UNIVERSITY
OF OULU

Evaluating Ethereum Development Environments

University of Oulu
Information Processing Science
Master's Thesis
Samu Honkamäki
6.6.2022

Abstract

Blockchain technology has been one of the hottest buzzwords in the early 2020s and one of the main reasons for that is the interest towards decentralized applications, which use the smart contracts located in the blockchain to serve the application’s business logic. Ethereum is the biggest platform for decentralized applications, and this study focuses on exploring what kind of support developers need for developing Ethereum based products. This is done by first examining the state of the art by conducting a semi-systematic literature review, followed by using a customized DESMET evaluation method, in which the requirements are mapped as features along with the evaluation criteria, to see how well the currently popular Ethereum development environments provide support for the developers. A total of three development environments by the names of Hardhat, Truffle, and Brownie are evaluated, and the achieved results are analysed to find the differences in the level of support they offer for the developers. At the end the findings of the study are summarized, the experiences from the customized DESMET evaluation method are reported, the validity towards the achieved results are inspected, and the possible directions to continue the work is discussed.

Keywords
Development environment, Ethereum, Smart contract, Smart contract development, Decentralized application, Decentralized application development

Supervisor
University teacher Jouni Lappalainen

Contents

1. Introduction

The potential of blockchain technology has been a hot topic on the IT-field especially for the past 5 years. Professionals working on the field have been discussing how to utilize blockchain technology in different instances, as for example both the structure and the availability of the data stored into a blockchain differs vastly from a traditional way of storing data in a centralized ledger. For information sharing, blockchain technology can offer improvements in several categories, like in transparency, privacy, reliability, and security (Ølnes et al., 2017). Blockchain technology has also been a hot topic outside the IT-field, as people from the major news outlets to politicians and all the way to cryptocurrency investors are interested with their own motives to see what the blockchain technology will develop into. These aspects certainly play a role in the currently happening investment rush towards the decentralized internet, which is powered by blockchain technologies.

It is important to understand the technology behind the blockchain to fully grasp the potential of it. Blockchain is a type of distributed ledger technology (DLT). Burkhardt et al. (2018) explain the idea of DLT to be based on a decentralized record keeping, while the main purpose of it is to log all the happened transactions. The explanation of blockchain is similar, as Drescher (2017) summarizes it to be a distributed peer-to-peer system of ledgers, that utilizes the same algorithm, which handles all the data that has been stored into chained blocks. In simpler terms, blockchain technology enables applications to use a publicly shared database, where the data should be impossible to alter once it is written.

Zarir et al. (2021) explain Ethereum to be a blockchain platform, which enables the blockchain-powered applications by supporting the usage of smart contracts. These kinds of applications that utilize blockchain technology are called decentralized applications (Dapps). What separates decentralized applications from centralized applications is the use of smart contracts, which act as a backend for decentralized applications. In other words, smart contracts serve the business logic for decentralized applications. Zou et al. (2021) explain smart contracts as low-level code scripts, which are stored and executed on a blockchain. Authors continue by explaining that smart contracts contain both data and executable code, and its purpose is to facilitate a contract between two parties without the need of a trusted third party.

The ability to cut off the middleman while retaining the trust towards the system is groundbreaking, as it would make the companies operating as a trusted third party unnecessary. While the obvious use cases for this kind of technology are within the financial sector, lately non-fungible tokens (NFTs) and non-cryptocurrency Dapps have also received some attention from the masses. Dowling (2021) describes NFTs to be a blockchain traded rights to own any digital assets, like videos, images, or music, and it is just one of the blockchain related technologies that is being explored to see what it shall develop into.

Like NFTs, the whole blockchain technology is still under development and people are trying to figure out where and how it should be utilized. It possesses a lot of good characteristics, and when implemented well in a suitable context, it has the potential to disrupt the existing markets.

1.1 Research problem

Regardless of the field, to produce a well-made product efficiently, a good set of tools is a necessity. For a software developer this would mean having a well-thought and designed software development environment (SDE), which would support and ease the programmer's tasks. Darrell Corbin (1991) describes SDE to be an integration of different tools, standards, methodologies, and other related elements.

As previously mentioned, the difference between decentralized- and centralized applications is that the decentralized applications use smart contracts to serve the application's business logic, thus being what a backend is for a centralized application. Before deploying a smart contract into the Ethereum blockchain, a developer will first develop, test, and compile the smart contract locally. Once deployed, smart contracts will be placed into a specific address in the blockchain. Users can then interact with these smart contracts by submitting a transaction into this address, which will execute a function defined on the smart contract. By default, smart contracts can't be deleted after deployment and every interaction with them is irreversible. (Ethereum, 2022f).

The difference between smart contracts and traditional backends is that whereas a developer is always able to make adjustments to the source code of the backend and push it into the deployment pipeline to update the backend, smart contracts can't be modified or deleted once they have been deployed. However, there are some ways to bypass this restriction, but overall, it isn't as straightforward and effortless as updating the backend of a centralized application. This increases the importance of certain aspects of the software development process, such as testing. While testing is an important part of software development by default, an increase in what deploying a faulty software costs should make the developers to be even more motivated in ensuring the correctness of the software they are about to deploy. Another aspect of the development process that differs greatly is deployment process. For centralized applications, deployment pipelines have been around for some time, and they have had the time to mature and improve their tools and practices. As non-cryptocurrency Dapps are just starting to get attention, it is understandable that the different development tools and practices have yet to be matured, and it is expected that there is room for improvements in the currently used tools and methods.

The aim of this study is to examine which tools and functionalities an Ethereum developer needs to have in their development environment to develop a high-quality software product efficiently and evaluate how well the currently existing solutions fulfill these requirements.

1.2 Research questions

The paper answers to the research problem through the following research questions:

RQ1	What are the requirements for developing and deploying smart contracts into the Ethereum blockchain and how can those requirements be categorized?
RQ2	How can a development environment fulfill these requirements?
RQ3	How do the existing development environments meet the requirements?

The purpose of these research questions is to divide the research problem into smaller, well-defined parts, so the research problem can be answered comprehensively.

1.3 Research methods

Two different research methods are used for different parts of the study in an attempt to produce objective results that can be considered to be scientifically valid. The first method is used in the making of the second and third chapter, in which the current state of Ethereum development is examined to provide a solid base for the upcoming Ethereum development environment evaluation. The literature review is conducted by using a semi-systematic approach with a snowballing technique, in which additional papers to those that are found during the systematic process, are identified by looking at references and citations of the already included papers (Wohlin, 2014).

The second used research method in this study is called DESMET, which was developed to evaluate software development tools and processes as objectively as possible. DESMET is designed to evaluate the tools and processes of a specific group, when they are performing similar tasks under similar conditions. The evaluation is based on an assumption that there are several alternative ways to execute the same task, and the aim of DESMET is to identify the best available option. A tool or method is evaluated based on the features it offers, the characteristics of its supplier, and the amount of training it requires. The qualitative research method of DESMET is called feature analysis, and the purpose of it is to identify requirements for a particular task and then map those requirements into features that the tool or method should possess. (Kitchenham, 1996a)

In this study, DESMET's feature analysis is slightly modified to fit the evaluation context better and to produce more accurate results. The made modification is about including the level of importance of each feature within the evaluation criteria by adjusting the maximum amount of points each feature will reward if the development environment provides a full support for that feature. How the feature set and the evaluation criteria are created is discussed in chapter five, and the experience and the evaluation of the success of modifications are examined in chapter eight.

1.4 Literature

The literature for the study is gathered using a semi-systematic literature review process, along with the snowballing technique. The vast majority of the scientific literature is identified using the semi-systematic literature process and the snowballing technique, but a few scientific papers were found conducting searches to the academic databases (e.g., IEEE, ACM, Google Scholar) with a specific search string. A paper by Zou et al. (2021) provided valuable information about the challenges that Ethereum developers are currently facing. However, as the selected topic of this study is advancing with great speed, it is necessary to also include information from Ethereum's documentation to ensure that the used information is up to date.

The aim of the semi-systematic literature review is to identify scientific articles that discuss the challenges of developing Ethereum-based products. The inclusion-, exclusion-, and quality assurance criterions for the papers included during the semi-systematic process are listed in chapter three.

During the feature analysis, once the requirements towards the development environment has been identified and the feature set is created following the literature by Kitchenham and Jones, the documentation of the development environment under evaluation is used heavily, as the main selected method to conduct the feature analysis (i.e., feature screening) is done by examining only the documentation of the tool or a process under evaluation.

1.5 Boundaries

The study will focus on inspecting and analyzing the developer's requirements towards a development environment when developing Ethereum-based products. The reason for focusing on Ethereum is that it is the largest blockchain that supports smart contracts by market capitalization, which is calculated by multiplying the current price of the blockchain's cryptocurrency with the circulating supply. Ethereum's market capitalization is about 215 billion euros, while the BNB's, which is second on the list, market capitalization is less than 50 billion euros. In short, Ethereum is the biggest blockchain platform by a large margin for the developers to develop smart contracts and Dapps. (CoinMarketCap, 2022)

This study is not about exploring Ethereum or smart contracts as a technology, nor does it inspect or evaluate in which situations choosing to use smart contracts would be a good decision. The aim is to identify the requirements and list them as a feature set and create an evaluation criteria for the ability to evaluate some of the currently existing Ethereum development environments. The created feature set does not take web-based development environments into consideration.

1.6 Overview of the results

The created feature set contains 40 features, which are categorized according to the ISO/IEC standard 25010:2011, which determines the characteristics of a high-quality software. The categories are functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability.

Overall, all the evaluated Ethereum development environments (i.e., Hardhat, Truffle, and Brownie) performed quite similarly across the feature set, but there are big differences on some of the features. Hardhat performed the best on the evaluation scoring 136 points out of the 164 available points, which translated to around 83 percent. Truffle came second with a score of 127, which translates to around 77 percent, while Brownie scored 114 points, which in turn translates to around 70 percent.

2. Ethereum blockchain

This chapter explores the different aspects of Ethereum and how it operates, to provide a context of the selected blockchain platform of the study. However, as the technical implementation of Ethereum isn't the focus of this study, the topic is examined in a simplified manner.

Ethereum is a form of blockchain and at its core, Ethereum is a transaction-based state machine. The state can hold information for example about accounts and their balances, and as long as information can be represented by a computer, it can be included in the Ethereum state. (Wood, 2014)

Hartel et al. (2019) describe blockchain as a peer-to-peer database, which contains the history of every transaction that has happened thus far. Zou et al. (2021) explain blockchain to be a chain of records called blocks, that contain some transaction data, the hash value of the previous block, and a timestamp of the exact moment when the block was mined and validated. These blocks are secured by using cryptography, and they form a chain as each time a new block is mined and validated, it is being linked to the existing chain of blocks. Blockchain can be understanded as a public ledger, where a part of all transactions is stored in each block. Unlike the traditional web hosting, in which the service is accessible on a specific address, the blockchain is stored on a network of nodes, which all hold a copy of the blockchain.

Wang et al. (2019) elaborate that because blockchain has adopted the peer-to-peer protocol, the blockchain system can tolerate a single point of failure. In other words, blockchain won't lose any data as long as the blockchain is running at least on a single instance (i.e., node). The consensus mechanism is used to ensure that all nodes hold the same identical version of blockchain. This is the core idea of blockchain, and because of the utilization of this mechanism, blockchain is considered to have characteristics such as decentralization, integrity, and auditability.

Blockchains can be divided into two categories: public and non-public blockchains. Non-public blockchain grants access to new users (nodes) to join the blockchain, whereas anyone can join a public one. Ethereum, the selected blockchain for this study, is an example of a public blockchain. (Zou et al., 2021)

The most popular use case of blockchain technology is decentralized finance (DeFi). DeFi is an umbrella term for all the financial functionalities that utilizes blockchain technology and in the case of Ethereum, it can be split into four layers: the Ethereum blockchain, the assets (i.e., cryptocurrencies such as ETH), the protocols (i.e., smart contracts that provides functionalities), and applications that enables users to use previously mentioned layers. DeFi is considered to be a new financial system that was built for the internet age, and it promises to be better than traditional finance by being an open platform and giving you the control and visibility of your money. It enables access into global markets with just an internet connection and there aren't any centralized authorities that could block payments or limit your financial access. (Ethereum, 2022a)

Figure 1 displays Ethereum's behaviour in a simplified manner to create an overview about the upcoming topics, and to show the purpose of each element in the system.

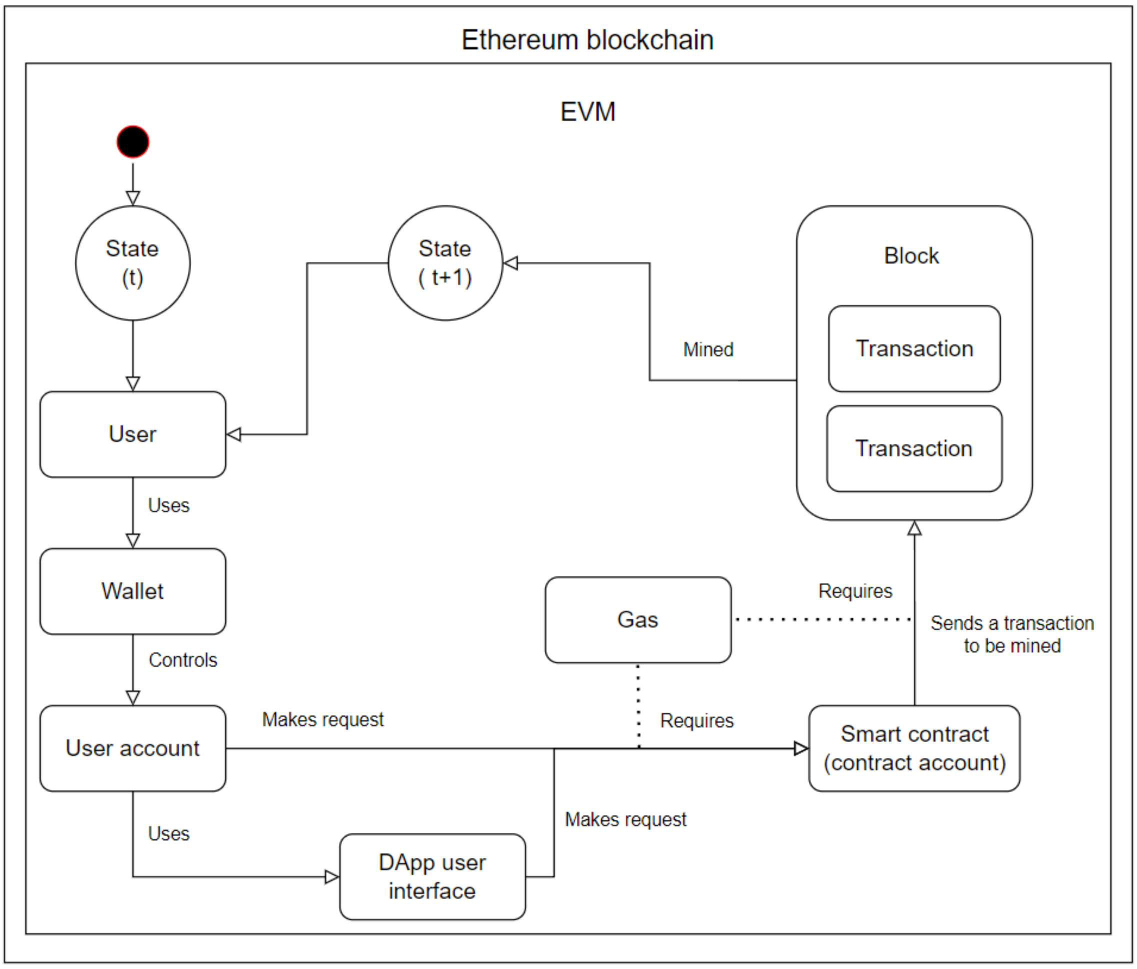

Figure 1. How Ethereum operates in a simplified format.

As shown in Figure 1, the start of the EVM's operating loop with a certain state. Once users have created requests through using wallet and user accounts, new transaction requests are created and sent for the miners to include them in a block. Once the block is mined and verified by all the nodes in the network, a new "canonical" state is created and agreed.

2.1 Ethereum virtual machine

Ethereum virtual machine (EVM) is a single entity, which is maintained by all the computers running the Ethereum client. Ethereum is a distributed state machine, and its state holds all the accounts and their balances, as well as a machine state, which can vary between the different blocks according to the predefined rules. (Ethereum, 2022b)

Both Hwang and Ryu (2020) and Zou et al. (2021), among many others, explain that Ethereum is one of the most popular blockchain platforms because it provides a decentralized Turing-complete virtual machine called Ethereum virtual machine (EVM). Hwang and Ryu (2020) continue by explaining that EVM can support on top of cryptocurrency also other types of applications, like games, by using smart contracts.

Like Bitcoin, Ethereum is also a cryptocurrency backed blockchain platform and it is also the second most valuable cryptocurrency after Bitcoin. The main innovation of Ethereum is the introduction of smart contracts, which are small computer programs stored on the blockchain. (Gupta & Shukla, 2019)

2.2 Ethereum accounts and wallet

Ethereum wallet is a tool that is used to manage Ethereum accounts. It enables the user to check the balance of the account, make new transactions, and connect to applications. Wallet isn't bound to the account, and one wallet can manage several different accounts at the same time. There are multiple types of wallets, which however have the same functionalities. Wallets can be either in a physical form (e.g., USB stick), a mobile application, a web wallet accessed via web browser, or a desktop application. (Ethereum, 2022c)

Unlike every blockchain, Ethereum has different accounts for different users on their blockchain. Account can be either an externally owned account (EOA) or a contract account. What separates these from each other is that the EOA account is controlled by a private key, and it has no associated code, while the contract account has an associated code, which will be executed when the account receives a transaction. Users can initialize transactions only through an EOA account. EOA account will send a transaction, which can include Ether and binary data as a payload. If the receiving account is a so-called zero-account, a new smart contract will be created based on the transaction payload. However, if the account receiving a transaction is a contract account, the account is activated and the associated code of it will be executed in the local EVM, using the transaction payload as an input. After this the transaction is sent to the blockchain network, where it will be verified by miners. (Wang et al., 2019)

2.3 Gas

As executing smart contracts requires computational resources, Ethereum discourages unnecessary smart contract usage by charging a fee called gas. Gas fee is charged for every use of computation, which includes use cases such as executing functions on smart contracts, making transactions, and creating new smart contracts. (Wood, 2014)

Chen et al. (2020) explain that Ethereum adopted the gas mechanism also for preventing denial-of-service (DoS) attacks, which in this case would aim to exhaust the computing resources of blockchain's nodes. The gas mechanism charges the transaction sender for deploying or invoking a smart contract. The size of the gas fee depends on the executed operations of the smart contract, or when deploying a new contract, its size itself determines the cost.

Zarir et al. (2021) describe gas as a measurement unit, which indicates the amount of computational work in the Ethereum blockchain. The amount of computation that is required to execute a transaction is called gas usage. The gas usage depends on the number of functions and their complexity, and on the size of data that is stored in the blockchain. Every transaction needs to be set up with two parameters before they can be triggered. These parameters are the gas limit and the gas price. Gas limit sets the maximum amount of gas that is allowed to be used for executing the transaction. Setting the gas limit too low can cause an out-of-gas error while the transaction is being executed. The gas price is given as a second argument, and it represents the per-unit price of gas, and it is given in a format of Ether (ETH) cryptocurrency. The total transaction fee that is paid by the transaction issuer is calculated by multiplying gas usage by gas price.

The above explained calculation format by Zarir et al. (2021) for calculating transaction fee was changed in August 2021 with the Ethereum's London upgrade. The motivation for the change was to make estimating the transaction fee easier. After the upgrade every block on blockchain has a base fee, which represents the minimum price of a gas unit to be included on the block, and it is calculated automatically by the network. Once a new block is mined, the base fees of it will be "burned" and removed from the circulation. Priority fee (i.e., tip) is used after the London upgrade to motivate the miners to include transactions into the blocks, because without the tips, it would be economically beneficial for miners to mine empty blocks. This same mechanic can be used to prioritize transactions. The larger the tip, the more likely it is to outbid other transactions and be included in the block. Wallet providers will automatically generate a suggestion of transaction fee, which includes both a base fee and a priority fee. Users have also been given an option to include a parameter, which indicates the maximum price they are willing to pay for the transaction to be executed. This limit must exceed the sum of the base fee and the tip. The difference between the max fee and the sum of base fee and tip is returned to the transaction sender. (Ethereum, 2022d)

2.4 Decentralized applications

Blockchain-powered applications (i.e., decentralized applications) use smart contracts for executing the requests created by the user through the frontend in a web browser (Zarir et al., 2021). For example, if the user wants to purchase a certain digital asset from an application that uses smart contracts to execute the business logic, the request is sent to the smart contract located in the blockchain, which will execute the request.

Decentralized applications differ from the traditional web applications due to the usage of smart contracts. Some benefits that using smart contracts provides are having "backend" constantly up and ready to execute incoming requests, having an increased privacy and zero censorship, and a complete data integrity. These improvements come naturally with some drawbacks, such as maintaining and scaling the application being harder, as the Ethereum network can process only a limited number of transactions per second. (Ethereum, 2022e)

2.5 Smart contracts

Smart contracts are low-level code scripts running on blockchain and their purpose is to facilitate a contract between two parties without the need of the trusted third party. In other words, a smart contract is a program that contains data of some sort and code, which is automatically executed when a certain pre-condition is met. (Zou et al., 2021)

Zhang et al. (2020), and Gupta and Shukla (2019) explain that smart contracts have their own addresses on the blockchain and are executed according to their logic by the EVM. The smart contract is invoked by sending a transaction to its address. Gupta and Shukla (2019) continue to explain that as the smart contract's source code is stored on the blockchain, it can be viewed and verified by anyone, and it also becomes tamper-proof as the blockchain is immutable. This characteristic makes smart contracts useful for different types of applications, and for example smart contracts have been used to address concerns regarding security and privacy on connected vehicles.

Blockchain technology is evolving rapidly, and cryptocurrencies are no longer the only applications that the technology is being used for (Pinna et al., 2019). Hartel et al. (2019)

found in their study that smart contracts are used for multiple different types of applications, while the biggest categories are finance and entertainment. However, the authors noted that the suitability of smart contract technology is being tested for different use contexts, even when the traditional software approach might even achieve a better result. Authors name health, energy, identity management, and other utilitarian topics as fields, in which smart contracts are being utilized at.

Bytecode of every deployed smart contract is always available for everyone, as they are stored in a public blockchain. The problem arises from the fact that the bytecode is not readable for humans, so the availability doesn't actually improve the trust of users or anything else. However, the developers have an option to publish the source code, so the users can read the smart contract by themselves. Authors name Etherscan.io as an example of a third-party verification service provider, which compares and verifies the source code and the bytecode to match. This verification confirms that the source code provided by the authors is the same source code that the bytecode is compiled from. (Pinna et al., 2019)

Zou et al. (2021) brought up how blockchain's each node needs to execute and validate every smart contract transaction. This reduces the privacy of smart contracts as all relevant transactions are being visible to on the blockchain and the gas costs of executing smart contracts can be high, especially if the smart contract contains complex computation.

It was widely reported in April 2022 that a DeFi project called Beanstalk was robbed of 182 million US dollars. The attacker used a flash loan attack, in which the attacker takes a so-called flash loan and buys an absurd number of tokens. As Beanstalk uses a proof-of-stake consensus mechanism, by buying enough tokens to have over 50% of the voting power, the attacker was able to transfer money from the protocol's liquidity pools to their own account. After this transaction, the attacker sold the tokens and returned the flash loan. All of this was done in 13 seconds. As it is being said, blockchain technology is still growing and improving.

3. Smart contract development

This chapter examines the smart contract development process and its challenges.

A smart contract is first developed by using a high-level programming language before it is compiled into EVM bytecode. Once the smart contract is compiled, it can be deployed into the Ethereum blockchain by a transaction, which requires gas. (Gupta & Shukla, 2019; Pinna et al., 2019; Zhang et al., 2020; Zou et al., 2021)

Ethereum smart contract developers can use different programming languages, such as Solidity and Vyper, to develop complex smart contract applications. These two languages are also the most active and maintained languages that are being used to develop smart contracts on Ethereum. (Ethereum, 2022h; Zou et al., 2021)

Solidity and Vyper have been developed with different intentions. Solidity is an object-oriented, curly-bracket programming language with a lot of functionalities, while Vyper is a pythonic language, which aims to be more secure and easily auditable by providing less functionalities. These languages are designed to be used by the majority of smart contract developers, but for advanced developers there are languages called Yul and Yul+, which are described to be simplistic and functional low-level languages. The main benefit of using these languages is the ability to optimize the gas usage of the smart contract. (Ethereum, 2022h)

There are multiple ways to attack against smart contracts and the importance of protection against these attacks is high, as smart contracts are used in security-critical applications like financial transactions. While Solidity is the most supported and maintained language for smart contract development, it has had multiple severe security issues in the past. (Hwang & Ryu, 2020)

Figure 2 displays a smart contract development process in a simplified manner to create an overview of the process.

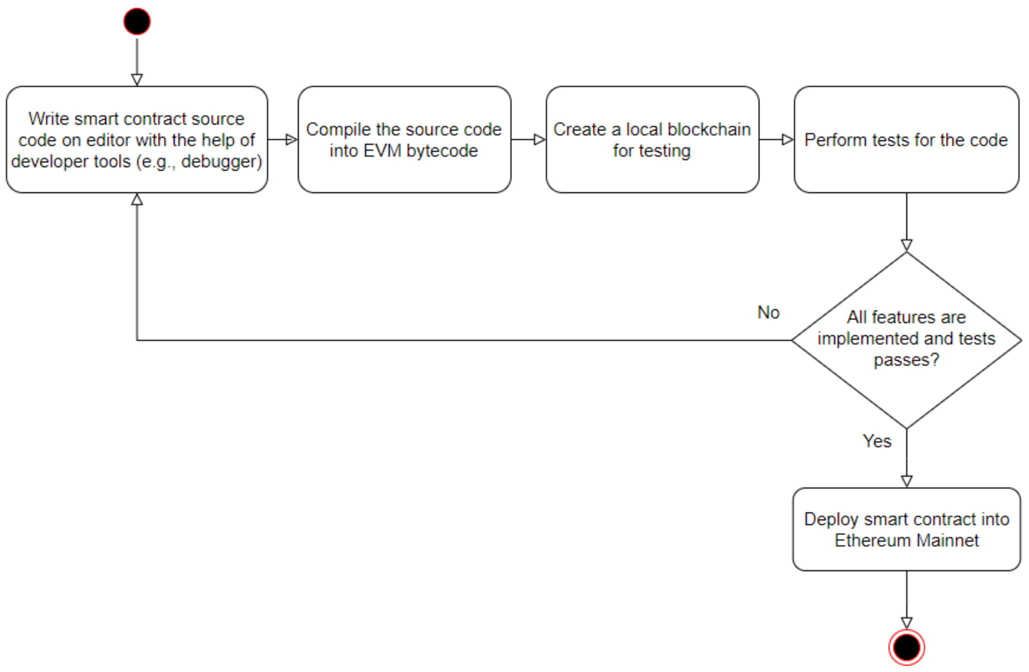
Write smart contract source code on editor with the help of developer tools (e.g., debugger)
Compile the source code into EVM bytecode
Create a local blockchain for testing
Perform tests for the code
All features are implemented and tests passes?
No
Yes
Deploy smart contract into Ethereum Mainnet

Figure 2. A simplified smart contract development process.

As shown in Figure 2, a typical smart contract development process contains a number of iterations, in which a new feature is added to the smart contract. First, a new source code is being written or already existing code is changed with the help of different tools, such as linters (i.e., static code analysis tools, which is used for example to discover errors in the code and ensure the same styling across the project) and debuggers. After this is done, the source code will be compiled into EVM bytecode, which is used for testing. If testing exposes some bugs or unwanted behaviour, the cycle will start again from the start. Once all the tests pass and there is no feature left to be implemented, the smart contract is ready to be deployed into the Ethereum mainnet.

It is recommended to use a development framework for developing smart contracts, especially if the developer is inexperienced on smart contract development, as the aim of the development environments and frameworks is to assist the developer during the development process. Building a Dapp requires the usage of a lot of different technologies and tools, and frameworks try to solve the problem of researching and selecting the tools by hand. Frameworks include different tools by default and provide a plugin system for the user to include external tools if they wish. These framework tools can have functionalities to compile and test smart contracts, create a local blockchain for development, and enable parallel frontend development. Ethereum's documentation highlights the most popular development frameworks and based on the stars given on GitHub, the most popular ones are Truffle (13,213), Scaffold-ETH (7,146), Embark (3,623), and Hardhat (3,332). (Ethereum, 2022g)

3.1 Challenges of smart contract development

As smart contract development, along with the whole IT field, is advancing quickly and the best practices are still being tried to be identified, it is understandable that the smart contract development process contains a lot of challenges that needs to be solved for blockchain technology to advance and to become fully utilized.

Zou et al. (2021) conducted a study to identify major challenges and desired improvements in smart contract development on the Ethereum platform. Authors conducted 20 semi-structured interviews with smart contract developers and formed a survey, which was then sent for other smart contract developers to validate their findings during the interviews. They received a total of 232 answers for this survey. The biggest overall challenge was guaranteeing the security of smart contracts, followed by the lack of powerful development tools.

Challenges related to debugging, gas optimizing, and code reviewing were that it was difficult to debug smart contract during the development, gas optimization was always painful especially if the smart contract contained complex logic, a lack of qualified smart contract developers who are able to find security flaws in code, and code reviewing being very time consuming. At least 63.4% of survey answerers agreed with these challenges.

The most agreed challenges regarding smart contract testing were the difficulty to consider all corner cases and scenarios, the potential of undiscovered flaws in compilers and virtual machines, and a lack of mature testing frameworks that other languages have. At least 49.5% of answerers agreed with these challenges.

The most wished improvements on Solidity were an increased number of general-purpose libraries, more powerful functions for error logging and reporting errors, more standard interfaces, and better support for checking the security of data types.

Authors noted that developers would be significantly more likely to rate a lack of online learning resources and community help as a major challenge if they have little experience on smart contract development.

Zou et al. (2021) summarized their results in five points. First was that as code security is a high requirement for smart contracts, a new tool is needed for ensuring this as developers don’t currently have an effective way to ensure the code security. Second point was that current debugging tools are primitive and inefficient, and using them in practice is very painful. Developers wished for more powerful debuggers, which would provide more informative error messages during debugging. Third point was that the Solidity, compiler, and EVM have undesired characteristics and because of them, it is difficult to develop smart contracts effectively and efficiently. Fourth point was the need for gas-estimation and optimization tools at the source code level, which would also consider readability of the code. Fifth point was a lack of development resources, such as best practices, code examples and third-party libraries.

Chen et al. (2020) found that a lot of deployed smart contracts contain multiple different types of gas-inefficient code, which by their definition means that a smart contract requires more gas than necessary. Authors explain that the cause of it is gas-inefficient patterns in smart contract’s source code. Authors believe that there are various reasons why gas-inefficient patterns exist on such a large scale on deployed smart contracts. They name reasons such as inexperienced developers, unawareness of the gas waste, and the absence of specialized tools for the task.

Hartel et al. (2019) concluded from the received survey answers that calculating the gas estimation for smart contracts is too challenging. Developers also expressed their need for integrated support for distributed storage, as it is not desirable or practical to store smart contract’s all data on the blockchain.

Hwang and Ryu (2020) examined 55,046 live smart contracts to see if the Solidity smart contract developers were aware of the importance of applying the Solidity security patches to their smart contracts. The conclusion was that they are not, as 98.14% of the examined smart contracts hadn’t applied the security patches for known security vulnerabilities, and they detected 13,943 smart contracts that are potentially vulnerable because of this. What’s even worse, the authors found hundreds of exploitable smart contracts of which about a quarter had thousands of users. The authors gave a recommendation for the developer to use the latest Solidity compiler to apply the security patches, and for the Solidity team to release more security patches for known vulnerabilities.

Sujeetha and Preetha (2021) conducted a literature review on smart contract testing and analysis to emphasize the positives and negatives of the smart contract development process. In the study authors identified two main challenges for smart contract testing, which were the lack of best smart contract development practices and the lack of specialized tools.

Hartel et al. (2019) observed from the survey they conducted that a common complaint about Ethereum smart contract development was that the Ethereum technology wasn’t mature enough, which was shown in the quality of the development tools. Authors also found that open sourcing and formal verification increases the credibility and popularity of the smart contract.

Overall, it has become clear that one of the biggest challenges on Ethereum smart contract development is the immaturity of the system like Zou et al. (2021) and Sujeetha and Preetha (2021) among others explain. It is natural, especially in software engineering, that it takes time to develop a standardized way of working. However, at the moment the developers desperately need improvements on documentations and tools (e.g., debuggers, gas optimizer, etc.), ways to ensure the security of the contract, and overall the smart contract development to mature.

3.2 Development tools and practices

This chapter introduces the commonly used smart contract development tools and practices. At the start the characteristics and the purpose of the tool or practice is explained, and at the end is the conclusion about the state of the tool or practice.

3.2.1 Analysis tools

The purpose of the software analysis tool is to read the source code of a program and then make some modifications or report about some problems within the code. For example, an analysis tool can be used to discover errors or unused parts (e.g., functions, variables) in the source code.

In their study Zhang et al. (2020) evaluated among other things the state-of-the-art smart contract analysis tools. During the study authors noticed that the bytecode-based tools often become unusable when EVM was updated. The reason for this is that there is often a lack of motivation to update and maintain smart contract development tools, and this leads to limited availability of this kind of tools. Despite this, the authors found that bytecode-based analysis tools have usually a higher precision of identifying a bug than their counterparts, because they usually use techniques like control flow analysis, which is used to determine the control flow of the software. As a result of their study, the authors recommended using Mythril, Slither, or Remix for smart contract analysis. In their study Mythril had the highest precision rate (i.e., how many of the detected bugs are relevant), Remix had the highest recall rate (i.e., how many relevant bugs were detected), and Slither detected most different kinds of bugs while having a good recall and precision rate. Authors also praised the easy installations of these tools and claimed that they were also very convenient to use.

Di Angelo and Salzer (2019) analyzed 27 different Ethereum smart contract development tools. These tools were analyzed from the aspects of availability, maturity level, used methods, and how they detect security issues on smart contracts. During the study authors concluded that tools tend to persist when they are being developed as an open-source project. As a result, authors highlighted five tools that they found inspiring. Two out of the highlighted five tools are still being actively developed as an open-source project, named Mythril and KEVM. Authors described Mythril as a prime example of a smart contract security analysis tool, which also has a high usability. KEVM was appraised as a favorite tool for formal verification of the smart contract due to its maturity. Authors explain formal verification as using formal methods (i.e., mathematical techniques) to ensure that the system works as intended. However, authors note that the drawback of the tool is that it requires a lot of expertise to use it.

Hartel et al. (2019) found that a majority of smart contract developers who answered their survey used or had used some sophisticated experimental smart contract development tools, such as Echidna, MythX, SmartCheck, and Slither. However, the authors report that many of the respondents expressed their opinion that the tools of development environments such as Truffle framework and Remix are more important than the sophisticated experimental tools. The reason for this was that the standard tools of different development environments already have enough shortcomings. Authors concluded that this indicates a lack of maturity in the available smart contract development tools.

In their study Gupta and Shukla (2019) noticed that as they expected, static smart contract analysis tools performed much faster than symbolic execution tools. Authors also noticed that there are on average about five contracts inside a single smart contract, and tools should take this fact into consideration when analyzing smart contracts. They also observed that many active smart contracts were compiled using an outdated version, and a lot of the code didn't follow the good practices. Overall, the authors observed that improper coding practices are the biggest issue in smart contract development.

It appears that there is a decent amount of analysis tools for developers to select from. In general, this appears to be a good thing, but the cost of having multiple options might be that it can take more time to mature into an excellent tool. However, Mythril and Slither appear to be the most praised options of analysis tools, and at the time of writing this, both tools are still popular and doing well.

3.2.2 Readability

In software engineering, readability refers to how easy it is for the developer to read and understand the source code of the program. Martin Fowler has a saying which describes readability well:

> *"Any fool can write code that a computer can understand. Good programmers write code that humans can understand." (Fowler, 2018, p. 22)*

Canfora et al. (2021) argued that since readability of source code is a key factor in software quality and heavily affects code maintainability, portability, and reusability, smart contracts would benefit from a tool, which would ensure better code readability. This is emphasized by the fact that smart contract developers often inspect and reuse parts from other smart contracts. Authors explained that the poor readability of smart contracts could be caused by the attempt to make the source code as efficient as possible to reduce the gas costs, which in practice means shorter source code.

Pinna et al. (2019) found that the publicly available source codes of smart contracts are usually well commented. As reusing the already existing code is a common practice in smart contract development, authors believed that the availability of well documented source code is a great asset for new developers to understand the smart contract development.

Readability in Ethereum smart contracts is a difficult topic as by improving the readability, it is likely to come with a price of increased need of gas. This is understandably a price that developers might not be willing to pay.

3.2.3 Code reuse

Code reuse means reusing already existing source code in another context/project. This can be understanded as using already existing knowledge, and as the old saying goes, there is no point to reinvent the wheel. There is a lot of reasons for software developers to reuse code, such as it shortens the time to implement new features to the application, it frees resources for other development tasks, and reusing code that implements features that the developer is not too keen about implementing frees time to focus on those more interesting features. (Haefliger et al., 2008)

Chen et al. (2021) studied code reuse in smart contracts. They explained that code reuse might predispose smart contracts to severe threats, such as security attacks and resource wasting. Authors defined the reused code blocks as “subcontracts” in their study. They collected 146,452 open-source Ethereum smart contracts projects for their analysis. The result was that code reuse is quite frequent in smart contract development, as over 26% of subcontracts are reused at an average of 14.6 times, and over 91% smart contract projects reuse more than one subcontracts. Authors found that the most reused subcontracts were related to ERC20 token applications, and that developers don’t tend to make major modifications to the reused subcontracts.

Pierro and Tonelli (2021) also studied source code duplication in smart contract development by analyzing 7500 smart contracts, which had been deployed in the Ethereum blockchain. They found that about 80% of the analyzed smart contracts contained code, which was copied from another deployed smart contract. Authors believe that the code cloning is caused either by the desire to copy successful smart contracts, or by the lack of package manager, which could import external dependencies into the project.

Gao (2020) had similar results about the code reuse in smart contracts on the Ethereum ecosystem. In the study over 22,000 smart contracts were gathered from the Ethereum blockchain and analyzed for code reuse, and the result was that the clone ratio of solidity code is about 90%, which is much higher than on the traditional software.

Gupta and Shukla (2019) also noticed that a high percentage of smart contracts are reused code. In their study they noticed that only 4.7% of smart contracts written in Solidity have a unique bytecode. Authors noted that this implies that reusing code is very common practice in smart contract development, and it emphasizes the importance of good security practices, because security vulnerabilities will transfer with the reused code to other smart contracts.

The results throughout the studies are similar; code reusing is a highly used method in smart contract development. Such a one-sided conclusion can be accepted as the studies had analyzed a great number of smart contracts, and the reasoning for it is well understandable. As a great number of smart contracts deal with different tokens as Chen et al. (2021) explained, one of the main benefits to reuse code is to free resources for developing other aspects of the software. It is easy to see that a developer would like to reuse existing code to implement a token within the smart contract to serve some purpose for the actual business logic of the smart contract. By not having to implement the used token system from scratch, developers can spend that time to implement those features, which will make people actually use the smart contract (most often through a frontend). However, the code reusing also creates problems, such as how to ensure the security of the reused code, as the authors pointed out.

4. Research methods

To produce as objective and scientifically valid results as possible, the study uses two research methods. Firstly, a semi-systematic literature review is conducted along with the snowballing technique to examine the current state of the art of Ethereum smart contract development. The second research method is a DESMET feature analysis, in which the user requirements for performing a certain task or activity under specific circumstances are identified, and then mapped to different features that a method or tool designed to support the requirement should possess (Kitchenham, 1996).

4.1 Literature review

Snyder (2019) describes a consideration of prior, relevant literature to be an essential part of every research project. The author continues by stating that the purpose of a literature review is to map and assess the research area to provide justification to the study and justify the research questions and hypothesis. Danson and Arshad (2014) worded this a bit differently, as they believe a literature review is conducted to educate oneself about the topic area before shaping an argument or justification in the study.

Danson and Arshad (2014) explain that there are different types of literature reviews, such as traditional-, narrative-, systematic-, and meta-analysis literature reviews, and they differ in their methodologies to provide options for studies to choose from as these different literature reviews produce different kinds of results. Authors define narrative- and systematic literature reviews to be the most dominant styles of literature reviews and explain their difference to be that whereas a narrative literature review has a variety of styles and methods, a systematic literature review has a highly structured approach towards reviewing existing literature, and it excels in identifying all the existing literature and ensuring that no existing literature or knowledge is missed during the literature review. Authors claim that the common aspect between these two review styles is the critical approach towards the literature. Snyder (2019) provides a slightly different version on the approaches to literature reviews. Author identifies three broad types of methods to conduct a literature review, which are systematic-, semi-systematic-, and integrative approaches. The typical purpose of a systematic approach is to synthesize and compare evidence, while the research question is specific. The purpose of the semi-systematic approach is to overview the research area and map out how it has developed over time, while the research questions aren't usually too narrowed. The integrative approach focuses on critiquing and synthesizing the literature, and the research questions can be either narrow or broad. The names of different approaches describe the nature of their search strategy well, and whereas a systematic approach deals with quantitative articles, the integrative approach is more focused on qualitative sources, which can be in the form of a research paper, a book or in another form of a published text. Systematic approach can produce evidence of effect or inform policy, whereas integrative approach can be used to create a theoretical model or a framework. Semi-systematic approach is described to be between the systematic- and integrative approaches, and it is used to produce a state of knowledge or a theoretical model (Snyder, 2019).

A purely integrative approach towards literature review would be a quite natural solution, as it was examined by Snyder (2019) to produce theoretical models or frameworks. As the topic area of this study is developing rapidly and the number of relevant scientific articles is relatively speaking low, the literature review must also include other types of qualitative sources than scientific articles. However, this shall not neglect nor weaken the objectivity of the literature review. As Budgen and Brereton (2006) wrote, both research papers and PhD theses in software engineering have rarely included a clear and systematic process to objectively collect the literature, and authors have rather just picked the parts that support the claims of their work. Kitchenham and Charters (2007) continue by stating that unless a literature review is explained thoroughly and presented in a fair way, it holds little scientific value. On top of the increased objectivity, reviewing literature with a systematic process may provide other benefits as well. Niazi (2015) found that having a formal literature review process leads to having additional high-quality primary studies, which wouldn't be found otherwise, and an increase in effectiveness in many aspects, such as data extraction, synthesis, and overall literature management. However, the informal literature review was able to identify more articles than the formal literature review, and the author recommends having a formal literature review with the snowballing technique, which Wohlin (2014) describes to be identifying additional papers by using the reference list of a paper or looking the citations to the paper. At the beginning of the snowballing procedure, a starting set of papers is identified by going through a tentative set of papers with inclusion- and exclusion criteria. After this, each paper will be snowballed both backwards and forwards. Each reference and citation will be examined by the title, publication venue, and authors of the paper, and every promising paper will be then examined more deeply by reading first the abstract, and eventually the relevant parts of the whole paper to find out if the paper can be included in your study.

Based on these reasons, this study will benefit from using both systematic- and integrative approaches for different purposes, so the semi-systematic approach is selected to be used for conducting the literature review. As Snyder (2019) wrote, a potential contribution of a semi-systematic review can be a synthesized state of the knowledge, which is the role of the literature review in this study. The purpose of the literature review is to examine Ethereum development tools and practices. The literature review process is started by examining non-scientific sources, such as documentation of Ethereum, to identify different aspects of Ethereum smart contract development, which the literature review should cover. The second phase of the literature review is the systematic process, which is conducted after the search string, inclusion- and exclusion criterions, and quality assessments have been identified. The search string is developed by executing test searches into the databases and evaluating the relevance and the number of results. As the purpose of the literature review in this study is to provide background information for the upcoming research methods and synthesize the current state of knowledge in the field, rather than aiming for achieving a quantitative type of data, the goal is to gather a relatively small amount of qualitative data as a form of high-quality and relevant scientific papers. In practice this shows in the usage of a specific search string, which will produce a relatively low number of results, but a decent percentage of the papers is expected to be included in the study. The purpose of the usage of the systematic approach in this literature review is to ensure the objectivity in the results and provide transparency and reproducibility for the literature review process. The third phase of the literature review is the usage of snowballing technique, in which additional sources are tried to be identified from the citations and quotations from the main sources of the study. The last phase of the literature review is ensuring that all the terms identified during the first phase have been covered. If they haven't been, additional sources are used to explain them.

As the scope of the study is limited into examining the Ethereum development environment, which consists of different tools and processes, the focus of the systematic literature review process is on learning how the smart contracts are being developed, what kind of tools and practices the developers use, and what are the common challenges they are facing during the development process. Table 1 displays terms related to Ethereum that were identified during the initial search to be beneficial to explore during the literature review.

Table 1. Identified relevant terms to explain the nature of Ethereum.

Term	**Description**
Blockchain	A public database.
Ethereum	A specific blockchain technology.
Ethereum account	An account, which is used to make transactions in blockchain.
Ethereum wallet	Used to manage your Ethereum account.
Decentralized finance (DeFi)	The most well-known use case of blockchain technology.
Decentralized application (Dapp)	An application that utilizes smart contracts as its backend.
Smart contract	A code script located in a specific address on blockchain. Acts as a backend for decentralized applications.
Gas	A fee that the developer has to pay each time their smart contract is executed on blockchain.
Ethereum virtual machine (EVM)	A virtual machine inside each full node of Ethereum blockchain. Executes the smart contracts.
EVM bytecode	A low-level programming language used by EVM to run smart contracts. Compiled from a high-level language, like Solidity.

All the terms mentioned in Table 1 are explained in chapters two and three. The used search string was developed by conducting tests searches to the databases and evaluating the number and the quality of the results. The final search string was:

(Ethereum) AND (Smart contract) AND (Develop)*

All the words included in the search string were required to be found inside the abstract of the paper. After the search was conducted with the above search string, both the inclusion- and exclusion criterions were applied. These criterions are displayed in Table 2.

Table 2. An inclusion-, exclusion-, and quality assessment criterias.

Inclusion	**Exclusion**	**Quality assessment**
Published between 2019 and 2022	Published in 2018 or earlier	The paper must contain information about Ethereum development tools or practices
Peer-reviewed	Not peer-reviewed	The paper provides value for the literature review
Written in English	Written in another language than English	
Either a journal or a conference paper	Not a journal of a conference paper	
	Not finished	

As shown in Table 2, for the paper to be included it must be published in 2019 or after, be peer-reviewed and written in English, and it must be either a journal or a conference paper. On top of these, the paper must contain information about Ethereum development tools or practices, and it must provide value for the literature review.

The search was conducted on 2.3.2022. Displayed in Figure 3, the search string resulted in 266 papers on IEEE and 61 on ACM. After papers had gone through inclusion- and exclusion criteria shown on Table 2, 50 papers from IEEE and 7 papers from ACM were selected for full text inspection. After this the quality assessment described in Table 2 was applied. Each paper was examined to determine if they passed the quality assessment criteria, and if they included information that would benefit the theoretical chapters.

An overview of the literature that systematic process produced is that a majority of the papers deals with developing a tool on a specific use context. These kinds of papers don't offer too much information other than highlighting specific problems that are existing on smart contract development. The most useful type of papers were the ones, which had mapped the most popular and the most difficult problems to solve on smart contract development. For example, Zou et al. (2021) mapped the challenges by conducting semi-structured interviews and a survey. This paper provided valuable information about the currently existing challenges on smart contract development and that information is heavily relied on in this study.

As expected, the semi-systematic literature process left some gaps to the theoretical chapters. A lightweight version of snowballing for the paper by Zou et al. (2021) was conducted, and as the information gaps were mainly regarding Ethereum and not the development tools or practices, the focus was on finding this kind of literature. The Ethereum yellow paper by Wood (2014) was identified, and along with the help of Ethereum's documentation, these information gaps were filled. Including information from Ethereum's documentation also ensured that the most up-to-date information was used.

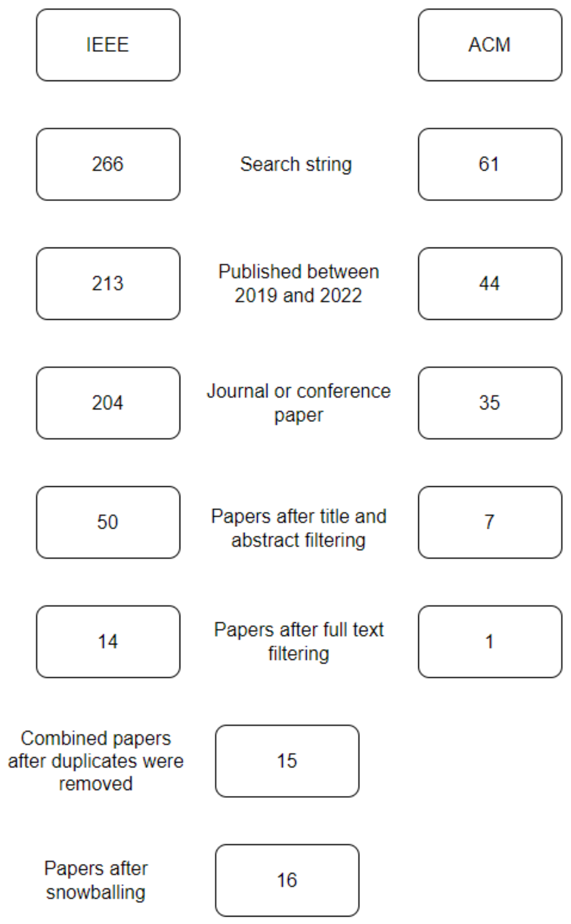
IEEE
ACM
266
Search string
61
213
Published between 2019 and 2022
44
204
Journal or conference paper
35
50
Papers after title and abstract filtering
7
14
Papers after full text filtering
1
Combined papers after duplicates were removed
15
Papers after snowballing
16

Figure 3. Number of papers in different during each step in systematic literature review.

As described above and shown in Figure 3, a total of 16 papers were used from the systematic part of the literature review. During the systematic literature review part, a paper written by Di Angelo and Salzer (2019) was identified. While conducting their study, the authors had made a perception of the state of smart contract tools studies. They had found that examining state-of-the-art smart contract tools from research papers has several shortcomings. First issue was that academic papers concentrate on the methods rather than on the tools themselves. Authors of these studies were presenting different tools on equal footing regardless of the state of the tool. This meant that proof-of-concept tools received the same attention as fully functional and accepted tools. Second problem was that these tool review studies relied often on the authors© previous studies. The last problem was that academic surveys tended to disregard tools outside academia. Authors give an example of an otherwise fine study, which had dismissed five smart contract tools, because these tools weren't accompanied by any academic paper, despite that information about these tools were available in other forms.

The experience from conducting the literature review for this study differs in some ways, but a lot of the literature that the used search string produced were exactly like Di Angelo and Salzer (2019) described. These papers were about implementing a tool for a specific purpose, with little to no consideration about the actual tool (e.g., usability, maintainability, portability etc.). However, the papers which compared different smart contract development tools did highlight Mythril and Slither as good analysis tools, which also appears to be a consensus within the smart contract development community.

4.2 DESMET

The DESMET method can be used to evaluate methods or tools that are being used by a software development team, which perform quite similar tasks under similar conditions, or by academic institutions in a context of experimental software engineering. The DESMET method was developed with a purpose to create an unbiased and reliable evaluation method, which maximizes the probability of identifying the best method or tool for the task. Making unbiased evaluations on the field of IT is a difficult task, as on top of the technical difficulties, sociological and managerial factors can play a role in the evaluation and it is quite easy to select a tool, which is not objectively the best choice for the task. The ideology behind DESMET is that there are multiple ways to perform a single software engineering task and the aim is to identify the best way for this specific circumstance. The best option might even change between the group of developers, who work on a similar task under similar circumstances (e.g., the developers in these groups have different preferences on their workflows). However, DESMET has also formal experiment evaluating methods, which are conducted under fully controlled conditions. These kinds of evaluating methods are expected to produce the same kind of results regardless of its organizer. (Kitchenham, 1996a)

As the purpose of the study is to evaluate development environments for Ethereum Dapps, the evaluating method must aim to produce results through an evaluation process that is as objective as possible, and which can be reproduced by other people. The DESMET evaluating method fulfills these requirements set for the research method and is thus selected to be used in this study.

4.2.1 DESMET evaluation

The DESMET method can be used to evaluate:

- A tool that is used to perform a specific task in software development.
- A method that contains a set of practices to execute a specific task.
- A generic method that contains a set of practices for a more generic software engineering related task.

A set of tools or methods can be evaluated together by evaluating them in the same way that they would be if they were to be evaluated as an individual tool or method. A tool-and-method -combination is evaluated either as a tool or method, depending on what the comparison will be done. Evaluation will be done as a method if the comparison is done between different paradigms, and as a tool, if the comparison is done within the same paradigms between different software support packages. (Kitchenham, 1996a)

The DESMET evaluation method can be used to measure an effect of a specific tool or method, or to measure how well a certain tool or method fulfills the needs of the development group on a specific task. Evaluations regarding the effect of a tool or process is referred to as a quantitative or objective evaluation, and these kinds of evaluations are based on developing measurable scales and conducting tests to evaluate whether a new tool or process delivers the expected improvements. Evaluations that are concerned about the appropriateness of a tool or method for a specific task are regarded as qualitative or subjective evaluations, and it is referred to as a feature analysis. It is also possible to mix both subjective and objective elements in evaluation, and DESMET refers to that kind of evaluation as a hybrid method. These three different evaluation types form the first evaluation dimension. The second evaluation dimension is formed by the different evaluation procedures, which are formal experiment, case study, and survey. In a formal experience, multiple people (i.e., software developers) perform a task using the tool or method that is being evaluated. A case study is conducted by investigating a tool or method while using it on a real software project, and in a survey, participants are people who have used the tool or method under inspection on previous projects and their task is to provide information about the evaluated tool or method. (Kitchenham, 1996a)

4.2.2 Evaluation methods

Kitchenham's (1996a) DESMET has quantitative-, qualitative-, and hybrid evaluation methods, and a total of nine different evaluation types:

1. *Quantitative experiment* evaluates the impact of a tool or process in a quantitative manner under formal conditions.
2. *Quantitative case study* evaluates the impact of a tool or process in a quantitative manner during a case study, in which the tool or process is used on a real software project.
3. *Quantitative survey* evaluates the impact of a tool or process in a quantitative manner through a survey, which is conducted after the participants have already used the tool or process.
4. *Qualitative screening* evaluates the impact of a tool or process in a qualitative manner, and it is done by a single person. Evaluation is done by first creating a feature list and the rating scale for the evaluation, and then conducting the rating process. This kind of evaluation is usually based on the literature of the topic rather than an experience of using the tool or method.
5. *Qualitative experiment* evaluates the impact of a tool or process in a qualitative manner through feature-based evaluation. The evaluators are a group of potential users, whose task is to test the tool or method in practice, and then rate the tool or method based on the feature list.
6. *Qualitative case study* evaluates the impact of a tool or process in a qualitative manner through feature-based evaluation after the evaluator has used the tool or process in a real software project.
7. *Qualitative survey* evaluates the impact of a tool or process in a qualitative manner through feature-based evaluation by people who have either used or studied the tool or method. Qualitative survey differs from qualitative experiment by giving the evaluator a choice, whether they would like to participate in the evaluation.
8. *Qualitative effect analysis* evaluates the impact of a tool or process as a subjective assessment in a quantitative manner by relying on the opinion of an expert.
9. *Benchmarking* usually evaluates the impact of a tool by testing it against other tools in standardized tests.

As we can see, DESMET provides a great number of choices of different evaluation methods to choose the best fitting one for your study. However, having multiple options raises the challenge of identifying the correct one, and Kitchenham (1996a) explains that the correct method depends on the maturity of the development group, and that task is for the evaluation organizer to execute. For this study, this will be done in the next subchapter.

4.2.3 Limitations

The DESMET method is intended to be used to evaluate one tool or method at the time, and it doesn't produce any information about how different tools or methods would interact, or what kind of results they would produce if they were used together. The exception for this is if hybrid methods are selected as an evaluation method (e.g., an opinion of an expert is used). (Kitchenham, 1996a)

For this study this limitation means in practice that the result of using a combination of different tools together can't be measured. However, this isn't the purpose of the study so this limitation can be disregarded.

Another limitation of DESMET is that the software development process needs to be standardized for the development group, or the results gained from the evaluation, which focused on a single software project, don't hold much of value. This is because the gained results are valid only if the software project uses the same software development process that was used on the project where the effect of the tool or process was evaluated. (Kitchenham, 1996a)

This limitation also doesn't concern this study, as the study is produced as a Master's thesis, and the selected evaluation procedure doesn't include evaluation on a real software project.

4.2.4 Selecting the evaluation method

Different evaluation methods require different things, both overall and from the software development team, in order to produce valid results. Kitchenham's (1996b) DESMET method uses seven different aspects to consider, which evaluation method would be the best option for your case:

1. Context
2. Expected impact
3. The type of the object under evaluation
4. Scope of impact
5. Maturity of the object under evaluation
6. Learning curve of the object under evaluation
7. Measurement capability of the organizer

Table 3 focuses on these aspects for determining which evaluation method should be used under different circumstances.

Table 3. Optional conditions for each evaluation method (Kitchenham, 1996b).

Evaluation method	**Favouring conditions**
Quantitative experiment	Benefits can be measured clearly Organization is willing to spend resources and has people available to participate in experiments Tool or method is used on a specific task Learning the tool or process doesn't take too much time Aim is to evaluate the tool or process in a specific context
Quantitative case study	Benefits can be measured in a single project Results are achievable before product retirement Standardized development process The performance of participants is measured beforehand to be able to compare results Evaluation timescale matches the length of typical software projects
Quantitative survey	Benefits can't be measured in a single project Data about productivity, quality, and tool or method from previous projects exists Experience of using the tool or method in a real software project
Qualitative screening	A lot of tools or methods to evaluate Limited time for evaluating
Qualitative experiment	Challenging to quantify achieved benefits Benefits can be observed from the result of the task Limited time for learning the tool or method The users of the tool or method vary greatly
Qualitative case study	Challenging to quantify achieved benefits Benefits can be observed during a single project Standardized development process Number of potential participants is limited Evaluation timescale matches the length of typical software projects
Qualitative survey	Challenging to quantify achieved benefits The users of the tool or method vary greatly Benefits can't be measured in a single project Experience of using the tool or method in a real software project, or intend to conduct projects to learn about the tool or method
Qualitative effects analysis	An expert opinion is available Development process is not standardized Necessary to combine the usage of different tools or methods Purpose is to evaluate generic tools or methods
Benchmarking	Tool or method doesn't relate to human-intensive tasks Results of testing can be ranked based on defined criteria

The evaluation method for this study is selected by eliminating those methods in Table 3 that have favouring conditions, which differs greatly from the conditions of this study. Every quantitative evaluation method can be eliminated as the expected results of this study can’t be clearly measured, which can be considered to be even a requirement for a successful evaluation. Benchmarking is eliminated as the included tools deal heavily with human-intensive tasks. Qualitative case study is eliminated also, because the evaluation won’t be done during a real software project, as this evaluation is done with academic purposes and not by an organization looking to improve their software development process. The methods left are qualitative screening, qualitative experiment, qualitative survey, and qualitative effects analysis, which all will be explored and analysed more thoroughly next.

4.2.5 Practical issues and the evaluation method selection

A golden rule for selecting the correct evaluation method is to choose the simplest method that satisfies the evaluation requirements, but there are also three practical aspects that can influence the choice. The first aspect is the time needed for conducting the evaluation, second is the required confidence towards the results of evaluation, and third is the cost of the evaluation. These aspects can be used to determine the right evaluation method choice if there are multiple options and the evaluation organizer is unable to select one method over another, or if the evaluation organizer wants to ensure the success of the evaluation. (Kitchenham, 1996c)

From the evaluation methods options left, Kitchenham (1996c) categorizes qualitative experiment and qualitative screening to take several weeks, while quantitative survey and -effects analysis would only take a few days, if the data for each method is already gathered. For this study the data hasn’t been gathered, so the data gathering process would probably take at least several weeks, as on effects analysis the expert to provide an opinion would need to be discovered, and on survey it would be the participants who would need to be found. This would put these methods to require more time than feature screening and experiment.

Comparison of the relative costs of the potential methods are similar to the time it would take to conduct them. Kitchenham (1996c) marks effects analysis evaluation as very low cost, and survey evaluation as low in relative cost. However, it is noted that this evaluation is done by assuming that the data is already gathered, and the gathering process would require significant investment. Both screening and survey are categorized to have a medium relative cost. On survey evaluation it is noted how creating the survey could possibly take some time, as well as the survey participants must be given some time to familiarize and answer the survey. On screening evaluation, it is noted how a single person can conduct it in a few weeks, but how it would take longer if a more detailed feature analysis were conducted. The comparison of relative cost of potential evaluation methods would push the selection towards selecting screening evaluation, because overall it appears as the most suitable option. Survey- and effects analysis evaluation would probably be too costly to conduct, due to the need of gathering the data, and survey evaluation remains quite neutral for the time being.

The last practical issue to consider is the relative risk of each evaluation method. Kitchenham (1996c) explains that you should choose an evaluation method, which gives you a good enough confidence to trust the results of the evaluation (e.g., if the tool under evaluation is going to be used in a critical process, which would cost a lot in case of a failure, you should choose an evaluation method, which produces highly reliable results).

From the evaluation methods left, the experiment evaluation has a low relative risk, survey evaluation has medium risk, and both screening- and effects analysis evaluations have very high risk. In this study the risk of failure is minimal, as it is conducted in academic purposes, and the achieved results are not going to be used directly in any commercial project. However, as this is a relatively new and certainly quickly advancing topic, the possibility must be recognized that there might be people, who could potentially come across this paper and use the achieved results on their projects. Still, the relative risk of the selected evaluation method must be recognized and mentioned as a threat to validity, so the reader is able to form their own opinion about the reliability of the results and decide whether they are going to use the information or not.

It appears that screening evaluation based on feature analysis is the most suitable evaluation method for this study. As the evaluation purpose of this study is to explore different tools and functionalities that Dapp development environments should possess, screening evaluation is the best choice as it is the only evaluation method that is suitable for evaluating a large number of tools or methods at the same time. It is also expected that documentations of different development environments and tools will be heavily relied on for the evaluation, which is the way that screening evaluation is done. Other aspects (i.e., relative risk, relative cost, and the length of evaluation) also indicates that the selection is correct, or at least they don't hint about it being a bad selection. The principles and the process of how the feature set is created are explained in the next chapter.

5. Feature analysis

Kitchenham and Jones (1997a) explain that the purpose of feature analysis evaluation is to try to put objective, rational reasoning behind a “gut feeling”, when it comes to selecting a tool or method to be used. This chapter explains the background of feature analysis more thoroughly, how feature set and evaluation criteria are created, showcases the produced feature set, and explains how the evaluation process is carried out.

A simple way of understanding a feature analysis is to think of it as a checklist of features a product should have. After an initial inspection you have limited your options into just a few, and you can conduct a more detailed evaluation on those options left. However, when it comes to software engineering, creating a simple checklist becomes a challenging task to do. Usually, the requirements towards different tools and methods are not so straightforward, that a simple yes or no answer would be sufficient enough to answer the question. For free to use tools and methods, the evaluation is often conducted to provide guidance for decisions to whether to use some specific tool or method on a specific task. In this scenario, the results must provide information about how well the tool or method suits the intended use case, are there any drawbacks that should be considered, and are there any other advantages that using the tool or method could bring. (Kitchenham and Jones, 1997a)

5.1 Screening mode

The reasoning and the process of selecting the screening mode to be the evaluation method is explained in the previous main chapter. Screening evaluation is conducted by a single person, and it is the best approach to evaluate a large set of tools or processes in a complex context. During the evaluation, the evaluator must identify some candidate tools or methods to be included on the evaluation, creating the evaluation criteria, gathering information and scoring each feature of each tool or method, and analyse and present the results. Evaluation is based only on documentation, and thus it produces not so reliable results. Due to being conducted by a single person only, the results can also be biased (Kitchenham and Jones, 1997c). However, the validity of the results can be improved by performing a more detailed evaluation (Kitchenham and Jones, 1997a).

In order to increase the validity of the results, this study uses some elements of the case study evaluation method. Even though the case study method wasn’t a good fit to be selected as a main evaluation method, it possesses some aspects that could benefit the screening mode evaluation, and thus the evaluation would produce more reliable results. How the evaluation is conducted is explained later in the chapter.

Once the evaluation method is selected, Kitchenham and Jones (1997a) explain that the next steps are:

1. Create a scoring/ranking system, which will be used to evaluate all the features.
2. Choose how the responsibilities of the actual feature evaluation will be shared.
3. Execute the evaluation.
4. Analyse the results.
5. Present the results.

5.2 How feature set is created

The aim of the feature analysis is to create an evaluation framework for smart contract development environments. The evaluation is concerned about identifying and ranking different features that developers would like to have within their development environment to aid the development process. Each identified feature is listed and the purpose of it explained, as well as is the reasoning for the evaluation criteria.

It is important to understand that a development environment/framework is not required at all in software development. The purpose of it is to ease the developer's task of developing a software by automating some of the tasks as well as providing different tools within the environment/framework to be used by the developer. Thus, none of the features is evaluated as a mandatory feature to have, as Kitchenham and Jones (1997c) put it as failing to meet a single mandatory feature would mean that the tool or process under evaluation would be unusable. However, as the development environment/framework is highly personal choice, which is influenced by the needs of the developer, the evaluation can't claim that some environment/framework is unusable if it doesn't possess a certain feature, as the developer might not care at all about that specific feature. This changes however, if the development environment has a certain feature that doesn't work as intended, as this would mean that the environment doesn't provide what it promises to deliver, and in this scenario the development environment can be considered to be unusable. However, in this study failing to execute some of the functionalities will be noticed in the evaluation as a reduction of score, but no development environment will be declared as unusable. In short, missing a feature doesn't make the development environment unusable, but failing to execute a promised feature reduces points.

The evaluation criteria features are created by merging the requirements identified in the literature review chapters with the high-quality product characteristics, which are described next.

5.3 Characteristics of a high-quality product

In order to create objective evaluation criteria for different tools, the characteristics of an ideal tool must be first identified. ISO (International Organization for Standardization) provides standards, which are developed from the expertise of the field's experts, and they describe optimal ways to do different tasks (e.g., how to manage people, how to reduce accidents in the workplace, and how software engineers could keep their users sensitive information secure) (ISO, 2022b). ISO standards have been widely accepted and used throughout different industries and in academics it has been considered to be a norm to use their standards for different purposes. ISO/IEC standard 25010:2011 provides among other things a product quality model, which contains a total of eight different characteristics along with their sub characteristics, that makes a high-quality product (ISO, 2022a). The most suitable characteristics for the evaluation context that are taken in consideration in this study are displayed in Figure 4.

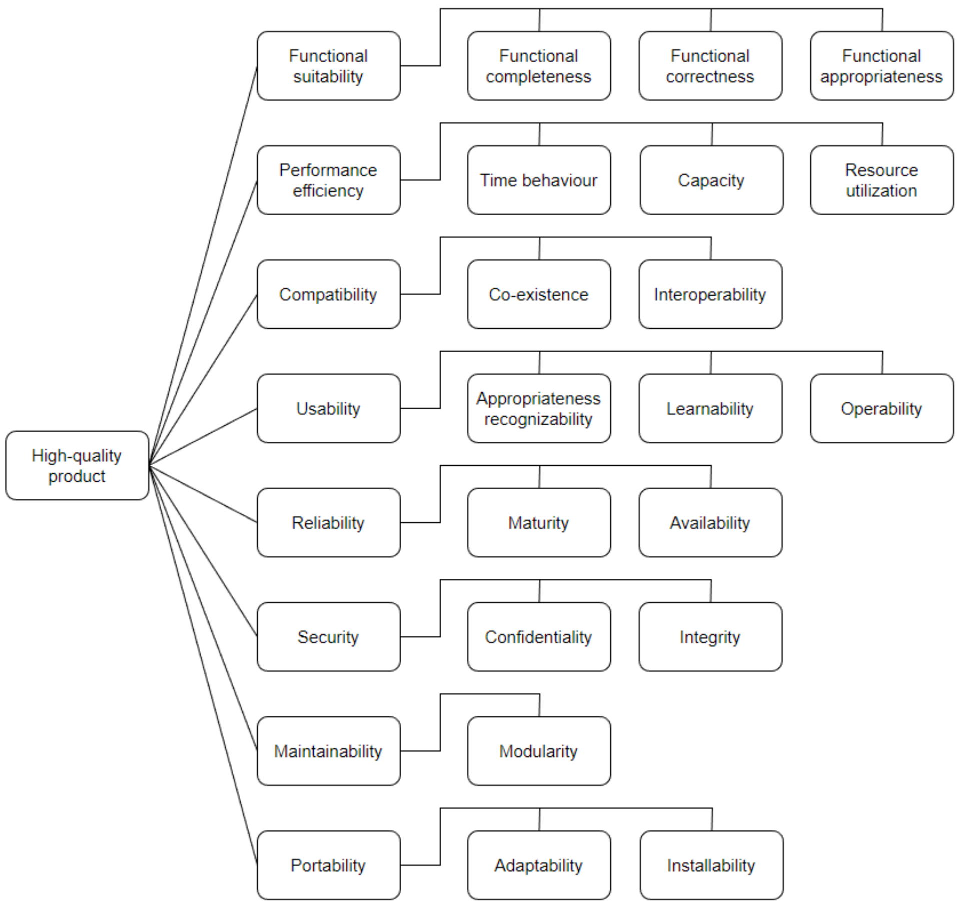

Figure 4. Relevant characteristics of a high-quality software product.

Each of the top-level high-quality characteristics have multiple second level attributes that essentially form the top-level characteristic. Functional suitability describes how well the functions provided by the product satisfy the needs of the users on specific tasks, and this study examines that by evaluating what functionalities the development environments possess out-of-the-box. Performance efficiency is about how efficiently the software product utilizes the resources it has, and it is evaluated by examining how much resources the development environment requires and does using it affect the performance of other programs. Compatibility is about how well the development environment works with other programs, and it is evaluated by whether the environment restricts using other tools or programs in any way. Usability indicates how well and efficiently the users are able to use the product, and it is evaluated by how easy and efficient it is to control the environment, and how excessive and good the documentation is. Reliability is about how well the program works under normal conditions, and it is examined by its availability and does it operate as expected. Security character category is concerned about the security aspects of the program, and it is evaluated by the reputation of the development environment and does any security related concerns arise during the evaluation. Maintainability is about updating and keeping the software operating, and it is evaluated by how easy it is to update the environment and how much effort it takes. The last characteristic is portability, which concerns how well the software can be used and transferred to other devices and environments, and in this study, it is evaluated by the difficulty of the installation process of the development environment, and the plugins as well as the external tools. (ISO, 2022a)

5.4 Feature set

As previously explained, not all the development environments are created to provide the same functionalities, but this study is concerned with comprehensive frameworks, so it is expected for the framework to have the most commonly requested functionalities (i.e., testing framework, debugger, compiler, deploying, ability to import external plugins/tools) that are identified in chapters two and three. This, however, becomes difficult to evaluate as it is common for the development environment to have an ability of importing an external plugin or a tool within the environment. For example, if testing functionality would be a mandatory feature of the development framework and one of the development frameworks would come without one, with the purpose of allowing users to import their favorite testing framework and good instructions of how to do it, that would mean that this development environment would be deemed as unusable, which in my opinion isn't the case. The same problem arises with determining the importance of the feature. Continuing with the testing example, in general having the functionality to test the smart contracts is clearly a mandatory feature, but once again the problem is with the evaluation scale, as in theory, it can be even a deliberate choice to ship the environment without such functionality.

These are the reasons why the evaluation criteria don't declare the importance of each feature like Kitchenham and Jones (1997b) suggests, but rather emphasizes the importance of each feature within the evaluation criteria by altering the rewarded points based on the importance. Now, for example, if the development environment comes without the testing functionality and it doesn't allow importing any testing framework, it will receive zero points. However, if the user is able to import a testing framework, it will receive 2 points, and if it comes out-of-the-box with the testing functionality, it will receive 5 points. This adds a bit of flexibility in the evaluation, and in general should produce more accurate results.

Conducting the evaluation this way results in leaving gaps between the grades, but there are reasons why it is the most suitable option in this case. As Kitchenham and Jones (1997d) explain, the aim of the feature analysis is to identify the tool or method that best meets the user requirements. With this approach, the achieved results from the evaluation can be compared by their absolute scores, as the evaluation criteria grading already contains the level of importance.

Tables 4 - 11 display the feature set by the characteristic categories along with the subcategory and the possible scores from the feature. The full feature set, which includes the evaluation criteria, can be viewed as an appendix A.

Table 4. Features related to functional suitability.

Id	**Description**	**Subcategory**	**Score**
1	How environment supports testing?	Functional completeness	0, 1, 2, 5
2	Environment supports mainnet forking for testing?	Functional completeness	0, 5
3	Environment automatically compiles the code when running tests (if there are any changes)?	Functional completeness, Usability	0, 1
4	How environment supports debugging?	Functional completeness	0, 1, 4
5	How environment supports compiling?	Functional completeness	0, 1, 2, 5
6	How environment supports deploying?	Functional completeness	0, 1, 5
7	How environment supports gas optimization?	Functional completeness	0, 2, 3, 5
8	How environment supports ensuring the security of the smart contracts?	Functional completeness	0, 3, 5
9	Does the development environment enable parallel frontend development (e.g., exposes interface for the frontend to connect)?	Functional completeness	0, 5
10	Development environment worked as intended during the evaluation?	Functional correctness	0, 1, 3
11	How much the environment supports and overall eases the development process?	Functional appropriateness	0, 1, 3, 5
12	Development environment supports Solidity and Vyper?	Functional completeness	0, 3, 5
13	Development environment provides an extension for VS Code?	Functional completeness	0, 3

Table 4 shows the features related to functional suitability characteristic category, which is concerned with how well the product satisfies the users requirements on a specific task, and it is evaluated with 13 features, which fall into three subcategories. Table 5 displays the features related to performance efficiency.

Table 5. Features related to performance efficiency.

Id	**Description**	**Subcategory**	**Score**
14	Does the development environment operate as fast as expected?	Time behaviour	0, 1, 2, 3, 4
15	Does the development environment require so much resources that it negatively affects using the PC?	Resource utilization	0, 1, 3
16	Development environment has restrictions regarding the size of the project?	Capacity	0, 5

As seen in Table 5, the performance efficiency, which is concerned with how efficiently the software operates, is evaluated with three features, which fall into three subcategories. Table 6 displays the features related to compatibility.

Table 6. Features related to compatibility.

Id	Description	Subcategory	Score
17	Can Open Zeppelin be imported (or is it by default) within the development environment?	Co-existence, Interoperability	0, 3
18	Can Ganache be imported (or is it by default) within the development environment?	Co-existence, Interoperability	0, 3
19	Can Waffle be imported (or is it by default) within the development environment?	Co-existence, Interoperability	0, 3
20	Environment allows importing plugins/ external tools within the environment?	Co-existence, Interoperability	0, 5

As shown in Table 6, compatibility, which is about how well the product operates with other programs, is evaluated through four features, which all contain two subcategories. Table 7 displays the usability related features.

Table 7. Features related to usability.

Id	Description	Subcategory	Score
21	Where the suitability and the main features of the environment can be determined?	Appropriateness recognizability	0, 1, 2, 3, 4
22	Development environment documentation have a search function?	Learnability	0, 1
23	Environment documentation provides up-to-date information/instructions for installing the dependencies?	Learnability	0, 4
24	Development environment has smart contract and Dapp project tutorials/examples?	Learnability	0, 1, 2, 4, 5
25	Generally (as there is always some outdated information), does the documentation appears to be up to date?	Learnability	0, 2, 5
26	Documentation explains how plugins, external tools, or alternative functionalities can be imported to the environment?	Learnability	0, 1, 2, 3
27	How much the documentation could be improved?	Learnability	0 - 5
28	Development environment has its own forum, in which developers can ask help from other developers?	Learnability	0, 1, 2
29	Are the default commands to execute functionalities/ operate environment long and difficult? (e.g., majority of them contain more than 20 characters, at first it is beneficial to save them in a text file?)	Operability	0, 1, 2
30	Are you able to start a new project under 10 minutes with the environment (including the installation and possible configuration)?	Operability	0, 2

As seen in Table 7, usability, which is about how well and efficiently the users can use the product, is evaluated through 10 features, which fall into three categories. Table 8 displays the features which are related to reliability.

Table 8. Features related to reliability.

Id	Description	Subcategory	Score
31	How many open issues the GitHub repository has from the past 30 days?	Maturity	0 – 5
32	Under normal circumstances, development environment can be used any given time (e.g., not down every night at the same time)?	Availability	0, 5

As shown in Table 8, reliability, which is concerned with how well the program works under normal circumstances, is evaluated by two features, which fall into two subcategories. Table 9 displays the security related features.

Table 9. Features related to security.

Id	Description	Subcategory	Score
33	Can security flaws be detected, or is there any reason to questionable the security (including template projects)?	Confidentiality, Integrity	0, 1, 2, 5
34	What is the package's health score on Snyk (https://snyk.io/advisor/)?	Confidentiality	0 - 5

As seen in Table 9, there are two security related features, which contain two subcategory characteristics. Table 10 displays features related to maintainability.

Table 10. Features related to maintainability.

Id	Description	Subcategory	Score
35	User can change some of the functionalities to alternative solutions (e.g., testing framework, development node)?	Modularity	0, 5
36	How much effort it takes to update the development environment?	(No subcategory)	0, 1, 3
37	What is the philosophy of the development environment about balancing new technologies/functionalities and stability?	(No subcategory)	0, 2, 5

As shown in Table 10, maintainability, which is concerned about updating and keeping the software functioning, is evaluated with three features, out of which only one has a subcategory. Table 11 displays portability related features.

Table 11. Features related to portability.

Id	Description	Subcategory	Score
38	Development environment can be used on Linux, macOS, and on Windows 10?	Adaptability	0, 5
39	Does the development environment have pre-requirements for installation, and how much effort installing those requires?	Installability	0, 2, 5
40	How much effort does the development environment installing process require?	Installability	0, 2, 5

As seen in Table 11, portability, which is about how the software can be used and transferred to new systems and platforms, is evaluated with three features, which fall into two subcategories.

As it can be seen on Tables 4 - 11, not every feature will reward a maximum of five points unlike Kitchenham and Jones (1997d) suggests, and this is due to the decision of including the level of importance within the evaluation criteria. It also requires a little, but important addition for presenting the results, as the reader is likely to assume that a development environment doesn't support a certain feature well if it receives only one point, despite it being the highest available grade. This is why the maximum available score is reported along with the results, as it improves the readability of the results and also presents the results in a much truthful manner.

5.5 How the evaluation is done

Evaluations are conducted in May 2022 using Arch Linux (a rolling release Linux distribution, version Linux x64 5.17.19-arch1-1) as an operating system. VSCodium (an open-source binary distribution of VS Code, version 1.67.2) is used as a text editor and version 16.15.0 of Node (newest long term supported version).

The same evaluation process is done for every development environment selected in the study. The process is started by identifying if there are any tutorial projects that are aimed for newcomers to try the development environment out. Having these kinds of tutorials is expected to be found from every development environment, as practically every popular web framework provides those for obvious reasons. The possible provided tutorial is followed to build an overall understanding of the characteristics and the behaviour of the environment. If the development environment doesn't have tutorials, the evaluator is to try the environment out by trying to build a simple project possibly by using the code examples in the development environment documentation, or at the last resort building something by themselves.

Once the evaluator has gained some familiarity with the development environment, it is time to go through the evaluation form feature by feature and give each of them a score. During the scoring it is necessary for the evaluator to read the documentation of the development environment and to continue using the environment to be able to score each feature. Once every feature on the evaluation form is scored, the scores from the form can be extracted and the graphs for presenting the results can be made.

6. Results

In this chapter, the achieved results from the evaluations are presented for each development environment separately by first introducing the development environment under evaluation, followed by displaying the results of the evaluation.

Three development environments are evaluated, which are Hardhat, Truffle, and Brownie. The reasoning behind selecting these development environments over others is that these specific environments appear to be the most comprehensive and well-liked environments among the Ethereum smart contract developers. Probably due to the immaturity of the technology, the term “development environment/framework” is broad and contains products that have different kinds of functionalities and tools. Another point worth noting is that it is quite common that you can import at least some part of a development environment to another one (e.g., you can use Truffle for testing inside Hardhat). Scaffold-eth, which is a popular development environment (i.e., is active and has 7,072 GitHub stars), is left out from the evaluation for this reason, as it heavily depends on Hardhat (i.e., for running local networks, deploying, and testing). In the chapter three mentioned Embark, which has the third most stars on GitHub repository, is replaced by a Python based development environment called Brownie, as Embark hasn’t received an update since 2020. More information about each evaluated environment is provided before presenting the results they received.

The results are presented through different kinds of graphs to enable readers to be able to quickly form an overview of the results. As the evaluation criteria has gaps between the possible scores, displaying results only by the percentage of the maximum available score doesn’t necessarily tell the truth as that feature might not be that important in a bigger picture (i.e., feature can either receive a score of zero or one). In this case, the graph will display that the feature received zero percent of the available score, but it fails to tell the importance of that feature. To resolve this, a graph, which displays the received score against the maximum available score, is also used.

The most interesting points (e.g., the results of the most important features) of each graph are analysed within this chapter but comparing and analysing the differences between the results of each development environment is done in the next chapter. The raw results are attached as an appendix B.

6.1 Hardhat

Hardhat is an open-sourced development environment created by Nomic Foundation. It self-describes to be an “Ethereum development environment for professionals”, and highlights running local blockchain for development and testing purposes, being flexible and extensible, and having their own plugin ecosystem (Hardhat, 2022). It has received 3,332 stars on GitHub.

Hardhat received a total of 136 points out of a maximum of 164, which translates to around 83 percent. Figure 5 displays the evaluation scores as a percentage by the characteristic categories of a high-quality software product.

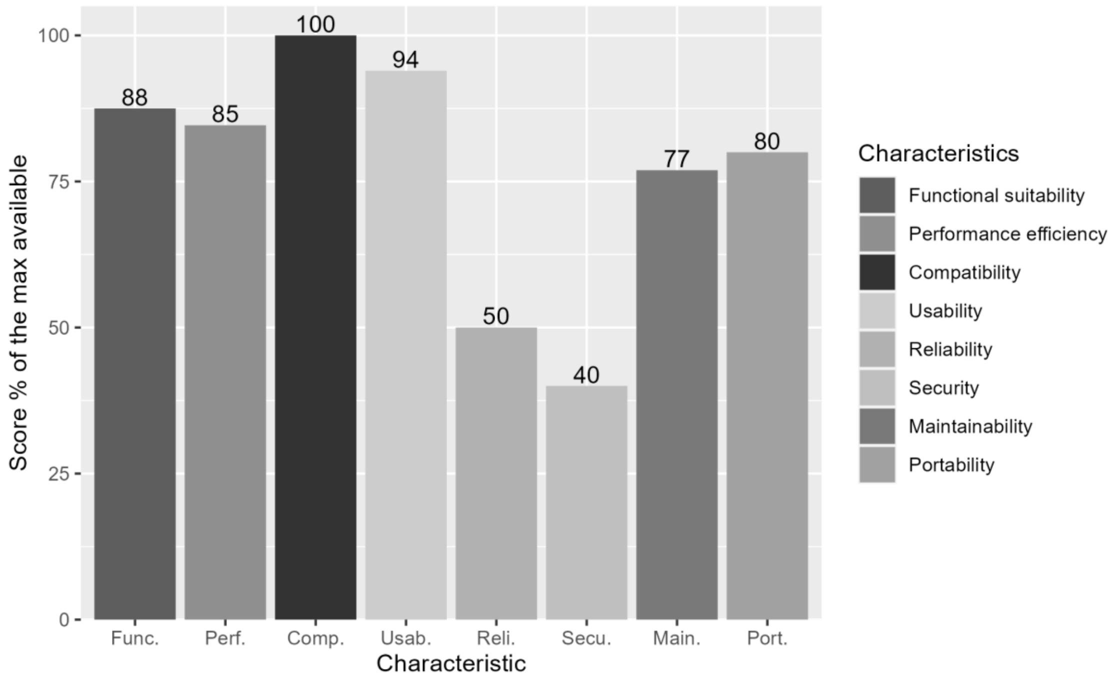

Figure 5. Hardhat evaluation score as a percentage by the characteristic category.

As an overview seen in Figure 5, Hardhat performed relatively well in six characteristic categories (i.e., having at least 77 percent of the maximum available score), and relatively bad in two (i.e., 50 percent or less of the maximum available). The strongest characteristic for it was compatibility, in which it received the highest score in every feature. Usability was the second strongest category, being six percent behind the compatibility. Security and reliability were the weakest characteristic categories for Hardhat.

Figure 6 displays the evaluation scores as a percentage by feature id.

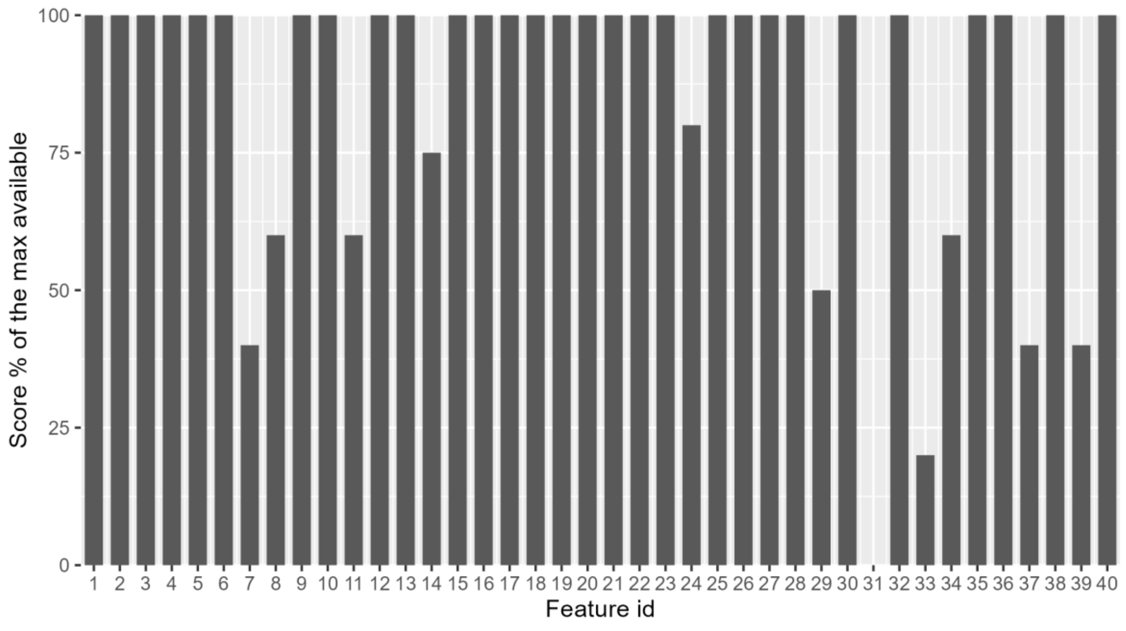

Figure 6. Hardhat evaluation score as a percentage by feature id.

By examining the evaluation score percentages by feature ids shown in Figure 6, it can be seen that only one feature out of 40 received a score of zero, while 29 features received the maximum scores. It is also worth noting that only five features received less than 50 percent of the maximum available score. Two features received less than 25 percent, three features between 25 percent and less than 50, four features between 50 percent and less than 75, and 30 features received 75 percent or more. These points indicate that Hardhat provides a comprehensive support for Ethereum smart contract developers.

Figure 7 shows the received scores against the maximum available scores between the features 1 and 13.

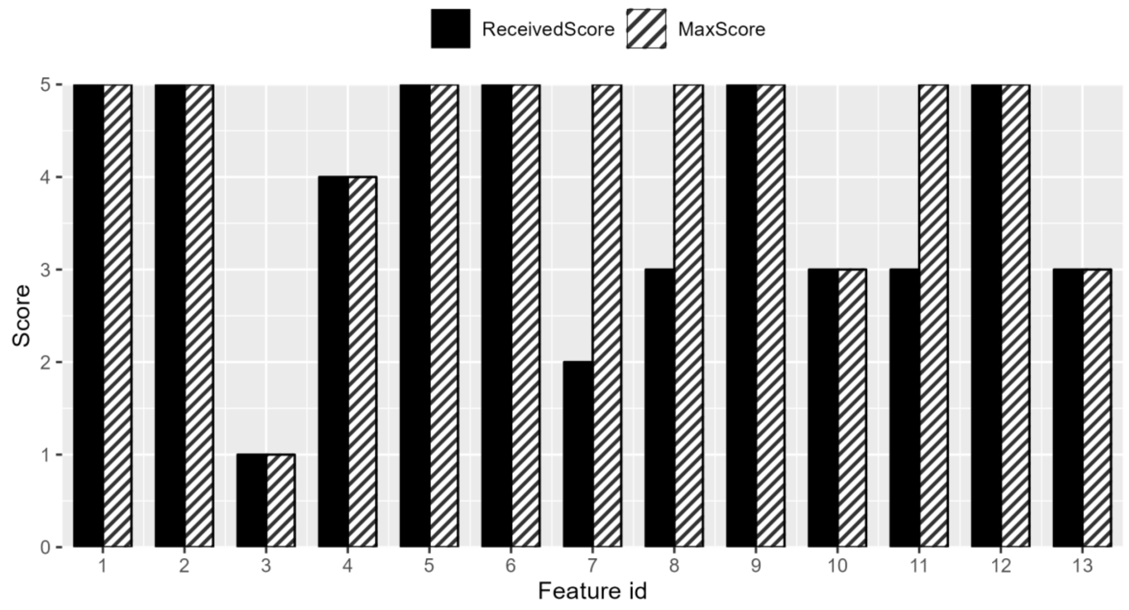

Figure 7. Hardhat scores against maximum scores on features 1 – 13.

As shown in Figure 7, on the most important features (i.e., those with a maximum score of five) Hardhat received full points on six out of nine. Those six features were related to testing, mainnet forking, compiling, deploying, parallel frontend development, and language support. The biggest lack of support is related to gas optimization, as Hardhat provides only instructions on how to import external tools into the development environment. The remaining two features which didn’t receive the maximum scoring were related to ensuring the security of the smart contracts and how much does the environment overall ease the development process. Hardhat provides documentation also on how to import external smart contract security tools, but it comes out-of-the-box without one. The lack of support in these areas is the reason why Hardhat doesn’t receive the full score on how much it supports during the development process. All the remaining four features received the maximum possible grading.

Figure 8 shows the received scores against the maximum available scores between the features 14 and 26.

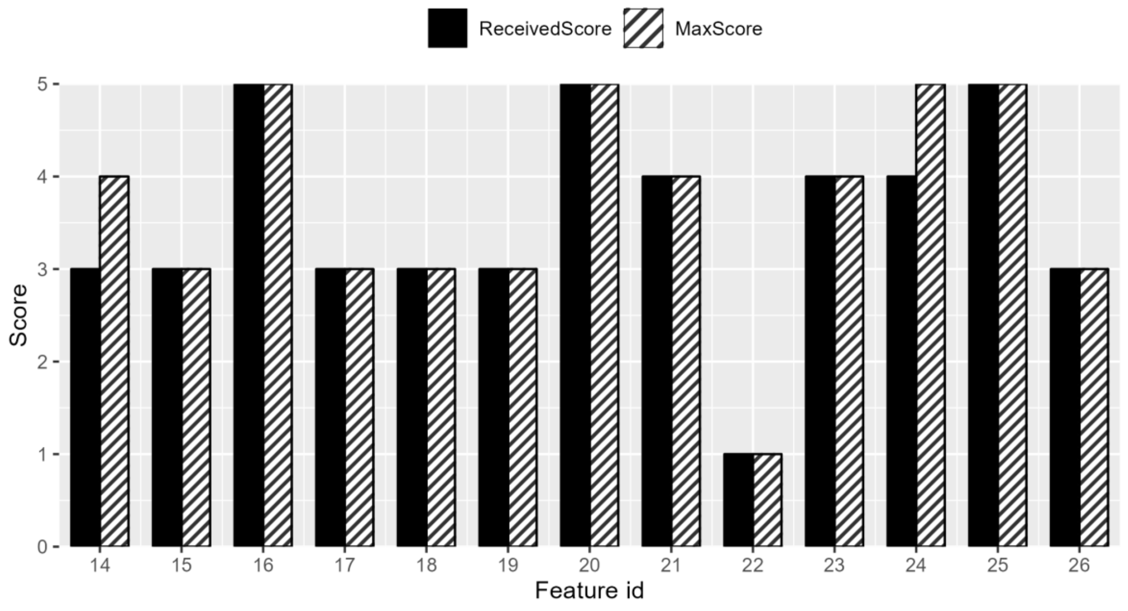

Figure 8. Hardhat scores against maximum scores on features 14 – 26.

On the features between 14 and 26 as shown in Figure 8, Hardhat received the full score on 11 features. The only five-point feature that didn't receive the full was related to tutorial projects, and the reason for it was that the frontend of the "hackathon boilerplate" tutorial project wasn't implemented using the current standard practices.

Figure 9 shows the received scores against the maximum available scores between the features 27 and 40.

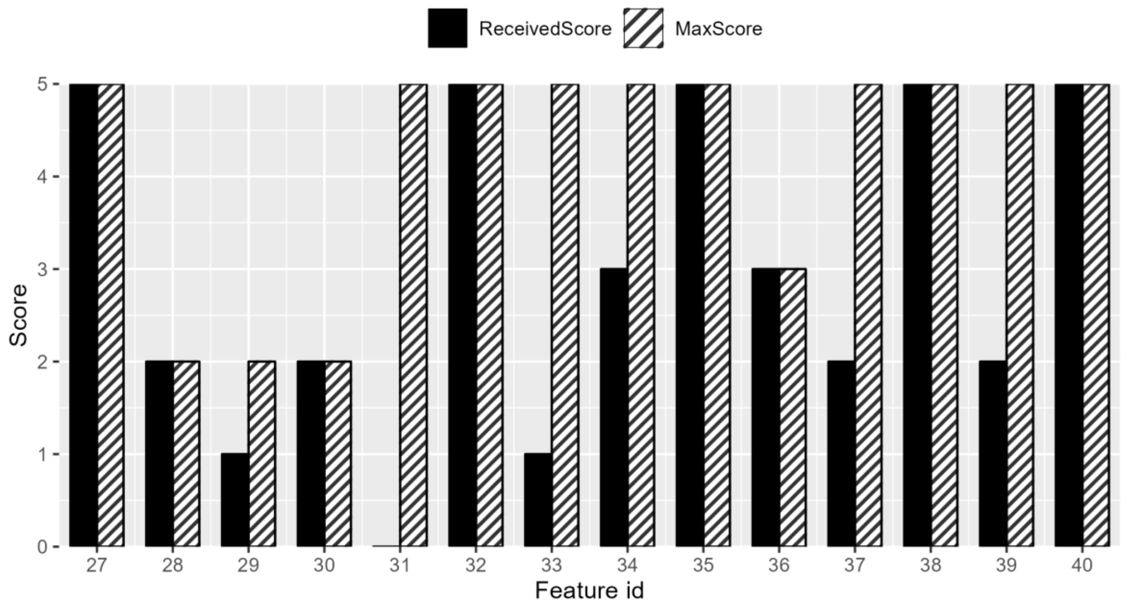

Figure 9. Hardhat scores against maximum scores on features 27 – 40.

As displayed in Figure 9, between the features 27 and 40, there are 10 five-point features and Hardhat received the full score only on five of them. The features that received the full score of five were related to quality of the documentation, availability of the development environment, ability to use alternative solutions for the functionalities (e.g., testing), availability on the three main operating systems (i.e., Windows, Linux, macOS), and installing process. Feature 31 was the only one that received zero points, and it was related to the number of open issues on the GitHub repository. Hardhat had 42 open issues and to receive a score of one, the repository should have 40 or less open issues. However, having a high number of open issues can be seen as a positive thing on the contrary of the product being faulty, as it can also mean that the product is popular, and the community are passionately trying to improve the quality of the product. This might be the case with Hardhat, as it has performed relatively well on the evaluation, and no major bugs were identified while conducting the evaluation. Feature 33 is the next worst graded five-point feature on the set, and it received the score because the dependencies of a template project were not up to date as it included one critical vulnerability. The last three five-point features that didn't receive the full scores were related to balancing between stability and upgrades, installation pre-requirement process, and the score that an external service called Snyk provides to the development environment.

6.2 Truffle

Truffle is an open-sourced development environment created by ConsenSys, and it self-describes to be "a world class development environment, testing framework and asset pipeline for blockchains using the Ethereum Virtual Machine (EVM), aiming to make life as a developer easier" (Truffle Suite, 2022). Its GitHub repository has 13,213 stars.

Truffle received a total of 127 points out of a maximum of 164, which translates to around 77 percent. Figure 10 displays the evaluation scores as a percentage by the characteristic categories of a high-quality software product.

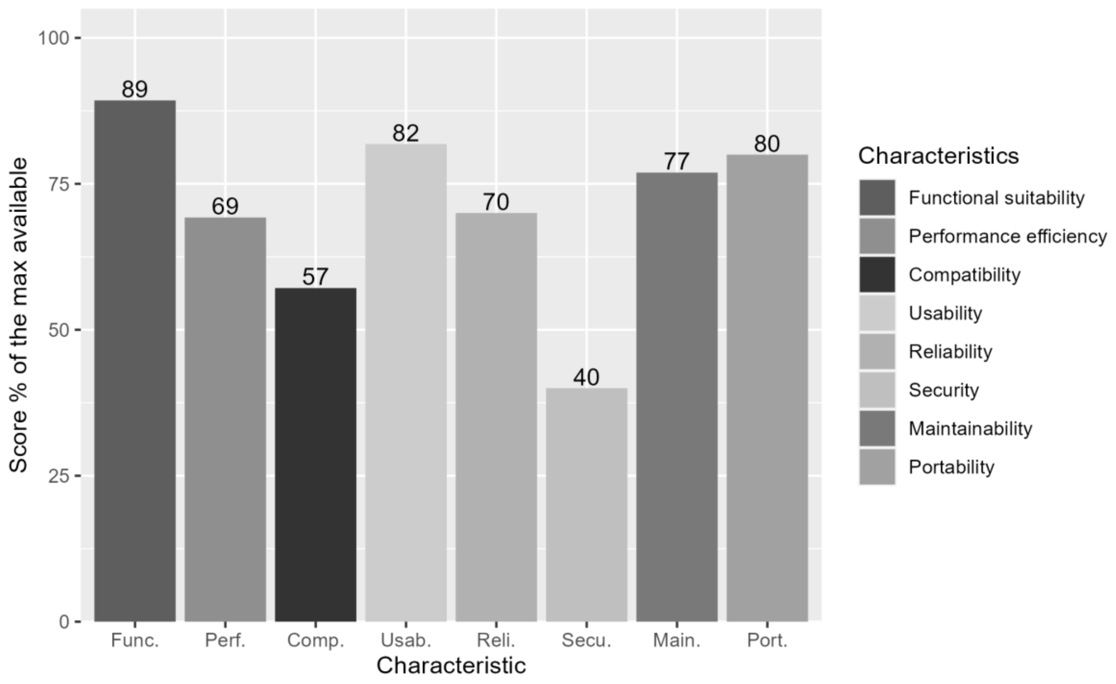

Figure 10. Truffle evaluation score percentage by the characteristic category.

As seen in Figure 10, Truffle performed decently well on four characteristic categories (i.e., functional suitability, usability, portability, and maintainability), decently on three (i.e., reliability, performance efficiency, and compatibility), and weakly on one (i.e., security). Functional suitability is the strongest characteristic of Truffle, as it scored 89 percent of the maximum available score. Truffle also scored noticeably consistently, as five characteristic categories out of the eight are within 13 percentage points, and six within 20.

Figure 11 shows the evaluation scores as a percentage by feature id.

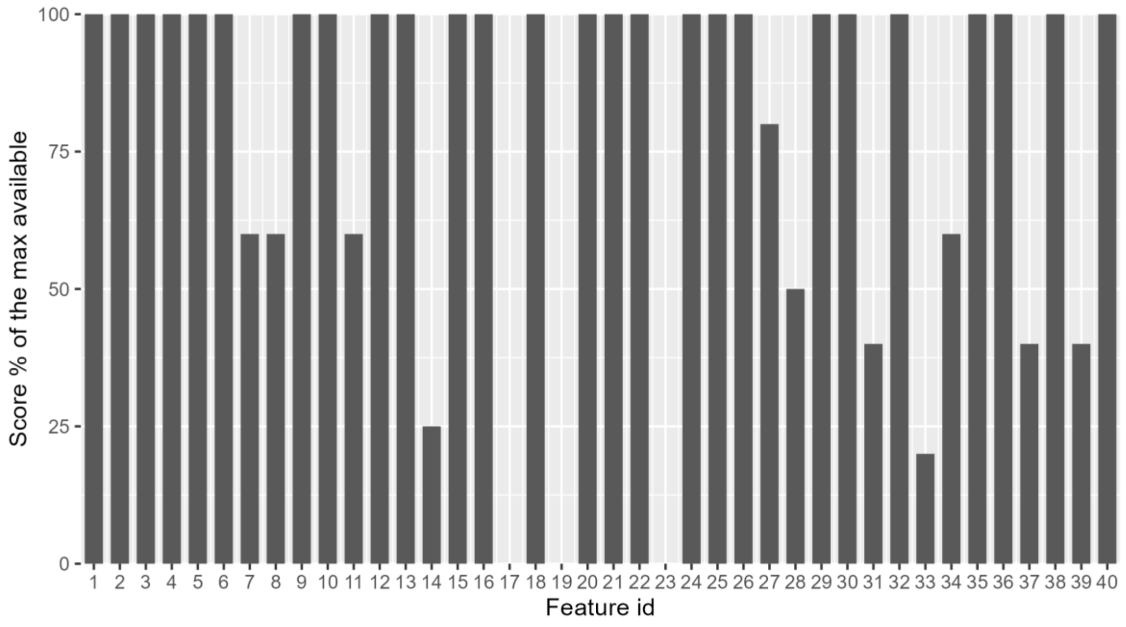

Figure 11. Truffle evaluation score as a percentage by feature id.

As shown in Figure 11, Truffle scored the maximum available score in 27 features out of 40, which indicates that Truffle supports developers very well in a lot of aspects. However, it totally fails on three occasions (i.e., doesn't score even a single point), and on top of those it provides weak support in six features (i.e., percentage 50 or less). Four features received less than 25 percent as well as between 25 percent and less than 50, five features between 50 percent and less than 75, and 27 features received 75 percent or more.

Figure 12 shows the received scores against the maximum available scores between the features 1 and 13.

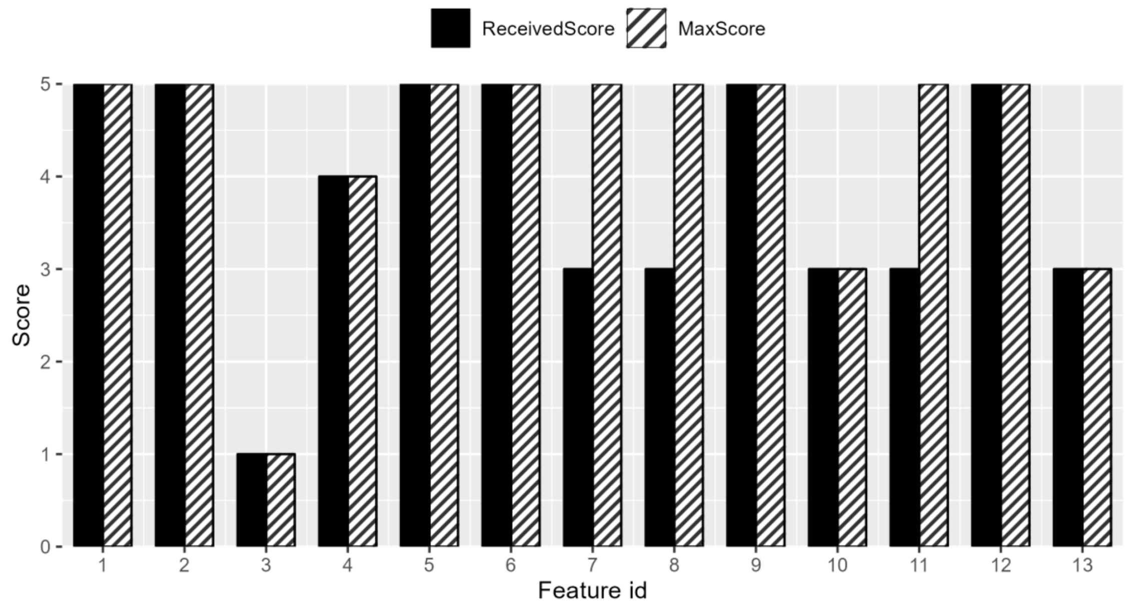

Figure 12. Truffle scores against maximum scores on features 1 - 13

As shown in Figure 12, on the features between 1 and 13, Truffle received the maximum amount of score on 10 features. The five-point features that provided full support were related to testing, mainnet forking, compiling, deploying, parallel frontend development, and language support. The two five-point features that failed to provide full support are related to gas optimization and ensuring the security of the smart contracts. All the rest features received the maximum amount of score available.

Figure 13 shows the received scores against the maximum available scores between the features 14 and 26.

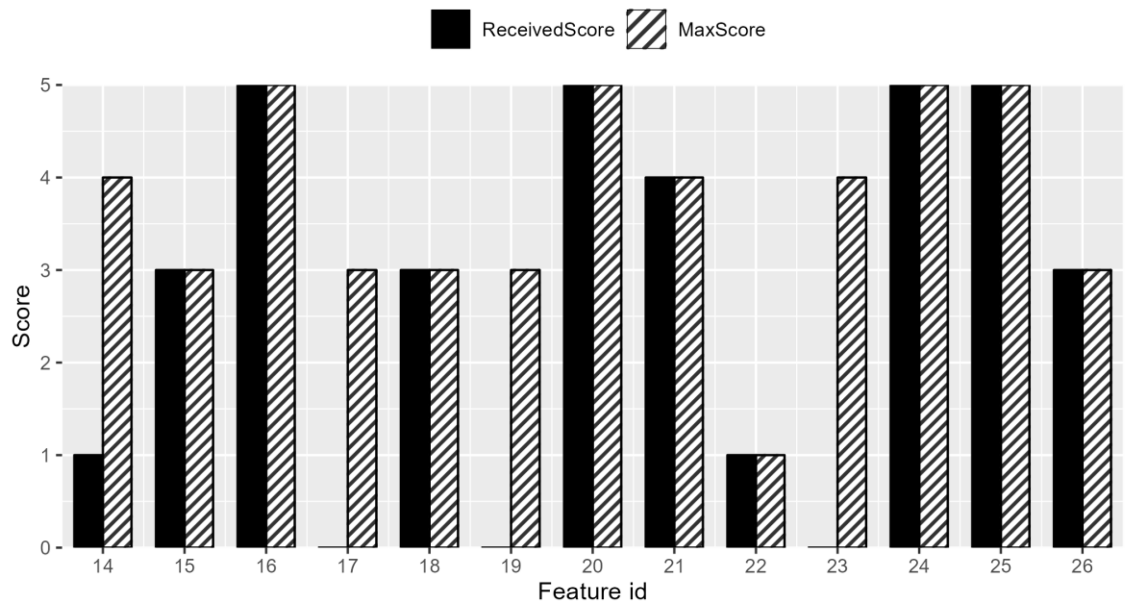

Figure 13. Truffle scores against maximum scores on features 14 – 26.

As shown in Figure 13, between the features 14 and 26, Truffle scored the maximum available score on every five-point feature, and they were related to restrictions to project size, importing external tools and plugins, tutorial/example projects, and documentation being up to date. What's noticeable is that Truffle doesn't provide any support on three features. These features are related to the ability to import Open Zeppelin or Waffle into the environment, and documentation failing to provide instructions on how to install the dependencies of the development environment. These instructions are provided only on the tutorial project, which is separated from the documentation, which contains a quick start section, which should have this information, or at least provide a link to where this information can be found.

Figure 14 shows the received scores against the maximum available scores between the features 27 and 40.

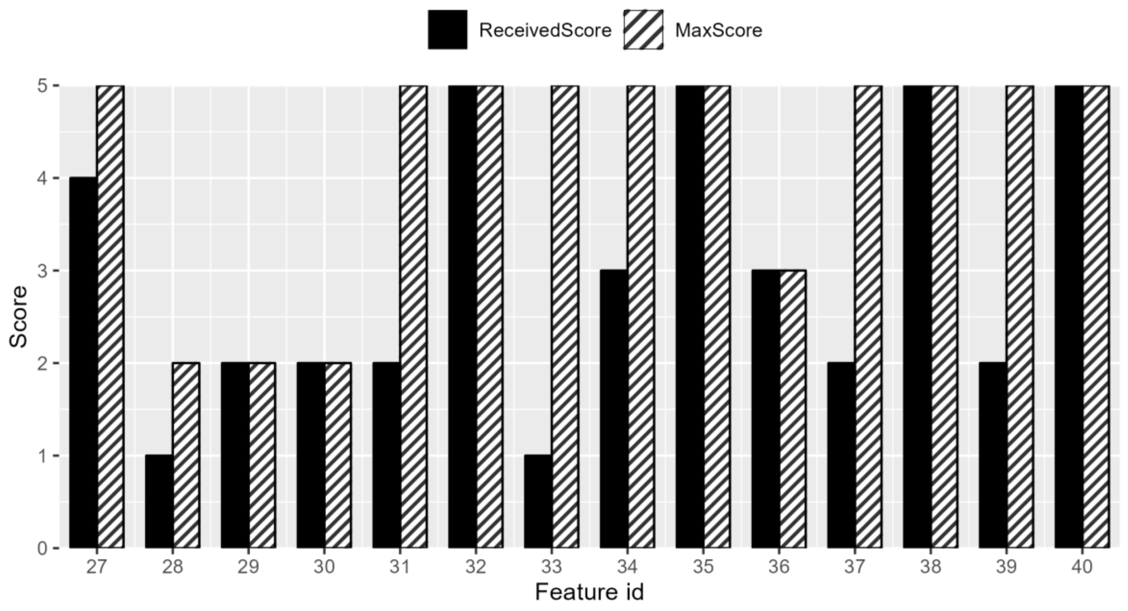

Figure 14. Truffle scores against maximum scores on features 27 – 40.

On features 27 to 40, displayed in Figure 14, Truffle received a maximum score on seven features. On five-point features, Truffle was scored to provide full support on four occasions out of 10, and these features were related to availability, using alternative solutions for functionalities, being supported on every main operating system, and installing process. The five-point features that could provide more support are related to documentation, GitHub issues, security, Snyk score, balancing between updates and stability, and pre-requirement installation process.

6.3 Brownie

Brownie is an open-source development environment, and it self-describes to be "a Python-based development and testing framework for smart contracts targeting the Ethereum Virtual Machine" (Brownie, 2022). It has 2,017 stars on the GitHub repository.

Brownie received a score of 114 out of 164, which translates to around 70 percent. Figure 15 displays the evaluation scores as a percentage by the characteristic categories of a high-quality software product.

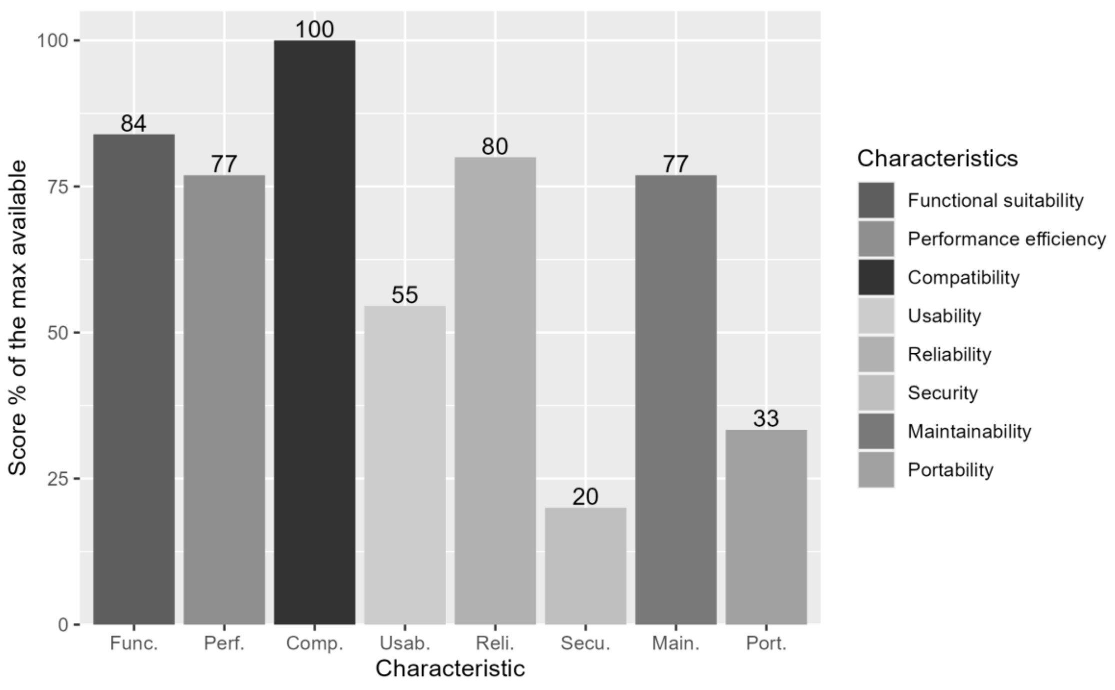

Figure 15. Brownie evaluation score percentage by the characteristic category.

As displayed in Figure 15, compatibility is the strongest characteristic of Brownie, receiving the maximum available score, and security the weakest, receiving only 20 percent. Four characteristics were within seven percentage points, and five characteristics received at least 77 percent of the maximum score. Only two characteristics received less than 50 percent of the available score, but the actual scores were much lower than that (i.e., 33 percent for portability and 20 for security). Brownie is lacking greatly in support on these characteristics, while providing good support on five characteristics.

Figure 16 displays the evaluation scores as a percentage by feature id.

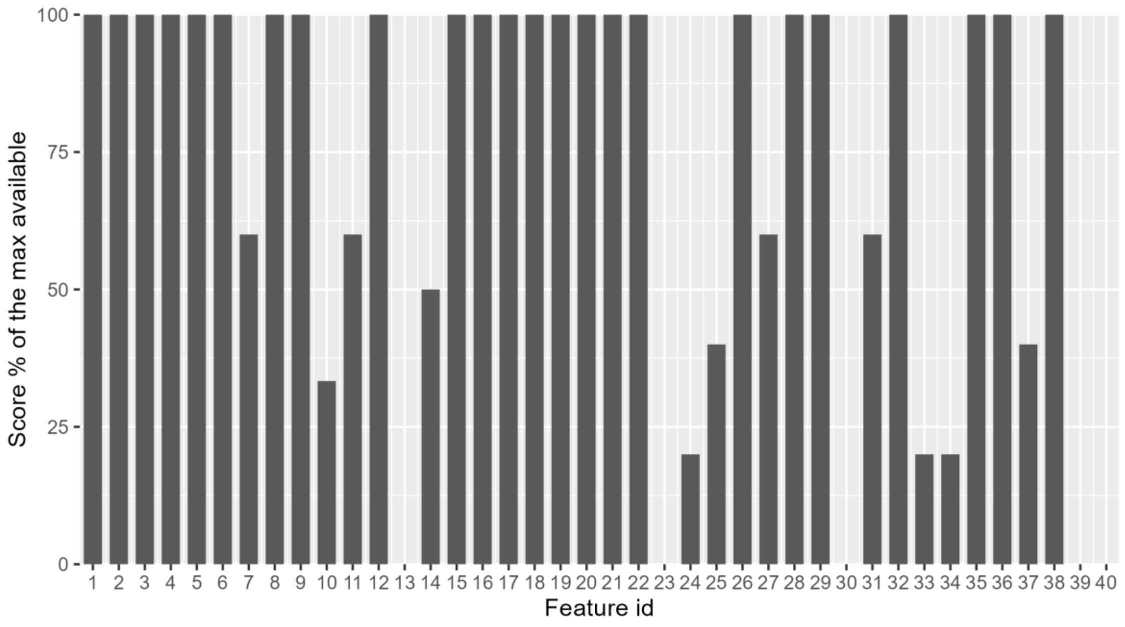

Figure 16. Brownie evaluation score as a percentage by feature id.

Shown in Figure 16, Brownie received a full score on 24 features out of 40, while failing to receive any points on five features. Eight features received less than 25 percent, three features between 25 percent and less than 50, five features between 50 percent and less than 75, and 24 features received 75 percent or more.

Figure 17 shows the received scores against the maximum available scores between the features 1 and 13.

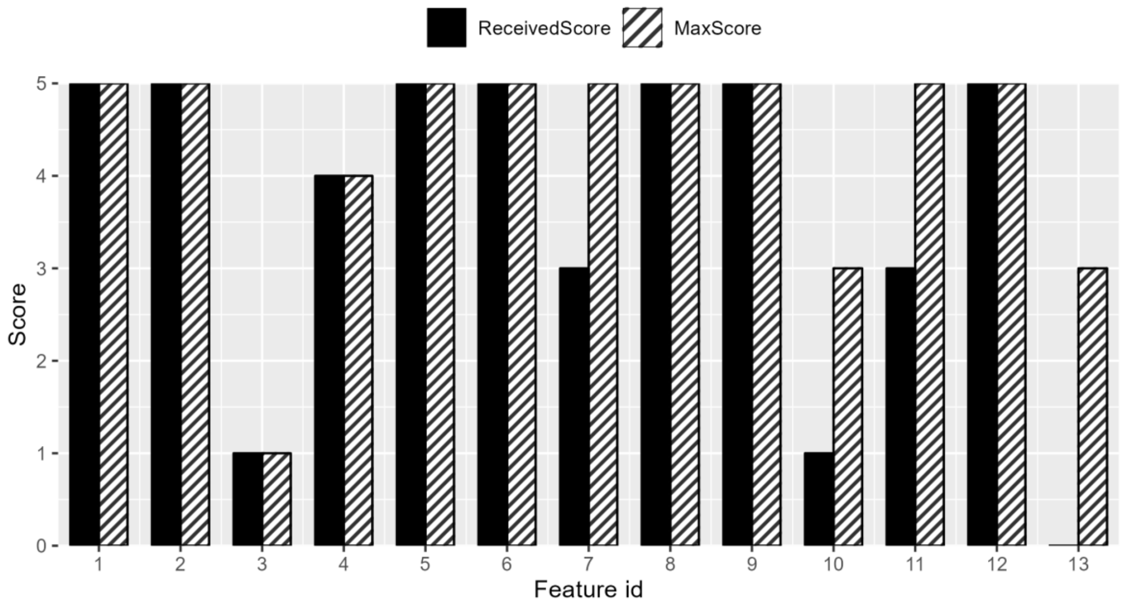

Figure 17. Brownie scores against maximum scores on features 1 – 13.

Between the features 1 and 13 shown in Figure 17, Brownie received a maximum grade on nine features. There are nine five-point features, and Brownie provides full support for seven of them. These fully supported features are related to testing, mainnet forking, compiling, deploying, ensuring smart contract security, parallel frontend development, and language support. The two five-point features that Brownie fails to provide full support are related to gas optimalization and how much does the environment ease the development process. There are four features that can receive a score less than five, and Brownie provides full support on two of them, while failing to provide any support on one of them.

Figure 18 shows the received scores against the maximum available scores between the features 14 and 26.

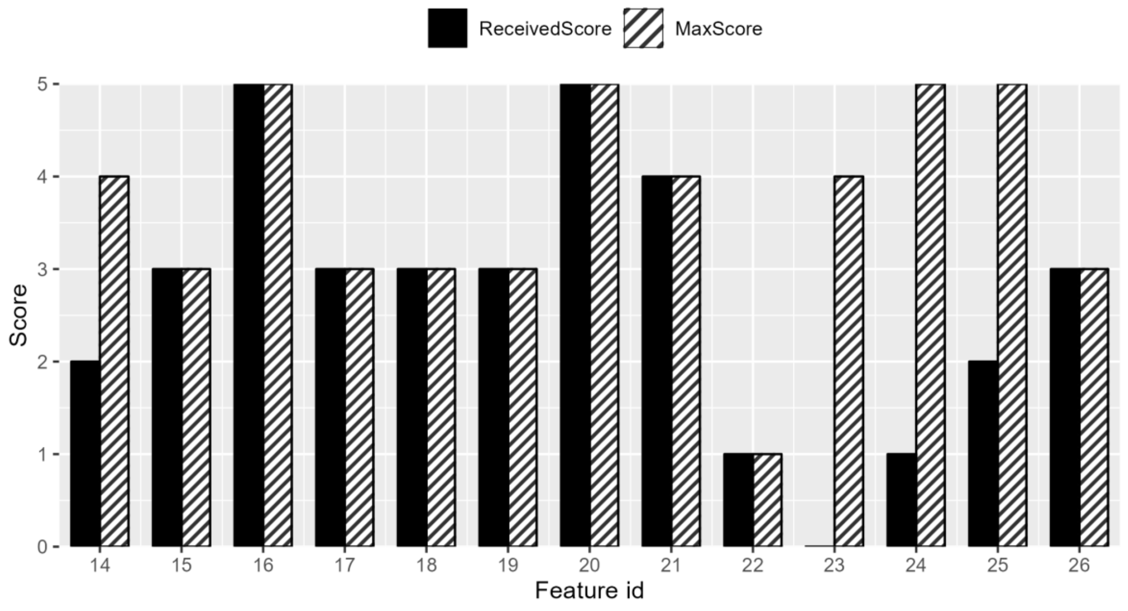

Figure 18. Brownie scores against maximum scores on features 14 – 26.

As shown in Figure 18, there are four five-point features between the features 14 and 26, and Brownie fully supports two of them, and they are related to project size and importing external tools and plugins. The two five-point features that Brownie fails to fully support are related to example/tutorial projects, and how up-to-date the documentation is. Figure 19 shows the received scores against the maximum ones between the features 27 and 40.

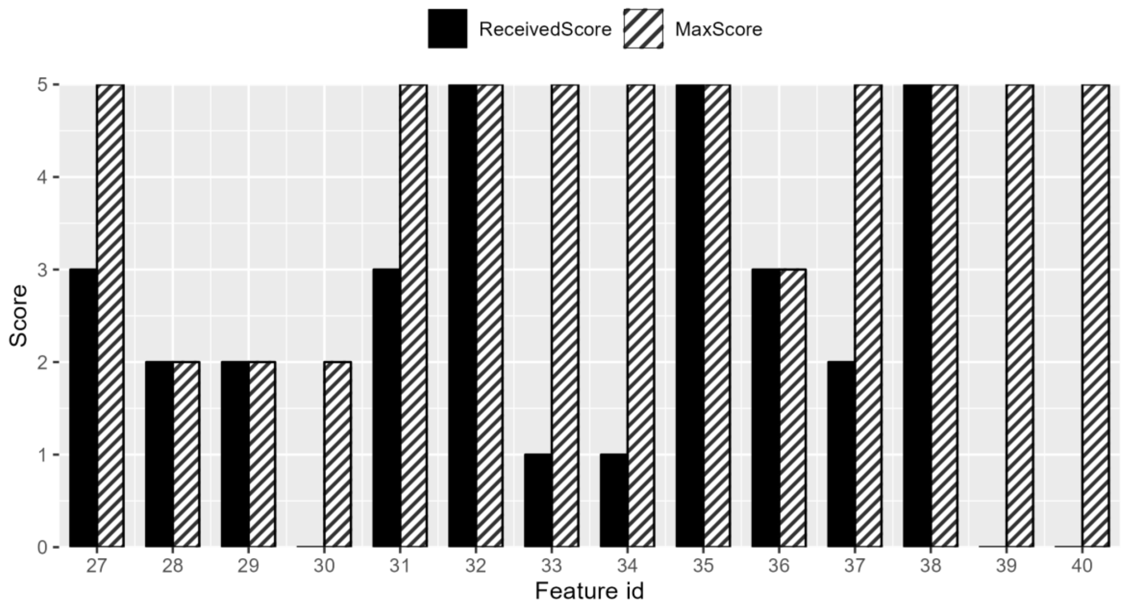

Figure 19. Brownie scores against maximum scores on features 27 – 40.

The features from 27 to 40, displayed in Figure 19, show that Brownie fails to provide full support on seven five-point features out of ten. These five-point features, that Brownie doesn't fully support, are related to documentation quality, the number of open GitHub issues, security, Snyk's score, the balance between new features and balance, pre-requirement installation process, and development environment installation process.

7. Comparative analysis of the results

In this chapter, the results that each development environment received are compared and analysed to identify what commonalities and differences they have, and which environment provides the most comprehensive support for Ethereum developers.

As the evaluation criteria doesn’t have an acceptance threshold for the reasons stated earlier, Kitchenham and Jones (1997d) explain that the scores from the evaluations must be compared against each other. This is done by first examining the score percentages, which each development environment received by characteristic categories. This comparison is displayed in Figure 20. After this, the scores of all evaluated development environments are presented feature by feature in Figures 21, 22, 23, and 24, and the identified differences are analyzed. The colors in the figures represent the theme color of each development environment, while black displays the maximum available score on each feature.

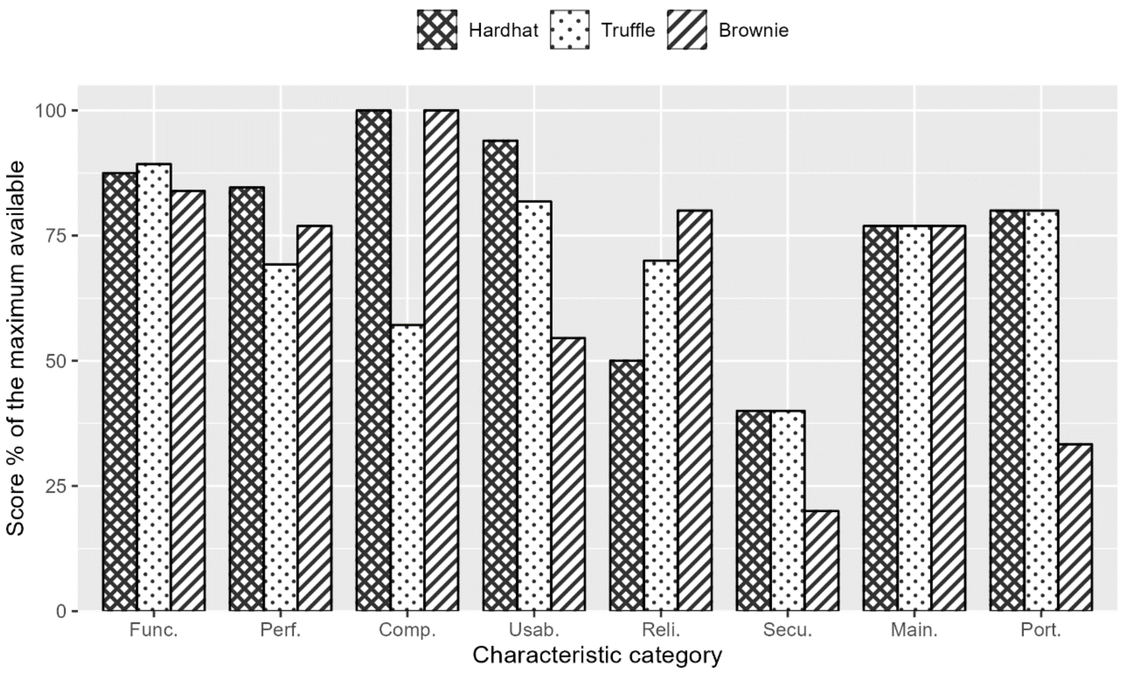

Figure 20. Comparison of results by characteristic categories.

As seen in Figure 20, for the most parts, the evaluated development environments performed quite similarly across the characteristics, maintainability being the prime example, in which every environment received the same score. Hardhat either won or shared the first place in six characteristic categories, while Brownie did so in three, and Truffle in four.

On functional suitability, all the development environments provide a very similar level of support for the developers, but Truffle wins the category narrowly with a percentage of 89, followed by Hardhat with 88, and then Brownie with 84. On performance efficiency Hardhat received the best score with 85 percentages, followed narrowly by Brownie with 77 and Truffle with 69. Hardhat and Brownie received the maximum available score on compatibility, while Truffle only 57 percent. On usability Hardhat won with a decent margin of 12 percentage points over Truffle, while Brownie received only 55 percent of the maximum score. Brownie scored the highest on reliability with 80 percent, followed by Truffle with 70 percent, and Hardhat with 50 percent. Hardhat and Truffle shared the top place on security with 40 percent, while Brownie received only 20 percent. All three development environments scored the same 77 percent on maintainability. Hardhat and Truffle shared the first place on portability with 80 percent, while Brownie scored only 33 percent.

Figure 21 displays the comparison of scores between the features 1 – 10.

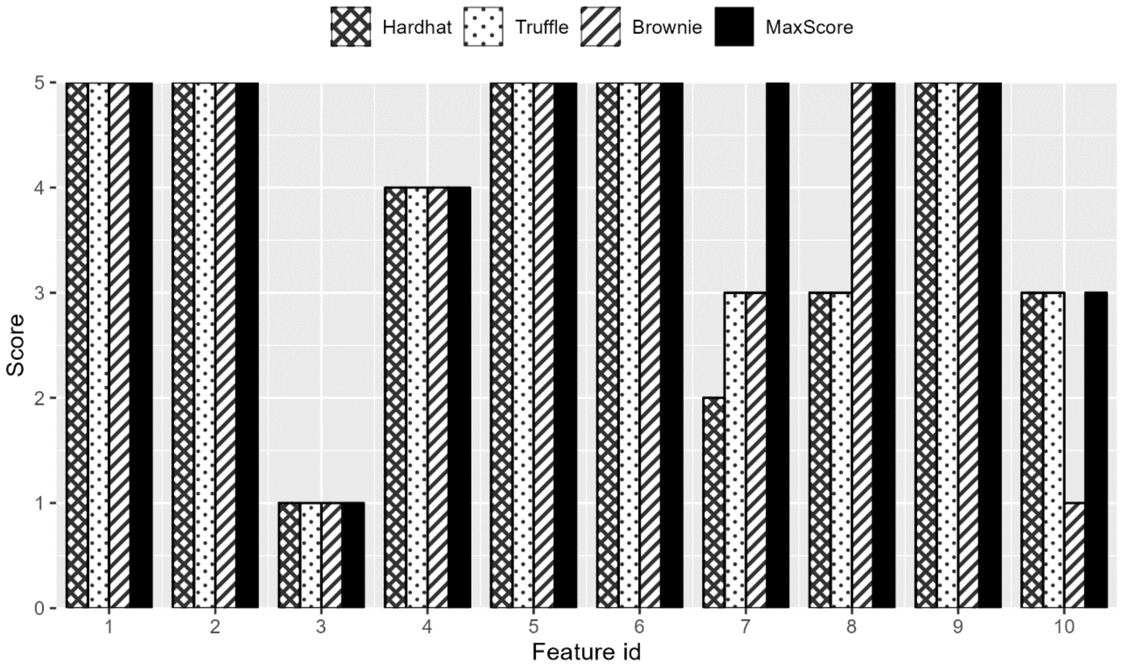

Figure 21. Comparison of scores between features 1 to 10.

As shown in Figure 21, all the development environments receive the maximum score on the first six features, and to do so, the development environment must come out-of-the-box with a testing functionality/framework along with a documentation about how to use it, it must support mainnet forking (i.e., forking the state of the Ethereum mainnet for testing purposes), compiling the smart contracts automatically when running tests, providing debugging support with a documentation out-of-the-box, and supporting deploying out-of-the-box. None of the environments provided full support on gas optimization, which is the feature number seven. Truffle and Brownie provide a tool or functionality to evaluate gas usage, while Hardhat provides documentation on how an external tool or plugin can be imported to the development environment. Brownie is the only one to provide full support on ensuring the security of the smart contracts, but it must be noted that the integrated tool is MythX, which requires a paid subscription to be used. Both Hardhat and Truffle provide instructions on how to integrate external tools. All the development environments support parallel frontend development within the same environment by providing an interface, in the form of a localhost port, for the frontend to connect to. Hardhat and Truffle worked as expected during the evaluation, whereas at first Brownie failed to run tests and to start a local blockchain. As Brownie was successfully installed, it should be able to execute all the functionalities it provides. However, the solution was related to the installation process (i.e., using pip instead of pipx, which Brownie suggests to be used for installation), and it is discussed in greater detail with the features 39 and 40.

Figure 22 displays the comparison of scores between the features 11 – 20.

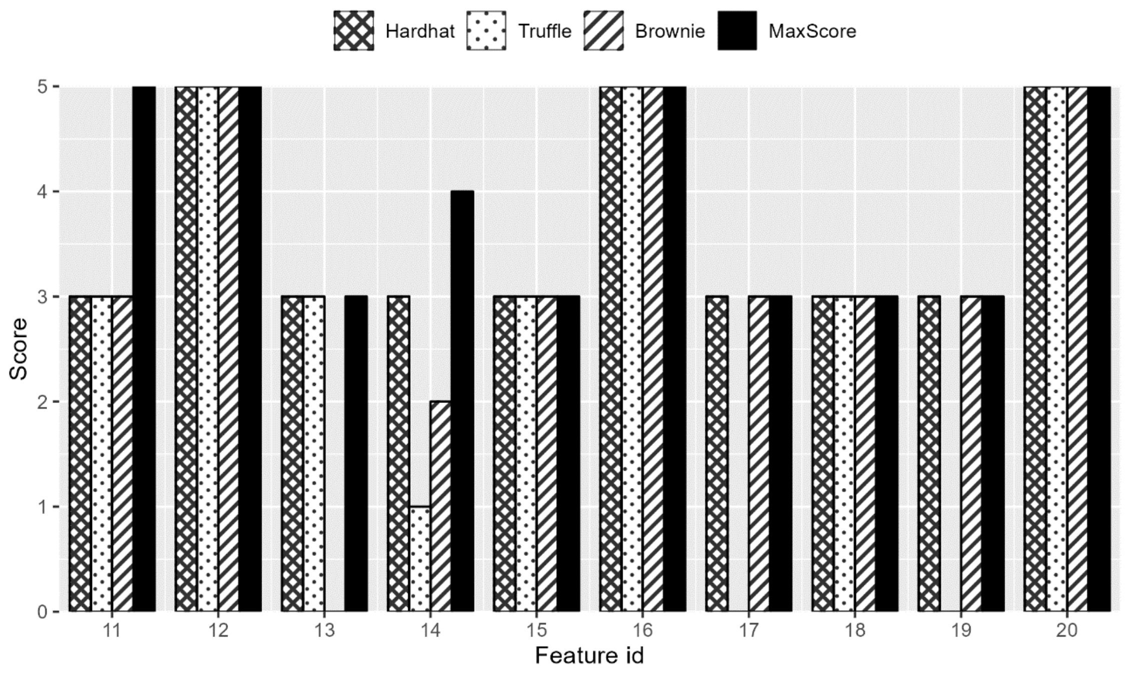

Figure 22. Comparison of scores between features 11 to 20.

On features between 11 and 20 displayed in Figure 22, all the development environments received the highest available score together on five features, and out of these five features, three are five-point features. To receive a full score on feature 11, the development environment must receive a full score of 43 on features between 1 and 10. Receiving this score would mean that the development environment provides a full support on every identified main Ethereum development environment functionality. As every evaluated environment has some functionalities that could be improved, none of them shall receive the full score. Every development environment supports both Solidity and Vyper high-level programming languages, and they receive the maximum score on feature 12. Both Hardhat and Truffle provide an extension for VS Code and receive the full score, while Brownie does not, and thus doesn't receive a single point. The purpose of the extension is to provide all the necessary and useful editor functionalities (e.g., syntax highlighting, shortcuts to editor) available within the editor. On feature 14, Hardhat receives the highest score in three points, as the out-of-the-box testing functionality works noticeably fast. Brownie scores 2 points as no attention was paid to the matter during the evaluation, and Truffle receives only a single point, because the out-of-the-box testing functionality is slow (e.g., executing five tests takes 21 seconds). None of the development environments are so resource heavy that using them could be noticed while using the computer, nor do they have any restrictions on the project size. On supporting one of the most popular Ethereum development tools, Hardhat and Brownie support all three selected tools (Open Zeppelin, Ganache, Waffle), while Truffle supports only Ganache. However, all three development environments allow external tools and plugins to be imported inside the environment.

Figure 23 displays the comparison of scores between the features 21 – 30.

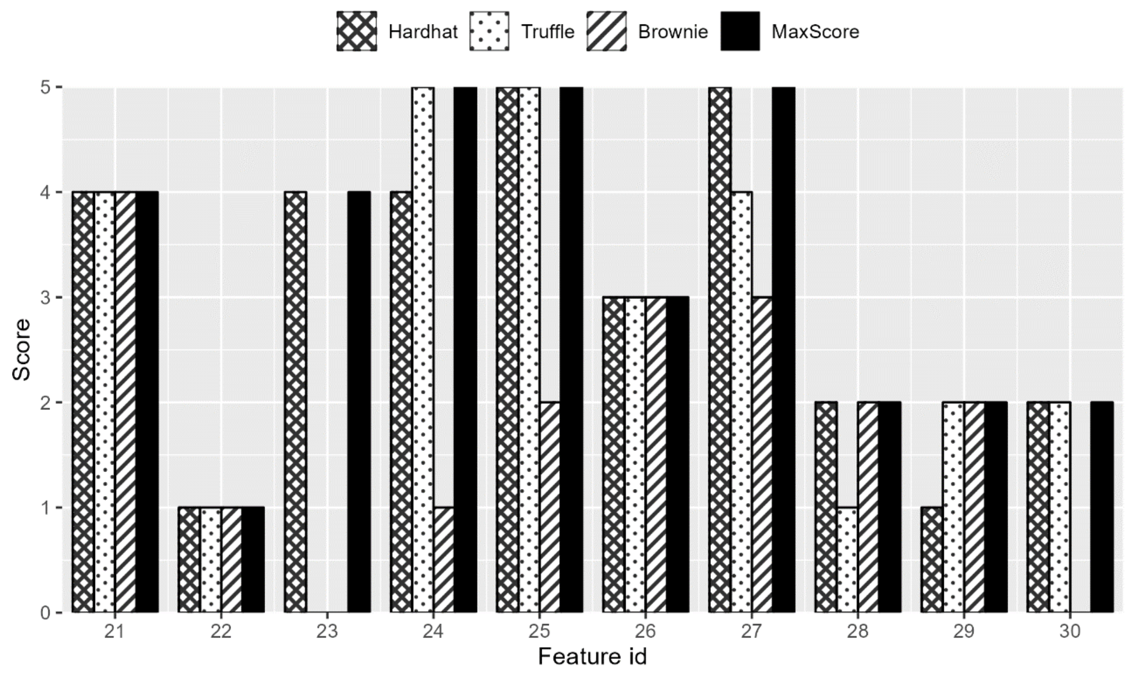

Figure 23. Comparison of scores between features 21 to 30.

As seen in Figure 23, all the development environments receive maximum scores on the features 21 and 22, and it means that the suitability and the main features of the development environment must be identifiable both on the homepage and on the GitHub repository. Hardhat is the only environment, which provides up-to-date documentation on how to install the dependencies, which is evaluated as feature 23. Truffle provides both smart contract- and Dapp tutorial projects, with up-to-date instructions, while Hardhat provides a good smart contract example, but the Dapp example project's frontend should be updated to follow the current best practices. Brownie has a smart contract example as well, but it still receives a single point as the Dapp examples don't have a good enough documentation to follow along, and for example testing doesn't work on these templates, at least out-of-the-box. Even though the installation process of Brownie was successful and all the documented dependencies, and those that are actually required, but which the documentation fails to mention, were installed, it is possible that the testing not working might be related to the environment. However, the fact that testing does work on the smart contract tutorial project would suggest otherwise. Both Hardhat and Truffle have their documentation up to date, while Brownie's documentation contains some outdated information (e.g., broken links, installation process should be updated). Feature 26 evaluates does the development environment's documentation include instructions how to import external tools within the environment, and all three evaluated environments provide that information. Regarding the overall quality of the documentation, Hardhat rises above the others. Its documentation is clear and well-structured, it provides code examples, and there is not much to be improved. Truffle also has excellent documentation as it provides code examples with a decent structure. It could improve it however by including the instructions to install dependencies within the documentation or at least explaining that the information for that can be found at the beginning of the tutorial. Brownie's documentation quality is what is expected from a technical documentation, as it does not lack in any aspect, but nor does it stand out either. Feature 28 is about having an own platform for discussion. Both Hardhat and Brownie have a discord community, while Truffle uses GitHub discussion, which isn't active. The commands used within Truffle and Brownie are simple and easily remembered, while Hardhat's are a bit longer, and storing the commands to a text file might be beneficial, at least when starting out to use it. Installing both the dependencies and the development environment, and starting a new project, can be done in under 10 minutes on Hardhat and Truffle, while with Brownie it took more than that.

Figure 24 displays the comparison of scores between the features 31 – 40.

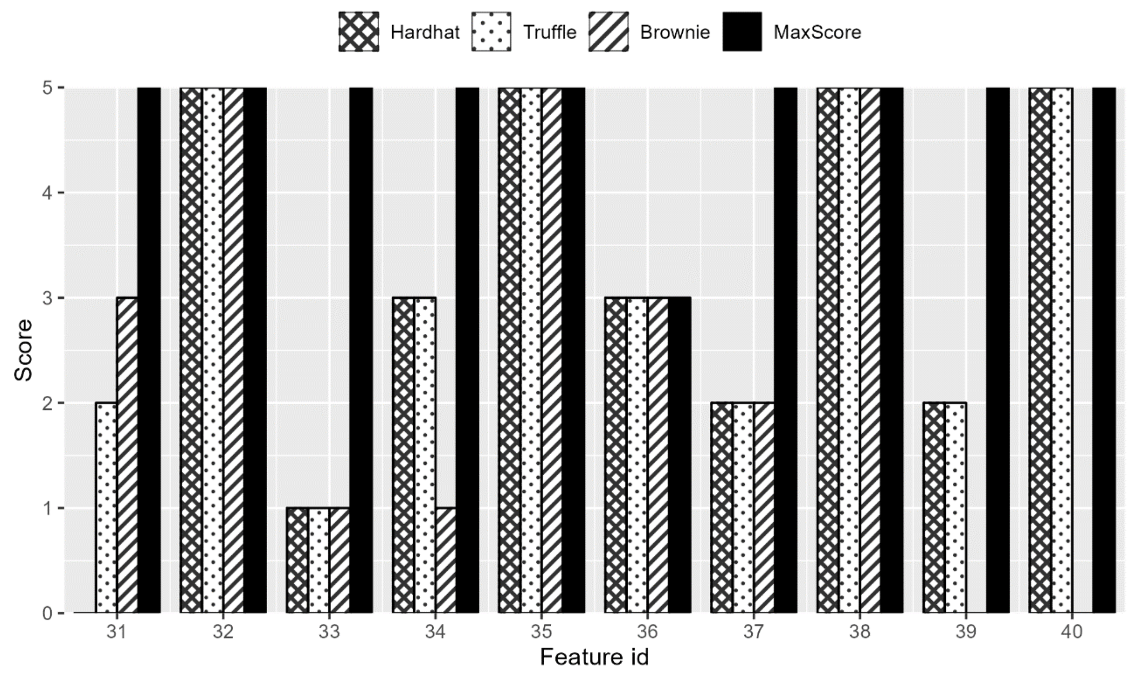

Figure 24. Comparison of scores between features 31 to 40.

As shown in Figure 24, feature 31 rewards points based on the number of open issues from the past 30 days in the GitHub repository. Brownie receives the highest score with 12 open issues, Brownie the second highest with 26, and Hardhat fails to get a single point with 42 open issues. As explained earlier, while having a lot of open issues doesn't necessarily indicate about a good quality and secure software, it can also be seen as a bright positive, as the community of the software is actively trying to improve the quality of the product. In a long run this approach might be more beneficial for the development environment. All the development environments score the highest available score of five points in feature 32, as none of them have any regular downtime, when the developer would not be able to use the environment. All the development environments have security vulnerabilities on their tutorial/example project dependencies. Hardhat has two, Truffle eight, and Brownie twelve critical dependencies, and they all receive a score of one on this feature. Next feature gives scores based on the package's health score on Snyk, which is takes into consideration the popularity, maintenance, security, and community of the open-source packages. Hardhat's score is 89 and Truffle's 86, and they both receive three points, while Brownie receives only a single point, with the package score of 79. All the development environments receive the full scores on features 35 and 36, as they all enable developers to change some of the functionalities to use alternative solutions, and all the development environments can be updated from terminal in under three minutes. All the development environments have a decent balance between introducing new features and technologies and having the stability by not introducing breaking changes. Feature 38 evaluates can the development environment be used on the main operating systems (i.e., Linux, macOS, Windows) and as all the environments in the evaluation can be used, they all receive the maximum amount of score. Regarding installing the pre-requirements/dependencies of the development environment, all the environments could do a better job. Hardhat and Truffle receive three points as they provide the information about pre-requirements in a tutorial, which is separate from the documentation. Out of these two Hardhat provides better support, as it has instructions on how to install the pre-requirements, whereas Truffle provides only links to the technologies it uses depends on. Brownie receives zero points, because while it does actually include the pre-requirement information within the documentation, the provided information is either outdated or simply wrong, as the development environment can be installed, but all the functionalities do not work. On the last feature, Hardhat and Truffle receive a full score of five points, as they can be installed from the terminal in under two minutes. Brownie receives zero points, because of the reasoning on the previous feature, although it can also be installed from terminal in under two minutes.

8. Discussion

In this chapter a summarization of the achieved results is presented, the importance of the results for different stakeholders are discussed, the used research method is evaluated by examining the produced results, and possible shortcomings of the study are discussed.

8.1 Summarization of the results

Overall, all the evaluated Ethereum development environments (i.e., Hardhat, Truffle, and Brownie) received for the most part quite similar evaluation scores across the characteristic categories. The maximum available score from the evaluation is 164, and Hardhat received the most points with a score of 136, which translated to around 83 percent of the available points. Truffle was the second-best development environment with a score of 127 points, which translated to around 77 percent, and Brownie was third with a score of 114, which in turn translates to around 70 percent. When it comes to either winning or sharing the first place in a characteristic category, Hardhat did so in six categories, while Truffle in four, and Brownie in three.

Hardhat performed well throughout the evaluation, as it provides full support for a lot of features, while lacking greatly only on a few. Hardhat shines by offering a fast, highly modifiable development environment with excellent documentation. Hardhat promotes a philosophy of enabling developers to import their own existing tools within the development environment, and the results are well in line with that. However, as Ethereum is constantly growing and new developers are transcending over, Hardhat could either provide even more support in the form of tools out-of-the-box, or maybe create a separate package which would be aimed at more inexperienced Ethereum developers, who don’t have any experience or preferences to which smart contract development tools they would like to use.

Truffle is the most popular development environment according to the amount of GitHub stars, but it came second in this evaluation. One of the biggest features that Truffle is lacking is testing, as executing tests simply takes too much time compared to the modern standards. However, besides testing, Truffle is a tested development environment, which has had time to mature, and this can be seen for example in the fluidity of operating the environment, and in a comprehensive documentation. The boxes (i.e., template projects) are a great way to kickstart a new project.

Brownie came last in the evaluation, but it must be noted how much the failure to provide correct instructions for the installation process affects the final score. If Brownie would have provided that instruction and worked correctly after installation, it would have received a total of 130 points, and it would have come second in the evaluation. This shows that Brownie is by no means a bad development environment in general once it is up and running correctly.

8.2 How the results compare to the literature review

Looking at the commonly complained aspects of Ethereum development in the literature review, the first common complaint was about not having a tool to ensure the security of the smart contracts. Brownie was the only one to come with an integrated security functionality, although the integrated tool was MythX, which requires a subscription to be used. Hardhat and Truffle provided instructions on how to integrate security related tools into the environment. Related to the topic, in the literature review praised Slither is available to be used through GitHub actions, so every development environment can use it in that way, and it could be one explanation of why Hardhat and Truffle ship without a security related tool.

A second complaint was about the need for a tool which would optimize the gas usage automatically at the source code level, and none of the development environments supported this feature fully. Truffle and Brownie provided a functionality to view the amount of gas that calling a function in the smart contract costs, while Hardhat provides instructions on how to import a plugin inside the environment to provide the same functionality that Truffle and Brownie ships with.

A third complaint that was mentioned often was about not having tools to debug the code. All the evaluated development environments come with a functionality to debug the code by logging, and Truffle provides even a debugger out-of-the-box. Debugging by logging events is for most of the people the way they learn to debug the code, and the fight over is it better to debug the code with a debugger or by logging is an ongoing argument, and for some people providing a debugger is without a doubt a major positive.

Comparing these common complains that occurred in the literature review to the achieved results, it can be said that Ethereum development environments provide a lot of the required functionalities out-of-the-box and they do overall ease the development process, but they can still do a better job by including a more comprehensive and effective tools and functionalities.

8.3 Importance of the results

As Ethereum will likely continue to grow, the results of this study can be used for multiple purposes by different stakeholders, main ones likely being Ethereum researchers and developers, development environment communities, companies behind the development environment development, and companies, which develop Ethereum based products.

The achieved results provide a good starting point for evaluating the Ethereum development environments in academia. As stated before, to the author’s best knowledge, no academic studies have been conducted to explore this topic, so this paper shall provide the base for the topic for others to expand from. As Ethereum is a quickly advancing technology, it is necessary for academia to produce new information to support its development, and all the other aspects related to Ethereum. Some possible directions to continue the academic research about the topic are listed in the next chapter.

For Ethereum developers, the results will provide a valuable insight of the differences that exist between the currently popular Ethereum development environments. Using the achieved results, the developers can compare the characteristics of each development environment and make an informed decision of which one they choose to use for developing Ethereum related software.

For both the communities and companies behind the development of development environments, the results provide direct feedback on those environments that were included in the evaluation, and those environments that were not included can use the created evaluation criteria to evaluate their own development environment to see how well they do support the needs of the developers.

The last stakeholder, who is likely to use the achieved results, is companies developing Ethereum based products. They can use the results to figure out which development environment they want to choose to be used in their organization, as well as what kind of support developers typically need while developing Ethereum-based products.

8.4 Evaluation of the results and the used research method

A customized version of the DESMET evaluation method was used in this study and doing so brought some positives and some negatives to the study. The DESMET method was changed by including the importance of the feature within the evaluation criteria. For example, providing a testing framework out-of-the-box is an important feature, so the maximum available score for it is five points, whereas providing a plugin for editor is more of a nice-to-have feature, as while it supports the developer in the task, it is not an essential part of the development environment, so the maximum score for providing it is only three points.

Importing the importance within the evaluation criteria rises some problems, like having features rewarding different amount of maximum scores, as providing full support for features of different level of importance must not receive the same amount of score (e.g., providing a testing framework with a comprehensive documentation is highly important and it rewards five points, while offering a plugin for VS Code is more of a nice-to-have feature, so it can reward only a maximum of three points). Combining this with the fact that there are some simple yes/no features on the feature list and the evaluation criteria might contain some gaps between the possible scores, will lead to some problems when presenting the achieved results and evaluating the correctness of the achieved results.

However, to solve the problem of how to present the results, the results are presented by using multiple different types of graphs to visualize the achieved results. First on the chapter six, the results of each evaluated development environment are presented by displaying on the first type of graph the percentage of achieved scores of each characteristic categories, followed by displaying on the second graph the achieved percentage score on each feature. After this, all the features are presented along with the evaluated score and the maximum available score in absolute numbers. Following the presentation of individual scores, on the chapter seven the achieved percentage scores by characteristic categories of every development environment are displayed in the same graph to visualize how their scores compare between each other. After this all the features are gone through while displaying the scores of each development environment against the maximum available score.

While the evaluated development environments shared a lot of commonalities, for academic purposes it would be beneficial to be able to draw more differences between the environments. In practice this could be done for example by focusing on a specific characteristic category and creating the feature set in a greater detail so the odds of receiving the same scores reduces, or by increasing the number of features in the set. The future directions are discussed more in the next chapter.

The positive side of including the importance within the evaluation criteria is the straightforwardness of reading the results and having a full control on setting the importance to each feature separately, as another option would be to have either a common multiplier on a certain character category, or to have a separated multiplier on each feature. Having the same multiplier on every feature inside the characteristic category is challenging, because it would mean that each feature should be created in a way that they all share the level of importance. The problem with having a separated importance on every feature is related to presenting the results, as displaying the results in a same graph would require importing the importance of the feature to the received score. With these alternative options, the choice of importing the importance within the evaluation criteria seems justified and a correct decision.

9. Conclusion

In this chapter, the research problem is answered by answering research questions, limitations and threats to the validity of the study are examined, and possible future directions from the study are discussed.

9.1 Answer to the research problem

The research problem of the study was to examine which tools and functionalities an Ethereum developer needs to have in their development environment to develop a high-quality software product efficiently and evaluate how well the currently existing solutions fulfill these requirements. This research problem was set to be answered through the following three research questions:

RQ1	What are the requirements for developing and deploying smart contracts into the Ethereum blockchain and how can those requirements be categorized?
RQ2	How can a development environment fulfill these requirements?
RQ3	How do the existing development environments meet the requirements?

As an answer for RQ1, functional requirements set for the Ethereum development environments are related to testing, mainnet forking, debugging, compiling, deploying, gas optimization, ensuring the security, enabling parallel frontend development, providing language support for both Solidity and Vyper, providing an extension for the editor, enabling the integration of external tools within the environment, providing tutorials and examples of Ethereum projects, and providing information on how to develop Ethereum projects. On top of these Ethereum related requirements, the development environment must also fulfill the requirements of a high-quality software product. According to the ISO/IEC standard 25010:2011, these characteristics can be categorized into functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability.

Regarding RQ2, Ethereum development environments can fulfill these requirements by providing support for the Ethereum related functional requirements as well as for the characteristics of a high-quality software. To evaluate the level of support a development environment provides for an Ethereum developer, a feature list including an evaluation criteria was created to score the amount of support the development environment provides. This feature list contains 40 features across the characteristic categories of a high-quality software, including the Ethereum specific functional requirements.

As for RQ3, a total of three Ethereum development environments were evaluated. Hardhat performed best at the evaluation and received around 83 percent of the maximum points, Truffle came second with around 77 percent, and Brownie last with around 70 percent of the maximum points. Out of the eight characteristic categories, only in security all the evaluated development environments failed to receive at least 50 percent of the maximum score. All three evaluated development environments performed well in the functional suitability characteristic category, as Hardhat received 88 percent of the maximum points, while Truffle did 89 and Brownie 84 percent. Based on these points, it can be said that the currently existing development environments provide at least a decent support in all characteristic categories but one, while providing good support in Ethereum related functional requirements.

9.2 Limitations and threats to validity

The study has one major limitation, and it is excluding the browser-based Ethereum development environments from the feature set, meaning that the feature set is not suitable for evaluating browser-based development environments. In the literature review mentioned and praised Remix is this kind of development environment, and it is often recommended as a quickest and simplest way to start developing Ethereum smart contracts.

Another limitation in the study is that it does not take into consideration the quality of the correctly operating functionalities of development environments, except the performance speed of the functionalities. This can lead to inaccurate results as in a theory, a development environment might be declared to be the best because it provides a lot of functioning functionalities out-of-the-box, while the environment that comes second, comes out-of-the-box with a more high-quality functionalities, which produce more accurate results than the functionalities provided by the environment that won the evaluation. As this can be a result of an informed decision of shipping the development environment without a certain functionality and providing information to the user on how to import the tool or functionality of their choosing, not winning the evaluation because of this doesn't represent the whole picture.

The existence of any biases was countered by implementing a semi-systematic process for reviewing the scientific papers for the study. The systematic literature process includes a systematic mapping of the papers included in each step, an inclusion- and exclusion criteria, used databases, and a data extraction form. To increase the transparency, the process of each step in the systematic process is documented.

It is necessary to recognize the ambitiousness of the study in trying to evaluate as objectively as possible something, that is by its nature a someone's preference of doing a specific task. As stated previously, development environments/frameworks are built on a specific idea of the most optimal way to do software development. Therefore, the achieved results must not be understanded as a ranking of the absolute best development environments, but rather as a good starting point for examining the requirements towards Ethereum development environments, and how well the evaluated environments fulfill these requirements.

9.3 Future work

As in the author's best knowledge no academic research has been conducted to evaluate Ethereum development environments, the produced feature set along with the evaluation criteria can be used for multiple different purposes.

First option is to continue to improve the evaluation criteria. As explained in the threats to validity, the topic under evaluation is difficult to be evaluated in an objective manner, and the evaluation criteria can be improved by examining the topic from other points of views. The feature set could also be improved in a way that browser-based Ethereum development environments could also be included in the evaluation.

The second way to continue the research would be examining some of the characteristics in greater detail. The features within the functional suitability category for example would be a great choice, as by doing so the quality of the provided functionalities could also be evaluated and taken into consideration when determining the best Ethereum development environment.

The third option would be updating the feature set along with the evaluation criteria in the future, when the requirements towards the Ethereum development environments have changed. As Ethereum development and the blockchain technology as a whole are advancing with great speed, it can be said with a great certainty that sooner or later the requirements towards the Ethereum development environment will change. When this happens, the evaluation criteria presented in this study needs to be updated with the current knowledge about the topic to evaluate those features that the smart contract developers are requiring at that moment.

The fourth option is to use the produced results as they are to serve some purpose in other studies, such as comparing the results that this feature set produces against the results that produced by using another feature set.

References

Brownie. (2022). *Brownie.* Retrieved June 6, 2022, from https://eth-brownie.readthedocs.io/en/stable/

Budgen, D., & Brereton, P. (2006, May). Performing systematic literature reviews in software engineering. In *Proceedings of the 28th international conference on Software engineering* (pp. 1051-1052).

Burkhardt, D., Werling, M., & Lasi, H. (2018, June). Distributed ledger. In *2018 IEEE international conference on engineering, technology and innovation (ICE/ITMC)* (pp. 1-9). IEEE.

Canfora, G., Di Sorbo, A., Fredella, M., Vacca, A., & Visaggio, C. A. (2021, September). iSCREAM: a suite for Smart Contract REAdability assessMent. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 579-583). IEEE.

Chen, T., Feng, Y., Li, Z., Zhou, H., Luo, X., Li, X., ... & Zhang, X. (2020). Gaschecker: Scalable analysis for discovering gas-inefficient smart contracts. *IEEE Transactions on Emerging Topics in Computing*, *9*(3), 1433-1448.

Chen, X., Liao, P., Zhang, Y., Huang, Y., & Zheng, Z. (2021, March). Understanding code reuse in smart contracts. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 470-479). IEEE.

CoinMarketCap. (2022). *Top smart contract tokens by market capitalization*. Retrieved June 6, 2022, from https://coinmarketcap.com/view/smart-contracts/

Corbin, D. S. (1991). Establishing the software development environment. *Journal of Systems Management*, *42*(9), 28.

Danson, M., & Arshad, N. (2014). The literature review. *Research methods for business and management: A guide to writing your dissertation*, 37-57.

Di Angelo, M., & Salzer, G. (2019, April). A survey of tools for analyzing ethereum smart contracts. In *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)* (pp. 69-78). IEEE.

Dowling, M. (2021). Fertile LAND: Pricing non-fungible tokens. *Finance Research Letters*, 102096.

Drescher, D. (2017). *Blockchain Basics: A Non-Technical Introduction in 25 Steps*. Apress.

Ethereum. (2022a). *Decentralized finance (DeFi).* Retrieved January 31, 2022, from https://ethereum.org/en/defi/

Ethereum. (2022b). *Ethereum virtual machine.* Retrieved April 4, 2022, from https://ethereum.org/en/developers/docs/evm/

Ethereum. (2022c). *Ethereum wallets.* Retrieved April 4, 2022, from https://ethereum.org/en/wallets/

Ethereum. (2022d). *Gas and fees.* Retrieved April 3, 2022, from https://ethereum.org/en/developers/docs/gas/

Ethereum. (2022e). *Introduction to dapps.* Retrieved May 5, 2022, from https://ethereum.org/en/developers/docs/dapps/

Ethereum. (2022f). *Introduction to smart contracts.* Retrieved April 27, 2022, from https://ethereum.org/en/developers/docs/smart-contracts/

Ethereum. (2022g). *Set up your local development environment.* Retrieved June 5, 2022, from https://ethereum.org/en/developers/local-environment/

Ethereum. (2022h). *Smart contract languages.* Retrieved May 11, 2022, from https://ethereum.org/en/developers/docs/smart-contracts/languages/

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Gao, Z. (2020, December). When deep learning meets smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (pp. 1400-1402).

Gupta, B. C., & Shukla, S. K. (2019, October). A study of inequality in the ethereum smart contract ecosystem. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)* (pp. 441-449). IEEE.

Haefliger, S., Von Krogh, G., & Spaeth, S. (2008). Code reuse in open source software. *Management science*, *54*(1), 180-193.

Hardhat. (2022). *Hardhat.* Retrieved June 6, 2022, from https://hardhat.org/

Hartel, P., Homoliak, I., & Reijsbergen, D. (2019). An empirical study into the success of listed smart contracts in ethereum. *IEEE Access*, *7*, 177539-177555.

Hwang, S., & Ryu, S. (2020, June). Gap between theory and practice: An empirical study of security patches in solidity. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (pp. 542-553).

ISO. (2022a). *ISO/IEC 25010:2011(en).* Retrieved April 28, 2022, from https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en

ISO. (2022b). *Standards.* Retrieved April 27, 2022, from https://www.iso.org/standards.html

Kitchenham, B. A. (1996a). Evaluating software engineering methods and tool part 1: The evaluation context and evaluation methods. *ACM SIGSOFT Software Engineering Notes*, *21*(1), 11-14.

Kitchenham, B. A. (1996b). Evaluating software engineering methods and tool part 2: Selecting an appropriate evaluation method – technical criteria. *ACM SIGSOFT Software Engineering Notes*, *21*(2), 11-15.

Kitchenham, B. A. (1996c). Evaluating software engineering methods and tool part 3: selecting an appropriate evaluation method – practical issues. *ACM SIGSOFT Software Engineering Notes*, *21*(4), 9-12.

Kitchenham, B. A., & Jones, L. (1997a). Evaluating software engineering methods and tool part 5: the influence of human factors. *ACM SIGSOFT Software Engineering Notes*, *22*(1), 13-15.

Kitchenham, B. A., & Jones, L. (1997b). Evaluating software engineering methods and tool part 6: identifying and scoring features. *ACM SIGSOFT Software Engineering Notes*, *22*(2), 16-18.

Kitchenham, B. A., & Jones, L. (1997c). Evaluating software engineering methods and tool part 7: planning feature analysis evaluation. *ACM SIGSOFT software engineering Notes*, *22*(4), 21-24.

Kitchenham, B. A., & Jones, L. (1997d). Evaluating software engineering methods and tool part 8: analysing a feature analysis evaluation. *ACM SIGSOFT Software Engineering Notes*, *22*(5), 10-12.

Kitchenham, B., & Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering.

Niazi, M. (2015). Do systematic literature reviews outperform informal literature reviews in the software engineering domain? An initial case study. *Arabian Journal for Science and Engineering*, *40*(3), 845-855.

Pierro, G. A., & Tonelli, R. (2021, March). Analysis of source code duplication in ethreum smart contracts. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 701-707). IEEE.

Pinna, A., Ibba, S., Baralla, G., Tonelli, R., & Marchesi, M. (2019). A massive analysis of ethereum smart contracts empirical study and code metrics. *IEEE Access*, *7*, 78194-78213.

Snyder, H. (2019). Literature review as a research methodology: An overview and guidelines. *Journal of business research*, *104*, 333-339.

Sujeetha, R., & Preetha, C. S. D. (2021, October). A Literature Survey on Smart Contract Testing and Analysis for Smart Contract Based Blockchain Application Development. In *2021 2nd International Conference on Smart Electronics and Communication (ICOSEC)* (pp. 378-385). IEEE.

Truffle Suite. (2022). *Truffle.* Retrieved June 6, 2022, from https://trufflesuite.com/docs/truffle/

Wang, S., Ouyang, L., Yuan, Y., Ni, X., Han, X., & Wang, F. Y. (2019). Blockchain-enabled smart contracts: architecture, applications, and future trends. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, *49*(11), 2266-2277.

Wohlin, C. (2014, May). Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering* (pp. 1-10).

Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, *151*(2014), 1-32.

Zarir, A. A., Oliva, G. A., Jiang, Z. M., & Hassan, A. E. (2021). Developing cost-effective blockchain-powered applications: A case study of the gas usage of smart contract transactions in the ethereum blockchain platform. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *30*(3), 1-38.

Zhang, P., Xiao, F., & Luo, X. (2020, September). A framework and dataset for bugs in ethereum smart contracts. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 139-150). IEEE.

Zou, W., Lo, D., Kochhar, P. S., Le, X. B. D., Xia, X., Feng, Y., ... & Xu, B. (2021). Smart Contract Development: Challenges and Opportunities. *IEEE Transactions on Software Engineering*, *47*(10), 2084-2106.

Ølnes, S., Ubacht, J., & Janssen, M. (2017). Blockchain in government: Benefits and implications of distributed ledger technology for information sharing. *Government Information Quarterly*, *34*(3), 355-364.

Appendix A. Evaluation form.

Id	**Description**	**Category**	**0 points**	**1 point**	**2 points**	**3 points**	**4 points**	**5 points**	**Max score**
1	How environment supports testing?	Functional completeness **(Functional suitability)**	No support	Comes with a testing functionality/framework, but it has shortcomings (e.g., performance is remarkably slow, documentation limited, etc.), and it can't be changed to an alternative solution	Provides documentation how external testing frameworks can be integrated into the environment			Comes with a testing functionality/ framework	5
2	Environment supports mainnet forking for testing?	Functional completeness **(Functional suitability)**	No					Yes	5
3	Environment automatically compiles the code when running tests (if there are any changes)?	Functional completeness, Usability **(Functional suitability)**	No	Yes					1
4	How environment supports debugging?	Functional completeness **(Functional suitability)**	No support	Provides documentation how an external debugger/ debugging tools can be integrated into the environment			Comes with debugging tool(s)		4
5	How environment supports compiling?	Functional completeness **(Functional suitability)**	No support	Provides documentation how a compiler can be integrated into the environment or how the source code can be compiled	Comes with a compiler which can't be configured, or the configuration isn't explained in the documentation			Comes with a compiler and a documentation how it can be configured	5

Id	Description	Category	0 points	1 point	2 points	3 points	4 points	5 points	Max score
6	How environment supports deploying?	Functional completeness **(Functional suitability)**	No support	Provides documentation how deploying can be done				Comes with a tool or a functionality to deploy smart contracts	5
7	How environment supports gas optimization?	Functional completeness **(Functional suitability)**	No support		Provides documentation how an external tool or plugin can be integrated into the environment	Comes with a tool or a functionality to evaluate gas usage		Comes with a tool or a functionality to optimize gas usage	5
8	How environment supports ensuring the security of the smart contracts?	Functional completeness **(Functional suitability)**	No support			Provides documentation how an external tool or plugin can be integrated into the environment		Comes with a tool or a functionality to improve the security	5
9	Does the development environment enable parallel frontend development (e.g., exposes interface for the frontend to connect)?	Functional completeness **(Functional suitability)**	No					Yes	5
10	Development environment worked as intended during the evaluation?	Functional correctness **(Functional suitability)**	No, either constantly minor troubles or some functionality simply doesn't work	A few minor problems may occur (e.g., an unexpected need for configuration), but the provided functionalities still operate as expected		Yes, no problem occurs during the evaluation			3

Id	Description	Category	0 points	1 point	2 points	3 points	4 points	5 points	Max score
11	How much the environment supports and overall eases the development process?	Functional appropriateness **(Functional suitability)**	Not at all	Just a bit, lacking support at least in one functionality		Environment provides a decent support and benefits gained from using it are unquestioned, but some functionalities could be either improved or the environment is lacking some essential functionalities		Environment provides full support for developers in identified functionalities (score from feature ids 0-9 = 43)	5
12	Development environment supports Solidity and Vyper?	Functional completeness **(Functional suitability)**	No			Yes, only another		Yes, both	5
13	Development environment provides an extension for VS Code?	Functional completeness **(Functional suitability)**	No			Yes			3
14	Does the development environment operate as fast as expected?	Time behaviour (**Performance efficiency)**	No, user constantly (either on multiple functionalities or on an hourly basis) thinks how slow it is	No, and sometimes (either on some functionality or on a daily basis) the user thinks that it is slower than expected	Yes, user doesn't pay any attention to the matter	Yes, and sometimes (either on some functionality or on a daily basis) the user notices that it is even faster than expected	Yes, user constantly (either on multiple functionalities or on an hourly basis) thinks that it is faster than expected		4

Id	Description	Category	0 points	1 point	2 points	3 points	4 points	5 points	Max score
15	Does the development environment require so much resources that it negatively affects using the PC?	Resource utilization **(Performance efficiency)**	Yes, and it can be noticed constantly (on an hourly basis)	Yes, and it can be noticed sometimes (on a daily basis)		No, using the development environment doesn't negatively affect using the PC			3
16	Development environment has restrictions regarding the size of the project?	Capacity **(Performance efficiency)**	Yes					No	5
17	Can Open Zeppelin be imported (or is it by default) within the development environment?	Co-existence, Interoperability **(Compatibility)**	No			Yes			3
18	Can Ganache be imported (or is it by default) within the development environment?	Co-existence, Interoperability **(Compatibility)**	No			Yes			3
19	Can Waffle be imported (or is it by default) within the development environment?	Co-existence, Interoperability **(Compatibility)**	No			Yes			3
20	Environment allows importing plugins/ external tools within the environment?	Co-existence, Interoperability **(Compatibility)**	No					Yes	5
21	Where the suitability and the main features of the environment can be determined?	Appropriateness recognizability **(Usability)**	Unable to determine from the internet	From the internet	Official documentation	Either on the homepage or on the GitHub repository README	Both on the homepage and on the GitHub repository README		4

Id	Description	Category	0 points	1 point	2 points	3 points	4 points	5 points	Max score
22	Development environment documentation have a search function?	Learnability **(Usability)**	No	Yes					1
23	Environment documentation provides up-to-date information/instructions for installing the dependencies?	Learnability **(Usability)**	No				Yes		4
24	Development environment has smart contract and Dapp project tutorials/examples?	Learnability **(Usability)**	No project tutorials/ examples	Yes, but the example(s) has a lot of problems (docs outdated, confusing, etc.) and following it is highly challenging	Yes, but the example(s) has some problems (docs outdated, confusing, etc.), but tutorial is still fairly easily followable		Just one of the tutorials exists, and it is accurate, up-to-date, and easily followable	Yes, both tutorials exist, and they are accurate, up-to-date, and easily followable	5
25	Generally (as there is always some outdated information), does the documentation appears to be up to date?	Learnability **(Usability)**	No		Mostly (certain section is not up to date)			Yes	5
26	Documentation explains how plugins, external tools, or alternative functionalities can be imported to the environment?	Learnability **(Usability)**	Doesn't support importing any of these, or documentation doesn't explain how it can be done	Documentation is confusing and hard to follow	Documentation contains all the relevant information, but it has some shortcoming (e.g., confusing layout, missing commands etc.)	Documentation contains all the relevant information, and it provides examples of terminal commands and/or code examples			3

Id	Description	Category	0 points	1 point	2 points	3 points	4 points	5 points	Max score
27	How much the documentation could be improved?	Learnability **(Usability)**	Pretty much any change would be an improvement	A lot, there is major problems with structure, presentation, language etc.	Some, there is problems with structure, presentation, language etc.	A bit, the standard level, provides all the basic information in a decent format, but doesn't stand out in any way	Not by much, few minor changes and improvements to structure, presentation, language etc.	Documentation is nearly perfect, comes to personal preference whether any change would improve the documentation	5
28	Development environment has its own forum, in which developers can ask help from other developers?	Learnability **(Usability)**	No	Yes, but it isn't active (there are clearly more suitable platforms for discussion, such as StackOverflow)	Yes, and it is active (has at least 1000 members or most of the questions are being answered within two days)				2
29	Are the default commands to execute functionalities/ operate environment long and difficult? (e.g., majority of them contain more than 20 characters, at first it is beneficial to save them in a text file?)	Operability **(Usability)**	Yes	Some, but not all	No				2
30	Are you able to start a new project under 10 minutes with the environment (including the installation and possible configuration)?	Operability **(Usability)**	No		Yes				2

Id	Description	Category	0 points	1 point	2 points	3 points	4 points	5 points	Max score
31	How many open issues the GitHub repository has from the past 30 days?	Maturity **(Reliability)**	Over 40 open issues	40<=issues<=30	30<issues<=20	20<issues<=10	10<issues<=5	Less than 5 open issues	5
32	Under normal circumstances, development environment can be used any given time (e.g., not down every night at the same time)?	Availability **(Reliability)**	No					Yes	5
33	Can security flaws be detected, or is there any reason to questionable the security (including template projects)?	Confidentiality**,** Integrity **(Security)**	Security flaws can be detected during the evaluation	There are valid reasons to question the security of the development environment (e.g., critical vulnerabilities on dependencies)	The security of the development environment doesn't spark trust towards it, but no valid reasons to question it (e.g., vulnerabilities on dependencies)			No reason to question the security of the development environment	5
34	What is the package's health score on Snyk (https://snyk.io/advisor/)?	Confidentiality **(Security)**	Score <= 75	75< score <=80	80< score <=85	85< score <=90	90< score <=95	Score > 95	5
35	User can change some of the functionalities to alternative solutions (e.g., testing framework, development node)?	Modularity **(Maintainability)**	Unable					Yes	5

Id	Description	Category	0 points	1 point	2 points	3 points	4 points	5 points	Max score
36	How much effort it takes to update the development environment?	**(Maintainability)**	Requires a lot of work and searching for information (>10 min)	Requires few Google searches or reviewing documentation (<10 min)		Effortless, can be done from terminal under 3 minutes			3
37	What is the philosophy of the development environment about balancing new technologies/functionalities and stability?	**(Maintainability)**	An extreme, uses either outdated technologies or constantly introduces breaking changes with new technologies		A somewhat balanced, uses relatively new technologies and doesn’t often introduce breaking changes			A perfect blend, uses the newest technologies, but doesn't still introduce breaking changes on releases	5
38	Development environment can be used on Linux, macOS, and on Windows 10?	Adaptability **(Portability)**	No					Yes	5
39	Does the development environment have pre-requirements for installation, and how much effort installing those requires?	Installability **(Portability)**	Pre- requirements require a lot of effort (e.g., process is difficult and confusing, >15 min)		Pre- requirements require some effort (e.g., documentation about the process could be clearer, <15 min)			No pre- requirements, or can be done from terminal under 3 minutes with the instructions on documentation	5
40	How much effort does the development environment installing process require?	Installability **(Portability)**	Installation process is difficult, and it requires a lot of work (>10 min)		Requires few Google searches or some reading of the documentation (<10 min)			Can be installed from terminal under 2 minutes with the instructions on documentation	5

Appendix B. Evaluation results.

Id	Feature	Max score	Hardhat score	Truffle score	Brownie score
1	How environment supports testing?	5	5	5	5
2	Environment supports mainnet forking for testing?	5	5	5	5
3	Environment automatically compiles the code when running tests (if there are any changes)?	1	1	1	1
4	How environment supports debugging?	4	4	4	4
5	How environment supports compiling?	5	5	5	5
6	How environment supports deploying?	5	5	5	5
7	How environment supports gas optimization?	5	2	3	3
8	How environment supports ensuring the security of the smart contracts?	5	3	3	5
9	Does the development environment enable parallel frontend development (e.g., exposes interface for the frontend to connect)?	5	5	5	5
10	Development environment worked as intended during the evaluation?	3	3	3	1
11	How much the environment supports and overall eases the development process?	5	3	3	3
12	Development environment supports Solidity and Vyper?	5	5	5	5
13	Development environment provides an extension for VS Code?	3	3	3	0
14	Does the development environment operate as fast as expected?	4	3	1	2
15	Does the development environment require so much resources that it negatively affects using the PC?	3	3	3	3
16	Development environment has restrictions regarding the size of the project?	5	5	5	5
17	Can Open Zeppelin be imported (or is it by default) within the development environment?	3	3	0	3
18	Can Ganache be imported (or is it by default) within the development environment?	3	3	3	3
19	Can Waffle be imported (or is it by default) within the development environment?	3	3	0	3

Id	Feature	Max score	Hardhat score	Truffle score	Brownie score
20	Environment allows importing plugins/ external tools within the environment?	5	5	5	5
21	Where the suitability and the main features of the environment can be determined?	4	4	4	4
22	Development environment documentation have a search function?	1	1	1	1
23	Environment documentation provides up-to-date information/instructions for installing the dependencies?	4	4	0	0
24	Development environment has smart contract and Dapp project tutorials/examples?	5	4	5	1
25	Generally (as there is always some outdated information), does the documentation appears to be up to date?	5	5	5	2
26	Documentation explains how plugins, external tools, or alternative functionalities can be imported to the environment?	3	3	3	3
27	How much the documentation could be improved?	5	5	4	3
28	Development environment has its own forum, in which developers can ask help from other developers?	2	2	1	2
29	Are the default commands to execute functionalities/ operate environment long and difficult? (e.g., majority of them contain more than 20 characters, at first it is beneficial to save them in a text file?)	2	1	2	2
30	Are you able to start a new project under 10 minutes with the environment (including the installation and possible configuration)?	2	2	2	0
31	How many open issues the GitHub repository has from the past 30 days?	5	0	2	3
32	Under normal circumstances, development environment can be used any given time (e.g., not down every night at the same time)?	5	5	5	5
33	Can security flaws be detected, or is there any reason to questionable the security (including template projects)?	5	1	1	1
34	What is the package's health score on Snyk (https://snyk.io/advisor/)?	5	3	3	1
35	User can change some of the functionalities to alternative solutions (e.g., testing framework, development node)?	5	5	5	5
36	How much effort it takes to update the development environment?	3	3	3	3

Id	Feature	Max score	Hardhat score	Truffle score	Brownie score
37	What is the philosophy of the development environment about balancing new technologies/functionalities and stability?	5	2	2	2
38	Development environment can be used on Linux, macOS, and on Windows 10?	5	5	5	5
39	Does the development environment have pre-requirements for installation, and how much effort installing those requires?	5	2	2	0
40	How much effort does the development environment installing process require?	5	5	5	0