
        _;
    }

    function nextStage() internal {
        stage = Stages(uint(stage) + 1);
    }

    function
        deposit() public payable timedTransitions atStage(Stages.AcceptingDeposits) {
        balances[msg.sender] += msg.value;
    }

    function withdraw() public timedTransitions atStage(Stages.ReleasingDeposits) {
        uint amount = balances[msg.sender];
        balances[msg.sender] = 0;
        msg.sender.transfer(amount);
    }
}

```

Listing 5.5: A contract based on a state machine to represent a deposit lock, which accepts deposits for a period of one day and releases them after seven days.

Commit and Reveal

PATTERN	Commit and Reveal
Problem	All data and every transaction is publicly visible on the blockchain, but an application scenario requires that contract interactions, specifically submitted parameter values, are treated confidentially.
Solution	Apply a commitment scheme to ensure that a value submission is binding and concealed until a consolidation phase runs out, after which the value is revealed, and it is publicly verifiable that the value remained unchanged.
Benefits	Application scenarios based on the transmission of confidential information can be carried out publicly in a generally verifiable manner.
Drawbacks	Under this scheme, two separate transactions are required, resulting in higher transaction costs. In addition, such a methodology is not readily comprehensible to lay users.

A characteristic of blockchains is that it is not possible to restrict any human or computer from reading contents of a transaction or transaction's state. Anyone can observe all previous data and changes in the blockchain. Solidity variables can be annotated as being `private`, but this only prevents other contracts from accessing data directly, any other party can still read data in the blockchain.

There are many use cases where this transparency leads to problems, especially when contract participants compete with each other, for example at an auction or a game. Knowledge about the behavior of other contract participants can give a party a considerable advantage. Thus, it is necessary to take measures in order to obfuscate actions and contract inputs. Such a measure is a cryptographic commitment scheme. In principle, it allows a party to commit to a selected value while keeping it concealed to others, with the ability to reveal the committed value later. The commitment scheme is specifically designed to prohibit a party from changing the value or statement after it committed to it. The whole process can be divided into two parts. First, the commit phase in which a party commits to a value that is hidden for other participants. Second, the reveal phase in which the previously committed value is revealed, and everyone can verify that it is in fact, what one committed to. In the context of smart contracts, this is achieved by hashing a chosen value with a secret (e.g. random string) and sending it to the contract. Later, this commitment is revealed by sending the value and the secret in plain text, thus it can be verified that the value and the secret yield the previously committed hash. A contract implementing the described strategy is shown in Listing 5.6.

```

pragma solidity ^0.4.17;

contract CommitReveal {
    struct Commit {string choice; string secret; string status;}
    mapping(address => mapping(bytes32 => Commit)) public userCommits;

    event LogCommit(bytes32, address);
    event LogReveal(bytes32, address, string, string);

    function CommitReveal() public {}

    function commit(bytes32 _commit) public returns (bool success) {
        var userCommit = userCommits[msg.sender][_commit];
        if(bytes(userCommit.status).length != 0) {
            return false; // commit has been used before
        }
        userCommit.status = "c"; // comitted
        LogCommit(_commit, msg.sender);
        return true;
    }

    function reveal
        (string _choice, string _secret, bytes32 _commit) public returns (bool success) {
        var userCommit = userCommits[msg.sender][_commit];
        bytes memory bytesStatus = bytes(userCommit.status);
        if(bytesStatus.length == 0) {
            return false; // choice not committed before
        } else if (bytesStatus[0] == "r") {
            return false; // choice already revealed
        }
        if (_commit != keccak256(_choice, _secret)) {
            return false; // hash does not match commit
        }
        userCommit.choice = _choice;
        userCommit.secret = _secret;
        userCommit.status = "r"; // revealed
        LogReveal(_commit, msg.sender, _choice, _secret);
        return true;
    }

    function traceCommit(address _address, bytes32
        _commit) public view returns (string choice, string secret, string status) {

```

```

var userCommit = userCommits[_address][_commit];
require(bytes(userCommit.status)[0] == "r");
return (userCommit.choice, userCommit.secret, userCommit.status);
}
}

```

Listing 5.6: A contract that allows a party to commit to a choice and reveal it at a later point in time, traceable for anyone.

Oracle (Data Provider)

PATTERN	Oracle (Data Provider)
Problem	An application scenario requires knowledge contained outside the blockchain, but Ethereum contracts cannot directly acquire information from the outside world. On the contrary, they rely on the outside world pushing information into the network.
Solution	Request external data through an oracle service that is connected to the outside world and acts as a data carrier.
Benefits	As outside information is requested from the blockchain the data gathering process is transparent. It can be traced whether off-chain data was successfully provided (in time) or not.
Drawbacks	Oracles require a level of trust that contradicts the trustless and decentralized nature of blockchains, i.e. verifying the reliability of extrinsic information is problematic. Further, an oracle depends on event and transaction processing on the blockchain, which means that the oracle's response time depends on the speed of the blockchain.

Ethereum contracts run within their own ecosystem, where they communicate with each other and store and read data from the blockchain, but external data can only enter the system through outside interaction via a transaction (by passing data to a method). This is a drawback, because many contract use cases depend on external knowledge contained outside the blockchain (e.g. price feeds, weather data). A solution to this problem is to utilize oracles which have a connection to the outside world. Contracts depending on outside information can then request the necessary data from them. The oracle service acts as a data carrier, where the interaction between an oracle

5. Smart Contract Patterns

service and an Ethereum smart contract is asynchronous. First, a transaction invokes a function of a smart contract that contains an instruction to send a request to an oracle. Then, according to the parameters of such a request, the oracle will fetch a result and return it by executing a callback function placed in the primary contract. The described procedure involving an oracle contract and its consumer contract is illustrated by Listing 5.7 and Listing 5.8. In relation to the oracle patterns discussed in Chapter 4, this implementation represents an Pull-Based Inbound Oracle.

A shortcoming of oracles is that they contradict the blockchain theorem of a decentralized network, because contracts utilizing a sole oracle rely on a single party or group to be honest. Currently operating oracle services [103], [132] address this shortcoming by accompanying the resulting data with a proof of authenticity, that shows that the data fetched from the original data source is genuine and untampered. These authenticity proofs are based on auditable virtual machines and Trusted Execution Environment (TEE). Further, it should be noted that the oracle has to pay for the callback invocation, therefore an oracle usually requires a contract to pay an oracle fee, plus the Ether necessary to pay for the callback transaction.

```
Solidity  
  
pragma solidity ^0.4.17;  
  
contract Oracle {  
    address knownSource = 0x123...; // known source  
    struct Request {  
        bytes data;  
        function(bytes memory) external callback;  
    }  
    Request[] requests;  
  
    event NewRequest(uint);  
  
    modifier onlyBy(address account) {  
        require(msg.sender == account); _;  
    }  
  
    function query(bytes data, function(bytes memory) external callback) public {  
        requests.push(Request(data, callback));  
        NewRequest(requests.length - 1);  
    }  
  
    // invoked by outside world  
    function reply(uint requestID, bytes response) public onlyBy(knownSource) {
```



```

    requests[requestID].callback(response);
  }
}

```

Listing 5.7: An oracle contract that allows to request data from outside the blockchain.

Solidity

```

pragma solidity ^0.4.17;

import "./Oracle.sol";
contract OracleConsumer {
    Oracle oracle = Oracle(0x123...); // known contract

    modifier onlyBy(address account) {
        require(msg.sender == account); _;
    }

    function updateExchangeRate() {
        oracle.query("USD", this.oracleResponse);
    }

    function oracleResponse(bytes response) onlyBy(oracle) {
        // use the data
    }
}

```

Listing 5.8: An oracle consumer contract implementing a callback method to receive result data.

5.4.2. Authorization Patterns

Authorization is a group of patterns that control access to smart contract functions and provide basic authorization control, which simplify the implementation of “user permissions”.

Ownership

PATTERN **Ownership**

5. Smart Contract Patterns

Problem	By default any party can call a contract method, but it must be ensured that sensitive contract methods can only be executed by the owner of a contract.
Solution	Store the contract creator's address as owner of a contract and restrict method execution with a general modifier that checks the callers address.
Benefits	Allows easy annotation of functions that may only be executed by the contract owner.
Drawbacks	When using general modifiers, the execution flow jumps from one code line to a completely different section, which can make it difficult to track and review the code.

It is very common that only the owner of a contract should be eligible to call functions, which are sensitive and crucial for the correct operation of the contract. This pattern limits access to certain functions to only the owner of the contract; an example is shown in Listing 5.9. A typical application of this pattern is demonstrated in the Mortal pattern.

```
Solidity
```

```
pragma solidity ^0.4.17;

contract Owned {
    address public owner;

    event
        LogOwnershipTransferred(address indexed previousOwner, address indexed newOwner);

    modifier onlyOwner() {
        require(msg.sender == owner);
        -;
    }

    function Owned() public {
        owner = msg.sender;
    }

    function transferOwnership(address newOwner) public onlyOwner {
        require(newOwner != address(0));
        LogOwnershipTransferred(owner, newOwner);
        owner = newOwner;
    }
}
```

```

}
}

```

Listing 5.9: A simple contract to track the ownership of a contract.

Access Restriction

PATTERN	Access Restriction
Problem	By default a contract method is executed without any preconditions being checked, but it is desired that the execution is only allowed if certain requirements are met.
Solution	Define generally applicable modifiers that check the desired requirements and apply these modifiers in the function definition.
Benefits	When using general modifiers, they can be easily adapted or combined for different situations, which makes them highly reusable.
Drawbacks	When using general modifiers, the execution flow jumps from one code line to a completely different section, which can make it difficult to track and review the code.

Since there is no built in mechanism to control execution privileges, a common pattern is to restrict function execution. It is often required that functions should only be executed based on the presence of certain prerequisites. These prerequisites can refer to different categories, such as temporal conditions, caller and transaction info, or other requirements that need to be checked prior a function execution. Listing 5.10 illustrates how different prerequisites can be checked prior function execution.

Solidity

```

pragma solidity ^0.4.17;

import "./Ownership.sol";

contract AccessRestriction is Owned {
    uint public creationTime = now;

    modifier onlyBefore(uint _time) {

```

5. Smart Contract Patterns

```
    require(now < _time); _;
}

modifier onlyAfter(uint _time) {
    require(now > _time); _;
}

modifier onlyBy(address account) {
    require(msg.sender == account); _;
}

modifier condition(bool _condition) {
    require(_condition); _;
}

modifier minAmount(uint _amount) {
    require(msg.value >= _amount); _;
}

function f() payable onlyAfter(creationTime + 1 minutes
    ) onlyBy(owner) minAmount(2 ether) condition(msg.sender.balance >= 50 ether) {
    // some code
}
}
```

Listing 5.10: A contract demonstrating how to check certain requirements prior to function execution.

5.4.3. Lifecycle Patterns

Lifecycle is a group of patterns that control the creation and destruction of smart contracts.

Mortal

PATTERN	Mortal
Problem	A deployed contract will exist as long as the Ethereum network exists. If a contract's lifetime is over, it must be possible to destroy a contract and stop it from operating.

Solution	Use a self destruct call within a method that does a preliminary authorization check of the invoking party.
Benefits	Allows for the simple proclamation that a contract has reached the end of its life.
Drawbacks	As long as there is no binding implementation on how the destruction mechanism can be triggered, there is always the possibility that the mechanism will be maliciously abused to extract contract funds.

A contract is defined by its creator, but the execution, and subsequently the services it offers are provided by the Ethereum network itself. Thus, a contract will exist and be executable as long as the whole network exists, and will only disappear if it was programmed to self destruct. Mortal is a pattern that enables the creator of a contract to destroy it. The pattern uses a modifier to ensure that only the owner of the contract can execute the `selfdestruct` operation, which sends the remaining Ether stored within the contract to a designated target address (provided as argument) and then the storage and code is cleared from the current state. Listing 5.11 exemplifies the application of this pattern.

Solidity

```

pragma solidity ^0.4.17;

import "../authorization/Ownership.sol";

contract Mortal is Owned {
    function destroy() public onlyOwner {
        selfdestruct(owner);
    }

    function destroyAndSend(address recipient) public onlyOwner {
        selfdestruct(recipient);
    }
}

```

Listing 5.11: A contract that provides its creator with the ability to destroy it.

Automatic Deprecation

PATTERN Automatic Deprecation

5. Smart Contract Patterns

Problem	A usage scenario requires a temporal constraint defining a point in time when functions become deprecated.
Solution	Define an expiration time and apply modifiers in function definitions to disable function execution if the expiration date has been reached.
Benefits	When using general modifiers, it is easy to identify which functions will be deprecated.
Drawbacks	When using general modifiers, the execution flow jumps from one code line to a completely different section, which can make it difficult to track and review the code.

Automatic deprecation is a pattern that allows to automatically prohibit the execution of functions after a specific time period has elapsed. Listing 5.12 shows the automatic deprecation of functions based on an elapsed time period.

```
Solidity
```

```
pragma solidity ^0.4.17;

contract AutoDeprecate {
    uint expires;

    function AutoDeprecate(uint _days) public {
        expires = now + _days * 1 days;
    }

    function expired() internal view returns (bool) {
        return now > expires;
    }

    modifier willDeprecate() {
        require(!expired());
        _;
    }

    modifier whenDeprecated() {
        require(expired());
        _;
    }
}
```

```

function deposit() public payable willDeprecate {
    // some code
}

function withdraw() public view whenDeprecated {
    // some code
}
}

```

Listing 5.12: A contract interface that automatically deprecates after a specified time period has elapsed.

Solidity

```

pragma solidity ^0.4.17;

contract AutoDeprecate {
    uint public constant BLOCK_NUMBER = 4400000;

    modifier isActive() {
        require(block.number <= BLOCK_NUMBER);
        -;
    }

    function deposit() public isActive() {
        // some code
    }

    function withdraw() public {
        // some code
    }
}

```

Listing 5.13: A contract interface that automatically deprecates after a certain block number has been reached.

5.4.4. Maintenance Patterns

Maintenance is a group of patterns that provide mechanisms for live operating contracts. In contrast to ordinary distributed applications, which can be updated when bugs are detected, smart contracts

5. Smart Contract Patterns

are irreversible and immutable. This means that there is no way to update a smart contract, other than writing an improved version that is then deployed as new contract.

Data Segregation

PATTERN	Data Segregation
Problem	Contract data and its logic are usually kept in the same contract, leading to a closely entangled coupling. Once a contract is replaced by a newer version, the former contract data must be migrated to the new contract version.
Solution	Decouple the data from the operational logic into separate contracts.
Benefits	No data migration is required after upgrading a smart contract.
Drawbacks	Separating logic from storage adds complexity by requiring external calls to be made, which must be handled with care because they can cause unintended behavior. In addition, bypassing immutability can affect user trust in a contract if version change policies are not implemented.

The data segregation pattern separates contract logic from its underlying data. Segregation promotes the separation of concerns and mimics a layered design (e.g. logic layer, data layer). Following this principle avoids costly data migrations when code functionality changes. Meaning a new contract version would not have to recreate all of the existing data contained in the previous contract. The separation of contract data and contract logic is shown in Listing 5.14 and Listing 5.15. It is favorable to design the storage contract very generic so that once it is created, it can store and access different types of data with the help of setter and getter methods.

```
Solidity
```

```
pragma solidity ^0.4.17;

contract DataStorage {
    ...
}

function getUintValue(bytes32 key) public constant returns (uint) {
    return uintStorage[key];
}
```



```

function setUintValue(bytes32 key, uint value) public {
    uintStorage[key] = value;
}
...
}
}

```

Listing 5.14: The data is separated in its own contract.

Solidity

```

pragma solidity ^0.4.17;

import "./DataStorage.sol";
contract Logic {
    DataStorage dataStorage;

    function Logic(address _address) public {
        dataStorage = DataStorage(_address);
    }

    function f() public {
        bytes32 key = keccak256("emergency");
        dataStorage.setUintValue(key, 911);
        dataStorage.getUintValue(key);
    }
}

```

Listing 5.15: The contract logic can manipulate the data through a reference.

Satellite

PATTERN	Satellite
Problem	Contracts are immutable. Changing contract functionality requires the deployment of a new contract.

5. Smart Contract Patterns

Solution	Outsource functional units that are likely to change into separate so-called satellite contracts and use a reference to these contracts in order to utilize needed functionality.
Benefits	Altering contract functionality can be easily done through deploying new satellite contracts.
Drawbacks	Without implemented rules for functional changes, user trust in a contract can diminish as new satellite contracts may introduce potentially undesirable behavior.

The satellite pattern allows to modify and replace contract functionality. This is achieved through the creation of separate satellite contracts that encapsulate certain contract functionality. The addresses of these satellite contracts are stored in a base contract. This contract can then call out to the satellite contracts when it needs to reference certain functionalities, by using the stored address pointers. If this pattern is properly implemented, modifying functionality is as simple as creating new satellite contracts and changing the corresponding satellite addresses. Listing 5.16 and Listing 5.17 exemplify the application of this pattern.

```
Solidity
```

```
pragma solidity ^0.4.17;

contract Satellite {
    function calculateVariable() public pure returns (uint){
        // calculate var
        return 2 * 3;
    }
}
```

Listing 5.16: A satellite contract encapsulates certain contract functionalities.

```
Solidity
```

```
pragma solidity ^0.4.17;

import "../authorization/Ownership.sol";
import "./Satellite.sol";

contract Base is Owned {
```

```

uint public variable;
address satelliteAddress;

function setVariable() public onlyOwner {
    Satellite s = Satellite(satelliteAddress);
    variable = s.calculateVariable();
}

function updateSatelliteAddress(address _address) public onlyOwner {
    satelliteAddress = _address;
}
}

```

Listing 5.17: A base contract referring to a satellite contract in order to fulfill its purpose. The use of a satellite allows an easy contract functionality modification.

Contract Register

PATTERN	Contract Register
Problem	Contract participants must be referred to the latest contract version.
Solution	Let contract participants pro-actively query the latest contract address through a register contract that returns the address of the most recent version.
Benefits	Contract participants can work with the latest contract version and are aware about possible changes as soon as a new address is returned.
Drawbacks	Contract participants are responsible for tracking the latest contract version, otherwise they run the risk of working with an outdated version.

The register pattern is an approach to handle the update process of a contract. The pattern keeps track of different versions (addresses) of a contract and points on request to the latest one, as seen in Listing 5.18. In conclusion, before interacting with a contract, a user would always have to query the register for the contract's latest address. Failing to do so would risk interacting with an old version of the contract. When following this update approach, it is also important to determine how to handle existing contract data, when an old contract version is replaced. An alternative solution to point to the latest contract address would be to utilize the Ethereum Name Service (ENS). It is a register that enables a secure and decentralized way to resolve human-readable names, like 'mycontract.eth', into machine-readable identifiers, including Ethereum addresses.

5. Smart Contract Patterns

```
Solidity
```

```
pragma solidity ^0.4.17;

import "../authorization/Ownership.sol";

contract Register is Owned {
    address backendContract;
    address[] previousBackends;

    function Register() public {
        owner = msg.sender;
    }

    function changeBackend(address newBackend) public onlyOwner() returns (bool) {
        if(newBackend != backendContract) {
            previousBackends.push(backendContract);
            backendContract = newBackend;
            return true;
        }
        return false;
    }
}
```

Listing 5.18: A register contract to store the latest version of a contract.

Contract Relay

PATTERN	Contract Relay
Problem	Contract participants must be referred to the latest contract version.
Solution	Contract participants always interact with the same proxy contract that relays all requests to the most recent contract version.
Benefits	Contracts can be updated without changing anything for contract participants.

Drawbacks Contract complexity is increased, e.g. by the concepts of delegate calls and inline assembly, which increases the probability of introducing errors. In addition, updates must take into account contract storage layout limitations, i.e. existing fields can neither be deleted nor rearranged. Furthermore, user trust in a contract can diminish as new contract versions may introduce potentially undesirable behavior.

A relay is another approach to handle the update process of a contract. The relay pattern provides a method to update a contract to a newer version while keeping the old contract address. This is achieved by using a kind of proxy contract that forwards calls and data to the latest version of the contract, shown in Listing 5.19. This approach can forward function calls including their arguments, but cannot return result values. Another drawback of this approach is that the data storage layout needs to be consistent in newer contract versions, otherwise data may be corrupted.

```

Solidity

pragma solidity ^0.4.17;

import "../authorization/Ownership.sol";

contract Relay is Owned {
    address public currentVersion;

    function Relay(address initAddr) public {
        currentVersion = initAddr;
        owner = msg.sender;
    }

    function changeContract(address newVersion) public onlyOwner() {
        currentVersion = newVersion;
    }

    // fallback function
    function() public {
        require(currentVersion.delegatecall(msg.data));
    }
}

```

Listing 5.19: A relay contract to forward data and calls.

5.4.5. Security Patterns

Security is a group of patterns that introduce safety measures to mitigate damage and assure a reliable contract execution.

Emergency Stop (Circuit Breaker)

PATTERN	Emergency Stop (Circuit Breaker)
Problem	Since a deployed contract is executed autonomously on the Ethereum network, there is no option to halt its execution in case of a major bug or security issue.
Solution	Incorporate an emergency stop functionality into the contract that can be triggered by an authenticated party to disable sensitive functions.
Benefits	Allows to easily halt any sensitive contract functionality in case of an emergency.
Drawbacks	Can lead to unpredictable contract behavior unless a set of rules is implemented to govern when the stop mechanism can be triggered.

Reliably working contracts may contain bugs that are yet unknown, until revealed by an adversary attack. One countermeasure and a quick response to such attacks are emergency stops or circuit breakers. They stop the execution of a contract or its parts when certain conditions are met. A recommended scenario would be, that once a bug is detected, all critical functions would be halted, leaving only the possibility to withdraw funds. A contract implementing the described strategy is shown in Listing 5.20. The ability to fire an emergency stop could be either given to a certain party, or handled through the implementation of a rule set.

```
Solidity  
  
pragma solidity ^0.4.17;  
  
import "../authorization/Ownership.sol";  
  
contract EmergencyStop is Owned {  
    bool public contractStopped = false;  
  
    modifier haltInEmergency {  
        if (!contractStopped) _;  
    }  
}
```

```

modifier enableInEmergency {
    if (contractStopped) _;
}

function toggleContractStopped() public onlyOwner {
    contractStopped = !contractStopped;
}

function deposit() public payable haltInEmergency {
    // some code
}

function withdraw() public view enableInEmergency {
    // some code
}
}

```

Listing 5.20: An emergency stop allows to disable or enable specific functions inside a contract in case of an emergency.

Speed Bump

PATTERN	Speed Bump
Problem	The simultaneous execution of sensitive tasks by a huge number of parties can bring about the downfall of a contract.
Solution	Prolong the completion of sensitive tasks to take steps against fraudulent activities.
Benefits	Allows to slow down the execution of tasks according to specific requirements.
Drawbacks	Slowing down function execution complicates implementation, which increases the probability for introducing errors.

Contract sensitive tasks are slowed down on purpose, so when malicious actions occur, the damage is restricted and more time to counteract is available. An analogous real world example would be a bank run, where a large number of customers withdraw their deposits simultaneously due to concerns about the bank's solvency. Banks typically counteract by delaying, stopping, or limiting the amount of withdrawals. An example contract implementing a withdrawal delay is

5. Smart Contract Patterns

shown in Listing 5.21.

```


Solidity



```
pragma solidity ^0.4.17;

contract SpeedBump {
 struct Withdrawal {
 uint amount;
 uint requestedAt;
 }
 mapping (address => uint) private balances;
 mapping (address => Withdrawal) private withdrawals;
 uint constant WAIT_PERIOD = 7 days;

 function deposit() public payable {
 if (!(withdrawals[msg.sender].amount > 0))
 balances[msg.sender] += msg.value;
 }

 function requestWithdrawal() public {
 if (balances[msg.sender] > 0) {
 uint amountToWithdraw = balances[msg.sender];
 balances[msg.sender] = 0;
 withdrawals[msg.sender] = Withdrawal({
 amount: amountToWithdraw,
 requestedAt: now
 });
 }
 }

 function withdraw() public {
 if (withdrawals[msg.sender].amount > 0 && now > withdrawals[msg.sender].requestedAt + WAIT_PERIOD) {
 uint amount = withdrawals[msg.sender].amount;
 withdrawals[msg.sender].amount = 0;
 msg.sender.transfer(amount);
 }
 }
}
```


```

Listing 5.21: A contract that delays the withdrawal of funds deliberately.

Rate Limit

PATTERN	Rate Limit
Problem	A request rush on a certain task is not desired and can hinder the correct operational performance of a contract.
Solution	Regulate how often a task can be executed within a period of time with a general modifier.
Benefits	Allows easy control of how many times a given task can be performed within a given time period.
Drawbacks	Block timestamps can be affected to some degree by miners, which must be taken into account when applying a rate limit for function execution.

A rate limit regulates how often a function can be called consecutively within a specified time interval. This approach may be used for different reasons. A usage scenario for smart contracts may be founded on operative considerations, in order to control the impact of (collective) user behavior. As an example one might limit the withdrawal execution rate of a contract to prevent a rapid drainage of funds. Listing 5.22 exemplifies the application of this pattern.

Solidity

```

pragma solidity ^0.4.17;

contract RateLimit {
    uint enabledAt = now;

    modifier enabledEvery(uint t) {
        if (now >= enabledAt) {
            enabledAt = now + t;
        }
        _;
    }

    function f() public enabledEvery(1 minutes) {
        // some code
    }
}

```

Listing 5.22: A contract with a rate limit that avoids excessively repetitive function execution.

Mutex

PATTERN	Mutex
Problem	Re-entrancy attacks can manipulate the state of a contract and hijack the control flow.
Solution	Utilize a mutex in the form of a general modifier to hinder an external call from re-entering its caller function again.
Benefits	Allows to easily annotate any function to make it safe against re-entrancy attacks.
Drawbacks	When using general modifiers, the execution flow jumps from one code line to a completely different section, which can make it difficult to track and review the code.

A mutex (from mutual exclusion) is known as a synchronization mechanism in computer science to restrict concurrent access to a resource. After re-entrancy attack scenarios emerged, this pattern found its application in smart contracts to protect against recursive function calls from external contracts. An example contract is depicted below in Listing 5.23.

```


Solidity



```
pragma solidity ^0.4.17;

contract Mutex {
 bool locked;

 modifier noReentrancy() {
 require(!locked);
 locked = true;
 _;
 locked = false;
 }

 // f is protected by a mutex, calls from within msg.sender.call cannot call f again
 function f() noReentrancy public returns (uint) {
 require(msg.sender.call());
 return 1;
 }
}
```


```

Listing 5.23: A contract implementing a mutex pattern to avoid re-entrancy.

Balance Limit

PATTERN	Balance Limit
Problem	There is always a risk that a contract gets compromised due to bugs in the code or yet unknown security issues within the contract balance platform.
Solution	Limit the maximum amount of funds at risk by checking each deposit with a general modifier for not exceeding a specified overall contract limit.
Benefits	Limits the amount of possible losses.
Drawbacks	The approach cannot prevent the admission of forcibly sent Ether (e.g. mining rewards).

It is generally a good idea to manage the amount of money at risk when coding smart contracts. This can be achieved by limiting the total balance held within a contract. The pattern monitors the contract balance and rejects payments sent along a function invocation after exceeding a predefined quota, as seen in Listing 5.24. It should be noted that this approach cannot prevent the admission of forcibly sent Ether, e.g. as beneficiary of a `selfdestruct(address)` call, or as recipient of a mining reward.

Solidity

```

pragma solidity ^0.4.17;

contract LimitBalance {
    uint256 public limit;

    function LimitBalance(uint256 value) public {
        limit = value;
    }

    modifier limitedPayable() {
        require(this.balance <= limit);
        _;
    }

    function deposit() public payable limitedPayable {
        // some code
    }

```

5. Smart Contract Patterns

```
}

```

Listing 5.24: A contract limiting the total balance acquirable with payable function invocation.

Table 5.1.: Pattern usage examples in published source code contracts on the Ethereum mainnet.

Category	Pattern	Example Contract
Action and Control	Checks-Effects-Interaction	CryptoKitties
	Pull Payment	Cryptopunks
	State Machine	DutchAuction
	Commit and Reveal	ENS Registrar
	Oracle (Data Provider)	Etheroll
Authorization	Ownership	Ethereum Lottery
	Access Restriction	Etheroll
Lifecycle	Mortal	GTA Token
	Automatic Deprecation	Polkadot
Maintenance	Data Segregation	SAN Token
	Satellite	LATP Token
	Contract Register	Tether Token
	Contract Relay	Numeraire
Security	Emergency Stop	Augur/REP
	Speed Bump	TheDAO
	Rate Limit	etherep
	Mutex	Ventana Token
	Balance Limit	CATToken

5.5. Discussion

Our research covers 18 patterns grouped into five categories. Although some patterns are very basic, their real practical value is unfolded when patterns are combined. In this context, an examination of the patterns reveals a certain hierarchy structure, meaning some of the patterns act as foundation for others. For example, the Ownership pattern is often used as a prerequisite in combination with

Table 5.2.: A concise overview of smart contract design patterns.

	Pattern	Summary
Action & Control	Checks-Effects-Interaction	As calls to other contracts hand over control, avoid security issues by a functional code order.
	Pull Payment	As a send operation can fail, let the receiver withdraw the payment.
	State Machine	When different contract stages are needed, these are modeled and represented by a state machine.
	Commit and Reveal	As blockchain data is public, a commitment scheme ensures confidentiality of contract interactions.
	Oracle (Data Provider)	When knowledge outside the blockchain is required, an oracle pushes information into the network.
Authoriz.	Ownership	As anyone can call a contract method, restrict the execution to the contract owner's address.
	Access Restriction	When function execution checkups are needed, these are handled by generally applicable modifiers.
Lifecycle	Mortal	Since deployed contracts do not expire, self-destruction with a preliminary authorization check is used.
	Automatic Deprecation	When functions shall become deprecated, apply function modifiers to disable their future execution.
Maintenance	Data Segregation	As data and logic usually reside in the same contract, avoid data migration on updates by decoupling.
	Satellite	As contracts are immutable, functions that are likely to change are outsourced into separate contracts.
	Contract Register	When the latest contract version is unknown, participants proactively query a register.
	Contract Relay	When the latest contract version is unknown, participants interact with a proxy contract.
Security	Emergency Stop	Since contracts are executed autonomously, sensitive functions include a halt in the case of bugs.
	Speed Bump	When task execution by a huge number of users is unwanted, prolong completion for counter measures.
	Rate Limit	When a request rush on a task is not desired, regulate how often a task can be executed within a period.
	Mutex	As re-entrancy attacks can manipulate contract state, a mutex hinders external calls from re-entering.
	Balance Limit	There is always a risk that a contract gets compromised, thus limit the maximum amount of funds held.

5. *Smart Contract Patterns*

other patterns. Another example is the Access Restriction pattern, which is directly applied by other patterns, like the Mortal pattern.

A principle shared by several patterns is related to the problem of contract immutability, which is circumvented by using updatable object references. All Maintenance Patterns use this principle to decouple contract functionality, data, or even whole contracts through a proxy object.

As for the security patterns, the main problem that these patterns solve is the lack of execution control after a contract has been deployed, resulting from the distributed execution environment provided by Ethereum. This one-of-a-kind feature of Ethereum allows programs on the blockchain to be executed autonomously, but also has drawbacks. These drawbacks come in various forms, either as harmful callbacks, adverse circumstances on how and when functions are executed, or uncontrollably high financial risks at stake. By applying the security patterns presented, developers can address these security issues and mitigate typical attack scenarios.

As for the generalizability of the patterns, it might be assumed that other platforms face similar issues as Ethereum. In the real world, once a contract is changed, it needs to be revalidated by all involved parties. This concept is also encountered in Ethereum, where contracts are immutable and any change requires the creation of a new contract. Although real world contracts are conclusive and final through their written terms, their code implementations underlie inherent software concepts involving evolutionary code changes and bug fixes. This creates a divergence between contract immutability (a final version of a written agreement manifested in code) and the ability to modify that code (due to bugs or a necessary code updates). That is, the separation of code changes that modify contract terms and those that are necessary due to evolutionary adaptations is important. Altogether, because any software based smart contract ecosystem and its contained contracts require code updates, it can be assumed that maintenance patterns are generally applicable to other ecosystems as well.

5.6. Conclusion

In this chapter we derived Solidity design patterns from white and gray literature using a MLR and qualitative research methods borrowed from Grounded Theory. While many smart contracts have been written in Solidity for different purposes, we have identified, grouped, and described several globally applicable patterns and have discussed common principles and relationships among them. Each pattern is explained in a problem and solution based approach with benefits and drawbacks, to better illustrate the context and applicability of the pattern.

Looking at the patterns, it can be said that the blockchain's unique selling point of being an autonomous execution platform also becomes its problem, as many of the patterns describe ways to get around the immutability and lack of execution control of smart contracts. This reveals the

discrepancy that arises when an inherently autonomous execution environment is confronted with the need for maintainability and upgradeability.

For future work, the presented design patterns can be used to extract code building blocks, which could be integrated in automatic code generating frameworks. Further, the patterns could be incorporated into a certified set of libraries, covering typical and commonly occurring coding scenarios. Beyond that, the collated patterns could be compared to coding practices that evolve in other smart contract platforms. This could further reveal more abstract design patterns that are independent from the underlying implementation framework and are valid for smart contracts in general.

6. Domain Specific Language for Smart Contract Development

The notion to digitally articulate, execute, and enforce agreements with smart contracts has become a feasible reality today. Smart contracts have the potential to vastly improve the efficiency and security of traditional contracts through their self-executing autonomy. To realize smart contracts several blockchain-based ecosystems exist. Today a prominent representative is Ethereum. Its programming language Solidity is used to capture and express contractual clauses in the form of code. However, due to the conceptual discrepancy between contractual clauses and corresponding code, it is hard for domain stakeholders to easily understand contracts, and for developers to write code efficiently without errors. In this chapter we address this issues by the design and study of a domain-specific smart contract language based on a higher level of abstraction that can be automatically transformed to an implementation. In particular, we propose a clause grammar close to natural language, helpful coding abstractions, and the automatic integration of commonly occurring design patterns during code generation. Through these measures, our approach can reduce the design complexity leading to an increased comprehensibility and reduced error susceptibility. Several implementations of exemplary smart contract scenarios, mostly taken from the Solidity documentation, are used to demonstrate the applicability of our approach.

6.1. Introduction

A contract is a “promise or a set of promises, for the breach of which the law gives a remedy, or the performance of which the law in some way recognizes as a duty” [133]. Contracts are common in almost every facet of the business world. Like in many other areas, the trend towards digitization has also taken hold in this field and led to the concept of smart contracts [134], [135]. Smart contracts are a means to digitally facilitate, verify, and enforce the negotiation or execution of contracts and “represent a new era of contracting” [136]. This evolution is grounded on several technological advancements and transformations. Blockchain technology, with its underlying consensus mechanism (implemented through different protocols), allows various parties to reach agreements without requiring any trusted participants among them. This feature paved the way for a decentralized exchange of digital assets (cryptocurrencies), and the subsequent inclusion of

6. *Domain Specific Language for Smart Contract Development*

general scripting languages enabled the evolution towards distributed computing platforms. Both features, the build-in exchange of digital assets (as a means of payment) and the dispersed code execution (supporting distributed applications), are the prerequisites for an ecosystem that makes the notion of smart contracts feasible. Today's predominant ecosystem in this regard is Ethereum [24], a blockchain based distributed computing platform, that allows to formulate smart contracts in the platform's leading programming language Solidity.

The formalization of contracts in a machine-readable and executable form is a challenging task. Mapping the broad articulation space of contracts written in natural language to a conclusive and unambiguous digital representation requires a formalization approach to deduce a proper digital manifestation. In the context of Ethereum, this means translating contract statements from natural (legalese) language into equivalent Solidity code. There is therefore a high likelihood of translation loss. To make matters worse, the blockchain runtime environment and missing low-level abstractions complicate writing correct and secure smart contracts for Ethereum and other blockchain technologies even further.

Our work investigates how productivity can be increased in smart contract development and how to address the aforementioned issues with a Domain Specific Language (DSL) for smart contract formulation called Contract Modeling Language (CML). A DSL is a programming language of limited expressiveness focused on a particular domain [137]. When used properly DSLs can improve productivity by simplifying complex code, promoting communication between domain stakeholders, and eliminating development bottlenecks.

The objective of CML is to investigate how unstructured legal contracts can be uniformly modeled and specified (covering a variety of common contract situations) in order to improve their interpretation and the automatic generation of smart contract implementations. In particular we focus on a declarative and imperative formalization, since we are interested in the conceptual representation of contracts in a programming language. In this context our work seeks to address the following research questions: How and in how far is it possible to bring the abstraction level of smart contracts closer to the contract domain? Can higher abstraction levels in combination with code generation (considering platform-specific programming idioms) reduce the risk of smart contract errors?

The chapter is organized in the following way: First, we provide a short background on contractual stages and contract building blocks in Section 6.2 and discuss our research methodology in Section 6.3. Then, we present our domain specific language in Section 6.4, before we illustrate its practicality in Section 6.5, and evaluate and discuss our findings in Sections 6.6 and 6.7 respectively. Finally, we compare to related work in Section 6.8 and then draw conclusions in Section 6.9.

6.2. Background

6.2.1. Contract Stages

A contract is usually preceded by an abstract agreement that specifies elementary modalities and actions between parties. This informal agreement is then transferred into a written and more rigid contract that is enforceable by law. To avoid the ambiguities of natural language and to explicate terms and conditions as clearly and completely as possible, contracts are written in legalese. Legalese is characterized by a common and well established legal phrasing style that is used to formulate the specifics of a contract. A machine readable representation, usually in a formal language, is retrieved from a conversion of the traditionally written contract, although a contract could be immediately specified in a formal language. Above all, both representations should ideally be equivalent. Basis for the execution of a written contract is grounded on law, where enforceability is considered to be *ex post*, i.e. a party can enforce a settlement at court only after a contractual breach. This stands in contrast to the formal machine representation and its realization as a smart contract, where the execution is based on an architecture, that by design does not allow non-conformism, hence enforceability is considered to be *ex ante*. The above described principals are illustrated in Figure 6.1 which gives an overview of the different contractual stages, namely contract negotiation, contract formation, and contract performance.

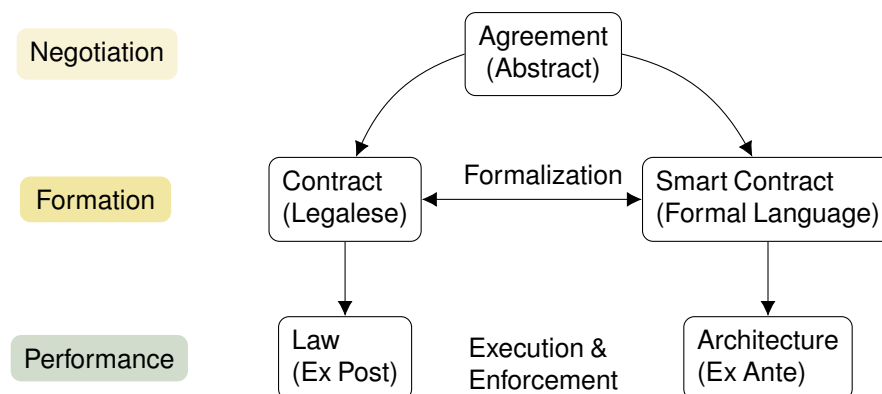


Figure 6.1.: Contract stages and the conceptual relation between traditional and smart contracts.

6.2.2. Contract Building Blocks

Although contracts cover a wide range of subject areas, most will share many common features, as exemplified by the sale of goods contract excerpt in Figure 6.2. Regarding their form, contracts usually build on structuring techniques such as sectional delimitation and paragraph division. Typically contracts are split into articles, sections, subsections and paragraphs which are numbered

6. Domain Specific Language for Smart Contract Development

to support referencing and grouping of particular provisions. This organization can be interpreted as macro and micro structure that eases the legibility and comprehensibility of contracts.

Regardless of the subject matter and contract type, contracts share basic building blocks that serve the same function in all contracts [138]. First, there are (A) definitions which isolate and specify important key terms and concepts that are usually repeated in the agreement. Definitions make the contract more consistent and easier to read as unnecessary repetitions are avoided (e.g., §1.). Second, there are (B) covenants, which are promises made by a party to undertake or refrain from certain actions in the future (e.g., §3. to §5. and §7.). These are the most important provisions of any contract and constitute the obligations of the involved parties to certain performances. Covenants are typically reciprocal, e.g., one party is obligated to pay, while the other is obligated to perform. Covenants are often scattered throughout the contract and are organized by subject matter (e.g., payment obligations, performance obligations). A party's failure to satisfy covenants typically entitles the other party to certain remedies. Third, there are (C) representations and warranties, which are statements of fact made by the parties to each other as of a particular point in time (e.g., §9., §10.). These statements assert the truth about assumptions that are important for the decision to enter the contract and are implicitly coupled with indemnification obligations when untruthful. Fourth, there are (D) conditions, which specify certain requirements that must exist so that a party is obligated to perform under the contract (e.g., §8.). These conditions can be classified by type into conditions precedent and conditions subsequent. Conditions precedent specify the events that must occur to start one's duties to perform under the contract, likewise conditions subsequent specify the events that must occur to end one's duties to perform under the contract. Table 6.1 gives an overview of essential building blocks regarded important for contract construction.

Table 6.1.: An overview of basic contractual building blocks.

	Covenant	Representation	Warranty	Condition
is a	promise	statement	statement or promise	statement
of	action or inaction	fact	fact or condition	condition
applies to	future	past or present	present and future	future
purpose	define activities that will (not) be carried out	make assurances to induce parties to enter contract	assure that facts and conditions are/will be true	define conditions affecting the party's contractual duty

CONTRACT FOR THE SALE OF GOODS

Paragraph 1. [], hereinafter referred to as Seller, and [], hereinafter referred to as Buyer, hereby agree on this [] day of [], in the year [], to the following terms.

A. Identities of the Parties

Paragraph 2. Seller, whose business address is [], in the city of [], state of [], is in the business of []. Buyer, whose business address is [], in the city of [], state of [], is in the business of [].

B. Description of the Goods

Paragraph 3. Seller agrees to transfer and deliver to Buyer, on or before 2019-09-01, the following goods: 1 x production line machinery at the price of \$7,500.

C. Buyer's Rights and Obligations

Paragraph 4. Buyer agrees to accept the goods and pay for them according to the terms further set out below.

Paragraph 5. Buyer agrees to pay for the goods half upon receipt, with the remainder due within 30 days of delivery. If Buyer fails to pay second half within 30 days, an additional fine of 10% has to be paid within 14 days.

Paragraph 6. Goods are deemed received by Buyer upon delivery to Buyer's address as set forth above.

Paragraph 7. Buyer has the right to examine the goods upon receipt and has 14 days in which to notify seller of any claim for damages based on the condition, grade, or quality of the goods.

Paragraph 8. The Buyer's obligation to complete the purchase of the goods is subject to the Buyer obtaining a financing commitment of at least \$5,000.

D. Seller's Obligations

Paragraph 9. Until received by Buyer, all risk of loss to the above-described goods is borne by Seller.

Paragraph 10. Seller warrants that the goods are free from any and all security interests, liens, and encumbrances.

Figure 6.2.: An exemplary contract for the sale of goods.

6.3. Research Study Design

Our approach to design a DSL is guided by the design science methodology where “knowledge and understanding of a problem domain and its solution are achieved in the building and application of the designed artifact” [7]. In particular, we employed an approach described by Wieringa [139], where the design process iterates multiple times over two activities: first designing an artifact that improves something for stakeholders (design cycle) and subsequently empirically investigating the performance of that artifact in its context (empirical cycle). Our focus was on the design cycles, where an improvement problem is investigated, alternative treatment designs are generated and validated, a design is selected and implemented, and experience with the implementation is evaluated. In our context, this meant to investigate smart contract implementation issues, to come up with possible abstract language constructs, implement these constructs in a DSL development framework, and subsequently evaluate and assess the suitability of the implementation. For the empirical cycles, we performed an analysis of multiple scenario cases to evaluate the improvements in the design cycles (Section 6.6).

6.4. Contract Modeling Language (CML)

CML is a high-level DSL using object-oriented abstractions for implementing smart contracts. It is designed with several intentions in mind. First, it should allow for the specification of common relevant contractual elements. Second, it should be easy to read and understand through a clause grammar close to natural language that resembles real-world contracts. Third, it should improve productivity and simplify complex code. Fourth, the defined contract logic should serve as basis for code generation, backed possibly by a variety of distributed ledger technologies. Regarding the last intention, for proof-of-concept, we focus on the generation of Solidity code, being the predominant language for smart contracts today.

The CML language is developed in Xtext [140], a framework for the development of programming languages and DSLs. For more information on Xtext and the full grammar definition of CML we refer to Appendix A.1. To promote reproducibility of our research, the CML language implementation source code is available on GitHub [141]. Further a CML web editor [142] for demonstration purposes exists.

6.4.1. Language Characteristics

The basic structure of a CML contract is similar to a class in object orientation. It consists of state variables and functions (actions), which read and modify these. In addition, a contract contains clauses, which mimic and capture covenants in a standardized way, close to natural language syntax.

6.4. Contract Modeling Language (CML)

These indicate the context under which the actions are to be called, meaning they combine different aspects that influence action execution. In its most simplistic form a clause specifies the obligation or permission of a party to execute a specific action.

CML is a white-space aware language (like Python), thus indentation is used to structure code blocks instead of braces. Each CML file starts with a namespace declaration to identify it uniquely. Then, a number of namespace imports can be declared. In order to avoid hardcoding basic classes or types directly in the grammar, CML follows a library approach. This approach allows the grammar to focus only on the syntax, and the language to be easily extendable through modification of library code. A default namespace is imported into every CML file by default: `mainlib.cml`. This library defines the basic concepts of the language, similar to the standard libraries in other popular programming languages like C and Java. More precisely, the library specifies essential data types and associated functions that facilitate the implementation of contracts. This approach allows to promote a simplification of the development process.

6.4.2. Type System

CML is statically-typed, i.e. the types of functions, function parameters, local variables, etc. need to be explicitly declared, with the sole exception that the return type of a function can be omitted if it is not needed. Thus each function possesses a well-defined type signature (input and output). This allows type-checking to be performed, meaning that operations on declared typed elements are checked to be consistent with expected types. Built-in data types in CML range from basic types to more complex types to cover common concepts in the field of smart contract design. The simplest of types are primitive types which describe the various kinds of atomic values allowed in CML. These include *Boolean*, *String*, *Integer*, *Real*, *DateTime*, and *Duration*. The last two types represent the basic temporal concepts of absolute and relative time, needed to express temporal constraints and relationships typically encountered in contracts. Absolute time is a definite time value (also called a time point) e.g. 2019-01-01 14:30:00 UTC, whereas the concept of relative time is used to model time duration that is independent from any time point, e.g. 2 hours, 7 days. Regarding temporal constraints and their verification in the context of contracts we refer interested readers to [143]. Beyond the aforementioned primitives, CML includes predefined and easily extensible structural composite types that are derived from literature on smart contract ontologies [144], [145], to embody common contract-specific concepts. These include *Party*, *Asset*, *Transaction*, and *Event*. *Party* denotes an individual or organization with an unique identifier that participates in a contract. *Asset* describes a resource (long-lived identifiable item) with a certain economic value. *Transaction* is used to describe a message that is submitted by a party along contract interaction. *Event* characterizes anything that happens, being either important or unusual. In addition, a few special variables (*caller*, *anyone*, *now*, *contractStart*, *contractEnd*) are defined which are always

present and often needed during contract definition.

6.4.3. Clause Structure

CML introduces clauses as syntactic elements based primarily on the covenants discussed in Section 6.2 and further inspired by the study of existing approaches found in [146]–[151]. Covenants are most relevant for smart contracts, since they enclose the expected actions to be performed. In view of the dynamics of the natural language, in which they are represented, their composition cannot be precisely defined. However, there are structural components that can be singled out. Most clauses consist of (at least) three parts: an actor, an action, and a modality for that action. A very basic covenant reflecting these components is: “The buyer must pay.” Moreover, other commonly occurring components can be extracted, which specify the context of a covenant more precisely, such as trigger events, conditions, and involved objects. Trigger events stipulate under which circumstances a clause must be taken into account and refer to either internal or external events. Internal events can be controlled by the contract parties (e.g., satisfied action, fulfilled clause), whereas external events cannot be controlled by the parties themselves (e.g., price feed). Clause conditions define time and state restrictions that must be subsequently met after a trigger event. Involved objects specify the people, places, things receiving an action or having an action done to them. An extended version of the previous example clause would look like the following: “Upon receipt of the product the buyer must pay within 14 days.”. Figure 6.3 shows a representation of fundamental covenant clause components discussed in this paragraph.

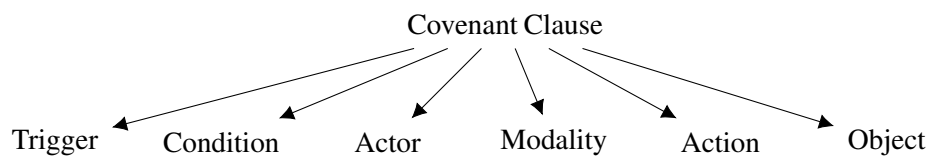


Figure 6.3.: A conceptual breakdown of covenant clause components.

Based on the above insights we propose a clause syntax, illustrated in Figure 6.4, for the transformation of covenants. Each clause has a unique identifier for referencing and must contain at least an actor, an action, and the deontic modality of this action (i.e., “may” or “must”). Optional elements include temporal or general constraints. Temporal constraints are indicated by the keyword “due” followed by a temporal precedence statement (i.e., “after” or “before”) and a trigger expression. The trigger expression refers to an absolute time or a construct from which an absolute time can be deduced. This includes the performance of a clause, the execution of an action, or the occurrence of an external event. Additionally, the “due” statement can be enriched by a duration statement (“within”) to further specify the considered time-frame, as well as a repeat statement (“every”) to model the re-

curing nature of a covenant. General constraints can be defined after the keyword “given” by multiple linked conditions that evaluate to true or false. These conditions usually refer to the contract state, conditions regarding the transaction input are handled within the functions. It is worth noting that the deontic “must” requires the specification of a terminating temporal constraint (declared by “within”) to evaluate the fulfillment of a covenant, since without it, a covenant can always be met in the future.

```

clause ID
  [due [within RT] [every RT from AT to AT] (after|before) TRIGGER]
  [given CONDITION]
  party ACTOR
  (may|must) ACTION {(and|or|xor) ACTION}

```

```

Trigger:      AT | ClauseTrigger | EventTrigger | ActionTrigger
ClauseTrigger: clause ID (fulfilled|failed)
ActionTrigger: ACTOR did ACTION
EventTrigger: event ID

```

RT...Relative Time, AT...Absolute Time

Figure 6.4.: The structure of a CML clause declaration.

6.4.4. CML by Example: Simple Open Auction

The application of clause constructs, predefined types, and type operations is shown by example. Listing 6.1 contains a CML contract specification for a simple auction in which anyone can bid during a bidding period. If the highest bid is raised, the previously highest bidder gets her bid back. After the end of the bidding period, the beneficiary can withdraw the highest bid.

Contract Modeling Language (CML)

```

namespace cml.examples

import cml.generator.annotation.solidity.*

@PullPayment
contract SimpleAuction
  Integer highestBid
  Party currentLeader
  Party beneficiary
  Duration biddingTime

  clause Bid
    due within biddingTime after contractStart

```

6. Domain Specific Language for Smart Contract Development

```
party anyone
may bid

clause AuctionEnd
  due after contractStart.addDuration(biddingTime)
  party beneficiary
  may endAuction

action init(Duration _biddingTime, Party _beneficiary)
  biddingTime = _biddingTime
  beneficiary = _beneficiary

action bid(TokenTransaction t)
  ensure(t.amount > highestBid, "There already is a higher bid.")
  caller.deposit(t.amount)
  if (highestBid != 0)
    transfer(currentLeader, highestBid)
  currentLeader = caller
  highestBid = t.amount

action endAuction()
  transfer(beneficiary, highestBid)
```

Listing 6.1: A CML contract for a simple open auction.

6.5. Solidity Code Generation

The concrete syntax (grammar) of CML is defined in Xtext, which generates the language infrastructure and derives a corresponding meta-model. Once a CML text input file is processed, the parser creates an in-memory instance of that meta-model, called Abstract Syntax Tree (AST). This representation is then traversed by the generator, which is written in Xtend [152], to produce Solidity code that further relies on static and dynamically created support libraries. These libraries either contain the implementation of declared CML type operations, or relate to libraries for secure smart contract development. Figure 6.5 illustrates this process.

6.5.1. CML to Solidity Mapping

Having specified essential domain-specific constructs in CML through predefined types, we propose a mapping that allows an automated generation of Solidity contracts. The conceptual equivalent of CML domain model definitions (*Party*, *Asset*, *Transaction*) is Solidity's struct, in

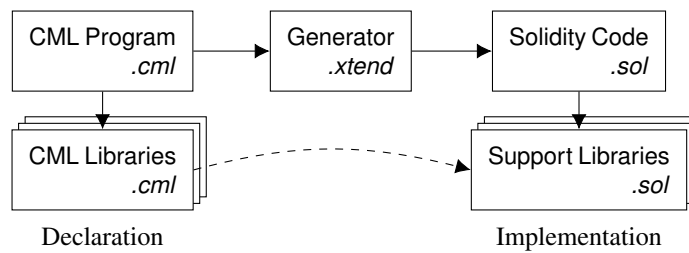


Figure 6.5.: The CML code generation process.

which the entire hierarchy of a given type is assembled. CML functions and enumerations are mapped to their respective conceptual equivalents in Solidity. Events that are contained in CML reflect external events and are mapped to functions in Solidity, since outside information can only enter the Solidity platform through interaction (passing data to a method). Regarding the constraints of clause constructs, these are reflected in a single function modifier that is added to clause functions and contains declarative checks that preempt improper execution. Table 6.2 summarizes the proposed CML-Solidity mapping.

To illustrate the mapping process, we are transferring the simple auction CML contract from Listing 6.1 to Solidity code presented in Listing 6.2. Beyond the generation of functional contract code, supportive code is generated to evaluate the execution context, which is required to apply any restrictions (e.g. time, caller, state) defined in the clause statements.

Table 6.2.: The mapping of CML constructs to Solidity constructs.

CML Construct	Solidity Construct
Party	Struct
Asset	Struct
Transaction	Struct
Enumeration	Enumeration
Event	Function
Function	Function
Top level function	Function with pure/view declaration
Clause constraints	Function modifier with conditional checks

Solidity

```

pragma solidity >=0.4.22 <0.7.0;
pragma experimental ABIEncoderV2;
  
```

6. Domain Specific Language for Smart Contract Development

```
import "./lib/cml/ConditionalContract.sol";
import "./lib/cml/DateTime.sol";
...
contract SimpleAuction is ConditionalContract, PullPayment {

    struct Party {
        address payable id;
    }

    uint highestBid;
    Party highestBidder;
    Party beneficiary;
    uint biddingTime;
    uint _contractStart;

    constructor(uint _biddingTime, Party memory _beneficiary) public {
        biddingTime = _biddingTime;
        beneficiary = _beneficiary;
        _contractStart = now;
    }

    function bid() public payable checkAllowed("Bid") {
        require(msg.value > highestBid, "There already is a higher bid.");
        if (highestBid != 0) {
            _asyncTransfer(highestBidder.id, highestBid);
        }
        highestBidder = Party(msg.sender);
        highestBid = msg.value;
    }

    function endAuction() public checkAllowed("AuctionEnd") {
        _asyncTransfer(beneficiary.id, highestBid);
    }

    function clauseAllowed(bytes32 _clauseId) internal returns (bool) {
        if (_clauseId == "Bid") {
            | require(onlyAfter(_contractStart
                , biddingTime, true), "Function not called within expected timeframe");
            return true;
        }
        if (_clauseId == "AuctionEnd") {
            | require(onlyBy(beneficiary.id), "Caller not authorized");
            | require(onlyAfter(DateTime.addDuration

```

Support Libraries

CML Domain Model Transformation

State Variables

Constraint Check

Derived Clause Constraints

```

        (_contractStart, biddingTime), 0, false), "Function called too early"); ↵
    return true;
}
return false;
}

function clauseFulfilledTime(bytes32 _clauseId) internal returns (uint) {
    uint max = 0;
    if (_clauseId == "Bid" && (callSuccess(this.bid.selector))) {
        if (max < callTime(this.bid.selector)) {
            max = callTime(this.bid.selector);
        }
        return max;
    }
    if (_clauseId == "AuctionEnd" && (callSuccess(this.endAuction.selector))) {
        if (max < callTime(this.endAuction.selector)) {
            max = callTime(this.endAuction.selector);
        }
        return max;
    }
    return max;
}
...
}

```

Derived
Clause
Constraints

Listing 6.2: The generated Solidity contract from CML definition in Listing 6.1.

6.5.2. Code Generation Idioms

Design Patterns

Design patterns are a commonly used technique to encode design guidelines or best practices (for more information on this topic, see [153], [154]). In Chapter 5 we have gathered design patterns for smart contracts in the Ethereum ecosystem along with corresponding code building blocks for Solidity, which can be directly integrated in the automatic code generation process. This procedure is exemplified with the “Ownership” and “PullPayment” pattern. The “Ownership” pattern satisfies a contract has an owner (by default the creator of a contract) and is used to limit access to sensitive functions to only that owner. The “PullPayment” pattern is used to mitigate security risks when sending funds by switching from a push to a pull payment, meaning that funds must be proactively withdrawn by the recipient. Listing 6.3 illustrates the application of patterns in CML and Listing 6.4 shows the generated code output.

6. Domain Specific Language for Smart Contract Development

```
Contract Modeling Language (CML)

namespace cml.examples

import cml.generator.annotation.solidity.*

@Ownership @PullPayment
contract BecomeRichest
    Party richest
    Integer mostSent

    clause BecomeRichest
        party anyone
        may becomeRichest

    action Boolean becomeRichest(TokenTransaction t)
        caller.deposit(t.amount)
        if(t.amount > mostSent)
            transfer(richest, token.quantity)
            richest = caller
            mostSent = t.amount
            return true
        return false
```

Listing 6.3: A CML contract with design pattern annotation for the “Ownership” and “Pullpayment” pattern.

```
Solidity

pragma solidity >=0.4.22 <0.7.0;
...
import "./lib/openzeppelin/Ownable.sol";
import "./lib/openzeppelin/PullPayment.sol";
...

contract BecomeRichest is ConditionalContract, Ownable, PullPayment {
...
    function becomeRichest() public payable
        checkAllowed("BecomeRichest")
        returns (bool)
    {
        if (msg.value > mostSent)
```

```

{
  _asyncTransfer(richest.id , address(this).balance);
  richest = Party(msg.sender);
  mostSent = msg.value;
  return (true);
}
return (false);
...
}

```

←
Asynchronous
Payment

Listing 6.4: An excerpt of the generated Solidity contract from Listing 6.3 utilizing design patterns.

Avoiding Overflows/Underflows

Signed and unsigned integers in Solidity are restricted in size to a range of values. For example, an unsigned 8-bit integer (uint8) may incarnate values between 0 and $255 - (2^8 - 1)$. If the result of an operation is outside of this supported range an overflow or underflow occurs and the result is truncated. To illustrate this behavior, when using 8-bit unsigned integers, $255 + 1 = 0$. This result is more apparent in binary representation, where $1111\ 1111_2 + 0000\ 0001_2$ should result in $1\ 0000\ 0000_2$. However, since only 8 bits are available, the leftmost bit is lost, resulting in a value of $0000\ 0000_2$. These overflows can have serious consequences that one should mitigate against. One approach is to use `require` to limit the size of inputs to a reasonable range, or use a library for secure smart contract development like OpenZeppelin’s [129] “SafeMath”, to cause a revert for all overflows. The annotation `@SafeMath` on top of a CML contract adheres to the latter approach and automatically replaces all occurrences of arithmetic operations with equivalent “SafeMath” library calls, as shown in Listings 6.5 and 6.6.

Contract Modeling Language (CML)

```

namespace cml.examples

import cml.generator.annotation.solidity.*

@SafeMath
contract Counter
  Integer counter = 0

  clause ChangeCounter
    party anyone

```

6. Domain Specific Language for Smart Contract Development

```
    may increaseCounter or decreaseCounter

    action increaseCounter()
        counter = counter + 1

    action decreaseCounter()
        counter = counter - 1
```

Listing 6.5: A CML contract with “SafeMath” annotation to indicate that arithmetic operations should be checked for overflows and underflows.

```
Solidity
```

```
pragma solidity >=0.4.22 <0.7.0;
...
import "./lib/openzeppelin/SafeMath.sol";
...
contract Counter is ConditionalContract {
...
    uint counter = 0;
...
    function decreaseCounter() public
        checkAllowed("ChangeCounter")
    {
        counter = SafeMath.sub(counter, 1);
    }
...
}
```

↑
Safe Arithmetic

Listing 6.6: An excerpt of the generated Solidity contract from Listing 6.5, containing wrapper calls for safe arithmetic operations.

Fixed Point Arithmetic

Solidity supports integer numbers, but decimal numbers are not yet supported. Although it is possible to declare fixed point number types, they cannot be assigned to or from. When dealing with decimals on systems that support only integers, fixed point arithmetic can be used. This is a technique for performing operations on numbers with fractional parts using integers. The approach builds on scaling an integer so that a certain (fixed) number of decimals are included, e.g. the value 1.23 can be represented as 123 with a scaling factor of 1/100. In other words, the decimal values are “nor-

malized” to integer values. Arithmetic operations are then executed on the underlying integers with the overhead of taking the scaling factors into account. The approach is demonstrated in Listings 6.7 and 6.8. It should be noted that during the interaction with the Solidity contract, the input and output number values are of fixed-point type and require conversion in respect to the chosen scaling value.

```

Contract Modeling Language (CML)

namespace cml.examples

import cml.generator.annotation.solidity.*

def Integer equation()
    return 8 / 2 * (2 + 2)

@FixedPointArithmetic(decimals=2)
contract FixedPointArithmetic

    clause Clause
        party anyone
        may calc1 or calc2

    action Integer calc1()
        return equation() / 2

    action Real calc2()
        return equation().toReal() * 2.5

```

Listing 6.7: A CML contract containing arithmetic operations and “FixedPointArithmetic” annotation.

```

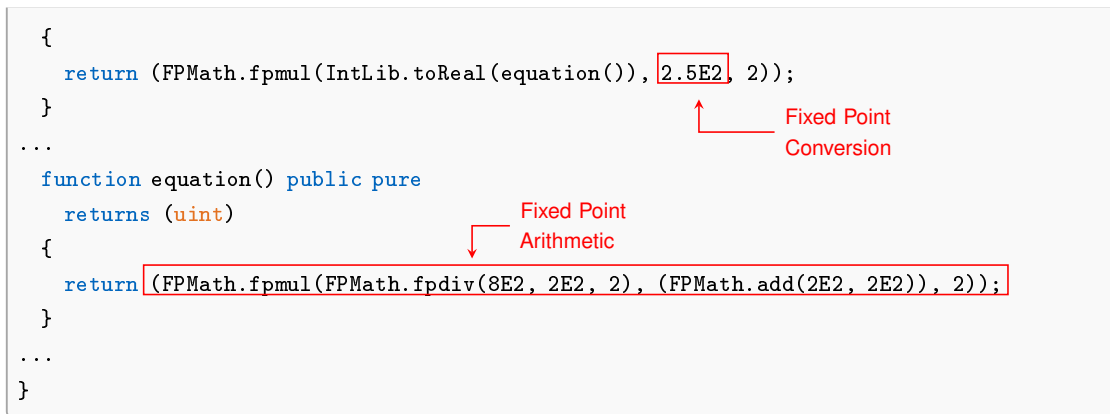
Solidity

pragma solidity >=0.4.22 <0.7.0;
...
import "./lib/cml/FPMath.sol";
...
contract FixedPointArithmetic is ConditionalContract {
...
    function calc2() public
        checkAllowed("Clause")
        returns (uint)

```

6. Domain Specific Language for Smart Contract Development

```
{
  return (FPMath.fpmul(IntLib.toReal(equation()), 2.5E2, 2));
}
...
function equation() public pure
  returns (uint)
{
  return (FPMath.fpmul(FPMath.fpddiv(8E2, 2E2, 2), (FPMath.add(2E2, 2E2)), 2));
}
...
}
```



Listing 6.8: An excerpt of the generated Solidity contract from Listing 6.7, with applied fixed point conversion and fixed point arithmetic wrapper calls.

Type Collections

Solidity supports the concept of arrays and mappings (dictionaries). Mappings can be seen as hash tables which are virtually initialized such that every possible key exists and is mapped to a value whose byte-representation is all zeros: a type's default value [155]. This has the drawback that mappings cannot be directly iterated over since there is no way to know how many keys exist, because they all exist. A common pattern is therefore to use an auxiliary array in combination with mappings to hold the keys that exist.

Collections of values in CML are denoted with [] after a type declaration and are transformed to a Solidity mapping, where the type identifier is used as key. Library code is generated for each collection containing a mapping and key store to provide basic editing and iteration functionality for the collection. Due to missing generics in Solidity, this code must be dynamically created for each mapping to accommodate for different types. In order to minimize the operational complexity of key lookups, a circular linked list is used instead of an array to track mappings that exist. Key existence is checked by verifying that a key node has a valid pointer to the previous and next node, thus iterating all key entries can be avoided. The usage of a circular linked list has also the advantage to support collection implementation variations, e.g. key ordering, a First In - Last Out (FIFO) stack, or a First In - First Out (FIFO) ring buffer. Listings 6.9 to 6.11 illustrate the described approach.

Contract Modeling Language (CML)

```
namespace cml.examples
```

```

asset Asset identified by inventoryNumber
  Integer inventoryNumber
  Integer aquisitionCost

contract Mapping
  Asset[] assets

  clause AssetInteraction
    party anyone
    may addAsset or removeAsset or countAssets or countValuableAssets

  action addAsset(Asset a)
    if (!assets.contains(a.inventoryNumber))
      assets.add(a)

  action removeAsset(Integer inventoryNumber)
    assets.rmv(inventoryNumber)

  action Integer countAssets()
    return assets.size()

  action Integer countValuableAssets()
    var Integer count = 0
    for (a in assets)
      if (a.aquisitionCost > 100)
        count++
    return count

```

Listing 6.9: A CML contract using a collection.

Solidity

```

pragma solidity >=0.4.22 <0.7.0;
...
import "./lib/cml/Model.sol";
import "./lib/cml/MapUintAsset.sol";
...
contract Mapping is ConditionalContract {
...
  using MapUintAsset for MapUintAsset.Data;
  MapUintAsset.Data internal assets;
...

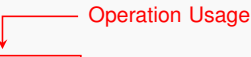
```

Library Usage

Streamlined
Type & Operations

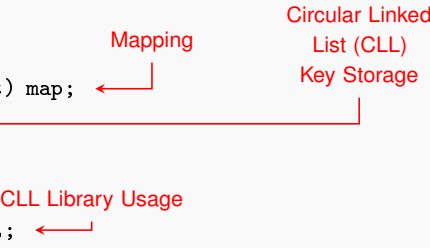
6. Domain Specific Language for Smart Contract Development

```
function addAsset(Model.Asset memory a) public
    checkAllowed("AssetInteraction")
{
    if (!assets.contains(a.inventoryNumber))
    {
        assets.add(a.inventoryNumber, a);
    }
}
...
function countValuableAssets() public
    checkAllowed("AssetInteraction")
    returns (uint)
{
    uint count = 0;
    for (uint i = 0; i < assets.size(); i++)
    {
        Model.Asset storage a = assets.getEntry(i);
        if (a.aquisitionCost > 100)
        {
            count++;
        }
    }
    return (count);
}
```



Listing 6.10: An excerpt of the generated Solidity contract from Listing 6.9.

```
pragma solidity >=0.4.22 <0.7.0;
...
import "./CLLUint.sol";
import "./Model.sol";
...
library MapUintAsset {
...
    struct Data {
        mapping(uint => Model.Asset) map;
        CLLUint.CLL mapIdList;
    }
...
    using CLLUint for CLLUint.CLL;
```



```

...
function size(Data storage self) public view returns (uint) { ← Provided Operation
    return self.mapIdList.sizeOf();
}
...
}

```

Listing 6.11: An excerpt of the generated Solidity collection library code.

6.6. Evaluation

In this section, we provide arguments to support the claim that our proposed language concepts lead, through a reduction of complexity, to an increased comprehensibility, and reduced error susceptibility. For this purpose we compare several contract use case scenarios (partly taken from the Solidity documentation) specified in CML with their respective Solidity implementations generated by our framework.

Solidity is a Turing complete language, which gives expressiveness and power, but can lead to less comprehensible code and a more difficult assessment of correctness. As a general rule, the more expressive a language is, the higher its complexity, and the more it is prone to include bugs and errors. In comparison with Solidity, our DSL contains abstraction much closer to the target domain, with the aim to promote clarity and comprehensibility.

In our context, to measure the complexity, we follow the characterization of complexity as detail complexity, defined by Senge [156] as “the sort of complexity in which there are many variables.” We relate this definition to measuring the logical lines of code and syntactic elements (AST nodes) that are contained in respective representations. The number of AST nodes is determined for Solidity with the help of an ANTLR parser [157] (taking into account all “ASTNode” expressions), whilst for CML it is regarded as the elements contained in the generated AST (parse tree). Please note that this is not intended as a precise and generalizable measurement, but rather to give a rough comparison of our approach compared to contracts encoded directly in Solidity. Looking at the results, which are summarized in Table 6.3, we can see that both metrics for the CML representation are always lower. On average CML performs 840% better in terms of Logical Lines Of Code (LLOC) and 610% better in terms of syntactic elements. Our results indicate that, in our examples, evidence for the higher abstraction level of CML can be found, which leads to lower complexity and in turn should lead to less susceptibility to errors.

6. Domain Specific Language for Smart Contract Development

Table 6.3.: A complexity comparison between CML representation and generated Solidity implementation.

Use Case	CML		Solidity		$\Delta\%$	
	LLOC ¹	SE ²	LLOC	SE	LLOC	SE
Become Richest	18	43	208	502	1056	1067
Purchase	39	116	216	584	454	403
Simple Auction	28	75	402	558	1336	644
Time Lock	27	70	344	475	1174	579
Voting	64	227	485	1655	658	629
AVG	35,2	106,2	331,0	754,8	840,3	610,7

¹Logical Lines of Code ²Syntactic Element: AST Node

6.7. Discussion

There are several challenges regarding the formalization of contracts as discussed by Pace and Schneider [146]. A survey of formal languages for contracts performed by Hvitved [158] gives an overview of possible approaches. With regard to contract formalization, one of the main problems is the succinct, consistent, and sufficient representation of contractual statements. Finding the right abstractions for frequently recurring components in legal contracts that are relevant for the description of smart contracts is crucial. On this basis, a generic description covering a wide range of contract scenarios can be worked out. To our best knowledge no literature exists that deals with the conceptual analysis of relevant components. It rather seems that each work in the contract formalization field has its own assumptions about an optimal description.

Regarding the sufficient representation of contracts, it is debatable whether every statement can be fully grasped as a deterministically calculable expression. Syntactic ambiguity is almost always present in natural language and often unintentional. Moreover, some legal statements are intentionally kept vague and are therefore difficult to express. These statements must then be interpreted into a clear and possibly controversial representation. Apart from this, consideration could be given to improving the drafting and formalization process of covenants through a graphical representation, as suggested by Martínez, Díaz, Cambroner, *et al.* [151], to better reflect the complex relationships of contractual clauses.

Our approach of having clause statements, in addition to the imperative declaration of actions, is based on the idea of providing an abstract level of description that is closer to conventional contract clauses and provides a general overview of contract behavior. This has the advantage of better

isolating execution context requirements for actions, which promotes comprehensibility and leaner actions, as only requirements depending on input parameters need to be checked within respective actions. To further extend the abstraction efforts, language constituents are used that are more closely related to application domain concepts, which helps to make their intended purpose more explicit. As an example, the *Party* and *Duration* type specifiers have a natural language meaning that is also accessible to non-programmers, as opposed to their corresponding representation as `address` and `uint` type in Solidity. CML builds a generic framework for contract formalization without an exuberant syntax leaking implementation details. The code generation process allows to alter abstracted contract specifics to a desired implementation form. Hence, a contract representation that is very compact can translate into a more verbose implementation language, in which the generator helps to construct the needed bulk.

Assuming that experts create the code generator, less experienced users can rely on a correct implementation. No manual coding effort is required, therefore accelerating the development time while decreasing the chance of errors when compared to manual coding from requirements. This is an important aspect, as there are many potential causes of programming errors in Solidity, like integer overflow and underflow, re-entrancy, or timestamp dependence, to just name a few [159]. Further, in case of a required adaption to new best practices, the code generator routines must only be updated in one place and can be reused to generate adapted code.

On the negative side, since abstraction is always connected to information loss, a code generator can hardly cover all cases and it might be required to alter the generated output to inject some custom code. Another problem is that code is generated with the goal of being generally applicable, which entails increased complexity. This might result in generated code that is more elaborate and less comprehensible. In the context of Solidity this may mean that the code efficiency in terms of transaction costs is not on par with use case optimized code. As the financial aspect is an important concern in Solidity, this might hinder the adoption of this approach, but could be out weight by the gained advantages concerning productivity and code quality.

Overall, if applied correctly, abstraction and code generation can increase the efficiency, clarity, and flexibility of code whilst reducing the susceptibility to errors.

6.8. Related Work

Several works pursue the approach to utilize a domain specific language and the concept of abstraction to facilitate the creation process of smart contracts.

Regnath and Steinhorst [149] have derived several language design concepts to approach a unified contract language demonstrated by a prototype implementation called SmaCoNat. The proposed concepts include the reliance on a small set of predefined operations and data types, an enforced

6. Domain Specific Language for Smart Contract Development

sectioned code structure, limited aliasing, and building on natural language identifiers. Hence, approaching a unified contract language that enables a common understanding of code semantics on higher abstraction layers. In comparison to our work, the abstraction is put on a relatively high-level, which can pose limitations on the expressiveness of the language (as being too generic).

Another approach by Frantz and Nowostawski [147] proposes a semi-automated method for the translation of institutional constructs in a human readable behavior specification to Solidity smart contracts. The applied conceptual approach is closely related to our work, in the sense that the institutional constructs describe the parties stipulations in a structured manner, similar to our clause formalization. Another commonality is the generation of Solidity code from an abstract representation, but unlike our work, the generated smart contracts contain only skeleton code and require considerable manual input to make them executable.

Yet another paper by He, Qin, Zhu, *et al.* [148] proposes a specification language for smart contracts called SPESC, which can define the specification of smart contracts for the purpose of collaborative design. The SPESC language contains term constructs that are akin to the clause constructs in our work, but the details of actions cannot be adequately specified and code generation is missing.

In general, the above mentioned publications contain interesting approaches and findings, but the proposed domain specific languages lack a model transformation to an executable smart contract implementation, which is demonstrated in this work. To our best knowledge, apart from our and the already mentioned work by Frantz and Nowostawski [147], there is only one further work by Mavridou and Laszka [123] that deals with the generation of Solidity code from an abstract representation. Regarding the clause formalization itself, our approach can be compared with works by Prisacariu and Schneider [150] and Martínez, Díaz, Cambroner, *et al.* [151] in which contract specification is based on the deontic notions of obligation, permission, and prohibition applied to actions. This approach is often used in the formalization of contracts.

In order to also point out efforts from the legal tech industry in regards to contract specific DSLs, the Accord Project [160] is to be mentioned. It is an open source, non-profit initiative developing specifications and open-source software tools for future smart legal contracting. The project aims to provide an open, standardized format for smart legal contracts that binds legally enforceable text in natural language to executable business logic. The proposed toolchain contains Ergo [161], a DSL with which the execution logic of legal contracts can be specified. The language features programming constructs specifically designed for legal contracts, thus it is also comparable to CML.

6.9. Conclusion

In this chapter we have analyzed important contract building blocks and proposed a high level smart contract language called CML. CML incorporates a fluently readable, clause like formalization

concept to describe the individual operational intents (commitments) of contract participants. This approach enables a representation that is conceptually and syntactically easier to grasp and thus also improves reasoning about a contract. Another key point is that CML describes contract semantics on an higher level and transfers the specifics of an implementation to lower levels. Consequently, the specification of a contract with its underlying model and defined behavior can be decoupled from the actual implementation. This aspect is demonstrated by transforming contracts from CML to Solidity code. It is possible to automate platform specific implementation steps, for example the inclusion of design patterns or coding abstractions. Thus, contract creators can be shielded from low level implementation specific tasks.

For future work, we plan to evaluate the efficiency of our approach in an experiment. Further, we plan to enhance the Solidity code generation process with other commonly occurring design patterns, coding abstractions, and more powerful code inference mechanisms. Beyond that, code generation support for another smart contract platform can be incorporated. This could provide further insight into the general applicability of the proposed smart contract abstraction mechanisms and lead to future improvements and extensions.

7. Blockchain DevOps

With the evolution and proliferation of blockchain, the technology is becoming more prevalent in enterprise software development. Using the already proven DevOps approach in this setting makes sense, as it can accelerate the general pace of software development and delivery, improve software quality, and increase overall productivity. However, there is currently a lack of guidance on a structured DevOps approach and a breakdown of the specifics in the context of blockchain-based software development. Therefore, we combine gray literature and DevOps application studies from pertinent GitHub projects to systematically investigate current practices and solution approaches for an efficient blockchain-oriented DevOps procedure in this chapter. In this process, we elaborated procedural steps and related activities according to the main stages of Continuous Integration (CI) and Continuous Delivery/Deployment (CD). Our research shows that core DevOps concepts and activities are similar to other areas and are entirely possible with already established CI/CD solutions that orchestrate the right tools, with the difference that more rigorous testing and differentiated deployment practices are required due to the inherent immutability of blockchain.

7.1. Introduction

DevOps and blockchains are two hype terms of the recent past. DevOps is a multi-layered concept that is not easy to grasp and can be defined in many ways [162]. In its broadest sense, DevOps refers to the combination of software development (Dev) and operations (Ops) with a focus on cross-organizational integration to bridge the gap between different stages of the software life cycle. Two core aspects of DevOps are Continuous Integration (CI) and Continuous Delivery/Deployment (CD), which support the DevOps principle of interlocking the two underlying disciplines through a high degree of automation. CI usually refers to integrating, building, and testing code, whilst CD is primarily about automating the deployment and release engineering process.

Blockchains, on the other hand, combine various computational and economic concepts to provide a fraud-free intermediary platform for efficiently settling transactions between different parties. In this context, shared business processes can be realized through application code running autonomously on the blockchain to digitally facilitate, verify, and enforce the execution of arbitrary terms via smart contracts. As a special feature, smart contracts are usually not subject to a normal

7. Blockchain DevOps

software life cycle, in which a new code version may add features or fix bugs. This circumstance means that software quality and reliability are important pillars in development, with frequent and varied tests attempting to ensure high requirements in these areas. In this and various other respects, DevOps can provide valuable support, be it through test automation or the provision of stable operating environments. However, at the moment there is a lack of a structured approach and breakdown of the specifics regarding DevOps usage in this area. To address this gap, we explore DevOps approaches and methods by gathering data from multiple sources and applying GT techniques to extract and identify common practices.

In order to concretize the research objectives, we ask the following research questions: What are typical stages and activities in a DevOps approach for blockchain-based applications? What are the particularities of using DevOps in blockchain-based software development?

For illustration purposes, this chapter describes DevOps in the context of Ethereum, a popular smart contract platform, and Solidity, the platform's leading programming language for smart contracts. However, it can be assumed that the presented concepts and basic practices are in principle transferable to other platforms as well.

This chapter is structured as follows: First, we discuss related work in Section 7.2 and our research methodology in Section 7.3. Then, we elaborate DevOps for blockchain-based solutions as main contribution in Section 7.4. Finally, we discuss findings in Section 7.5 and conclusions in Section 7.6.

7.2. Related Work

According to our research, there is currently no academic literature that specifically addresses DevOps in the context of blockchain-based software development. There are some works that deal with (different types of) testing such software that can be considered as extended literature (see [163] and referenced literature therein). That aside, here are some papers that at least decidedly mention DevOps in the context of blockchains and smart contracts. Koul [164] discusses challenges faced in testing blockchain-based applications. The paper describes different approaches to testing and acknowledges the need to devise specialized tools and techniques for this purpose to ensure quality standards. Continuous testing in the course of DevOps is also mentioned, but not described in more detail. Li, Xu, Hou, *et al.* [165] examine the challenges of developing and operating consortium blockchain solutions. Within their work, they discuss eight pairs of challenges and solutions for different phases of developing and operating such systems. One of the implications identified in their study is that applying DevOps culture and practices can be beneficial to overcome several challenges. Unfortunately there are no details on how to practically address this. Yussupov, Falazi, Breitenbücher, *et al.* [82] analyze how blockchain technology and smart contracts fit into

the serverless architectural style of developing cloud-native applications. The authors picture and derive a set of scenarios in which blockchains act as different component types in serverless architectures. Moreover, implementation requirements that have to be fulfilled to successfully use blockchains and smart contracts in these scenarios are formulated. In the course of this, DevOps requirements are also discussed, more specifically under the aspects to support the development of smart contracts and deployment automation, but not in sufficient detail. Other work in the broader context can also be cited that uses blockchain technology to improve DevOps and software development processes, particularly with respect to integrity and auditability. These include papers by Yilmaz, Tasel, Tuzun, *et al.* [166] to enhance development through a distributed record of software development events and Beller and Hejderup [167] to address trust issues through democratized build services or package repositories.

To our best knowledge, no academic work exists to date that addresses DevOps with a focus on blockchain-based software engineering. It is the goal of our work to make a first contribution in this respect in order to remedy this lack.

7.3. Research Study Design

Given the fact that our research objective is strongly linked to field applications of blockchain and that practical knowledge is often conveyed in practitioner reports, we decided to conduct a research methodology that is guided by the pattern derivation approach [37], where we define a pattern as the conceptual equivalent of (best) practices. In accordance with this scheme and based on our research questions, we applied GT techniques [38] [39] for theory building where patterns are discovered (“mined”) and codified (“written”). Driven by our research questions and known practices from our own experience, we searched the major search engines (e.g., Google, Bing) for the following search string (“Blockchain” OR “Smart Contracts”) AND (CI/CD OR “Continuous Integration” OR “Continuous Delivery” OR “Continuous Deployment” OR “DevOps” OR IAC OR “Infrastructure as Code”) in order to gather a number of technically in-depth sources from the so-called “gray” literature [40] (e.g., practitioner reports, practitioner blogs). In addition, we searched GitHub for typical CI/CD configuration files (e.g., Travis CI: `.travis.yml`, GitLab CI/CD: `.gitlab-ci.yml`) which contain smart contract development frameworks (e.g., Truffle [168], Hardhat [169] [formerly Buidler]) to study their configuration. This resulted in a total of 1343 (Truffle 1313, Hardhat 30) `.travis.yml` and 53 (Truffle 50, Hardhat 3) `.gitlab-ci.yml` hits. Overall, results were filtered and reviewed for suitability according to predefined inclusion and exclusion criteria, which encompassed to accept sources of any type that relate to the topics at hand and exclude works that seem unbalanced in presentation. The resulting source pool [170] was then analyzed using GT techniques. This included a close examination and labeling of materials with labels (“codes”) and optional memos explaining important

7. Blockchain DevOps

aspects of the findings while establishing conceptual relationships among the codes (“axial coding”) to identify candidate patterns. In this process, pattern discovery and validation occurred stepwise in several iterative phases, using new sources (inspired by previous iterations) to constantly compare, revise, and contrast patterns. The primary stopping criterion, as is common in GT-based studies, was theoretical saturation, i.e., a state in which adding new sources no longer yields new insights.

7.4. DevOps for Blockchain Smart Contracts

The core DevOps concepts and activities in the blockchain domain may not be very different from traditional software development. Developers work in a local branch on the source code for smart contracts and dependent applications, add new features or apply corrections to that code, test those changes, and submit their work to a source control management system from which a solution can be build. A release pipeline then deploys the smart contracts or dependent applications to one or more system environments. However, some inherent blockchain peculiarities cause specific constraints that need to be considered when adopting DevOps principles. In the following, we look at core aspects of DevOps for smart contracts and blockchain-based solutions, focusing on considerations and specific approaches for incorporating CI and CD. Since it is useful to divide CI/CD processes into phases, the content on these topics has been organized accordingly by key phases, namely for CI into the phases *Code*, *Build*, and *Test* and for CD into the phases *Release*, *Deploy*, *Operate*, and *Monitor*.

7.4.1. Preliminary Considerations

Before we turn to the details of CI/CD in the scope of blockchain smart contracts, we will first take up a number of considerations that should be made upfront in the course of a DevOps application in the blockchain area.

Governance and Responsibilities

It plays a role whether smart contracts are developed across members of a consortium or autonomously without the influence of others for a broad user base, which usually also affects the choice regarding a permissionless or permissioned blockchain. This aspect influences the distribution of competencies, authorizations, and runtime dependencies, which accordingly also have an impact on DevOps, e.g. with regard to the management of the source code, the development platforms, the infrastructure, the blockchain and the implementation of test and roll-out scenarios. For example, a holistic DevOps strategy for a consortium solution must take into account not only local development of a solution including testing, but also testing at the consortium level and testing by blockchain members for member-specific applications. Overall, DevOps in a multiparty

environment is more complex in terms of defining policies, responsibilities, roles, environments, and build/deployment pipelines.

Key Management

Cryptographic keys must be maintained to operate own nodes, set up test networks, and manage identities under which transactions are created/verified and blockchain code (updates) are deployed. Problems arise if the infrastructure holding these keys is compromised. CI/CD solutions, as such infrastructure, must therefore be subject to strong security measures and restrictions. As a general precaution, keys and other credentials should not be stored in source files, configuration files, environment variables or file systems. To prevent accidental disclosure of keys via the Version Control System (VCS), secret scanning tools can be used (e.g. gitleaks, truffleHog). It is recommended to run these tools locally before each commit, which can be realized with so-called hooks (pre-commit hooks for git) that run custom scripts when important actions take place in the VCS, or to integrate them into the build validation process to prevent merging code that reveals secrets. Storing keys in CI/CD systems may be an option in some cases as long as only keys for test environments are stored. As soon as keys are stored for production systems, it becomes dangerous because the security of a CI/CD system cannot be relied upon. A better approach is to use a dedicated key management solution (e.g., Hashicorp Vault [69]) for securely accessing secrets. Such a solution provides a unified interface for accessing secrets while enabling strict access control and recording a detailed audit trail. With this approach, secrets must be explicitly requested via a secure authentication method (e.g., JSON Web Token) when CI jobs are executed, rather than being provided via variables.

Upgradability

Smart contracts are immutable by design, meaning they cannot be upgraded once they are deployed. Although blockchain-based software benefits significantly from this fundamental immutability paradigm, some degree of mutability is required for bug fixing and evolving smart contracts over time. To resolve this contradiction, there are ways to update smart contracts (see Section 7.4.3) with different types of governance (e.g., single-authority, multi-authority) to control the update process. This aspect can become a weak point if not managed with great care. Interacting users need an increased level of trust in the upgrade process and its management, since updates can significantly change the behavior of the system. In the context of DevOps, the lifecycle management of smart contracts must be embraced. This means that scenarios such as updating existing contracts and compatibility with applications across multiple contract versions should be addressed and considered in testing as well.

7.4.2. Continuous Integration (CI)

Continuous Integration is a DevOps practice for automating the integration of code changes made by multiple developers. More specifically, it allows developers to frequently integrate changes (merge code) into a central repository, where each integration is validated by an automated build, including tests, to catch integration errors up front. The main goals of CI are to optimize software quality by detecting and fixing bugs faster, and to minimize the time needed to validate and deploy software updates. The general CI flow is as follows: Developers have a local copy of the code on which they make changes and run local tests, once tests are successful they commit their changes and then submit a merge request. This request to merge code changes into a shared code repository is then reviewed through an approval process and depends on the success of a series of automated tests included in the build pipeline. Whereby the build pipeline is typically triggered on every merge request that targets the main branch as well as whenever a commit is pushed to that branch.

These basic principles and action steps can also be applied to blockchain-based development. In our research, we found that already established CI solutions (e.g., Jenkins [171], Travis CI [172], CircleCI [173], Gitlab CI/CD [174], GitHub Actions [175]) provide sufficient means to build and test smart contracts and are also practically used for these purposes. These integrations usually consist of a dedicated CI environment (CI server) that monitors a code repository and performs automated actions in a (dockerized) shell environment when changes occur to check the state of that code along with the change that occurred. In general, there are a number of frameworks for smart contract development that support the management and automation of recurring tasks that occur during the creation process. As such, these frameworks are also essential in a CI approach to more easily automate certain repetitive tasks (like testing). Corresponding tools (e.g., Truffle [168], Hardhat [169], Embark [176], Brownie [177], Waffle [178]) aim to provide a comprehensive development solution with an integrated testing blockchain to facilitate compiling, deploying, testing, and debugging smart contracts. These tools are usually available as Command Line Interface (CLI), either as pre-built Docker images, or they can be easily installed and run in a (dockerized) shell environment. In this manifestation, they can also be easily applied in a CI pipeline during various processing steps.

Code

The code phase focuses on core development tasks within IDEs supported by appropriate plugins and frameworks. A suitable programming environment helps to avoid inconsistent styling, security vulnerabilities and code anti-patterns, which can later lead to the failure of automated tests.

Solidity code is typically written either in the web-based Remix IDE [179] with integrated compiler and Solidity runtime environment or locally in a code editor of choice. In the Remix IDE, plugins can perform a variety of tasks such as verifying contracts, linting, generating documentation,

compiling, debugging, deploying, and much more to support a rapid development cycle. When developing in a local IDE supported by a VCS system, as is typically the case when implementing larger projects, such focused integrated IDE support and abundance of development tools as in Remix is not yet present. As a way out, there is currently either the possibility to integrate the local file system into Remix, or conversely, there is the embryonic option to integrate Remix plugins into a local IDE (e.g., remix-vscode for Visual Studio Code [180]). Overall, the Remix IDE is designed for UI-driven ad hoc development characterized by non-repetitive tasks, while a local IDE along with the use of CLI tools supports automation.

The general design principle in software engineering of reducing complexity is especially true in the design of smart contracts. Emphasizing the creation of code that is simple and comprehensible, rather than clever, increases code reliability and minimizes the room for errors. Decomposition plays an important role in this context, i.e., breaking down a solution by separable processes and core entities (similar to microservices) to enable independent evolution of individual components. Code should be factored out into components which are well specified and easy to comprehend. Likewise, contracts should be focused on a single task or capability (preferring many simpler smart contracts over a few larger ones) and be designed to minimize the number/size of on-chain transactions/writes (to reduce costs) as well as the dependencies required for testing. For general concerns (e.g., access control), production-tested library contracts (e.g., OpenZeppelin [129]) and standardized contract implementations (e.g., ERC-20 [181]) should be used. Furthermore, contracts should be developed and tested by locking pragmas with a fixed compiler version to avoid the impact and risk of undiscovered bugs in newer compiler versions. In addition all public contract interfaces (everything in the Application Binary Interface (ABI)) should be fully annotated with specially tagged comments in the so-called NatSpec format [182] (inspired by Doxygen [183]) to provide documentation for functions, return variables, etc.

Build

The build phase includes all the steps required to generate the artifacts needed for execution from the source code. Regarding Solidity, this is the compiled bytecode for the EVM and the associated ABI as the interface required to interact with the EVM bytecode. To generate these artifacts there are two compilers, solc, written in C++, and solc-js, which uses Emscripten [184] to cross-compile from solc C++ source code to JavaScript, thus both use the same compiler source code. The solc compiler [185] is available via binary packages for Linux/MacOs and as a Docker image that contains the compiler executable and allows Solidity files to be compiled by mounting a local folder. The solc-js [186] compiler is available via npm as Node.js library [187] with fewer features and compiles purely using JavaScript, so it works in browser and Node.js environments. The recommended way to interface with the Solidity compiler especially for more complex and automated

7. Blockchain DevOps

setups is the so-called JavaScript Object Notation (JSON)-input-output interface. The compiler API expects a JSON formatted input and outputs the compilation result in a JSON formatted output. The compilers can be used either directly or via development frameworks mentioned earlier, which allow easier handling of compiler versions and compiler configuration. In the latter option, the compilation output format may vary depending on the framework used, but is usually represented as a JSON bundle containing useful information related to compiler input/settings (e.g. compiler name, compiler version) and output (e.g. bytecode, Application Binary Interface [ABI], SourceMap, etc.).

In some cases it may be necessary to use a preprocessor (e.g., `solpp` [188]) before compilation, e.g. to reduce the source files by merging referenced imports from the file system, Node.js modules or URLs and their dependencies into a single file. Another reason would be, if the source files contain symbols or macros that should be extended, or if they contain proprietary operations that are only useful during development (e.g. the `console.log()` command from Hardhat) and should be removed. For CI, it is best to keep raw source files in a separate directory and run the preprocessor to output the code to the pipeline's source directory before compiling a project.

Test

Once a build is successful, it is automatically deployed for review in a test environment where a series of automated tests are run. There are numerous ways to perform tests, and this also applies to blockchain-based software. A basic division according to separable components of the software to be tested and the test purpose is useful. For components, a subdivision according to architecture layers (i.e., application, smart contract, data, consensus, network) is suitable [163]. Regarding test purposes, these can be diverse and categorized in different ways, e.g., by the type of execution or focus. In terms of test complexity, a chronological order of unit, integration, system, and acceptance testing is generally used. Figure 7.1 provides an overview and aid to orientation with respect to possible test types. A detailed discussion of all presented test types is beyond the scope of this chapter. Therefore, we mainly focus on smart contract testing with currently established methods and practical tools considered useful in the context of CI.

Testing Environment For testing, smart contracts need to be deployed in a blockchain environment. This environment can be either an existing permanent or a purpose-built ephemeral environment that is specifically provisioned. In order to achieve reproducible results and avoid undue delays, the latter option is usually resorted to by utilizing a (temporary) local (in-memory) blockchain for testing and development purposes (e.g., Ganache [189]) that simulates the characteristics of a real blockchain network. Unlocked and funded accounts are provided and new transactions are mined instantly, making automated tests much faster and cheaper to run. It is also easier to manipulate the blockchain environment, such as changing the gas price, mining speed, and

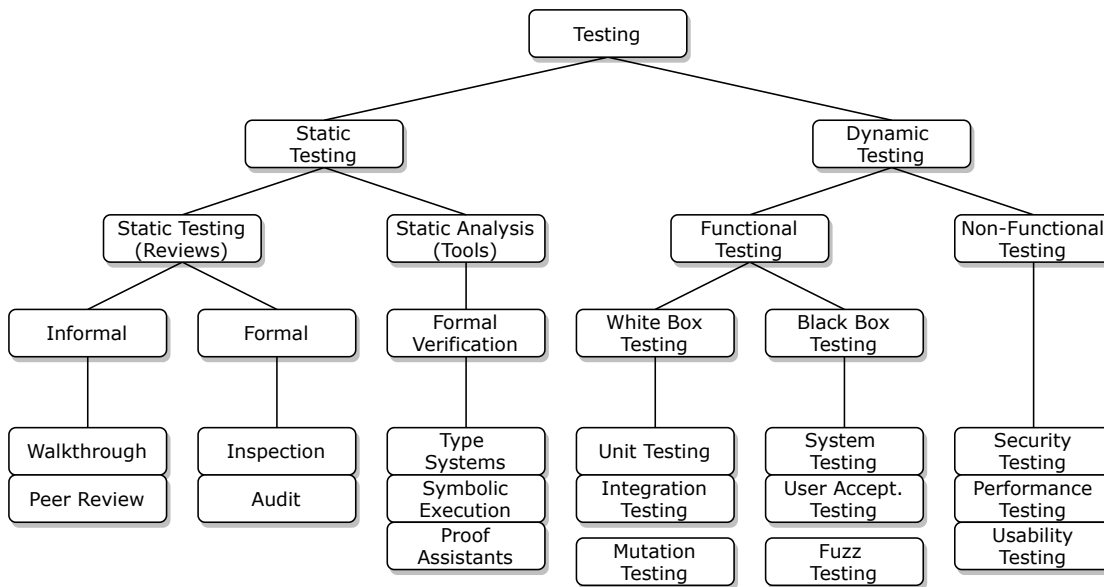


Figure 7.1.: An overview of test types for smart contracts and blockchain-based software.

time flow in general, which can be useful for testing. However, there are some situations where this technique reaches its limits. For example, when there is a need to test a contract against live test data (which is not fully reproducible) to confirm that certain code paths work correctly. Another example is testing compatibility with live contracts for which no source code is available and whose behavior can therefore not be replicated for testing. As a way out, there is the possibility of integrating production blockchains into a local blockchain testing environments (see next paragraph Test Data). In our research, we also came across an unconventional but constrained method of testing deployed contracts live without consuming gas [190]. This exploits the *estimateGas* command by clever construction, calling *estimateGas* on a contract deployment invocation for a contract with test code in the constructor and inferring success from the returned result.

Test Data The tight intertwining of data and logic and the embedding of transactions in a cryptographically secured blockwise data structure, make application testing and test data generation for blockchain-based applications more complex than for traditional applications. As these typically have a clearer separation between the application logic and an underlying database, for which there are usually means to prepopulate it with test data. Applications relying on a relational database, for example, can be setup with initialization scripts that contain *INSERT SQL* statements to prepopulate database tables for testing. However, such an approach is much more difficult with respect to blockchain applications. This is because a desired initial state for a prepopulated blockchain in the context of a business process typically consists of a series of signed transactions

7. Blockchain DevOps

from multiple parties that have taken place via a single or multiple smart contracts.

Generally speaking, there are three ways to equip a blockchain for testing. In the simplest case, one uses an empty blockchain without any transaction history, a blank slate, so to speak. This approach is a viable option before initial deployment and is suitable for local testing of transaction history-independent logic and is usually also the starting point for many test cases. Another possibility is to fill a blockchain synthetically. Transactions are grouped within blocks, so prepopulating a blockchain could be done with custom tooling that forms valid blocks, though it requires careful coordination and sequencing of transactions also in the context of multiple parties. A more practical option in this context, as is common when setting up tests, is to run specific sequences of transactions and save the resulting state using a snapshot. This approach can be promising in complicated test cases to speed up tests by setting up so-called test-fixtures, i.e. consistent test environments with all preconditions a system shall have. With this approach, similar to how one would set up a checkpoint in a video game, a snapshot of the blockchain state at the current block is saved and the blockchain can be restored to this state again and again, which simplifies automated tests. The last way to equip a blockchain for testing is to fork a production blockchain. With this approach, one can simulate the same state (i.e. entire transaction history) as the production blockchain within a local development blockchain. This is achieved by forking the production blockchain from a node endpoint at a given block into the local chain. This usually does not involve downloading the entire production blockchain, but only making calls to endpoint nodes for data as needed. Tasks related to new blocks are processed by the local chain, while tasks for older blocks (e.g., reading historical blockchain data) are processed by the original forked chain. Overall, forking provides a way to interact with production contracts and data in a deterministic manner to test complex interactions locally. Or in other words, it allows to work with existing smart contracts on the blockchain without recreating them and without making real transactions.

Unit and Integration Tests Unit tests are typically automated tests written and run by software developers to ensure that a section of an application (known as the “unit”) meets its design and behaves as intended [191]. According to a survey of Chakraborty, Shahriyar, Iqbal, *et al.* [192] regarding blockchain software, the most common technique to check correctness is unit tests followed by manual code reviews.

In the context of smart contracts, unit tests should ideally cover all contract methods, or at least those that are publicly exposed (i.e., public, external). As part of this, it should be ensured that method return values are as expected and invalid input parameters are rejected. Further, expected execution of reverts and event emitting should be checked, which is a bit more complex as this requires processing of transaction receipts/logs, but there are tools to help with this (e.g. truffle-assertions [193], OpenZeppelin Test Helpers [194]). Since most smart contracts introduce

some form of role-based access control, access privileges should also be verified. With respect to the interdependence of test cases, each test case should be executable in isolation without relying on the state imposed by other test cases. Although it is possible to reduce test execution time by writing cascading test cases, this should be avoided to clearly communicate the intent of test cases (to others) and avoid dependency on the execution of other test cases to minimize complexity.

Integration tests as the next level, are more complex than unit tests, as the behavior of different parts as a whole is tested. For smart contract testing, this can mean interactions and complex scenarios with multiple calls between different dependencies (i.e., users/contracts) of a single contract or across multiple contracts, as well as on all types of oracles and front-end client applications. Subsequently, one can assume two different areas for integration testing. One refers to inter-blockchain interaction between cooperating smart contracts, the other to interaction between smart contracts and dependent client applications (consortium member backends) running elsewhere. The first aspect can be covered with blockchain development frameworks, for the latter other tools for integration testing might be more efficient.

When it comes to writing automated tests for Ethereum, developers have basically two main options: Solidity and JavaScript/TypeScript. Solidity tests can basically test every single function in a contract in a bare-to-the-metal style, as the tests are written in the language of the components under test, resulting in test behavior that is close to EVM. When writing unit tests in Solidity it is necessary to create mock dependencies for oracles or dependent contracts to return predefined values for different methods and arguments. In this context, a common pattern is to create mock contracts as separate Solidity files that mimic the original dependency, but in a way that can be easily controlled by the developer for testing (e.g., [195]). To this end, the mock contract simply extends the dependency and adds/overrides functions. This approach can be cumbersome for several reasons: Another contract is created for each individual mock, thus the test setup is more complex and slower as multiple contracts need to be deployed and put into a specific state for each test. Further, test flexibility is limited to a predefined mock functionality. JavaScript tests, unlike Solidity tests, test contract behavior from an external client viewpoint (using contract abstractions and web3 [196]) and therefore can cover external and public, but not internal or private Solidity functions. Under the hood, JavaScript tests usually rely on established testing utilities such as the Mocha [197] testing framework paired with Chai [198] as an assertion library to test smart contracts asynchronously. This usually makes tests easier to implement and setting up a desired contract state less tedious. Some frameworks have also addressed the cumbersome mock situation of Solidity tests, and there are efforts to create mocks dynamically within the test code (e.g., Waffle [178], MockContract [199]). Some frameworks also allow to execute tests in parallel (e.g., OpenZeppelin [129], Truffle [168]), to speed up testing, if the tests are split across multiple files. In that case, each test file is executed by the test runner at the same time, saving (a massive amount of) time when running a large test suite. The runner as test

7. Blockchain DevOps

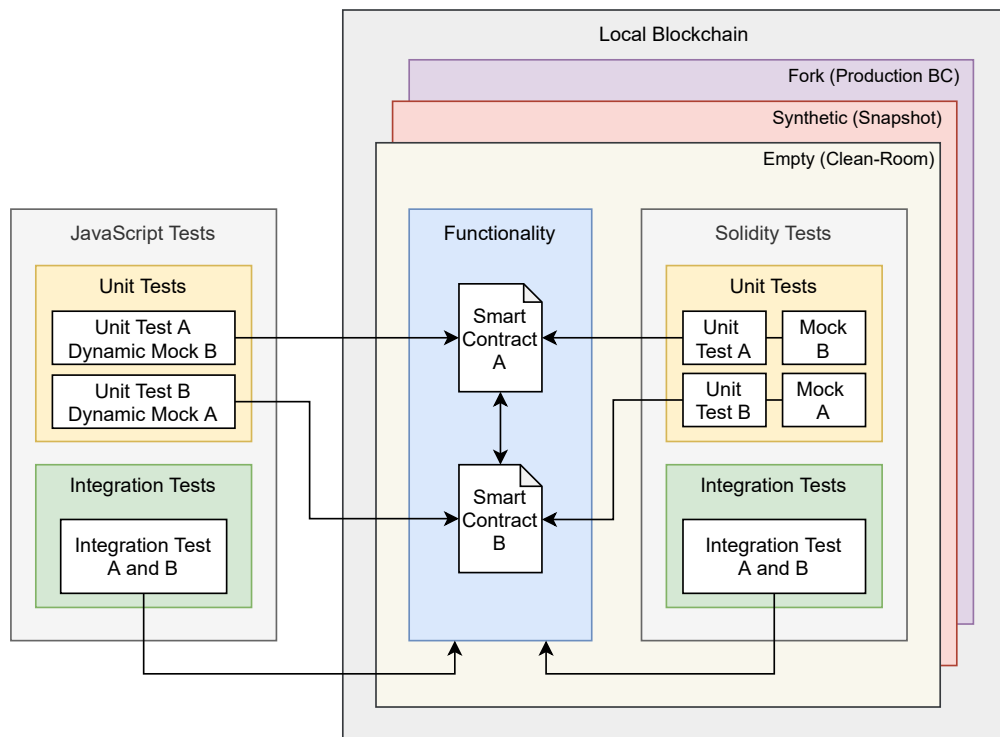


Figure 7.2.: An illustration of the unit and integration test structure for Solidity and JavaScript tests.

environment creates an independent local blockchain for each parallel run, so that there is no risk that the tests interact unexpectedly with each other. Altogether, to make an analogy to the distinction between Solidity and JavaScript tests, the testing of an API can be used, whereby the logic implementation can be tested either externally (including the transmission path) or directly internally. Consequently, one can say that JavaScript tests are more focused on integration tests and Solidity tests are more focused on unit tests, but in practice, one can also perform unit tests with JavaScript. Figure 7.2 illustrates this aspect and the structure for Solidity- and JavaScript-based testing.

Static/Dynamic Analysis While unit and integration tests verify that smart contracts behave as desired according to implemented test cases, they do not uncover potential vulnerabilities in the code itself. For this purpose, it is common to perform static analysis checks on smart contracts. Static code analysis is a debugging technique that examines code with heuristics without actually executing it. A linter can be understood as most basic form of static analysis and can help to improve the code quality and remove minor issues by e.g., checking syntax errors, structural problems, conformance against best practices, and code style guideline violations. A linter tool is typically one of the first applied measures to verify smart contracts. As a best practice, execution before each commit is a good idea, which can be realized with VCS hooks (pre-commit hooks for git). However,

there are also more advanced tools beyond a linter that extend on static analysis and are commonly used to detect security vulnerabilities. These use source code or generated bytecode to examine potential code behavior, vulnerable patterns, and errors that may occur during a program's runtime. Today, many such tools exist (e.g., MythX [200], Securify [201], SmartCheck [202], Slither [203], Manticore [204], Mythril [205]; for an overview see [206], [207]) some of which incorporate a suite of vulnerability detectors that build on analysis techniques such as dynamic analysis, symbolic execution, Satisfiability Modulo Theories (SMT) solving, to name a few. According to a survey by Ayman, Roy, Alipour, *et al.* [208] on the frequency of mentioning such tools on Medium and Stack Exchange, Mythril was mentioned most often.

In addition to these tools, there are also special test tools for smart contracts that rely on techniques already used in other software areas. One example is fuzz testing (e.g., Echidna [209]), an automated testing technique in which software is fed invalid, unexpected, or random data as input and monitored for vulnerable program states and exceptions (e.g., crashes, memory leaks). Another example is (code) mutation testing (e.g., Vertigo [210]), where certain components in a source code are intentionally changed to cause errors and ensure that a test suite is able to detect the changes. This technique can be used to evaluate the quality of existing and to develop new tests.

Overall, analysis tools have different capabilities and detect different types of problems, but they are not perfect, so one has to expect false positives and false negatives. In this regard, a best practice is a combination of different tools to have a better protection and safeguard against potential problems.

Reports An important metric when running tests is test coverage, which is a measure (usually given in percentage) that describes the extent to which the source code of a program is executed when a particular test suite runs. The idea is that tests should execute all code paths of the code under test. If this premise is (largely) fulfilled and the test results are as expected, the code is less likely to contain unforeseen errors. Code coverage tools also exist for Solidity (e.g., solidity-coverage [211], sol-coverage [212]). Since coverage generation tracks which lines are hit during test execution by instrumenting contracts with specific Solidity statements and detecting their execution in a coverage-enabled EVM, coverage detection distorts gas consumption and slows testing. Thus, it is best practice to run coverage as a separate CI job rather than assume its equivalence to an ordinary test procedure. When run in a CI system test coverage can be generated when developers push commits or merge branches.

Another important type of metric is tracking gas consumption. It can be useful to track gas usage per unit test and analyze gas metrics for method calls and deployments. In this context, a gas reporter tool (e.g., eth-gas-reporter [213]) can help to get an overview of the gas costs associated with a smart contract. In a CI environment, the automatic generation of gas reports can be useful

7. Blockchain DevOps

to show differences in gas consumption between code iterations.

7.4.3. Continuous Delivery (CD)

Continuous Delivery is a DevOps practice where software is built in such a way that it can be released to production at any time. For non-blockchain solutions, deployment pipelines deliver updated configurations and code to hosting environments, e.g., as a virtual machine, container, or a serverless function, or provision these environments with an Infrastructure as Code (IaC) approach. The core concepts are the same for blockchain solutions. A proper release pipeline would deploy needed environments and the smart contracts along with dependent applications to one or more system environments.

Release

The release phase is the point at which a build is ready for deployment to the production environment. At this stage, every code change has gone through a series of manual and automated tests, so it can be assumed that problems and regressions are unlikely. Since deploying smart contracts is a rather infrequent and delicate undertaking, it is desirable to have control over when builds are released to production. To this end, a manual approval process can be implemented in the release phase that allows only certain individuals within an organization to authorize a release for production. Before doing so, however, it may be advisable to take further precautions to minimize the risk of undetected issues prior to deployment. This includes conducting independent smart contracts security audits, preferably at least two from different organizations (e.g., Consensys [214], OpenZeppelin [129]), which is especially important for safety-critical areas that manage large amounts of capital. Another advisable aspect in the release phase is to collect all artifacts generated for the deployment and store them in a shared environment to ensure that collaborating parties are fully aligned. Information such as metadata, ABI, bytecode, SourceMap, etc. can be stored centrally for each release. Based on this central archiving it is possible to provide useful services, e.g. an API for client applications to retrieve the version specific ABI for a smart contract. Assuming that one knows the Ethereum account from which contracts are to be deployed, it is even possible to calculate the deployment address of a contract in advance, since the address of a contract is calculated deterministically from the address of its creator (sender) and the number of transactions (nonce) the creator has sent.

Deploy

The deploy phase handles the process of pushing release builds into a production environment. There are several measures to automate the process to make releases reliable and less cumbersome.

Infrastructure as Code (IaC) IaC automates the deployment of system environments to achieve consistency of components, topology, and configuration in order to mitigate discrepancies that can result from the direct application of manual changes to a system. This approach is particularly beneficial for the provisioning process of permissioned blockchains, which is usually complicated and should therefore be automated to avoid errors caused by manual intervention and further save time and resources. There are some IaC utilities on the market to automate the provisioning process (e.g., Terraform [215], Ansible [216], Puppet [217], Chef [218]), which can also be configured appropriately for this purpose, but we have hardly come across this approach in our research. BaaS offerings to ease provisioning are far more common, but are subject to vendor-specific limitations in terms of supported blockchain platforms and hardware infrastructure, furthermore dovetailing with DevOps is more difficult. In the context of permissioned blockchain provisioning, one project worth mentioning that may also be integrated into a DevOps approach is the Blockchain Automation Framework (BAF) [219]. The BAF is a toolkit consisting of various software solutions that are linked together to configure and automatically deploy a scalable and secure blockchain solution with freedom both in terms of the blockchain and infrastructure used.

Smart Contract Deployment When deploying smart contracts to test and production networks, automated solutions are needed to ensure proper deployment in the respective environments. Contracts need to be initialized in a certain order with certain parameters and possibly put to a certain state by calling functions (e.g., to set permissions), depending on the respective deployment target. Specifically, this means that in order to deploy a smart contract, required libraries and dependent contracts must first be deployed. To achieve flexibility with respect to different environments, especially for testing, dependency information is typically passed into the contract via the constructor and is not hard-coded in the contract. One can imagine that manually performing the deployment steps described above for multiple environments is not only time-consuming, but also error-prone. Fortunately, suitable tools (e.g., Truffle Migrations [220], Hardhat Ignition [221]) can be used to reliably automate the necessary steps for linking contracts to other contracts and populating contracts with initial data.

Upgradeable Smart Contracts The only way to upgrade a contract is to deploy a new version of that contract. This procedure requires manually migrating all state information of the old contract and propagating the new contract address to users. To avoid this, there are upgrade mechanisms that can be used to replace contract implementations while preserving their address, state, and balance. Most commonly, a proxy pattern (see Contract Relay in Chapter 5) is used for this purpose in combination with the *delegatecall* mechanism, which allows a function from another contract to be executed as if the function were from the calling contract. Based on these concepts, it is possible

7. Blockchain DevOps

to develop a solution where users interact directly with a proxy that is responsible for handling state information and delegating transactions (via *delegatecall*) to and from other exchangeable (updatable) contracts that contain the associated logic. To avoid requiring the proxy to expose the entire interface of logic contracts, which would be difficult to maintain and make the interface itself not upgradeable, a dynamic forwarding mechanism can be used. In this case, the proxy can forward any call of any function with any set of parameters (with the *fallback* function) to the logic contract; depending on the caller address calls to manage the proxy can also be handled directly. One drawback of the proxy approach is that the proxy contract and its delegate/logic contracts use the same storage layout. Therefore, when handling state variables, care must be taken to avoid scoping and storage collisions between the proxy and logic contracts or between different versions of the latter. In principle, there are three patterns to mitigate this problem: Inherited, Eternal, and Unstructured Storage (see [222], [223]). Inherited Storage refers to a pattern where the proxy contract and logic contracts inherit a storage contract that specifies the storage structure and contains the storage variables they use. Eternal Storage is a pattern where Solidity mappings are used to create the same generic, immutable storage structure for each type variable for any contract. Unstructured Storage is based on a pattern that uses assembly to bypass Solidity's storage layout to set and store values at arbitrary positions in contract storage, usually deriving the storage position from hashing a value that follows some structure. As of Solidity 0.6.4, it is possible to create pointers to structs at arbitrary locations in contract storage, which makes the approach even more appealing, since structs promote encapsulation and can hold an arbitrary number of state variables of any type.

After all, smart contracts can be updated under additional effort and increased complexity. To support developers in this regard, efforts are being made to develop and establish standards and frameworks based on the concepts described above. For example, the EIP-2535 [224] Ethereum Improvement Proposal, titled “Diamonds, Multi-Facet Proxy”, formulates a standard for building modular smart contract systems that can be extended in production. Another example is OpenZeppelin's Upgrades Plugins [225], which can be integrated into existing development environments and workflows to support the deployment and management of upgradeable smart contracts.

Testnet It is common practice to test contracts on a public test network (aka testnet) before deployment on the mainnet. In this context, the organization of releases in stages (alpha, beta) through testnet and mainnet and the tendering of bug bounties should be considered. There are several public testnets for Ethereum, which differ mainly in the consensus algorithm, block time, and supported clients. The main testnets are Goerli, a cross-client Proof-of-Authority (PoA) testnet with a block time of 15 seconds, Rinkeby and Kovan also PoA based testnets with fewer client support and a block time of 15 respectively 4 seconds, and Ropsten, the most similar testnet to the Ethereum mainnet with a PoW consensus and a block time of under 30 seconds. These testnets

simulate the behavior of the mainnet, accordingly Ether is needed to cover the gas costs to run code. However, Ether in testnets has only a dummy function and no value. It can either be mined (in Ropsten) or obtained through a service called a faucet, which issues funds in the form of free Ether to a specified address. This allows developers to test their smart contracts without using real Ether. In general, it is advantageous to test the behavior of smart contracts first with a PoA and later with a PoW test network, as the former are usually more stable and the latter can have unpredictable block times and frequent chain reorganizations.

Operate

When a build is deployed to production, it is important to make sure that everything is running as intended. In this context, when deploying smart contracts on a permissionless blockchain, it may be necessary to publicly verify the deployed contract to establish trust with others. This involves using a recognized service (e.g., Etherscan [226], Sourcify [227]) to confirm that an uploaded and publicly viewable source code is the same as the code compiled on the blockchain. This creates transparency as users know exactly what is being deployed on the blockchain, and allows the public to audit and verify the code to ensure it is actually doing what it is supposed to do. This task can be automated as part of CD with the right tooling (e.g., truffle-plugin-verify [228], hardhat-etherscan [229]).

Monitor

To ensure the health, performance, and reliability of smart contracts and dependent applications, it is necessary to monitor their operation. Monitoring and analyzing the behavior of smart contracts can be done based on various metrics or events to detect erroneous or suspicious behavior. For simple checking purposes, one can use a block explorer (e.g., Etherscan [230], Etherchain [231], Blockchair [232]; for further details, see [233]), which acts as an analytics platform or search engine that allows users to look up real-time data on blocks, transactions, miners, accounts, balances, and other on-chain activities for both the main Ethereum network and the testnets. In addition to these closed services, which cannot be independently verified, there is also the possibility of operating one's own blockchain explorer (e.g. BlockScout [234], Expedition [235]), for which implementations for private EVM-based networks also exist (e.g. Eternal [236]). In addition, it is in principle possible to run a dedicated blockchain node and implement a smart contract activity tracker that sends JSON RPC requests to that node to request transactions, blocks, and logs and scan these for specific characteristics. Following this principle, there are also service providers (e.g. Tenderly [237], OpenZeppelin Sentinel [238], Parsiq [239]) that do the heavy lifting of such a solution and offer a convenient setup of complex events to be monitored. This makes it possible to easily monitor e.g. state variables, function calls, function arguments, function reverts, emitted

7. Blockchain DevOps

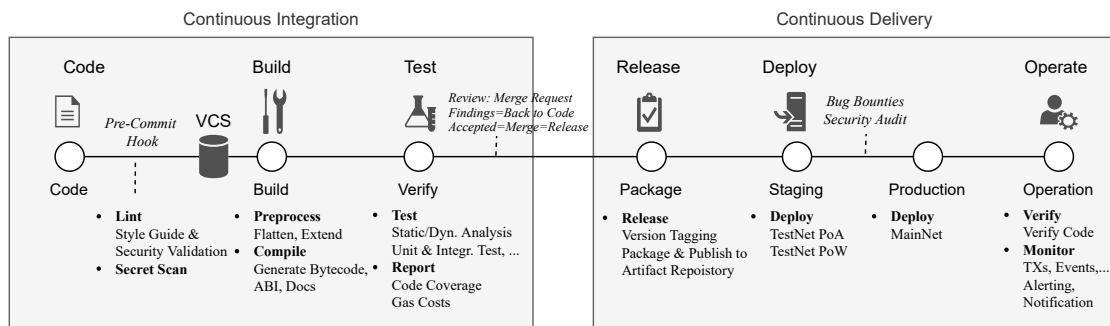


Figure 7.3.: An overview of DevOps stages for smart contracts.

events, gas consumption deviations and send alarms in case of critical behavior via different means (e.g., Webhooks).

In the event that own infrastructure is deployed, it should be monitored as well. There are several monitoring solutions (e.g., Prometheus [240], cAdvisor [241]) that make it possible to observe the activity of network nodes by collecting statistics that can be used to analyze and optimize resource usage and for a better overall understanding of system operation.

7.4.4. CI/CD Overview

To summarize the solution aspects and approaches presented so far in a simple visual manner, Figure 7.3 shows a representation of typical DevOps stages and associated activities that can be incorporated in a CI/CD pipeline. While there are many degrees of freedom in setting up these pipelines, such rough guidance can help in initial setups. A rough sequence for a smart contract project to be built, tested, and deployed might look like the following: First, the sources are pre-processed and compiled, then static/dynamic vulnerability testing and unitary/integration testing is performed, thereafter reports and a release are generated, and finally deployment to the staging/production environment is done followed by verification. To practically illustrate the process, we set up a sample smart contract project (available on GitHub [242]) based on the Hardhat development framework and various helpful development tools, and implemented a GitLab CI/CD pipeline to demonstrate a holistic DevOps approach. For more information on the exact configuration, see Appendix A.2. An interesting finding in the above setup is that while the order of DevOps stages is clear, the order of jobs within stages or the dependencies between different jobs is only to some extent predetermined. For example, it is not always mandatory that the code is already compiled in order to perform static or dynamic code analysis jobs, since respective tools take care of the compilation or trigger it again in the course of their processing. In this case, care should be taken to ensure that the compiler versions used are consistent between different jobs. The fuzzy processing order can also be used to advantage, e.g. jobs can be executed in parallel, since the processing of upstream

jobs is not always mandatory. Thus, the execution time of a pipeline can be significantly reduced.

7.5. Discussion and Threats to Validity

Developing smart contracts at scale is difficult, especially for a distributed team. Add in key management, various responsibilities, different testing strategies, varying computing and testing environments, etc., and this leads to disparate development experiences among developers. The remedy is a DevOps approach and an appropriate tool framework that allows teams to not only manage their development and deployment process, but also integrate threat analysis and release management, for example. While the core DevOps concepts and practices in this area are basically the same as for any other type of software project, there are some peculiarities due to the decentralized nature of blockchain and its immutability. These include a greater focus on testing, in particular the use of testing tools to detect vulnerabilities using static and dynamic code analysis. During our research we have repeatedly encountered the inclusion of various such tools, one can say that they are an essential part of testing smart contracts. Conventional (unit) tests certainly also play an important role, but they are much more costly to implement. Here, the fact that functions of smart contracts can be called publicly means that testing can be compared with the testing of APIs. Therefore, when testing implementations, understanding interfaces and communication points is a key challenge to ensure consistency with defined processes and legacy code. Consequently, UI/end-user-based testing plays a rather minor role, while integration (API) testing should make up the bulk of testing.

Another peculiarity is the differentiated approach to otherwise standard software deployment practices, as smart contracts can be hardly updated. It is worth mentioning here that during our research we found that deployments to production systems are usually done manually, either by deploying locally from a developer's machine or by manually triggering a deployment job defined in a pipeline. When deployment is triggered manually in a DevOps pipeline, it is referred to as Continuous Delivery rather than Continuous Deployment, since the latter involves fully automated deployment of new releases to production. Continuous Deployment for smart contracts would also be conceivable in principle, also with regard to the aforementioned proposals for upgrade mechanisms, but is hardly an issue at present because deployment is currently a rather rare and delicate undertaking over which developers want to have tight control. Therefore, mechanisms associated with CD such as feature toggles (turning functionality on or off at runtime of the software), A/B testing (evaluation of two variants of a system), Canary Releases (gradual rollout of releases to a subset of users) are currently also uncommon. It is likely that future improvements in the ecosystem and recent initiatives to build modular smart contract systems that can be extended in production will make updating smart contracts a matter of course, and thus Continuous Deployment will become more important alongside Continuous Integration. This is also a logical step given

7. *Blockchain DevOps*

necessary code changes and bug fixes that are an integral part of software development.

The presented work is subject to a number of limitations and threats to validity. The practices are mined using a qualitative research method, thus possible biases of individual researchers cannot be fully excluded and might have influenced the results. This includes both the experience- and search-based procedure for finding knowledge sources and the pattern-finding and coding procedure, as other researchers may take a different search approach and interpret and thus code the found content differently. Bias introduced in finding knowledge sources is greatly minimized by the chosen research method, which requires only additional sources according to the inclusion and exclusion criteria, not a specific distribution of sources. To mitigate the threat to internal validity that important information might be missing in sources, we looked at many more sources than needed for theoretical saturation, presuming that it is unlikely that many different sources miss the same important information. However, there is a risk that our approach may lead to a kind of unconscious exclusion of certain sources. We mitigated this problem by using very general and broad search terms. Given the large number of sources included, it is likely that our results can be generalized. Nonetheless, the threat to external validity remains that our results are only applicable to (permissionless) Ethereum based blockchains.

7.6. Conclusion

Over the past decade, DevOps principles have been applied to a wide range of software development industries and disciplines. Essentially, the same principles can be applied to the development and operation of blockchain-based applications. However, applying DevOps in this domain requires a structured approach and corresponding guidelines. To this end, we studied practitioner reports and existing solution approaches, which we analyzed with GT techniques, to infer typical DevOps practices. In the process, we elaborated procedural steps according to the main stages of CI and CD. Based on our findings, we compiled a typical DevOps approach highlighting possible stages and associated activities.

The use of DevOps in the blockchain environment is quite possible today with already established CI/CD solutions and is also lived in practice to make the development process faster, more pleasant, and more controlled. The development tools for smart contracts are sufficiently mature, also with regard to process automation and aggressive testing. The challenge is rather to specify the necessary test and deployment requirements and to select and orchestrate the appropriate tools from a constantly changing set of available development utilities and frameworks. Overall, it can be said that the core DevOps concepts and activities are similar to other areas, with the difference that more rigorous testing and differentiated deployment practices are required due to the inherent immutability of blockchain.

7.6. Conclusion

As blockchain technology has grown, so have the requirements around building and testing related applications. In this regard, in addition to formulating best practices for development, future research can be devoted to devising testing strategies that meaningfully combine the variety of techniques and tools in the field to integrate with DevOps and address outstanding challenges in ensuring reliable blockchain-based applications.

8. Conclusions and Future Work

In this chapter, we revisit the research questions under study in order to summarize the main contributions of this thesis. In addition, we discuss the limitations of our research and introduce open challenges for future work.

8.1. Research Questions Revisited

This thesis contributes to several topics in blockchain software engineering and development. In general terms, this includes the elaboration of fundamental considerations and practices related to the design, development and implementation of blockchain solutions, both from a general architectural perspective and for specific architectural components such as blockchain oracles and smart contracts.

In Section 1.2, we introduced the main research problems covered by this thesis and formulated several main research questions and their refinements (or sub-questions), which were then elaborated in subsequent chapters. In this section, we revisit the research questions in order to summarize our main contributions. To this end, we restate each research question and explain our contributions in a subsequent paragraph.

Research Question 1 (RQ 1)

What design decisions, options, and components need to be considered for blockchain-based applications?

RQ 1.1 *What are the (key) architectural design decisions for blockchain-based applications?*

RQ 1.2 *What are possible design options regarding these decisions and the associated (best) practices?*

RQ 1.3 *Which conceptual components are relevant in the architectural design and what are their relations?*

RQ 1 has been addressed in Chapter 3.

8. Conclusions and Future Work

Starting with Research Question 1, in which we asked what design decisions, options, and components need to be considered for blockchain-based applications, we can note the creation of a feature model as the main contribution. It clearly illustrates possible architectural design decisions as well as decision options and their connections to conceptual components. In addition, we have identified high-level architectural blueprints or patterns in which we describe key components along with their purpose and interaction. A key finding of our research is that designing software architectures for blockchain-based systems, just as in other domains, requires various trade-off decisions to balance desired quality attributes. In the case of blockchain integration, however, this usually boils down to balancing decentralization on the one hand and scalability, privacy, and usability on the other. The feature model provides a good overview of this tension and can be seen as a helpful contribution to a more differentiated understanding. An important insight in this context is that the more decentralized a solution is, the more difficult it is to ensure the quality characteristics of scalability, privacy, and usability. To tackle this challenge, a hybrid architecture approach currently offers a good compromise. This combines decentralized and centralized components so that the advantages of both can be leveraged. In this light, many of the presented design options can be understood as a way to circumvent current blockchain disadvantages by using centralized elements.

Research Question 2 (RQ 2)

What (fundamental) patterns exist to implement blockchain oracles and how do they differ regarding cost and performance?

RQ 2.1 What are best practices and design patterns for implementing blockchain oracles?

RQ 2.2 What are the characteristics of these regarding cost and performance?

RQ 2 has been addressed in Chapter 4.

With respect to Research Question 2, which asks about (fundamental) blockchain oracle patterns and their differences regarding cost and performance, our main contribution is the abstraction of individual technical oracle solutions and the decomposition of these by data flow (inbound vs. outbound and pull vs. push) resulting in four basic oracle patterns. The elaborated patterns are designed in such a way that they consist of distinct building blocks that allow for a separation of concerns which can be encapsulated into modular and reusable implementation components. A quantitative analysis of the four patterns in terms of time performance (latency) and cost impacts reveals that neither cost nor latency are particularly high for a single invocation of any of the patterns. Our experiments in this regard also suggest that the patterns are subject to different distributions with respect to latency, which in most cases can be narrowed down to a certain likely

range, but can also be left by outliers and dominated by the transaction inclusion time.

Research Question 3 (RQ 3)

What are common design patterns and Solidity coding practices for Ethereum smart contracts?

RQ 3.1 *Which design patterns commonly appear in the Ethereum ecosystem and what problems do they solve?*

RQ 3.2 *How do these design patterns map to Solidity coding practices?*

RQ 3 has been addressed in Chapter 5.

We now turn to Research Question 3, which addresses common design patterns in Solidity-encoded smart contracts for the Ethereum blockchain. To this end, we can attribute as our main contribution the identification, grouping, and description of several globally applicable patterns, as well as the elaboration of common principles and relationships among them. Our research shows that the patterns are widely used to address application requirements and common implementation problems. In particular, the patterns resolve issues related to smart contract operation, access control, management, and security. They are explained in a problem- and solution-oriented approach along with advantages and disadvantages to better illustrate their context and applicability. As such, they can be used by developers to solve common problems related to smart contract coding. A generalizing observation is that many of the patterns deal with circumventing the immutability and lack of execution control of smart contracts. This highlights the tension that arises when an autonomous execution environment is confronted with the need for maintainability and upgradeability.

Research Question 4 (RQ 4)

What might a secure-by-design approach to smart contracts look like that starts from a higher level of abstraction and generates an implementation leveraging design patterns?

RQ 4.1 *How and in how far is it possible to bring the abstraction level of smart contracts closer to the contract domain?*

RQ 4.2 *Can higher abstraction levels in combination with code generation (considering platform-specific programming idioms) reduce the risk of smart contract errors?*

RQ 4 has been addressed in Chapter 6.

8. Conclusions and Future Work

Looking at Research Question 4, which is concerned with a secure-by-design approach for smart contracts, our main contribution is the design and study of a DSL with a higher level of abstraction that can be transferred to an implementation. Here, we build on a fluently readable, clause like formalization concept to describe the individual operational intents (commitments) of contract participants. Thus, the contract semantics are described on a higher level and the specifics of an implementation are shifted to lower levels. This decoupling of contract specification and implementation allows free intervention regarding the formulation of the final implementation. It is thus possible to automate platform-specific implementation steps, e.g., by including design patterns or coding abstractions. We have demonstrated this approach by implementing coding idioms and selected patterns that we have elaborated in the previous research question. Overall, our research shows that abstraction and code generation, when used properly, can be a viable way to formulate smart contracts and can be used to increase the efficiency, clarity, and flexibility of code whilst reducing the susceptibility to errors.

Research Question 5 (RQ 5)

What does a typical DevOps approach for blockchain-based applications look like and what are the differences compared to a DevOps approach for traditional software projects?

RQ 5.1 *What are typical stages and activities in a DevOps approach for blockchain-based applications?*

RQ 5.2 *What are the particularities of using DevOps in blockchain-based software development?*

RQ 5 has been addressed in Chapter 7.

The last research question, Research Question 5, deals with a DevOps approach for blockchain-based applications and possible differences compared to a DevOps approach for traditional software projects. In this regard, our main contribution is the compilation of a typical DevOps approach, aligned according to typical phases of CI and CD, highlighting possible stages and associated activities. Our research shows that the core DevOps concepts and activities are similar to those in other domains and are quite possible with already established solutions, with the difference that more rigorous testing and differentiated deployment practices are required due to the inherent immutability of blockchain. In this context, the actual DevOps implementation is not in itself a difficult undertaking. Rather, the challenge is to specify the necessary testing and deployment requirements and to select and orchestrate the appropriate tools from an ever-changing set of available development tools and frameworks.

8.2. Limitations and Threats to Validity

The work presented in this thesis contributes to a better understanding and formulation of recommended actions in the implementation of blockchain-based applications, but some issues remain. A discussion of the limitations of our research is partially included in the relevant chapters of the thesis. In this section, we holistically summarize the main limitations of our work.

In our research, we have mainly relied on qualitative research methods, as the research questions under study are aimed at gaining in-depth insights into topics that are not yet well elaborated or understood in order to comprehend concepts, thoughts or experiences and subsequently derive theories. Accordingly, our work is also subject to the limitations associated with qualitative research. This includes, for example, the way data and observations are described and interpreted and how the data may be deliberately or inadvertently altered to fit a particular theory [243]. In this context, there is also always some bias on the part of the researcher who interprets the data according to his or her own beliefs and views or includes only data that he or she considers relevant.

We discuss below some of the limitations and potential threats to the validity of our study and steps we have taken to minimize or mitigate them. The discussion takes place in the context of the most commonly cited quality criteria in qualitative research: credibility, reliability, confirmability, and transferability [244]. These are well suited for evaluating exploratory research for theory building and derive from the classical criteria of test validity (internal validity, external validity, reliability, and objectivity).

Credibility (Internal Validity): Credibility is concerned with the confidence in the truth of the data and derived interpretations. The patterns and recommendations in this thesis are mined using a qualitative mining process (as it is usual) based on collected data. Here, a threat to credibility lies in the bias of selected data, i.e. the sources included in our pool of objects under study. It is possible that the data does not represent the entire area or over represents sub-areas about which we want to make assumptions. To overcome this, snowballing has been used to track down references and citations in data sources, and information was collected until no new knowledge could be obtained. In this context, it should be mentioned that most available sources only deal with a handful of more prominent blockchain solutions, so there is a risk that our results can only be applied to similar architectures. Another concern in regards to credibility are possible misinterpretations or biases of individual researchers, which cannot be completely ruled out and may have influenced our results. This includes the pattern discovery and codifying procedures, as other researchers may have different interpretations and may code differently. Here, one concern for subjectivity is that the same researchers have been involved in the consolidation of patterns and recommendations. Another example for unavoidable subjectivity concerns the selection of higher-level dimensions in categorizations and their granularity, which has a major impact on the representation of results. We

8. *Conclusions and Future Work*

attempted to mitigate the stated subjectivity issues by having no prior hypotheses or caveats when interpreting the data and analyzing all data even if it appeared unhelpful at first glance. Nonetheless, we can not claim viability or any form of completeness for our results.

Transferability (External Validity): Transferability is concerned with the extent to which findings or results can be transferred to other contexts and settings. Due to the fact that basic functional principles are the same across different blockchain implementations, it is likely that our results can be generalized. This is especially true for the elaborated architectural design decisions and oracle patterns. Although we have relied on the Ethereum platform to describe these topics, the results are fundamental in nature and can be generalized to other blockchain platforms. As for the smart contract patterns and the DevOps approach, the results are likely to have limited generalizability, as the patterns and recommendations were elaborated specifically in the scope of the Ethereum platform. A generalization is therefore only possible in an attenuated form, where the highlighted areas of concern in the development and implementation of smart contracts are likely to apply to other platforms as well. For the developed DSL, generalizability is given in that the abstract language constructs for formulating smart contracts are independent of an explicit implementation, but as a result the expressive power suffers and is not comparable to manual implementations.

Dependability (Reliability): Dependability concerns the process for showing that the findings are consistent and could be repeated. In the best case, this would mean that exactly the same results are obtained when we our research is repeated. Since this cannot be fully expected in a qualitative research setting, alternative criteria are overall comprehensibility, flow of arguments, and logic. We attempted to meet these criteria by documenting our proceedings, tracking sources of knowledge, and being consistent in both the process and product of our research.

Confirmability (Objectivity): Confirmability refers to the potential for agreement on interpretation between two or more people. In other words, it is the extent to which the results can be confirmed by others. Here, a threat to confirmability is that our work lacks empirical evidence and feedback by practitioners. However, one can say that the confidence that the results can be confirmed or corroborated by practitioners in the field is increased by the fact that we have based much of our analysis on gray literature, which can mostly be attributed to practitioners. In addition, as a further measure to increase confirmability and confidence in the quality of the proposed patterns and recommendations, we have always endeavored to find several instances that substantiate our claims and assertions.

8.3. Future Work

Although this work contributes to the above research questions, additional work is needed to further improve the state of the art. In the following, we conclude with a summary of open challenges and research questions for the future that have surfaced during the execution of our research as well

as from the aforementioned limitations.

At the beginning of our work, we dealt with the architectural design of blockchain-based applications. This topic is essentially about the efficient integration of on-chain and off-chain worlds, where certain technical trade-offs have to be made in order to achieve predefined quality goals. Future research in this area could deal with finding metrics that allow a quantitative and accurate assessment of performance changes of individual design decisions. Another starting point for future research would be to investigate (architectural) migration patterns for transferring existing functionality or architectural components to blockchain technology, provided that blockchain is considered as a service component. Other research ideas include the development of a decision-making tool to select the most suitable blockchain and patterns for a given context.

With respect to our work on oracle patterns in the context of the Ethereum blockchain, our research could be deepened in the future with further studies for different blockchain platforms. Moreover, the patterns could be applied in different contexts to other use cases that span different application domains. In addition, it would be interesting to study the combination of different oracle patterns or the effects on execution costs in regards to the exchanged data rate and data volume.

Regarding the design patterns for smart contracts, future research could be directed to propose new patterns, also with regard to specific topics such as efficiency in terms of gas optimization to minimize execution costs. The subject area has not yet been exhausted, and due to constant development, it is likely that patterns will need to be changed, added, or removed as surrounding conditions change. With a large number of patterns, the elaboration of an ontology to identify and distinguish patterns and their relationships, as well as a taxonomy to formalize the terminology and structure of hierarchical relationships between patterns, is also an interesting research task. In addition, the collected patterns could be compared to coding practices that are evolving in other smart contract platforms. This also reveals a weakness in the research field addressed, namely that most research in this area is focused on the Ethereum blockchain. If research were also conducted off the beaten path, more abstract design patterns could be uncovered that are independent of the underlying implementation framework and apply to smart contracts in general.

With regard to the proposed smart contract DSL, there are also several opportunities for further research. For example, the efficiency of our approach could be tested in an experiment with practitioners and the insights gained could be used to optimize the language for further improvements and extensions. With respect to the abstract formalization, additional semantic checks could be introduced at the clause level to detect conflicting clause specifications. Furthermore, the Solidity code generation process could be extended to include more common design patterns, coding abstractions, and more powerful code inference mechanisms. To draw even more insights, code generation for another smart contract platform could be integrated. This could provide insight into the general applicability of the proposed smart contract abstraction mechanisms and their

8. Conclusions and Future Work

transferability to a target implementation.

In view of our work on DevOps in the context of the Ethereum blockchain, our research could be deepened in the future by further studies on DevOps operations for other blockchain platforms. Research perspectives arise especially with respect to the description of appropriate testing strategies and scenarios as well as delivery strategies. Devising testing strategies is particularly recommended to combine the variety of techniques and tools in this area in a meaningful way in order to overcome the remaining challenges in ensuring reliable blockchain-based applications.

Bibliography

- [1] “Global Cryptocurrency Market Charts | CoinMarketCap,” [Online]. Available: <https://coinmarketcap.com/charts/> (visited on Feb. 23, 2022).
- [2] H. Cooper, L. V. Hedges, and J. C. Valentine, *The handbook of research synthesis and meta-analysis*. Russell Sage Foundation, 2019.
- [3] B. Kitchenham and S. Charters, “Guidelines for performing systematic literature reviews in software engineering,” 2007.
- [4] V. Garousi, M. Felderer, and M. V. Mäntylä, “Guidelines for including the grey literature and conducting multivocal literature reviews in software engineering,” *arXiv preprint arXiv:1707.02553*, pp. 1–26, 2017. arXiv: 1707 . 02553. [Online]. Available: <http://arxiv.org/abs/1707.02553>.
- [5] J. W. Creswell and J. D. Creswell, *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2017.
- [6] Charmaz, *Constructing Grounded Theory research*. sage, 2014, pp. 33–59, ISBN: 0-85702-913-4.
- [7] A. R. Hevner, S. T. March, J. Park, and S. Ram, “Design science in information systems research,” *MIS Quarterly: Management Information Systems*, 2004, ISSN: 02767783. DOI: 10 . 2307/25148625.
- [8] A. Hevner and S. Chatterjee, “Introduction to Design Science Research,” 2010. DOI: 10 . 1007/978-1-4419-5653-8_1.
- [9] V. Vaishnavi and W. Kuechler, “Design Science Research Methods and Patterns: Innovating Information and Communication Technology,” in *Design Science Research Methods and Patterns: Innovating Information and Communication Technology*. 2007, p. 226, ISBN: 9780429119385. DOI: 10 . 1201/9781420059335.
- [10] M. Wohrer and U. Zdun, “Smart contracts: Security Patterns in the Ethereum Ecosystem and Solidity,” in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, vol. 2018-Janua, IEEE, Mar. 2018, pp. 2–8, ISBN: 978-1-5386-5986-1. DOI:

Bibliography

- 10.1109/IWB0SE.2018.8327565. [Online]. Available: <https://ieeexplore.ieee.org/document/8327565/>.
- [11] M. Wohrer and U. Zdun, "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, IEEE, Jul. 2018, pp. 1513–1520, ISBN: 978-1-5386-7975-3. DOI: 10.1109/Cybermatics_2018.2018.00255. [Online]. Available: <https://ieeexplore.ieee.org/document/8726782/>.
- [12] M. Wohrer and U. Zdun, "Domain Specific Language for Smart Contract Development," in *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, IEEE, May 2020, pp. 1–9, ISBN: 978-1-7281-6680-3. DOI: 10.1109/ICBC48266.2020.9169399. [Online]. Available: <https://ieeexplore.ieee.org/document/9169399/>.
- [13] M. Wohrer and U. Zdun, "From Domain-Specific Language to Code: Smart Contracts and the Application of Design Patterns," *IEEE Software*, vol. 37, no. 5, pp. 37–42, Sep. 2020, ISSN: 0740-7459. DOI: 10.1109/MS.2020.2993470. [Online]. Available: <https://ieeexplore.ieee.org/document/9089272>.
- [14] R. Mühlberger, S. Bachhofner, E. Castelló Ferrer, *et al.*, "Foundational Oracle Patterns: Connecting Blockchain to the Off-Chain World," in *Lecture Notes in Business Information Processing*, vol. 393 LNBIP, 2020, pp. 35–51, ISBN: 9783030587789. DOI: 10.1007/978-3-030-58779-6_3. arXiv: 2007.14946. [Online]. Available: https://link.springer.com/10.1007/978-3-030-58779-6_3.
- [15] M. Wohrer and U. Zdun, "Architectural Design Decisions for Blockchain-Based Applications," in *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, IEEE, May 2021, pp. 1–5, ISBN: 978-1-6654-3578-9. DOI: 10.1109/ICBC51069.2021.9461109. [Online]. Available: <https://ieeexplore.ieee.org/document/9461109/>.
- [16] M. Wohrer, U. Zdun, and S. Rinderle-Ma, "Architecture Design of Blockchain-Based Applications," in *2021 3rd Conference on Blockchain Research and Applications for Innovative Networks and Services (BRAINS)*, IEEE, Sep. 2021, pp. 173–180, ISBN: 978-1-6654-3924-4. DOI: 10.1109/BRAINS52497.2021.9569813. [Online]. Available: <https://ieeexplore.ieee.org/document/9569813/>.
- [17] M. Wohrer and U. Zdun, "DevOps for Ethereum Blockchain Smart Contracts," in *2021 IEEE International Conference on Blockchain (Blockchain)*, IEEE, Dec. 2021, pp. 244–251,

- ISBN: 978-1-6654-1760-0. DOI: 10.1109/Blockchain53845.2021.00040. [Online]. Available: <https://ieeexplore.ieee.org/document/9680513/>.
- [18] Ethereum. “Ethereum Project,” [Online]. Available: <https://www.ethereum.org/> (visited on Jan. 7, 2020).
- [19] “Scalable Blockchain Infrastructure: Billions of transactions and counting | Solana: Build crypto apps that scale,” [Online]. Available: <https://solana.com/> (visited on Feb. 22, 2022).
- [20] “Polkadot: Decentralized Web 3.0 Blockchain Interoperability Platform,” [Online]. Available: <https://polkadot.network/> (visited on Feb. 22, 2022).
- [21] “Ergo,” [Online]. Available: <https://ergoplatform.org/en/> (visited on Feb. 22, 2022).
- [22] “Algorand | The Blockchain for FutureFi | Algorand,” [Online]. Available: <https://www.algorand.com/> (visited on Feb. 22, 2022).
- [23] “Cardano | Home,” [Online]. Available: <https://cardano.org/> (visited on Feb. 22, 2022).
- [24] G. Wood, “Ethereum: a secure decentralised generalised transaction ledger,” *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, 2014, ISSN: 1098-6596. DOI: 10.1017/CBO9781107415324.004. arXiv: arXiv:1011.1669v3.
- [25] Ethereum Foundation. “LLL PoC 6.” (2014), [Online]. Available: <https://github.com/ethereum/cpp-ethereum/wiki/LLL-PoC-6/7a575cf91c4572734a83f95e970e9e7ed64849ce>.
- [26] “Serpent | Ethereum Wiki,” [Online]. Available: <https://github.com/ethereum/wiki/wiki/Serpent>.
- [27] “ethereum/viper: New experimental programming language,” [Online]. Available: <https://github.com/ethereum/viper>.
- [28] “Solidity — Solidity 0.4.24 documentation,” [Online]. Available: <http://solidity.readthedocs.io/en/v0.4.24/>.
- [29] S. Tai, J. Eberhardt, and M. Klems, “Not ACID, not BASE, but SALT: A transaction processing perspective on blockchains,” in *CLOSER 2017 - Proceedings of the 7th International Conference on Cloud Computing and Services Science*, 2017, ISBN: 9789897582431. DOI: 10.5220/0006408207550764.
- [30] S. Porru, A. Pinna, M. Marchesi, and R. Tonelli, “Blockchain-oriented software engineering: Challenges and new directions,” *2017 IEEE/ACM 39th International Conference*

Bibliography

- on Software Engineering Companion (ICSE-C)*, no. February, pp. 169–171, 2017. DOI: 10.1109/ICSE-C.2017.142. arXiv: 1702.05146.
- [31] F. Wessling, C. Ehmke, O. Meyer, and V. Gruhn, “Towards Blockchain Tactics: Building Hybrid Decentralized Software Architectures,” in *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, IEEE, 2019, pp. 234–237, ISBN: 9781728118765. DOI: 10.1109/ICSA-C.2019.00048.
- [32] M. Marchesi, L. Marchesi, and R. Tonelli, “An Agile Software Engineering Method to Design Blockchain Applications,” in *Proceedings of the 14th Central and Eastern European Software Engineering Conference Russia*, 2018, pp. 1–8, ISBN: 9781450361767. DOI: 10.1145/3290621.3290627. arXiv: 1809.09596.
- [33] L. Marchesi, M. Marchesi, and R. Tonelli, “ABCDE—agile block chain DApp engineering,” *Blockchain: Research and Applications*, vol. 1, no. 1-2, p. 100 002, Dec. 2020, ISSN: 20967209. DOI: 10.1016/j.bcra.2020.100002. arXiv: 1912.09074. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/S2096720920300026>.
- [34] C. Udokwu, H. Anyanka, and A. Nort, “Evaluation of Approaches for Designing and Developing Decentralized Applications,” in *Proceedings of the 2020 4th International Conference on Algorithms, Computing and Systems*, 2020, pp. 55–62.
- [35] U. Bodkhe, S. Tanwar, K. Parekh, *et al.*, “Blockchain for Industry 4.0: A comprehensive review,” *IEEE Access*, vol. 8, pp. 79 764–79 800, 2020, ISSN: 21693536. DOI: 10.1109/ACCESS.2020.2988579.
- [36] R. Viswanathan, D. Dasgupta, and S. R. Govindaswamy, “Blockchain Solution Reference Architecture (BSRA),” *IBM Journal of Research and Development*, vol. 63, no. 2/3, pp. 1–12, 2019. DOI: 10.1147/JRD.2019.2913629.
- [37] D. Riehle, N. Harutyunyan, and A. Barcomb, “Pattern Discovery and Validation Using Scientific Research Methods,” *arXiv preprint arXiv:2107.06065*, 2021.
- [38] B. G. Glaser and A. L. Strauss, *Discovery of grounded theory: Strategies for qualitative research*. Routledge, 2017, ISBN: 9780203793206. DOI: 10.4324/9780203793206.
- [39] J. M. Corbin and A. Strauss, “Grounded theory research: Procedures, canons, and evaluative criteria,” *Qualitative Sociology*, 1990, ISSN: 01620436. DOI: 10.1007/BF00988593.
- [40] V. Garousi, M. Felderer, M. V. Mäntylä, and A. Rainer, *Benefitting from the grey literature in software engineering research*, 2019. DOI: 10.1007/978-3-030-32489-6_14.
- [41] “maxwoe/bc_architecture_design: Architectural Design Decisions for Blockchain-Based Applications - Knowledge Sources,” [Online]. Available: https://github.com/maxwoe/bc_architecture_design.

- [42] C. Richardson. “Microservices Patterns.” arXiv: 1 - 933988 - 16 - 9. (2017), [Online]. Available: <https://microservices.io/patterns/index.html>.
- [43] K. Wu, Y. Ma, G. Huang, and X. Liu, “A first look at blockchain-based decentralized applications,” *Software: Practice and Experience*, no. April, pp. 1–18, 2019, ISSN: 0038-0644. DOI: 10.1002/spe.2751. arXiv: arXiv:1909.00939v1.
- [44] K. M. Kina-Kina, H. E. Cutipa-Arias, and P. Shiguihara-Juarez, “A comparison of performance between fully and partially decentralized applications,” in *Proceedings of the 2019 IEEE 26th International Conference on Electronics, Electrical Engineering and Computing, INTERCON 2019*, 2019, ISBN: 9781728136462. DOI: 10.1109/INTERCON.2019.8853524.
- [45] F. Wessling and V. Gruhn, “Engineering Software Architectures of Blockchain-Oriented Applications,” *Proceedings - 2018 IEEE 15th International Conference on Software Architecture Companion, ICSCA-C 2018*, pp. 45–46, 2018. DOI: 10.1109/ICSCA-C.2018.00019.
- [46] Q. Zhou, H. Huang, Z. Zheng, and J. Bian, “Solutions to Scalability of Blockchain: a Survey,” *IEEE Access*, vol. 8, no. January, pp. 16 440–16 455, 2020, ISSN: 21693536. DOI: 10.1109/ACCESS.2020.2967218.
- [47] L. Gudgeon, P. Moreno-Sanchez, S. Roos, P. McCorry, and A. Gervais, “SoK: Layer-Two Blockchain Protocols,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Springer, vol. 12059 LNCS, 2020, pp. 201–226, ISBN: 9783030512798. DOI: 10.1007/978-3-030-51280-4_12.
- [48] R. Zhang, R. Xue, and L. Liu, “Security and privacy on blockchain,” *ACM Computing Surveys*, vol. 52, no. 3, 2019, ISSN: 15577341. DOI: 10.1145/3316481. arXiv: 1903.07602.
- [49] J. Partala, T. H. Nguyen, and S. Pirttikangas, “Non-interactive Zero-knowledge for Blockchain: A Survey,” *IEEE Access*, 2020, ISSN: 21693536. DOI: 10.1109/ACCESS.2020.3046025.
- [50] X. Sun, F. R. Yu, P. Zhang, Z. Sun, W. Xie, and X. Peng, “A survey on zero-knowledge proof in blockchain,” *IEEE Network*, vol. 35, no. 4, pp. 198–205, 2021.
- [51] Y. Wang, F. Luo, Z. Dong, Z. Tong, and Y. Qiao, “Distributed meter data aggregation framework based on Blockchain and homomorphic encryption,” *IET Cyber-Physical Systems: Theory & Applications*, vol. 4, no. 1, pp. 30–37, 2019.
- [52] W. Xu, L. Wu, and Y. Yan, “Privacy-preserving scheme of electronic health records based on blockchain and homomorphic encryption,” *Journal of Computer Research and*

Bibliography

- Development*, vol. 55, no. 10, p. 2233, 2018.
- [53] S. Dziembowski, S. Faust, and K. Hostáková, “General state channel networks,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 949–966.
- [54] S. Dziembowski, L. Eckey, S. Faust, J. Hesse, and K. Hostáková, “Multi-party virtual state channels,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 2019, pp. 625–656.
- [55] D. J. Pacella, A. Sardesai, M. Tadayon, *et al.*, *Blockchain micro-services framework*, 2019.
- [56] “fabric-gateway | Go, Java and Node SDKs for Fabric embedded Gateway,” [Online]. Available: <https://hyperledger.github.io/fabric-gateway/> (visited on Apr. 26, 2022).
- [57] “ActiveMQ,” [Online]. Available: <https://activemq.apache.org/> (visited on Mar. 9, 2022).
- [58] “Messaging that just works - RabbitMQ,” [Online]. Available: <https://www.rabbitmq.com/> (visited on Mar. 9, 2022).
- [59] “Apache Kafka,” [Online]. Available: <https://kafka.apache.org/> (visited on Mar. 9, 2022).
- [60] “Apache Pulsar,” [Online]. Available: <https://pulsar.apache.org/> (visited on Mar. 9, 2022).
- [61] “Blockchain Streaming Analytics,” [Online]. Available: <https://www.youtube.com/watch?v=rY1fEaCvwXk>.
- [62] “Kafka Connect | Confluent Documentation,” [Online]. Available: <https://docs.confluent.io/platform/current/connect/index.html> (visited on Mar. 9, 2022).
- [63] “Ocean Bounty: Smart Contract Event Monitoring Tool,” [Online]. Available: <https://explorer.bounties.network/bounty/2146>.
- [64] “Kafka-web3-connector,” [Online]. Available: <https://github.com/satran004/kafka-web3-connector>.
- [65] “PouchDB, the JavaScript Database that Syncs!” [Online]. Available: <https://pouchdb.com/> (visited on Mar. 9, 2022).
- [66] “GUN - the database for freedom fighters - Docs v2.0,” [Online]. Available: <https://gun.eco/> (visited on Mar. 9, 2022).
- [67] “IPFS Powers the Distributed Web,” [Online]. Available: <https://ipfs.io/> (visited on Mar. 9, 2022).

- [68] “Swarm,” [Online]. Available: <https://www.ethswarm.org/> (visited on Mar. 9, 2022).
- [69] “Vault by HashiCorp,” [Online]. Available: <https://www.vaultproject.io/> (visited on Mar. 9, 2022).
- [70] “Go Ethereum,” [Online]. Available: <https://geth.ethereum.org/> (visited on Mar. 9, 2022).
- [71] “Parity Ethereum Client - OpenEthereum | Parity Technologies,” [Online]. Available: <https://www.parity.io/technologies/ethereum/> (visited on Mar. 9, 2022).
- [72] “Ethereum API | IPFS API and Gateway | ETH Nodes as a Service | Infura,” [Online]. Available: <https://infura.io/> (visited on Mar. 9, 2022).
- [73] “Hosted Blockchain Infrastructure as a Service | QuickNode,” [Online]. Available: <https://www.quicknode.com/> (visited on Mar. 9, 2022).
- [74] X. Xu, C. Pautasso, L. Zhu, Q. Lu, and I. Weber, “A pattern collection for blockchain-based applications,” in *ACM International Conference Proceeding Series*, ACM, 2018, 3:1–3:20, ISBN: 9781450363877. DOI: 10.1145/3282308.3282312.
- [75] V. Rajasekar, S. Sondhi, S. Saad, and S. Mohammed, “Emerging design patterns for blockchain applications,” in *ICSOFT 2020 - Proceedings of the 15th International Conference on Software Technologies*, 2020, pp. 242–249, ISBN: 9789897584435. DOI: 10.5220/0009892702420249.
- [76] Y. Liu, Q. Lu, X. Xu, L. Zhu, and H. Yao, “Applying Design Patterns in Smart Contracts,” in *Proceedings Blockchain-ICBC*, June, vol. 10974, 2018, pp. 92–106. DOI: 10.1007/978-3-319-94478-4_7. [Online]. Available: http://link.springer.com/10.1007/978-3-319-94478-4_7.
- [77] I. Weber, “Blockchain and Services – Exploring the Links: Keynote Paper,” *Lecture Notes in Business Information Processing*, vol. 367, no. October 2019, pp. 13–21, 2019, ISSN: 18651356. DOI: 10.1007/978-3-030-32242-7_2.
- [78] F. Daniel and L. Guida, “A Service-Oriented Perspective on Blockchain Smart Contracts,” *IEEE Internet Computing*, vol. 23, pp. 46–53, 2019, ISSN: 19410131. DOI: 10.1109/MIC.2018.2890624.
- [79] G. Falazi, A. Lamparelli, U. Breitenbuecher, F. Daniel, and F. Leymann, *Unified Integration of Smart Contracts through Service Orientation*, 2020. DOI: 10.1109/MS.2020.2994040.
- [80] M. M. H. Onik and M. H. Miraz, “Performance Analytical Comparison of Blockchain-as-a-Service (BaaS) Platforms,” *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST*, vol. 285, pp. 3–18,

Bibliography

- 2019, ISSN: 18678211. DOI: 10.1007/978-3-030-23943-5_1. [Online]. Available: http://dx.doi.org/10.1007/978-3-030-23943-5_1.
- [81] A. Kernahan, U. Bernskov, and R. Beck, “Blockchain out of the Box – Where is the Blockchain in Blockchain-as-a-Service?” *Proceedings of the 54th Hawaii International Conference on System Sciences*, vol. 0, pp. 4281–4290, 2021. DOI: 10.24251/hicss.2021.520.
- [82] V. Yussupov, G. Falazi, U. Breitenbücher, and F. Leymann, “On the serverless nature of blockchains and smart contracts,” *arXiv*, 2020, ISSN: 23318422. arXiv: 2011.12729.
- [83] “Hyperledger Fabric - Hyperledger Foundation,” [Online]. Available: <https://www.hyperledger.org/use/fabric> (visited on Apr. 12, 2022).
- [84] “ConsenSys Quorum | ConsenSys,” [Online]. Available: <https://consensys.net/quorum/> (visited on Apr. 12, 2022).
- [85] X. Xu, I. Weber, and M. Staples, *Architecture for Blockchain Applications*. Springer, 2019, ISBN: 978-3-030-03034-6. DOI: 10.1007/978-3-030-03035-3.
- [86] D. R. Wong, S. Bhattacharya, and A. J. Butte, “Prototype of running clinical trials in an untrustworthy environment using blockchain,” *Nature Communications*, vol. 10, no. 1, pp. 1–8, 2019, ISSN: 20411723. DOI: 10.1038/s41467-019-08874-y.
- [87] B. S. Glicksberg, S. Burns, R. Currie, *et al.*, “Blockchain-authenticated sharing of genomic and clinical outcomes data of patients with cancer: A prospective cohort study,” *Journal of Medical Internet Research*, vol. 22, no. 3, e16810, 2020, ISSN: 14388871. DOI: 10.2196/16810.
- [88] S. Ahmed and N. Broek, “Blockchain could boost food security,” *Nature*, vol. 550, no. 7674, p. 43, 2017, ISSN: 14764687. DOI: 10.1038/550043e.
- [89] J. Mendling, I. Weber, W. Van Der Aalst, *et al.*, “Blockchains for business process management - Challenges and opportunities,” *ACM Transactions on Management Information Systems*, vol. 9, no. 1, 2018, ISSN: 21586578. DOI: 10.1145/3183367. arXiv: 1704.03610.
- [90] I. Weber, X. Xu, R. R. Riveret, G. Governatori, A. Ponomarev, and J. Mendling, “Untrusted business process monitoring and execution using blockchain,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9850 LNCS, Springer, 2016, pp. 329–347, ISBN: 978-3-319-45347-7. DOI: 10.1007/978-3-319-45348-4_19.
- [91] J. Eberhardt and S. Tai, “On or off the blockchain? Insights on off-chaining computation and data,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in*

- Artificial Intelligence and Lecture Notes in Bioinformatics*), vol. 10465 LNCS, 2017, pp. 3–15, ISBN: 9783319672618. DOI: 10.1007/978-3-319-67262-5_1.
- [92] J. Heiss, J. Eberhardt, and S. Tai, “From oracles to trustworthy data on-chaining systems,” in *Proceedings - 2019 2nd IEEE International Conference on Blockchain, Blockchain 2019*, 2019, pp. 496–503, ISBN: 9781728146935. DOI: 10.1109/Blockchain.2019.00075.
- [93] A. Beniiche, “A Study of Blockchain Oracles,” *arXiv preprint arXiv:2004.07140*, 2020. arXiv: 2004.07140. [Online]. Available: <http://arxiv.org/abs/2004.07140>.
- [94] “Orisi - Distributed Bitcoin Oracles,” [Online]. Available: <https://orisi.org/> (visited on Jun. 7, 2020).
- [95] “Provable - blockchain oracle service, enabling data-rich smart contracts,” [Online]. Available: <https://provable.xyz/> (visited on Jun. 7, 2020).
- [96] “axic/tinyoracle: Simple data provider toolkit for Ethereum,” [Online]. Available: <https://github.com/axic/tinyoracle> (visited on Jun. 7, 2020).
- [97] “Blockchain Oracles for Hybrid Smart Contracts | Chainlink,” [Online]. Available: <https://chain.link/> (visited on Jun. 7, 2020).
- [98] “Witnet: the decentralized oracle network,” [Online]. Available: <https://witnet.io/> (visited on Mar. 29, 2022).
- [99] A. S. de Pedro, D. Levi, and L. I. Cuende, “Witnet: A Decentralized Oracle Network Protocol,” *CoRR*, vol. abs/1711.0, 2017. DOI: 10.13140/RG.2.2.28152.34560. arXiv: 1711.09756. [Online]. Available: <http://dx.doi.org/10.13140/RG.2.2.28152.34560>.
- [100] N. Neidhardt, C. Köhler, and M. Nüttgens, “Cloud service billing and service level agreement monitoring based on blockchain,” in *CEUR Workshop Proceedings*, ser. CEUR Workshop Proc. Vol. 2097, 2018, pp. 65–69.
- [101] X. Xu, C. Pautasso, L. Zhu, *et al.*, “The blockchain as a software connector,” in *Proceedings - 2016 13th Working IEEE/IFIP Conference on Software Architecture, WICSA 2016*, IEEE Computer Society, 2016, pp. 182–191, ISBN: 9781509021314. DOI: 10.1109/WICSA.2016.21.
- [102] J. Adler, R. Berryhill, A. Veneris, Z. Poulos, N. Veira, and A. Kastania, “Astraea: A Decentralized Blockchain Oracle,” in *Proceedings - IEEE 2018 International Congress on Cybermatics: 2018 IEEE Conferences on Internet of Things, Green Computing and Communications, Cyber, Physical and Social Computing, Smart Data, Blockchain, Computer and Information Technology, iThings/Gree*, IEEE, 2018, pp. 1145–1152, ISBN: 9781538679753. DOI: 10.1109/Cybermatics_2018.2018.00207. arXiv: 1808.00528.

Bibliography

- [103] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, “Town crier: An authenticated data feed for smart contracts,” *Proceedings of the ACM Conference on Computer and Communications Security*, vol. 24-28-Octo, pp. 270–282, 2016, ISSN: 15437221. DOI: 10.1145/2976749.2978326.
- [104] ISO, “ISO/TR 23455:2019 Blockchain and distributed ledger technologies — Overview of and interactions between smart contracts in blockchain and distributed ledger technology systems,” ISO, Tech. Rep., 2019, p. 42. [Online]. Available: <https://doi.org/10.3403%2F30384568>.
- [105] C. Di Ciccio, A. Cecconi, M. Dumas, *et al.*, “Blockchain Support for Collaborative Business Processes,” *Informatik-Spektrum*, vol. 42, no. 3, pp. 182–190, 2019, ISSN: 1432122X. DOI: 10.1007/s00287-019-01178-x.
- [106] “ethereum/web3.py: A python interface for interacting with the Ethereum blockchain and ecosystem.” [Online]. Available: <https://github.com/ethereum/web3.py> (visited on Jun. 7, 2020).
- [107] “code-kotis/qr-code-scanner: A simple, fast and useful progressive web application,” [Online]. Available: <https://github.com/code-kotis/qr-code-scanner> (visited on Jun. 7, 2020).
- [108] “MacOS/blockchain-oracles-data-collection: Software Patterns for Blockchain Oracles,” [Online]. Available: <https://github.com/MacOS/blockchain-oracles-data-collection> (visited on Sep. 10, 2020).
- [109] I. Weber, V. Gramoli, A. Ponomarev, *et al.*, “On availability for blockchain-based systems,” in *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, vol. 2017-Septe, 2017, pp. 64–73, ISBN: 9781538616796. DOI: 10.1109/SRDS.2017.15.
- [110] Z. A. El Houda, A. Hafid, and L. Khoukhi, “Co-IoT: A collaborative DDoS mitigation scheme in IoT environment based on blockchain using SDN,” in *2019 IEEE Global Communications Conference, GLOBECOM 2019 - Proceedings*, 2019, pp. 1–6, ISBN: 9781728109626. DOI: 10.1109/GLOBECOM38437.2019.9013542.
- [111] O. Delgado-Mohatar, J. M. Sierra-Cámara, and E. Anguiano, “Blockchain-based semi-autonomous ransomware,” *Future Generation Computer Systems*, 2020, ISSN: 0167-739X.
- [112] S. Krejci, M. Sigwart, and S. Schulte, “Blockchain- and IPFS-Based Data Distribution for the Internet of Things,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 12054 LNCS, 2020, pp. 177–191, ISBN: 9783030447687. DOI: 10.1007/978-3-030-44769-4_14.

- [113] L. García-Bañuelos, A. Ponomarev, M. Dumas, and I. Weber, “Optimized execution of business processes on blockchain,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, ser. Lecture Notes in Computer Science, Springer, vol. 10445 LNCS, Springer, 2017, pp. 130–146, ISBN: 9783319649993. DOI: 10.1007/978-3-319-65000-58. arXiv: 1612.03152.
- [114] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of*, 1994.
- [115] V. Garousi, M. Felderer, and M. V. Mäntylä, “The need for multivocal literature reviews in software engineering,” in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering - EASE '16*, ACM, 2016, pp. 1–6, ISBN: 9781450336918. DOI: 10.1145/2915970.2916008. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=2915970.2916008>.
- [116] K.-J. Stol, P. Ralph, and B. Fitzgerald, “Grounded Theory in Software Engineering Research : A Critical Review and Guidelines,” in *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, IEEE, 2015, pp. 120–131, ISBN: 9781450339001. DOI: <http://dx.doi.org/10.1145/2884781.2884833>.
- [117] M. Alharby and A. van Moorsel, “Blockchain-based Smart Contracts: A Systematic Mapping Study,” *Computer Science and Information Technology*, pp. 125–140, 2017.
- [118] J. Bontje. “DApp Design Patterns.” (2015), [Online]. Available: <https://www.slideshare.net/mids106/dapp-design-patterns>.
- [119] Cjgdev. “Smart-Contract Patterns written in Solidity, collated for community good.” (2016), [Online]. Available: <https://github.com/cjgdev/smart-contract-patterns>.
- [120] X. Xu, I. Weber, M. Staples, *et al.*, “A Taxonomy of Blockchain-Based Systems for Architecture Design,” *Proceedings - 2017 IEEE International Conference on Software Architecture, ICSA 2017*, pp. 243–252, 2017. DOI: 10.1109/ICSA.2017.33. [Online]. Available: <http://ieeexplore.ieee.org/abstract/document/7930224/>.
- [121] M. Bartoletti and L. Pompianu, “An Empirical analysis of smart contracts: Platforms, applications, and design patterns,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10323 LNCS, pp. 494–509, 2017, ISSN: 16113349. DOI: 10.1007/978-3-319-70278-0_31. arXiv: 1703.06322. [Online]. Available: <http://arxiv.org/abs/1703.06322>.
- [122] P. Zhang, J. White, D. C. Schmidt, and G. Lenz, “Applying Software Patterns to Address Interoperability in Blockchain-based Healthcare Apps,” *arXiv preprint arXiv:1706.03700*,

Bibliography

- 2017, ISSN: 2331-8422. arXiv: 1706.03700. [Online]. Available: <http://arxiv.org/abs/1706.03700>.
- [123] A. Mavridou and A. Laszka, “Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach,” *arXiv preprint arXiv:1711.09327*, 2017. arXiv: 1711.09327. [Online]. Available: <http://arxiv.org/abs/1711.09327>.
- [124] C. Wohlin, “Guidelines for snowballing in systematic literature studies and a replication in software engineering,” in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering - EASE '14*, ser. EASE '14, New York, NY, USA: ACM, 2014, pp. 1–10, ISBN: 9781450324762. DOI: 10.1145/2601248.2601268. arXiv: arXiv:1011.1669v3. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=2601248.2601268>.
- [125] ConsenSys. “Ethereum Contract Security Techniques and Tips.” (2017), [Online]. Available: <https://github.com/ConsenSys/smart-contract-best-practices>.
- [126] “Ethereum Development and DApps,” [Online]. Available: <https://www.reddit.com/r/ethdev/>.
- [127] “Ethereum Stack Exchange,” [Online]. Available: <https://ethereum.stackexchange.com/questions/20781/at-which-point-the-smart-contracts-get-executed>.
- [128] F. HENGLEIN. “Smart Contracts Are Neither Smart Nor Contracts.” (2017), [Online]. Available: <https://www.infoq.com/presentations/blockchain-introduction>.
- [129] OpenZeppelin. “OpenZeppelin/zeppelin-solidity: OpenZeppelin, a framework to build secure smart contracts on Ethereum,” [Online]. Available: <https://github.com/OpenZeppelin/zeppelin-solidity> (visited on Dec. 5, 2017).
- [130] Modular-Network. “Modular-Network/ethereum-libraries: Library contracts for Ethereum,” [Online]. Available: <https://github.com/Modular-Network/ethereum-libraries>.
- [131] “maxwoe/solidity_patterns: Smart Contracts Design Patterns in the Ethereum Ecosystem and Solidity Code,” [Online]. Available: https://github.com/maxwoe/solidity_patterns.
- [132] Oraclize. “Blockchain Oracle Service, Enabling Data-Rich Smart Contracts.” (2017), [Online]. Available: <http://www.oraclize.it/>.
- [133] H. E. Willis, “Restatement of the Law of Contracts of the American Law Institute,” *Ind. LJ*, vol. 7, p. 429, 1931.

- [134] N. Szabo. “Smart Contracts.” (1994), [Online]. Available: <http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/L0Twinterschool2006/szabo.best.vwh.net/smart.contracts.html> (visited on Sep. 10, 2019).
- [135] N. Szabo. “The idea of smart contracts.” (1997), [Online]. Available: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/L0Twinterschool2006/szabo.best.vwh.net/smart_contracts_idea.html (visited on Sep. 10, 2019).
- [136] M. K. Woebeking, “The impact of smart contracts on traditional concepts of contract law,” *J. Intell. Prop. Info. Tech. & Elec. Com. L.*, vol. 10, no. 1988, p. 105, 2019.
- [137] M. Fowler and R. Parsons, *Domain-Specific Languages*. Pearson Education, 2010, ISBN: 9780321712943. DOI: 10.1007/978-3-642-03034-5.
- [138] L. Johnson, “Effective Contract Drafting: Identifying the Building Blocks of Contracts,” *Scholarly Works*, Jul. 2013. [Online]. Available: <https://scholars.law.unlv.edu/facpub/882>.
- [139] R. J. Wieringa, *Design science methodology: For information systems and software engineering*. Springer, 2014, ISBN: 9783662438398. DOI: 10.1007/978-3-662-43839-8.
- [140] “Xtext framework,” [Online]. Available: <https://www.eclipse.org/Xtext/> (visited on Jul. 10, 2019).
- [141] “maxwoe/cml: Contract Modeling Language,” [Online]. Available: <https://github.com/maxwoe/cml>.
- [142] “CML Web Editor,” [Online]. Available: <http://cml.swa.univie.ac.at/>.
- [143] O. Marjanovic and Z. Milosevic, “Towards formal modeling of e-contracts,” *Proceedings - 5th IEEE International Enterprise Distributed Object Computing Conference*, vol. 2001-Janua, no. January, pp. 59–68, 2001, ISSN: 15417719. DOI: 10.1109/EDOC.2001.950423. [Online]. Available: <http://ieeexplore.ieee.org/document/950423/>.
- [144] J. De Kruijff and H. Weigand, “Ontologies for commitment-based smart contracts,” in *OTM Confederated International Conferences "On the Move to Meaningful Internet Systems"*, Springer, 2017, pp. 383–398.
- [145] D. McAdams, “An Ontology for Smart Contracts,” *IOHK Paper*, p. 3, 2017. [Online]. Available: <https://iohk.io/research/library/#QCNr6SCZ>.
- [146] G. J. Pace and G. Schneider, “Challenges in the specification of full contracts,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5423 LNCS, Springer, Berlin, Heidelberg, 2009,

Bibliography

- pp. 292–306, ISBN: 3642002544. DOI: 10.1007/978-3-642-00255-7_20. [Online]. Available: http://link.springer.com/10.1007/978-3-642-00255-7_20.
- [147] C. K. Frantz and M. Nowostawski, “From institutions to code: Towards automated generation of smart contracts,” *Proceedings - IEEE 1st International Workshops on Foundations and Applications of Self-Systems, FAS-W 2016*, pp. 210–215, 2016, ISSN: 978-3-319-42446-0. DOI: 10.1109/FAS-W.2016.53. [Online]. Available: <https://ieeexplore.ieee.org/document/7789470>.
- [148] X. He, B. Qin, Y. Zhu, X. Chen, and Y. Liu, “SPESC: A Specification Language for Smart Contracts,” *Proceedings - International Computer Software and Applications Conference*, vol. 1, pp. 132–137, Jul. 2018, ISSN: 07303157. DOI: 10.1109/COMPSAC.2018.00025. [Online]. Available: <https://ieeexplore.ieee.org/document/8377649/>.
- [149] E. Regnath and S. Steinhorst, “SmaCoNat: Smart Contracts in Natural Language,” *Forum on Specification and Design Languages*, vol. 2018-Sept, 2018, ISSN: 16369874. DOI: 10.1109/FDL.2018.8524068. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/8524068>.
- [150] C. Prisacariu and G. Schneider, “A Formal Language for Electronic Contracts,” *Lecture Notes in Computer Science*, vol. 4468, pp. 174–189, 2007, ISSN: 03029743. DOI: 10.1007/978-3-540-72952-5_11. [Online]. Available: <http://www.springerlink.com/index/10.1007/978-3-540-72952-5>.
- [151] E. Martínez, G. Díaz, E. Cambronero, and G. Schneider, “A model for visual specification of e-contracts,” *Proceedings - 2010 IEEE 7th International Conference on Services Computing, SCC 2010*, vol. 8625 LNAI, no. section 3, pp. 1–8, Jul. 2010, ISSN: 16113349. DOI: 10.1109/SCC.2010.32. arXiv: 1406.5691. [Online]. Available: <https://folk.uio.no/gerardo/scc2010.pdf>.
- [152] “Xtend - Modernized Java,” [Online]. Available: <https://www.eclipse.org/xtend/> (visited on Mar. 30, 2022).
- [153] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1996, p. 395, ISBN: 020163361-2. DOI: 10.1093/carcin/bgs084.
- [154] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Chichester, UK: Wiley, 2000, ISBN: 978-0-471-60695-6. [Online]. Available: <https://www.safaribooksonline.com/library/view/pattern-oriented-software-architecture/9781118725177/>.

- [155] “Solidity 0.5.13 documentation,” [Online]. Available: <https://solidity.readthedocs.io/en/v0.5.13/> (visited on Nov. 25, 2019).
- [156] P. M. Senge, *The fifth discipline: The art and practice of the learning organization*. Broadway Business, 2006.
- [157] “GitHub - federicobond/solidity-parser-antlr: A Solidity parser for JS built on top of a robust ANTLR4 grammar,” [Online]. Available: <https://github.com/federicobond/solidity-parser-antlr> (visited on Feb. 20, 2020).
- [158] T. Hvitved, “Contract Formalisation and Modular Implementation of Domain-Specific Languages,” Ph.D. dissertation, Citeseer, 2011. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.724.7779&rep=rep1&type=pdf>.
- [159] “Known Attacks - Ethereum Smart Contract Best Practices,” [Online]. Available: https://consensys.github.io/smart-contract-best-practices/known_attacks/ (visited on Dec. 3, 2019).
- [160] “Accord Project,” [Online]. Available: <https://www.accordproject.org/> (visited on Feb. 20, 2020).
- [161] “Ergo - Accord Project,” [Online]. Available: <https://www.accordproject.org/projects/ergo/> (visited on Feb. 20, 2020).
- [162] W. de Kort, “What Is DevOps?” *DevOps on the Microsoft Stack*, pp. 3–8, 2016. DOI: 10.1007/978-1-4842-1446-6_1.
- [163] C. Lal and D. Marijan, “Blockchain Testing: Challenges, Techniques, and Research Directions,” *arXiv preprint arXiv:2103.10074*, 2021. arXiv: 2103.10074. [Online]. Available: <http://arxiv.org/abs/2103.10074>.
- [164] R. Koul, “Blockchain Oriented Software Testing - Challenges and Approaches,” *2018 3rd International Conference for Convergence in Technology, I2CT 2018*, pp. 1–6, 2018. DOI: 10.1109/I2CT.2018.8529728.
- [165] S. Li, Q. Xu, P. Hou, *et al.*, “Exploring the Challenges of Developing and Operating Consortium Blockchains: A Case Study,” *ACM International Conference Proceeding Series*, pp. 398–404, 2020. DOI: 10.1145/3383219.3383276.
- [166] M. Yilmaz, S. Tasel, E. Tuzun, U. Gulec, R. V. O’Connor, and P. M. Clarke, *Applying Blockchain to Improve the Integrity of the Software Development Process*. Springer International Publishing, 2019, vol. 1060, pp. 260–271, ISBN: 9783030280048. DOI: 10.1007/978-3-030-28005-5_20. [Online]. Available: http://dx.doi.org/10.1007/978-3-030-28005-5_20.

Bibliography

- [167] M. Beller and J. Hejderup, “Blockchain-based software engineering,” *Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results, ICSE-NIER 2019*, pp. 53–56, 2019. DOI: 10 . 1109 / ICSE - NIER . 2019 . 00022.
- [168] “Truffle Suite - Truffle Suite,” [Online]. Available: <https://trufflesuite.com/> (visited on Mar. 9, 2022).
- [169] “Hardhat | Ethereum development environment for professionals by Nomic Foundation,” [Online]. Available: <https://hardhat.org/> (visited on Mar. 9, 2022).
- [170] “maxwoe/sc-devops-knowledge-sources: DevOps for Ethereum Blockchain Smart Contracts - Knowledge Sources,” [Online]. Available: <https://github.com/maxwoe/sc-devops-knowledge-sources> (visited on Sep. 2, 2021).
- [171] “Jenkins,” [Online]. Available: <https://www.jenkins.io/> (visited on Mar. 13, 2022).
- [172] “Travis CI - Test and Deploy Your Code with Confidence,” [Online]. Available: <https://travis-ci.org/> (visited on Mar. 13, 2022).
- [173] “Continuous Integration and Delivery - CircleCI,” [Online]. Available: <https://circleci.com/> (visited on Mar. 13, 2022).
- [174] “GitLab CI/CD | GitLab,” [Online]. Available: <https://docs.gitlab.com/ee/ci/> (visited on Mar. 13, 2022).
- [175] “GitHub Actions Documentation - GitHub Docs,” [Online]. Available: <https://docs.github.com/en/actions> (visited on Mar. 13, 2022).
- [176] “Embark into the Ether. | Embark,” [Online]. Available: <https://framework.embarklabs.io/> (visited on Mar. 9, 2022).
- [177] “eth-brownie/brownie: A Python-based development and testing framework for smart contracts targeting the Ethereum Virtual Machine.,” [Online]. Available: <https://github.com/eth-brownie/brownie> (visited on Mar. 9, 2022).
- [178] “Waffle,” [Online]. Available: <https://getwaffle.io/> (visited on Mar. 9, 2022).
- [179] “Remix - Ethereum IDE,” [Online]. Available: <https://remix.ethereum.org/> (visited on Mar. 9, 2022).
- [180] “ethereum/remix-vscode: Remix VS Code extension,” [Online]. Available: <https://github.com/ethereum/remix-vscode> (visited on Mar. 9, 2022).
- [181] “ERC-20 Token Standard | ethereum.org,” [Online]. Available: <https://ethereum.org/en/developers/docs/standards/tokens/erc-20/> (visited on Mar. 9, 2022).

- [182] “NatSpec Format - Solidity 0.8.12 documentation,” [Online]. Available: <https://docs.soliditylang.org/en/v0.8.12/natspec-format.html> (visited on Mar. 9, 2022).
- [183] “Doxygen: Doxygen,” [Online]. Available: <https://www.doxygen.nl/index.html> (visited on Mar. 9, 2022).
- [184] “Main - Emscripten 3.1.6-git (dev) documentation,” [Online]. Available: <https://emscripten.org/> (visited on Mar. 9, 2022).
- [185] “ethereum/solc-bin: This repository contains current and historical builds of the Solidity Compiler,” [Online]. Available: <https://github.com/ethereum/solc-bin> (visited on Mar. 9, 2022).
- [186] “ethereum/solc-js: Javascript bindings for the Solidity compiler,” [Online]. Available: <https://github.com/ethereum/solc-js> (visited on Mar. 9, 2022).
- [187] “Solc-js - npm,” [Online]. Available: <https://www.npmjs.com/package/solc-js?activeTab=readme> (visited on Mar. 9, 2022).
- [188] “merklejerk/solpp: A solidity preprocessor and flattener CLI and library,” [Online]. Available: <https://github.com/merklejerk/solpp> (visited on Mar. 9, 2022).
- [189] “trufflesuite/ganache: A tool for creating a local blockchain for fast Ethereum development,” [Online]. Available: <https://github.com/trufflesuite/ganache> (visited on Mar. 9, 2022).
- [190] “Oxcert - Testing smart contracts live without spending gas,” [Online]. Available: <https://0xcert.org/news/live-testing-smart-contracts-with-estimategas-william-entriken-tadej-vengust/> (visited on Aug. 2, 2021).
- [191] P. Hamill, *Unit Test Frameworks: Tools for High-Quality Software Development*, 2004.
- [192] P. Chakraborty, R. Shahriyar, A. Iqbal, and A. Bosu, “Understanding the software development practices of blockchain projects: A survey,” *International Symposium on Empirical Software Engineering and Measurement*, 2018, ISSN: 19493789. DOI: 10.1145/3239235.3240298.
- [193] “rkalis/truffle-assertions: Assertions and utilities for testing Ethereum smart contracts with Truffle unit tests,” [Online]. Available: <https://github.com/rkalis/truffle-assertions> (visited on Mar. 9, 2022).
- [194] “Test Helpers - OpenZeppelin Docs,” [Online]. Available: <https://docs.openzeppelin.com/test-helpers/0.5/> (visited on Mar. 9, 2022).
- [195] “OpenZeppelin/openzeppelin-contracts,” [Online]. Available: <https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contracts/mocks> (visited on Mar. 9, 2022).

Bibliography

- on Aug. 16, 2021).
- [196] “ChainSafe/web3.js: Ethereum JavaScript API,” [Online]. Available: <https://github.com/ChainSafe/web3.js> (visited on Mar. 21, 2022).
 - [197] “Mocha - the fun, simple, flexible JavaScript test framework,” [Online]. Available: <https://mochajs.org/> (visited on Mar. 9, 2022).
 - [198] “Chai,” [Online]. Available: <https://www.chaijs.com/> (visited on Mar. 9, 2022).
 - [199] “gnosis/mock-contract: Simple Solidity contract to mock dependent contracts in truffle tests.,” [Online]. Available: <https://github.com/gnosis/mock-contract> (visited on Mar. 9, 2022).
 - [200] “MythX: Smart contract security service for Ethereum,” [Online]. Available: <https://mythx.io/> (visited on Mar. 9, 2022).
 - [201] “eth-sri/securify2: Securify v2.0,” [Online]. Available: <https://github.com/eth-sri/securify2> (visited on Mar. 9, 2022).
 - [202] “smartdec/smartcheck: SmartCheck – a static analysis tool that detects vulnerabilities and bugs in Solidity programs (Ethereum-based smart contracts).,” [Online]. Available: <https://github.com/smartdec/smartcheck> (visited on Mar. 9, 2022).
 - [203] “crytic/slither: Static Analyzer for Solidity,” [Online]. Available: <https://github.com/crytic/slither> (visited on Mar. 9, 2022).
 - [204] “trailofbits/manticore: Symbolic execution tool,” [Online]. Available: <https://github.com/trailofbits/manticore> (visited on Mar. 9, 2022).
 - [205] “ConsenSys/mythril: Security analysis tool for EVM bytecode. Supports smart contracts built for Ethereum, Hedera, Quorum, VeChain, Roostock, Tron and other EVM-compatible blockchains.,” [Online]. Available: <https://github.com/ConsenSys/mythril> (visited on Mar. 9, 2022).
 - [206] M. Di Angelo and G. Salzer, “A survey of tools for analyzing ethereum smart contracts,” *Proceedings - 2019 IEEE International Conference on Decentralized Applications and Infrastructures, DAPPCON 2019*, pp. 69–78, 2019. DOI: 10.1109/DAPPCON.2019.00018.
 - [207] S. Sayeed, H. Marco-Gisbert, and T. Caira, “Smart Contract: Attacks and Protections,” *IEEE Access*, vol. 8, pp. 24 416–24 427, 2020, ISSN: 21693536. DOI: 10.1109/ACCESS.2020.2970495.
 - [208] A. Ayman, S. Roy, A. Alipour, and A. Laszka, “Smart contract development from the perspective of developers: Topics and issues discussed on social media,” *Lecture Notes in*

- Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 12063 LNCS, pp. 405–422, 2020, ISSN: 16113349. DOI: 10.1007/978-3-030-54455-3_29. arXiv: 1905.08833.
- [209] “crytic/echidna: Ethereum smart contract fuzzer,” [Online]. Available: <https://github.com/crytic/echidna> (visited on Mar. 9, 2022).
- [210] J. J. Honig, M. H. Everts, and M. Huisman, “Practical Mutation Testing for Smart Contracts,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11737 LNCS, pp. 289–303, 2019, ISSN: 16113349. DOI: 10.1007/978-3-030-31500-9_19.
- [211] “Solidity-coverage/faq.md at master - sc-forks/solidity-coverage,” [Online]. Available: <https://github.com/sc-forks/solidity-coverage/blob/master/docs/faq.md> (visited on Mar. 9, 2022).
- [212] “OxProject/tools,” [Online]. Available: <https://github.com/OxProject/tools> (visited on Mar. 9, 2022).
- [213] “cgewecke/eth-gas-reporter: Gas usage per unit test. Average gas usage per method. A mocha reporter.” [Online]. Available: <https://github.com/cgewecke/eth-gas-reporter> (visited on Mar. 9, 2022).
- [214] “Blockchain Technology Solutions | Ethereum Solutions | ConsenSys,” [Online]. Available: <https://consensys.net/> (visited on Mar. 9, 2022).
- [215] “Terraform by HashiCorp,” [Online]. Available: <https://www.terraform.io/> (visited on Mar. 9, 2022).
- [216] “Ansible is Simple IT Automation,” [Online]. Available: <https://www.ansible.com/> (visited on Mar. 9, 2022).
- [217] “Powerful infrastructure automation and delivery | Puppet,” [Online]. Available: <https://puppet.com/> (visited on Mar. 9, 2022).
- [218] “Configuration Management System Software - Chef Infra | Chef,” [Online]. Available: <https://www.chef.io/products/chef-infra> (visited on Mar. 9, 2022).
- [219] “hyperledger-labs/blockchain-automation-framework: An automation framework for rapidly and consistently deploying production-ready DLT platforms,” [Online]. Available: <https://github.com/hyperledger-labs/blockchain-automation-framework> (visited on Sep. 1, 2021).
- [220] “Running Migrations - Truffle Suite,” [Online]. Available: <https://trufflesuite.com/docs/truffle/getting-started/running-migrations.html> (visited on

Bibliography

- Mar. 9, 2022).
- [221] “NomicFoundation/hardhat-ignition: Hardhat Ignition is a solidity tool for building, deploying, versioning and changing Ethereum smart contract infrastructure.” [Online]. Available: <https://github.com/NomicFoundation/hardhat-ignition> (visited on Mar. 9, 2022).
- [222] “Proxy Patterns - OpenZeppelin blog,” [Online]. Available: <https://blog.openzeppelin.com/proxy-patterns/> (visited on Sep. 1, 2021).
- [223] P. Klinger, L. Nguyen, and F. Bodendorf, *Upgradeability Concept for Collaborative Blockchain-Based Business Process Execution Framework*. Springer International Publishing, 2020, vol. 12404 LNCS, pp. 127–141, ISBN: 9783030596378. DOI: 10.1007/978-3-030-59638-5_9. [Online]. Available: http://dx.doi.org/10.1007/978-3-030-59638-5_9.
- [224] “EIP-2535: Diamonds, Multi-Facet Proxy,” [Online]. Available: <https://eips.ethereum.org/EIPS/eip-2535> (visited on Mar. 9, 2022).
- [225] “Upgrades Plugins - OpenZeppelin Docs,” [Online]. Available: <https://docs.openzeppelin.com/upgrades-plugins/1.x/> (visited on Mar. 30, 2022).
- [226] “Verify and Publish Contract Source Code | Etherscan,” [Online]. Available: <https://etherscan.io/verifyContract> (visited on Mar. 9, 2022).
- [227] “Sourcify,” [Online]. Available: <https://sourcify.dev/> (visited on Mar. 9, 2022).
- [228] “rkalis/truffle-plugin-verify: Verify your smart contracts on Etherscan from the Truffle CLI,” [Online]. Available: <https://github.com/rkalis/truffle-plugin-verify> (visited on Mar. 9, 2022).
- [229] “hardhat/packages/hardhat-etherscan at master NomicFoundation/hardhat,” [Online]. Available: <https://github.com/NomicFoundation/hardhat/tree/master/packages/hardhat-etherscan> (visited on Mar. 9, 2022).
- [230] “Ethereum (ETH) Blockchain Explorer,” [Online]. Available: <https://etherscan.io/> (visited on Mar. 9, 2022).
- [231] “Ethereum (ETH) Blockchain Explorer - etherchain.org - 2022,” [Online]. Available: <https://etherchain.org/> (visited on Mar. 9, 2022).
- [232] “Blockchair - Universal blockchain explorer and search engine,” [Online]. Available: <https://blockchair.com/> (visited on Mar. 9, 2022).
- [233] G. A. Pierro, R. Tonelli, and M. Marchesi, “An organized repository of ethereum smart contracts’ source codes and metrics,” *Future Internet*, vol. 12, no. 11, pp. 1–15, 2020, ISSN:

19995903. DOI: 10.3390/fi12110197.
- [234] “blockscout/blockscout: Blockchain explorer for Ethereum based network and a tool for inspecting and analyzing EVM based blockchains.” [Online]. Available: <https://github.com/blockscout/blockscout> (visited on Mar. 9, 2022).
- [235] “xops/expedition: A block explorer for the Ethereum stack.” [Online]. Available: <https://github.com/xops/expedition> (visited on Mar. 9, 2022).
- [236] “tryethernal/ethernal: Ethernal is a block explorer for EVM-based chains,” [Online]. Available: <https://github.com/tryethernal/ethernal> (visited on Mar. 9, 2022).
- [237] “Tenderly | Ethereum Developer Platform,” [Online]. Available: <https://tenderly.co/> (visited on Mar. 9, 2022).
- [238] “Sentinel - OpenZeppelin Docs,” [Online]. Available: <https://docs.openzeppelin.com/defender/sentinel> (visited on Mar. 9, 2022).
- [239] “PARSIQ,” [Online]. Available: <https://www.parsiq.net/en/> (visited on Mar. 9, 2022).
- [240] “Prometheus - Monitoring system and time series database,” [Online]. Available: <https://prometheus.io/> (visited on Mar. 9, 2022).
- [241] “google/cadvisor: Analyzes resource usage and performance characteristics of running containers.” [Online]. Available: <https://github.com/google/cadvisor> (visited on Mar. 9, 2022).
- [242] “maxwoe/sc-devops: DevOps for Ethereum Blockchain Smart Contracts,” [Online]. Available: <https://github.com/maxwoe/sc-devops> (visited on Aug. 30, 2021).
- [243] J. A. Maxwell, *Qualitative research design: An interactive approach*. Sage publications, 2012.
- [244] Y. S. Lincoln and E. G. Guba, *Naturalistic inquiry*. sage, 1985.

Acronyms

ABI Application Binary Interface.

API Application Programming Interface.

AST Abstract Syntax Tree.

BaaS Blockchain as a Service.

BLOB Binary Large Object.

BOSE Blockchain-Oriented Software Engineering.

BPMN Business Process Model and Notation.

CAS Content Adressable Storage.

CD Continuous Delivery/Deployment.

CDN Content Delivery Network.

CI Continuous Integration.

CLI Command Line Interface.

CML Contract Modeling Language.

CPU Central Processing Unit.

CQRS Command-Query Responsibility Segregation.

CRUD Create, Read, Update and Delete.

DApp Decentralized Application.

DSL Domain Specific Language.

EBNF Extended Backus-Naur Form-like.

Acronyms

EDA Event Driven Architecture.

ENS Ethereum Name Service.

ERP Enterprise Resource Planning.

EVM Ethereum Virtual Machine.

FaaS Function as a Service.

FIFO First In - First Out.

FIFO First In - Last Out.

FSM Finite-State Machine.

GT Grounded Theory.

HE Homomorphic Encryption.

HTTP Hypertext Transfer Protocol.

HTTPS Hypertext Transfer Protocol Secure.

IaaS Infrastructure as a Service.

IaC Infrastructure as Code.

IDE Integrated Development Environment.

IPFS InterPlanetary File System.

JSON JavaScript Object Notation.

LLOC Logical Lines Of Code.

MLR Multivocal Literature Review.

NoSQL Not only Structured Query Language.

P2P Peer-to-Peer.

PaaS Platform as a Service.

PoA Proof-of-Authority.

PoW Proof-of-Work.

QR Quick Response.

REST Representational State Transfer.

RPC Remote Procedure Call.

SaaS Software as a Service.

SDK Software Development Kit.

SGX Software Guard Extensions.

SLR Systematic Literature Review.

SMT Satisfiability Modulo Theories.

SQL Structured Query Language.

TEE Trusted Execution Environment.

UI User Interface.

VCS Version Control System.

XaaS Everything as a Service.

ZKP Zero-knowledge proof.

A. Appendix

A.1. DSL

for Smart Contracts - Contract Modeling Language (CML)

The CML language we introduced in Chapter 6 is developed in Xtext [140], a framework for developing programming languages and DSLs. In this section, we give a brief introduction to Xtext and then present the developed Xtext grammar.

Xtext DSL Framework

The CML language is developed in Xtext [140]. Xtext is a framework for the development of programming languages and domain-specific languages (DSLs). Xtext uses a specifically designed grammar language to define custom languages and provides a fully corresponding infrastructure (parser, linker, typechecker, compiler) as well as editing support for Eclipse. Figure A.1 shows an overview of the Xtext framework, which can be separated in a meta-level layer and an instance-level layer. The meta-level layer is used for developing the general infrastructure, defines the syntax of the DSL and the transformations (to code), whereas the instance-level layer is used to develop a corresponding implementation. The syntax (grammar) of the DSL is defined in an Xtext (.xtext) file by using Extended Backus-Naur Form-like (EBNF) expressions from which a meta model in Ecore (.ecore) and a class model for the abstract syntax tree is derived. Often the grammar is referred to as concrete syntax, whereas the Ecore model is referred to as abstract syntax. To utilize the DSL, a textual DSL instance (.mydsl) is created that conforms to the concrete syntax. This instance can likewise be represented as an instance of the Ecore model (.xmi). Model transformations are typically written using the Xtend language (.xtend), which is a flexible and expressive dialect of the Java programming language to allow the conversion of models to code, visualizations, or other representations (*.*)).

Xtext Grammar of the Smart Contract DSL

Listing A.1 shows the complete Xtext grammar of our smart contract abstraction DSL. The full CML language implementation source code is available on GitHub [141].

A. Appendix

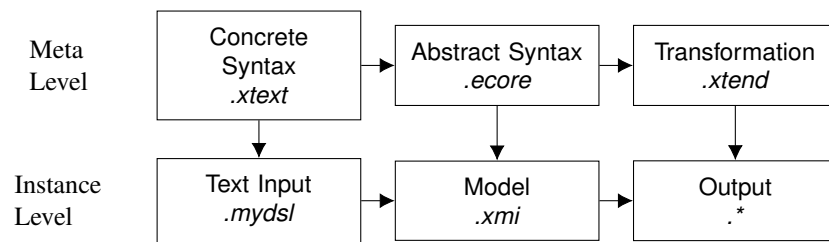


Figure A.1.: An overview of the Xtext DSL Framework.

```

Xtext
grammar at.ac.univie.swa.Cml with org.eclipse.xtext.common.Terminals
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
generate cml 'http://swa.univie.ac.at/Cml'

CmlProgram:
  ('namespace' name=QualifiedName)?
  imports+=Import*
  declarations+=Declaration*;

Import:
  'import' importedNamespace=QualifiedNameWithWildcard;

Declaration:
  CmlClass | Party | Asset | Event | Contract | Enumeration | Attribute | Operation |
  AnnotationDeclaration;

CmlClass:
  isAbstract? kind='concept' name=ID typeVariables? extends? identifiedBy?
  (BEGIN features+=FeatureAttributeAndOperation* END)?;

Contract returns CmlClass:
  annotations+=Annotation*
  kind='contract' name=ID extends?
  (BEGIN features+=FeatureAttributeAndOperationAndClause* END)?;

Party returns CmlClass:
  external?='external'? kind='party' name=ID extends? identifiedBy?
  (BEGIN features+=FeatureAttribute+ END)?;

Asset returns CmlClass:
  kind='asset' name=ID extends? identifiedBy?
  (BEGIN features+=FeatureAttribute+ END)?;

Event returns CmlClass:
  kind='event' name=ID
  (BEGIN features+=FeatureAttribute+ END)?;

Enumeration returns CmlClass:
  kind='enum' name=ID
  BEGIN features+=FeatureEnumeration+ END;

EnumerationElement:
  name=ID;

AnnotationDeclaration:
  '@' 'interface' name=ID
  (BEGIN features+=Attribute+ END)?;

Annotation:
  '@' declaration=[AnnotationDeclaration
  |QualifiedName] ('(' (args+=(AnnotationElement) (',' args+=AnnotationElement)*)?
  ')')?;

AnnotationElement:
  param=[Attribute] '=' value=Expression;

Type:
  CmlClass;

TypeReference:

```

A.1. DSL for Smart Contracts - Contract Modeling Language (CML)

```

ParameterizedTypeReference => ({GenericArrayTypeReference.componentType=current} '[' ''])*;
ParameterizedTypeReference:
  type=[DeclaredType
    |QualifiedName] => '<' typeArgs+=TypeReference (',' typeArgs+=TypeReference)* '>'?;
DeclaredType:
  TypeTypeRef | TypeVarRef;
TypeVarRef returns ParameterizedTypeReference:
  type=[TypeVariable];
TypeTypeRef returns ParameterizedTypeReference:
  type=[Type|QualifiedName];
TypeVariable:
  name=ID;
NamedElement:
  CmlProgram | CmlClass | Feature | VariableDeclaration | AnnotationDeclaration | TypeVariable;
Feature:
  Attribute | Operation | Clause | EnumerationElement;
FeatureAttribute returns Feature:
  Attribute;
FeatureAttributeAndOperation returns Feature:
  Attribute | Operation;
FeatureAttributeAndOperationAndClause returns Feature:
  FeatureAttributeAndOperation | Clause;
FeatureEnumeration returns Feature:
  EnumerationElement;
Attribute:
  (constant?='constant')? type=TypeReference name=ID ('=' expression=Expression)?;
Operation:
  ('def' | 'action') (type=TypeReference)? name=ID
  '(' (params+=Attribute (',' params+=Attribute)*)? ')'
  (body=Block)?;
Clause:
  'clause' name=ID
  BEGIN constraint=Constraint actor=Actor action=DeonticAction END;
Constraint:
  {Constraint} temporal=TemporalConstraint? general=GeneralConstraint?;
TemporalConstraint:
  'due' timeframe=Timeframe
  ? period=PeriodicTime? precedence=TemporalPrecedence reference=(TimeQuery | Expression);
Timeframe:
  'within' window=Expression;
GeneralConstraint:
  'given' expression=Expression;
Actor:
  'party' party=Expression;
enum Deontic:
  MUST='must' | MAY='may' /* | MUST_NOT='must_not'*/;
DeonticAction:
  deontic=Deontic compoundAction=CompoundAction;
CompoundAction:
  XorCompoundAction;
XorCompoundAction returns CompoundAction:
  OrCompoundAction => ({XorCompoundAction.left=current} op='xor') right=OrCompoundAction)*;
OrCompoundAction returns CompoundAction:
  AndCompoundAction ({OrCompoundAction.left=current} op=('or') right=AndCompoundAction)*;
AndCompoundAction returns CompoundAction:

```

A. Appendix

```
PrimaryCompoundAction
  ({AndCompoundAction.left=current} op=('and') right=PrimaryCompoundAction)*;

PrimaryCompoundAction returns CompoundAction:
  {AtomicAction} operation=[Operation] |
  {NestedCompoundAction} '(' child=CompoundAction ')';

PeriodicTime:
  'every' period=
    Expression 'from' start=(TimeQuery | Expression) ('until' end=(TimeQuery | Expression))?;

enum TemporalPrecedence:
  BEFORE='before' |
  /*BEFORE_OR_EQUALS='onOrBefore' |
  EQUALS='on' |
  AFTER_OR_EQUALS='onOrAfter' |*/
  AFTER='after';

ClauseQuery:
  'clause' clause=[Clause] status=ClauseStatus;

enum ClauseStatus:
  FULFILLED='fulfilled' |
  FAILED='failed';

EventQuery:
  'event' event=[Attribute];

ActionQuery:
  party=[Attribute] query='did' action=[Operation];

TimeQuery:
  ClauseQuery | EventQuery | ActionQuery;

Block:
  {Block} BEGIN statements+=Statement* END;

Statement:
  VariableDeclaration | IfStatement | ThrowStatement | ReturnStatement | SwitchStatement
  | WhileStatement | DoWhileStatement | ForLoopStatement | ForBasicStatement | Expression;

VariableDeclaration:
  'var' type=TypeReference name=ID '=' expression=Expression;

ReturnStatement:
  {ReturnStatement} 'return' (=> expression=Expression)?;

IfStatement:
  'if' '('(' condition=Expression ')' | => condition=Expression) thenBlock=ConditionalBlock
  (=> 'else' elseBlock=ConditionalBlock)?;

ConditionalBlock returns Block:
  statements+=Statement | Block;

ThrowStatement:
  'throw' expression=Expression;

SwitchStatement:
  'match' '('(' declaration=Expression ')' | => declaration=Expression)
  BEGIN
  (cases+=CasePart)*
  (default?='default' ':' defaultBlock=ConditionalBlock)?
  END;

CasePart:
  'case' case=Expression (':' thenBlock=ConditionalBlock /* / fallThrough?='','*/);

WhileStatement:
  'while' '('(' condition=Expression ')' | => condition=Expression) block=ConditionalBlock;

DoWhileStatement:
  'do' block=ConditionalBlock 'while' '('(' condition=Expression ')' | => condition=Expression);

ForLoopStatement:
  'for' '('(' declaration=TypeVariable 'in' forExpression=Expression ')' block=Block;

ForBasicStatement:
  'for' '('(' declaration
  =VariableDeclaration ';' condition=Expression ';' progression=Expression ')' block=Block;

Expression:
```

A.1. DSL for Smart Contracts - Contract Modeling Language (CML)

```

AssignmentExpression;
AssignmentExpression returns Expression:
  OrExpression
    (= > ({AssignmentExpression.left=current} op=('=' | '+=' | '-=') right=Expression));
OrExpression returns Expression:
  AndExpression (= > ({OrExpression.left=current} op='or') right=AndExpression)*;
AndExpression returns Expression:
  EqualityExpression (= > ({AndExpression.left=current} op='and') right=EqualityExpression)*;
EqualityExpression returns Expression:
  RelationalExpression
    (= > ({EqualityExpression.left=current} op=('==' | '!=')) right=RelationalExpression)*;
RelationalExpression returns Expression:
  OtherOperatorExpression
    (= > ({InstanceOfExpression.expression=current} 'is') type=TypeReference |
    => ({RelationalExpression
      .left=current} op('>' | '<' | '>=' | '<=')) right=OtherOperatorExpression)*;
OtherOperatorExpression returns Expression:
  AdditiveExpression
    (= > ({OtherOperatorExpression.left=current} op='=>') right=AdditiveExpression)*;
AdditiveExpression returns Expression:
  MultiplicativeExpression
    (= > ({AdditiveExpression.left=current} op('=' | '-')) right=MultiplicativeExpression)*;
MultiplicativeExpression returns Expression:
  UnaryExpression (= > ({MultiplicativeExpression
    .left=current} op('*' | '/' | '**' | '%')) right=UnaryExpression)*;
UnaryExpression returns Expression:
  {UnaryExpression} op('not' | '!' | '-' | '+') operand=UnaryExpression | CastedExpression;
CastedExpression returns Expression:
  PostfixExpression (= > ({CastedExpression.target=current} 'as') type=TypeReference)*;
PostfixExpression returns Expression:
  FeatureSelectionExpression
    (= > ({PostfixExpression.operand=current} op("++" | "--" | "@pre")) |
    => ({ArrayAccessExpression.array=current}
    '[' indexes+=Expression ']' (= > '[' indexes+=Expression '])*);
FeatureSelectionExpression returns Expression:
  PrimaryExpression
    (= > ({FeatureSelectionExpression
    .receiver=current} ("." | explicitStatic?="::")) feature=[Feature]
    (= > opCall?='(' (args+=Expression (',' args+=Expression)*)? ')')?)*;
PrimaryExpression returns Expression:
  LiteralExpression |
  {ThisExpression} 'this' |
  {SuperExpression} 'super' |
  {ReferenceExpression} reference
    = [NamedElement] (= > opCall?='(' (args+=Expression (',' args+=Expression)*)? ')')?
    (= > ({ArrayAccessExpression
    .array=current} '[' indexes+=Expression ']' (= > '[' indexes+=Expression '])*)? |
  {NewExpression}
    'new' type=[Type|QualifiedName] '(' (args+=Expression (',' args+=Expression)*)? ')' |
  {NestedExpression} '(' child=Expression ')';
LiteralExpression returns Expression:
  {IntegerLiteral} value=INT |
  {BooleanLiteral} value=('false' | 'true') |
  {StringLiteral} value=STRING |
  {RealLiteral} value=REAL |
  {DurationLiteral} value=INT unit=TimeUnit |
  {DateTimeLiteral} value=DATE |
  {NullLiteral} 'null' |
  {ArrayLiteral} '{' (elements+=Expression (',' elements+=Expression)*)? '}' |
  {Closure} '[' expression=ExpressionInClosure ']';
ExpressionInClosure returns Expression:

```

A. Appendix

```
{Block} (expressions+=Expression ';'?)*;
enum TimeUnit:
  SECOND='seconds' | MINUTE='minutes' | HOUR='hours' | DAY='days' | WEEK='weeks';
QualifiedName:
  ID ('.' ID)*;
QualifiedNameWithWildcard:
  QualifiedName '.*'?;
fragment typeVariables *:
  '<' typeVars+=TypeVariable (',' typeVars+=TypeVariable)* '>';
fragment identifiedBy *:
  'identified' 'by' identifier=[Feature];
fragment isAbstract *:
  abstract?='abstract';
fragment extends *:
  'extends' superclass=TypeReference;
terminal DATE:
  '0'..'9' '0'..'9' '0'..'9' '0'..'9' '-' '0'..'9' '0'..'9' '-' '0'..'9' '0'..'9';
terminal REAL returns ecore::EBigDecimal:
  INT '.' INT;
terminal BEGIN:
  'synthetic:BEGIN';
terminal END:
  'synthetic:END';
```

Listing A.1: DSL specification in Xtext for the Contract Modeling Language (CML).

A.2. DevOps for Ethereum Smart Contracts

The DevOps approach that we introduced in Chapter 7 is implemented in GitLab CI/CD [174]. GitLab CI/CD is a tool for software development (using the continuous methodologies) that enables the automation, customization, and execution of software development workflows. Listing A.2 shows the complete YAML file (.gitlab-ci.yml) for project configuration, which is placed in the repository root and defines the pipelines, jobs, and environments. The full sample project code from which this excerpt is taken can be found on GitHub [242].

```
variables:
  NPM_TOKEN: ${CI_JOB_TOKEN}
  NODE_VERSION: 16.7.0
default:
  image: node:${NODE_VERSION}
  before_script:
    - npm ci --cache .npm --prefer-offline
    - |
      {
        echo "@${CI_
↔  _PROJECT_ROOT_NAMESPACE}:registry=${CI_API_V4_URL}/projects/${CI_PROJECT_ID}/packages/npm/"
        echo "$_
↔  {CI_API_V4_URL#https?}/projects/${CI_PROJECT_ID}/packages/npm/:_authToken=${CI_JOB_TOKEN}"
      } | tee --append .npmrc
    - chmod +x ./utils/setup-env.sh
```

Yaml


```

- ./utils/setup-env.sh
cache:
  key: ${CI_COMMIT_REF_SLUG}
  paths:
    - .npm/
    - node_modules/
stages:
- build
- test
- report
- release
- deploy
- operate
build-src:
  stage: build
  script:
    - echo "Compiling the code..."
    - npm run compile
  artifacts:
    paths:
      - abi
      - artifacts
build-doc:
  stage: build
  script:
    - echo "Generating docs..."
    - npm run docgen
  artifacts:
    paths:
      - docs
lint-test:
  stage: test
  script:
    - echo "Linting code..."
    - npm run lint:sol
    - echo "No lint issues found."
vulnerability-test 1/2:
  stage: test
  script:
    - apt-get update && apt-get install -y python3-pip
    - pip3 install slither-analyzer
    - echo "Running slither..."
    - slither .
  allow_failure: false
vulnerability-test 2/2:
  stage: test
  #image: mythril/myth
  before_script:
    - apt-get update
    - apt-get install -y software-properties-common
    - add-apt-repository ppa:ethereum/ethereum
    - apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys 1C52189C923F6CA9
    - apt-get update
    - apt-get install -y solc
    - apt-get install -y libssl-dev python3-dev python3-pip libleveldb-dev
    - pip3 install mythril
  script:
    - chmod +x ./utils/mythril-check.sh
    - ./utils/mythril-check.sh
  allow_failure: false
unit-test:
  stage: test
  script:
    - echo "Running unit tests..."
    - npm run test
secrets-test:
  stage: test
  image: python:latest
  before_script: ["."] # noop, override default before_script
  script:
    - echo "Running secrets scanner..."
    - pip install trufflehog3
    - trufflehog3 --no-history -v
  allow_failure: false
coverage-report:
  stage: report
  script:
    - echo "Generating code coverage report..."
    - npm run coverage
  artifacts:
    paths:
      - coverage

```

A. Appendix

```
- coverage.json
allow_failure: true
gas-report:
  stage: report
  script:
    - echo "Generating gas report..."
    - npm run test # creates gas-report.txt
    - echo "<pre>" > gas-report.html
    - npx ansi-to-html gas-report.txt --fg black --bg white >> gas-report.html
  artifacts:
    paths:
      - gas-report.html
release:
  stage: release
  script:
    - tar -zcvf abi.tar.gz abi --ignore-failed-read
    - tar -zcvf docs.tar.gz docs --ignore-failed-read
    - tar -zcvf artifacts.tar.gz artifacts --ignore-failed-read
    - tar -zcvf coverage.tar.gz coverage coverage.json --ignore-failed-read
    - tar -zcvf gas-report.tar.gz gas-report.html --ignore-failed-read
    - npm run semantic-release
  only:
    - main
  except:
  refs:
    - tags
  variables:
    - $CI_COMMIT_TITLE =~ /^RELEASE: .+$/
deploy_staging:
  stage: deploy
  extends: .git:push
  when: manual
  script:
    - echo "Deploy to staging environment"
    - npm run deploy:staging
    - rsync -a deployments "${CI_COMMIT_SHA}"
  environment: staging
  only:
    - main
  artifacts:
    paths:
      - deployments
deploy_prod:
  stage: deploy
  extends: .git:push
  when: manual
  script:
    - echo "Deploy to production environment"
    - npm run deploy:production
    - rsync -a deployments "${CI_COMMIT_SHA}"
  environment: production
  only:
    - main
  artifacts:
    paths:
      - deployments
verify_staging:
  stage: operate
  needs: [deploy_staging]
  script:
    - npm run verify:staging
verify_delpoy:
  stage: operate
  needs: [deploy_prod]
  script:
    - npm run verify:production
.git:push:
  extends: default
  before_script:
    - apt-get update && apt-get install -y rsync
    - git clone "https://root:$GITLAB_TOKEN@${CI_SERVER_HOST}/${CI_PROJECT_PATH}.git"
    ↪ "${CI_COMMIT_SHA}" # nosecret
    - git config --global user.email "${GIT_USER_EMAIL:-$GITLAB_USER_EMAIL}"
    - git config --global user.name "${GIT_USER_NAME:-$GITLAB_USER_NAME}"
  after_script:
    - cd "${CI_COMMIT_SHA}"
    - git add .
    - |
      CHANGES=$(git status --porcelain | wc -l)
      if [ "$CHANGES" -gt "0" ]; then
        git status
        git commit -m "Updated by job with ID: $CI_JOB_ID [ci skip]"
```

A.2. DevOps for Ethereum Smart Contracts

```
git push origin "${CI_DEFAULT_BRANCH}" -o ci.skip  
fi
```

Listing A.2: GitLab CI/CD configuration for Ethereum smart contracts.