Characterizing (un)successful open source blockchain projects and their testing practices

by

Luisa Fernanda Palechor Anacona

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering and Intelligent Systems

Department of Electrical and Computer Engineering
University of Alberta

© Luisa Fernanda Palechor Anacona, 2022

Abstract

The most well-known blockchain applications are cryptocurrencies, e.g., Ether and Bitcoin, which both sum a market cap of more than 560 billion US dollars. Besides cryptocurrency applications, programmable blockchain allows the development of different applications, e.g., peer-to-peer selling of renewable energy from smart grids, digital rights management, and supply chain tracking and operation. These applications can be developed and deployed on the blockchain through smart contracts, which are small programs that run on the blockchain under particular conditions.

As bugs in blockchain applications (in particular, cryptocurrencies) can have large financial impact, it is important to ensure that these applications are well-developed and well-tested. However, currently software development and testing practices of blockchain projects are largely unstudied. In this thesis, we study data from GitHub and CoinMarketCap to understand the characteristics of successful and unsuccessful blockchain projects and reveal the testing practices in Solidity projects with the aim of helping developers to identify projects from which they can learn, or should contribute to.

In the first part of the thesis, we study data from CoinMarketCap and GitHub to gain knowledge about the characteristics of successful and unsuccessful blockchain projects. We build a random forest classifier with 320 labelled projects and metrics from 3 dimensions (activity, popularity, and complexity). We found that a large number of stars and a project's age can help distinguish between successful and unsuccessful projects. Additionally, we found that code cloning practices tend to be common in unsuccessful projects written in Python, C++, Java and Solidity.

In the second part of the thesis, we explore how quality is addressed in blockchain applications by studying how 139 open source Solidity projects are tested. We show that core development team members are the developers who usually contribute to testing files, leaving external contributions rare. In addition, our results indicate that only functional testing is practiced among the majority of Solidity projects, with Truffle and Hardhat being the tools commonly used to test Solidity smart contracts. Moreover, security testing is a practice rarely conducted, and performance testing is not conducted at all. We finally found that audits by a third party are common in several smart contracts.

Future researchers and developers can use our findings to understand what characterizes successful and unsuccessful blockchain projects and be aware of the testing practices developers conduct in open source blockchain projects.

Preface

The research of this thesis has been conducted in the Analytics of Software, GAmes, and Repository Data (ASGAARD) lab led by Dr. Cor-Paul Bezemer.

Chapter 3 of this thesis has been published as "L. Palechor and C. Bezemer, 2022. How are Solidity smart contracts tested in open source projects? An exploratory study. *2022 IEEE/ACM International Conference on Automation of Software Test (AST)* (p. 165-169)" [1]. I was responsible for the collection and cleaning of data, building and analysis of models, and manuscript composition. Dr. Bezemer was the supervisory author and was involved in concept formation and manuscript composition.

Acknowledgements

I would like to thank my supervisor Dr. Cor-Paul Bezemer for giving me the opportunity to endeavour this research project. Dr. Bezemer 's guidance and support throughout this journey have contributed to the success of this research project. I have learned about performing and presenting research in an accurate, responsible, and efficient manner, thanks to Dr. Bezemer. I greatly appreciate all your help during my MSc. studies.
I would like to thank Dr. Marek Reformat and Dr. Edmond Lou for being part of my thesis examiners.

Thanks to all the Asgaardians who assisted me academically and non-academic through my studies; especially I would like to thank Finlay for our deep talks in the lab.

Last but not least, I would like to thank my parents and my brother for their unwavering support. I am very grateful to my fiance, Cristhian, for all the encouragement, patience, humour and love every step of the way.

Table of Contents

1 Introduction and Background **1**

1.1 Introduction and Background 1

1.2 Blockchain 4

1.2.1 Blockchain Cryptoassets 6

2 Can we predict whether a blockchain project will be successful using software engineering metrics? **8**

2.1 Introduction 9

2.2 Background 12

2.2.1 Blockchain Cryptoassets 12

2.2.2 CoinMarketCap 12

2.2.3 Code cloning 13

2.3 Related Work 13

2.3.1 Software engineering metrics 13

2.3.2 Code cloning in blockchain 14

2.4 Methodology 14

2.4.1 Gathering data 14

2.4.2 Identifying important metrics for successful projects 19

2.4.3 Collecting historical data of metrics 19

2.4.4 Identifying similar blockchain projects 19

2.5 Preliminary analysis 20

2.6 RQ1. What are the most important metrics that characterize successful blockchain projects in CoinMarketCap? 22
2.6.1 Motivation 22
2.6.2 Approach 24
2.6.3 Findings 30
2.7 RQ2. How does development activity change across time in blockchain projects? 32
2.7.1 Motivation 33
2.7.2 Approach 33
2.7.3 Findings 34
2.8 RQ3. How similar are blockchain projects to each other? 36
2.8.1 Motivation 36
2.8.2 Approach 37
2.8.3 Findings 39
2.9 Threats to validity 41
2.9.1 Construct Validity 41
2.9.2 Internal Validity 41
2.9.3 External Validity 42
2.10 Conclusion 42

3 How are Solidity smart contracts tested in open source projects? An exploratory study 44

3.1 Introduction 45
3.2 Background and related work 46
3.3 Methodology 47
3.3.1 Gathering data 47
3.3.2 Identifying test files for smart contracts 48
3.3.3 Identifying configuration files 49

3.4 RQ1. Who are the developers involved in testing Solidity smart contracts? . . . 49
3.4.1 Motivation . . . 49
3.4.2 Approach . . . 49
3.4.3 Findings . . . 50
3.5 RQ2. What are the preferred tools and testnets for testing Solidity smart contracts? . . . 51
3.5.1 Motivation . . . 51
3.5.2 Approach . . . 51
3.5.3 Findings . . . 51
3.6 RQ3. What types of tests are performed on Solidity smart contracts? 53
3.6.1 Motivation . . . 53
3.6.2 Approach . . . 54
3.6.3 Findings . . . 54
3.7 Threats to validity . . . 55
3.7.1 Internal validity . . . 55
3.7.2 Construct validity . . . 55
3.7.3 External validity . . . 56
3.8 Conclusion . . . 56

4 Conclusions & Future Work 57

4.1 Conclusion . . . 57
4.2 Future Work . . . 58

Bibliography 59

List of Tables

2.1 The metrics used in the random forest model to classify successful projects 17

2.2 P-values and Cliff's deltas of the differences between successful and unsuccessful project metrics from activity, popularity and complexity domain. 20

2.3 Mean optimism values for precision, recall, F1-score and AUC. 28

2.4 The ranking of the most important metrics in the random forest classifier, divided into statistically distinct groups using the Scott-Knott Effect Size clustering. 30

2.5 Number of projects containing at least one file in the studied code languages and the percentage of cloned projects grouped by programming language. 39

3.1 Testing tools used by Solidity open source projects. 52

List of Figures

1.1 Example of blockchain application flow 5

2.1 Overview of our methodology. 15

2.2 Distribution of 12 metrics across successful and unsuccessful projects. The y-axis label Succ and Unsucc stand for successful and unsuccessful projects, respectively. Blue and orange distributions represent data for successful and unsuccessful projects, respectively. 21

2.3 Overview of the steps we follow to build the random forest classifier. . 23

2.4 Hierarchical cluster of metrics as stated by squared Spearman correlations coefficients. Read dotted line represents the threshold of $|p^2| = 0.7$. 24

2.5 Metric importance value for all 9 metrics studied in the random forest classifier. 29

2.6 The optimism-reduced value distribution for performance metrics of our random forest: Precision, Recall, F1-score and AUC. 32

2.7 Change in the number of commits for successful and unsuccessful projects in six-month periods. The blue color represents the distribution of commits in unsuccessful projects, and the orange indicates successful projects. The median of the number of commits in each semester is represented by the black dash in each box. 34

2.8 Change in the number of contributors for successful and unsuccessful projects in six-month periods. The blue color represents the distribution of contributors in successful projects, and the orange indicates unsuccessful projects. The median of the number of contributors in each semester is represented by the black dash in each box. 35
2.9 Percentage of file similarity for file pairs belonging to different projects written in C, C++, Solidity, Python and Java. 38

3.1 Overview of the steps taken in our methodology. 48
3.2 Portion of test developers who are core developers. 50
3.3 Number of testnets per project. 52
3.4 Popular testnets within OS projects. 53

Chapter 1

Introduction and Background

1.1 Introduction and Background

Blockchain is a decentralized technology that records transactions between parties [2]. Those transactions are listed or chained in data blocks and are replicated on each of the network's nodes. The process of adding a new transaction to the blockchain includes a consensus between the nodes of the network to agree on the present state of recorded transactions. This process can be executed without the intervention of any trusted third party. Therefore, blockchain provides a trusted decentralized network.

The field of application in blockchain is very broad. The core of this technology is a database of transactions or information that can be used in any kind of application domain. A programmable blockchain allows the deployment of applications on the top of the blockchain. For example, Ethereum, one of the most popular blockchain, can deploy and run applications via smart contracts.

Studies have focused on many blockchain-application domains proposing solutions to existing challenges that traditional applications face. For example, in the public sector, blockchain-based voting systems have been proposed and implemented in some countries to overcome voter access and voter fraud [3]. Likewise, supply chain management issues, such as data manipulation and single point of failure, are being faced with blockchain-based traceability solutions to improve supply chain transparency [4]. The health care sector is researching new ways of improving and simplifying the shar-

ing of health record information using blockchain [5]. In education, studies are exploring ways to share academic credentials using blockchain-based applications that guaranteeing integrity without consulting the issuing institution [6]. Financial industry, which is the most popular sector on implementing blockchain applications which has a total market capitalization of $961B, focuses on developing cryptoassets or digital assets to prove ownership, e.g., tokens and cryptocurrencies [7]. The mentioned blockchain applications demand a high-quality software development to ensure security, reliability and integrity of the data.

Open source projects have implemented different blockchain applications, but little is known about the software activities conducted in this domain. We investigate characteristics of successful and unsuccessful projects to better understand how software metrics can describe these projects. We further investigate the testing practices developers conduct.

In this thesis, we focused on extracting valuable insights from blockchain projects available on GitHub. In particular, we concentrate on projects listed on the top and bottom of CoinMarketCap and the software metrics which characterize them. Additionally, we studied open source blockchain projects developed in Solidity to identify the types of testing these projects perform. We conducted the following studies:

Research Study 1: Can we predict whether a blockchain project will be successful using software engineering metrics? (Chapter 2)

Motivation: In the last few years, a large number of cryptocurrencies have been deployed in different blockchain networks. Some of these cryptocurrencies are worth a few cents of dollars while others worth hundreds of dollars; it is unknown how software metrics can describe these projects.

We want to identify the software development activities performed by developers in open source blockchain projects and, at the same time, understand the software engineering metrics that better describe successful projects. Using this knowledge,

we want to predict early on when a project will become successful and provide helpful information to investors and developers interested in either, contributing or investing in a particular project.

In this study, we collected data from 320 open source blockchain projects hosted on GitHub. We classify them as successful or unsuccessful based on their market cap position. Extracting data about projects' activity, popularity, and complexity, we build a random forest model to determine the characteristics of the successful and unsuccessful projects.

Findings: The most important findings of this study are that software metrics are valuable to classify if a project is successful or not but cannot predict whether a project will become successful. The number of stars, project age and programming language are the most important metrics for our random forest model to classify successful and unsuccessful projects. In other words, a successful project may likely be popular and perform software maintenance activities. However, since these metrics do not provide historical information, we cannot use them to predict when a project will become successful. We also found that the number of commits and contributors has been decreasing since 2015 in both successful and unsuccessful projects. We found that this downtrend may be related to code cloning practices among the projects. Finally, unsuccessful projects are usually clones, especially in projects with files written in Solidity, Python, Java and C++.

Research Study 2: How are Solidity smart contracts tested in open source projects? An exploratory study (Chapter 3)

Motivation: We found in Chapter 2 several projects, in particular unsuccessful projects are clones.

According to Göde [8], most clones are rarely changed. We want to further study the software development activities of the blockchain projects in a narrow domain, i.e., smart-contract-based projects. To further study software development activities in blockchain projects, we investigate the state of smart contract testing in open

source projects because, similar to traditional applications, testing is a vital phase of the software development process.

Ethereum-based applications have been susceptible to multiple attacks, e.g., DAO and Roni attacks [9, 10], where losses rose to more than $60M in 2016 and $600M in 2022, respectively. Therefore, high-quality applications could potentially avoid bugs that lead to massive losses.

Ethereum smart contracts development has limited development support from the open source testing community compared to the support for traditional applications. Studies have shown that developers complain about open source development tools because these tools do not satisfy their needs [11–13].

We investigate the state of the smart contract testing in open source projects by analyzing 139 smart contract projects written in Solidity. We provide an analysis of the developers working on the tests, the testing frameworks and testnets used and the type of conduct tests.

Findings: We found that 80% of the test developers were also part of the core team of the studied projects. In addition, 50% of the projects tested their smart contracts mainly using functional frameworks. Less than 10% of the projects performed security testing, and none ran traditional performance testing.

The outcomes of the first study are helpful for future researchers to understand how software metrics characterize successful and unsuccessful projects, which could help developers decide which open source project to contribute to. These findings could also help future researchers in blockchain project classifiers and code similarity in blockchain projects.

1.2 Blockchain

A blockchain is a list of ordered transactions organized into linked lists of blocks. These blocks are usually linked to their predecessor by cryptographic hashes.

A blockchain system comprises a network of distributed servers or nodes that

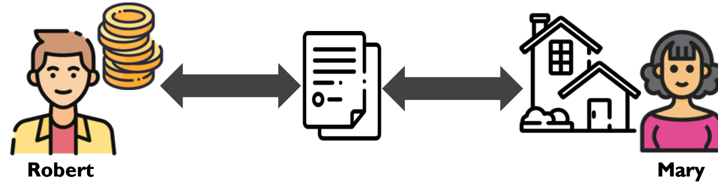

Figure 1.1: Example of blockchain application flow

contain a complete or partial copy of the blockchain data structure, and a network protocol to specify rights, obligations, and methods of communication, verification, validation, and consensus among the nodes.

According to the users that participated in the blockchain network, the blockchain could be classified as private or public. In private blockchain, the participation of new nodes in the network must be approved by others. In contrast, in public blockchain, anyone could join the network without approval. This means that in public blockchains, nodes could enter and leave the network without any required permission, verify data added to the blocks and process new transactions while receiving incentives from the network [2].

One of the blockchain applications includes notary services (proof of ownership). Figure 1.1 shows a conceptual example of Robert buying Mary's house using a blockchain application. Say Robert and Mary agree on the house price and targets accounts. The blockchain has a record that proves Mary owns the house; then, the final goals of the transactions are Mary having the payment and Robert the house ownership. After Robert transfers the amount to buy the house in an agreed account, e.g., Mary's account, the blockchain application validates Robert's payment and Mary's balance. If the transaction for the house payment is correct, the application will transfer the ownership of the house. Otherwise, the transaction is rejected (the house ownership won't be transferred) if the payment is incorrect.

Some advantages of using blockchain are that data cannot be manipulated due to their immutability characteristic and that the time to finish some processes could be

reduced. For example, in the mentioned scenario, selling a house could become faster since there is no need to have third parties such as a lawyer, notary, or public registry.

1.2.1 Blockchain Cryptoassets

Financial innovations are focused on blockchain applications, mainly in the cryptoassets. A cryptoasset is a blockchain application created to prove ownership of an asset. The cryptoassets, deployed in a blockchain public network, can securely send and receive transactions from other accounts. The types of cryptoassets include [7]:

Tokens: There are utility tokens and security tokens. Utility tokens are used to probe access rights to services or products. These tokens can be used to buy a product or service from a particular blockchain network. Utility tokens are usually used for a started business to raise money through a sale or an auction in an initial cryptoasset offering. In Ethereum, the utility tokens are implemented by standards, e.g. the ERC-20, which defines how to implement standard tokens and how to access the token's data to secure transfer tokens and get the confirmation of the balance [14].

On the other hand, security tokens refer to investments. Security token value is derived from an external and tradeable asset. Moreover, these tokens are subject to Federal laws that govern security. Examples of security tokens are shares of a company's ownership or intellectual property.

Non-fungible Token (NFT): These cryptoassets are tokens that cannot be replicated and record the ownership of a unique object such as a house, song, digital image, or video. In Ethereum, the ERC-721 defines an interface to track and transfer NFTs [15].

Cryptocurrency: or digital currency, is the most popular cryptoasset. Similar to real currency, this asset can be used as digital money to buy products. These assets can also be used in trading markets to exchange for other cryptocurrencies or stored as valuable assets.

The remainder of this thesis is organized as follows: Chapter 2 presents our study

on successful and unsuccessful blockchain projects, development activity trends, and project similarities. Chapter 3 presents our exploratory study on Solidity smart contracts, and the testing approaches developers take in practice. Finally, Chapter 4 concludes this thesis and highlights our findings and the potential future research directions.

Chapter 2

Can we predict whether a blockchain project will be successful using software engineering metrics?

Blockchain is a recent technology with many fields of applications. One of the hottest topics in this domain is financial applications, whose market cap is moving more than 2B dollars. This billion-dollar market is supported by software applications running in distributed blockchain networks. We intend to learn about these projects' software development activities to understand what characterizes a successful project that leads to millions of dollars in revenue. Distinguishing a blockchain project categorized as successful based on the market cap using software metrics is an unexplored field. We use data from 320 GitHub blockchain projects and identify the most important metrics among 3 categories (activity, popularity, and complexity) that best explain a successful blockchain project by building a random forest model. We study how commits and contributors' activity change across time and analyze similarities between code written in C, C++, Solidity, Java or Python.

Our study reveals the most important metrics for the model are the number of stars, project age and main programming language. Commits and contributors have a downtrend for successful and unsuccessful projects, and one possible reason is that there are similar projects in our dataset. We found at least 80% similarity between projects in 41.1% of the C++, Python and Solidity projects, 68.3% of the C projects

and 80.9% of the Java projects.

2.1 Introduction

Storing data using chained data blocks and cryptography across a blockchain network assures immutable applications which provide trust without the involvement of a third party. The blockchain network is composed of decentralized nodes that are in charge of data processing, validation and storing data that circulates in the blockchain.

Blockchain-based applications are experiencing massive growth and revenue. Particularly, the financial sector. This sector is the most popular field in blockchain holding, as of June 2022, with a total market cap of $1.2T [16]. One type of blockchain financial application is the cryptocurrency or digital currency. Digital currencies such as Bitcoin (BTC) and Ethereum (ETH) have a market cap of $560B and $212B, respectively, as of June 2022 [16], which makes them the top two most successful blockchain projects. Cryptocurrencies are the revolution of the financial sector, in particular banking, because they do not require external authorities to operate, the fees are arguably reduced, and there are no geographic limits.

Cryptocurrencies market cap varies from coin to coin. Predicting a successful cryptocurrency ahead of time can help developers and investors decide on which project to work on.

Many studies have investigated metrics to identify open source project success from different software engineering metrics. Subramanian et al. [17] and Stewart [18] studied project success by analyzing the impact of restrictive open source licenses on developer and user interests. Midha [19] investigated the success of open source projects through project popularity and developer activity metrics. Daniel and Stewart [20] studied the impact of software coupling and interactive discussions on the success of an open source project. Crowston [21] explored metrics to study how popularity, community size (i.e. number of active contributors to the project), and the time taken to fix bugs affect the open source project success and show that these

metrics are strongly correlated. Even though previous studies have focused on identifying open source projects, none have used software metrics to identify successful open source blockchain projects.

In this chapter, we leverage software metrics to identify successful blockchain projects. Since blockchain projects are programs built on software development, software engineering metrics can provide valuable information about their software life cycle. We want to study the association of these metrics to predict the blockchain project's success.

We study 13 metrics from open source blockchain projects to understand the characteristics that explain blockchain projects' success from a software perspective. For example, the number of stars, commits, files, and primary programming language. We study the correlation between the metrics and build a random forest model to determine if a project is successful or not. We define successful projects based on the market cap ranking provided by the standard price tracking source, CoinMarketCap. From CoinMarketCap, we collect blockchain projects metadata including the market cap ranking position to define successful and unsuccessful projects. We selected GitHub projects, mine their repositories, and perform data analysis to answer the following questions:

RQ1. What are the most important metrics that characterize successful blockchain projects in CoinMarketCap?

Motivation: Some cryptocurrency become successful, while others never reach the top positions in the CoinMarketCap rank. With this research question, we want to investigate across 13 metrics and 3 dimensions, i.e., activity, popularity, and complexity, the relation between software engineering metrics and the success of a project.

Findings: We found that number of stars, project age and programming language were the three most important metrics. These results mean that successful projects tend to be more popular and tend to have more development activity when compared to unsuccessful projects.

RQ2. How does development activity change across time in blockchain projects?

Motivation: The results of research question 1 show some metrics can characterize successful projects. Since we are interested in predicting early on whether a project will be successful, we further study if these metrics can provide meaningful information across time to make these predictions.

Findings: We found a downtrend in the number of commits and contributors for successful and unsuccessful projects. For this reason, even though results in research question 1 showed the metrics that characterize successful and unsuccessful projects, it becomes infeasible to predict a project's likelihood of success.

RQ3. How similar are blockchain projects to each other?

Motivation: Since the results of research question 2 show development activity decreases over time, we want to investigate if this trend results from new projects copying files from other projects. We believe cloning practices may be related to this trend since previous studies have revealed that clones in software projects might not change across time [8, 22] and others have shown that cloning files are very common in Ethereum smart contracts [23].

Findings: We found projects in our dataset are similar to at least one other project in more than 40% of the cases when comparing projects in either C++, Python, Solidity, C or Java.

These study will bring knowledge about the GitHub blockchain projects and their software metrics which can help developers to understand this technology better.

Chapter outline: We organize the remainder of this chapter as follows: In Section 2.2 we provide background information on Blockchain application and CoinMarketCap. Section 2.3 gives an overview of related work. Section 2.4 explains our methodology. Section 2.5 shows our preliminary analysis. Results for the 3 research questions in Section 2.6, 2.7 and 2.8. In Section 2.9, we identify threats to validity. Finally, we conclude the chapter with Section 2.10.

2.2 Background

This section provides background about blockchain financial applications or cryptoassets, CoinMarketCap, and code cloning.

2.2.1 Blockchain Cryptoassets

A Cryptoasset is a blockchain application created to prove ownership of an asset. The cryptoassets, deployed in a blockchain public network, can securely send and receive transactions from other accounts. The types of cryptoassets include [7]:

Tokens: There are utility tokens and security tokens. Utility tokens are used to probe access rights to services or products. These tokens can be used to buy a product or service from a particular blockchain network. Utility tokens are usually used for a started business to raise money through a sale or an auction in an initial cryptoasset offering. In Ethereum, the ERC-20 interface defines how to implement standard tokens and how to access the token's data to secure transfer tokens and get the confirmation of the balance [14].

Non-fungible Token (NFT): These cryptoassets are tokens that cannot be replicated and record the ownership of a unique object such as a house, song, digital image, or video. In Ethereum, the ERC-721 defines an interface to track and transfer NFTs [15].

Cryptocurrency: or digital currency, is the most popular cryptoasset. Similar to US dollars, this asset can be used as digital money to exchange products. These assets can also be used in trading markets to exchange for other cryptocurrencies or stored as valuable assets.

2.2.2 CoinMarketCap

CoinMarketCap is a website that provides information about the capitalization of cryptoassets. The data provided by CoinMarketCap includes a cryptoasset rank based on the total number of coins circulating and their current price on the market.

2.2.3 Code cloning

Clones are segments of code that are "extremely similar" or identical. Bellon et al. [24] proposed the following cloning classification in order to distinguish between the different code cloning types:

- Type-1 Clone: identical pieces of code except for differences in whitespace, and comments.
- Type-2 Clone: pieces of code syntactically identical, but variable, type, or function identifiers are altered.
- Type-3 Clone: a copied piece of code with further modifications, e.g., added, altered or removed a few statements.

2.3 Related Work

2.3.1 Software engineering metrics

Researchers have used process metrics and product metrics to capture information about the quality of software.

- Software process metrics are related to the timescale measure in the software development process [25]. Examples of these metrics are the number of commits in a release duration or the number of developers who changed a file in the same time frame [26].
- Software product metrics measure deliverables or artifacts of the development process, e.g., the size or complexity of the code. Some examples of software product metrics are lines of code, number of methods, or comment to code ratio [25, 26]

In a previous study, Rahman suggested that code metrics are less stable and less portable than process metrics [26]. Therefore, we primarily focus on development process metrics in this study.

Software metrics in blockchain: Blockchain previous studies have paid attention to development process metrics. For example, Choi et al. studied the maintenance effort in 592 blockchain projects by looking at 32 features grouped in metrics for developers' engagement, metrics for popularity and metrics for code updates. Among the features, they studied the number of watchers, stars, forks, issues, commits, and contributors [27]. They found that 48% of the projects in their dataset were not updated for the last six months, and 36.4 % of the projects disappeared.

Osman et al.studied 481 Bitcoin-related projects on GitHub to analyze the Bitcoin ecosystem [28]. The authors studied trends based on software metrics, e.g., the number of watchers, forks, commits, and contributors, to analyze popularity, maturity, activity and code equality. Their findings suggested the health of the majority of their studied projects is assessed as low risk.

2.3.2 Code cloning in blockchain

Kondo1 et al. used the Deckard tool to Ethereum smart contracts to study code cloning in Ethereum [23]. They used study different type of clones and found that more than 79% of the studied contracts were clones. They also compared the smart contracts to the OpenZeppelin project, which is a framework that offers well-tested smart contract templates, finding identical code blocks between the studied contracts and the ones from OpenZeppelin.

2.4 Methodology

In this section, we introduce the methodology of our study on identifying successful and unsuccessful blockchain projects. Figure 2.1 gives an overview of our study.

2.4.1 Gathering data

We collected the blockchain project data from two different sources, CoinMarketCap and GitHub APIs. From the CoinMarketCap we extracted data about successful and

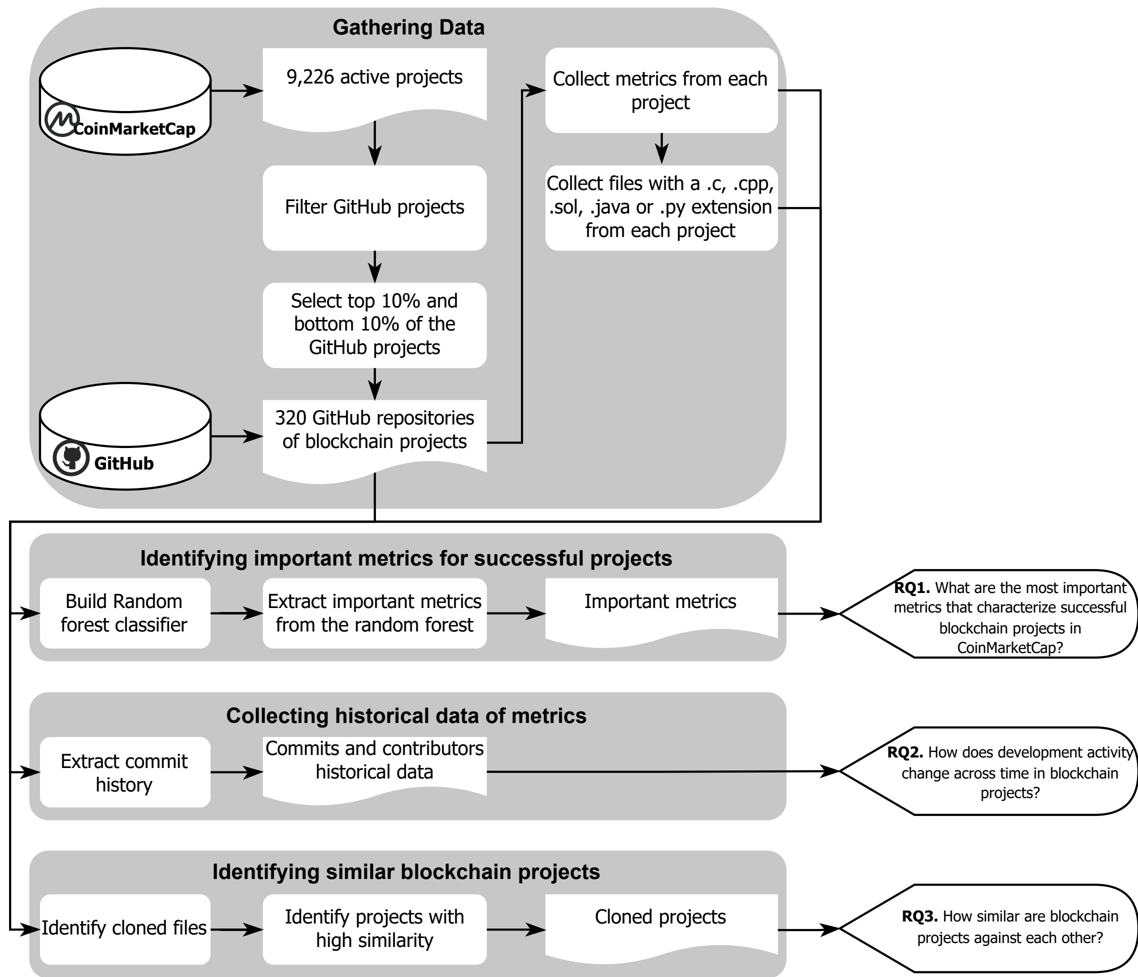

Figure 2.1: Overview of our methodology.

unsuccessful projects, and from GitHub, we gathered metrics we analyzed from each project.

CoinMarketCap is one of the most reliable price-tracking websites for blockchain cryptocurrencies, and its API has RESTful JSON endpoints to gather data about blockchain projects. This source has been used to extract blockchain data by Oliva et al. in their previous study [29]. From this source, we extracted metadata from active blockchain projects such as the blockchain project name, the blockchain network where the project was deployed, and the URL of the source code. Additionally, to get an insight about the success of each blockchain project, we obtained the rank available in CoinMarketCap. This rank refers to the relative position the project has in the market at a certain point and, it is calculated by using the cryptocurrencies price and the number of coins or tokens available for trading. We gathered data from 9,226 blockchain projects as of January 2022.

To collect software metrics from each project we use the information of the source code URL. 48% of the source code URLs provided from CoinMarketCap were empty, 49% pointed to GitHub; and 3% pointed to other types of repositories such as GitLab, BitBucket, or blockchain web explorers such as Etherscan. Since we were interested in open source projects and GitHub had most of the open source projects mentioned in CoinMarketCap, we selected only the projects stored on GitHub. There were 4,508 projects stored on GitHub.

We were focused on collecting software metrics from blockchain projects, then we only kept projects mapped to the GitHub repository because they provided information about the development work of open source projects. We removed 20 projects pointed to gist.github.com, 7 projects pointing to gitbub.com, 2,576 projects pointing to a GitHub organization and, 8 projects of which the repository no longer existed. We ended up with 1,897 projects.

CoinMarketCap provided data with different projects sharing the same GitHub repository URL. For example, the cryptocurrencies Koda Cryptocurrency and Bitcoin

both point to /bitcoin/bitcoin GitHub repository. In this case, we easily infer which cryptocurrency has the wrong URL but to avoid bias, we filtered out projects that shared GitHub repository URLs.

Since we were interested in studying the characteristics of successful projects, we sorted the records based on the project's rank and selected the top 10% and bottom 10% of the projects. In the chosen projects, the lowest rank for the 10% top was 1,418, and the highest rank for the 10% bottom was 7,096. We ended up with 320 projects.

Table 2.1: The metrics used in the random forest model to classify successful projects

Category	**Metric**	**Value**	**Description**
Activity	Mean # of commits per month	Numerical	The number of commits in a GitHub repository may determine how active a project is and, at the same time, provide understanding about the maintenance the project receives [30]. We would expect to see more commits per month in successful projects.
	# of days since last commit	Numerical	A short period may indicate developers are still working on fixing bugs or creating new features for the projects; in contrast, a longer period may suggest the project's abandonment. A longer period might be a characteristic of unsuccessful projects.
	Project age	Numerical	The experience acquired in a project, measured as the number of days from the first commit until the last one, might be positive correlated with the project's quality and, arguably, the project's success.

continued on next page

continued from previous page

Category	Metric	Value	Description
	Mean of the # of releases per month	Numerical	The number of releases provides insight into the new features, and bug fixes a project has delivered to its users. We investigate how this number is related to successful projects, instinctively thinking successful projects would have a lower number of releases.
Popularity	Mean # of contributors per month	Numerical	Popular projects are more likely to receive pull requests from contributors [31].
	# of watchers	Numerical	The number of watchers or developers who want to know the details of the activity involved in a GitHub project have been consider a measure of popularity in previous studies [32, 33]. We would expect successful projects having more number of watchers
	# of forks	Numerical	The number of forks as a metric of popularity has been used in previous investigations [32–34]. We would expect successful projects will be forked more times compared to unsuccessful ones.
	# of stars	Numerical	Number of stars has been used as a popularity metric in different studies [34–36].
	# of opened issues	Numerical	
	# of closed issues	Numerical	
Complexity	Length of readme file	Numerical	
	# of files in the repository	Numerical	
	Main programming language	Categorical	

We cloned 320 projects and extracted metrics related to the activity, popularity, and, complexity to analyze the characteristics of successful projects and study how they differ with unsuccessful projects. Activity metrics measure the software development contributions within each projects. Popularity metrics inform about how much interest developers have in the project. Complexity metrics collect how large a

project is in terms of files. Table 2.1 presents the description of the metrics used.

2.4.2 Identifying important metrics for successful projects

We used the collected metrics to build a random forest model and obtain the most important metrics for the model to classify successful and unsuccessful projects through the model's built-in feature importance.

2.4.3 Collecting historical data of metrics

Having cloned the GitHub projects, we extracted the commit history from each project through git commands. We obtained both commits and contributors' historical information from commits historical data.

2.4.4 Identifying similar blockchain projects

To identify similar blockchain project, we first identify files written in C, C++, Solidity, Java and Python. We grouped files under the same extension using regular expressions, i.e., .c, .cpp, .sol, .py, and .java.

We analyzed type-1 clone using the `diff` Unix command. We used these command because it is able to measure how similar a file is against another file by comparin line by line of each of the files, and because `diff` is language-agnostic. We contemplated clone detection tools such as Deckard and JPlag, but these tools did not meet the expectations to reach the goals of this study. The JPlag tool did not support the analysis of projects written Solidity. The Deckard tool supported Solidity but did not analyze the similarity between completed Solidity files. Instead, this tool analyzes code similarity only in code fragments.

We categorized cloned and non-cloned files based on the file pairwise similarity distribution. Our similarity metric is the percentage of cloned files in a project across all project combinations.

Table 2.2: P-values and Cliff's deltas of the differences between successful and unsuccessful project metrics from activity, popularity and complexity domain.

Metric	$p-value$	**Cliff's delta**
# of commits per month (mean)	<1.3x10-11	medium
# of days since last commit	=0.057	negligible
Project's age	<2.2x10-16	large
# of releases per month (mean)	=1.4x10-11	medium
# of contributions per month	=1.241x10-15	large
# of watchers	<2.2x10-16	large
# of forks	<2.2x10-16	large
# of stars	<2.2x10-16	large
# of opened	<2.12x10-13	medium
# of closed issues	<9.52x10-08	small
Length of readme file	=1.85x10-11	medium
# of files in the repository	<8.41x10-13	medium

2.5 Preliminary analysis

Table 2.1 presents the description of the 13 metrics used in this study. We calculate basic statistics about these software metrics for successful and unsuccessful projects. From this analysis we get an overview of the activity, popularity, and complexity of the successful and unsuccessful projects on GitHub.

We calculate the Wilcoxon Signed-Rank test to evaluate if the successful and unsuccessful project distributions are significantly different. In addition, we used Cliff's delta effect size [37] d to quantify the difference between the distributions, using the thresholds mentioned by Romano et al. [37]: negligible if $|d| \leq 0.147$; small if $0.147 < |d| \leq 0.33$; medium if $0.33 < |d| \leq 0.474$; and large if $0.474 < |d| \leq 1$.

Figure 2.2 shows the distribution of the 12 metrics across 3 domains when analyzing successful and unsuccessful projects. We found that the distributions for successful projects were different from unsuccessful projects, except # of days since last commit ($p-value = 0.057$), after compute the Wilcoxon Signed-Rank test. Table 2.2 shows

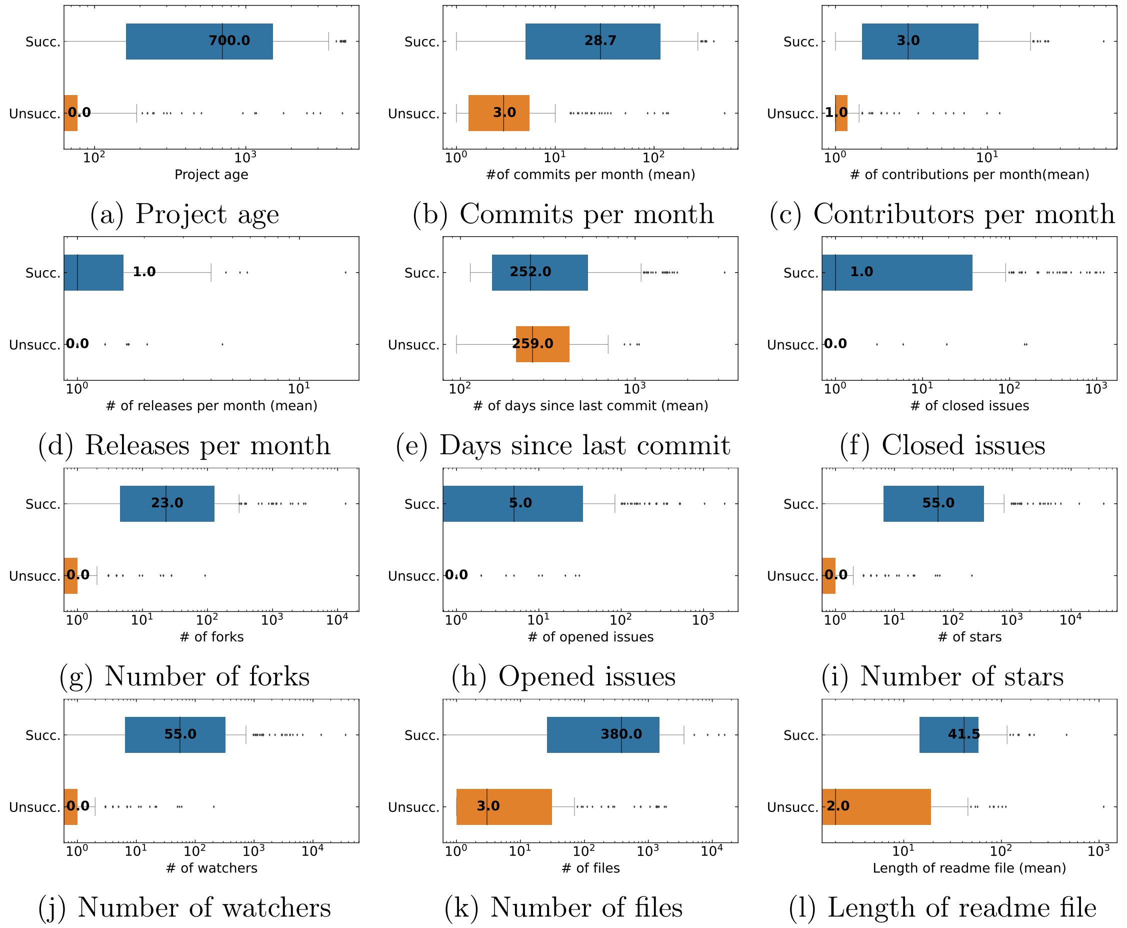

Figure 2.2: Distribution of 12 metrics across successful and unsuccessful projects. The y-axis label Succ and Unsucc stand for successful and unsuccessful projects, respectively. Blue and orange distributions represent data for successful and unsuccessful projects, respectively.

the $p-value$ for Wilcoxon Signed-Rank tests and the effect size ranges from negligible to large.

Unsurprisingly, the median values of watchers (Figure 2.2j), forks (Figure 2.2g) and stars (Figure 2.2i) are higher in successful projects, meaning that successful projects tend to be more popular than unsuccessful projects. Both open and closed issues in GitHub are low across most projects. Even though the distributions of the number of open and closed issues are statistically different, the effect sizes are medium and small.

In this preliminary analysis, we found the selected metrics provide meaningful information to distinguish between the two project categories.

2.6 RQ1. What are the most important metrics that characterize successful blockchain projects in CoinMarketCap?

In this section, we present the motivation, approach, and results of research question 1.

2.6.1 Motivation

Previous studies [17–21] have investigated the success in open source projects through software engineering metrics. We want to understand how software development activity, popularity and complexity explain the success of a cryptocurrency blockchain project. The findings can lead us to the metrics that could predict ahead of time which cryptocurrency will succeed.

To study the metrics that best distinguish successful projects from unsuccessful ones, we developed a random forest model to determine, from the metrics that we chose, what are the best that explain a successful blockchain project. The approach that we followed is shown in Figure 2.3. Below we detail each step.

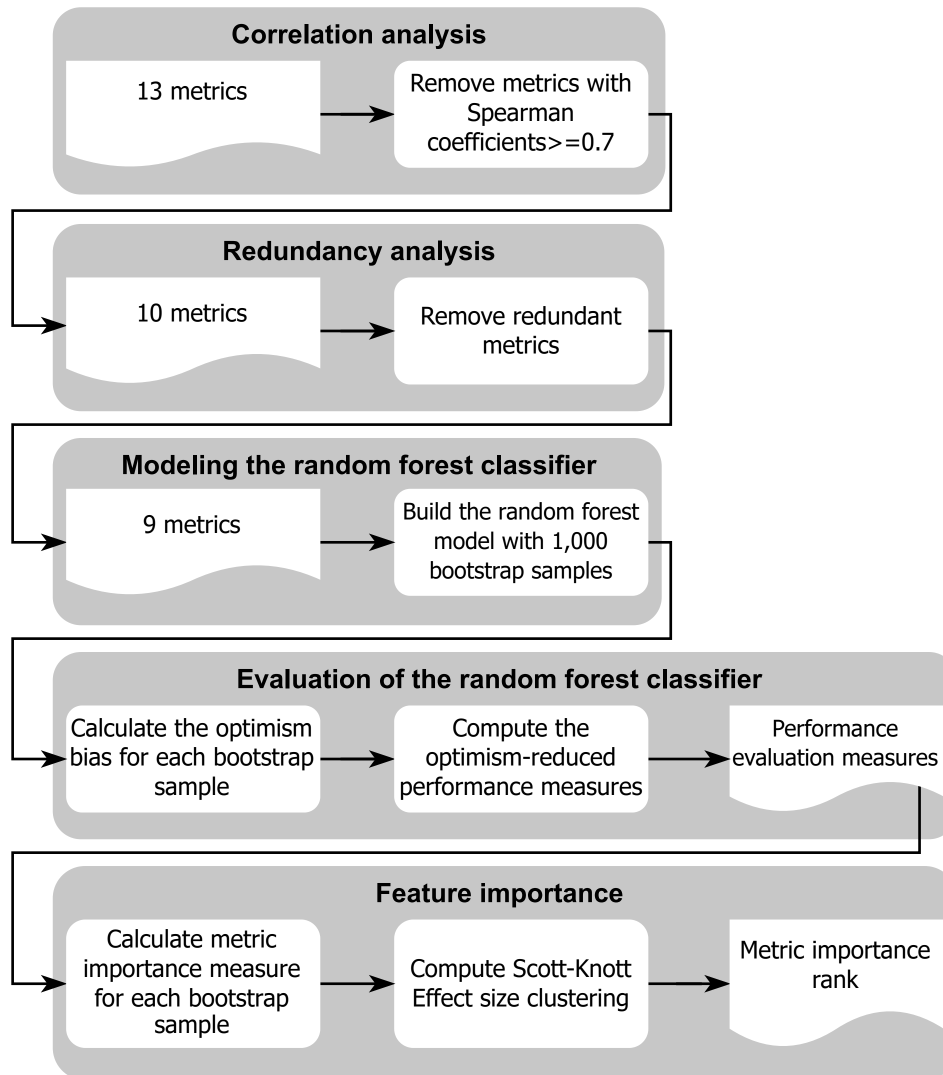

Figure 2.3: Overview of the steps we follow to build the random forest classifier.

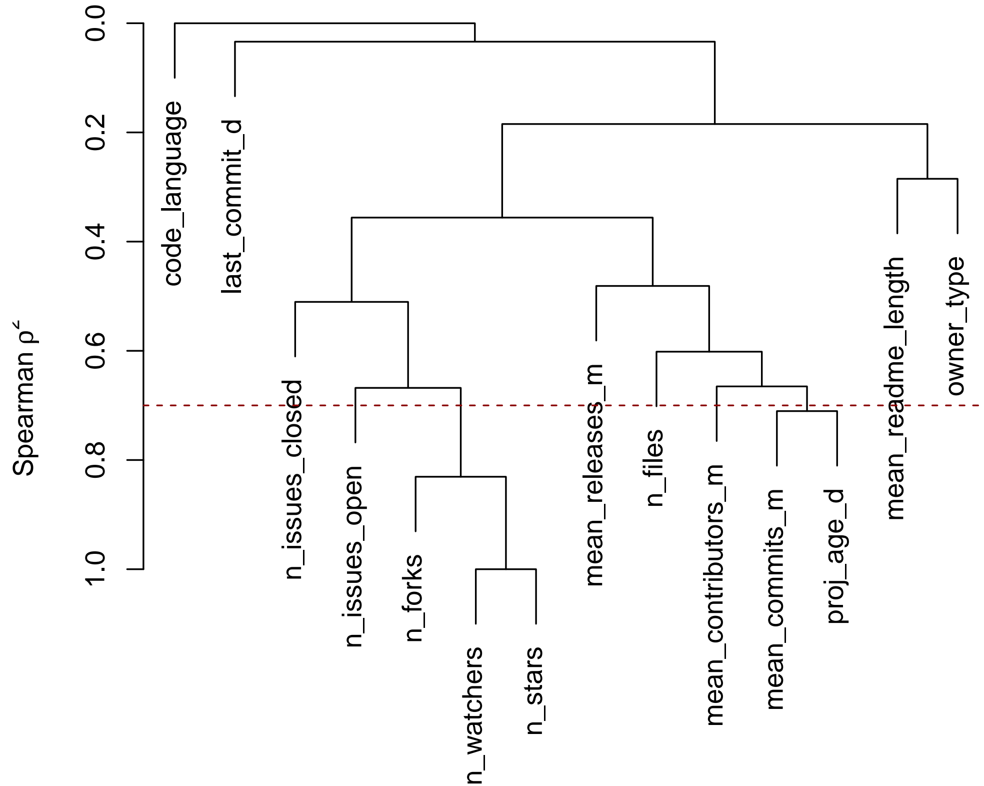

Figure 2.4: Hierarchical cluster of metrics as stated by squared Spearman correlations coefficients. Read dotted line represents the threshold of $|p^2| = 0.7$.

2.6.2 Approach

Correlation analysis: This analysis is performed to identify and remove strongly correlated metrics. Correlated variables can affect the model's performance (overfitting) and reduce the statistical significance of the variables. In this step, we analyze pairwise metrics that are strongly correlated and filter one of them out to better study the importance of the metrics within the model.

We perform the Spearman correlation test in `R` using the package `Hmisc` (Harrell Miscellaneous) and the `varclus` function. The `varclus` function allowed us to obtain the squared Spearman correlation coefficients to identify monotonic relationships between variables [38].

To identify a strong correlation between metrics, we use the value 0.7 as the Spearman correlation coefficient's threshold following a suggestion of prior research [39]. If

the Spearman coefficient is equal or higher than the threshold, we define the metrics as strongly correlated. Figure 2.4 shows the Spearman correlation coefficients across the metrics.

After executing the Spearman correlation test, we found the # of stars and # of watchers were strongly correlated metrics ($|p|^2 > 0.7$). These were also strongly correlated with the # of forks. We filtered out # of watchers and # of forks. In addition, we found the mean # of contributors per month and project age were strongly correlated. We removed the mean # of contributors per month to avoid overfitting. After this step, we had ten remaining metrics.

Redundancy analysis: Following the correlation analysis, we performed the redundancy analysis to filter out metrics that can potentially interfere with other independent variables. In other words, we checked if the linear combination of metrics explained any of the remaining ones. We used the `redun` function from the `Hmisc` package in `R` to remove redundant metrics. The redundancy results show the # of open issues can be predicted by the linear combination of the other 9 metrics. We removed the # of open issues.

Modeling the random forest classifier: We select the random forest classifier because other studies [36, 40] have demonstrated it performs well on software engineering data. The random forest algorithm is an ensemble of individual decision tree algorithms [41], which provides a straightforward interpretation and understanding of the model.

In practice, the random forest uses groups of metrics, e.g., mean # of contributors per month, # of files, and # of stars in different random subsets of training data to learn under which conditions a project is likely to succeed. We consider 9 metrics, i.e., # of commits per month (mean), # of days since last commit, # of releases per month (mean), # of contributions per month, # of stars, # of closed issues, length of readme file, and # of files in the repository, as the independent variables.

We define the likelihood of having success as the dependent variable. Moreover,

our model has the following possible outcomes:

- True Positives (TPs): the successful projects correctly detected.
- False Negatives (FNs): the successful projects wrongly classified as unsuccessful projects
- True Negatives (TNs): the unsuccessful projects correctly classified.
- False Positive (FPs): the unsuccessful projects wrongly classified as successful projects

We trained the random forest model and evaluated its performance by measuring the following metrics:

Precision: The ratio of blockchain projects correctly classified as successful over projects distinguished as successful. A high precision value means the random forest model can accurately classify successful projects across all observations.

$Precision = TP/TP + FP$

Recall: This metric is the portion of the projects correctly classified as successful over the total successful projects. Recall measures the capacity of the model to detect successful projects correctly; a high recall value means more successful projects are detected.

$Recall = TP/TP + FN$

F1-score: The F1-score is used to represent the combination of Precision and Recall in a single metric [41] through the calculation of the harmonic mean of these two metrics. The F1-score penalizes low values for either Precision or Recall. Therefore, high F1-score values only can be obtained when both recall and precision are high.

$F1 = 2 * (precision * recall)/(precision + recall)$

Area Under the Curve (AUC): This metric measures how capable the classifier is in discriminating between successful and unsuccessful projects. The area under the curve of the true positive rate (the portion of the projects correctly classified as

successful) against the false positive rate (the portion of successful projects wrongly classified) is known as AUC. The AUC interval is between 0.5 and 1. An AUC of 1 means the classifier can always correctly classify a project; conversely, an AUC of 0.5 means the classifier is no better than a random classifier. We evaluate the AUC value using the "performance" function from the `ROCR` package in `R`.

Bootstrap aggregation: We implemented bootstrap aggregation to reduce variance. Bootstrap is the process of taking repeated samples with replacements from the original dataset. The bootstrap samples have a significant overlap with the original data. Two-thirds of the original data points come out in each bootstrap dataset. Therefore, some data points may appear multiple times, and others may not.

To avoid performance overestimation of the classifier, we calculated the optimism in the bootstrap sample as defined by Efron [42]. The optimism of the performance estimation is calculated as follows:

- We considered the size of each bootstrap dataset as the exact size of the original dataset, 320 projects.
- We build the random forest model using 1,000 different bootstrap datasets and the original dataset.
- We calculate the random forest performance metrics defined above for each bootstrap dataset and their model, i.e., precision, recall, F1-score, and AUC. In addition, we compute the same metrics for each model and the original dataset.
- We compute the difference between the performance measures of the original sample and the performance measures of the bootstrap sample, and we define the difference as the optimism bias.

We calculated the optimism mean of the 1,000 optimism bias calculated in the previous step; the results of the mean optimism values are shown in Table 2.3. The

Table 2.3: Mean optimism values for precision, recall, F1-score and AUC.

Precision optimism	**Recall optimism**	**F1-score optimism**	**AUC optimism**
$4.0 \times 10-4$	$5.0 \times 10-3$	$2.2 \times 10-3$	$3.4 \times 10-2$

small optimism values mean the random forest model is very stable, and the likelihood that the initial classifier overfits the data is reduced.

The optimism-reduced performance measures for precision, recall, F1-score measure and AUC are calculated by subtracting each measure's optimism mean from their corresponding measure for the original data sample.

Important metrics in the model: We calculated the importance of metrics in the random forest classifier using the permutation measurement. The permutation metric importance concept was first introduced by Breiman [43] as the quantification of the effect on the random forest model accuracy of randomly reshuffling each metric. We used the `importance` function (`scale=F`) from the `randomForest` package in `R`. During the bootstrapping process, the classifier computes and stores two different error rate values: the error rate by using the testing data and the error rate after randomly permuting each of the 9 metrics at a time in the testing data. The classifier then computes the difference and averages the values over all trees. The change in the error rate evidences how much model performance drops when the values of a metric are permuted. The higher the error rate difference obtained by permuting a metric, the higher the importance of the metric.

After obtaining the measured importance of the metrics, we executed the Scott-Knott Effect size clustering (SK-ESD) to compute the metrics' importance rank based on their effect size [44, 45]. These ranks determine the order of importance of metrics in distinguishing successful projects.

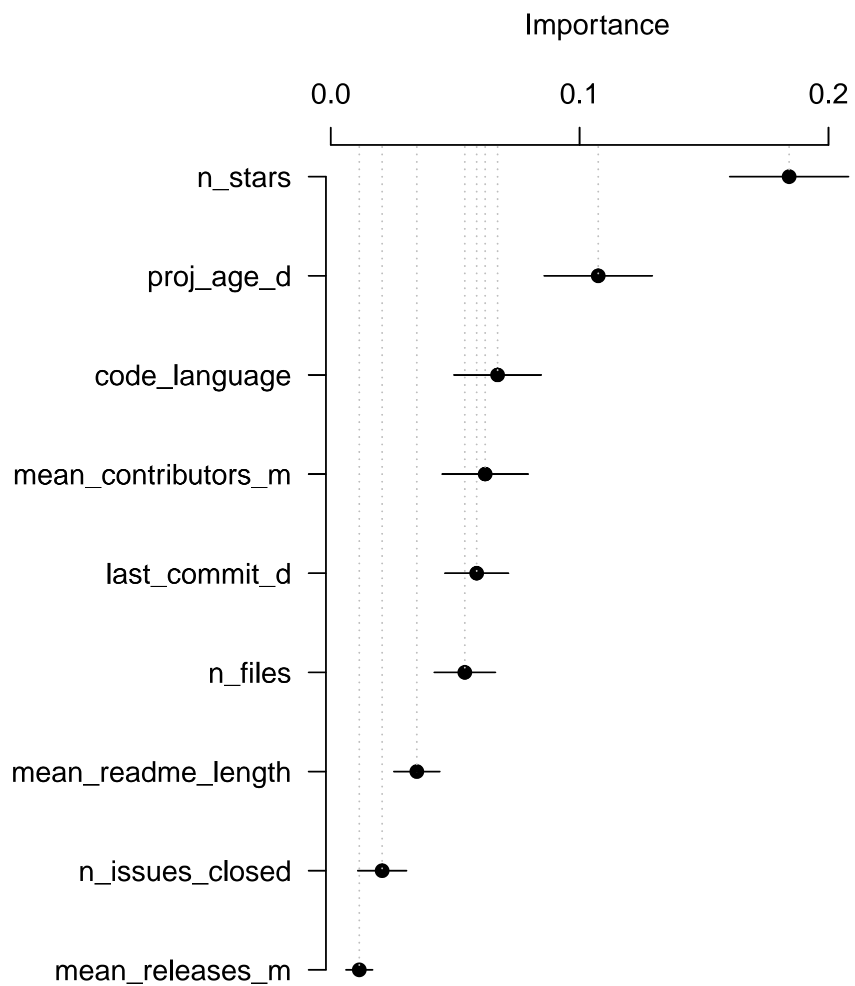

Figure 2.5: Metric importance value for all 9 metrics studied in the random forest classifier.

Table 2.4: The ranking of the most important metrics in the random forest classifier, divided into statistically distinct groups using the Scott-Knott Effect Size clustering.

Rank	Metric	Importance
1	n_stars	0.184
2	proj_age_d	0.106
3	code_language	0.066
4	mean_contributors_m	0.060
5	last_commit_d	0.058
6	n_files	0.053
7	mean_readme_length	0.034
8	n_issues_closed	0.019
9	mean_releases_m	0.010

2.6.3 Findings

Number of stars, project age and primary code language are the top 3 most important metrics in the random forest model. Table 2.4 shows all the metrics ranked by their importance scores, as determined by the Scott-Knott test. Figure 2.5 illustrates the metric importance values of the studied metrics.

The classifier uses the number of stars to distinguish between successful and unsuccessful projects. Unsurprisingly, the number of stars has a higher median number of stars in successful projects when compared to unsuccessful projects. In other words, successful projects are more popular compared to unsuccessful projects.

Project age is the second most important metric in the random forest classifier. Figure 2.2 shows that the median project age is 0 days for unsuccessful projects and 700 days for successful projects. This result suggests that making changes on unsuccessful projects, i.e., adding, modifying, or removing the source code, are not common tasks in at least 50% of the unsuccessful projects. On the contrary, 50% of the successful projects commit up to 700 days after the GitHub project creation. In other words, software maintenance activities that help projects improve or correct issues are not common in at least 50% of the unsuccessful projects.

We also observed 68% of the successful projects first committed on GitHub before 2020 and 8% of the unsuccessful projects first committed on GitHub before the same year. These results mean that since 2020, 92% of the unsuccessful projects have been created. This result could be related to the boost cryptocurrency markets had after the WHO identified of a worldwide pandemic. Cobert et al. [46] suggested that substantial investment flows entered cryptocurrency markets, which may have an association with the number of new cryptocurrency projects.

The third most important metric is code language. We noticed that 64% of the unsuccessful projects had Solidity as the primary programming language versus 17% of the successful projects; 18% of the successful projects had JavaScript as the primary programming language compared to 3% of the unsuccessful ones. Having Solidity as the main code language could indicate a project containing only smart contract files and lacking deep testing within the project. Solidity can be used to test smart contracts, but frameworks working with JavaScript provide more options for both unit and integration testing, i.e., using the HardHat testing framework.

The random forest classifier distinguishes between successful and unsuccessful projects with an AUC of 0.95. This result suggests the 9 proposed metrics have significant explanatory power as the random forest classifier accurately distinguishes between successful and unsuccessful projects. In addition, the random forest model has a precision of 0.88, a recall of 0.86, and an F1-score of 0.87.

The distributions of the random forest's performance measures minus the corresponding optimism bias are presented in Figure 2.6.

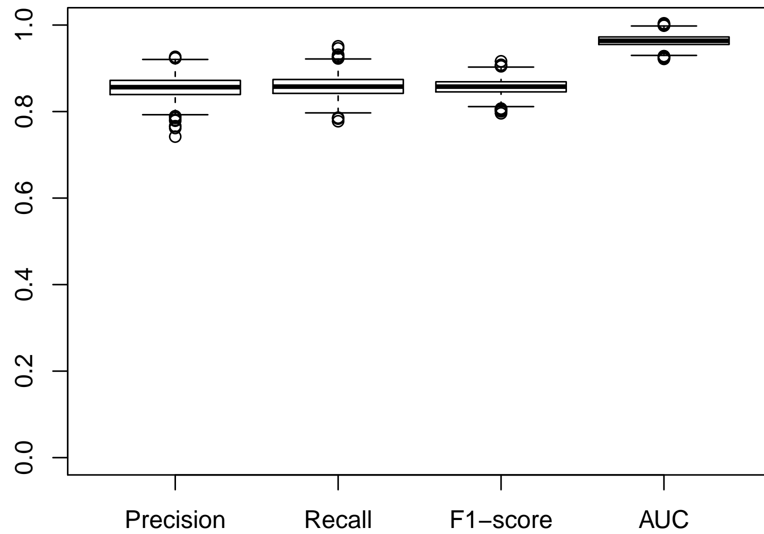

Figure 2.6: The optimism-reduced value distribution for performance metrics of our random forest: Precision, Recall, F1-score and AUC.

Summary

In our dataset, a successful project could be characterized by the number of stars, project age and code language are the most important metrics, which means successful projects tend to be more popular and more active than unsuccessful projects. Finally, JavaScript is more used as a first language in successful projects which may be an indication of testing activity.

2.7 RQ2. How does development activity change across time in blockchain projects?

In this section, we present the motivation, approach, and results of research question 2.

2.7.1 Motivation

The results of research question 1 show that the number of stars, project age and main programming language are the most important metrics that characterize successful projects. In this research question, we further investigate whether these metrics can provide meaningful information to predict beforehand whether a project will become successful. Therefore, we analyze historical data from commits and contributors of the projects between 2009-08 and 2022-01.

2.7.2 Approach

We used the commit information provided by the `git log` command to obtain both commits and contributors' information and analyze the evolution of software development in our dataset in 6-month periods, starting in January or July.

First, we grouped commits executed in the same semester to obtain the trends of commits and contributors across time. For each period, based on the `git log` information, we computed the total number of commits and contributors per project per semester.

We excluded data from the periods 2009-08 to 2009-12 and 2022-01 since we cannot complete 6-month periods with them, and this may affect the analysis.

We study the statistical significance of the trends by executing the Cox-Stuart test [47] using the median of the commits in each period. The Cox-Stuart test tests the null hypothesis that there is no trend. The alternative hypothesis is there is a trend in the data (decreasing or increasing). By dividing the observations in half, the Cox-Stuart test compares them using a sign test. The null hypothesis is rejected if the p-value is less than or equal to 0.05.

We ran the Wilcoxon-signed-rank test [48] to analyze if the distribution of historical successful project data was statistically different from the distribution of historical unsuccessful project data. The Wilcoxon-signed-ranked test has been used to evaluate the null hypothesis of two non-parametric distributions being identical. We calculated

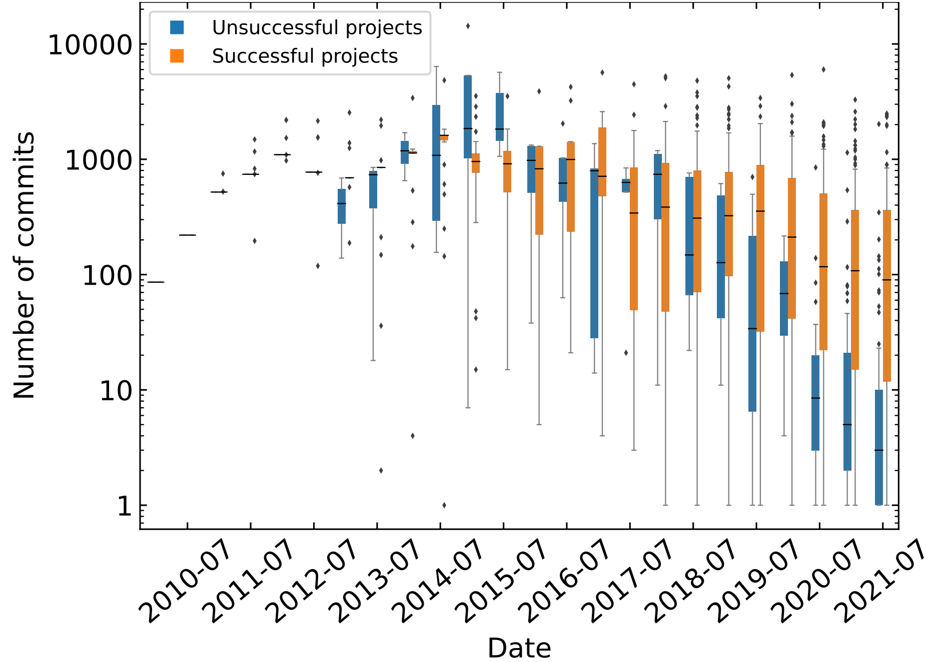

Figure 2.7: Change in the number of commits for successful and unsuccessful projects in six-month periods. The blue color represents the distribution of commits in unsuccessful projects, and the orange indicates successful projects. The median of the number of commits in each semester is represented by the black dash in each box.

Cliff's delta d to quantify the difference between the mentioned distributions using the same thresholds used in research question 1.

2.7.3 Findings

The number of commits and number of contributors has a downtrend for both successful and unsuccessful projects starting from 2014-07 and 2015-01, respectively. Figure 2.7 and Figure 2.8 shows respectively the number of commits created and contributors active every semester. We observed that the number of commits decreases after 2014-07 (p-value = 0.0078125) for success projects and decreases after 2015-01 for unsuccessful projects (p-value = 0.03125). For example, from 2014-07 to 2021-07, the median of the number of commits per semester falls from 1613 to 90 commits (1523 commits difference) in successful projects and from 1849 to 3 commits (1846 units commits difference) in unsuccessful projects.

Similarly, the number of contributors decreases after 2014-07 for successful projects

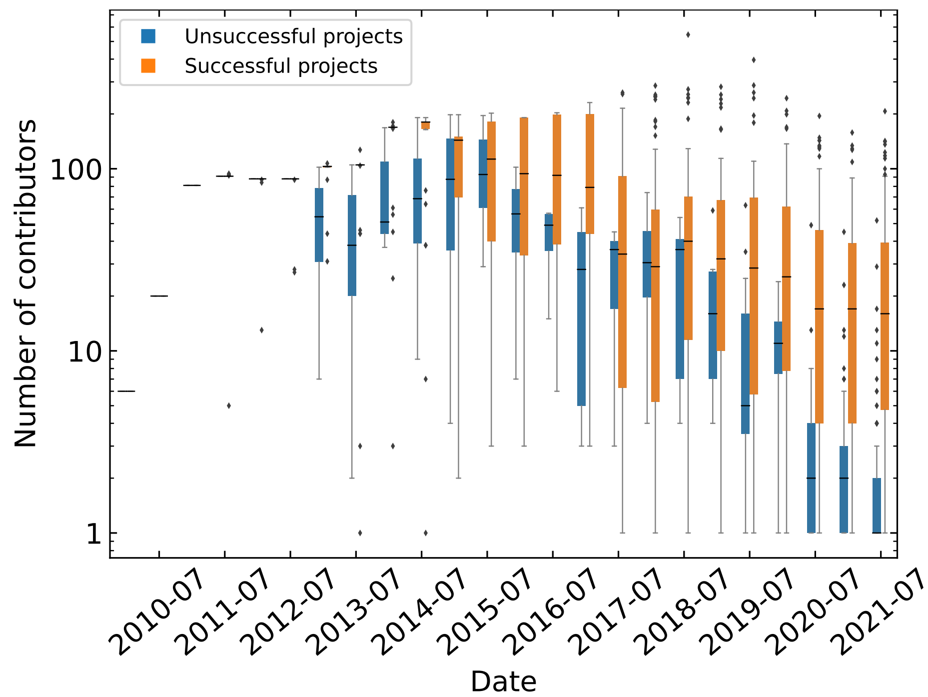

Figure 2.8: Change in the number of contributors for successful and unsuccessful projects in six-month periods. The blue color represents the distribution of contributors in successful projects, and the orange indicates unsuccessful projects. The median of the number of contributors in each semester is represented by the black dash in each box.

(p=0.0078) and after 2015-07 for unsuccessful projects (p-value = 0.0156).

There is no significant statistically difference between the medians of the commits in successful and unsuccessful projects (p-value = 0.68), however, there is a statistically significant difference in the median number of for contributors (p-value = 0.04882).

The obtained results suggest that software maintenance activity for the studied is decreasing in the last years few years in the studied projects. Since previous studies [8, 22, 49–51] have revealed that duplicated code has a tendency to be modified less frequently than non-duplicated code, we believe the downtrend could be related to code cloning.

In addition, the downtrend in the number of commits and contributors could be related to forked projects. In our dataset, there are 9 projects forked from /bitcoin/bitcoin GitHub project. Since Bitcoin was first released in 2009, more than a decade ago, these forked projects could not be as active as they were before 2014.

Summary

There has been a downtrend in the number of commits since 2014 and the number of contributors since 2015. We suggest that the decrease in the development activity may be related to cloning practices since previous studies have shown that clones usually are not actively maintained by developers.

2.8 RQ3. How similar are blockchain projects to each other?

In this section, we present the motivation, approach, and results of the research question 3.

2.8.1 Motivation

In research question 2, we found a downtrend in commits and contributors' activity. We investigate if the reason for this behaviour is related to code cloning activities since

researchers [8, 22, 49–51] have demonstrated cloned code change less than non-cloned code.

2.8.2 Approach

We studied the projects' similarity by analyzing the source code similarity of the files written in C, C++, Solidity, Python, and Java. To compare code similarity between the source code files of the different projects, we executed the `diff -u` command. We ended up using the Unix command because, to the best of our knowledge, there is not one clone detection tool available that supports clone file detection per a completed file when analyzing files in the studied programming languages. By using the `diff -u` command we identified type-1 clones. Next are the details steps we followed:

First, we grouped files under the same extension using regular expressions. We executed the `grep` command to identify the files under .c, .cpp, .sol, .py, and .java extensions across all projects. We were interested in the code similarity between files when comparing the source code lines; we removed comments and empty lines using the CLOC tool.

After this step, we counted the number of lines of each file and filtered out files with no lines.

We execute `diff -u` on all file pairs written in the same programming language but belonging to other projects. When comparing file A from project X against file B from project Y, we calculated how similar file A is compared to file B by computing the percentage of the number of equal lines from file A found in B divided by the total number of lines in file A. A *file similarity* result of 100% means that file A was identical to file B (but not necessarily the other way around). We calculated the file similarity of each project file against the files of the other projects and stored the results of all the result file pairs. Figure 2.9 illustrates the distribution of the file similarity percentage across all projects.

We defined a similarity percentage cutoff based on the file similarity distribution to

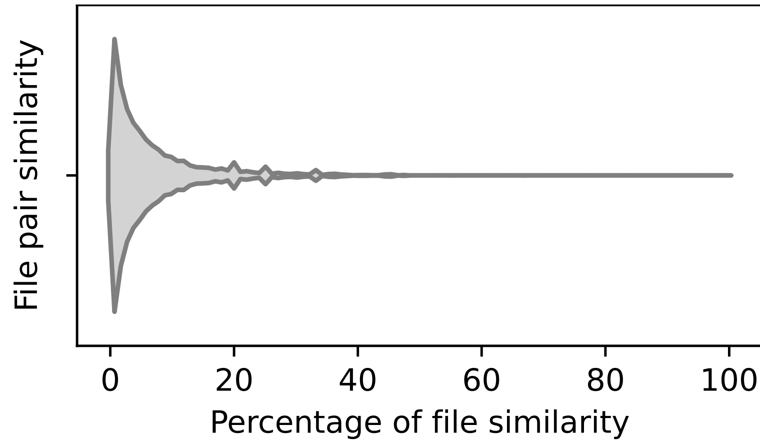

Figure 2.9: Percentage of file similarity for file pairs belonging to different projects written in C, C++, Solidity, Python and Java.

categorize cloned files. We look for a bimodal distribution that allows us to classify the clone and non-clone files. We analyzed thresholds based on the file similarity starting at 40% because Figure 2.9 shows a reasonable cutoff at this point. The file similarity of 99% of the file pairs is equal to or less than 40%. We use the threshold at 75% to accurately determine if a file was a clone or not. If the file similarity is higher than 75%, we classify the file as a clone; otherwise, if a file similarity percentage is equal to or less than 75%, we determine a non-clone file. We ended up with 264,497,022 (99.93% of the total file pairs) non-cloned file pairs and 180,226 (0.07% of the total file pairs) cloned files.

Next, we calculate the project similarity of each project pair. In this step, we counted the number of files cloned and non-cloned for each project pair and programming language. We counted the cloned files only once. We computed the percentage of the *project similarity* as the percentage of the number of files cloned in a project divided by the total number of files of the project.

We define a cloned project in the case of having a project similarity greater than 80%

We computed the Pearson's chi-squared test (χ^2) to analyze if the distribution of projects' success and clone project had a statistical relationship. The null hypothesis

Table 2.5: Number of projects containing at least one file in the studied code languages and the percentage of cloned projects grouped by programming language.

Programming languages	# of projects	% of cloned projects
Java	47	81%
C	60	68%
C++	56	41%
Solidity	191	41%
Python	79	41%

we test was the successful projects are independent from cloned projects and there is no relationship.

2.8.3 Findings

At least 41% of the projects are clones in the studied languages. Table 2.5 illustrates the number of the projects containing at least one file written in the studied programming languages, and the percentage of the projects with high similarity when analyzing cloned files in Java, C, C++, Solidity and Python. Java has the highest percentage of cloned projects. After manually reviewing files written in Java, we found Java files are mainly used to implement the digital signatures under the elliptic curve (Secp256k1) to ensure the integrity of messages created in the blockchain protocol. 38 projects use Java to build the elliptic curve for public key cryptography. We checked their Java files and noticed the pattern secp256k1 as part of their name, i.e., NativeSecp256k1.java.

Similar to Java, C projects are also used to implement the secp256k1. Likewise, C files implement cryptography hash functions and a key-value storage library, i.e., the LevelDB library developed by Google.

78% of the C++ cloned projects have between 168 to 333 cloned files. The high number of C++ similar files results from a blockchain protocol implementation. For example, the project /decenomy/pny has 328 files written in C++ and is 99.7% sim-

ilar to /birake/birakecoin (and vice-versa). When manually comparing the main.cpp between the projects, we only noticed a difference in the cryptoasset name printed on the messages. Similar case to the blockchain.cpp file.

Python files are mainly used for testing, e.g., to perform functional testing for transactions, permissions, and authentication for json-rpc connections; these files are stored in the project under functional, qa, and test paths.

We found 21 cloned projects common in C++ and Python. For example, 3 projects (/infinitecoin-project/infinitecoin, /supaafrikan/jolofcoin, /aspireorg/gasp) which have median of 96.6% project similarity when comparing the file pairs in C++, have more than 99% project similarity when comparing their 113 Python files. Similarly, /nodestats-ns/nodestats and /dequant-project/dequant have 99.2% project similarity when comparing C++ files, and share 100% project similarity when comparing their 50 Python files.

34 Solidity projects have 100% project similarity with other project. These projects have the characteristic of having only 1 Solidity file. These files might be the result of using templates to develop smart contracts, i.e., templates from OpenZeppelin to create cryptocurrencies.

Successful projects are related to low project similarity and unsuccessful projects to high project similarity in Solidity, Python, Java and C++. We applied Pearson's chi-squared test (χ^2) to the project similarity categories (cloned and non-cloned project) found in successful and unsuccessful projects and obtained that the relation between project similarity and project success was statistically significant for Solidity ($p<.0001$), Python ($p<.0016$), Java ($p<.0015$) and C++ ($p<.03$); however, we accept the null hypothesis, there is no relationship between project similarity and project success for C projects ($p=0.7$).

Summary

41% of the projects are clones of the studied projects. We found that code reuse from other sources to create their own projects is prevalent in cryptocurrency projects. Some of the clones we found are related to libraries that implement digital signatures used in blockchain, but other clones explored in C++ and Solidity are related to the creation of cryptocurrency projects.

2.9 Threats to validity

2.9.1 Construct Validity

Our approach studies similarities between projects taking into account one version of the project. We do not study similarities across historical project versions. Our results overlook code cloning that might occur before.

In this study, we rely on the `diff` command even the `diff` command allows comparing two files line by line and reports equal lines written in the files, this command is not intended to be a clone detector. We considered broadly-used clone detection tools, i.e. the Deckard and JPlag tools, but we discarded them because the Deckard algorithm performs the similarity analysis based on code blocks instead of projects, and the Jplag tool does not support Solidity programming language.

2.9.2 Internal Validity

Our data is limited by the accuracy of CoinMarketCap and GitHub APIs, and relevant projects may be excluded from our analysis, for example, projects listed in the CoinMarketCap rank without the source code URL or projects that share the source code URL.

An additional threat to the internal validity of our study is we only studied the top 15% and bottom 23% of the ranked blockchain projects that were on a GitHub repository. However, the top 15% and the bottom 23% ensure a clear contrast between successful and unsuccessful projects. In addition the selected projects are not close

to the median number of the total projects in the rank, avoiding the uncertainty of a project being successful or unsuccessful.

An additional threat to the internal validity of our study is we chose 75% of file similarity as the threshold to decide if the file was cloned. We analyzed the number of file pairs at 70% and 80% cutt and obtained a similar number of file pairs, the variation between 70% and 80% was 0.015% of the total number of file pairs. We did a similar analysis for the threshold to define cloned project 80%.

2.9.3 External Validity

We rely on CoinMarketCap rank collected on January 2022 to define a successful and unsuccessful project. However, the CoinMarketCap rank is susceptible to change according to the market capitalization of the cryptocurrencies. Therefore, new blockchain projects with different characteristics may be encountered. In addition, we only consider open source projects on GitHub. More extensive studies are needed to see whether our findings can be generalized to other types of projects.

Our study focuses on financial applications (cryptocurrencies); other domains, i.e., smart grid applications, may not reflect our findings.

2.10 Conclusion

Even if we can distinguish between successful and unsuccessful projects using a random forest model, we cannot predict when a project will become successful. In this chapter, we study 320 blockchain projects from GitHub to understand how 13 software metrics characterize successful and unsuccessful projects finding that number of stars, project age, and main programming language are the most important features. To complement the analysis, We studied trends from historical data of commits and contributors, and analyzed the code similarity in our projects. The most important findings of this chapter are:

- A random forest classifier can distinguish successful from unsuccessful projects

by using metrics from different domains, i.e., activity, complexity and popularity.

- Even the majority of the projects first commit after 2017, there is a downtrend in the number of commits and contributors analyzed every six months.

- Our results suggest that the downtrend could be related to projects with high similarity in our dataset.

Based on our findings, we suggest future work to investigate how the classifier is affected when the application type is taken into account. We didn't include the type of application, e.g., implementation of a protocol, wallet or token. Researchers could create recommendations for code reuse based on the type of application.

Chapter 3

How are Solidity smart contracts tested in open source projects? An exploratory study

Smart contracts are self-executing programs that are stored on the blockchain. Once a smart contract is compiled and deployed on the blockchain, it cannot be modified. Therefore, having a bug-free smart contract is vital. To ensure a bug-free smart contract, it must be tested thoroughly. However, little is known about how developers test smart contracts in practice. Our study explores 139 open source smart contract projects that are written in Solidity to investigate the state of smart contract testing from three dimensions: (1) the developers working on the tests, (2) the used testing frameworks and testnets and (3) the type of tests that are conducted. We found that mostly core developers of a project are responsible for testing the contracts. Second, developers typically use only functional testing frameworks to test a smart contract, with Truffle being the most popular one. Finally, our results show that functional testing is conducted in most of the studied projects (93%), security testing is only performed in a few projects (9.4%) and traditional performance testing is conducted in none. In addition, we found 34 projects that mentioned or published external audit reports.

3.1 Introduction

Academic and industrial attention in the blockchain technology has exploded in recent years. Blockchain is a decentralized, distributed technology that enables trust between entities without the involvement of a third party by storing permanent and unalterable *blocks* that ensure the integrity of the stored data. New data can be stored on the blockchain in several ways, one of them being through smart contracts. A smart contract is a self-executing script of which the proper execution is given when predefined conditions are met [52].

Ethereum is the most popular blockchain that stores smart contracts, with Solidity being the most popular high-level language for defining these contracts. Solidity is still a relatively immature language that is changing rapidly [53]. The rapid changes in Solidity add challenges to smart contract development and make it hard to thoroughly test smart contracts. Testing smart contracts is essential as bugs in smart contracts can lead to large financial losses (as demonstrated by the DAO bug which resulted in a loss of $60M [9]). Similar to traditional software development, our expectation is that the quality and thoroughness of the test suite of a smart contract is associated with the quality of the smart contract itself.

Several studies have proposed solutions for improving the quality of smart contracts [54–59]. In addition, there exist a small number of open source development platforms, e.g., `Truffle` [60] and `Hardhat` [61], to support the development and testing of smart contracts. However, three surveys [11–13] revealed that open source developers grumble about the lack of support for testing smart contracts compared to that for testing traditional applications.

In this chapter, we perform an exploratory study on how Solidity smart contracts are tested in open source projects. We studied 139 Solidity open source projects from GitHub repositories focusing on the following research questions:

RQ1. Who are the developers involved in testing Solidity smart con-

tracts? We found that mainly the core developers of a project are responsible for testing smart contracts, even though there are usually several non-core developers that contribute to the rest of the project.

RQ2. What are the preferred tools and testnets for testing Solidity smart contracts? We observed that `Truffle` and `Hardhat` are by far the most used tools for testing smart contracts. Few projects used other tools, in particular those stemming from academic research.

RQ3. What types of tests are performed on Solidity smart contracts? Smart contracts running in blockchain is a recent revolutionary technology. As in traditional applications, testing is essential to guarantee high-quality development. However, smart contract testing is affected by new technical challenges, e.g., the decentralization and immutability features. This research question shows smart contracts are tested mostly by functional test cases.

The remainder of this chapter is organized as follows. Section 3.2 discusses background concepts and related work. In Section 3.3, we describe our exploratory study setup and methodology. Section 3.4, 3.5 and 3.6 present our study results for each of our research questions. Section 3.7 discusses the threats to the validity and Section 3.8 concludes our study.

3.2 Background and related work

Blockchain and smart contracts. Blockchain is a decentralized, distributed technology that allows secure transactions to be made between entities without requiring a third party (such as a bank). An important property of a blockchain is that records (*blocks*) cannot be altered once they are added to the blockchain. The most famous example of blockchain technology is the Bitcoin currency which runs on the Bitcoin blockchain. However, recently other implementations of blockchain have become popular, with the Ethereum blockchain [62] being the most notable example.

Ethereum is open source and allows users to develop their own programs that

run on the blockchain. An example of such a program is a smart contract, a self-executing script (often written in Solidity) that executes in the Ethereum Virtual Machine (EVM) once the specified conditions in the contract are met. For example, the ownership of a house is transferred once a certain amount of money is transferred into the account that is specified in the contract. The performed transactions consume computational effort in the EVM that is measured in gas units. Since gas fees are paid in Ether (the Ethereum currency), gas represents real money. Hence, any transaction on the Ethereum blockchain costs money.

Testnets. To test a blockchain application, testnets can be used. Testnets are public blockchains that permit developers to test smart contracts in a staging environment with free gas. The most popular testnets for Ethereum are `Kovan`, `Rinkeby` and `Ropsten`, which allow to test smart contracts' functionalities and interactions in an EVM. Several EVM-based testnets exist that mimic the Ethereum blockchain but at the same time differ in technical configurations.

Tools for testing smart contracts. Most of the proposed testing tools focus on the security of smart contracts (e.g., `MythX` [63], `SmartCheck` [64], `ContractFuzzer` [54], `Harvey` [65]), `Madmax` [66], `Securify` [67], `Slither` [58], `Manticore` [56], `Echidna` [68], and, `ReGuard` [57]. Only a few frameworks focus on other aspects of testing smart contracts such as their functionality (e.g., `Truffle` [60], `Hardhat` [61], `dapp.tools` [69], and `Brownie` [70]).

3.3 Methodology

In this section, we explain the methodology of our exploratory study, which is depicted by Figure 3.1.

3.3.1 Gathering data

We queried the GitHub API to collect open source Solidity projects using a custom query in August, 2021. To ensure mature and active Solidity projects were considered,

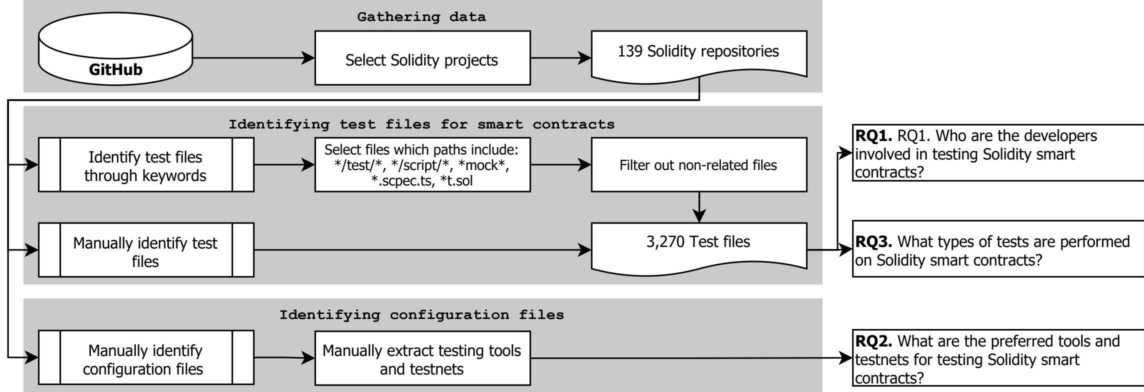

Figure 3.1: Overview of the steps taken in our methodology.

we limited our search to Solidity projects with at least ten stars and one hundred commits, likewise, we only considered non-archived projects. After these steps our dataset had 199 Solidity projects. Finally, we manually reviewed the Solidity projects and filtered out projects that were not intended for smart contract development. We excluded tutorials, repositories with no description, and testing frameworks, libraries and tools. At the end of this step, we obtained 139 repositories that contained at least one smart contract.

3.3.2 Identifying test files for smart contracts

The next step is to identify the test files in the projects. We were interested in actual test files and other files that reveal information about how testing is done in Solidity projects. To create our dataset, we first used heuristics to identify test files in each Solidity project. We included files that matched at least one of the next patterns: `*/test*/*`, `*/script/*`, `*mock*`, `*.spec.*` or `*t.sol*`. Second, the first author reviewed the obtained files and kept the files that provided data about the unit and integration testing, performance test cases and used testing tools. We ended up having 3,270 test files in our dataset. In addition, we manually reviewed the names of the folders in each project to make sure that we did not overlook relevant files. If we came across additional files that contained relevant information about the testing process, such as `Makefile` that specify the used testing framework, we considered

those for RQ2 and RQ3 as well.

Finally, we reviewed YAML files to analyze if the test was performed within a continuous integration (CI) pipeline. We recorded the testing script and the CI tool name that was used.

3.3.3 Identifying configuration files

We manually identified configuration files for testing tools (e.g., `truffle.js`). Our final dataset is available online [71].

3.4 RQ1. Who are the developers involved in testing Solidity smart contracts?

In this section we present the motivation, approach and findings for our research question 1.

3.4.1 Motivation

We study which developers are involved in smart contract testing to get a better idea of how this task is approached in open source projects. Our hypothesis is that this type of testing is not a popular contribution to make for open source contributors.

3.4.2 Approach

We used the same metric as in our prior work [72] to identify if developers working on testing are core developers of a project. First, we ran the git command `git shortlog -s -n HEAD` to identify the developers that committed to each test file. Second, we determine the top-n developers working on the project, where n was the total number of developers coding test files. We identify these n developers as the core developers of the project. Finally, we calculated the ratio between the total number of developers working on the tests and the top-n developers of every project.

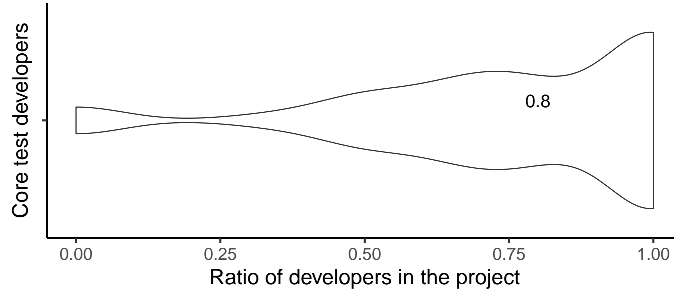

Figure 3.2: Portion of test developers who are core developers.

After obtaining the (test) developers names, we manually deduplicated them. For example, in the /balancer-labs/balancer-core project we merged the records for "fernandomartinelli" and "fernando martinelli" as they clearly point to the same person. In total, we merged 11% of the developers because they contributed to the profile under different names.

To compare the distributions, we utilized the Wilcoxon signed-rank test. The Wilcoxon signed-ranked test is a non-parametric statistical test of which the null hypothesis states that two input distributions are identical. We used Cliff's delta d to quantify the difference between the distributions, using the thresholds mentioned by Romano et al. [37]: negligible if $|d| \leq 0.147$; small if $0.147 < |d| \leq 0.33$; medium if $0.33 < |d| \leq 0.474$; and large if $0.474 < |d| \leq 1$.

3.4.3 Findings

In the studied projects a median of 80% of the test developers was also part of the core team. Figure 3.2 shows the portion of test developers who are core among the total of test developers. We observe a median of 0.80 for test developers who are part of the core team. This result suggests that testing smart contracts is a responsibility of core developers and having contributions from external developers is unlikely. The Wilcoxon signed-rank test shows that the two distributions are significantly different with a large effect size.

3.5 RQ2. What are the preferred tools and testnets for testing Solidity smart contracts?

In this section we present the motivation, approach and findings for our research question 2.

3.5.1 Motivation

Investigating which tools and testnets developers use to test smart contracts reveals which testing approaches developers take in practice. Moreover, even though all studied contracts are intended to end up on the Ethereum blockchain or any blockchain that adopts EVM as its smart contract runtime, for testing this is usually avoided as executing transactions over an EVM-compatible blockchain is not free (i.e., because of the gas consumption). Instead, developers can use testnets which are public blockchain networks to test the smart contracts in a real-world environment.

3.5.2 Approach

We investigated which testing tools and testnets are used by the studied projects by manually reviewing their configuration and script files.

3.5.3 Findings

69 (50%) of the studied projects use `Truffle` for testing. Table 3.1 shows that the most used testing frameworks are `Truffle`, `Hardhat` and `dapp.tools` which are mainly used for unit testing. Contrary, the least used testing tools in our data are `Manticore`, `Embark` and `Solgraph`.

48 (34.5%) of 139 projects do not provide information about using testnets as part of their testing process. This result suggests that these projects might test the smart contracts in a private blockchain without interaction with existing smart contracts, e.g., a private blockchain offered by `Truffle`, `Hardhat` or `dapp.tools`. Another explanation is that testnets are used but configured locally,

Table 3.1: Testing tools used by Solidity open source projects.

Tool	Description	# of projects
`Truffle`	Testing framework	69
`Hardhat`	Testing framework	54
`dapp.tools`	Testing framework	15
`Waffle`	Testing framework	8
`Slither`	Security analysis	6
`Jest`	Testing framework	5
`Brownie`	Testing framework	4
`Echidna`	Security analysis	4
`pytest`	Testing framework	3
`Mythx`	Security analysis	2
`Cetora`	Security analysis	2
`Manticore`	Security analysis	1
`Embark`	Testing framework	1
`Solgraph`	Security analysis	1

e.g., within an integrated development environment (IDE). Such configuration would imply that the tests are not intended for automated execution, e.g., in a CI pipeline.

While the median number of used testnets is 1, several projects test their smart contracts on more than one testnet. Figure 3.3 depicts the distribution of the number of testnets per project.

The /sushiswap/sushiswap project uses 13 testnets for testing its smart contracts,

Figure 3.3: Number of testnets per project.

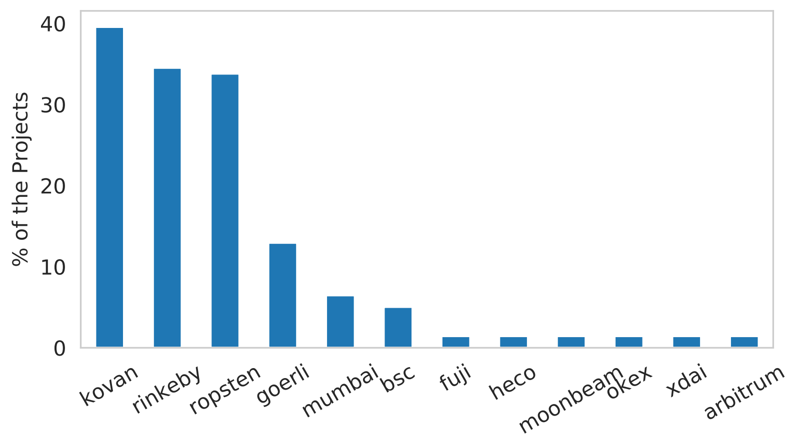

Figure 3.4: Popular testnets within OS projects.

/sushiswap/shoyu uses 8, /nftfy/nftfy-v1-core 7, and 3 projects perform their tests on 6 testnets. These projects all support smart contracts in multiple EVM-compatible blockchains (such as `Polygon`, `xDAI`, `Binance`, `Huobi`, `Avalanche Fuji` (testnet), `okexchain`, `arbitrum and celo`).

55 (39%) of 139 projects use `Kovan` **and 48 (34%) use** `Rinkeby`. Figure 3.4 shows the distribution of used testnets in the studied projects. The `Kovan`, `Rinkeby` and `Ropsten` testnets are the most popular. This is not surprising since these testnets are meant to test smart contracts in Ethereum which was the first blockchain to support smart contracts.

3.6 RQ3. What types of tests are performed on Solidity smart contracts?

In this section we present the motivation, approach and findings for our research question 3.

3.6.1 Motivation

Applications running in smart contracts are enhanced by features like decentralization and immutability. However, building a bug-free smart contract given these new features is tricky. This research question investigates which tests developers conduct

on smart contracts to overcome these challenges.

3.6.2 Approach

We manually went through the identified test files and classified the type of test as either a functional, security or performance test. We did not come across any other types of tests in the studied projects.

3.6.3 Findings

9 (6.5%) projects do not provide details about how they test their smart contract(s). After inspecting each of these projects, we did not find test-related code. We highlight, however, that 3 of the reviewed projects may include test files privately. For example, the /PancakeBunny-finance/Bunny project contains audit report files which specify the code coverage index for 4 smart contracts, the test report of 95 scenarios and a vulnerability analysis report. Furthermore, the `gitignore` file in the /yieldyak/smart-contracts project defines "test" within the list of untracked files. Finally, the /nftfy/nftfy-v1-core project includes a CI file that points to tasks related to testing the smart contracts of the project, e.g., `truffle test`.

130 (93.5%) of the projects in our data include functional testing as part of the testing strategy. Our results corroborate the findings of prior research [11, 12].

57 (41%) of the studied projects develop Solidity mocks to test the smart contracts by creating presumed scenarios. The mocks we found in the studied projects were developed to either override, add or reset internal functions to simulate particular scenarios within the blockchain application. For example, the /88mphapp/88mph-contract project has 5 mock files, that inherit from the ERC20 smart contract. Every mock file initializes the smart contract in a different way to simulate and test different scenarios.

81 (58%) of the studied projects include testing in the CI pipeline. We

noticed that the CI tools used were `Actions`, `TravisCI`, and `CircleCI`, similar to traditional open source projects. We also found 3 projects that include security testing in their CI pipeline.

13 (9.4%) projects conduct security tests. We observed that fuzzy testing and audit reports are used to test the security of smart contracts. 9 (6.5%) of the projects conduct fuzzing testing by feeding the code with random and erroneous data. These projects attempt to execute test cases multiple times while changing a particular parameter's value. As a consequence, multiple sequences of transactions are generated and reviewed automatically.

34 (24.5%) of the studied projects include audit reports performed by third parties. The audit reports have different formats across projects, but overall, they show the smart contracts' potential vulnerabilities and provide recommendations for improvements. For example, the audit report for the /balancer-labs/balancer-core project discusses a vulnerability that allows assets to be stolen.

None of the projects implement traditional performance testing. We did not find traditional performance tests (e.g., execution time) for the smart contracts. However, we did find 41 (29.5%) projects that report gas consumption in their tests.

3.7 Threats to validity

3.7.1 Internal validity

We filter the Solidity projects based on their number of stars and number of commits to measure maturity. Future studies should investigate if our findings would vary for a different set of projects.

3.7.2 Construct validity

Our manual review of the test files for smart contracts could be biased. Similarly, there are testing tools that have command-line interface (CLI) options available that allow developers to test their smart contracts without leaving a trace in the source

code repository. As a result, the numbers in our study may be lower estimates.

3.7.3 External validity

Our results are limited to Solidity projects and may not generalize to other languages for writing smart contracts. In addition, we focus on open source projects only. Future studies should extend our study to include other languages and proprietary projects.

3.8 Conclusion

This chapter analyzes how Solidity smart contracts in 139 open source projects are tested. We found that testing files for smart contracts do not receive many contributions from developers who are not already part of the core development team of a project. In addition, we found that functional testing is the most common for smart contracts, followed by security testing. The most popular testing tools are `Truffle` and `Hardhat`, and several projects use a third party to conduct an audit of their smart contracts.

Chapter 4

Conclusions & Future Work

4.1 Conclusion

In this thesis, we studied the million-dollar blockchain applications, the cryptocurrencies. We found that even though these applications have a large financial market, software development activity is limited.

- In Chapter 2, we studied the success of GitHub blockchain projects based on the market cap rank obtained from CoinMarketCap to understand their characteristics. We found that popularity and activity can distinguish between successful and unsuccessful projects. In other words, having a large number of stars and maintaining the projects after their creation explain successful projects.We also found that commits and contributors for the studied projects are decreasing over time. Finally, we found unsuccessful projects are similar to other projects (similarity between 80% to 100%).

- In Chapter 3, we explored how smart contracts written in Solidity tested their smart contracts. Our study showed that contributing to testing the Solidity smart contracts is not a prevalent task among external contributors; core developers mainly modify the smart contracts' test files. Additionally, we found developers used open source tools like Truffle and HardHat to test the contracts, and developers used Kovan and Rinkeby to test functionalities with other con-

tracts in an EVM environment. Finally, our results evidenced that most of the projects use functional testing. Many projects implement continuous automation pipelines to include their test cases. Different types of testing, i.e., security and performance testing, are conducted by very few or no projects.

4.2 Future Work

We list the potential future research directions in the following list:

- **studying the characteristics of successful applications on domains different from open source** We studied a project classifier using public metadata from CoinMarketCap API. However, enterprises solutions may have a different state of the software development activities in blockchain projects. For example, study supply chain applications deployed in private blockchain, such as Hyperledger.

- **Studying the promotion-as-a-service on GitHub and the impact on open source blockchain projects** We found the number of stars was an important metric for classifying successful projects. However, there are accounts on GitHub that conducted advertising services in GitHub, e.g., performing paid star and fork operations on repositories. Future studies should investigate the impact on this type of service on open source blockchain projects.

- **Studying the state of smart contract testing with other languages** Following Solidity, Vyper is also a popular language for developing smart contracts. Future studies should contemplate studying the testing state by different programming languages.

Bibliography

[1] L. Palechor and C.-P. Bezemer, "How are solidity smart contracts tested in open source projects? an exploratory study," in *2022 IEEE/ACM International Conference on Automation of Software Test (AST)*, 2022, pp. 165–169.

[2] X. Xu, I. Weber, and M. Staples, *Architecture for blockchain applications*. Springer, 2019.

[3] N. Kshetri and J. Voas, "Blockchain-enabled e-voting," *IEEE Software*, vol. 35, no. 4, pp. 95–99, 2018. DOI: 10.1109/MS.2018.2801546.

[4] J. Sunny, N. Undralla, and V. M. Pillai, "Supply chain transparency through blockchain-based traceability: An overview with demonstration," *Computers & Industrial Engineering*, vol. 150, p. 106 895, 2020.

[5] A. Hasselgren, K. Kralevska, D. Gligoroski, S. A. Pedersen, and A. Faxvaag, "Blockchain in healthcare and health sciences—a scoping review," *International Journal of Medical Informatics*, vol. 134, p. 104 040, 2020.

[6] D. J. Skiba *et al.*, "The potential of blockchain in education and health care," *Nursing education perspectives*, vol. 38, no. 4, pp. 220–221, 2017.

[7] W. Spaeth and T. Peráček, "Cryptocurrencies, electronic securities, security token offerings, non fungible tokens: New legal regulations for "crypto securities" and implications for issuers and investor and consumer protection," in *Developments in Information & Knowledge Management for Business Applications*, Springer, 2022, pp. 217–238.

[8] N. Göde and R. Koschke, "Frequency and risks of changes to clones," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, New York, NY, USA: Association for Computing Machinery, 2011, 311–320, ISBN: 9781450304450. DOI: 10.1145/1985793.1985836. [Online]. Available: https://doi.org/10.1145/1985793.1985836.

[9] Etherscan. "Thedao smart contract." (2022), [Online]. Available: http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code.

[10] R. Newsletter. "Community alert: Ronin validators compromised." (2022), [Online]. Available: https://roninblockchain.substack.com/p/community-alert-ronin-validators.

[11] J. Chen, X. Xia, D. Lo, J. Grundy, and X. Yang, "Maintenance-related concerns for post-deployed ethereum smart contract development: Issues, techniques, and future challenges," *Empirical Software Engineering*, vol. 26, no. 6, pp. 1–44, 2021.

[12] P. Chakraborty, R. Shahriyar, A. Iqbal, and A. Bosu, "Understanding the software development practices of blockchain projects: A survey," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ACM, 2018, ISBN: 9781450358231.

[13] A. Bosu, A. Iqbal, R. Shahriyar, and P. Chakraborty, "Understanding the motivations, challenges and needs of blockchain software developers: A survey," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2636–2673, 2019.

[14] F. Vogelsteller and V. Buterin. "Eip-20." (2015), [Online]. Available: https://eips.ethereum.org/EIPS/eip-20.

[15] W. Entriken, D. Shirley, J. Evans, and N. Sachs. "Eip-721: Non-fungible token standard." (2018), [Online]. Available: https://eips.ethereum.org/EIPS/eip-721.

[16] CoinMarketCap. "Today's cryptocurrency prices by market cap." (2022), [Online]. Available: https://coinmarketcap.com/.

[17] C. Subramaniam, R. Sen, and M. L. Nelson, "Determinants of open source software project success: A longitudinal study," *Decision Support Systems*, vol. 46, no. 2, pp. 576–585, 2009.

[18] K. J. Stewart, A. P. Ammeter, and L. M. Maruping, "Impact of license choice and organizational sponsorship on success in open source software development projects," *Information System Research*, vol. 17, no. 2, pp. 126–144, 2006.

[19] V. Midha and P. Palvia, "Factors affecting the success of open source software," *Journal of Systems and Software*, vol. 85, no. 4, pp. 895–905, 2012.

[20] S. Daniel and K. Stewart, "Open source project success: Resource access, flow, and integration," *The Journal of Strategic Information Systems*, vol. 25, no. 3, pp. 159–176, 2016.

[21] K. Crowston, J. Howison, and H. Annabi, "Information systems success in free and open source software development: Theory and measures," *Software Process: Improvement and Practice*, vol. 11, no. 2, pp. 123–148, 2006.

[22] N. Göde and J. Harder, "Clone stability," in *2011 15th European Conference on Software Maintenance and Reengineering*, IEEE, 2011, pp. 65–74.

[23] M. Kondo, G. A. Oliva, Z. M. J. Jiang, A. E. Hassan, and O. Mizuno, "Code cloning in smart contracts: A case study on verified contracts from the ethereum blockchain platform," *Empirical Software Engineering*, vol. 25, no. 6, pp. 4617–4675, 2020.

[24] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007. DOI: 10.1109/TSE.2007.70725.

[25] M. Scotto, A. Sillitti, G. Succi, and T. Vernazza, "A relational approach to software metrics," in *Proceedings of the 2004 ACM Symposium on Applied Computing*, ser. SAC '04, Association for Computing Machinery, 2004, 1536–1540, ISBN: 1581138121. DOI: 10.1145/967900.968207. [Online]. Available: https://doi-org.login.ezproxy.library.ualberta.ca/10.1145/967900.968207.

[26] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 432–441.

[27] J. Choi *et al.*, "Attack of the clones: Measuring the maintainability, originality and security of bitcoin'forks' in the wild," *arXiv preprint arXiv:2201.08678*, 2022.

[28] K. Osman and O. Baysal, "Health is wealth: Evaluating the health of the bitcoin ecosystem in github," in *2021 IEEE/ACM 4th International Workshop on Software Health in Projects, Ecosystems and Communities (SoHeal)*, IEEE, 2021, pp. 1–8.

[29] G. A. Oliva, A. E. Hassan, and Z. M. J. Jiang, "An exploratory study of smart contracts in the ethereum blockchain platform," *Empirical Software Engineering*, vol. 25, no. 3, pp. 1864–1904, 2020.

[30] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.

[31] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in github," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 805–816.

[32] A. S. Badashian and E. Stroulia, "Measuring user influence in github: The million follower fallacy," in *Proceedings of the 3rd International Workshop on CrowdSourcing in Software Engineering*, 2016, pp. 15–21.

[33] W. Ma, L. Chen, Y. Zhou, and B. Xu, "What are the dominant projects in the github python ecosystem?" In *2016 Third International Conference on Trustworthy Systems and their Applications (TSA)*, IEEE, 2016, pp. 87–95.

[34] K. Aggarwal, A. Hindle, and E. Stroulia, "Co-evolution of project documentation and popularity within github," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 360–363.

[35] H. Borges and M. T. Valente, "What's in a github star? understanding repository starring practices in a social coding platform," *Journal of Systems and Software*, vol. 146, pp. 112–129, 2018.

[36] H. Borges, A. Hora, and M. T. Valente, "Predicting the popularity of github repositories," in *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2016, pp. 1–10.

[37] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen'sd indices the most appropriate choices," in *annual meeting of the Southern Association for Institutional Research*, Citeseer, 2006, pp. 1–51.

[38] Datacamp. "Varclus: Variable clustering." (2022), [Online]. Available: https://www.rdocumentation.org/packages/Hmisc/versions/4.7-0/topics/varclus.

[39] H. C. Kraemer, G. A. Morgan, N. L. Leech, J. A. Gliner, J. J. Vaske, and R. J. Harmon, "Measures of clinical significance," *Journal of the American Academy of Child & Adolescent Psychiatry*, vol. 42, no. 12, pp. 1524–1529, 2003.

[40] L. Guo, Y. Ma, B. Cukic, and H. Singh, "Robust prediction of fault-proneness by random forests," in *15th international symposium on software reliability engineering*, IEEE, 2004, pp. 417–428.

[41] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

[42] B. Efron, "How biased is the apparent error rate of a prediction rule?" *Journal of the American statistical Association*, vol. 81, no. 394, pp. 461–470, 1986.

[43] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[44] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "The impact of automated parameter optimization on defect prediction models," *IEEE Transactions on Software Engineering*, vol. 45, no. 7, pp. 683–711, 2018.

[45] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 1–18, 2016.

[46] S. Corbet, Y. G. Hou, Y. Hu, C. Larkin, B. Lucey, and L. Oxley, "Cryptocurrency liquidity and volatility interrelationships during the covid-19 pandemic," *Finance Research Letters*, vol. 45, p. 102 137, 2022.

[47] D. R. Cox and A. Stuart, "Some quick sign tests for trend in location and dispersion," *Biometrika*, vol. 42, no. 1/2, pp. 80–95, 1955.

[48] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics*, Springer, 1992, pp. 196–202.

[49] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto, "Is duplicate code more frequently modified than non-duplicate code in software evolution? an empirical study on open source software," in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, 2010, pp. 73–82.

[50] J. Krinke, "Is cloned code more stable than non-cloned code?" In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, IEEE, 2008, pp. 57–66.

[51] J. Krinke, "Is cloned code older than non-cloned code?" In *Proceedings of the 5th International Workshop on Software Clones*, 2011, pp. 28–33.

[52] N. Szabo, "The idea of smart contracts," *Nick Szabo's papers and concise tutorials*, vol. 6, no. 1, 1997.

[53] Ethereum. "Solidity v0.8.11." (2021), [Online]. Available: https://docs.soliditylang.org/en/v0.8.11/.

[54] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ACM, 2018, 259—269, ISBN: 9781450359375.

[55] Y. Lu, X. Mao, Z. Li, Y. Zhang, T. Wang, and G. Yin, "Does the role matter? an investigation of the code quality of casual contributors in github," in *23rd Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, 2016, pp. 49–56.

[56] M. Mossberg *et al.*, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1186–1189.

[57] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: Finding reentrancy bugs in smart contracts," in *IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, 2018, pp. 65–68.

[58] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, IEEE, 2019, pp. 8–15.

[59] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[60] C. S. Inc. "Truffle suite." (2022), [Online]. Available: https://trufflesuite.com/.

[61] N. L. LLC. "Hardhat - ethereum development environment for professionals." (2022), [Online]. Available: https://hardhat.org/.

[62] V. Buterin, "A next-generation smart contract and decentralized application platform," *white paper*, vol. 3, pp. 1–36, 2014.

[63] ConsenSys. "Consensys mythx." (2021), [Online]. Available: https://mythx.io/.

[64] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," ACM, 2018, 9—16, ISBN: 9781450357265.

[65] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ACM, 2020, pp. 1398–1409.

[66] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Analyzing the out-of-gas world of smart contracts," *Commun. ACM*, vol. 63, no. 10, 87—95, 2020, ISSN: 0001-0782.

[67] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2018, pp. 67–82.

[68] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: Effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ACM, 2020, 557—560, ISBN: 9781450380089.

[69] dapp.tools. "Dapp.tools." (2022), [Online]. Available: https://dapp.tools/.

[70] Brownie. "Brownie - brownie 1.17.2 documentation." (2021), [Online]. Available: https://eth-brownie.readthedocs.io/en/stable/.

[71] L. Palechor and C. Bezemer. "How are solidity smart contracts tested in open source projects? an exploratory study." (2022), [Online]. Available: https://doi.org/10.5281/zenodo.5862800.

[72] P. Leitner and C.-p. Bezemer, "An exploratory study of the state of practice of performance testing in java-based open source projects," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ACM, 2017, pp. 373–384, ISBN: 9781450344043.