SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Automated Unit Testing of Solidity Smart Contracts in an Educational Context

Batuhan Erden

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Automated Unit Testing of Solidity Smart Contracts in an Educational Context

Automatisiertes Unit Testing von Solidity Smart Contracts im Bildungskontext

Author:	Batuhan Erden
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	Felix Hoops, M.Sc. & Burak Öz, M.Sc.
Submission Date:	15.11.2023

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.11.2023 Batuhan Erden

Acknowledgements and Code Repositories

I wish to convey my heartfelt gratitude to my advisors, Felix Hoops, M.Sc., and Burak Öz, M.Sc., for their strong leadership and support during the duration of my thesis, especially during times of uncertainty. Additionally, profound appreciation is directed to my supervisor, Prof. Dr. Florian Matthes, not only for his expertise in helping shape the thesis topic and introducing innovative ideas but also for affording me the opportunity to write this thesis under his chair for Software Engineering for Business Information Systems (SEBIS). I express further thanks to my advisors for granting comprehensive access to the teaching material for the Blockchain-based Systems Engineering (BBSE) course and the *BBSE Bank 2.0* project. Special acknowledgment goes to my family and friends, as well as my two musical bands, who provided unwavering support even in the most stressful times.

The code developed during this thesis is publicly accessible and can be found in the following GitHub repositories:

- **Comparison of Test Runner Frameworks:**
  https://github.com/erdenbatuhan/automated-smart-contract-tester-comparison
- **Automated Smart Contract Testing Service:**
  https://github.com/erdenbatuhan/automated-smart-contract-tester
- **Web Application:**
  https://github.com/erdenbatuhan/automated-smart-contract-tester-web

The following are some additional GitHub repositories that are not directly related to the main project, but still have some involvement:

- **Dockerized MongoDB Replicas:**
  https://github.com/erdenbatuhan/dockerized-mongodb-replicas
- **Example Usage of the Developed Helper Library for RabbitMQ:**
  https://github.com/erdenbatuhan/rabbitmq-playground

Abstract

In the rising era of modern computing, blockchain technology has emerged as a crucial player, enabling secure transactions within an immutable record-keeping system. Capturing major attention from both academia and industry, its growth has encouraged continuous development, especially in the area of smart contracts. Due to the unalterable nature of a blockchain, it is imperative to test smart contracts to guarantee that they are free of vulnerabilities before deployment. Therefore, automated testing of smart contracts has become an important notion that has also found its way into educational environments to build foundational knowledge.

This thesis proposes a scalable service for automated unit testing of Solidity smart contracts within an educational context, allowing students to upload their smart contract inputs, which are executed against instructor-provided tests to offer constructive feedback on their contracts. The comparative analysis of the testing tools — Truffle, Hardhat, and Foundry — informs the selection of the optimal one to be used for smart contract testing, considering factors like usability, development experience, features, performance, and containerization capabilities. The service is designed with a microservice architecture and developed with the chosen tool used for smart contract testing. Furthermore, it is containerized using Docker and orchestrated with Docker Compose. Following that, inter-service communication is facilitated through RabbitMQ for stability under high loads, and Docker Swarm is utilized to enable horizontal scaling.

The evaluation of the testing service encompasses security, stability, efficiency, and scalability, confirming its ability to handle the simultaneous load of multiple submissions in a secure and stable software package. The work concludes with discussion on the summary of the work, possible future directions, and the extensive documentation offered for future maintainability.

We claim that this testing service will significantly contribute to the technological developments in educational settings, aiding students in creating more secure, reliable, and robust smart contracts before deploying them in critical applications. By utilizing an automated smart contract tester, students can have their contracts evaluated against tests developed by instructors, enhancing the learning process and eliminating the need for them to write their own tests. Moreover, the scalable and load-balancing nature of the service will allow for a smooth user experience, even during times of heavy load.

Contents

1. Introduction

The first chapter of the thesis explains the motivation behind the work, sets out the objectives and purpose, and defines the research questions along with the core use case. It concludes by describing the overall structure of the thesis.

1.1. Motivation

Blockchains have recently become increasingly prevalent, revolutionizing how information is stored and processed; this technology is structured as a sequence of blocks where once a transaction is recorded in a block and the block is added to the chain, the record of that transaction is immutable and cannot be tampered with or changed [1, 2]. The concept of this decentralized, distributed, and public ledger system, along with its immutability, security, and transparency, has attracted significant attention amongst educational institutions and business enterprises [3].

For the continued advancement of blockchain technology, it is imperative that educational institutions keep pace with the latest developments. Consequently, educational programs have been integrating blockchain into their curricula, as evidenced by the Blockchain-based Systems Engineering (BBSE) course [4] at the Technical University of Munich (TUM), demonstrating the growing influence of this technology across several industries, such as finance, healthcare, and data analysis [5, 6, 7]. Moreover, the decentralized and accessible nature of this technology allows anyone to partake in the blockchain world, simplifying engagement with blockchain systems [8].

Bitcoin [9], the first example of blockchain technology, invented the concept of a decentralized ledger, thereby fostering the emergence of decentralized transactions that significantly boosted the technology's popularity. Since then, the emerging blockchain technology, a rapidly evolving field, has been seeing a shift toward a decentralized computing paradigm [10]. The applications on blockchains, known as Decentralized Applications (dApps), are a sort of software in which the application execution is not controlled by a single party [3].

Ethereum [11], as a decentralized blockchain, was designed to provide developers with a unified environment for building software within a new and trustworthy framework, focusing on the secure exchange of information between objects [12]. Leveraging this decentralized nature has allowed the creation of smart contracts, which are programs written using a Turing-complete programming language called Solidity [11, 13]. They are stored on the blockchain and enforce rules, checking whether specified conditions are met before any transaction [14]. Since smart contracts are encoded in the blockchain system, they benefit from a tamper-resistant and transparent environment, allowing them to enforce terms based

on specific conditions without third-party involvement [15]. Moreover, Solidity continues to be the leading language for smart contracts and is even supported by various blockchains beyond Ethereum, such as Binance Smart Chain and Avalanche [16]. These technological advancements, coupled with the introduction of a new programming language, have highlighted the significance of blockchain and smart contract development. This, in turn, will lead to the increased incorporation of smart contract concepts within educational domains, as critical and popular technologies are commonly integrated into educational curricula, providing students with the opportunity to build a strong foundation in these key areas [5].

Ethereum dApps can deploy smart contracts to utilize the computational and storage capabilities of the Ethereum blockchain for executing business logic [3]. These contracts can be deployed and tested using Ethereum smart contract development tools, hereafter referred to as **test runner frameworks**, which are continuously refined to be more robust and adaptable to various architectures. Testing smart contracts ensures that they work as intended and are free of vulnerabilities, especially before deployment since they cannot be altered post-deployment [17]. Specifically, even minor errors could lead to serious consequences, from which it is technically not feasible to come back due to blockchain's immutable design. For instance, in financial settings involving numerous non-refundable transactions, any errors could lead to irreversible defective transactions, resulting in significant loss of funds wired by mistake, while also incurring transaction fees regardless [18]. However, there has been an exceptional instance, such as with the hack of Decentralized Autonomous Organization (DAO) [19], an occurrence of a re-entrancy attack[1], where Ethereum's core developers solved the issue by deploying a hard fork, subsequently creating a new version of the existing chain [21]. Therefore, it is imperative to carefully test smart contracts before deploying them to the blockchain.

With the ongoing advancements of the blockchain ecosystem and smart contracts, the emphasis on smart contract testing has become increasingly critical, reflecting the growing need for automated testing methods. Educational programs are responding by incorporating smart contract testing into their curricula, which enables students to develop a solid understanding of the field [5]. This approach aims to educate students through unit-tested exercises based on real-world scenarios, ensuring that students comprehend the importance of implementing robust smart contracts that can withstand multiple critical tests. It is crucial for students to learn about the risks of taking certain actions in smart contract development before deploying them to the blockchain to avoid irreversible scenarios.

[1] A re-entrancy attack occurs when a primary contract makes a call to an external contract, which then calls back into the original contract within the same transaction [20].

1.2. Objectives and Purpose of the Thesis

The primary objective of this thesis is the development and implementation of a service, hereinafter referred to as the **testing service**, designed to test Solidity smart contract inputs provided by users, with a particular emphasis on an educational context. The first stage in the development process is to choose a high-performing test runner framework, which will support the service and ensure that it is scalable and able to handle several requests at once without experiencing system faults. The criteria for selecting this framework will be centered on several key factors: speed, efficiency, ease of deployment (with a particular focus on deployment as Docker [22] containers), and ongoing maintainability and scalability throughout the software development lifecycle. While the use of Docker containers is favored for its enhanced security and convenience, the practicality of implementing Docker within Docker demands further investigation, and alternatives will be explored should this approach prove unfeasible.

Upon establishing a solid framework, a service will be developed, leveraging the chosen high-performing test runner framework. This service will work to provide its users (i.e., BBSE students) with insightful performance metrics and test results that will improve the effectiveness and functional correctness of their implementations of smart contracts. This service must not only offer a wide range of performance data, but must also do it in a way that is simple to use and uncomplicated without losing depth or detail. Moreover, safeguarding the integrity and functionality of the service is crucial. This involves ensuring that the service remains operational, even in the face of errors in submitted smart contracts, and establishing preventive measures, such as submission frequency limits and optimal execution durations for smart contracts. Additionally, considering potential security features and protections against both intentional and unintentional disruptions, such as smart contracts that may result in infinite loops, will be vital to maintain the reliability and availability of the service. In order to design a service that can easily expand to meet the changing needs and demands of smart contract developers in both professional and educational contexts, this thesis will examine the many facets of these different elements. It aims to create a service that is not only secure and efficient but also horizontally scalable.

In summary, the outcome of this thesis will contribute to a comprehensive understanding of the strengths and weaknesses of various test runner frameworks. Ultimately, the work aims to develop a secure, efficient, and easily scalable service that provides a streamlined and effective method for testing smart contracts, thereby enabling users to create reliable and robust smart contracts.

1.3. Research Questions

In this thesis, the subsequent research questions will be thoroughly addressed through an exhaustive combination of academic study, practical investigation, and applied work, all aiming to unveil the complexity of the research topic.

- **RQ01:** What are the requirements for educational unit testing?
  a) What is the core use case?
  b) What are exemplary exercises that we would like students to do?
- **RQ02:** What is the status quo in automated smart contract testing?
  a) Are there examples of smart contract testing as a service?
  b) Which tools are most commonly used for smart contract testing?
  c) How can we characterize those tools in terms of their key features and performance measurement capabilities?
- **RQ03:** What do we have to consider regarding security and stability when using a testing tool in a way that is not entirely intended?
  a) How can errors and crashes in the contract execution be handled?
  b) What measures do we need to take to prevent accidental or intentional system overload?
- **RQ04:** How can a learning platform giving feedback through automated smart contract unit testing be developed?
  a) What considerations need to be made to ensure the service is scalable and expandable?

1.4. Core Use Case

The requirements for automated unit testing in an educational setting include assisting students in evaluating their smart contract implementations through tests provided by the teaching staff. The intended service is required to get smart contract inputs from the students, test those inputs, and then provide a set of performance metrics that show how effective the aforementioned smart contracts are.

The core use case involves students submitting their smart contracts to the provided service, thereby obtaining comprehensive insights into the performance of their smart contracts, particularly in an educational and academic context. Anticipated users of this service encompass both instructors and students, starting with those participating in the BBSE course at TUM. In this scenario, the instructors are expected to create exercises or projects and provide materials for projects that include the tests and smart contracts that pass those tests. Students are then allowed to submit their smart contract inputs; however, to ensure that the testing remains unbiased, they are not given access to the tests or the smart contracts provided by the instructors. After submission, the service runs the instructor-provided tests against the students' smart contract inputs and then delivers the test execution results, providing detailed feedback on various aspects of the submitted contracts.

The exemplary exercises, which are intended to increase student involvement, are centred on the effectiveness and validity of smart contracts. Such exercises encompass a variety of test cases that explore many aspects of the smart contracts and make use of techniques like fuzz testing to evaluate the effectiveness and reliability of a student's smart contract implementation (see Listing A.1 for an example of a Solidity test case). This approach not only strengthens the learning experience with practical applications but also promotes an extensive understanding of smart contract testing among students. In addition, efficiency is further evaluated by applying predetermined gas limits to the student-submitted smart contract inputs. Overall, these exercises should be designed to challenge students' smart contracts with real-world scenarios, ensuring that the feedback from test execution results is comprehensive enough to help them improve their smart contracts.

1.5. Structure of the Thesis

In section 1.4, *Requirements and Core Use Case*, of chapter 1, *Introduction*, a focused discussion addresses Research Question **RQ01**: "What are the requirements for educational unit testing?", along with its related sub-questions. Particularly, it covers **RQ01-a**, which examines the core use case, and **RQ01-b**, which looks at exemplary exercises suitable for students. The structure of the subsequent chapters, aiming to answer the remaining research questions, is outlined as follows:

- In chapter 2, *Background*, the necessary background to grasp the thesis is established. This chapter seeks to partially address Research Question **RQ02** with a discussion regarding the status quo in automated smart contract testing. More precisely, it focuses on **RQ02-b**, identifying the most frequently used tools in smart contract testing.
- In chapter 3, *Related Work*, the current research in the field of automated smart contract testing is examined, further contributing to Research Question **RQ02**. This chapter also explores **RQ02-a**, which inquires about examples of smart contract testing as a service.
- In chapter 4, *Methodology*, a comprehensive discussion of the methodology is presented, encompassing the selection criteria for the test runner frameworks and a detailed explanation of how the testing service is built.
- In chapter 5, *Comparative Analysis of Test Runner Frameworks*, a comparative analysis of the test runner frameworks is conducted to identify the best-performing framework. This analysis seeks to address Research Question **RQ02-c**, which inquires: "How can the tools be distinguished by their key features and performance assessment capabilities?"
- In chapter 6, *System Design and Implementation*, the emphasis is placed on discussing the system's design and development, concurrently addressing the educational unit testing requirements. This chapter aims to provide in-depth insights into **RQ04**: "How can a learning platform that provides feedback through automated smart contract unit testing be developed?", and additionally responds to **RQ04-a** in section 6.6, assuring that the service is both scalable and expandable. Moreover, section 6.5 in this chapter provides insights into **RQ03**, addressing what considerations are essential for security and stability when utilizing a testing tool beyond its intended use. Within this context, **RQ03-a** and **RQ03-b** are explored in greater detail, looking at how mistakes and crashes in contract execution can be managed and determining the necessary precautions to prevent either intentional or unintentional system overload.
- In chapter 7, *Results and Evaluation*, the evaluation focuses on the security, stability, and performance aspects of the developed service to determine if it satisfies the non-functional requirements.
- In chapter 8, *Conclusion and Future Work*, the thesis is concluded by summarizing the major findings and making recommendations for further research.

2. Theoretical Background

This chapter provides the background needed to follow the discussions in the subsequent sections of the thesis, offering a concise overview of the main topics covered in this work.

2.1. Blockchain Technology

Initially, blockchains gained attention for transforming record-keeping through a structure that chains together blocks of transactions, with each entry becoming permanent and immutable once added to the chain [1, 2]. In Ethereum, "*transaction*" refers to the signed data package containing a message sent from an externally owned account, which is an account managed by private keys without any associated code, and messages can be dispatched from such accounts by creating and signing a transaction [11]. Furthermore, the immutable nature of blockchain, combined with its decentralized and distributed ledger, has not only provided a secure and transparent method for processing data but also attracted significant attention from various sectors due to its potential to ensure data integrity [3].

Bitcoin is recognized as the initial application of blockchain, having developed and popularized the decentralized ledger concept, considerably increasing the technology's popularity and leading the way for decentralized transactions [9]. It also introduced the first decentralized digital currency, or cryptocurrency, distributing bitcoins via an open-source release of its peer-to-peer software [2]. The blockchain field has evolved with the introduction of Ethereum, as previously mentioned, moving towards a decentralized computing paradigm exemplified by applications known as dApps, which function without centralized control [3].

Dannen (2017) divides blockchain into three parts, stating that *"A blockchain can be thought of as a database that is distributed, or duplicated, across many computers. The innovation represented by the word blockchain is the specific ability of this network database to reconcile the order of transactions, even when a few nodes on the network receive transactions in various order. ... What is widely called a blockchain is really the combination of three technologies, a recipe first concocted by Bitcoin's pseudonymous creator."* [23], which consists of:

- **Peer-to-peer networking:** A network topology exemplified by BitTorrent [24] in which computers connect directly with one another, without the use of a central authority, hence eliminating a single point of failure [23].

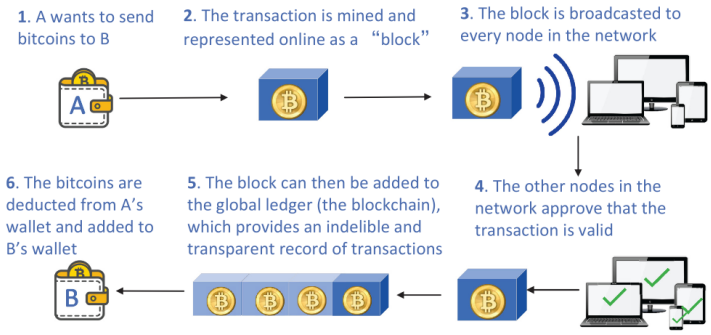

Figure 2.1.: The Functioning of Blockchain Technology (*Source:* [2])

- **Asymmetric cryptography:** A secure communication method that allows computers to send encrypted messages to specific recipients, ensuring the sender's identity is publicly verifiable while maintaining the privacy of the message content. In Bitcoin and Ethereum, this cryptography is employed to create unique credentials for user accounts, which authorize token transactions [23].

- **Cryptographic hashing:** A method for creating a distinct and concise "*fingerprint*" for each set of data, allowing for the quick comparison of enormous data sets while maintaining data integrity. Bitcoin and Ethereum use the Merkle tree structure to store transaction sequences, creating a "*fingerprint*" that network computers can use to efficiently accomplish synchronization [23].

This is supplemented by the work of Zhang et al. (2019), who characterize blockchain technology as a significant advancement in secure computing within a decentralized, open network system. The authors view it as a data management solution, wherein blockchain serves as a distributed ledger that systematically organizes transactions into hierarchical blocks. Further addressing the technology's security, they explain that blockchain is built and maintained through a peer-to-peer network that utilizes decentralized cryptography and the collective computational power of numerous nodes to safeguard its integrity [2].

The operation of blockchain is illustrated in Figure 2.1, demonstrating that a bitcoin transfer from individual A to B is only concluded and considered legitimate once the transaction's block wins approval from other nodes and is formally appended to the blockchain [2]. This consensus is reached through mining, a process where nodes, also known as miners, validate transactions and collectively agree on their order within the network [23]. Mining is incentivized by a reward system, where miners are compensated with cryptocurrency, such as ether (the main internal cryptocurrency of Ethereum) in the Ethereum network, for the computing power, time, and energy spent during the verification process [23]. One of the most widely-deployed consensus mechanisms is called Proof of Work (PoW), which was introduced by Bitcoin and relies on miners using their computational resources to solve complex problems, thus earning the privilege to append new blocks to the blockchain [25].

Regarding the security of PoW, Gervais et al. (2016) further highlight that PoW's security is predicated on the assumption that no single entity can possess over half of the network's computational power (i.e., 50%), as this would grant them the ability to control the system by maintaining the longest chain of blocks [25]. This is of utmost importance for the integrity of the blockchain and the prevention of centralized control. Although theoretically possible, in practice, such control is deemed challenging to achieve given the expected difficulty of acquiring such an extensive amount of computing power within a sufficiently large and active network [26]. Nevertheless, a real-world incident, known as a 51% attack, occurred in May 2018 when an attacker successfully double-spent approximately $18 million worth of Bitcoin Gold [27], a cryptocurrency based on Bitcoin's principles [28].

2.2. Smart Contracts

Ethereum was created to enhance the capabilities of blockchain, introducing a platform where developers can create smart contracts (self-executing contracts with rules written directly into code) using Solidity, a Turing-complete language designed specifically for this purpose [11, 13]. Stored on the blockchain, these smart contracts execute certain actions when the specified criteria are met, leveraging the blockchain's immutability and transparency to facilitate trustful and secure transactions without intermediaries [14, 15]. The widespread adoption of Solidity as the language of choice for smart contract development confirms its strength and adaptability, now extending its use to other blockchain platforms as well [16].

Vitalik Buterin, the architect of Ethereum, discusses the crucial difference between Ethereum and Bitcoin, attributing it to Ethereum's ability to execute smart contracts via Solidity. Buterin explains, *"As a data structure, it works kind of the same way that Bitcoin works, except the difference in Ethereum is, it has this built-in programming language."* [23].

In the Ethereum White Paper, Buterin (2014) describes Bitcoin's scripting as an early form of the concept of smart contracts, upon which Ethereum's framework builds and improves. Additionally, Buterin addresses Ethereum's solution to the lack of Turing-completeness of Bitcoin's scripting, which omits loop constructs to prevent infinite loops during transaction verification. The author mentions that while script programmers could theoretically replicate loops using multiple if statements, this results in highly space-inefficient scripts. Further, Buterin articulates Ethereum's vision to combine and refine the concepts of Bitcoin's scripting, thereby enabling the creation of various consensus-driven applications that offer scalability, standardization, comprehensive features, ease of development, and interconnectivity. Finally, Buterin states that this is achieved by establishing an abstract foundational layer: a blockchain with a Turing-complete programming language that allows anyone to create smart contracts and dApps with specific rules for ownership, transaction formats, and state transitions. The author concludes by how the Ethereum platform enables the creation of smart contracts, which are cryptographic containers that contain value and release it only when certain conditions are met [11].

Ethereum Virtual Machine (EVM). Smart contracts on Ethereum are executed by the EVM, a single, global 256-bit "*computer*"[1]. The EVM operates using its own language, known as *the EVM bytecode*, which high-level languages like Solidity are compiled into and then deployed on the blockchain for execution. Each action on the EVM is assigned a specific *gas* cost [23].

Gas as a Unit of Work. Dannen (2017) describes *gas* in Ethereum as a unit of work, which quantifies the computational effort required for operations and transactions[2]. The author mentions that the gas system serves two primary purposes: First, it provides miners with a prepaid incentive to execute code and maintain network security, even if the execution fails. Second, it prevents operations from exceeding their assigned computation time, addressing the *halting problem*[3]. Dannen further highlights the significance of gas pricing in ensuring that computational time on the network is correctly valued, with costs paid in small amounts of ether. Furthermore, the author contrasts Ethereum with Bitcoin, where fees are calculated based on transaction size in kilobytes, whereas in Ethereum's EVM, fees depend on the computational work involved, not the size of the transaction. This pricing model accounts for the varying complexity of Solidity code, where length does not always correspond to either complexity or required computational effort [23].

2.2.1. Solidity Programming Language

Dannen (2017) simply defines smart contracts as *"Smart contracts are often equated to software applications, but this a reductive analogy; they're more like the concept of classes in conventional object-oriented programming. When developers speak of writing smart contracts, they are typically referring to the practice of writing code in the Solidity language to be executed on the Ethereum network. When the code is executed, units of value may be transferred as easily as data."* [23].

Drawing from the research of Wöhrer and Zdun (2018), Solidity emerges as a high-level, Turing-complete programming language, bearing a resemblance to JavaScript in its syntax. It is characterized by static typing and supports programming features such as inheritance and polymorphism. Furthermore, it allows for the use of libraries and the creation of complex user-defined types. In the context of contract development, Solidity treats contracts similarly to classes in object-oriented languages, with contract code consisting of variables and functions that interact with these variables, similar to traditional imperative programming [31].

Expanding further, Buterin (2014) outlines the fundamental structure of the Ethereum state, which comprises unique "accounts," each identified by a 20-byte address. These accounts transition between states by exchanging value and information, facilitated by the EVM as it processes the instructions in contract code [11].

Solidity contracts, therefore, have access to transaction details such as value, sender, and data, as well as metadata contained in block headers [23]. Access to these details enables

[1]The EVM is not a physical computer but a virtual computation engine for Ethereum, maintaining the blockchain's state and facilitating the operation of smart contracts [29].

[2]The total fee incurred by a transaction is calculated by multiplying the total amount of gas used by the price paid for the gas [23].

[3]The halting problem is a concept in computer science that refers to the difficulty of determining whether a program will eventually stop or continue to execute indefinitely in a loop [30].

the development and implementation of a wide range of rules on the blockchain, taking advantage of Solidity's powerful capabilities to enforce logic inside the Ethereum ecosystem.

As an example, the code listing provided in Listing 2.1 below, sourced from Wöhrer and Zdun (2018), presents a basic Solidity smart contract designed for handling deposits, withdrawals, and balance inquiries. An initial examination reveals a syntax that closely resembles that of JavaScript [31].

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

Listing 2.1: A contract to deposit/withdraw funds and verify account balances (*Source:* [31])

To examine the syntax, a contract first declares the version of the Solidity compiler to be used (*Line 1*), and then utilizes the `contract` keyword, similar to a class in object-oriented programming, to define the structure of the contract (*Line 3*).

Contracts usually utilize the Solidity language's `mapping` structure (*Line 5*) to link Ethereum addresses with their respective balance values, reflecting a key-value pair akin to dictionaries in many other programming languages.

Functions within the contract, denoted by the `function` keyword (*Lines 14, 19, and 23*), contain the logic to interact with the contract's state [31]. They are declared with modifiers that define their behavior and access levels. For instance, `public` sets the accessibility of the

function, allowing external calls to it, `view` indicates that the function will not modify the state, and `payable` enables the function to accept `ether`, which is necessary for functions that access the value of the message (*Line 14*). Additionally, the `returns` keyword, followed by a type, is used to specify the return type of a function, as illustrated in the `getBalance` function (*Line 19*).

A convenient feature in Solidity is the custom modifiers. These code constructs alter the execution flow of functions by encapsulating conditional checks and are invoked through a list appended to the function's name, with the original function body inserted where the modifier indicates with an underscore, " _ " (*Line 9*) [31]. For instance, incorporating the `minAmount` modifier to the `deposit` function mandates that callers must have a balance of at least the specified `ether` to invoke the function, which is `1 ether` in Listing 2.1 (*Line 14*).

Solidity also provides special variables, such as `msg.sender` and `msg.value`, which contain information about the caller and the value transferred, respectively, and are used within functions to implement certain rules and logic.

Events in Solidity, such as `LogDepositMade` (*Line 7*), are a valuable feature that allow smart contracts to emit signals, which can be monitored by user interfaces and applications at minimal cost for responsive actions, and also serve as logs by storing their arguments in the transaction's log within the blockchain, accessible externally due to their association with the contract's address [31]. For example, the event named `LogDepositMade` is declared in *Line 7* and subsequently triggered in *Line 16*.

Finally, beyond the constructs depicted in Listing 2.1, error handling is implemented either through the `require` keyword, which enforces that specified conditions are satisfied, or the `revert` keyword, which throws an error and aborts the transaction.

2.2.2. Smart Contract Testing

Ethereum dApps make use of smart contracts to harness the Ethereum blockchain's processing and storage capabilities for carrying out their core functions [3]. These smart contracts can be deployed and tested using what are commonly known as test runner frameworks, details of which are provided in the following section. It is vital to conduct rigorous testing of smart contracts to ensure their functionality and security since, after deployment, they become immutable; this immutability makes any errors permanent and could lead to significant consequences [17]. This highlights the necessity of testing before blockchain deployment.

Integration Testing vs. Unit Testing. As highlighted in a recent study [32], smart contracts frequently rely on the functionality of other interconnected contracts, with the complexity and unpredictability of behaviors increasing as the network expands. The study also notes that the blockchain is considered a self-contained system where smart contracts are limited to interacting with data already on the chain. The authors further point out that since all contracts on the blockchain interact through a shared "*World State*" [32], their testing aligns more closely with *integration testing*, which examines the overall functionality, as opposed to *unit testing*, which isolates and analyzes individual components [32].

Further Importance of Testing and Challenges. A study by Zou et al. (2019) identifies the deployment of secure and error-free code as the primary concern among industry professionals involved in blockchain development [33]. Barboni et al. (2022) state that ensuring the reliability of code for smart contracts is of paramount importance. The authors outline several key attributes that underscore the vital nature of smart contract testing [32]:

- **High stakes:** Smart contracts carry significant risks due to their management of valuable assets, such as financial resources and sensitive data, and untested contracts can result in unpredictable behavior on the blockchain, leading to financial losses or malicious exploitation [32].
- **Immutability:** Due to the nature of the blockchain as a decentralized and immutable record system, any unwanted transactions performed during contract execution are irreversible, leaving users with no option to obtain a refund unless the original recipient issues a new transaction to return the assets [32].
- **Lack of standardized best practices for smart contract design and development:** Barboni et al. (2022) note the difficulty in developing high-quality smart contract code, attributing it to a lack of established best practices, standards, and guidance in the development process. They also observe that while smart contract development has advanced in the recent years, there remains an absence of thorough guidance for developers on how to create reliable smart contract code, especially when contrasted with traditional software [32]. Additionally, Pierro et al. (2020) examine how the similarity in syntax between traditional languages such as JavaScript and smart contract-specific languages like Solidity can lead blockchain software developers to a false sense of security [34].

2.2.3. Test Runner Frameworks

A study by Palechor et al. (2022) observes that there are a few open-source platforms available, such as Truffle [35] and Hardhat [36], that aid in the development and testing of smart contracts for developing high-quality smart contract [37]. As previously introduced in section 1.1, these platforms are known as the test runner frameworks.

Palechor et al. (2022) also highlight the deficiency in testing support for smart contracts relative to the established frameworks available for traditional applications. Nonetheless, their detailed examination of several test runner frameworks identifies the most often used ones, offering light on the practical testing methodologies used by developers. The study reveals that Truffle is the framework of choice for half of the listed projects, making it the most popular. Further discussion in their work covers Hardhat and DappTools [38], which rank as the next frequently used tools for smart contract testing. The study also details less commonly used tools in the domain, such as Manticore, Embark, and Solgraph [37].

A recent blog post [39] offers a comparison of the top three test runner frameworks, which are Hardhat, Truffle, and Foundry [40]. The blog post highlights Foundry as an emerging star with lightning-fast testing speeds. Foundry is a framework that is a thorough rewrite of

the DappTools testing framework, an innovation made possible by the foundational work of the DappHub team who developed DappTools [41].

In summary, test runner frameworks facilitate the development and testing of smart contracts, with the latter being the focal point in this thesis. Truffle, Hardhat, and Foundry have been identified as the most commonly used frameworks for smart contract testing. Further in this work, these three frameworks will be compared to determine the most suitable one to be used in the final testing service.

2.3. Containerization

Before discussing the concept of containerization, one must mention virtualization, the precursor to containerization. In a past study by Chiueh et al. (2005), the concept of Virtual Machines (VMs) was examined, tracing back to the 1960s. The authors highlighted that it was first developed by IBM to enable multiple users to interact with a mainframe computer simultaneously. They also noted how a VM acted as a replica of the physical computer and gave users the impression that they were using the actual hardware. This method was a smart, seamless, and transparent way to facilitate time-sharing and resource-sharing on costly equipment. Moreover, the authors described each VM as a secure and independent version of the main system, where users could execute, develop, and test software without the risk of affecting the systems of other mainframe users. Therefore, virtualization was a technique introduced to reduce hardware costs and increase productivity by allowing a larger number of users to share resources [42]. The authors continue their exploration by examining the practical applications of virtualization, outlining the following essential scenarios [42] that may aid in understanding the concept of containerization within this thesis:

- **Sandboxing:** The authors describe the use of VMs to create secure, isolated environments, or sandboxes, for the execution of potentially unsafe or unverified applications, hence contributing to the development of secure computing environments.
- **Multiple Execution Environments:** The paper elaborates on how virtualization allows for the establishment of multiple execution environments, which can improve service quality by ensuring dedicated resources.
- **Multiple Simultaneous Operating Systems:** The paper points out that virtualization technology enables the use of several Operating Systems (OSs) at once, accommodating a variety of applications to run concurrently.
- **Debugging:** The authors suggest that virtualization can be a valuable tool for debugging complicated software by letting the user execute them on a simulated system with full control over the software.
- **Software Migration:** In the paper, virtualization is highlighted as a facilitator for the migration of software, which assists in the mobility of software applications.

Table 2.1.: Comparison between VMs and Containers

Parameter	VMs	Containers
Guest OS	Each VM runs on virtual hardware, and the kernel is loaded into its own memory region.	All the guests share the same OS and kernel. The kernel image is loaded into the physical memory.
Communication	Communication is through Ethernet devices.	Mechanisms like signals, pipes, sockets, etc., are used.
Security	It depends on the implementation of the hypervisor.	Mandatory access control can be leveraged.
Performance	VMs suffer from a small overhead as the machine instructions are translated from guest to host OS.	Containers provide near-native performance compared to the underlying host OS.
Isolation	Sharing libraries, files, and other resources between guests and between guests and hosts is not possible.	Subdirectories can be transparently mounted and shared.
Startup Time	VMs take a few minutes to boot up.	Containers can be booted up in a few seconds, in contrast to VMs.
Storage	VMs require much more storage as the entire OS kernel and its associated programs have to be installed and run.	Containers require a lower amount of storage as the base OS is shared.

Source: [45]

- **Appliances:** The authors mention how an application with its required operating environment can be packaged into a single deployable unit.
- **Testing/QA:** The paper underscores the role of virtualization in generating arbitrary test cases that might be challenging to produce in reality, thereby aiding in the software testing process.

Containerization, on the other hand, is a virtualization solution that delivers all the scenarios mentioned above with a lightweight portable runtime. It provides the ability to develop, test, and deploy applications from a single host to cluster of containers[4] [44]. This approach allows a swift and straightforward process of porting applications to different machines and environments, thereby increasing development speed and application scalability [22]. Furthermore, containerization ensures application isolation from the host and encapsulates only the necessary components required for the application to function.

In their study, Dua et al. (2014) investigate how containerization differs from traditional virtualization, evaluating factors such as performance, isolation, security, networking, and storage. They explain that a container within a containerized environment operates as a lightweight OS within the host system, executing instructions directly on the core CPU, thereby eliminating the necessity for instruction-level emulation or just-in-time compilation.

[4]A container bundles an application's code together with the required libraries and dependencies [43].

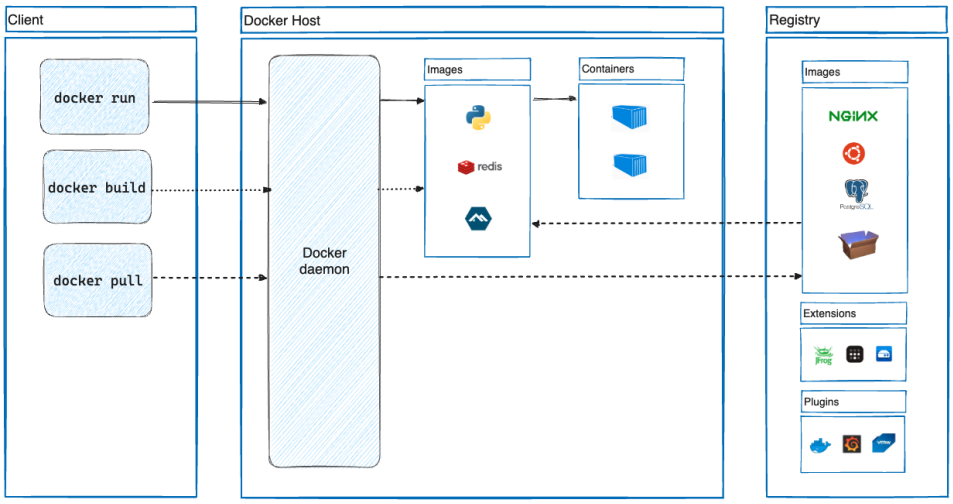

Figure 2.2.: The Client-server Architecture of Docker (*Source:* [46])

In Table 2.1, sourced from their paper, it is evident that containers can match VMs in providing a secure and isolated environment but with significantly faster boot times, often just a few seconds. They facilitate a reduction in resource usage by avoiding the typical overhead associated with virtualization while still offering isolation. In comparison to VMs, containers also use less storage because they share the underlying OS. Furthermore, they allow subdirectories to be shared, a feature not available with VMs [45].

2.3.1. Docker

There are several tools available that effectively facilitate containerization. Among these tools, Docker, a well-established containerization solution, has been a popular choice among developers in the DevOps ecosystem [47]. This open-source technology allows developers to build, ship, and run distributed applications [22]. Its adoption by companies such as Spotify, Yelp, and eBay attests to its widespread popularity [48]. Moreover, Docker can deploy more virtual environments on the same hardware compared to other tools [49]. It offers extensive community support with a wide range of materials and guides online, making it the chosen tool for containerization in this work. Docker's comprehensive library of pre-built images available on Docker Hub [50] simplifies the process of pulling and working with existing Docker images. For instance, in this project, most of the images will be based on the latest Ubuntu image from the Docker registries.

In a study focusing on Docker's security, Bui (2015) highlights Docker's unique approach among various frameworks, which includes offering straightforward interfaces for the secure creation and management of containers. Additionally, Docker encapsulates applications within lightweight containers that can operate across diverse environments with little to no changes needed [48].

Docker Engine. Bui (2015) describes the Docker Engine as a portable tool for packaging that uses container-based virtualization[5]. The author explains that the *Docker daemon* is responsible for the operation and management of Docker containers. In addition, the Docker client presents an interface for users to interact with the containers, processes user commands, and forwards them to the Docker daemon via the REST API. Bui (2015) also emphasizes that this communication approach allows the Docker client to operate either on the same host as the containers or on separate hosts [48]. Furthermore, communication between the Docker client and the Docker daemon can occur over UNIX sockets or a network interface [46]. The client-server architecture of Docker, which illustrates how the client interacts with the daemon, is depicted in Figure 2.2.

Docker Image & Container. A Docker image is constructed from a base layer, such as an Ubuntu base image, and builds up additional layers with user changes, such as installing MySQL [48]. The build process is optimized through caching, which rebuilds only the changed layers. This process is directed by a special file, a *Dockerfile*, which contains sequential instructions to build the Docker image (see Figure C.1 for a sample *Dockerfile*). Moreover, when Docker images are executed, they create Docker containers, which are stand-alone environments generated by running a specific command on the images.

Docker Hub. Docker Hub is a Software as a Service (SaaS) platform and a centralized repository where users can distribute their custom images, search for and download existing ones using the Docker client, while relying on the authenticity and integrity of the images, which is assured by Docker's verification process at the time of submission [48].

Docker Compose. The official documentation of Docker Compose [51] defines it as an essential tool for defining and running multi-container Docker applications. The tool uses a YAML file for service configuration and a single command to start all services. It is adaptable and offers extensive management capabilities for the application lifecycle. Its usefulness originates from its ability to run multiple isolated environments on the same host, preserve volume data upon container creation, and selectively recreate only the changed containers, among other capabilities [51].

[5]Container-based virtualization is a lightweight virtualization approach that utilizes the host's kernel to operate multiple virtual environments, commonly known as containers [48].

3. Related Work

Before initiating the development process, it is essential to clearly define the requirements and the problem at hand. This section provides a brief review of related work and examines any previous endeavors similar to the automated smart contract testing service developed in this thesis, which is primarily focused on educational applications. Ultimately, a combination of scientific and pragmatic research will be used to explore several strategies for implementing such a modular service for smart contract testing.

Examples of Student Submission Systems in Educational Contexts. The core application of the final testing service in this thesis is centered on an educational context. Hence, investigating existing student submission methods is critical for the investigation of related works. *Artemis* [52] at TUM, an open-source interactive learning platform that delivers individualized feedback to students, is a prominent example of such a system. Artemis enhances the learning experience by providing programming exercises in which students use version control systems and obtain automated evaluations through test cases connected with continuous integration tools. The platform allows students to submit their solutions numerous times before the deadline, enabling them to refine their work using the feedback provided [52]. This iterative submission and feedback process is aligned with the key functionalities anticipated for the final testing service. Furthermore, the Parallel Programming course [53] at TUM features a dedicated submission system through which students can upload their exercise solutions. This system utilizes tools to manage scalability and employs message queuing[1] to effectively handle the concurrent load from multiple student submissions. Such technologies could also be integrated into the final testing service to enhance its scalability, stability, and overall performance.

Status Quo in Smart Contract Testing. This paragraph focuses on automated smart contract testing and its usefulness in understanding the current state of the field. Driessen et al. (2021) investigate automated unit testing on the Ethereum blockchain for Solidity smart contracts. They observe that current testing methods are mainly focused on detecting vulnerabilities within smart contracts and the blockchain through basic strategies such as fuzzing. While recognizing the importance of vulnerability detection, the authors argue that having access to a good test suite may be even more beneficial. They introduce an automated system for generating test cases for smart contracts, which creates scenarios involving transactions from different accounts to test sender-dependent functionality [55]. Adding to the conversation on vulnerability detection, Mi et al. (2023) emphasize its significance as a critical research

[1] Message queuing facilitates inter-application communication by transferring data through queues that are processed in sequence [54].

problem and propose an automated framework for detecting vulnerabilities that employs a metric learning-based neural network [56]. Moreover, Benabbou et al. (2021) also stress the importance of testing smart contracts, emphasizing that bug-free smart contracts improve both the reliability and cost-effectiveness of these contracts. They categorize current testing solutions into categories, examining and contrasting their advantages and disadvantages [57]. Barboni et al. (2022) point out that the current state of smart contract testing is lacking a consistent set of guidelines, best practices, and platform-independent tools for testing. These authors also mention the absence of applied and cost-effective techniques to assess the adequacy of tests [32]. Finally, according to Palechor et al. (2022), functional testing is the most commonly used methodology for smart contract testing, followed by security testing. They also point out the absence of traditional performance assessment in the field, such as execution time measurements. Instead, they discovered that a large majority of projects measure performance through gas usage [37].

Existing Analyses on Test Runner Frameworks. In this thesis, the initial step is comparing widely-used test runner frameworks, specifically Truffle, Hardhat, and Foundry, to determine the most suitable one for the final testing service. A review of prior comparative analyses on these frameworks is crucial. Palechor et al. (2022) conduct a comparative analysis of various test runner frameworks by examining numerous open-source smart contract projects coded in Solidity. They investigate the state of smart contract testing from the perspective of developers and the test runner frameworks used by them [37]. Furthermore, several blog posts provide in-depth discussions on various frameworks. For instance, a blog post by Celo Academy [58] provides a detailed comparison of Truffle and Hardhat, highlighting the significance of selecting the appropriate framework based on specific project requirements and outlining their unique strengths and weaknesses. It also presents use cases and considerations for selecting either framework based on their distinct features. Additionally, another blog post [59] explores the differences between Hardhat and Foundry, focusing on aspects such as developer experience and performance. It describes unique features including Foundry's *cheatcodes*, which provide developers with expanded testing options such as changing block numbers and accounts. The post also compares the performance of both frameworks and finishes with a discussion of their advantages and disadvantages. In conclusion, the approaches used in the aforementioned studies will be useful for this work in comparing various test runner frameworks since understanding the details of these comparisons is crucial for identifying the best effective evaluation strategy.

Examples of Automated Smart Contract Testing as a Service. The testing service developed in this thesis is presented as a software package that provides automated testing of smart contracts. To our knowledge, there is not a service that tests smart contracts using provided test cases, but there are similar works. *DefectChecker* [60], proposed by Chen et al. (2021), examines bytecodes to find defects in smart contracts. Although it employs bytecode analysis rather than test runner frameworks, *DefectChecker* is still considered related work due to its automated testing capabilities. Li et al. (2019) introduced *MuSC* [61], a mutation testing tool for evaluating test adequacy in Ethereum smart contracts, which is relevant in this

discussion for supporting smart conract testing on user-defined testnets[2]. Moreover, industry examples of automated smart contract testing services bear a closer resemblance to the service developed in this thesis. *ConsenSys Diligence* [63], for instance, provides enterprise-level smart contract testing, utilizing its robust fuzzer with millions of transactions for stress testing[3] to find bugs and vulnerabilities before they are exploited. It also comes with several open-source tools to perform vulnerability detection and security analysis for EVM bytecode, delivering comprehensive reports [63]. Similarly, *Suffescom Solutions* [65] delivers several smart contract testing services, offering automated tools to assure the dependability, security, and performance of blockchain systems. They provide end-to-end testing for smart contracts that involve many transactions in the blockchain ecosystem. They use testing methods such as functionality, security, and automation testing, with the latter utilizing popular tools such as Truffle. These enable businesses to create error-free and secure blockchain applications or dApps. Furthermore, these services perform smart contract penetration testing to identify potential security weaknesses by mimicking real-world attack scenarios [65]. Lastly, open-source tools like *Etheno* [66] provide differential tests that check for discrepancies between implementations, such as gas consumption differences, which could be a valuable performance metric later in this thesis.

Example Docker Containerization Use Cases. In this work, Docker is employed to containerize the final testing service and its associated microservices. This section examines Docker's application in various domains. Chung et al. (2016) utilized Docker containers for deploying High Performance Computing (HPC) applications to evaluate performance between VMs and containers [67]. Additionally, Bonacorsi et al. (2016) describe the popularity of Docker as the leading standard for container-based virtualization, highlighting its adoption by major IT providers such as Spotify for continuous delivery, service testing, and deployment. They also encapsulated several Content Management System (CMS) applications in Docker containers and found that Docker offers several options for deploying and benchmarking large-scale projects without compromising performance [68]. A paper by Naik et al. (2016) provides a simulated creation of a virtual system of systems, which enables distributed software development across several cloud platforms using Docker Swarm, which broadens the scope of Docker's container-based software development to span across multiple hosts and clouds [69]. Furthermore, Jansen et al. (2016) illustrate the use of Docker Swarm [70] for distributed computing across multiple nodes, handling the substantial data volumes common in biomedical analysis, like imaging and biosignal processing. Their work emphasises Docker Swarm's efficiency in data-intensive jobs by giving a case study that employs machine learning to analyze biosignal features, all while avoiding data transfer bottlenecks [71].

[2]Testnets allow developers to deploy and interact with smart contracts without incurring the gas fee [62].
[3]Stress testing is a software testing technique that pushes the software to its boundaries [64].

4. Methodology

This chapter describes the approach that will be used throughout this thesis, providing an in-depth description of the process of selecting the optimal test runner framework and building the final testing service.

4.1. Test Runner Framework Evaluation and Selection

Before implementing the end service, the first step will be to compare various test runner frameworks to explore their convenience for automated smart contract testing. The most widespread ones, chosen as candidates in this work, include Truffle, Hardhat, and Foundry. Through this analysis, the strengths and limitations of each test runner framework will be assessed to provide valuable insights into their effectiveness and suitability for the core use case. In-depth evaluations of their documentation, community support, and extensibility will also be conducted to determine not just their current capabilities but also their potential for future adaptations and troubleshooting.

The final evaluation criteria for the frameworks will encompass speed, efficiency, ease of deployment as Docker containers, and manageability throughout the entire software development process, aiming to ensure that the developed service remains easily maintainable and scalable. Furthermore, the candidate test runner frameworks will be set up and tested locally by writing a common test case in compatible formats that each of them understands or different test cases for different frameworks. This stage will incorporate a meticulous analysis of the error messages and logs generated during the test runs, enabling a deeper understanding of each framework's debugging and problem-resolution capabilities.

Only after all of the frameworks have been configured to run in the local environment will methods for deploying them as Docker containers be investigated, as Docker is the chosen containerization solution. To assess the practicality and performance of each framework when containerized, an in-depth analysis will be conducted, taking into account factors such as container size, startup time, and resource utilization. Once everything is set, each framework will be benchmarked locally to determine which one is the most promising, ensuring that the chosen one not only excels in performance but also in robustness, documentation quality, and community support, thereby confirming its position as a sustainable and reliable preference for the upcoming service implementation.

4.2. Design and Appraisal of the Final Testing Service

After selecting the most suitable test runner framework for smart contract testing based on usability, development experience, features, performance, and containerization capabilities, a final testing service will be designed and developed. This service will utilize Docker containers to manage a workflow that processes user inputs to build and execute Docker images, and subsequently removes the containers upon completion of their use. Furthermore, the service will encompass multiple Dockerized microservices or applications, implemented in TypeScript with Node.js. TypeScript, preferred over JavaScript, is chosen for its strict typing system, which is believed to enhance code maintainability [72]. The design will emphasize stability, security, performance, and scalability to ensure that the service's requirements are met. Additionally, the service should also provide high maintainability, facilitating the easy integration of new features and updates.

- The core worker service, hereafter referred to as the **test runner service**, will manage the creation of Docker images that will contain the submitted project. Executing these images with smart contract inputs will allow testing those smart contracts against the tests uploaded with the projects and return the results. It is worth noting that this worker service will use the host machine's Docker daemon to avoid executing Docker within Docker, which would have resulted in performance concerns.
- The backend services will function as a user gateway. Utilizing REST APIs, it will acquire input from users and provide it to the test runner service, either for the construction of Docker images for projects or the execution of Docker containers for testing smart contracts. This service will be responsible for managing user authentication and authorization, securely storing users in the database via the use of JSON Web Token (JWT) tokens. Furthermore, the database will include information about the projects and smart contract submissions, as well as the uploaded files for each of them, making them available for download on demand.
- To facilitate user engagement with the system, a frontend application will be developed. This application, which only has access to the backend service, will validate that the functionalities of the thesis work as expected.

4.2.1. Simultaneous Submission Management

The service should accommodate a load of approximately *200-300* developers, the majority of whom are expected to submit their contracts during peak times. In the core use case, for instance, students are required to submit their contracts before a pre-defined deadline. It is anticipated that most submissions will occur during the final hours leading up to the deadline. Consequently, the service should be fortified to manage this demand. In order to facilitate this, the incorporation of a message queueing mechanism, such as RabbitMQ [73], is anticipated to systematically regulate the flow of requests. Depending on server capacity, this queue will modulate the load, executing contracts in small batches.

Furthermore, instead of backend services sending requests to REST APIs of the test runner service, an architecture will be designed in which both services are oblivious of each other's existence, interacting instead via RabbitMQ. Backend services will send messages to the queue, which will subsequently be consumed by the test runner service.

4.2.2. Service Security and Stability

Ensuring the integrity of the service against all submitted code is essential. The service must identify and handle errors in submitted smart contracts, including those that cannot be compiled. Additionally, utilizing gas limits, establishing submission frequency limits, and setting runtime durations for smart contracts could facilitate control over code submissions. This is critical, especially when dealing with code submissions that either become trapped in an infinite loop or take an inordinate amount of time to complete.

4.2.3. Scalability and Service Distribution

The service must also be designed with horizontal scalability in mind to handle potential future increases in user volume. This versatility is expected to be aided by the incorporation of Docker Swarm to automatically distribute containers across the cluster, enabling the service to scale up or down as necessary. To realize this, the entirety of the service must be constructed with consideration for its potential to operate independently in the future.

4.2.4. Efficiency and Performance Analysis

The ultimate evaluation of the proposed solution necessitates an assessment of its efficiency in addressing the aforementioned problems. The message queue should enable the service to manage a substantial workload without failures, deliver accurate test results, and maintain high availability. Conclusively, the development of a straightforward frontend application that employs this service will serve as an exemplary means to demonstrate and test that the service is functioning as anticipated. The findings and results will be used in the thesis to show that the built service is useful, powerful, and reliable for users and instructors.

5. Comparative Analysis of Test Runner Frameworks

This chapter presents a comprehensive comparative analysis of several test runner frameworks, namely Truffle, Hardhat, and Foundry. It assesses their usability, development experience, feature sets, reporting mechanisms for resulting metrics, performance, and containerization capabilities. Based on this analysis, the chapter concludes with the selection of the most suitable framework, which is to be used for smart contract testing in the final testing service.

5.1. Overview of Test Runner Frameworks

In this section, a brief introduction to each of the test runner frameworks will be presented, explaining their respective purposes and significance in the context of smart contract development and testing.

5.1.1. Truffle

Truffle is one of the most widely used test runner frameworks among Ethereum developers, with over 1.5 million lifetime downloads [74]. Developers commonly employ the Truffle framework to test Ethereum smart contracts, using JavaScript to write tests that can effectively interact with the contracts on test networks or the main Ethereum network [75]. The framework provides certain abstractions, enabling easy interaction with smart contracts as JavaScript classes or objects. For instance, in the code snippet `let balance = await instance.getBalance()`, the smart contract's balance can be retrieved effortlessly. Additionally, Truffle uses Node Package Manager (npm) for managing and installing dependencies.

Furthermore, Truffle incorporates an integrated debugger similar to most command line debuggers. This debugger enables smooth debugging of transactions made against the smart contracts, adding to the convenience and usability of the framework [76].

Moreover, to set up the local blockchain, another tool called Ganache [77] is used. It is an Ethereum simulator that serves as a local in-memory blockchain for development and testing purposes, simulating key features of an actual Ethereum network and providing users with a range of accounts funded with test Ether [78].

In this thesis work, the focus is on testing smart contracts rather than deploying them on a local blockchain separately and working on them. Running `$ truffle test` automatically sets up Ganache as a local test instance, facilitating contract deployment during test time. This will ease the testing of the smart contract project and further simplify the containerization process as a single command simply takes care of the whole testing pipeline.

Last but not least, Truffle is not only very popular among many developers but also benefits from great community support, which allows for an effortless resolution of issues or bugs.

5.1.2. Hardhat

Hardhat is a relatively newer development environment for Ethereum software compared to Truffle, but it has quickly gained popularity in the smart contract development community. It offers a comprehensive suite of tools for editing, compiling, debugging, and deploying smart contracts and dApps, creating a complete development environment [79]. Furthermore, it operates through scripts, functioning as a task runner that allows for the automation of the development workflow.

One of the key selling points of Hardhat is its improved speed in comparison to Truffle, especially during the testing phase. Compared to Truffle, Hardhat provides enhanced flexibility in smart contract development, offering faster testing processes by leveraging best practices as outlined in the Hardhat documentation [79, 80]. Similar to Truffle, the tests are also written with JavaScript and it offers the flexibility to replicate all the functionalities provided by Truffle. Additionally, dependency management in Hardhat is carried out via npm. An added advantage is the incorporation of the Hardhat Chai Matchers [81], which enhances the readability of smart contract tests by introducing Ethereum-specific capabilities to the Chai assertion library. For example, developers can use this plugin to assert that a specific event is fired by a contract or that a contract reverts with a specific message. Additionally, Hardhat provides easy ways to track specific changes to a wallet's Ether or token balance resulting from a transaction [79]. With Hardhat, developers can use `console.log` instead of a debugger, offering a convenient and straightforward way to resolve bugs. However, one drawback of Hardhat is the absence of address management, requiring more effort to interact with a specific contract in tests compared to Truffle, as it demands hard-coded addresses in the code.

While Truffle relies on Ganache, Hardhat employs its own node, the built-in Hardhat Network, to set up the local blockchain. This local Ethereum network node possesses similar functionalities to Truffle's Ganache, enabling the deployment, testing, and debugging of smart contracts.

Hardhat utilizes Ethers.js [82] as the default library, in contrast to Truffle Suite which defaults to Web3.js [83]; however, developers also have the flexibility to use Web3.js as the default library in Hardhat. Additionally, it is worth noting that Web3.js is the older library, while Ethers.js is a newer one with a more user-friendly syntax and extensive documentation. Furthermore, since Web3.js is written in node.js and Ethers.js is in TypeScript, Ethers.js has a comparatively smaller bundle size compared to Web3.js [84].

Typescript, offering a statically typed experience for JavaScript development, ensures strong typing and enhanced code integrity during development [85]. By utilizing TypeScript, developers can easily incorporate custom functionality into their programs and effortlessly integrate external tools, enabling the creation of highly extensible projects when working with Hardhat.

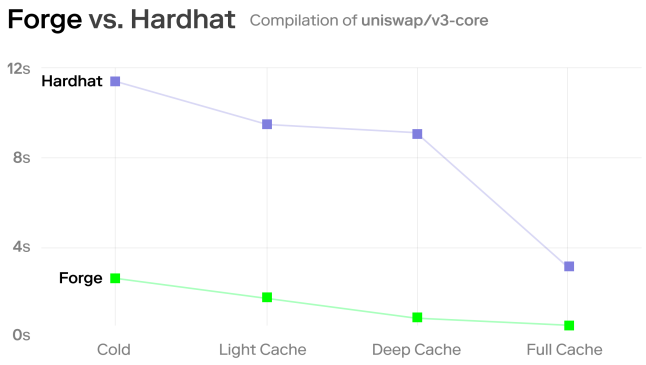

Figure 5.1.: Forge (Foundry) vs. Hardhat - Compilation of `uniswap/v3-core` (*Source:* [41])

5.1.3. Foundry

Foundry, a newer test runner framework compared to Truffle and Hardhat, has gained significant attention in the smart contract development community due to its promise of being considerably faster [39]. Written in Rust, Foundry has demonstrated remarkable speed in its performance [41]. According to the information available on Foundry's GitHub repository [41], the framework outperforms DappTools, another test runner framework on which Foundry is built, with an impressive speedup of up to *140x*, and consistently compiles faster than Hardhat by a factor of *1.7-11.3x* (see Figure 5.1). Furthermore, Foundry's portability, with a compact size of just 5-10MB, and its straightforward installation process that does not require any package manager contribute to its overall appeal [41].

Foundry's architecture differs slightly from that of Truffle and Hardhat. Notably, the dependency management is done via git submodules instead of npm packages, potentially making it more lightweight. Moreover, the tests are all written in Solidity, a departure from JavaScript, and similar to Hardhat, it permits the use of `console.log` for simplified debugging. The choice of Solidity for tests offers several advantages, such as eliminating the need for conversions like BigNumber and bypassing the requirement for an Application Binary Interface (ABI) for contract interaction. Having the tests in Solidity simplifies the technical stack of the project, as it eliminates the need to include any other language apart from Solidity. Additionally, the use of the EVM implementation in Foundry proves to be highly convenient for testing purposes.

Forge, a command-line tool bundled with Foundry, facilitates testing, building, and deploying smart contracts. It offers advanced testing methods, including Fuzz Testing, Invariant Testing, Differential Testing, and eventually Symbolic Execution and Mutation Testing. Fuzz Testing is particularly valuable in assessing smart contract vulnerabilities and ensuring method resilience under diverse inputs. It is a testing technique that where a specific program is tested with inputs that are systematically generated in a random and iterative manner to test a specific program [86]. Forge supports property-based testing, which allows for

testing general behaviors rather than isolated scenarios. The tool simplifies Fuzz Testing by automatically generating multiple scenarios, representing different inputs, and executing test methods with each of them [40].

Additionally, Foundry encompasses several other essential components, which are Cast, Anvil, and Chisel. Cast serves as a versatile tool, enabling seamless interactions with EVM smart contracts, facilitating transactions, and managing chain data. Anvil functions as the local Ethereum node, similar to Truffle's Ganache and Hardhat's Hardhat Network. Finally, Chisel offers a fast and comprehensive Solidity Read-Eval-Print Loop (REPL), empowering developers to experiment with code snippets and efficiently test and explore Solidity code [41].

Moreover, the Foundry framework is shipped with a set of cheatcodes [87] to manipulate the blockchain's state and test for specific reverts and events. They enable easy modification of parameters such as the block number and runner's identity, eliminating the need to keep track of signers and contract addresses. In addition, they allow developers to dynamically change the identity of an account during testing, making it effortless to use and fund multiple dummy accounts within a single test method [40, 87].

Foundry also provides invaluable output for testing, such as gas reports and snapshots, aiding in the estimation and tracking of gas consumption during tests. This feature may particularly be essential for smart contract testing in an educational context, where gas consumption stands as a significant performance metric. Moreover, Foundry boasts a fast and flexible compilation pipeline with automatic detection and installation of the Solidity compiler, incremental compilation, caching, and parallel compilation as needed. The framework also supports fast Continuous Integration (CI) to automate the building and testing of code changes [40].

In summary, Foundry proves to be a promising alternative to Truffle and Hardhat, offering advantages such as speed, lightweightness, convenience, and advanced testing capabilities with built-in tools.

5.2. Selected Smart Contract Projects for Analysis

In this thesis work, two smart contract projects have been deemed feasible for analysis. The first project is the *Vending Machine* [88], a lightweight implementation consisting of a single contract. This smart contract facilitates the purchase of donuts using Ether and includes functionality for restocking the machine once all the donuts are sold out.

The second project is *BBSE Bank 2.0* [89], an evolved and more complex version of *BBSE Bank* [90], which was previously presented in the BBSE course at TUM. *BBSE Bank 2.0* is a dApp consisting of 3 smart contracts that make use of the OpenZeppelin Contracts [91], a library for secure smart contract development. This dApp provides users with the opportunity to earn interest by depositing Ether. The amount of interest earned is determined by a predetermined annual return rate. Users have the flexibility to withdraw their Ether back at any time, but they stand to accumulate more interest the longer their deposit remains untouched. As a reward for their participation, users receive BBSE tokens, which are built on

the OpenZeppelin's ERC20 token standard. Moreover, in the latest version, *BBSE Bank 2.0*, users have the added capability to borrow ETH by offering their BBSE tokens as collateral [89].

Both of these projects offer a diverse range of characteristics and functionalities, making them ideal candidates for the research and analysis. Additionally, it is worth noting that *BBSE Bank 2.0* is a more complex and resource-intensive smart contract project compared to the *Vending Machine*. With the 3 smart contracts developed for the project that import OpenZeppelin Contracts, the compilation of *BBSE Bank 2.0* requires a total of 8 smart contracts to be compiled, whereas the *Vending Machine* only consists of a single compiled smart contract. This contrast in complexity adds depth to the analysis and provides developers with the opportunity to explore a wide range of scenarios and performance characteristics.

5.3. Usability and Development Experience

There are two main options for writing automated tests for Ethereum smart contracts: JavaScript and Solidity. As discussed above, the main testing language for Truffle and Hardhat is JavaScript, whereas Foundry uses Solidity tests. It is worth mentioning that TypeScript can also be used with both Truffle and Hardhat. However, for the purposes of this comparison, JavaScript was chosen for use with both Truffle and Hardhat due to its simplicity in implementing concise scripts. It should be addressed that while Truffle also accepts tests written in Solidity, this project scope will only include the tests written in Solidity with the Foundry framework.

5.3.1. Truffle

The Truffle framework comes bundled with the Mocha testing framework [92] and the Chai assertion library [93]. Developers use the testing syntax of Mocha and perform the actual assertions via Chai's assertion functions [94].

Setting up Truffle is a simple process. By globally installing Truffle with npm and running Truffle's `$ truffle init` command, a bare Truffle project can be created. Moreover, Truffle provides pre-configured sample applications and project templates with helpful boilerplates, known as Truffle Boxes, which makes project initiation effortless [76]. Dependency management is handled by npm, and all project configurations are defined in the *truffle-config.js* file. Furthermore, executing the `$ truffle test` command after creating a simple smart contract with a test script will run the test for the smart contract. Additionally, contracts can be pre-compiled using the `$ truffle compile` command, a useful feature that will be explored in the upcoming sections when containerization of the projects is discussed.

Truffle is a mature framework with extensive documentation [76], and it has a strong community support. These aspects can be decisive factors, even if the framework may have slightly slower runtimes. Furthermore, JavaScript, a widely adopted programming language, is familiar to many developers, making Truffle accessible to developers with varying levels of experience. The learning curve of the framework is relatively smooth, allowing developers to

quickly adapt and utilize its capabilities effectively.

To initiate the analysis, Truffle was selected as the starting point. For the smaller smart contract project, *Vending Machine*, 7 distinct tests were written, providing 100% Line and Branch Coverage. These tests cover various scenarios, such as verifying if donuts can be purchased by single or multiple accounts, or if the purchase of donuts is prevented due to insufficient payment. In the case of the bigger and more complex project, *BBSE Bank 2.0*, 3 JavaScript test files, containing 27 tests in total, for 3 different smart contracts were already available on the GitHub repository. These tests achieved 100% Line Coverage and 93.33% Branch Coverage, extensively covering different scenarios by utilizing various accounts with different Ether balances. The test functionalities utilized in these tests are extremely comprehensive, as they effectively assess multiple scenarios. Therefore, implementing equivalent functionalities with the other two frameworks, Hardhat and Foundry, will offer adequate insights into these frameworks.

For illustrative purposes, the following code block (see Listing 5.1) highlights several significant functions that represent how specific operations are performed with Truffle.

```
// A test suite is defined like this
// Available accounts (addresses) are passed as an argument by default by Truffle
contract("BBSEBank", (accounts) => { /** Tests go here! */ })

// Sending 10 Ether from an unused account to the BBSE Bank
await web3.eth.sendTransaction({
    from: accounts[6], to: bbseBank.address,
    value: web3.utils.toWei("10", "ether") // Ether needs to be parsed to Wei
})

// Get the balances of the second account and the BBSE Bank
web3.eth.getBalance(accounts[1])
web3.eth.getBalance(bbseBank.address)

// Deposit 5 Ether to the BBSE Bank using the third account
await bbseBank.deposit({
    from: accounts[2],
    value: web3.utils.toWei("5", "ether")
}, /** Rest of the arguments */)

// Update the rate using the owner, the first account
await oracle.updateRate(33, /** Rest of the arguments */, { from: accounts[0] })

// Call oracle.getRate function using a non-owner account
// If the following call does not throw an error, then "GetNewRate" is emitted
const tx = await oracle.getRate({ from: accounts[1] })
truffleAssertions.eventEmitted(tx, "GetNewRate", (event) => {})
```

Listing 5.1: Several Important Functions in JavaScript with Truffle

In summary, Truffle provides a developer-friendly experience, making it easy to retrieve some test accounts (with addresses provided by the framework) preloaded with some Ether. It also simplifies sending transactions between accounts or contracts and checking balances of accounts or contracts. Furthermore, the framework offers a convenient approach to modify the caller's identity when invoking a function and to specify a particular amount of Ether for the call. This is achieved by passing the payload object, which contains `from` and `value`, as the final argument to the function. It should be noted that when passing Ether to a function, it must first be converted to *Wei* using the `toWei` function of `web3.utils` package. Additionally, Truffle comes with some assertion methods that enable the verification of whether a specific event has been triggered. To utilize this functionality, the truffle-assertions [95] package needs to be installed.

5.3.2. Hardhat

The Hardhat framework relies on Ethers to connect to Hardhat Network, and similar to Truffle, it also employs Mocha and Chai for the tests but still offers its own built-in assertion library. This library includes custom Chai matchers and the Hardhat Network Helper that streamline test code writing and eliminate the need to import external libraries like Chai [79]. This makes Hardhat more optimized and tailored for testing Ethereum smart contracts. Despite these advantages, Chai was also chosen during the transition from Truffle to Hardhat to minimize modifications in the existing tests written with Truffle.

The installation process for Hardhat is straightforward. Unlike Truffle's global installation, Hardhat is locally installed in the project via npm, providing developers with a reproducible environment to avoid future version conflicts [79]. To create a new project, developers start an npm project using the *npm init* command, add Hardhat as an npm dev dependency[1], and then execute `$ npx hardhat` to generate an empty or sample project structure. Having the dependency management also handled by npm, Hardhat keeps all project configurations in the *hardhat.config.js* file. Besides, it manages everything with scripts, although this is not relevant for the testing purposes. The tests are simply run with the `$ npx hardhat test` command. In addition, the contracts can be compiled as well with the `$ npx hardhat compile` command. It should be noted that Truffle, similar to Hardhat, offers the option for local installation via project dependencies. This aspect will be further explored and detailed in the upcoming sections focusing on the containerization process.

Regarding the development experience, Hardhat did not exhibit significant differences when compared to Truffle. The main visible distinction between the two frameworks is that Truffle includes a more graphics-based interface that prioritizes ease of use for developers, while Hardhat primarily emphasizes the use of the command line [94]. Rewriting all the tests for both projects, 7 tests for *Vending Machine* and 27 tests for *BBSE Bank 2.0*, required some research and reconfiguration, as identity, ether management, and the use of test accounts in Hardhat differ slightly from Truffle. Furthermore, the transition from Web3.js to Ethers.js also

[1]Development dependencies, often referred to as *dev dependencies*, are specific npm dependencies required exclusively for the project development.

took some time, but ultimately resulted in a more readable code.

In the code block of the preceding section (see Listing 5.1), the implementation of several important functions with Truffle was demonstrated. Now, the subsequent code block (see Listing 5.2) illustrates the implementation of these functions using Hardhat.

```
1  // A test-suite is defined as follows, but this time no test account is passed
2  describe("BBSEBank", () => { /** Tests go here! */ })
3
4  // The test accounts (signers) are retrieved using Hardhat's ethers library
5  const accounts = await ethers.getSigners();
6
7  // Sending 10 Ether from an unused account to the BBSE Bank
8  await accounts[6].sendTransaction({
9      to: bbseBank.target, // "target" instead of "address"
10     value: ethers.parseEther("10") // Truffle: web3.utils.toWei
11 })
12
13 // Get the balances of the second account and the BBSE Bank
14 ethers.provider.getBalance(accounts[1].address) // Truffle: web3.eth.getBalance
15 ethers.provider.getBalance(bbseBank.target) // "target" instead of "address"
16
17 // Deposit 5 Ether to the BBSE Bank using the third account
18 await bbseBank.connect(accounts[2]).deposit({
19     value: ethers.parseEther("5")
20 }, /** Rest of the arguments */)
21
22 // Update the rate using the owner, the first account
23 await oracle.connect(accounts[0]).updateRate(33)
24
25 // Check if calling oracle.getRate using a non-owner account emits "GetNewRate"
26 await expect(oracle.getRate({ from: accounts[1] })).to.emit(oracle, "GetNewRate")
27
28 // Expect bbseBank.payLoan to be reverted with the following message
29 await expect(bbseBank.connect(accounts[1]).payLoan({
30     value: ethers.parseEther("0.0001") // Less than the borrowed Ether is provided
31 })).to.be.revertedWith("The paid amount must match the borrowed amount")
```

Listing 5.2: Several Important Functions in JavaScript with Hardhat

In terms of the coding differences, the same functionality was successfully achieved in the tests with Hardhat, although some variations were observed (see Listing 5.2). Firstly, Hardhat utilizes Ethers.js as its default library, while Truffle uses Web3.js. As a result, certain functions, such as parsing Ether and retrieving balance, are performed using *ethers* instead of *web3*. Secondly, as shown in the code block above, Hardhat does not automatically pass test accounts as arguments to the test suite, as evident in *Line 2* of Listing 5.2. Instead, one must call the `getSigners` function from Hardhat's *ethers* library to obtain the accounts (*Line 5*). It should be noted that the accounts returned are *HardhatEthersSigner* objects, unlike Truffle, where the accounts are represented as addresses. `HardhatEthersSigner` includes

various functions, such as the `sendTransaction` function used for sending Ether. Thus, when sending Ether between two accounts, the sender's `sendTransaction` function is called instead of specifying the sender's address in the payload object (*Line 8*). Furthermore, to get the balance of an account, the address attribute of the account object must be provided (*Line 14*). In addition, the balance of a contract is obtained using the contract's address, which, unlike Truffle, is accessed via the *target* field instead of *address* (*Line 15*). Thirdly, changing the identity before calling a function is not accomplished through the `from` field in the payload object; instead, it is achieved through a contract's connect function (*Line 18*). Lastly, Hardhat provides special Chai matchers known as Hardhat Chai Matchers, which extend `expect` and allow checking if a specific event has been triggered (*Line 26*) or if a particular function call is reverted (*Line 29*). In Truffle, the latter check is performed using a *try/catch* block.

In a nutshell, despite the variations in test implementation, both frameworks, Truffle and Hardhat, achieve the same functionalities. Hardhat, however, offers enhanced flexibility in the development of smart contracts [80].

5.3.3. Foundry

The Foundry framework employs Solidity as the language for its tests. The functionality of the tests will be replicated using this framework, with all tests being rewritten in Solidity, which is a language fundamentally different from JavaScript. Consequently, more extensive code changes are anticipated during this process. Unlike JavaScript tests that can only cover external and public Solidity functions, and also evaluate contract behavior from an external client's perspective using contract abstractions and web3/ethers, Solidity tests enable a more direct examination of each function in a contract in a bare-to-the-metal fashion [96]. This advantage arises because Solidity tests are written in the same language as the contracts being tested.

Though Foundry is a relatively newer framework compared to the others, the official documentation, Foundry Book [40], offers a great way to get familiar with the framework. The initial steps with Foundry, ranging from installation to obtaining the initial results, were straightforward and thoroughly documented. In contrast to the previous frameworks, Foundry is not shipped as an npm package. The framework can be installed by building the project from source with the Rust compiler and Cargo, the Rust package manager, or simply using Foundryup, the Foundry toolchain installer. For the sake of speed, simplicity, and having to run fewer commands, the latter method was opted for. In the initial step, Foundryup is installed using the command `$ curl -L https://foundry.paradigm.xyz | bash`, followed by the execution of `$ foundryup` to obtain the latest (*nightly*) precompiled binaries of Foundry's essential components: *forge*, *cast*, *anvil*, and *chisel* [40]. Subsequently, *Forge*, a command-line tool packaged with Foundry, can be utilized to create a Foundry project using the command `$ forge init`. Additionally, as discussed in the previous sections, Foundry manages dependencies through git submodules rather than npm packages. Hence, to install a dependency, the command `$ forge install package_name` is sufficient. The package is then installed by cloning its source code from GitHub under the `lib` directory by default and the git submodule is added to the *.gitmodules* file of the repository, indicating

that the package is another git repository, and its contents will not be committed to the current repository. For instance, OpenZeppelin Contracts are installed via the command `$ forge install Openzeppelin/openzeppelin-contracts`. Once the Foundry project is initialized, and the dependencies are installed, running tests located in the `src` directory can be accomplished using the `$ forge test` command (this is the default configuration, similar to Truffle and Hardhat's use of the `contracts` directory). Overall, this simplicity in setup and testing enhances the usability and accessibility of the Foundry framework.

Foundry, just like the other frameworks, comes with comprehensive documentation, and with the Solidity programming language being around for a few years, there are adequate resources available online. However, being a relatively new framework, finding online help or resolving issues specifically related to the framework can be more cumbersome compared to Truffle and Hardhat. Certain errors encountered during this work required additional time to resolve compared to those encountered in Truffle and Hardhat.

The development process with Foundry required extra time due to the difference in the programming language used for writing tests. Consequently, the transition from Truffle to Foundry was not as seamless as the transition from Truffle to Hardhat. For instance, in Foundry, changing the identity and using different test accounts is not done via some accounts provided by the framework, as in Truffle and Hardhat, but via Foundry's cheatcodes. These cheatcodes provide developers with powerful assertions and the ability to change the state of the EVM and perform mocking. The Solidity interface for all cheatcodes in Forge is accessible in the Cheatcodes Reference section of the Foundry Book [87].

The implementation of various key functions, previously showcased with Truffle and Hardhat in the code blocks of the preceding sections (see Listing 5.1 and Listing 5.2), required further adaptations to align with Foundry's unique approach. These adaptations are demonstrated in the upcoming code blocks (see Listing 5.3 and Listing 5.4). Initially, the cheatcodes are defined through a custom interface, as shown in the code block below (see Listing 5.3).

```
1  // Cheatcodes: https://github.com/foundry-rs/forge-std/blob/master/src/Vm.sol
2  interface Vm {
3      function prank(address) external; // Change msg.sender for the next call
4      function startPrank(address) external; // Change msg.sender until stopPrank
5      function stopPrank() external; // Reset msg.sender changed by startPrank
6      function deal(address, uint256) external; // Set the balance of an address
7      function roll(uint256) external; // Set the block number (block.number)
8      function expectRevert() external; // Assert the next call reverts
9      function expectRevert(bytes calldata) external; // ... with a message
10     function expectEmit() external; // Assert a specific log (emit) is emitted
11     /** ... and more as needed */
12 }
13
14 abstract contract ExtendedDSTest is DSTest {
15     // HEVM_ADDRESS is 0x7109709ECfa91a80626fF3989D68f67F5b1DD12D
16     Vm internal constant vm = Vm(HEVM_ADDRESS);
17
18     // address(0) is reserved for console.log and address(this) is the owner!
19     // Start at 10 to skip the first few accounts
20     uint160 internal constant firstTestAccId = 10;
21 }
```

Listing 5.3: Defining Foundry's Cheatcodes Interface

Implementing cheatcodes is straightforward (see Listing 5.3); it involves creating an interface and writing the definitions of the functions intended to be used. Normally, instantiating this interface in a test contract with the address `HEVM_ADDRESS`, which makes the cheatcodes available, allows the use of the cheatcodes defined in the interface (*Line 16*). To improve code readability and increase reusability of these cheatcodes across multiple test files, an abstract contract named `ExtendedDSTest` is created (*Line 14*). Other test contracts then inherit from `ExtendedDSTest` to utilize these cheatcodes.

It is worth noting that `ExtendedDSTest` inherits from `DSTest` [97], a library bundled with the Forge Standard Library, which provides basic logging and assertion functionalities [40] (*Line 14*). By inheriting from `ExtendedDSTest`, test contracts automatically gain access to these functionalities as well, as `ExtendedDSTest` already inherits from `DSTest`. Thus, there is no need to inherit from `DSTest` in the test contracts.

Another critical aspect to consider is the difference in the use of test accounts (addresses). In the other frameworks, Truffle and Hardhat, `accounts[0]` represents the owner, and the test accounts start from index 1 (`accounts[1]`, `accounts[2]`, etc.). However, it is observed that Foundry designates `address(this)` as the owner's address, while `address(0)` is reserved for `console.log`. Hence, for achieving consistency, the test accounts are started from an arbitrary index, defined as `firstTestAccId`, and set to 10 within the `ExtendedDSTest` contract (*Line 20*). Subsequently, this variable is used in the test contracts by incrementing its value as needed. For instance, the address of the first and second test accounts would be `address(firstTestAccId)` and `address(firstTestAccId + 1)` respectively. It is essential to note that the type of this variable is `uint160` as the `address` function expects the input to

be of type `uint160`. This approach simplifies the process of obtaining addresses and avoids unnecessary conversions.

The following code block (see Listing 5.4) illustrates how these cheatcodes are employed and how several crucial functions, previously implemented with Truffle and Hardhat, are now adapted to Foundry.

```
1  // A test suite, a test contract, is created like this
2  contract BBSEBankTest is ExtendedDSTest { /** Tests go here! */ }
3
4  // Sending 10 Ether to the BBSE Bank
5  vm.deal(address(bbseBank), 10 ether);
6
7  // Get the balances of a random account (42) and the BBSE Bank
8  address(42).balance;
9  address(bbseBank).balance;
10
11 // Deposit 5 Ether to the BBSE Bank using the third test account
12 vm.roll(block.number + 1); // Increment block number to simulate a real chain
13 vm.prank(firstTestAccId + 2); // Inject a change of user
14 vm.deal(firstTestAccId + 2, 5 Ether); // Deal 5 Ether to that user
15 bbseBank.deposit{value: 5 Ether}(); // Deposit 5 Ether to the bank
16
17 // Update the rate using a non-owner account by using a different cheatcode
18 vm.startPrank(firstTestAccId + 6); // Identity stays the same until stopPrank
19 oracle.updateRate(ORACLE_RATE); // + more function calls with the same identity
20 vm.stopPrank(); // Identity injection ends
21
22 // Check if calling oracle.getRate using a non-owner account emits "GetNewRate"
23 vm.prank(firstTestAccId); // Inject a change of user
24 vm.expectEmit(); emit GetNewRate("ETH/BBSE"); // Expect before function call
25 oracle.getRate(); // Call the function now
26
27 // Expect bbseBank.payLoan to be reverted with the following message
28 vm.expectRevert("The paid amount must match the borrowed amount");
29 bbseBank.payLoan{value: 0.0001 ether}(); // Must provide the borrowed Ether
30
31 // No need to use expectRevert if the function name starts with "testFail"
32 function testFail_5_RevertWhen_PayingLoanWithInvalidLoanAmount() public { ... }
```

Listing 5.4: Several Important Functions in Solidity with Foundry

Foundry successfully achieves the same functionality in testing as the other two frameworks. While each framework offers the ability to test the same scenarios, they differ in efficiency and coding style. The presented code blocks (see Listing 5.3 and Listing 5.4) demonstrate the definition and utilization of cheatcodes. To begin, a test contract is created by inheriting from `DSTest`, or in this project, from `ExtendedDSTest`, an extension to `DSTest` developed in this project for readability and reuseability (see *Line 2* in Listing 5.4). Crucially, the `DSTest` library also provides certain assertion functions such as `assertEq`, `assertTrue`, and `assertGt`.

Furthermore, the cheatcodes in Foundry enable identity changes and Ether transactions. For instance, the cheatcode `deal` provides the ability to charge Ether to an account or a contract using its address (*Line 5*). Additionally, the balance available in those addresses can be retrieved by the `balance` field of the object returned by calling the `address` function (*Lines 8 and 9*). To modify the state of the chain smoothly, the `roll` cheatcode can be used to alter the `block.number`, simulating a real chain when required by certain functions in the contracts (*Line 12*). Furthermore, Foundry's `prank` cheatcode is used for changing identity. It sets `msg.sender` to the specified address but only for the next call, including static calls, but not calls to the cheatcode address. As an illustration, depositing Ether into the bank involves simulating a user change using the `prank` cheatcode (*Line 13*), followed by the `deal` cheatcode to allocate Ether to the user's account (*Line 14*). Since invoking `deal` represents a call to a cheatcode, the caller of the subsequent `deposit` function (*Line 15*) is the same user injected via the `prank` cheatcode. To expand on this, as an alternative to `prank`, the cheatcodes `startPrank` and `stopPrank` can be used to enable identity changes for a sequence of calls. Any function call made between the invocation of `startPrank` and `stopPrank` will be executed by the user specified with the `startPrank` cheatcode (*Lines 18 to 20*). In the context of event listening, Foundry's cheatcodes can also be used to listen for events. For example, the `expectEmit` cheatcode, followed by `emit SomeEvent("Your event message")`, sets the test runner to expect the specified event to be emitted with the specified message (*Line 24*). Regarding testing conventions, Foundry's `expectRevert` cheatcode is utilized to verify if a call reverts, with a specified message if provided as an argument to the cheatcode (*Line 28*). Finally, Foundry also follows a specific naming convention for tests, where test names starting with `test_` are expected to succeed, while those starting with `testFail_` are expected to fail (*Line 32*). Therefore, if the test name starts with `testFail_`, it will pass in case of a revert without the need to explicitly use `expectRevert` [40].

To summarize the development process, the learning curve for Foundry was slightly steeper compared to Truffle and Hardhat, primarily due to the limited resources and the transition from JavaScript to Solidity for writing tests. Nonetheless, the development process becomes remarkably more streamlined and the code readability is enhanced due to fewer conversions and the use of native function calls, since both contracts and tests are written in Solidity. Besides, the use of cheatcodes to change the state of the chain, inject different identities and listen for event triggers enhances the attractiveness of the Foundry framework. Overall, the conclusion drawn is that the development experience with Foundry surpasses that of Hardhat and Truffle, providing a more seamless and efficient workflow.

5.4. Features and Tooling

This section highlights some of the significant features offered by each framework, contributing to the overall development and testing process.

5.4.1. Code Coverage

Each framework is capable of reporting code coverage with a detailed output, providing both line and branch coverage values. Hardhat and Foundry include this feature out of the box, whereas Truffle requires the installation of an additional package, solidity-coverage [98], to obtain coverage results.

5.4.2. Assertion Libraries for Testing

As for the assertion libraries, each framework has a primary library equipped with numerous assertion functions. By default, Truffle lacks an assertion library that can fully address certain scenarios, such as checking if an event has been emitted. However, developers can overcome this limitation by incorporating the truffle-assertions package into their projects, which adds these missing assertion functions. In contrast, Hardhat has a built-in library or plugin, Hardhat Chai Matchers, to enhance code quality and add testing capabilities for Ethereum. As discussed in the previous section, this plugin offers essential assertion functions, including the ability to verify if a contract call reverted. Meanwhile, Foundry comes with a built-in assertion library called DSTest, which is part of the Forge Standard Library. This comprehensive library not only includes basic logging and assertion functionality but also encompasses all the additional features found in other frameworks' additional libraries, along with some extras. In summary, each framework adopts its approach to performing proper assertions during testing, and all are fully capable of achieving the same functionality, although some may require an additional library installation.

5.4.3. Debugging

All the frameworks come with various debugging options to assist in identifying and fixing errors. It is essential to note that debugging a transaction on the blockchain differs from debugging in traditional programming languages like Java, C++, Python, etc. In those programming languages, the debuggers runs the code and interacts with it in real-time, whereas debugging transactions involves working with the historical execution of transactions on the blockchain, as long as the compilation or build artifacts are not cleared [76].

Truffle comes with an integrated debugger that resembles traditional command line debuggers. Any contract operation wrapped with the dedicated *debug* function creates a breakpoint, and when the test reaches this point, Truffle interrupts the normal flow and initiates the debugger, enabling developers to inspect the state, Solidity variables, and more [76]. On the other hand, although Hardhat does not have a dedicated debugger, it enables the ability to use `console.log`. Developers can take advantage of this feature to conveniently output logging

messages and contract variables to the console. This debugging approach is often favored by developers as it significantly simplifies the debugging process in most cases [79]. Finally, Foundry, ships with the most interactive debugger [99] among all the frameworks, offering a Graphical User Interface (GUI), which facilitates navigation through the execution history and more. It divides the terminal into four distinct pieces, each providing vital information on contract behavior, such as the program counter and the accumulated gas usage.

In brief, among the debuggers provided by each framework, Foundry's debugger exceeds the debugging capabilities of other frameworks and clearly distinguishes itself as a powerful tool for developing and testing smart contracts.

5.4.4. Mocking

Mocking is a valuable technique that simulates the behaviour of a function call without actually executing it. It significantly contributes to the overall smart contract development and testing process by allowing the examination of various scenarios. For instance, mocking can help address questions such as, "Does the function behave as expected when a function call reverts?" or "Does it properly set a specific variable to the specified value if the result from another function call is below a certain threshold?". Hence, mocking empowers developers to be able to test the contract's business logic thoroughly and precisely, enabling assertions across diverse scenarios.

Regarding the mocking capabilities of each test runner framework, Truffle and Hardhat do not offer a built-in mocking mechanism. Nevertheless, alternative solutions are available. One such option is to use Waffle [100], a separate library designed also for writing and testing smart contracts. Waffle facilitates the mocking of smart contracts with detailed documentation on installation procedures and how mocking operations are performed [101]. For example, the function returning the balance of an address can be mocked such that it returns a specific value when called during testing. In contrast, Foundry distinguishes itself with its remarkable cheatcodes interface. These cheatcodes provide some mock functions capable of simulating function calls and even function reverts, thereby offering similar functionalities as those available in Waffle [102]. This once again makes Foundry a very interesting choice for testing smart contracts.

5.4.5. Fuzz Testing

Fuzz testing (a.k.a. fuzzing) is an automated software testing technique utilized to detect bugs by injecting random and unexpected inputs into a program. This approach has been broadly employed as a fundamental method for conducting vulnerability testing of smart contracts [103]. By applying this technique, edge cases in a function can be thoroughly tested and analyzed.

During the investigation into fuzz testing with the test runner frameworks, it was discovered that Truffle and Hardhat does not ship with a mechanism for fuzzing but there are some alternative tools that can be employed to address this limitation. However, this would require yet another setup and integration, which could not only be time-consuming but

also add complexity and increase the project's size. Fortunately, Foundry takes care of this inconvenience with its built-in support for fuzzing. Foundry's forge incorporates a fuzzer [104], which facilitates property-based testing. Implementing fuzz testing with Foundry is straightforward – any test with at least one parameter will automatically undergo a fuzz run, generating 256 distinct scenarios. For instance, the function `testDeposit(uint96 amount)` gets executed multiple times with different input values of type `uint96` each time. It should be noted that the test contract is provided with a default amount of Ether equal to 2^{96} *Wei*, thus restricting the input value type to `uint96`. Last but not least, this fuzzer can also be configured globally or on a per-test basis, allowing more control over the fuzzing process (e.g. to have more or less than 256 distinct scenarios) [104].

5.4.6. Gas and Memory Limit

Through flags passed to the test command, developers have the flexibility to set the block gas limit and the memory limit of the EVM in bytes (with a default of 32 MB) [40]. While this feature may seem too specific, it plays an important role in controlling the execution of runs. For example, setting these limits could allow the service to terminate faulty or even fraudulent smart contracts that might otherwise run in the background, consuming excessive resources and keeping the service busy. By leveraging this capability, developers can safeguard the service from potential disruptions, thus enhancing its overall availability.

5.5. Test Output and Performance Metrics

When running the test command (without using any flags) with all the frameworks, the generated test output of each contains nearly identical information. It contains the names of test functions and their pass or fail status, along with a final summary that demonstrates the total number of tests executed across all test contracts. However, Foundry goes a step further to include a summary for each test contract, offering more detailed test results. Additionally, Foundry natively provides information about the gas consumption for each test function, a crucial metric for assessing the efficiency of smart contracts. In terms of visual differences, the test outputs (excluding the compilation logs) of Truffle and Hardhat appear identical. In addition, both include the execution times of poorly-performing test functions. In contrast, Foundry presents a distinct output, prioritizing detailed insights into the tests.

Moreover, Foundry does a great job in logging execution and setup traces offering a valuable feature that can be accessed easily using specific flags (e.g., *-vv*) [40]. This can help identify potential issues in the contracts, and the tree-like visualization of transactions can also be utilized for providing metrics the students in the educational context.

Finally, in the context of educational performance assessment, one of the simplest metrics that can be employed is the execution times of the tests. This metric does not differentiate between the frameworks and can be easily calculated, if deemed feasible for the educational purpose. It may serve as the most direct indicator of the overall efficiency of the smart contracts.

5.5.1. Gas Usage as a Performance Metric

Gas usage can serve as a valuable performance metric to measure how well contracts perform during the tests executed against them. All the frameworks offer gas usage estimation capabilities, although through different functions. Truffle makes use of its built-in *estimateGas* function, while Hardhat estimates gas using the *estimateGas* from the *ethers* library [76, 79]. Foundry, conversely, introduces its own gas tracking mechanism that displays the amount of gas consumed in each test function within the test results [40]. On the other hand, Truffle and Hardhat only allows gas consumption information to be printed through `console.log`, which may raise security concerns as these logs could be disrupted with the `console.log` statements written in the contracts [76, 79].

Additionally, Foundry also provides two reporting tools for gas tracking, *Gas Reports* and *Gas Snapshots*. While *Gas Reports* produces and logs estimations of the gas consumed by individual functions, *Gas Snapshots* outputs how much each test consumes in gas. Comparatively, *Gas Snapshots* are faster to generate but offer less granular reports compared to *Gas Reports*. However, *Gas Snapshots* have the advantage of providing more built-in tools, like the ability to compare gas usage between two snapshots (e.g., before and after changing the code) [40].

A noteworthy aspect of this tool is its potential value in the educational context. It can be utilized to generate a performance metric for students' smart contract projects. In this scenario, instructors uploading their projects can also include the contracts they have written. Once these instructors upload their contracts, the tool will generate the first gas snapshot. Then, when students upload their contracts, the tool will compare the gas snapshots of both the instructor-written contracts and the contracts of the students. This insightful comparison derives an informative metric that demonstrates the percentage of how well a student's contract performed in comparison to the instructor-written one. Consequently, this feature proves highly valuable for accurately assessing and scoring students' smart contracts, providing them with meaningful feedback to allow potential improvements to their smart contract implementations.

5.6. Truffle vs. Hardhat vs. Foundry: Intermediate Performance Results

The process of writing tests for both projects, *Vending Machine* and *BBSE Bank 2.0*, utilizing the Truffle, Hardhat, and Foundry frameworks, provided valuable insights into the distinct capabilities of each. Foundry emerged as the most robust platform for smart contract testing, followed by Hardhat, and then Truffle. This evaluation delivered a detailed understanding of the developer-friendliness, feature sets, and convenience offered by these frameworks, enabling a comprehensive exploration of their strengths and weaknesses. Such insights are crucial for making an informed decision about selecting the most suitable framework for the final testing service. Nonetheless, it is essential to also consider performance as a vital decisive factor, where even milliseconds matter, as this service is expected to be called frequently, and faster simultaneous runs will result in reduced waiting times for users.

Table 5.1.: Compilation & Test Execution Times of Frameworks

Framework	Project	
	Vending Machine	BBSE Bank 2.0
Truffle	2.71s	4.98s
Hardhat	1.90s	2.99s
Foundry	0.96s	1.59s

To evaluate performance, the tests were executed against the smart contracts in both projects using all three frameworks on a well-equipped system[2]. The performance results were obtained as the median of the time taken (measured in seconds) in 100 executions, involving both compilation and testing of the smart contracts. It is important to mention that to ensure consistent timing for each run, the cache was cleared after each execution to require recompilation prior to the subsequent tests.

As demonstrated in Table 5.1 (visually represented in Figure B.1 as a graph), the frameworks' prior rankings based on their developer-friendliness and feature sets are reflected in the performance results, demonstrating that convenience does not come at the expense of performance. Foundry continued to excel as the top performer, having achieved median compilation and testing times of 0.96 seconds and 1.59 seconds, respectively, for both the small (*Vending Machine*) and large (*BBSE Bank 2.0*) projects. Hardhat secured the second position, with execution times of 1.90 seconds and 2.99 seconds, while Truffle performed the poorest, taking 2.71 seconds and 4.98 seconds for the same projects.

Even on a system overpowered for testing just a single smart contract project at a time (a system with 10 CPU cores), Truffle took almost 5 seconds to compile and test the *BBSE Bank 2.0* project. This alone raises concerns, as its performance is likely to degrade further in containerized setups with less powerful computing resources. Compared to the lighter *Vending Machine* project, Truffle exhibited an approximate 83.76% increase in execution time[3] in the *BBSE Bank 2.0* project, which features larger test suites and more complex tests. This leads to the conclusion that the bottleneck is in the test-running process for Truffle. Meanwhile, Hardhat's execution times were about 57.37% longer in the more intricate project, with the entire testing pipeline taking just under 3 seconds. Foundry, on the other hand, successfully delivered exceptional performance, with the entire testing pipeline taking nearly 1.6 seconds. While Foundry's overall speed is significantly higher than that of Hardhat, the percentage increase in execution time between the two projects, *Vending Machine* and *BBSE Bank 2.0* was more noticeable for Foundry, with roughly 65.63% longer execution times. This indicates that while Hardhat may be faster in compilation, Foundry is exceptionally quick in test execution, with tests running in milliseconds, pointing to compilation as Foundry's bottleneck.

[2] Apple's ARM-based M1 Pro chip featuring 10 CPU cores (8 performance cores and 2 efficiency cores) and 16 GB of system memory (256-bit LPDDR5 SDRAM).

[3] An X% increase in execution time means that the execution duration is X% longer than a baseline measurement. For instance, if the baseline execution took 5 seconds, and it now takes 10 seconds, this is a 100% increase in execution time.

To summarize, these findings underscore the criticality of performance consideration, especially in scenarios where this service may be deployed and run frequently. Selecting a framework with faster overall execution times can remarkably enhance the efficiency of the service. It can be observed from Table 5.1 that all the frameworks demonstrated promising results completing the testing pipeline consisting of compilation and testing under 3 seconds for the lighter project and 5 seconds for the larger project. Foundry stands out as the winner, being blazing fast in finishing the testing process of *BBSE Bank 2.0* in 1.59 seconds, approximately 1.88 times faster than Hardhat. Hardhat, securing the second place, consistently proves its performance by executing the tests of *BBSE Bank 2.0* in 2.99 seconds, roughly 1.67 times faster than Truffle. Finally, Truffle, despite its maturity and community support, falls behind with an execution time of 4.98 seconds, coming in the last in this comparison.

5.7. Containerization and Scalability Assessment

This section will build on the previous evaluations of usability, development experience, tooling, and performance for the test runner frameworks, which are Truffle, Hardhat, and Foundry, focusing specifically on their containerization capabilities. The objective is to determine if these frameworks can maintain their performance while operating in a containerized environment, as facilitated by Docker, the chosen tool for containerization as outlined in section 2.3, without compromising the features they offer.

In theory, the setup and testing process for smart contract projects should not differ significantly from traditional applications when considering containerization. For this comparison, most images will be based on the latest Ubuntu image from the Docker registries, providing a reliable foundation for containerized environments. Specifically, Foundry is the only framework that provides its own Docker image and guidance for container use, demonstrating the feasibility of smart contract development and testing in such an environment. It is of utmost importance that testing smart contracts in a containerized environment is possible without compromising essential features or performance. Therefore, it is crucial to thoroughly analyze the containerization capabilities of each framework, examining how practical and straightforward it is to containerize applications using these tools, and evaluating any potential performance trade-offs when they are operated in isolated containers.

5.7.1. Setup

A project can be containerized by configuring it using either the *Dockerfile* or *docker-compose*. While *docker-compose* is more suitable for containerizing multiple applications or microservices, the *Dockerfile* is more appropriate for the purpose of this comparative analysis, which involves building a single smart contract project and executing tests on the smart contracts. It is a specific file used by Docker as a blueprint to build an image and run containers. Docker also utilizes caching, allowing it to avoid redundant operations, such as reinstalling dependencies when there are only changes in the code.

The *Dockerfiles* in this project are created by following the exact steps used in the local development and testing process, where each line corresponds to a step taken in the local environment to build and run the application. For the base image, the latest Ubuntu image, `ubuntu:latest`, is used for each of the frameworks. Additionally, for the Foundry framework, Foundry's own Docker image, named `ghcr.io/foundry-rs/foundry` and hereafter referred to as the **foundry-rs** image, is employed as well. This image is designed to be lightweight and provides a pre-configured environment for building and running Foundry projects. Therefore, it was deemed essential to also analyze the containerization capabilities of the framework using the image created specifically for it.

The complete pipeline for this comparison involves setting up the environment, copying project files to the image, installing packages and dependencies, and finally executing the tests. As mentioned before, the goal of the final service is to allow students to upload their smart contract inputs, which will then be tested with pre-written tests. This means that once the image has been built, the only thing that needs to change is the `src/contracts` directory. For this reason, two separate *Dockerfiles* are written for each project and framework, resulting in the creation of two images. The *build image* is responsible for preparing the project folder by installing the dependencies, copying the tests into it, and building the project. On the other hand, the *app image*, built from the *build image* (as the base image), contains only the `src/contract` directory. When this *app image* is run as a container, it executes the tests with the copied contracts. This approach ensures that the project is set up and built only once, allowing for multiple test executions without the need to repeat the build process.

However, it should be mentioned that during the implementation of the final service, this approach may change to a more efficient one that creates a single image per project, as current approach would result in a build image and an app image for each submission. This would lead to as many images as there are submissions, quickly becoming unmanageable due to the large number of images, consuming excessive disk space. A potential final approach could involve having a single base image for the project, and each submission would then run containers from that image by first copying the contracts into it. This approach would significantly reduce the number of images in the system, making it more possible to handle a large number of submissions without running into storage limitations.

In summary, the initial version of *Dockerfiles* for each framework is created by replicating the exact steps required to run the framework on a local machine. These steps typically involve installing the framework itself, copying essential project files (such as configuration files and test code) into the image, installing necessary dependencies, and then finally moving the contracts into it. The resulting image can then be used to run a container, which executes all the tests. Additionally, it is worth mentioning that both Truffle and Hardhat were used locally, meaning that they were installed as dev dependencies via npm rather than having a global installation for the frameworks.

5.7.2. Optimization

In this section, some of the bottlenecks encountered during the containerization process will be discussed. The final version of containerization will be utilized in the implementation of the final service.

Containerization Version 1 (*v1*)

This is the base and simplest version discussed in the previous section.

Containerization Version 2 (*v2*)

In the initial version (*v1*) of containerization, the Solidity compiler was downloaded and installed each time the Docker container was executed. Clearly, this presented an additional overhead that could easily be remedied if this step was taken only once during the build process. To address this redundancy, the first improvement involved downloading and installing the compiler during the build process in the *base image*. As a result, the compiler was no longer downloaded and installed during test execution, speeding up the testing pipeline. Furthermore, there was still room for additional optimization. In the smart contracts, there are often some external library imports, such as importing OpenZeppelin's contracts like the ERC20 token in *BBSE Bank 2.0* project. These imported libraries are also compiled everytime the tests are run in a Docker container, although this is redundant as modifying the project code has no effect on them. Therefore, the compilation of these external libraries could also be moved to the build process. This was achieved by implementing an empty contract that includes all the required external libraries. Before copying the tests directory to the *base image*, the empty contract was copied, and then the project was built and compiled. As a result, the external libraries were compiled in the *base image*, not having to be recompiled during test execution, leading to a faster runtimes during testing. To summarize, in this optimized version of containerization, the Solidity compiler is downloaded and installed during the build process, and the external libraries required for the contracts are compiled as well. These improvements eliminate redundant operations during test execution and significantly improve the overall testing speed.

Containerization Version 3 (*v3*)

This optimization builds on top of version 2 (*v2*) by incorporating additional compilation tasks into the build process. It should be noted that this optimization is specific to Foundry because, unlike Truffle and Hardhat, which use JavaScript for testing, Foundry's tests are written in Solidity. Therefore, in Truffle and Hardhat, there is no distinction between versions 2 and 3 (*v2* and *v3*) in terms of performance. In contrast, Foundry could experience performance differences. This optimization primarily focuses on the helper contracts written for the tests, such as the one demonstrated earlier in Listing 5.3 where there is another contract file that imports necessary libraries for the test and defines some of the cheatcodes. In this version of containerization, such contracts and the libraries they import are compiled just once during

Table 5.2.: BBSE Bank 2.0 - Image Sizes with Containerization Versions

		Containerization Version		
Framework	Base Image	v1	v2	v3
Truffle	ubuntu	1050 MB	1080 MB	1080 MB
Hardhat	ubuntu	612 MB	647 MB	647 MB
Foundry	ubuntu	291 MB	311 MB	316 MB
Foundry	ghcr.io/foundry-rs/foundry	113 MB	133 MB	137 MB

the build process. Although this might not appear to be a significant improvement, it can still contribute to performance gains. This is achieved by writing the simplest version of the smart contracts, where only the constructor and functions are included without their implementation. To wrap up, this containerization version 3 (*v3*) specifically benefits the Foundry framework by eliminating redundant recompilation of helper Solidity code within the test directory, whereas Truffle and Hardhat do not benefit from this optimization due to the fact that the tests are written in JavaScript.

5.7.3. Image Sizes

The size of the container can play a huge role in determining the most feasible framework. It is critical for a Docker image to occupy as little space as possible on a machine, especially considering that the final service is expected to handle multiple projects. Furthermore, the service itself might be deployed on a machine with limited disk space and storage limitations. Hence, efficiently utilizing resources depends on keeping the images as minimal as possible.

As shown in Table 5.2, there is a noteworthy difference in image sizes even when using the same base images. Evidently, Truffle appears as the least lightweight option, with its image size exceeding 1 GB. This oversized Docker image is significantly inefficient, especially when compared to its competitor Hardhat, which manages to keep its image size just under 650 MB while still utilizing npm for package management. The difference of over 400 MB in image size between Truffle and Hardhat emphasizes that Truffle itself is a resource-intensive framework.

Conversely, Foundry is containerized using two different base images in this project: the `ubuntu` image and the `foundry-rs` image. Foundry proves to be more efficient than the other two frameworks, occupying approximately 51% and 71% less disk space than Hardhat and Truffle respectively, while still maintaining full functionality in a 316 MB package. The primary reason behind this efficiency is Foundry's reliance on git submodules for dependencies, as opposed to npm packages.

Finally, Foundry's own base image, `foundry-rs`, results in an even smaller image size of 137 MB, managing to occupy roughly 57%, 79% and 87% less space than Foundry built on the `ubuntu` image, Hardhat, and Truffle, respectively. However, it must be considered that this image is based on a different architecture, *x86_64*, than the host[4] and the `ubuntu` image,

[4]Apple M1 Pro (2021, 10-core CPU, 16 GB RAM).

Table 5.3.: Test Execution Times with Containerization Versions

		Containerization Version			
Project	Framework	v1	v2	v3	Not Dockerized
Vending Machine	Truffle (ubuntu image)	4.70s	3.58s	-	2.72s
	Hardhat (ubuntu image)	4.31s	2.68s	-	1.33s
	Foundry (ubuntu image)	1.76s	1.02s	0.71s	0.70s
	Foundry (foundry-rs image)	8.88s	7.69s	5.28s	0.70s
BBSE Bank 2.0	Truffle (ubuntu image)	8.27s	7.29s	-	4.97s
	Hardhat (ubuntu image)	7.02s	6.03s	-	2.42s
	Foundry (ubuntu image)	2.76s	1.52s	1.42s	1.33s
	Foundry (foundry-rs image)	14.56s	11.29s	10.91s	1.33s

aarch64, which may result in poor performance or failure if run via emulation, as Docker's warning message states. On the other hand, the `ubuntu` image offers a variety of architectures, still making it a reasonable choice despite the size difference.

5.7.4. Performance Results

This section evaluates the performance results of all frameworks across different containerization versions, with execution times[5] recorded in Table 5.3 (visually represented in Figure B.2 and Figure B.3 as graphs). These times indicate how long it took to run the Docker containers[6]. For comparative analysis, non-containerized execution times were also included. Unlike Table 5.1, which includes the combined compilation and testing times, Table 5.3 lists only the test execution times, excluding the compilation stage. To measure performance outside of a Docker container accurately, the pipeline from the containerization process was duplicated in a non-containerized setting. As part of this approach, the directory was prepared after each run to ensure that the same files were compiled during testing as those in the containerization process. This guaranteed that both containerized and non-containerized setups underwent identical testing steps, enabling an accurate comparison.

Containerization Performance of *v1*, *v2* and *v3*

The initial containerization approach, designated as *v1*, involves downloading and installing the Solidity compiler and compiling all of the Solidity files, including the dependencies. As indicated in Table 5.3, this version introduces a performance bottleneck across all frameworks. In the refined *v2* approach, where the compiler is preinstalled and dependencies are compiled, all frameworks complete the testing pipeline faster, with speedup[7] values ranging from approximately 1.13 to 1.82. Specifically, for the *Vending Machine* project, this optimization

[5]The performance results reflect the median execution times (measured in seconds) from 100 test executions.
[6]The resources provided for each container were 5 CPU cores and 8192 MB of memory.
[7]$Speedup$ is defined as $T1$ / $T2$, where $T1$ is the execution time before optimization and $T2$ is the execution time after optimization.

has the most benefits on Hardhat and Foundry (built with the `ubuntu` image) framework, performing about 1.60 and around 1.73 times faster, respectively. In the case of the more complex *BBSE Bank 2.0* project, the gains are less distinctive, except for Foundry (built with the `ubuntu` image), which exhibits a significant speedup value of approximately 1.82, while other frameworks only show speedup values ranging from around 1.13 to 1.29.

The transition from the second version, *v2*, to the third one, *v3*, avoids redundant compilations of certain Solidity files within the tests directory by facilitating their compilation during the build process. As discussed in the preceding sections, this advancement is exclusive to Foundry, as Truffle and Hardhat use JavaScript for testing. Foundry (built with both the `ubuntu` and `foundry-rs` images) seems to benefit from this optimization, seeing a boost in performance with a speedup value of about 1.45 for both images in the *Vending Machine* project. However, in the *BBSE Bank 2.0* project, the improvement is less significant, at just approximately 1.05 times faster, which could even be attributed to the margin of error between runs. The difference in performance between the two projects might come down to the number of test contracts. The *Vending Machine* project includes a single smart contract, which is tested by a single test contract, whereas the *BBSE Bank 2.0* project has three, with both projects involving a helper test contract that imports necessary libraries and defines cheatcodes. Thus, in *v3*, the number of solidity files that need to be compiled during test time decreases from three to one in the *Vending Machine* project (the actual test contract, helper contract, and DSTest library imported in that helper contract). However, in the *BBSE Bank 2.0* project, it decreases from five to three (two more test contracts for the additional smart contracts). As a result, this might be the reason why the improvement is more noticeable in projects with fewer smart contracts to test. Nevertheless, it is still a valuable improvement, reducing the number of files that need to be recompiled.

To summarize, both Truffle and Hardhat demonstrate similar performance results, with Hardhat outperforming Truffle with speedup values of around 1.34 and roughly 1.21 for *Vending Machine* and *BBSE Bank 2.0*, respectively. Although this places Hardhat in second place in terms of containerization performance, the difference between the two is so small that it alone cannot justify choosing Hardhat over Truffle. However, as observed in previous sections, Hardhat proves to be a more robust framework than Truffle in various aspects. Therefore, it can be considered the winner in this comparison between the two frameworks. Foundry, on the other hand, emerges as the top performer once again, surpassing the other frameworks with excellent execution times of 0.71 and 1.42 seconds in the *Vending Machine* and *BBSE Bank 2.0* projects, respectively. This places Foundry in the first position, outperforming both Truffle and Hardhat in terms of the containerization performance, with significant speedup values that are hard to overlook.

However, it must be noted that using the `foundry-rs` image instead of the `ubuntu` image introduces a significant performance degradation, causing the framework to run both projects approximately 7.44 and around 7.68 times slower, respectively. This drop in performance not only places the framework behind Truffle but also results in test execution times of 5.28 and 10.91 seconds, which are simply not feasible for practical use. The reason for this performance drop is related to using a different architecture than the host, which requires emulation. This

raises concerns, as the `ubuntu` image is shipped with a range of architectures and would likely be a more robust choice to work in different systems with different architectures. Additionally, even though the `foundry-rs` image results in a Docker image that occupies more than half a space, as shown in Table 5.2, Foundry with the `ubuntu` image still produces a lightweight image of 316 MB, which is more than twice as light as that of Hardhat. Consequently, it is safe to say that Foundry with both base images works, but the one built with the `ubuntu` image, being more robust, comes out as the winner.

Containerized vs Not Containerized

In the context of running the frameworks in a containerized environment, it is commonly expected to observe a drop in performance due to virtualization. However, it is crucial to assess the extent of performance loss for each framework when executed within a container. To conduct a meaningful comparison, the focus is primarily on the larger project, *BBSE Bank 2.0*, as it applies more pressure on the frameworks.

When comparing Truffle and Hardhat, some surprising trends can be observed from Table 5.3. Despite previously outperforming Truffle in all the analyzed aspects, Hardhat experienced significant slowdowns in a containerized environment, becoming approximately 2.49 times slower when executed within a container. In contrast, Truffle's performance only declined by 1.47 times compared to its non-containerized performance. As a result, it becomes evident that Hardhat does not efficiently handle containerization.

Meanwhile, Foundry, which outperformed the other two frameworks by completing the pipeline in a considerably shorter time, impressively maintains its performance even within a container, performing nearly identically to its non-containerized version. Completing the entire pipeline in 1.33 seconds outside a container and 1.42 seconds within a container, Foundry demonstrates exceptional efficiency in terms of containerization. The reason behind this great performance might be explained in the following section, where each container is executed with different hardware resources. It is plausible that Foundry's effectiveness regardless of containerization can be attributed to its minimal resource requirements.

5.7.5. Scalability Assessment

This comparison highlights the efficiency of each framework in terms of scalability. The tests in the *BBSE Bank 2.0* project were executed 100 times with each framework. In these executions, the CPU core count systematically adjusted across 10 groups of executions, each comprising 10 runs with different CPU core counts. The results presented in Table 5.4 (visually represented in Figure B.4 as a graph) demonstrate the median test execution times, measured in seconds, obtained from each set of 10 runs performed with varying CPU core counts for each framework.

To conduct these tests, the CPU core count was adjusted using the *cpus* flag while running the Docker containers (e.g., `$ docker run --rm -m 8192M --cpus 3.0 bbsebank2/foundry/app:v3`). The memory allocation was kept at 8192 MB, in line with the previous tests. The CPU core counts values used were 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.5, 2.0, 2.5, and 3.0. As noticed, the CPU

Table 5.4.: BBSE Bank 2.0 - Test Execution Times with CPU Core Counts

Framework	CPU Core Count									
	0.5	0.6	0.7	0.8	0.9	1.0	1.5	2.0	2.5	3.0
Truffle (v3)	80.91s	40.70s	25.83s	20.37s	16.66s	14.60s	9.10s	7.97s	7.72s	7.53s
Hardhat (v3)	34.87s	24.20s	18.76s	15.38s	13.22s	11.48s	7.37s	6.67s	6.39s	6.23s
Foundry (v3)	2.85s	2.35s	2.00s	1.78s	1.61s	1.48s	1.43s	1.42s	1.42s	1.42s

core count limit can be expressed as a floating-point number, allowing containers to access a specific percentage of a CPU core [105]. To provide an example, configuring the limit to 0.5 would grant the container a maximum of 50% of a CPU core.

Regarding memory configuration, adjusting this value had minimal impact on results as long as it exceeded a certain threshold, the minimum memory required for each framework's functionality. Truffle demanded a minimum of 512 MB, whereas Hardhat's requirement was lower, at 256 MB. In contrast, Foundry demonstrated the ability to operate with as little as 128 MB of memory. It must be noted that memory allocations were set in powers of 2.

As depicted in Table 5.4, both Truffle and Hardhat struggle when operating with limited resources. Truffle's test execution time approaches 100 seconds with 0.5 CPUs, and the framework cannot have feasible results until at least 1.0 - 1.5 CPUs are provided, a trend similarly observed with Hardhat. However, Hardhat manages to significantly outperform Truffle at low CPU settings such as 0.5 and 0.6 CPUs. Nonetheless, these resources are still inadequate for practical use in the context of these frameworks. Both Truffle and Hardhat need at least 1.5 CPUs to complete testing in 9.10 seconds and 7.37 seconds, respectively. While they perform close to their optimal speeds at 3.0 CPUs, they still remain slightly slower.

In contrast, Foundry's scalability analysis presents more satisfactory outcomes with fewer CPU cores. It reaches its optimal performance using 0.8 to 1.0 CPU cores. In comparison, Truffle and Hardhat show significant performance declines under the same conditions. Truffle's performance falls behind by approximately 11.44 to 9.86 times, while Hardhat's performance lags by about 8.64 to 7.76 times, compared to Foundry's peak efficiency using the same resources. Furthermore, with even fewer cores, the performance gap widens further, with Foundry surpassing Truffle and Hardhat by speedup values of approximately 28.39 and roughly 12.24, respectively, at 0.5 CPU cores. In conclusion, this showcases Foundry's established resource efficiency. Achieving optimal performance with just a single CPU core would significantly enhance the final testing service by effectively executing more submissions simultaneously. As a result, Foundry stands out as an easily scalable framework, capable of efficiently executing multiple instances concurrently on a single machine.

5.7.6. Lessons Learned and Conclusion

Insights into the containerization process were gained, considering both ease of containerization and performance for the three frameworks. Some key lessons learned include:

1) Truffle's containerization process was relatively effortless, but it suffered from long build times due to its large framework size.

2) Hardhat's containerization required specific packages to work, making it less robust and smooth than Truffle. Both Truffle and Hardhat needed to have Node.js installation on the image as the dependency management and framework installation were managed by Node.js.

3) Foundry's containerization process was straightforward as well, but it stood in need of some experimentation to find an efficient *Dockerfile* setup. Moreover, dependencies were hardcoded in the *Dockerfile*, which may in the future be in need of a script to read them from the *.gitsubmodules* file and add them to the *Dockerfile*. However, this approach might raise some issues if the projects were in subdirectories and there was another git project in the parent directory. Conversely, Truffle and Hardhat were easier in managing the dependencies, as all were defined in the *package.json* file, and simply running `$ npm install` sufficed.

In evaluations of framework performance within containerized settings, Foundry consistently outperformed the other two frameworks in all aspects. Remarkably, Foundry exhibited not only reduced disk space consumption, as shown in Table 5.2, with smaller Docker images, but it also exhibited faster test execution speeds. Specifically, as demonstrated in Table 5.3, for a lightweight project, Foundry was about 3.77 times faster than Hardhat and around 5.04 times faster than Truffle. For the more complex project with heightened test complexities, Foundry was approximately 4.25 times faster than Hardhat and roughly 5.13 times faster than Truffle. Hence, Foundry's performance dominance intensified as the number of tests increased, underscoring Foundry's superior efficiency in test execution. In brief, Foundry admirably preserved its outstanding performance even in a containerized environment.

Moreover, Foundry's lightweight nature and efficiency in resource utilization enabled it to deliver almost the same performance results in a container. This is crucial as the frameworks will be deployed as containers, and their performance inside containers is of great importance, regardless of their speed outside a container.

Lastly, Foundry showcased greater scalability potential by maintaining its performance even when allocated fewer CPU cores, as illustrated in Table 5.4. It started operating optimally at 0.8 CPU cores and exhibited viable results even at levels as low as 0.5 CPU cores. This characteristic empowers Foundry to function optimally in situations with limited computational resources or in scenarios involving multiple simultaneous executions.

In conclusion, considering all these aspects, it is evident that Foundry would be a safe and efficient choice for both containerized and non-containerized environments. Its efficient resource utilization would allow for more simultaneous executions by reducing the CPU count allocated for each run.

5.8. Discussion and Recommendation

In conclusion, the preceding sections comprehensively examined the test runner frameworks: Truffle, Hardhat, and Foundry. This analysis encompassed a detailed overview of each framework, assessing their usability, development experience, supplementary features, tooling, test output, performance metrics, containerization capabilities, and scalability.

Each of these frameworks features a straightforward installation process that is carefully documented on their respective websites, and they have all demonstrated successful testing in the *Vending Machine* and *BBSE Bank 2.0* projects. However, significant differences in usability and development experience emerged, as highlighted in section 5.3. While Truffle and Hardhat offered easier complexity in writing tests due to the tests being written in JavaScript, Foundry offered a more satisfying development experience, leading to enhanced code readability. After becoming familiar with the framework, it became clear that Foundry would be an attractive choice due to its user-friendly development environment. In addition, using Solidity in tests allowed for fewer conversions and direct function calls, which improved code efficiency and readability. In addition, Foundry's special features, particularly Foundry's cheatcodes, further simplified tasks such as state manipulation, caller identity changes, and specific revert and event testing.

Regarding feature sets and tooling, each framework offers code coverage analysis, although Truffle requires additional libraries for this feature. Truffle also relies on additional libraries for specific essential assertions, whereas Hardhat and Foundry come equipped with built-in robust assertion functions. Moreover, in terms of the debugging process, Foundry excels in debugging capabilities, providing the most powerful debugger among the three frameworks, with a GUI for transaction interaction. Additionally, Foundry offers built-in mocking possibilities through cheatcodes, eliminating the need for separate installations as required by the other two frameworks. Similar considerations apply to fuzz testing, a method for detecting edge cases by injecting random inputs. Foundry simplifies this process simply through the use of arguments in test functions, while the other frameworks require additional tools. Finally, all three frameworks allow the setting of gas and memory limits to enhance system security, such as terminating faulty or potentially fraudulent smart contracts.

Each of the three test runner frameworks provides test output, illustrating both successful and failed tests, with distinct output styles and levels of detail. As performance metrics, execution times and gas usage are viable candidates for measurement. While measuring execution times is straightforward, gas estimation requires specific functions. All three frameworks provide gas estimation functions, with Foundry going a step further by offering a metric illustrating the gas difference between different versions of the same project. This enables a clear testing of whether an optimization or code change has had a negative or positive effect. Moreover, this feature might also be valuable for academic use cases, allowing the comparison of student contracts with instructor-written ones.

Performance emerges as a crucial factor to consider, as clearly illustrated in Table 5.1. Foundry impressively showcased significantly faster median execution times (measured in seconds from 100 individual runs) compared to the other frameworks, with Hardhat closely trailing, and Truffle lagging behind with the slowest execution times. To elaborate, Foundry

outperformed Hardhat by completing the testing pipeline around 1.88 times faster and left Truffle far behind with a remarkable 3.13 times speed advantage.

The analysis of containerization capabilities has clearly favored Foundry. It not only maintained an image size 51% and 71% smaller than its competitors, as demonstrated in Table 5.2, but also outperformed the other two frameworks even within a containerized environment. As indicated by the median execution times from 100 individual runs in Table 5.3, Foundry exhibited exceptional performance compared to the other frameworks. It completed the entire pipeline for the more complex project (*BBSE Bank 2.0*) in just 1.42 seconds, while Truffle and Hardhat took significantly longer with execution times of 7.29 and 6.03 seconds, respectively. This performance difference is impossible to overlook, and it greatly enhances the feasibility of using Foundry in the final testing service as the test runner framework. When one framework outperforms the others by more than 4 times, even within a containerized setup, the choice becomes evident. Additionally, comparing Foundry's performance inside and outside of a container, the difference in results is minimal, further highlighting Foundry's efficient nature when working with containers.

In summary, this comprehensive analysis of the test runner frameworks, Truffle, Hardhat, and Foundry, has highlighted Foundry as the clear top performer in almost every aspect, consistently surpassing expectations with each section. Foundry resulted in more readable code with tests written in Solidity and offered a rich set of features, including mocking, fuzz testing, and comprehensive debugging with a GUI. Furthermore, Foundry also excelled in both non-containerized and containerized environments, maintaining exceptional performance even when operating within a container. In addition, as illustrated in Table 5.4, Foundry stood out, particularly in scenarios with lower CPU core counts. It maintained its remarkable performance, even with as few as 0.5 CPU cores, and achieved peak performance within the range of 0.8 to 1.0 CPUs, whereas the other frameworks struggled to produce feasible and workable results with such limited resources. With this assessment of scalability, Foundry secures its position as the most developer-friendly, efficient, fast, and scalable framework among the three options. Therefore, Foundry has been chosen as the preferred test runner framework for this project, and it will be further utilized in this work, integrated into the final testing service to facilitate the testing of submitted smart contracts.

6. System Design and Implementation

This chapter offers a detailed discussion on the design and implementation of the final testing service, incorporating the selected test runner framework identified as the most suitable for smart contract testing.

6.1. Stakeholders and Requirements

Before digging into the design and architecture of the testing service, it is essential to clearly define the stakeholders and requirements. This section outlines the stakeholders involved and specifies the requirements of the testing service.

6.1.1. Stakeholders

1) **Students:** In the core use case, students stand as the primary users who submit their smart contract inputs to obtain test and performance results.

   - **Needs & Challenges:** For optimal utilization, students must have unhindered access to the service, particularly during peak times, which are usually close to the deadline, as they could be graded depending on the needs of the instructors or educational institutions. Crucially, students should have the capability to view uploaded projects and effortlessly submit their smart contract inputs for evaluation. It is imperative that they receive clear feedback on the status of their smart contract inputs, including specifics like the number of tests passed/failed, the total or per-test gas consumption, and the gas difference compared to the instructor-provided solutions. Additionally, they need to be able to download their uploaded solutions. However, one anticipates potential delays during peak times, given the fact that students tend to make last-minute submissions. Furthermore, if students upload non-executable smart contracts, the resultant feedback may be unclear. To avoid this, students should pre-validate their smart contracts locally before submitting them.
   - **Influence on Design:** Students have a direct influence on the design, as this service is being developed for them and their educational development. It is critical to ensure that the testing service has a user-friendly interface; students should be able to easily navigate the process of submitting their smart contract inputs for specified projects and then analyzing the testing results.

2) **Instructors:** These are the individuals who define and upload projects. The uploaded projects include the test contracts against which students' smart contract inputs are assessed.

   - **Needs & Challenges:** In addition to the functionalities provided to students, instructors need the capability to upload and edit projects, view those projects, and analyze the execution results alongside test results. To be considered "valid," a project must include a functional smart contract input. Furthermore, the service must provide the instructors with an easy-to-use interface for setting an execution timeout, which serves as the maximum permissible time for the smart contract inputs submitted by students. Instructors should also be able to set specific test execution parameters, such as a gas limit for submissions. Furthermore, the service must provide instructors with project and student submission filtering features, as well as the ability to download any material published to the platform, whether it is a project or a student submission. Just like the students, the instructors are also strongly advised that they validate the projects locally before uploading, especially since projects that fail to pass all the tests may provide unclear execution results, especially if they fail due to excessive gas consumption.
   - **Influence on Design:** Instructors also play an essential role in influencing the service's design. Their decisions determine which resources are made available to students and the level of detail that students can view. In addition, instructors should be able to define submission quotas for students within a certain time frame.

3) **Educational Institutions:** These comprise schools and universities that might adopt this service for use in courses.

   - **Needs & Challenges:** For educational institutions, seamless implementation and utilization of this service in courses is significant. The provided repository should contain comprehensive information to set the service up effortlessly. Challenges may arise, however, when facilities have requirements that deviate significantly from the core scope of service.
   - **Influence on Design:** Although this stakeholder group has no direct influence on the overall design, it is important to consider the needs of different institutions in the design of the service, such as what measurements are made visible to students and how they are informed of their results.

4) **Developers:** Developers are an essential stakeholder group. They are responsible for building, maintaining, and improving this testing service.

   - **Needs & Challenges:** Similar to educational institutions, it is imperative that the setup and portability of the service are simplified for developers. They should also be provided with comprehensive technical documentation to facilitate service maintenance and the development of additional features.

- **Influence on Design:** While developers do not have a direct influence on the current design, they possess the capacity to evolve the design based on the introduction of new features and requirements.

6.1.2. Functional Requirements

This section outlines the functional requirements developed for the testing service based on the thesis objectives.

System Overview

The automated testing service facilitates the submission and evaluation of Solidity smart contracts in an educational context. The testing service leverages Foundry, a tool that was identified as the best-performing after evaluating various test runner frameworks.

User Roles and Functionalities

1) **User Authentication:**
   - **FR1.1**: The testing service shall offer a registration interface for students to create their accounts.
   - **FR1.2**: The testing service shall offer a login interface for all user roles, including students and instructors.
2) **Instructors:**
   - **FR2.1**: Instructors shall have the capability to upload smart contract projects.
   - **FR2.2**: The testing service shall provide an interface for instructors to edit previously uploaded projects.
   - **FR2.3**: The testing service shall validate the uploaded projects' correctness by running the uploaded smart contracts against the uploaded tests.
   - **FR2.4**: Instructors shall be able to set execution timeout parameters, indicating the maximum allowable duration for a student's smart contract to execute.
   - **FR2.5**: Instructors shall be able to define specific test execution parameters (e.g. gas limit for submissions).
   - **FR2.6**: Instructors shall be able to filter and view projects and student submissions.
   - **FR2.7**: The testing service must ensure that any material submitted to the platform can be downloaded by instructors, whether it is a project or a student submission.

3) **Students:**
   - **FR3.1**: Students shall be able to submit their smart contract inputs for testing against instructor-uploaded tests.
   - **FR3.2**: The testing service must display the results of the uploaded smart contract inputs, including the number of tests passed/failed, gas consumption, and gas difference measured against the instructors' smart contract inputs.
   - **FR3.3**: Students shall be able to filter and view their own submissions.
   - **FR3.4**: The testing service must ensure that students can download their uploaded submissions.

6.1.3. Non-Functional Requirements

In line with the thesis objectives, this section describes the non-functional requirements that the testing service should meet.

1) **Usability:**
   - **NFR1.1**: The user interface of the testing service must be user-friendly, allowing students and instructors to navigate and perform tasks with minimal complication.
   - **NFR1.2**: The execution details, containing all testing results, should be presented to users in a clear and straightforward manner.

2) **Reliability:**
   - **NFR2.1**: The testing service should have a high uptime, ensuring continuous service to students and instructors.

3) **Security:**
   - **NFR3.1**: Passwords and sensitive information should be encrypted.
   - **NFR3.2**: All user data, including smart contract uploads and test results, should be kept secure and protected from unauthorized access. While admins (instructors) should have complete access, users (students) should only be able to access their own submissions.

4) **Stability:**
   - **NFR4.1**: The testing service shall provide safeguards to handle the execution of inefficient or malicious code during smart contract testing.

5) **Efficiency & Performance:**
   - **NFR5.1**: The testing service must evaluate student-submitted smart contracts using the tests provided by the instructors efficiently, ensuring optimal performance.
   - **NFR5.2**: The testing service should be able to handle multiple simultaneous submissions from students, using message queueing, especially during peak times.

6) **Horizontal Scalability:**

- **NFR6.1**: The infrastructure and database should be structured to allow the testing service to scale out according to the number of users and submissions.

7) **Maintainability & Portability:**

- **NFR7.1**: The codebase should be well-documented and written in accordance with best practices to facilitate maintenance and potential feature additions.
- **NFR7.2**: The testing service should be built in such a way that it can be quickly transferred, deployed, or migrated to new platforms or environments.

6.2. High-Level Flow

Before digging into the system's overall architecture, it is critical to first analyze the testing service's high-level flow with a sequence diagram. Studies have established that sequence diagrams are one of the most widely used Unified Modeling Language (UML) diagrams, and they are a convenient way to depict a set of interacting objects and the sequence of messages that are exchanged between them, capturing the precise behaviour of the object classes in the system [106].

The primary purpose of the testing service is to enable instructors to upload exercises or projects. These uploaded files establish the groundwork for students to submit their smart contract inputs, which are afterwards checked against the instructor-provided tests, resulting in execution and test results. The overall flow of this procedure, from the instructor's exercise submission to the student's retrieval of test results, is illustrated in Figure 6.1. It is crucial to note that the high-level sequence diagram in Figure 6.1 is not a deep dive into the complexities of the testing service, but rather provides a broader perspective. For instance, the "*services*" object depicted in Figure 6.1 might not necessarily represent a singular service but could encompass multiple microservices.

To outline the exact flow: Instructors begin by authenticating and logging into the system. Once authenticated, they can upload an exercise. This step triggers the creation of a Docker image from the uploaded zip folder containing the project files. Before this exercise data is stored in the database, its correctness is verified by running a Docker container built from the recently constructed Docker image. If successful, the exercise information is committed to the database, and the instructors receive input regarding the execution and testing results. Following this, students can then authenticate themselves by logging into the testing service and proceed to upload their smart contract inputs. Upon submission, the system retrieves relevant exercise data, such as the Docker image ID, and records the submission in the database with an "*Inconclusive*" status. Utilizing the submitted smart contracts, a Docker container is then launched using the exercise's Docker image, running the tests against the submitted smart contracts, subsequently producing test results which are then returned to the students. After that, performance metrics such as execution time, gas consumption, and gas differences are derived from this output, and the submission details are updated in the database, with the status being set to either "*Success*" or "*Failure*" based on the test results.

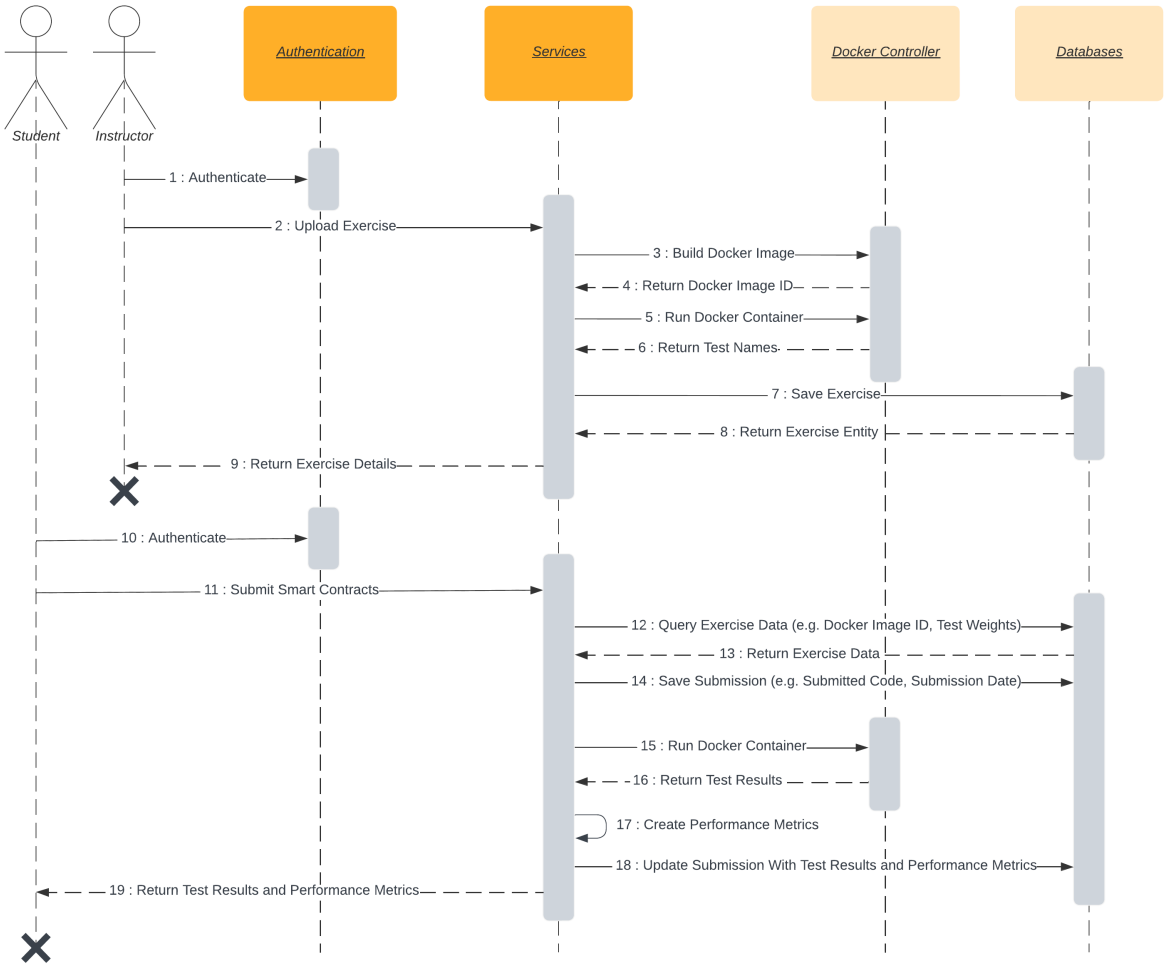

Figure 6.1.: High-level Sequence Diagram for Exercise Upload & Code Submission

Conclusively, students are presented with a comprehensive report on the testing results of the executions of their smart contracts.

6.3. Architecture

The entire testing service, as illustrated in Figure 6.2, is managed as a single Docker Compose project. This includes three services: Test Runner, Backend Services, and the RabbitMQ instance, in addition to the MongoDB instances, which are also Dockerized. These services are all initialized as Docker containers.

Docker Compose facilitates the composition of applications with numerous containers by describing the individual components and their relationships, easing the deployment and maintenance of multi-container applications, and ensuring dependencies between containers are met [107]. For instance, considering the cross-container dependencies within this testing

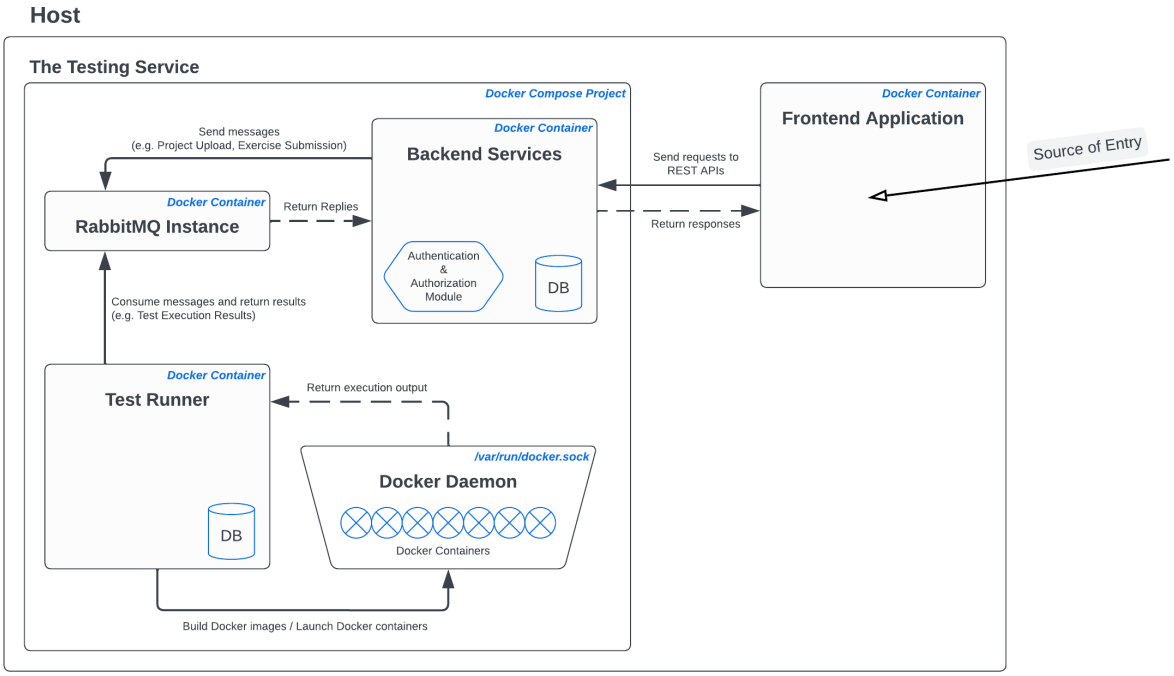

Figure 6.2.: The Architecture of the Testing Service

service, the Test Runner container requires the RabbitMQ container to be healthy before initiating, and the Backend Services container waits for the Test Runner container's readiness, ensuring accessibility to the Docker Daemon.

6.3.1. Test Runner

This is the primary service, written in TypeScript, which is responsible for smart contract testing and has a dedicated MongoDB database to store data related to Docker images and the histories of executed Docker containers. Within this Docker Compose setup, a three-node replica set is configured. A replica set is a collection of MongoDB servers that keep an identical data set. This setup enhances data redundancy and ensures automatic recovery in case of system outages [108]. It also boosts availability and supports database transactions, allowing developers to initiate database sessions and commit changes only when the operation is error-free. For efficiency, to be further detailed in subsequent sections, the Test Runner service steers clear of the Docker-in-Docker approach and instead uses the host's Docker daemon to build Docker images for projects and execute Docker containers for smart contract testing. This process is facilitated by Docker volume binding: the host's Docker daemon socket is linked to the container's Docker socket. Hence, when the Test Runner application attempts to engage with the Docker daemon, it communicates via the linked socket, allowing it to access the host's Docker daemon rather than a nested Docker daemon within the service's container.

6.3.2. Backend Services

This backend application, written also in TypeScript, offers REST APIs to users for interacting with the testing service. Similar to the Test Runner, it possesses its own replicated database, and it stores data related to uploaded projects, submissions, and their corresponding execution results, inclusive of uploaded files. Additionally, an integrated authentication and authorization module makes use of JWTs. These JWTs, composed of a header, payload, and signature, allow for user authentication without requiring repeated connections to the resource server or database [109]. User credentials are securely stored in the database with encrypted passwords.

6.3.3. RabbitMQ Instance (Message Queueing)

To ensure that the Test Runner and Backend Services operate in isolated environments, RabbitMQ, a message broker, is employed. This message broker enables message queueing, thereby isolating the two services and providing fine-grained control over the request flow from Backend Services to Test Runner. This restriction ensures that only a limited number of requests are processed at the same time. Further specifics of RabbitMQ will be discussed in the following sections.

RabbitMQ is integrated as a service within the Docker Compose project for simpler setup and increased security, supporting its building and execution as a Docker container. It runs within a designated Docker network, to which both the Test Runner and Backend Services are connected. This setup ensures the security for the Test Runner since it can only be accessed through this specific Docker network.

The RabbitMQ instance's operational flow is as follows: The Test Runner consumes messages from RabbitMQ and returns results once processed. These results include both project creations and test executions depending on the type of message sent. On the other hand, Backend Services interact with the Test Runner entirely by sending messages to RabbitMQ. These messages can be about project creation and exercise submission. Subsequently, Backend Services subscribe to a specific reply channel, to which the Test Runner sends response messages.

6.3.4. Frontend Application

This client application, developed using the Vue.js [110] framework, acts as the source of entry for users to access this testing service and showcases that the service functions as expected. This application only communicates with the Backend Services, sending REST API requests and receiving replies. It is set up as a distinct Docker container rather than being deployed within the same Docker Compose project as the testing service. This design decision ensures that the testing service can be used as a library in other projects without the need for a redundant user interface application.

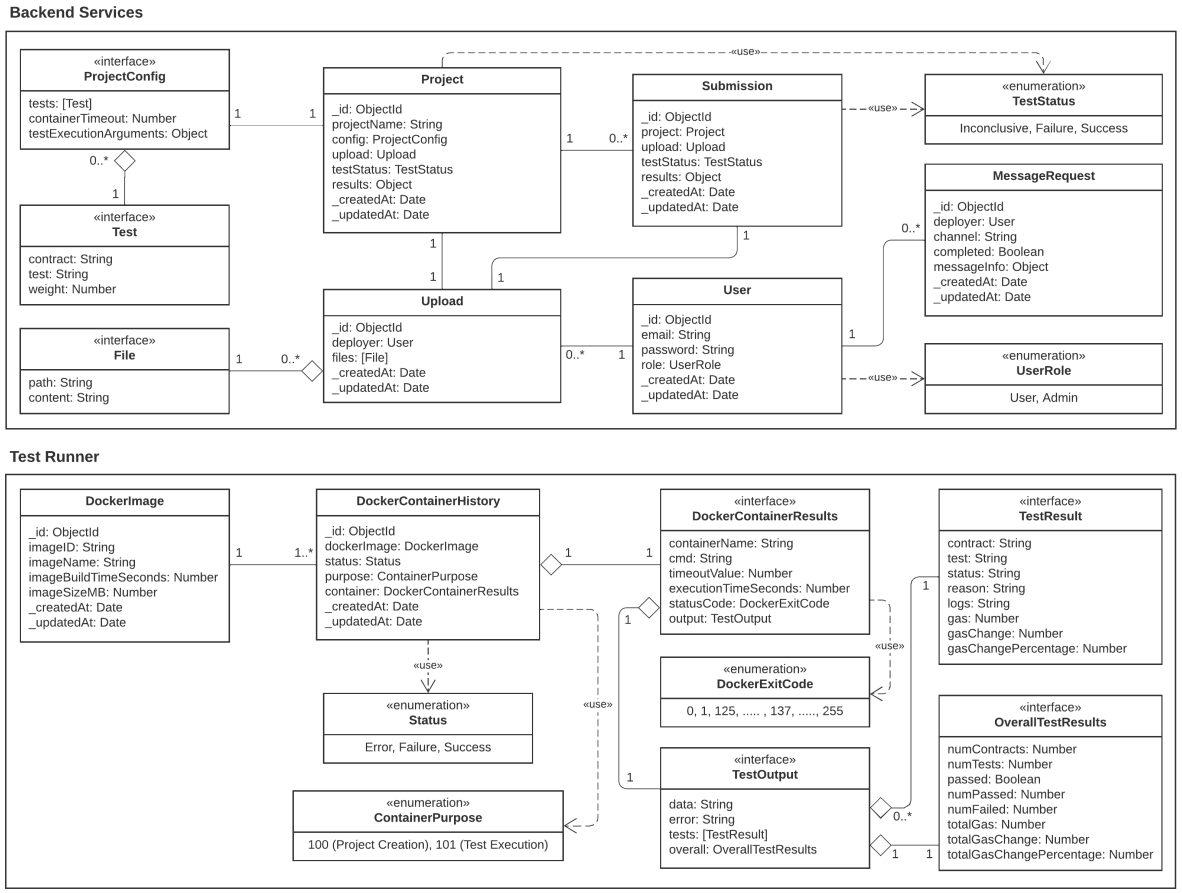

Figure 6.3.: Data Model of the Testing Service

6.4. Implementation Details

This section primarily outlines the data model and the implementation details of the testing service. Additionally, it covers the establishment of inter-service communication, network configuration, service isolation, and secrets management.

6.4.1. Database Selection and Data Model

It is critical to design an optimal data model that aligns well with the functional requirements. The database choice, be it a relational database like PostgreSQL or a non-relational one such as MongoDB, has a substantial impact on the overall performance of the testing service. It crucial that the database chosen is compatible with the data model, ensuring optimal efficiency during read and write operations.

MongoDB, a NoSQL database, was chosen for this testing service. Given that this application's data model was unlikely to contain many relations, a non-relational database was considered more suitable. In contrast to table-based storage structures, MongoDB is schema-

less and efficient at managing unstructured data [111]. Many fields in this service contain JSON objects, and model changes are likely in the future due to changing requirements or advancements made to the test runner framework used. Because of its efficiency for storing unstructured data, MongoDB was an obvious choice. Scalability was another consideration in this decision. While relational databases struggle with scalability at times, frequently relying on vertical scalability via the addition of more hardware, non-relational databases excel at horizontal scalability [111]. This means that rather than simply increasing the power of a current node, the service can grow by adding more nodes and sharing data across those multiple nodes.

The data model used by the Test Runner is demonstrated in Figure 6.3. The Test Runner's data storage is primarily focused on storing the current state of the Docker daemon and the execution history of Docker containers. Such a configuration facilitates the monitoring and representation of the current state of the Docker daemon. At the same time, it keeps historical data, allowing for retrospection of all Docker container executions and their results. *DockerImage* and *DockerContainerHistory* are the main collections. The former records information about the Docker images and their attributes, while the latter records information on the execution of the launched Docker containers. Given that each Docker image is validated by subsequent Docker container execution to ensure it is error-free, each *DockerImage* is linked to at least one *DockerContainerHistory*. Additionally, nested schemas were incorporated for the execution output, stored in a schema named *DockerContainerResults*. This design choice was made to ensure structured management of the object storing the execution results.

Similarly, Figure 6.3 illustrates the data model used by the Backend Services, which is directly linked to the specified realization of the functional requirements given in subsection 6.1.2. For authentication, the *User* collection stores email and encrypted passwords, and for role-based access management, it stores user roles. The *Project* collection allows instructors to upload and view projects. It stores project configurations and execution results. The *Submission* collection stores the students' smart contract input submissions. Each project might be linked with zero or more submissions, tying a *Submission* to a certain *Project*. Both of these collections have a *results* field for the execution output and a *status* enumeration that represents test execution results. Furthermore, the *Upload* collection facilitates the downloading of files uploaded for both projects and submissions. It keeps the paths and contents of the uploaded files, alongside the uploading user's identity, the deployer, allowing an association where a user might be linked with zero or more uploads. Finally, the *MessageRequest* collection, which was designed for recording messages sent to the RabbitMQ instance, stores several message properties and contents. This collection maintains a Many-To-One relationship with the *User* collection.

6.4.2. Test Runner

The Test Runner stands as the central pillar of the testing service, designed and implemented to execute student smart contracts against tests uploaded by instructors. This service is encapsulated within a Docker container and integrated into the Docker Compose project. Developed in TypeScript, its primary task is to construct Docker images using the instructor-

provided projects and then run these images as containers to test the smart contract inputs of the students. It makes use of Dockerode [112] as the Docker Remote API module to communicate with the Docker daemon. The service is developed using the Node.js web application framework, Express.js [113], which is used primarily to run the service as a server on a designated port. Since this service is exclusively accessed through RabbitMQ, detailed further in subsequent sections, the REST API functionalities of Express.js are redundant. However, the framework still offers the potential to create REST endpoints that could act as entry points. The following sections dive deep into the main functionalities of this service.

Healthcheck API

The service provides a simple REST endpoint for verifying connectivity to the Docker daemon, essentially determining whether the Docker daemon is reachable. This health check is critical for the entire testing service in the Docker Compose project. Services that depend on this service, for example, will wait for it to be "*healthy*" before starting.

Building Docker Images from Uploaded Projects

Projects, along with their names, can be uploaded to the service, from which a Docker image is created. The uploaded zip file is initially processed and stored in a temporary folder within the Docker container's file system. This zip file should unpack to a directory that contains specific folders and files. Thus, the zip file is checked to ensure that it includes the necessary folders, such as the `test` folder (containing the test contracts that will be used to test student submissions) and the `src` folder (containing the instructor-developed smart contracts that pass all tests). Additionally, the zip file is validated to see if it contains the essential files: *remappings.txt*, which lists the paths to project libraries, and *.gitmodules*, used for defining the libraries for installation, a method employed by Foundry as previously discussed in section 5.3.

Upon validation, the contents of the zip file are moved to a uniquely named temporary folder within the container's file system. Following that, certain generic project files, namely *Dockerfile*[1], *foundry.toml*, and *install_libraries.sh*, are copied over. The *Dockerfile* is a file that, like a *Makefile*, is used to construct images through a series of instructions [114]. The *foundry.toml* [115] serves as a configuration file for the Foundry project and is maintained uniformly across different projects. Lastly, the *install_libraries.sh* script is developed to be employed during the Docker image building phase to extract dependencies from *.gitmodules* file and install them in the container.

After unzipping the uploaded zip file and writing it to a temporary folder, the contents of the temporary project folder are used to build a Docker image using the *buildImage* function from the *Dockerode* module. The name of this image corresponds to the provided project name. The *Dockerode* module also provides details about the built Docker image, including its ID and size in MB. Additionally, the time taken to build the image is calculated in seconds. Once the Docker image is successfully built, the project is tested by executing the image as a

[1]The specific *Dockerfile* used for building Docker images from uploaded projects is shown in Figure C.1.

container using the `src` folder, returning test execution results. This test step ensures that the uploaded smart contract can pass all the tests. Detailed discussions on Docker image execution and the derivation of results will be presented in the following section.

If the Docker container runs successfully (exiting with code 0) and the smart contracts pass all the tests, the execution results are processed. Subsequently, the temporary folder is deleted and both the image and container execution details are stored in the database. Conversely, if any errors arise during these steps, the service initiates pruning, removing the Docker image and any containers associated with image construction.

It is crucial to highlight that during the build process, *Gas Snapshots* are produced. As elaborated in section 5.5, these capture the gas usage of the smart contracts provided by the instructors and are later utilized to determine gas consumption differences when testing the smart contract inputs of the students.

Executing Docker Images as Containers for Testing Uploaded Smart Contracts

The procedure for uploading smart contracts for testing closely resembles the initial steps of project upload previously outlined – the uploaded zip file is unzipped, and its contents are moved to a temporary directory. However, a noteworthy difference is that this zip file only contains smart contracts, eliminating the need for any requisite file verification. In addition, when uploading smart contracts, two additional parameters accompany the zip file: the project name associated with the uploaded smart contracts and the project configuration, which is optional and includes container timeout and test execution arguments, e.g., gas limit.

Initially, the database is queried using the specified project name. If no image can be found, the service returns an HTTP "*404 Not Found*" status code. Otherwise, the uploaded zip file is unzipped to a temporary folder containing smart contracts, the contents of which will be used for running the Docker image as a container. Following that, a container is created using the *createContainer* function from the *Dockerode* module. The temporary folder containing the smart contracts is then copied into this container.

Once created, the container (*Dockerode.Container*) is started using the *start* function. Particularly, the "*snapshot difference*" command from Foundry's Forge is used instead of the default command in the *Dockerfile* (as shown in Figure C.1). When executed with the specified test arguments, this command (`$ forge snapshot --silent -vv --allow-failure --json --diff <snapshot_file>`) outputs extensive information about the test results, keeps running the tests even if they fail, and formats the results as JSON. Furthermore, this command runs tests, returns test execution results, and compares gas consumption against the instructor-developed smart contracts. Should there be any additional test execution arguments, like gas or memory limits, they are appended to this command.

Following the container's startup, another promise concurrently starts, resolving after a designated timeout. If no specific container timeout is specified, a default value of *30 seconds* is used. These two promises race against each other in parallel. If the container's execution does not finish before the timeout promise resolves, a kill signal is sent to the container, which then gets removed, and a timeout response is returned. This mechanism acts as a safeguard, ensuring that inefficient smart contracts or those caught in an infinite loop have no adverse

effect on the testing service, regardless of whether they were submitted to the service on purpose or by accident.

Upon completion of the container's execution, data such as the container name, Docker exit code[2], and its logs are collected. The container logs include either the test results or execution errors, depending on the status of the test execution. Once the execution details are populated into an object, the container is deleted. In the event of a successful execution, the logs are processed to yield structured test execution results. Subsequently, an object containing the execution details and the processed results is saved in the database. It is tagged with a "*Success*" status if all tests pass, and a "*Failure*" status if they do not. Conversely, in the case of an unsuccessful execution, an object with the error, labeled with a "*Failure*" status, is saved in the database. Lastly, the temporary directory containing the uploaded smart contracts is deleted.

The following section contains a detailed discussion on extracting test execution results from container logs.

Extracting Test Execution Results from Container Logs

The container logs are returned in the form of a *Buffer* object when the Docker container is executed. This is then converted into a string, which represents the console output when the test execution command is executed within the container. To utilize this output effectively, essential test results are extracted and saved within an object. If Docker exits with a non-zero code, indicating an error during test execution, the container logs, which contain the error messages, are saved directly to the database with the status "*Failure*". On the other hand, if Docker exits with a success code of 0, it indicates that there were no execution errors, indicating that the test results are to be processed further. However, a success code does not necessarily indicate that all the tests have passed. This is decided after the container logs are processed and a structured object containing the test results is derived from them.

The container logs are divided into two parts. The initial part contains a JSON string with the results of the test execution. This encompasses data like the tests that passed or failed, the gas consumed for each test, and any relevant logs. The latter part presents the actual gas snapshot output, specifically highlighting gas differences when compared to the instructor-developed solution for each test. Initially, any color codes and non-ASCII characters in the Foundry's test output are removed to facilitate processing. Subsequently, the JSON part is parsed to remove any unnecessary information. After successfully parsing the JSON part, additional data points such as the number of successful tests, failed tests, and total gas consumption are computed. Furthermore, the overall test outcome is determined as either "*Success*" or "*Failure*", depending on the success of all tests. Afterward, the gas snapshot output in the second part is processed by utilizing regular expressions; the gas differences and their respective percentage values are extracted for each test. Finally, all the extracted data is merged into a single structured object that represents the test execution results and is ready to be stored in the database or returned to the caller.

[2]For more information on Docker exit codes, refer to Table C.1.

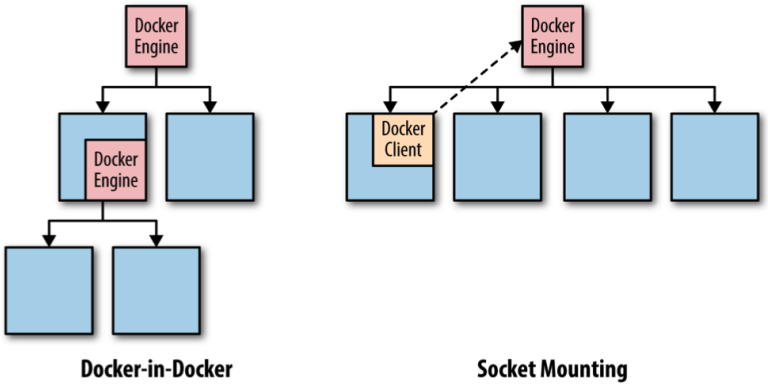

Figure 6.4.: Docker-In-Docker versus socket mounting (*Source:* [116])

Alternative to Using Docker-in-Docker

Running Docker within another Docker container is often referred to as Docker-in-Docker, or DinD [116]. As briefly described in the previous sections, purely for performance reasons, Test Runner uses the host machine's Docker daemon to build and run images, as opposed to Docker-in-Docker, which would instantiate a Docker daemon within the Test Runner container itself. This is realized via socket mounting, where the Docker socket[3] from the host is mounted into the container, thereby allowing the Test Runner to generate "*sibling*" containers on the host machine. Both of these approaches are illustrated in Figure 6.4.

Employing the host's Docker daemon offers optimal performance, mainly because the allocated resources come directly from the host rather than a Docker container. On the other hand, using Docker-in-Docker allocates a volume to the `/var/lib/docker` directory, and neglecting to remove this volume can quickly consume storage space when the container is deleted [116]. Moreover, if the Test Runner were to unexpectedly terminate or fail, all data associated with previously built images would be lost with the Docker-in-Docker setup, which is not ideal. While such data can be preserved using methods like mounting the build cache from the host, this approach can often have negative impacts [116]. Regardless, one of the advantages of the Docker-in-Docker approach is its improved isolation, which can prove invaluable in some circumstances [116].

In conclusion, considering the benefits and drawbacks of socket mounting versus the Docker-in-Docker approach, leveraging the host's Docker daemon via socket mounting is the better choice due to its streamlined configuration, easier maintenance, and, most importantly, increased efficiency. Therefore, socket mounting is employed in the Docker configuration for the Test Runner [116].

[3]The Docker IPC socket, commonly found at `/var/run/docker.sock`, facilitates communication between the client and daemon, though it can also be accessed through TCP or systemd-style sockets [116]. In this project, the IPC socket is accessed through the file `/var/run/docker.sock`.

6.4.3. Backend Services

The Backend Services function as the gateway to the testing service, offering its users with specific REST endpoints to facilitate communication with the whole testing service. This service is also Dockerized and integrated into the Docker Compose project. Developed also in TypeScript, its key duties are communicating with the Test Runner through RabbitMQ for smart contract testing, storing the data related to the educational use case in its database, and ensuring that users gain appropriate access to the testing service based on authentication and authorization levels. Like the Test Runner, it is built with the Express.js framework, but this time it not only runs the service as a server but also exposes certain REST endpoints (refer to Table D.1) to its users for communicating with the service. The following sections explore how the functional requirements of the testing service are implemented.

Authentication and Authorization

One widely used technique for implementing authentication and authorization for the end-user is the use of JWTs, which offers a robust and structured security mechanism that reduces server overheads [117]. All the REST endpoints of the Backend Services are secured using this technique, ensuring only authenticated users have API access. Moreover, with JWTs, role-based access to specific APIs is facilitated through implemented middleware functions. This not only increases the security of the service, but it also adheres to best practises in backend service development.

Users can sign up for this service with their email address and a secure password; however, for added security, only pre-registered admins can register other admins. Upon registration, user information is saved in the database under the *User* collection, with the password hashed before storage. Password hashing, which involves converting plaintext using algorithms such as bcrypt or SHA, is critical in ensuring that compromised passwords remain unreadable to potential adversaries [118]. This method ensures that hashed passwords cannot be decrypted. To log in, users simply enter their registered email address and password; the service validates the entered password by comparing it to the hashed version stored in the database.

Upon successful authentication, the service generates a JWT with user details, excluding the password, as its payload. This JWT, signed with a specific secret, is set to expire after *one week* by default within this service; though, this is adjustable. The method of securely signing the JWT ensures the integrity of the token, making certain that it cannot be tampered with in the absence of the corresponding secret. Additionally, the generated JWT is saved in the response cookie, so that that any subsequent requests following the authentication contain the JWT in the cookie.

Uploading Projects for Smart Contract Testing

Uploading a new project or updating an existing one involves submitting a zip folder with the necessary project files, accompanied by the project name and its configuration. If there is an existing project with the same name in the database, it is retrieved from the *Project* collection; otherwise, a new project is created. The project configuration object is first added

to the *Project* document, and then the uploaded zip file is saved in the *Upload* collection. This connects the newly created or updated *Upload* document to the corresponding *Project* document. These operations make use of database sessions and transactions. If any errors arise, the transaction is immediately aborted. If everything goes smoothly, the transaction is committed and the changes are saved in the database. Once a project is either created or updated, its test status attribute is set to "*Inconclusive*", indicating that the project is being uploaded to the Test Runner and the results remain pending.

Upon successful storage of the project in the database, a new message related to the project upload is sent to a RabbitMQ fan-out exchange, ensuring all the consumers receive this message. Further details of RabbitMQ and message queuing will be elaborated on in the upcoming sections. Following the successful delivery of the message, the project response is immediately returned from the endpoint. When the consumers, namely the Test Runner instances, process this message and either create a new Docker image or modify an existing one using the submitted zip file, a response message, encapsulating the test execution results for the project upload, is sent to the reply queue. This reply queue was specified in the project upload message when it was sent. The Backend Services, after dispatching the original message, waits for messages in this reply queue. Upon receipt of the response, the stored *Project* document is updated with the test execution results provided by the Test Runner.

Uploading Submissions for Smart Contract Testing

With a few exceptions, the process for uploading a new submission is very similar to that of uploading a project. Particularly, a submission does not include a configuration, and it is not allowed to update existing submissions; only new ones can be created. Just like the project upload, the uploaded zip folder for the submission is stored in the *Upload* collection and linked to the newly created *Submission* document. The test status attribute is set to "*Inconclusive*", and a message is sent to the Test Runner via message queueing. However, in this case, instead of using a RabbitMQ fan-out exchange, a simple RabbitMQ queue is utilized since only one Test Runner instance will handle the message. The Test Runner then executes tests against the uploaded smart contracts as previously described. After testing, it communicates the results back to the Backend Services through a reply queue. The *Submission* document is then updated with the results, mirroring the final steps of the project upload.

Listing Projects and Submissions

Projects and submissions are retrieved from the database based on their respective collections: *Project* and *Submission*. The project details include attributes such as the project name, project configuration, test status, and the deployer. For submissions, the returned details are the associated project, test status, and the deployer. Additionally, the test execution results for both these collections are included in the returned object.

The returned response is determined by the role of the authenticated user, which can be either *Admin* or *User*, through the use of some implemented middleware functions. While both roles have access to all projects, only admins are granted the ability to view all submissions.

Users, on the other hand, can only access their own submissions. This ensures that, for instance, students in the core use scenario cannot access or view their peers' submissions, thereby increasing the security of the whole testing service.

Editing and Removing Projects

A project provides two main modification options: editing and removal. Both functions are only available to users with the *Admin* role. A project's editing is divided into two distinct processes: re-uploading and editing the configuration. When re-uploading, this action triggers the creation of a new Docker image from the uploaded zip file. On the other hand, project configuration can be edited separately. While it is possible to change the configuration by re-uploading, there is also a specialized API just for configuration updates. With this API, the configuration object must be included in the request body. The project is then retrieved from the *Project* collection. If the retrieval fails, an HTTP "*404 Not Found*" status code is returned. Otherwise, if the project is successfully retrieved, the project configuration is updated with the provided configuration object.

The removal procedure, on the other hand, slightly requires more steps. Initially, the service queries the database for the project. Once retrieved, any uploaded files associated with the project are removed from the *Upload* collection. After successfully removing these files, the project is removed from the *Project* collection. This procedure also employs database sessions and transactions. Following these operations, since each project is linked to a specific Docker image, the Test Runner must be informed of any project deletions. This is accomplished by dispatching a deletion message to a RabbitMQ fan-out exchange, just like a project upload message. The Test Runner, upon receiving this message, proceeds to delete the Docker image. It must be noted that even if this deletion fails, the image will be deleted later during the subsequent service pruning performed after each Docker operation in the Test Runner.

Downloading Uploaded Files

In earlier sections, it was mentioned how both projects and submissions are uploaded in the form of zip files. Before the projects and submissions are stored in the database, the contents of the uploaded zip files are unzipped, and a new record is created in the *Upload* collection. This record is then linked to its respective project or submission.

To download the uploaded files, the associated project or submission is simply fetched from the database, along with its corresponding uploaded files. As discussed in previous sections, the uploaded files are stored as a structured list that includes path and content of each file. Using this data, the contents of the uploaded files are converted into *Buffer* objects and added to a zip file according to their specific paths. This assembled zip file is then converted back to a single *Buffer* and returned as a downloadable response.

For all zip-related operations, the adm-zip [119] library is employed.

Utilizing the Message Requests

Message requests serve a simple purpose: they log information about messages sent to RabbitMQ. This includes details such as completion status, time taken, message content, and so on. A new message request is created in the database after a message is sent to RabbitMQ. When a message is received on one of the reply channels, its corresponding record in the database is updated with the response message. Retrieving these message requests is straightforward; the necessary records are fetched from the database. It is worth noting that similar to the access controls applied to submissions, role-based authorization is also applied for message requests. While admins can view all message requests, users are restricted to accessing only the message requests created by them.

6.4.4. Message Queueing and Inter-Service Communication

Message queuing enables applications to communicate with one another by transporting data messages in sequentially processed queues [54]. In this testing service, such communication between the Test Runner and Backend Services is managed using message queuing. Advanced Message Queuing Protocol (AMQP) is a protocol that employs high-level message queues to ensure reliable data transfer between applications [120]. RabbitMQ, the message broker or the messaging system utilized in this testing service, supports the AMQP protocol among others [121].

Message queueing was selected as the communication method between the Test Runner and Backend Services due to its ability to provide isolation, stability, and efficiency. Instead of using REST APIs, the Backend Services, acting as the message producer, dispatches messages to the Test Runner, the message consumer, without requiring knowledge of the Test Runner's specifics like host, port, or endpoints. This level of isolation simplifies the development of both services, making sure that they remain independent. Furthermore, message queueing also improves system stability and efficiency by setting constraints on concurrent message processing. For instance, in the core use case where a large number of students is expected to flood the system with requests during peak usage hours, uncontrolled simultaneous processing could cause inefficiencies, bottlenecks, and slow down the system due to suboptimal use of hardware resources and threads, potentially rendering the system unavailable. By enforcing these limits, such problems can be avoided. In addition, RabbitMQ enhances scalability through its ability to group multiple RabbitMQ instances across different nodes to form a RabbitMQ cluster, facilitating load distribution based on node status [122]. Hence, a RabbitMQ instance is integrated into the Docker Compose project for the testing service. This configuration provides communication, with the Backend Services sending messages and the Test Runner consuming them.

As previously mentioned, the testing service is designed to allow horizontal scalability. Although the details of its horizontal scaling will be discussed further in subsequent sections, it is important to note that in the future, many Test Runner instances may be distributed across various nodes. Hence, for clarity in this section, it is assumed that there are multiple Test Runners.

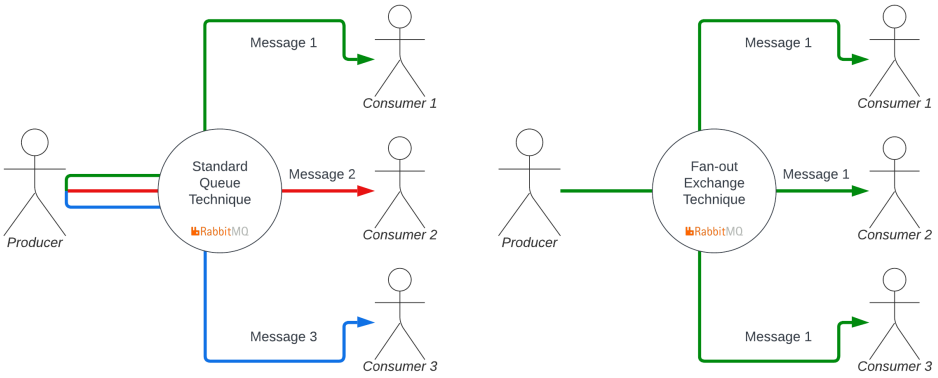

Figure 6.5.: RabbitMQ Techniques - Standard Queue (left) and Fan-out Exchange (right)

A compact library is developed to allow RabbitMQ communication between two services. This library contains generic methods for sending, consuming, and awaiting replies from designated reply queues. In the testing service, two types of communication techniques are employed: a standard queue and a fan-out exchange, both of which are demonstrated in Figure 6.5.

The standard queue technique represents the typical RabbitMQ message queuing mechanism. Here, the Backend Services, acting as the message producer, sends messages to specific queues, which are then received and processed by the Test Runner instances, the consumers. Using this approach, only one Test Runner instance handles a message, selected based on its current workload and capacity. For instance, consider a scenario where a consumer can only process 10 messages simultaneously and there are two consumers: A and B. If consumer A is already handling 10 messages, the next message will be directed to consumer B due to A's current workload. This method is used when sending a submission from the Backend Services to the Test Runner, where it is sufficient that only a single Test Runner instance processes the submission and executes the tests against the uploaded smart contracts.

In a fan-out exchange, on the other hand, a dispatched message from the producer is received and processed by all consumers waiting messages on that exchange [121]. This method is used for messages related to the projects. When a project is uploaded or deleted, it is vital that all Test Runner instances update the state of the Docker daemon they are responsible for, either by building a Docker image, updating an existing one, or removing it.

Whenever a message is sent to either a queue or a fan-out exchange, a new queue is created to receive reply messages. The name of this queue is included with the dispatched message, enabling the consumer(s) processing the message to send their results back to the producer. This mechanism ensures that the Backend Services retrieves the test execution results from the Test Runner after sending the uploaded projects or submissions.

Crucially, sending the message and not waiting for the response helps prevent blocking the message producer service. This is of utmost important as there is no certainty regarding how busy the queues are or how fast the messages will be processed. Moreover, the messages in the queues are preserved even in the event of system failures. For instance, consider a

scenario in the core use case where students may submit their correct smart contracts shortly before a deadline. In such a case, students should still receive passing results, even if the testing service momentarily fails and restarts. Finally, it is worth mentioning that by default, a consumer can process up to *10 messages* at a time, a threshold that can be modified if necessary.

6.4.5. Service Network Configuration and Isolation

When a Docker container runs, it operates in isolation from its host. As a result, any application running within this container on a given port is not directly accessible by the host machine. To facilitate access, one can either bind a specific port on the host to the one used by the application within the container, or deploy the container within a Docker bridge network. In the latter scenario, containers connected to the same bridge network can communicate with one another, enabling access to the application through its specific port. This is because a bridge network employs a software bridge mechanism, allowing communication between containers connected to that particular bridge network, while isolating them from containers not connected to it [123].

Within the Docker Compose project for the testing service, multiple bridge networks have been defined to isolate services from each other and external interactions. Only services that need to communicate with one another are connected to the same bridge network, adhering to best practices. Initially, each service and its corresponding MongoDB replicas are connected to their own dedicated bridge network that no other service has access to. This configuration ensures that an application's database is accessible only by that specific application. Moreover, there is a dedicated bridge network to which the RabbitMQ instance and the two services are connected, ensuring the Backend Services can interact with the Test Runner exclusively via RabbitMQ. Additionally, there is another bridge network that only the Backend Services is connected to, allowing for potential access by client applications, such as a frontend application. In summary, these Docker bridge networks are intentionally set up to strengthen service security against external access and to ensure services operate within a secure network environment.

6.4.6. Secret Management

The secrets for the testing service, such as the MongoDB URI and the JWT secret, must be protected from unauthorized access. A common method for storing secrets is through environment variables. However, this approach carries risks: secrets stored as environment variables may be mistakenly exposed, and the contents of a *.env* file may be revealed during debugging sessions. To address this concern, Docker provides a method called Docker secrets for securely transmitting secrets to containers. This not only involves encrypting the secrets but also ensures protection against unintentional leaks [124]. Hence, Docker secrets were chosen as the preferable technique for handling and storing secrets for the testing service. These secrets, which are defined within the Docker Compose project, are read from a particular folder in the file path. This technique ensures that secrets are safely stored.

6.4.7. Frontend Application

The Dockerized frontend application, which utilizes the Backend Services for its backend operations, is built using the Vue.js framework. Although this application was primarily developed to showcase the functionalities of the testing service, it still adheres to best practices in web development. To achieve more flexible code organization, enhanced logic reuse, and a reduced production bundle size, the Composition API, a recent addition to Vue, was adopted [110]. Several reusable components have been implemented, ensuring a user-friendly interaction with the testing service and simplifying maintainability for future developers. Ultimately, a single-page web application was created using Vuetify [125], a UI framework built on top of Vue.js.

Given that this frontend application is supplementary to the testing service library, the details are avoided, leaving simply a brief summary. To view the final appearance of the testing service with the frontend application, please see the screenshots in Appendix E.

6.5. Security and Stability

The testing service was built using Foundry, a test runner framework selected from a comparative analysis of various frameworks. These frameworks are typically used for more than just testing; they are also used to deploy smart contracts. A smart contract project is often tested (if tests are provided), and if all tests pass, the project is deployed. However, the testing service designed in this work is exclusively dependent on the results of test executions, which are generated after running the smart contracts against the tests in a Docker container.

Given that these test runner frameworks are used in this testing service in a manner they were not originally designed for, the security and stability of the service must be considered. This is especially vital in an educational context where instructors might make use of this testing service to grade students. It becomes imperative to maintain the service's integrity at all times. Thus, the service must gracefully handle any errors or crashes that may occur during contract executions to ensure its continuous availability, especially when students want to submit their smart contract assignments. In addition, safeguards against potential system overloads must be in place. Such overloads might be accidental, such as a sudden surge of submissions during peak hours, or intentional, where the service could be bombarded with a flood of requests to render it unavailable. The next subsections discuss how the testing service was designed to address these considerations.

Regardless of these considerations, it is important to note that the testing service, deployed as a Docker Compose project, leverages Docker's error-handling capabilities. Specifically, if a service within the testing service crashes, the corresponding container running the service restarts due to the *"restart"* attribute being set to *"unless-stopped"* in the Docker Compose configuration file. Hence, potential crashes in the service do not impact its availability.

6.5.1. Handling Errors and Crashes in Contract Executions

The Test Runner, as previously stated, utilizes Docker containers for smart contract testing. For each smart contract input submitted to the system, a new Docker container is started. Since these containers operate within their isolated environments, any crash during the contract execution will only terminate the specific container running that smart contract, without affecting the service, the Test Runner, that spawns it. In addition, it is essential to convey execution errors to the user, especially students in the core use case, to inform them of potential mistakes in their smart contract inputs. This feedback mechanism allows them to learn, correct their mistakes, and improve their smart contracts. Leveraging certain command-line arguments from Foundry ensures that even in the event of an error, the contract execution is not interrupted and the logs are returned back to the Test Runner upon successful container execution. In summary, errors or crashes during contract executions have no negative impact on the testing service, since every execution is encapsulated within its dedicated Docker container.

6.5.2. Mitigating Accidental or Intentional System Overloads

System overloads can occur either by accident or on purpose. Regardless of the cause, the integrity of the service must be maintained constantly. Initially, the Backend Services, which serves as the primary gateway to the testing service, has a set of restrictions on the number of requests originating from the same IP address. While this measure acts as an additional safeguard, it is possible to improve the security even more by enforcing this limit on authenticated users rather than just IP addresses. Such measures protect the service against intentional system overloads, in which potential attackers may send an overwhelming number of requests to the service, leading to their temporary access suspension.

Moreover, the Test Runner, which is responsible for smart contract testing, gets its inputs from RabbitMQ, the message broker for this service that facilitates message queuing. The amount of messages that the Test Runner can process at the same time is determined by a predefined threshold. In situations where a system overload is not intentional and arises due to a high volume of users, this will not destabilize the service but might extend wait times for users. If these waiting times continually skew higher, it is not necessarily a security or stability concern, but rather an indication that scaling is required, which will be discussed further in the upcoming section.

On the other hand, the service may occasionally receive inefficient smart contract submissions, such as a contract with a function that gets stuck in a loop or takes too much time to return. To counteract such scenarios, various constraints have been set. As previously mentioned, projects are submitted to the system along with container timeout values. This timeout parameter is used to halt smart contract executions by terminating their associated containers. This ensures that the service stays stable even when fed with potentially disruptive smart contract inputs. Instructors may also apply additional limits, such as gas or memory limits, which, if exceeded, result in an immediate termination of the execution of the smart contracts.

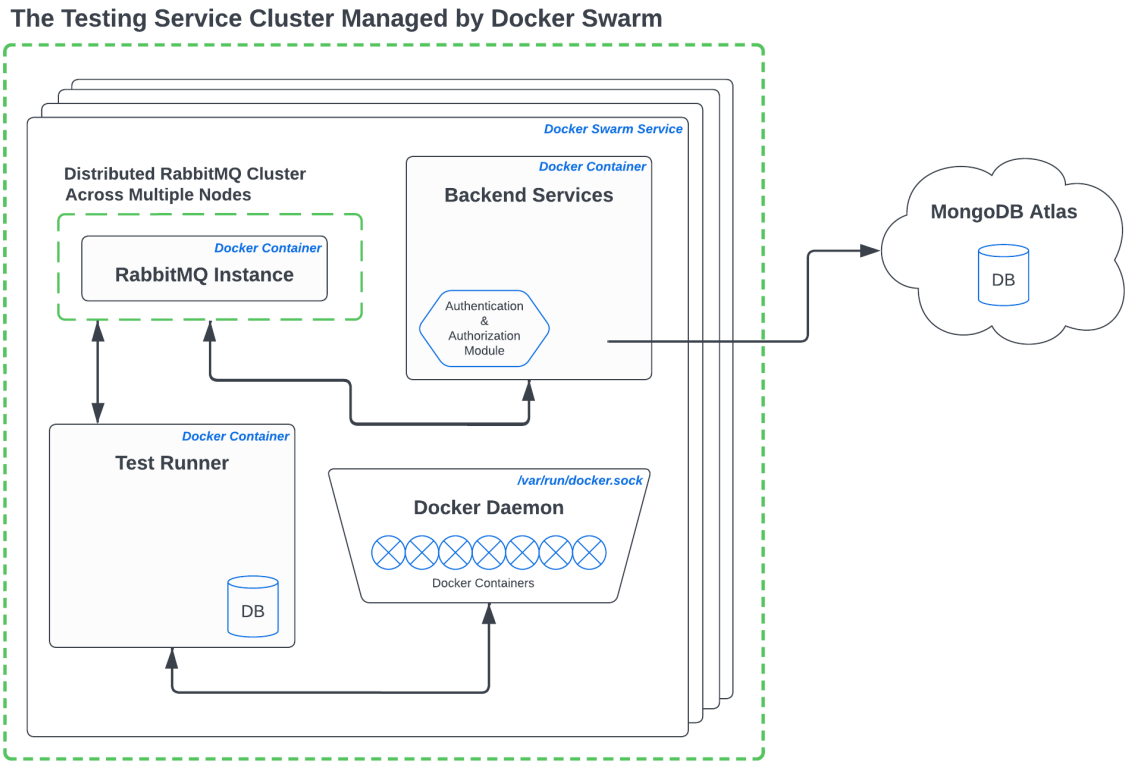

Figure 6.6.: Horizontal Scalability of the Testing Service

To conclude, the testing service is protected against both accidental and intentional system overloads. It employs a variety of safeguards, including limiting users from sending an excessive amount of concurrent requests, assuring that the service always stays available through message queuing, and rapidly terminating the execution of inefficient smart contracts that could otherwise harm the entire testing service.

6.6. Scalability

The testing service was implemented to efficiently handle a significant volume of concurrent requests, specifically a simultaneous load of approximately *50* students on a single machine, particularly during peak times such as the hours leading up to deadlines. Anticipating future growth, especially with students from the BBSE course at TUM, which typically sees about *1000* registrations each semester, scalability emerges as a primary concern. Therefore, the testing service was designed from the start with horizontal scalability in mind. This decision is based on the limitations of vertical scaling as well as the observed performance improvements of containerized applications, such as the testing service, when scaled horizontally across distributed systems [126]. The design for the testing service's horizontal scalability is illustrated in Figure 6.6.

6.6.1. Container Orchestration Tool: Kubernetes or Docker Swarm?

The initial consideration was the technology selection for system scalability, with the options being two container orchestration tools: Kubernetes [127] and Docker Swarm. At first glance, Docker Swarm stands out for its simplicity and user-friendliness, whereas Kubernetes, although more complex, delivers advanced features such as self-healing and automated scaling as standard [128]. While Kubernetes is generally considered superior and more effective than Docker Swarm, its installation process is significantly more challenging and its configuration demands more time [129]. Docker Swarm, on the other hand, enables the seamless deployment of existing Docker Compose projects as "services" within the swarm. It necessitates some modifications to the existing Docker Compose configuration and requires images to be built and deployed to Docker Hub, a cloud-based registry for Docker images. Yet, its simplicity aligns nicely with the scope and requirements of this testing service. Having already packaged everything within a single Docker Compose project, Docker Swarm seems a more appropriate choice than Kubernetes for this project. It also offers a built-in network for facilitating load balancing among services [70]. Additionally, as Beltre et al. (2019) observed, Kubernetes and Docker Swarm offer comparable performance, even for HPC workloads [130], where performance is a decisive factor. Given Docker Swarm's performance, which is similar to that of Kubernetes, and its greater simplicity, it was chosen as the container orchestration tool for this testing service. The service is ready for deployment to the swarm with the addition of new nodes to the cluster. The current Docker Compose configuration is geared for such scalability, needing only minor adjustments, like deploying each individual service to Docker Hub. The complete system configuration, post-scaling with multiple nodes, is demonstrated in Figure 6.6.

6.6.2. Database Considerations

In a single-node architecture, the Test Runner and Backend Services of the testing service have their own dedicated databases. However, as the testing service grows horizontally, adaptations to the database become unavoidable. As highlighted earlier, the testing service was designed with horizontal scalability in mind. Consequently, this design decision naturally influenced the data model and the architecture of each service within this testing service.

The Test Runner possesses its own database, storing the state of its associated Docker daemon as well as a history of past executions. This makes sure that there is no need for modifications when the service is horizontally scaled across multiple nodes; each Test Runner instance can still maintain its distinct database, operating independently.

On the other hand, the database of the Backend Services records information on users, projects, submissions, and uploads. It is vital that this data is accessible to every Backend Services instance across the entire cluster. There are two main approaches to realize this. The first method involves keeping the database within the Backend Services, where the database of each instance must be deployed within the same replica set to maintain consistent state and have the same data at all times. Additionally, implementing MongoDB sharding can help distribute data throughout nodes in the cluster, with each node handling a data subset

[131]. While this approach is feasible, it requires a complex configuration and is challenging to manage. This complexity makes the second option, using a cloud database provider, more appealing. MongoDB Atlas [132] stands out as such a provider, offering a high-performance replicated MongoDB cluster on the cloud, with sharding capabilities. Given its user-friendly setup and robust features, it was incorporated for this project. Besides, even the free tier offers sufficient load balancing for the initial demands of this testing service. Using this cloud database allows the instances of Backend Services across nodes to access the same shared database. As illustrated in Figure 6.6, the Backend Services no longer operates its dedicated database but instead leverages the cloud database supplied by MongoDB Atlas.

6.6.3. RabbitMQ Cluster

The testing service relies on the RabbitMQ message broker to manage inter-service communication. As the service scales horizontally, maintaining consistent communication within the cluster becomes crucial. To ensure this, each RabbitMQ instance across different nodes must be part of a unified cluster that effectively routes requests to the available Test Runners.

In this cluster of RabbitMQ instances, multiple nodes collaborate to function as a singular message broker, distributing the workload efficiently [133]. This collaboration is made possible through a shared secret known as the "*Erlang cookie*". By using this cookie, nodes authenticate each other, enabling discovery and seamless communication [122]. By integrating this cookie into the existing Docker Compose configuration, multiple RabbitMQ instances automatically communicate with each other within the same Docker Swarm network, effectively acting as a single message broker, as detailed in the RabbitMQ clustering guide [122]. When a message is received by one of the RabbitMQ instances, it is distributed across all instances. Depending on availability, the message is then processed by a Test Runner instance, which acts as the message consumer. This approach ensures that message queuing in a horizontally scaled testing service is efficiently managed across all nodes within the cluster.

6.6.4. Preparing New Nodes

When a new node is added to the cluster of testing services, it should be ready without the need for any additional manual steps. As previously stated, the project-related operations use a RabbitMQ fan-out exchange rather than queues. This makes sure that all Test Runner instances across many nodes have the same Docker images for the projects. When a new node joins the cluster, its associated Docker daemon initially lacks these images. However, it is imperative that the Docker images corresponding to the projects in the database exist on this new node. To address this, every time the Backend Services starts, they retrieve all projects from the database, along with their respective uploaded files. Project creation messages are then dispatched to the RabbitMQ fan-out exchange associated with project uploads. This step guarantees that the Test Runner on that node creates the necessary Docker images. As a result, any new node added to the cluster synchronizes its Docker state, ensuring a consistent set of Docker images across all nodes.

6.7. Deployment

The current state of the testing service is deployed to both showcase its features and improve its testability. It was deployed on a machine within the TUM cluster, confirming the accuracy and effectiveness of the deployment process detailed in the project documentation.

The GitHub repository[4] of the web application associated with this service includes a comprehensive *README* that offers in-depth instructions for setting up the service and deploying it on any machine. As outlined in section 6.3, the entire service is encapsulated within a Docker Compose project. In addition, the frontend project is Dockerized to simplify deployability and is also encompassed within a Docker Compose project. Thus, the only necessary installation is Docker, eliminating the need for system-specific software like Nginx [134] or Apache [135]. Once Docker is set up, an environment variable file, named *.env.production.local*, is all that is necessary for the service to be deployed and executed on a machine. This file, whose structure is defined in the project documentation, specifies the host name and port for the testing service. Additionally, once the service is up and running, it is essential to ensure that the specified port is open for external access.

[4] `https://github.com/erdenbatuhan/automated-smart-contract-tester-web`

7. Results and Evaluation

This chapter evaluates the developed testing service by focusing on its security and stability, followed by an assessment of its efficiency and performance when subjected to simultaneous system loads.

7.1. Security and Stability

As explored in section 6.5, the testing service was designed to be robust against external threats and stable, ensuring optimal availability. To evaluate the service's security and stability, the following tests were conducted:

- **High-Volume Request Test:** The system was subjected to a large volume of requests during this test. A rate limit was configured to allow only 50 requests from the same IP address within a 2-minute window. Following that, 100 requests were submitted to the service. The system processed the first 50 requests and blocked the rest, successfully ensuring the service's stability.
- **Faulty Contract Submission Test:** The service was tested with two sorts of smart contract inputs: one that fails to compile and another that causes a custom error through a revert operation. The system returned the proper error messages in the container execution output, therefore neither input compromised the service's stability. The submissions were successfully stored in the database, along with their respective error messages (see Figure E.8 and Figure E.9).
- **Infinite Loop Security Test:** The service was tested using a smart contract designed with a function containing an infinite loop. The service's stability remained unaffected by this input because it successfully returned a timeout response after a predetermined interval, matching either the container timeout specified during project upload or the default container timeout value (see Figure E.10).
- **Gas Limit Compliance Test:** During this test, a new project was initially uploaded to the service with a predefined gas limit of 7 million. The smart contract input to be submitted was then modified to include a loop, purposefully extending the function's execution to consume more gas than the limit specified. Upon submission, the system's stability was not compromised, and it responded by including an error message in the test execution results indicating that the gas limit had been exceeded. This safeguard measure is provided inherently by Foundry, the test runner framework selected for smart contract testing (see Figure E.11).

Table 7.1.: Total Processing Time for Simultaneous Execution of All Submissions

Number of Simultaneous Submissions	Hardware Setting	
	10-core CPU	2-core CPU
1	2.18s	4.17s
10	4.84s	21.72s
20	11.96s	42.01s
50	21.29s	103.25s
100	42.79s	209.44s
250	99.68s	> 500s
500	209.30s	> 500s

7.2. Efficiency and Performance

The core use case of the testing service is crucially dependent on its efficiency to manage a substantial user load. To this end, a comprehensive performance evaluation was conducted using a machine[1] equipped with a 10-core CPU. The assessment measured the median time required (over a series of 10 runs) for the testing service to process and return test execution results for simultaneous submission requests, ranging in size from 1 to 500. These requests were sent consecutively to the submission endpoint, each without waiting for the response.

The evaluation results, presented in Table 7.1, demonstrate that the testing service, running on a machine with a 10-core CPU, effectively managed 250 and even 500 simultaneous submissions. However, receiving more than 50 submissions at the same time is unexpected, even with an active user base of 500 to 1000. Thus, the service's capability to efficiently process up to 50 concurrent submissions in roughly 20 seconds is quite promising, considering the improbability of encountering higher numbers of parallel submissions. Nevertheless, should the need arise to handle the load of more simultaneous submissions during peak periods, the service would require horizontal scaling to distribute the load across multiple nodes. Additionally, the efficacy of the message queuing mechanism is evident, as processing a single submission takes about 2 seconds, and processing 50 times more submissions takes only about 10 times longer, demonstrating an efficient distribution of the workload.

Additionally, the same evaluation was replicated on a separate machine[2], where the testing service had been deployed to verify deployability. As detailed in Table 7.1, the 2-core CPU of this machine proved insufficient for such a demanding load, taking approximately 4.17 seconds for a single submission, 21.72 seconds for 10, and 103.25 seconds for 50 simultaneous submissions. In contrast, the machine with a 10-core CPU had handled all 500 concurrent submissions seamlessly, though requiring about 200 seconds to return all test execution results. These findings suggest that a little increase in resources, specifically a minimum of 5 CPU cores, may be necessary to meet the service's performance expectations.

[1] Apple M1 Pro (2021, 10-core CPU, 16 GB RAM).

[2] A machine with 2 CPU cores allocated from an Intel® Xeon® CPU E5-2697A v4 @ 2.60GHz and 4 GB RAM.

8. Conclusion

The final chapter of the thesis provides a summary of the work, explaining how each research question has been addressed. It also offers a discussion of potential directions for future work and outlines the project's documentation to facilitate further development if necessary.

8.1. Summary

In conclusion, a testing service has been developed that meets the functional requirements outlined in subsection 6.1.2. This platform not only adheres to the non-functional criteria described in subsection 6.1.3 but also facilitates automated smart contract unit testing for instructors and students within the core use case, as outlined in section 1.4.

Initially, **RQ01** was addressed by discussing the requirements for automated smart contract unit testing in an educational context, including the core use case (**RQ01-a**) and exemplary exercises (**RQ01-b**). In the core use case, instructors provide students with exercises to assess their problem-solving abilities, and students receive feedback on how their submitted smart contract inputs perform against instructor-provided tests, aiding in their development of more effective smart contracts.

The chapters, chapter 2 and chapter 3, addressed **RQ02** by exploring the current state of automated smart contract testing. These chapters investigated sub-questions **RQ02-a**, which examined examples of smart contract testing services, and **RQ02-b**, which focused on identifying the most commonly used test runner frameworks for this purpose, specifically Truffle, Hardhat, and Foundry.

Following a thorough discussion about the adopted methodology in chapter 4, a comparative analysis in chapter 5 evaluated the test runner frameworks based on their usability, development experiences, feature sets, provided metrics, and performance in both containerized and non-containerized environments. This analysis, which distinguished test runner frameworks by their key features and performance assessment capabilities, addressed **RQ02-c**, concluding with Foundry being the most suitable one, followed by Hardhat and Truffle.

A comprehensive discussion in chapter 6 on the design and implementation of the testing service, utilizing Foundry as the test runner framework for smart contract testing, addressed **RQ04** through exploring the development of an automated smart contract unit testing platform for educational feedback. Security and stability considerations, pivotal to **RQ03**, were addressed by outlining strategies to manage contract execution crashes (**RQ03-a**) and to prevent system overloads, whether intentional or unintentional (**RQ03-b**). Additionally, the design for scalability and expansion to accommodate potential user growth was detailed in section 6.6, responding to **RQ04-a**.

The assessment process included extensive testing to corroborate the service's security and stability, as explained in section 7.1. The service's performance evaluation, as documented in section 7.2, demonstrated that it could withstand a heavy user load efficiently. Data in Table 7.1 confirmed the service's capability to handle significant numbers of concurrent users, maintaining good performance even with *250* and *500* simultaneous requests.

In essence, the testing service developed effectively fulfills all the established functional and non-functional requirements, offering students and instructors within the core use case a service for automated smart contract testing with a user-friendly user interface. This application is secure, stable, and performs efficiently, offering a smooth user experience. For a visual representation of the user interface, refer to the screenshots provided in Appendix E.

8.2. Future Work

The testing service has been designed to support potential extensions and enhancements, accompanied by comprehensive documentation to ease future development. Details on the documentation are further elaborated in section 8.3. Subsequent paragraphs will explore the integration of some potential new features into the testing service as part of future work.

Currently, user authentication within the testing service is managed via JWTs, allowing for open user registration with admin roles assigned by existing admins. Many educational institutions already have established authentication services that can be effortlessly incorporated into the testing service. This could be done by either replacing the current authentication system or by adding a middleware microservice that handles authentication, thus streamlining the login process and automatically assigning the correct roles to instructors and students. For instance, TUM employs its own authentication system, which could be securely integrated into the service for authenticating its instructors and students.

Furthermore, the implementation of a leaderboard to rank students based on their smart contract input performance could provide continual incentives for students to improve their contracts. This feature would promote a culture of continuous development, encouraging students to optimize their smart contracts beyond simply passing the test cases.

Moreover, the proposed addition of graphs to the frontend application would enhance the visualization of performance results, effectively complementing the detailed performance data provided in tables. As Boers (2018) emphasizes, graphical representations are optimal for communicating large amounts of data and facilitating the identification of trends, but tables excel at displaying smaller amounts of data and demonstrating simple relationships between variables [136]. Therefore, incorporating graphs to summarize the performance results, which are currently presented in tables, could speed up comprehension for the users, allowing them to quickly grasp the results without the need to carefully analyze tables.

Finally, the testing service could be improved by offering a selection of various test runner frameworks, allowing users to choose the one that best fits their needs. Such an enhancement could attract additional educational institutions that have existing exercises utilizing other test runner frameworks like Truffle or Hardhat, as opposed to Foundry. Providing this choice would streamline the process for these institutions to transition to using this testing service.

8.3. Documentation and Continued Maintenance

The setup of the testing service has been carefully designed to provide future developers with a smooth experience in understanding the code base and releasing new features. The GitHub repositories for the testing service offer extensive documentation on how to set up and run the testing service from scratch, and potentially contribute to it in the future. Separate setup guides are available for both the testing service and the frontend application. These include a range of steps, such as managing secrets, building Docker images, starting Docker containers, clearing Docker resources, and purging the database and RabbitMQ data. They also cover optional tasks such as overriding application properties. Moreover, a Postman Workspace is provided to act as a quick reference for endpoints and their application. Through a custom script, this workspace is populated with example data for requests. Additionally, each of the services within the testing service can be run independently by following the specific instructions created for them, allowing them to operate separately when required. Finally, the deployment process is also covered in detail, as briefly mentioned in section 6.7.

The testing service was developed in adherence to current best practices, which enhanced its code quality and maintainability. Tools such as ESLint [137] and Prettier [138] have been employed to elevate the code's quality. These tools are configured alongside Husky [139], which uses pre-commit hooks to trigger ESLint and Prettier via lint-staged [140] prior to each commit. This guarantees that the code changes meet the predefined quality standards. For optimal results, developers are advised to integrate ESLint into their preferred Integrated Development Environments (IDEs) for real-time linting notifications.

A. Sample Solidity Test Cases

The code snippet presented in Listing A.1 provides an example of a Solidity test case from the *BBSE Bank 2.0* project utilizing the Foundry framework. This particular test is designed to confirm the success of a deposit transaction within the bank. It accomplishes this by conducting a series of checks: it starts by depositing Ether into the bank, proceeds to validate that the account has been correctly registered as an investor at the bank with an active deposit, and concludes by ensuring that the balance of the bank has been updated correctly.

```
// A helper function to deposit Ether into the bank from a specified user address
function depositToBank(address userAddr, uint256 amount) internal {
    vm.roll(block.number + 1); // Increment block number by 1 to simulate a chain
    vm.prank(userAddr); // Inject a change of user
    vm.deal(userAddr, amount); // Deal Ether to that user

    bbseBank.deposit{value: amount}(); // Deposit
}

// Test to verify that deposits are processed correctly
function test_3_SucceedIf_DepositSucceeds() public {
    // Deposit Ether into the bank
    uint256 depositAmount = 1 ether;
    depositToBank(address(FIRST_ACC_ID), depositAmount);

    // Check the account has been correctly registered as an investor at the bank
    // after the deposit
    (bool hasActiveDeposit, uint256 investorAmount, uint256 investorStartTime) =
        bbseBank.getInvestor(address(FIRST_ACC_ID));
    assertTrue(investorHasActiveDeposit,
        "The investor should have an active deposit");
    assertEq(investorAmount, depositAmount,
        "The investor's deposited amount should match the expected value");
    assertGt(investorStartTime, 0,
        "The investor's start time should be recorded and greater than 0");

    // Check if the balance of the bank has been updated correcly after the deposit
    assertEq(address(bbseBank).balance, depositAmount,
        "The bank's balance should increase by the amount of the deposit");
    assertEq(bbseBank.totalDepositAmount(), depositAmount,
        "The bank's total deposit amount should match the amount deposited");
}
```

Listing A.1: Example Solidity Test Case - Verifying Successful Deposit

B. Performance Figures of Test Runner Frameworks

The following graphs serve to visually demonstrate the performance of the test runner frameworks: Truffle, Hardhat, and Foundry. While the performance metrics are presented in tables within the thesis, these graphs provide a visual representation that may enhance the understanding of the data and reveal underlying trends.

For a comprehensive examination of the results corresponding to Figure B.1, refer to section 5.6.

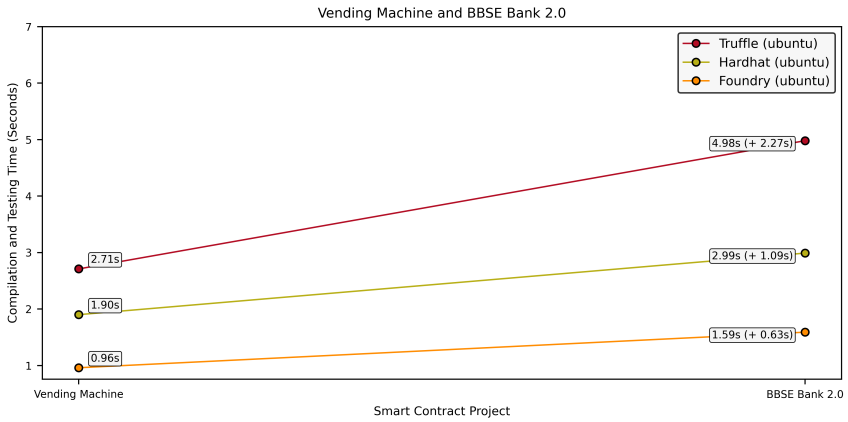

Figure B.1.: Test Execution Times of Different Frameworks (seconds)

B.1. Containerization

For an in-depth analysis of the results corresponding to Figure B.2 and Figure B.3, see subsection 5.7.4.

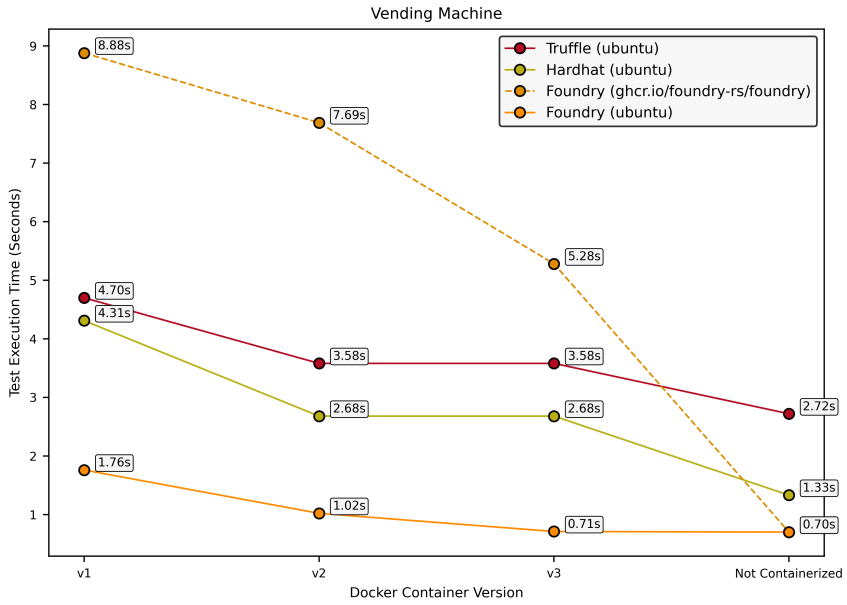

Figure B.2.: Vending Machine - Test Execution Times (seconds)

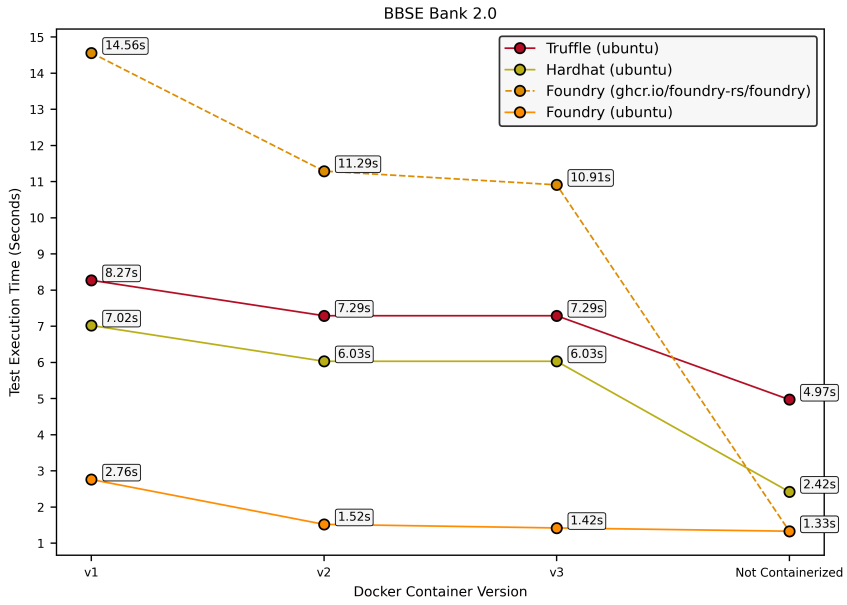

Figure B.3.: BBSE Bank 2.0 - Test Execution Times (seconds)

B.2. Scalability Capabilities

For detailed information on the results corresponding to Figure B.4, consult subsection 5.7.5.

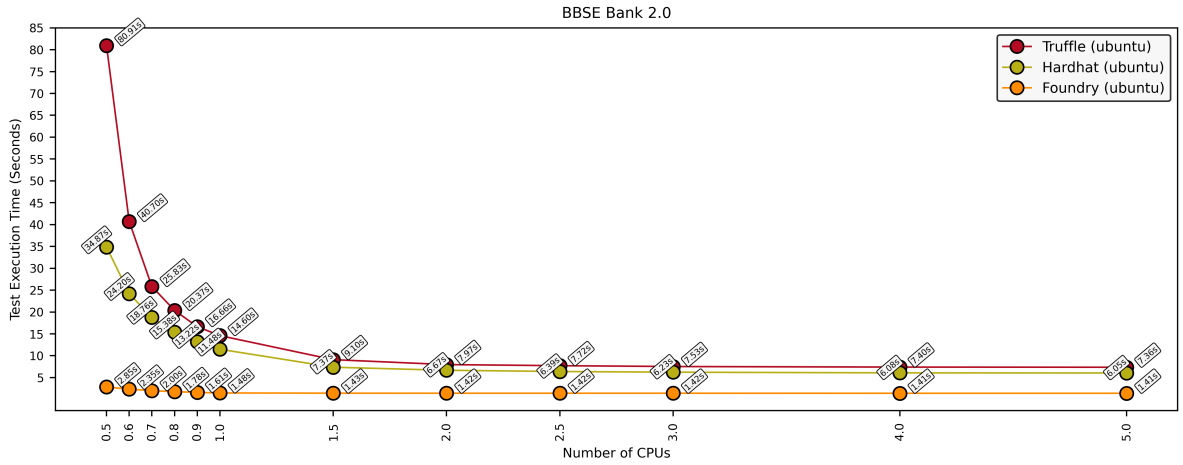

Figure B.4.: BBSE Bank 2.0 - Test Execution Times with Different Number of CPUs (seconds)

C. In-depth Docker Configurations

This chapter offers supplementary information on Docker, including aspects like Docker exit codes, and presents in-depth information on the specific Docker configurations used in this project, such as Dockerfiles.

C.1. Dockerfile for Project Image Creation

Presented in Figure C.1 is the Dockerfile used to create Docker images from the uploaded exercises or projects.

```
# Use the latest Ubuntu image as the base
FROM ubuntu:latest

# Install essential utilities: curl & git
RUN apt-get -y update
RUN apt-get -y install curl
RUN apt-get -y install git

# Install Foundry
RUN curl -L https://foundry.paradigm.xyz | bash
ENV PATH="${PATH}:/root/.foundry/bin"
RUN foundryup

# Set the working directory to /app
WORKDIR /app

# Copy the configuration files into the container
COPY foundry.toml .
COPY remappings.txt .

# Install the libraries after copying the required files needed for that purpose into the container
COPY .gitmodules .
COPY install_libraries.sh .
RUN git init
RUN ./install_libraries.sh

# Copy the tests into the container
COPY test test

# (1) "forge build": Ensures that the compiler is pre-installed and the dependencies are pre-created
# (2) "forge snapshot": (1) + Generates gas snapshots for all the test functions using the solution (the src folder) provided
COPY src src
# RUN forge build (Redundant, as "forge snapshot" already compiles the project)
RUN forge snapshot --snap .gas-snapshot
RUN rm -rf src/*

# Remove the build artifacts and cache directories
RUN forge clean

# Run the tests (make sure to copy the "src" folder containing the implemented contracts before running the container!)
CMD ["forge", "test", "-vv"]
```

Figure C.1.: Dockerfile Utilized for Project Image Creation

C.2. Docker Exit Codes

Various exit codes in Docker describe the status or cause for a container's termination. These exit codes along with their respective descriptions are listed in Table C.1.

Table C.1.: Docker Exit Codes

Exit Code	Description
0	Purposely stopped
1	Application error or incorrect reference
125	Container failed to run error
126	Command invoke error
127	File or directory not found
128	Invalid argument used on exit
134	Abnormal termination (SIGABRT)
137	Immediate termination (SIGKILL)
139	Segmentation fault (SIGSEGV)
143	Graceful termination (SIGTERM)
255	Exit status out of range

Source: [141]

D. Listing of REST Endpoints

This chapter provides a list of REST Endpoints exposed by the microservices in the testing service, which are essential for inclusion in this thesis.

D.1. Endpoints for Backend Services

The available REST endpoints of the Backend Services are listed in Table D.1. It is important to note that all these endpoints are prefixed with:
`/api/automated-smart-contract-tester/services/v1`

Table D.1.: REST Endpoints for Backend Services

Description	**Method**	**Endpoint (with Path Variables)**
Healthcheck	GET	*/healthcheck*
Signup	POST	*/auth/signup*
Login	POST	*/auth/login*
Logout	GET	*/auth/logout*
Get all users	GET	*/users*
Get a specific user	GET	*/users/:userId*
Delete a user	DELETE	*/users/:userId*
Get test execution args	GET	*/projects/descriptions/test-execution-arguments*
Get all projects	GET	*/projects*
Get a specific project	GET	*/projects/:projectName*
Upload a new project	POST	*/projects/:projectName/upload*
Update an existing project	PUT	*/projects/:projectName/upload*
Update project config	PUT	*/projects/:projectName/update*
Download a project	GET	*/projects/:projectName/download*
Delete a project	DELETE	*/projects/:projectName*
Get all submissions	GET	*/projects/:projectName/submissions*
Get a specific submission	GET	*/projects/:projectName/submissions/:submissionId*
Upload a submission	POST	*/projects/:projectName/submissions*
Download a submission	GET	*/projects/:projectName/submissions/:submissionId/download*
Delete a submission	DELETE	*/projects/:projectName/submissions/:submissionId*
Get all message requests	GET	*/message-requests*
Get a message request	GET	*/message-requests/:messageRequestId*

E. Testing Service Screenshots

The accompanying screenshots illustrate the primary features of the final testing service on the frontend application. The user experience begins with the login screen, displayed in Figure E.1 (the signup interface is identical). Uploading a new project and a submission are illustrated in Figure E.2 and Figure E.3, respectively. Listing previously uploaded projects and submissions are demonstrated in Figure E.4 and Figure E.5, consecutively. The test execution results for a submission can be analyzed, as shown in Figure E.6, which exhibits a submission with *25 passing* and *2 failing* tests and a *gas consumption difference of +0.56%*, indicating a higher total gas usage compared to the reference contracts, a negative outcome. This figure also shows the gas consumption and difference per test. Furthermore, user feedback is provided through custom alerts, an example of which is presented in Figure E.7 following a submission download.

For an in-depth view of the test results for faulty and successful submissions, refer to section E.1 and section E.2.

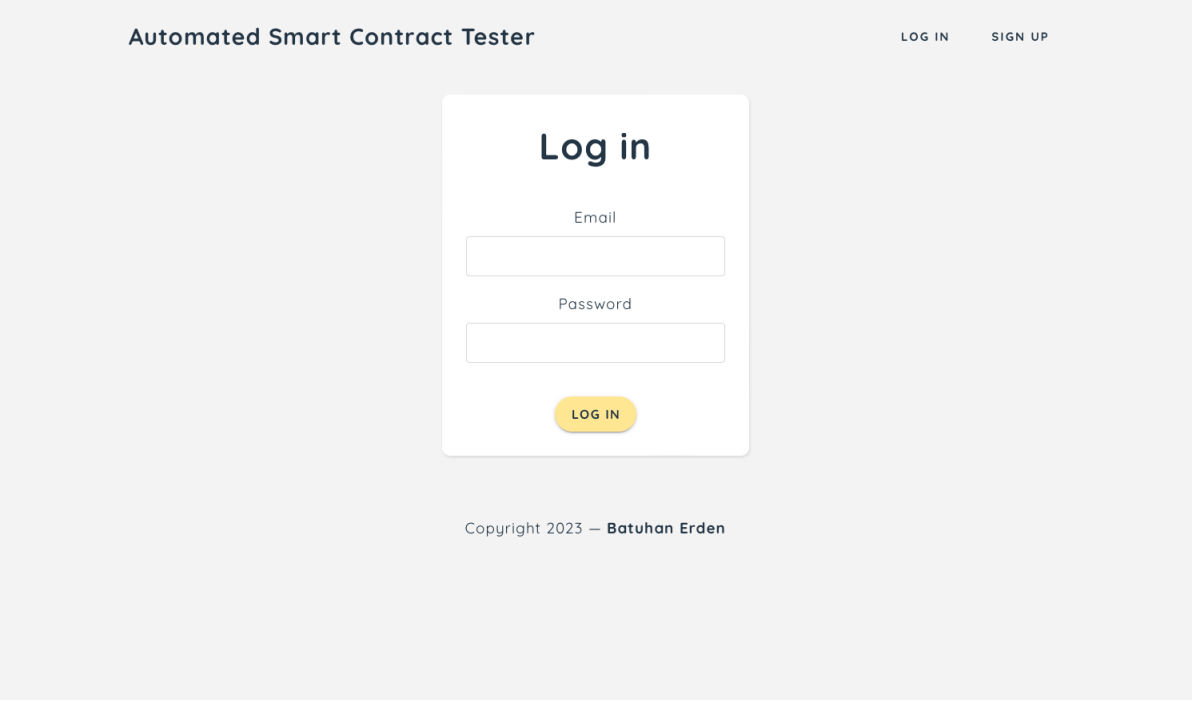

Figure E.1.: Logging into the Testing Service

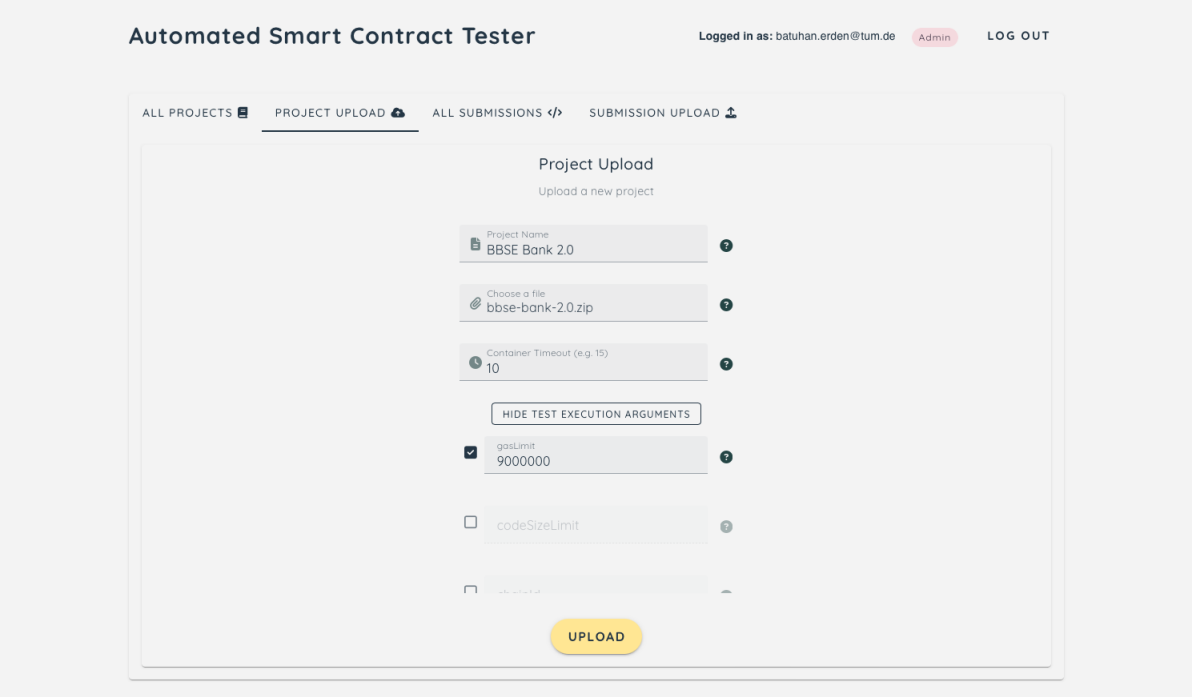

Figure E.2.: Uploading a New Project

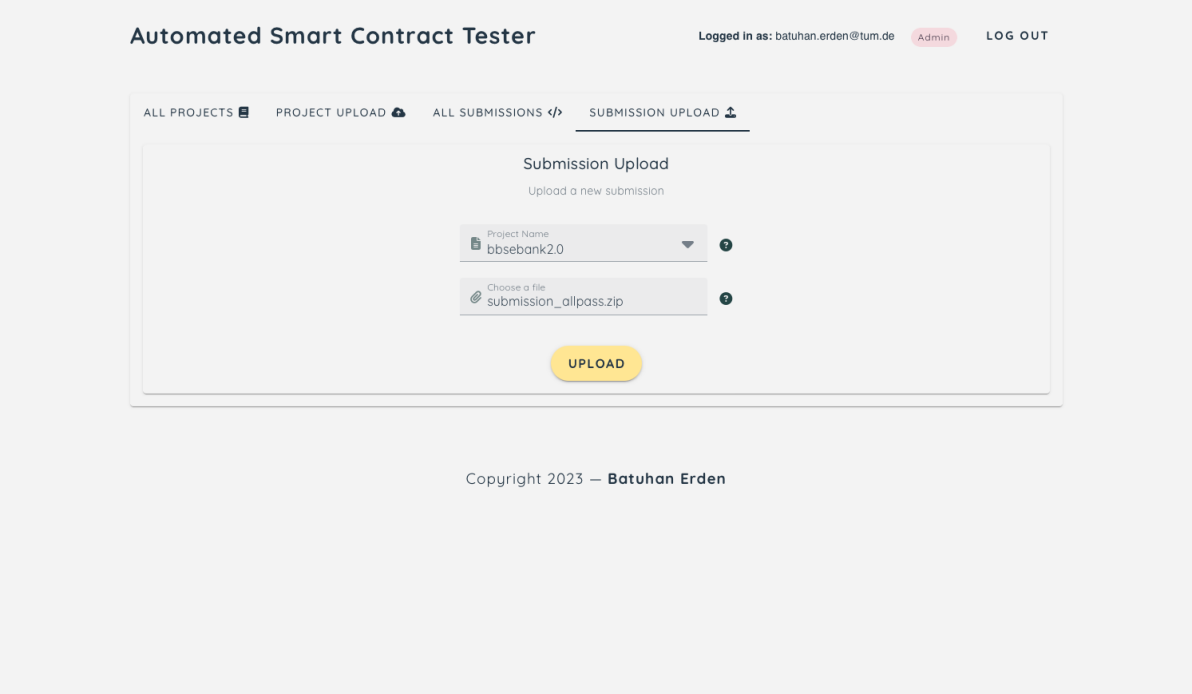

Figure E.3.: Uploading a New Submission

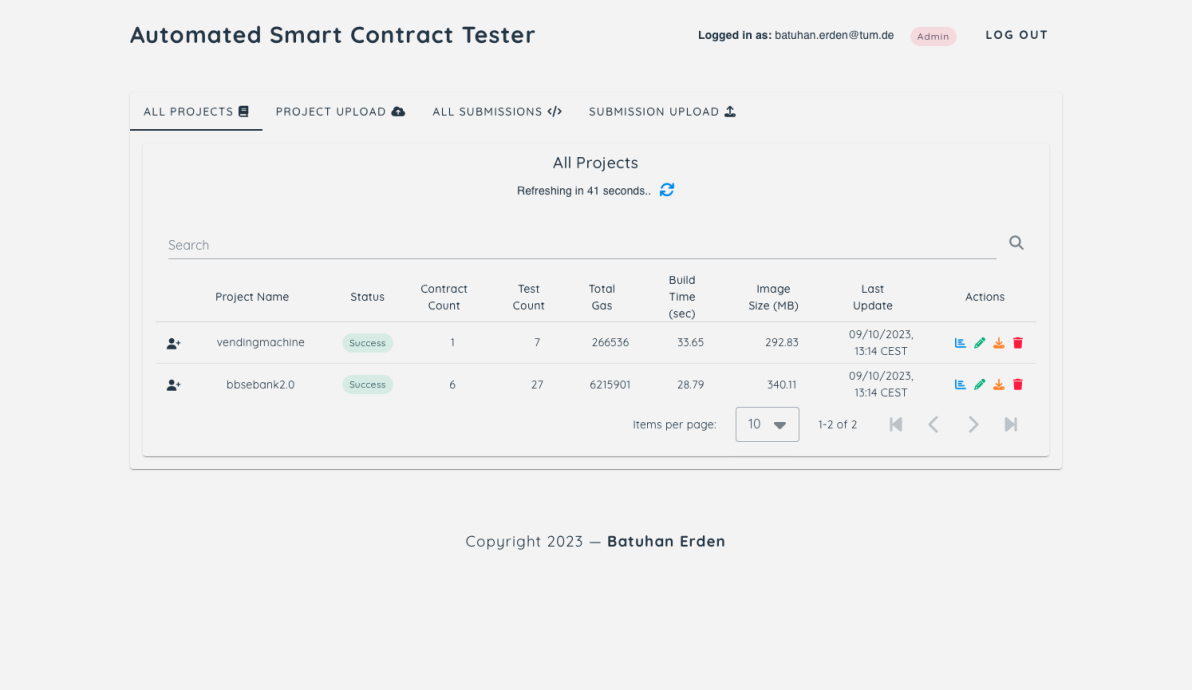

Figure E.4.: Viewing Uploaded Projects

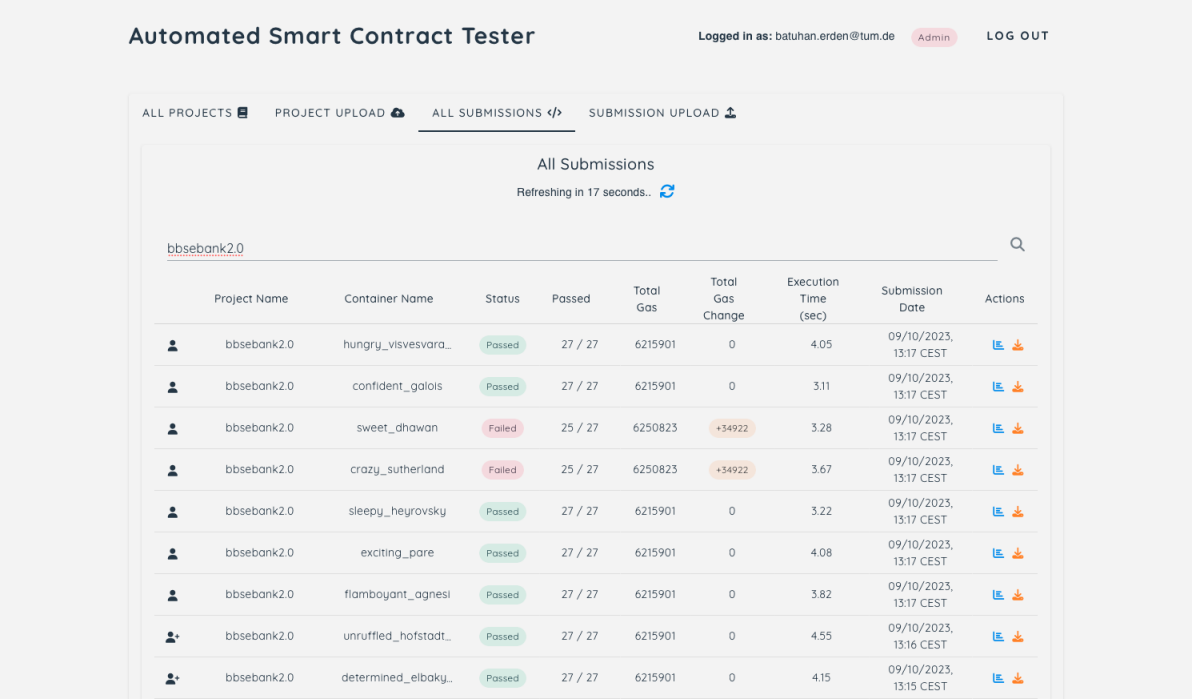

Figure E.5.: Viewing Uploaded Submissions

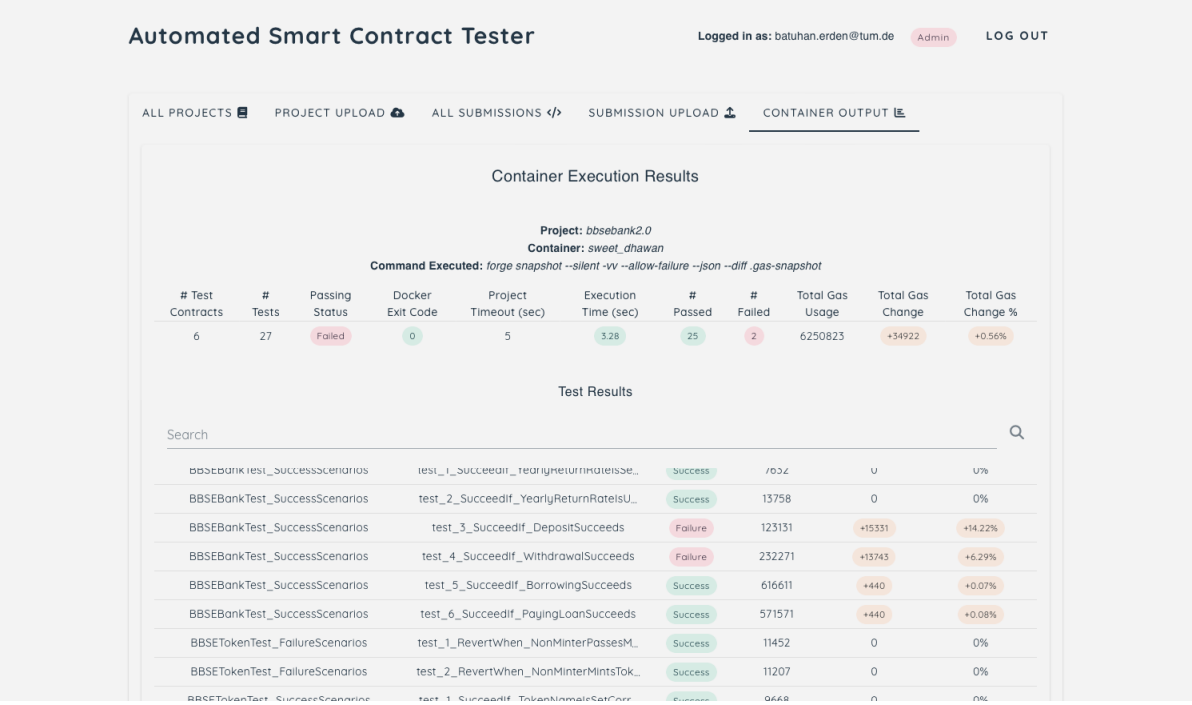

Figure E.6.: Test Execution Results for a Submission

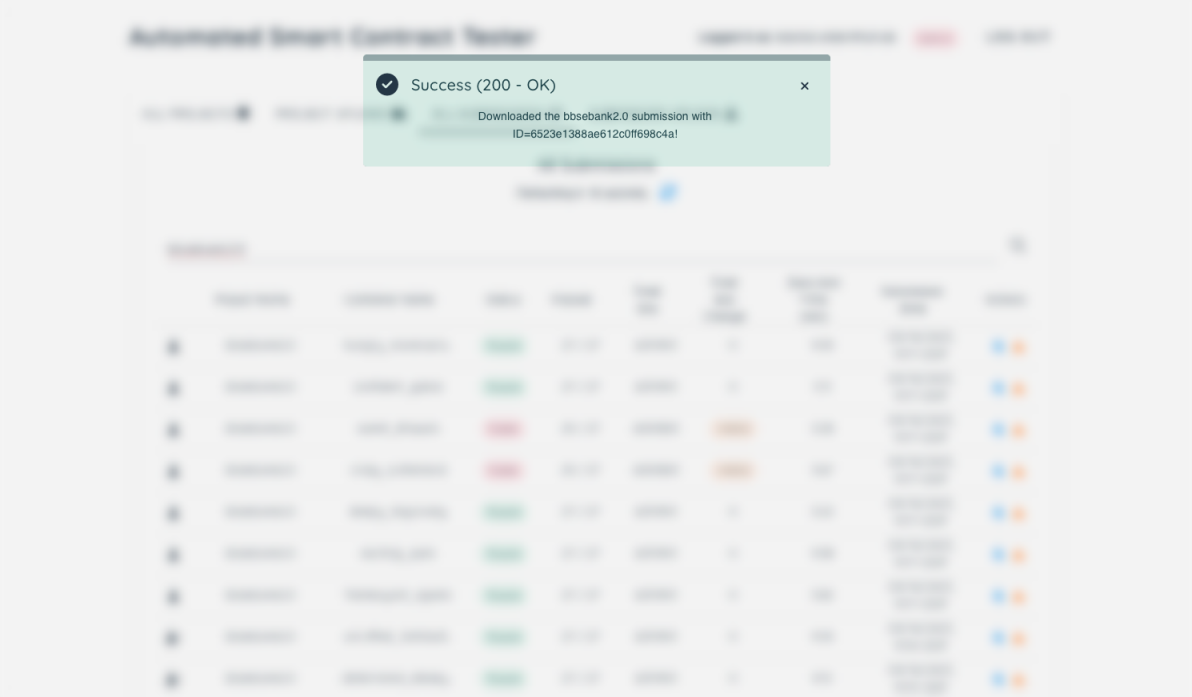

Figure E.7.: Success Alert Displayed after Downloading a Submission

E.1. Analysis of Faulty Submissions

The presented figures serve as visual aids for analyzing faulty submissions on the testing service. Each figure corresponds to a unique error scenario encountered during test execution.

Initially, Figure E.8 showcases a submission with failing tests, and detailed error messages are accessible by hovering over each failed test's indicator.

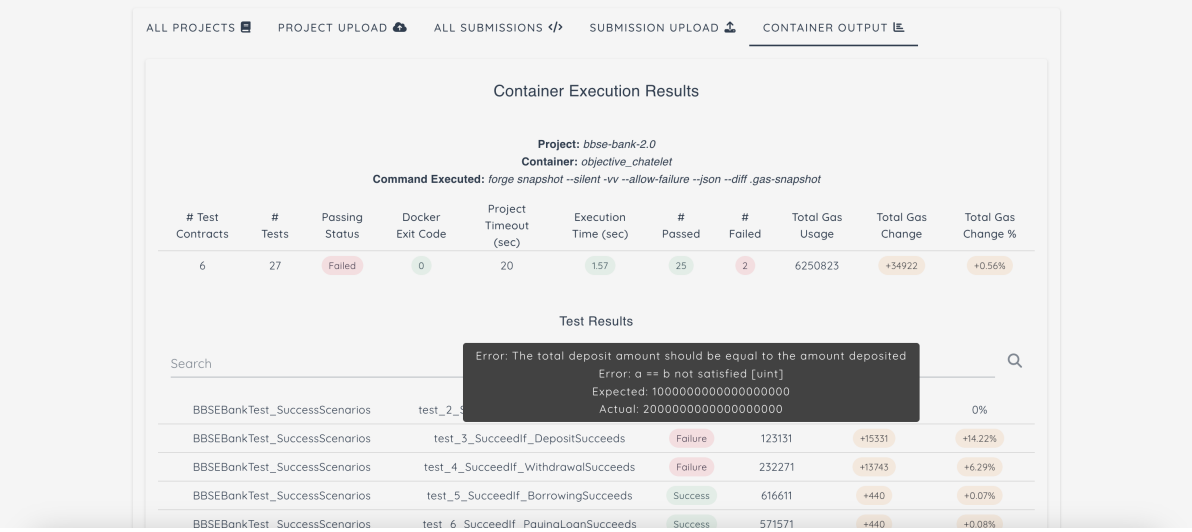

Figure E.8.: Failed Submission - Test Execution with Failing Tests

Additionally, Figure E.9 illustrates a submission that resulted in a contract error where the contract reverted with an error, with detailed information accessible upon hovering over the associated failed test.

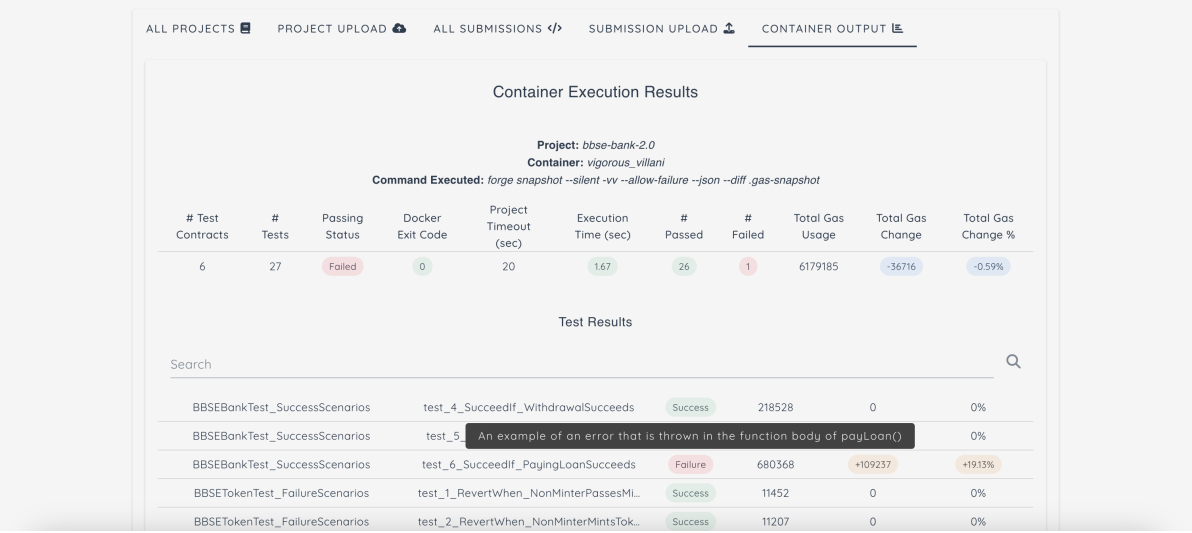

Figure E.9.: Failed Submission - Test Execution with Contract Error

Furthermore, Figure E.10 presents a submission that exceeded the time limit. The project's time limit was set to 20 seconds, and the showcased submission timed out at this limit. This particular test case contained a smart contract with an infinite loop to intentionally cause a timeout, also testing the stability of the service against inefficient smart contract inputs.

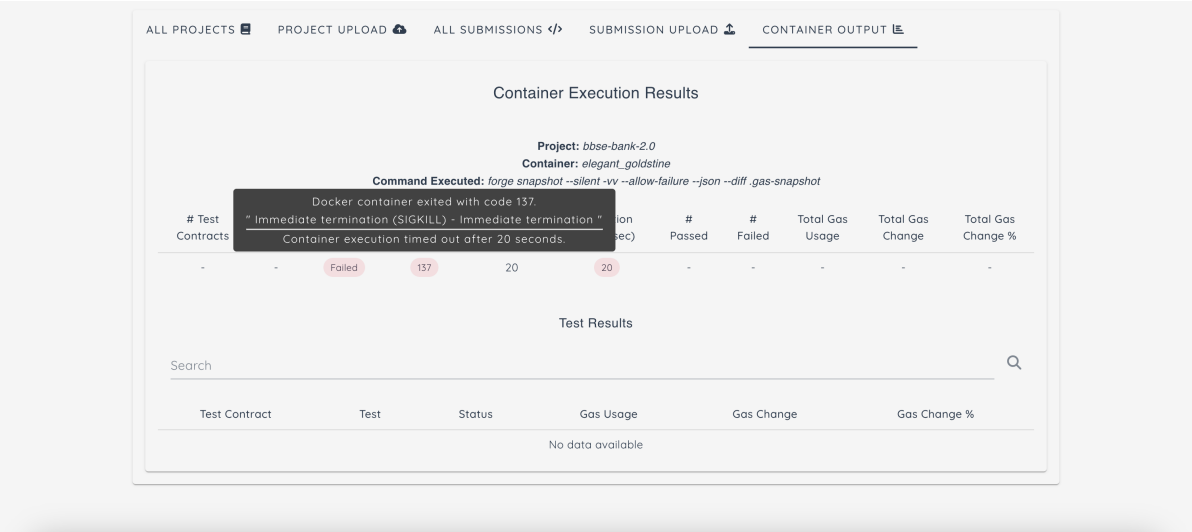

Figure E.10.: Failed Submission - Test Execution Timeout

Lastly, Figure E.11 displays a submission that fails due to excessive gas consumption. The test exceeded the project's gas limit, which is indicated by a message displayed upon hover, while the gas usage values are presented in adjacent columns, thus evaluating the testing service's adherence to gas limits.

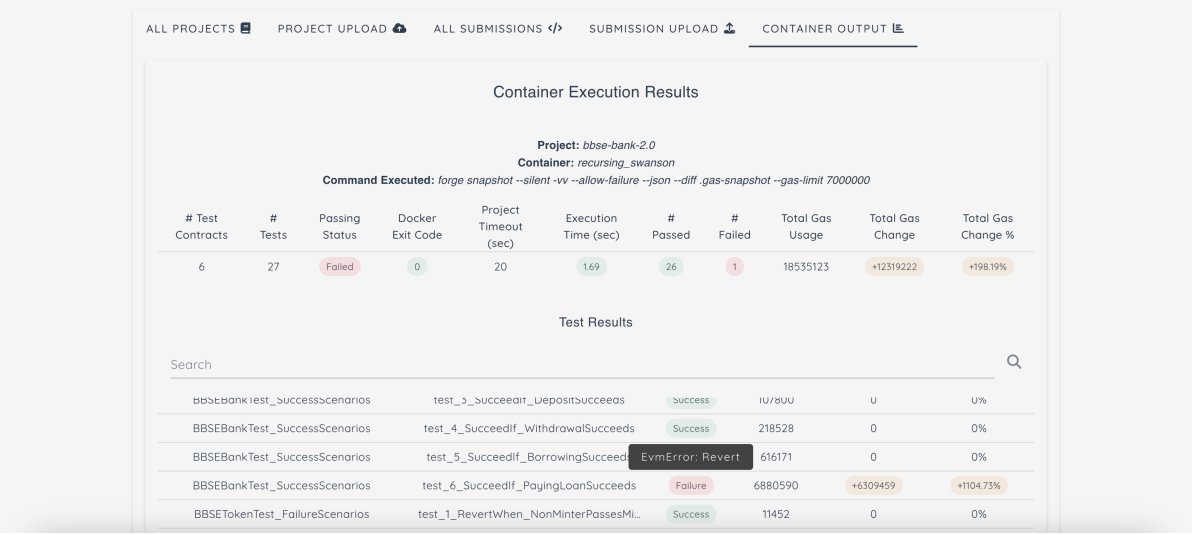

Figure E.11.: Failed Submission - Test Execution with Excessive Gas Usage

E.2. Analysis of Successful Submissions

The test execution results for an exemplary successful submission are shown in Figure Figure E.12, illustrating a scenario where all tests have passed and gas consumption is identical to the smart contracts originally uploaded with the project. Although a submission consuming the *exact same* amount of gas as the solution is a rare case, for the demonstration of a *perfect* submission, the *very same* contracts provided with the project were used as a submission in this test case.

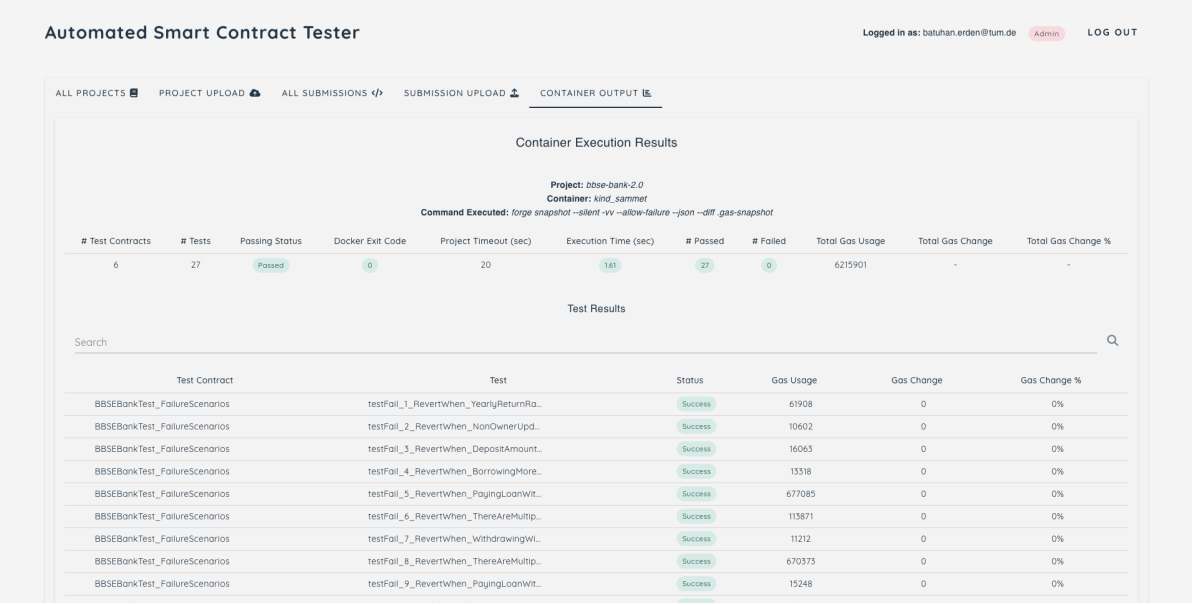

Figure E.12.: Successful Submission - Test Execution with All Passing Tests

List of Abbreviations

ABI Application Binary Interface. 26

AMQP Advanced Message Queuing Protocol. 70

BBSE Blockchain-based Systems Engineering. iii, 1, 3, 5, 27, 75

CI Continuous Integration. 27

CMS Content Management System. 20

DAO Decentralized Autonomous Organization. 2

dApp Decentralized Application. 1, 2, 7, 9, 12, 20, 25, 27

EVM Ethereum Virtual Machine. 10, 20, 26, 27, 33, 39

GUI Graphical User Interface. 38, 51, 52

HPC High Performance Computing. 20, 76

IDE Integrated Development Environment. 83

JWT JSON Web Token. 22, 60, 67, 72, 82

npm Node Package Manager. 24–26, 28, 30, 32, 43, 45

OS Operating System. 14–16

PoW Proof of Work. 8, 9

REPL Read-Eval-Print Loop. 27

SaaS Software as a Service. 17

SEBIS Software Engineering for Business Information Systems. iii

TUM Technical University of Munich. 1, 5, 18, 27, 75, 78, 82

UML Unified Modeling Language. 57

VM Virtual Machine. 14–16, 20, 100

List of Figures

List of Tables

Listings

Bibliography

[1] F. A. Sunny, P. Hajek, M. Munk, M. Z. Abedin, M. S. Satu, M. I. A. Efat, and M. J. Islam. "A Systematic Review of Blockchain Applications". In: *IEEE Access* 10 (2022), pp. 59155–59177. DOI: 10.1109/ACCESS.2022.3179690.

[2] R. Zhang, R. Xue, and L. Liu. "Security and privacy on blockchain". In: *ACM Computing Surveys (CSUR)* 52.3 (2019), pp. 1–34.

[3] K. Wu, Y. Ma, G. Huang, and X. Liu. "A first look at blockchain-based decentralized applications". In: *Software: Practice and Experience* 51.10 (2021), pp. 2033–2050. DOI: https://doi.org/10.1002/spe.2751. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2751. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2751.

[4] The Chair of Software Engineering for Business Information Systems at TUM. *Blockchain-based Systems Engineering*. https://wwwmatthes.in.tum.de/pages/enf3vo4lqv74/Blockchain-based-Systems-Engineering. Accessed: November 10, 2023.

[5] A. Alkhajeh. *Blockchain and smart contracts: The need for better education*. Rochester Institute of Technology, 2020.

[6] A. Hughes, A. Park, J. Kietzmann, and C. Archer-Brown. "Beyond Bitcoin: What blockchain and distributed ledger technologies mean for firms". In: *Business Horizons* 62.3 (2019), pp. 273–281. ISSN: 0007-6813. DOI: https://doi.org/10.1016/j.bushor.2019.01.002. URL: https://www.sciencedirect.com/science/article/pii/S0007681319300023.

[7] E. J. De Aguiar, B. S. Faiçal, B. Krishnamachari, and J. Ueyama. "A survey of blockchain-based strategies for healthcare". In: *ACM Computing Surveys (CSUR)* 53.2 (2020), pp. 1–27.

[8] M. Foth. "The promise of blockchain technology for interaction design". In: *Proceedings of the 29th Australian conference on computer-human interaction*. 2017, pp. 513–517.

[9] S. Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: *Available at: https://bitcoin.org/bitcoin.pdf* (2008). Accessed: October 2, 2023.

[10] R. M. Parizi, A. Dehghantanha, K.-K. R. Choo, and A. Singh. "Empirical vulnerability analysis of automated smart contracts security testing on blockchains". In: *arXiv preprint arXiv:1809.02702* (2018).

[11] V. Buterin. "A next-generation smart contract and decentralized application platform". In: *White paper* 3.37 (2014), pp. 2–1.

[12] G. Wood. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

[13] S. Bragagnolo, H. Rocha, M. Denker, and S. Ducasse. "SmartInspect: solidity smart contract inspector". In: *2018 International workshop on blockchain oriented software engineering (IWBOSE)*. IEEE. 2018, pp. 9–18.

[14] S. Rouhani and R. Deters. "Security, Performance, and Applications of Smart Contracts: A Systematic Survey". In: *IEEE Access* 7 (2019), pp. 50759–50779. DOI: 10.1109/ACCESS.2019.2911031.

[15] B. K. Mohanta, S. S. Panda, and D. Jena. "An overview of smart contract and use cases in blockchain technology". In: *2018 9th international conference on computing, communication and networking technologies (ICCCNT)*. IEEE. 2018, pp. 1–4. DOI: 10.1109/ICCCNT.2018.8494045.

[16] E. Sunday. *Top 5 smart contract programming languages for blockchain*. https://blog.logrocket.com/smart-contract-programming-languages/. Accessed: November 03, 2023. 2021.

[17] H. Zhou, A. Milani Fard, and A. Makanju. "The state of ethereum smart contracts security: Vulnerabilities, countermeasures, and tool support". In: *Journal of Cybersecurity and Privacy* 2.2 (2022), pp. 358–378. ISSN: 2624-800X. DOI: 10.3390/jcp2020019. URL: https://www.mdpi.com/2624-800X/2/2/19.

[18] D. Perez and B. Livshits. "Broken metre: Attacking resource metering in EVM". In: *arXiv preprint arXiv:1909.07220* (2019).

[19] C. Jentzsch. "Decentralized autonomous organization to automate governance". In: *White paper, November* (2016).

[20] M. Rodler, W. Li, G. O. Karame, and L. Davi. "Sereum: Protecting existing smart contracts against re-entrancy attacks". In: *arXiv preprint arXiv:1812.05934* (2018).

[21] I. Puddu, A. Dmitrienko, and S. Capkun. *µchain: How to Forget without Hard Forks*. Cryptology ePrint Archive, Paper 2017/106. https://eprint.iacr.org/2017/106. 2017. URL: https://eprint.iacr.org/2017/106.

[22] Docker, Inc. "Docker". In: *línea].[Junio de 2017]. Disponible en: https://www.docker.com/what-docker* (2020).

[23] C. Dannen. *Introducing Ethereum and Solidity*. Vol. 1. Springer, 2017.

[24] Rainberry, Inc. *BitTorrent*. https://www.bittorrent.com. Accessed: November 5, 2023.

[25] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun. "On the security and performance of proof of work blockchains". In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, pp. 3–16.

[26] F. A. Aponte-Novoa, A. L. S. Orozco, R. Villanueva-Polanco, and P. Wightman. "The 51% attack on blockchains: A mining behavior study". In: *IEEE Access* 9 (2021), pp. 140549–140564.

[27] Bitcoin Gold. *Bitcoin Gold*. https://bitcoingold.org. Accessed: November 5, 2023.

[28] Hacken and B. Barwikowski. *51% Attack: The Concept, Risks & Prevention*. https://hacken.io/discover/51-percent-attack. Accessed: November 5, 2023.

[29] F. Kochan. *What is the Ethereum Virtual Machine (EVM)?* https://www.quicknode.com/guides/ethereum-development/getting-started/what-is-the-ethereum-virtual-machine-evm. Accessed: November 5, 2023.

[30] L. Burkholder. "The halting problem". In: *ACM SIGACT News* 18.3 (1987), pp. 48–60.

[31] M. Wöhrer and U. Zdun. "Smart contracts: security patterns in the ethereum ecosystem and solidity". In: *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE. 2018, pp. 2–8.

[32] M. Barboni, A. Morichetta, and A. Polini. "Smart contract testing: challenges and opportunities". In: *Proceedings of the 5th International Workshop on Emerging Trends in Software Engineering for Blockchain*. 2022, pp. 21–24.

[33] W. Zou, D. Lo, P. S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu. "Smart contract development: Challenges and opportunities". In: *IEEE Transactions on Software Engineering* 47.10 (2019), pp. 2084–2106.

[34] G. A. Pierro, R. Tonelli, and M. Marchesi. "An organized repository of ethereum smart contracts' source codes and metrics". In: *Future internet* 12.11 (2020), p. 197.

[35] ConsenSys Software Inc. *Truffle Suite*. https://trufflesuite.com. Accessed: July 25, 2023.

[36] Nomic Labs LLC. *Hardhat*. https://hardhat.org. Accessed: July 27, 2023.

[37] L. Palechor and C.-P. Bezemer. "How are Solidity smart contracts tested in open source projects? An exploratory study". In: *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*. 2022, pp. 165–169.

[38] DappHub. *DappTools*. https://dapp.tools. Accessed: November 8, 2023.

[39] L. Tran. *Compare the top 3 smart contract frameworks for web3 projects in 2022: HardHat, Truffle, and Foundry*. https://medium.com/coinmonks/compare-the-top-3-smart-contract-frameworks-for-web3-projects-in-2022-hardhat-truffle-and-ca26c638c597. Accessed: October 30, 2023. 2022.

[40] O. Nordbjerg and other contributors. *Foundry*. https://book.getfoundry.sh. Accessed: July 29, 2023.

[41] G. Konstantopoulos and other contributors. *Foundry GitHub Repository*. https://github.com/foundry-rs/foundry. Accessed: July 28, 2023. 2021.

[42] S. N. T.-c. Chiueh and S. Brook. "A survey on virtualization technologies". In: *Rpe Report* 142 (2005).

[43] Amazon Web Services, Inc. *What's the Difference Between Containers and Virtual Machines?* https://aws.amazon.com/compare/the-difference-between-containers-and-virtual-machines. Accessed: November 8, 2023.

[44] C. Pahl. "Containerization and the paas cloud". In: *IEEE Cloud Computing* 2.3 (2015), pp. 24–31.

[45] R. Dua, A. R. Raja, and D. Kakadia. "Virtualization vs containerization to support paas". In: *2014 IEEE International Conference on Cloud Engineering*. IEEE. 2014, pp. 610–614.

[46] Docker Inc. *Docker Documentation*. `https://docs.docker.com`. Accessed: November 8, 2023.

[47] T. Combe, A. Martin, and R. Di Pietro. "To docker or not to docker: A security perspective". In: *IEEE Cloud Computing* 3.5 (2016), pp. 54–62.

[48] T. Bui. "Analysis of docker security". In: *arXiv preprint arXiv:1501.02967* (2015).

[49] C. Burniske. *Containers: The Next Generation of Virtualization?* `https://ark-invest.com/articles/analyst-research/containers-virtualization`. Accessed: November 8, 2023.

[50] Docker Inc. *Docker Hub*. `https://hub.docker.com`. Accessed: October 27, 2023.

[51] Docker Inc. *Docker Compose*. `https://docs.docker.com/compose`. Accessed: October 25, 2023.

[52] The Chair of Applied Software Engineering at TUM. *Artemis: Interactive Learning with Individual Feedback*. `https://docs.artemis.cit.tum.de`. Accessed: November 9, 2023.

[53] The Chair of Computer Architecture and Parallel Systems at TUM. *Parallel Programming (IN2147)*. `https://www.ce.cit.tum.de/caps/lehre/ss23/vorlesungen/parallel-programming`. Accessed: November 9, 2023.

[54] S. N. Raje. "Performance Comparison of Message Queue Methods". PhD thesis. University of Nevada, Las Vegas, 2019.

[55] S. Driessen, D. Di Nucci, G. Monsieur, D. A. Tamburri, and W.-J. v. d. Heuvel. "Automated test-case generation for solidity smart contracts: The AGSolT approach and its evaluation". In: *arXiv preprint arXiv:2102.08864* (2021).

[56] F. Mi, C. Zhao, Z. Wang, S. Halim, X. Li, Z. Wu, L. Khan, and B. Thuraisingham. "An Automated Vulnerability Detection Framework for Smart Contracts". In: *arXiv preprint arXiv:2301.08824* (2023).

[57] C. Benabbou and Ö. Gürcan. "A survey of verification, validation and testing solutions for smart contracts". In: *2021 Third International Conference on Blockchain Computing and Applications (BCCA)*. IEEE. 2021, pp. 57–64.

[58] Celo Academy. *Truffle vs Hardhat: A Comprehensive Comparison for Developing on the Celo Blockchain*. `https://celo.academy/t/truffle-vs-hardhat-a-comprehensive-comparison-for-developing-on-the-celo-blockchain/2672`. Accessed: November 9, 2023.

[59] A. Ufano. *Smart Contract Frameworks – Foundry vs Hardhat: Differences in Performance and Developer Experience*. `https://chainstack.com/foundry-hardhat-differences-performance`. Accessed: November 9, 2023. 2022.

[60] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen. "Defectchecker: Automated smart contract defect detection by analyzing evm bytecode". In: *IEEE Transactions on Software Engineering* 48.7 (2021), pp. 2189–2207.

[61] Z. Li, H. Wu, J. Xu, X. Wang, L. Zhang, and Z. Chen. "Musc: A tool for mutation testing of ethereum smart contract". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2019, pp. 1198–1201.

[62] M. Myeza. *Deploying Your Smart Contracts on a Testnet? This Is What You Can Expect*. `https://www.rmb.co.za/page/deploying-your-smart-contracts-on-a-testnet-this-is-what-you-can-expect`. Accessed: November 9, 2023.

[63] Consensys. *ConsenSys Diligence*. `https://consensys.io/diligence`. Accessed: November 9, 2023.

[64] N. Cohen. *What is Performance Testing? What is Load Testing? What is Stress Testing? A Comparison*. `https://www.blazemeter.com/blog/performance-testing-vs-load-testing-vs-stress-testing`. Accessed: November 9, 2023.

[65] Suffescom Solutions Inc. *Suffescom Solutions - Smart Contract Testing Services*. `https://www.suffescom.com/smart-contract-testing-service`. Accessed: November 9, 2023.

[66] Crytic. *Etheno*. `https://github.com/crytic/etheno`. Accessed: November 9, 2023.

[67] M. T. Chung, N. Quang-Hung, M.-T. Nguyen, and N. Thoai. "Using docker in high performance computing applications". In: *2016 IEEE Sixth International Conference on Communications and Electronics (ICCE)*. IEEE. 2016, pp. 52–57.

[68] D. Bonacorsi, G. Eulisse, T. Boccali, and E. Mazzoni. "Containerization of CMS applications with docker". In: *PoS* (2016), p. 007.

[69] N. Naik. "Building a virtual system of systems using docker swarm in multiple clouds". In: *2016 IEEE International Symposium on Systems Engineering (ISSE)*. IEEE. 2016, pp. 1–3.

[70] Docker Inc. *Docker Swarm*. `https://docs.docker.com/engine/swarm`. Accessed: October 26, 2023.

[71] C. Jansen, M. Witt, and D. Krefting. "Employing docker swarm on openstack for biomedical analysis". In: *Computational Science and Its Applications–ICCSA 2016: 16th International Conference, Beijing, China, July 4-7, 2016, Proceedings, Part II 16*. Springer. 2016, pp. 303–318.

[72] C. Nance. *Typescript essentials*. Packt Publishing Ltd, 2014.

[73] VMware, Inc. *RabbitMQ*. `https://www.rabbitmq.com`. Accessed: October 23, 2023.

[74] *Truffle - Blockchain Platform for Ethereum Smart Contracts*. `https://www.kaleido.io/blockchain-platform/truffle`. Accessed: July 25, 2023.

[75] P. Hartel and M. van Staalduinen. "Truffle tests for free – Replaying Ethereum smart contracts for transparency". In: 2019. arXiv: `1907.09208 [cs.SE]`.

[76] ConsenSys Software Inc. *Truffle Documentation*. https://trufflesuite.com/docs/truffle. Accessed: July 25, 2023.

[77] ConsenSys Software Inc. *Ganache*. https://trufflesuite.com/ganache. Accessed: July 25, 2023.

[78] Coinmonks. *Deploying a Smart Contract with Truffle & Ganache*. https://medium.com/coinmonks/deploying-a-smart-contract-with-truffle-ganache-fde535318ed5. Accessed: July 25, 2023.

[79] Nomic Labs LLC. *Hardhat Documentation*. https://hardhat.org/docs. Accessed: July 27, 2023.

[80] N. Rockson. *Hardhat Versus Truffle: Which Smart Contract Framework is Best?* https://www.slashauth.com/post/hardhat-vs-truffle. Accessed: October 30, 2023.

[81] Nomic Labs LLC. *Hardhat Chai Matchers*. https://hardhat.org/hardhat-chai-matchers. Accessed: July 27, 2023.

[82] R. Moore. *Ethers.js*. https://docs.ethers.org. Accessed: July 27, 2023.

[83] Ethereum Revision. *Web3.js*. https://web3js.readthedocs.io. Accessed: July 27, 2023.

[84] D. Dauliya. *Ethers vs Web3*. https://guideofdapp.com/posts/ethers-vs-web3. Accessed: July 27, 2023.

[85] G. Bierman, M. Abadi, and M. Torgersen. "Understanding TypeScript". In: *ECOOP 2014 – Object-Oriented Programming*. Ed. by R. Jones. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 257–281. ISBN: 978-3-662-44202-9.

[86] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. "Evaluating Fuzz Testing". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 2123–2138. ISBN: 9781450356930. DOI: 10.1145/3243734.3243804. URL: https://doi.org/10.1145/3243734.3243804.

[87] O. Nordbjerg and other contributors. *Foundry Cheatcodes Reference*. https://book.getfoundry.sh/cheatcodes. Accessed: July 29, 2023.

[88] J. Spruance. *Vending Machine Source Code*. https://github.com/jspruance/block-explorer-tutorials/blob/main/apps/VendingMachine/vending-machine/contracts/VendingMachine.sol. Accessed: July 31, 2023. 2022.

[89] B. Öz. *BBSE Bank 2.0 GitHub Repository*. https://github.com/sebischair/bbse-bank-2.0. Accessed: July 31, 2023. 2022.

[90] B. Öz. *BBSE Bank GitHub Repository*. https://github.com/sebischair/bbse-bank. Accessed: July 31, 2023. 2022.

[91] zOS Global Limited and contributors. *OpenZeppelin GitHub Repository*. https://github.com/OpenZeppelin/openzeppelin-contracts. Accessed: July 31, 2023. 2016.

[92] OpenJS Foundation and Mocha contributors. *Mocha*. https://mochajs.org. Accessed: July 31, 2023.

[93] Chai.js Assertion Library. *Chai Assertion Library*. https://www.chaijs.com. Accessed: July 31, 2023.

[94] S. Nnebe, C. Okafor, T. Onyeyili, and G. Nathaniel. "Design and Implementation of Decentralized Voting System on the Ethereum Blockchain". In: *International Journal of Computer (IJC)* 45.1 (2022), pp. 95–104.

[95] R. Kalis. *truffle-assertions*. https://github.com/rkalis/truffle-assertions. Accessed: July 30, 2023.

[96] M. Wöhrer and U. Zdun. "DevOps for Ethereum Blockchain Smart Contracts". In: *2021 IEEE International Conference on Blockchain (Blockchain)*. IEEE. 2021, pp. 244–251. DOI: 10.1109/Blockchain53845.2021.00040.

[97] O. Nordbjerg and other contributors. *Foundry DSTest*. https://book.getfoundry.sh/reference/ds-test. Accessed: July 30, 2023.

[98] A. Rea. *solidity-coverage*. https://github.com/sc-forks/solidity-coverage. Accessed: July 30, 2023.

[99] O. Nordbjerg and other contributors. *Foundry Debugger*. https://book.getfoundry.sh/forge/debugger. Accessed: July 30, 2023.

[100] Ethworks sp z o.o. *Waffle*. https://github.com/TrueFiEng/Waffle. Accessed: July 31, 2023.

[101] Ethworks sp z o.o. *Waffle's Mocking Mechanism*. https://ethereum-waffle.readthedocs.io/en/latest/mock-contract.html. Accessed: July 31, 2023.

[102] O. Nordbjerg and other contributors. *Foundry Mocking*. https://book.getfoundry.sh/cheatcodes/mock-call. Accessed: July 31, 2023.

[103] Y. Kissoon and G. Bekaroo. "Detecting vulnerabilities in smart contract within blockchain: a review and comparative analysis of key approaches". In: *2022 3rd International Conference on Next Generation Computing Applications (NextComp)*. IEEE. 2022, pp. 1–6.

[104] O. Nordbjerg and other contributors. *Forge Fuzzer*. https://book.getfoundry.sh/forge/fuzz-testing. Accessed: July 31, 2023.

[105] M. Goldmann. *Resource Management in Docker*. https://goldmann.pl/blog/2014/09/11/resource-management-in-docker. Accessed: August 9, 2023. 2016.

[106] P. Samuel and A. T. Joseph. "Test sequence generation from UML sequence diagrams". In: *2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*. IEEE. 2008, pp. 879–887.

[107] M. H. Ibrahim, M. Sayagh, and A. E. Hassan. "A study of how Docker Compose is used to compose multi-component systems". In: *Empirical Software Engineering* 26 (2021), pp. 1–27.

[108] A. Chauhan. "A review on various aspects of MongoDb databases". In: *Int. J. Eng. Res. Sci. Technol* 8.5 (2019), pp. 90–92.

[109] S. Ahmed and Q. Mahmood. "An authentication based scheme for applications using JSON web token". In: *2019 22nd international multitopic conference (INMIC)*. IEEE. 2019, pp. 1–6.

[110] E. You. *Vue.js*. `https://vuejs.org`. Accessed: October 26, 2023.

[111] M. A. Mohamed, O. G. Altrafi, and M. O. Ismail. "Relational vs. nosql databases: A survey". In: *International Journal of Computer and Information Technology* 3.03 (2014), pp. 598–601.

[112] P. Dias. *Dockerode*. `https://github.com/apocas/dockerode`. Accessed: October 27, 2023.

[113] StrongLoop, IBM, and other contributors. *Express.js*. `https://expressjs.com`. Accessed: October 27, 2023.

[114] J. Xu, Y. Wu, Z. Lu, and T. Wang. "Dockerfile tf smell detection based on dynamic and static analysis methods". In: *2019 ieee 43rd annual computer software and applications conference (compsac)*. Vol. 1. IEEE. 2019, pp. 185–190.

[115] O. Nordbjerg and other contributors. *Foundry Configuration File (foundry.toml)*. `https://book.getfoundry.sh/config`. Accessed: October 28, 2023.

[116] A. Mouat. *Docker*. O'Reilly Japan, Incorporated, 2016.

[117] A. Chaturvedi et al. "Comparison of Different Authentication Techniques and Steps to Implement Robust JWT Authentication". In: *2022 7th International Conference on Communication and Electronics Systems (ICCES)*. IEEE. 2022, pp. 772–779.

[118] J. Jung. *What are Salted Passwords and Password Hashing?* `https://www.okta.com/blog/2019/03/what-are-salted-passwords-and-password-hashing`. Accessed: October 20, 2023. 2021.

[119] Another-D-Mention Software and other contributors. *adm-zip*. `https://github.com/cthackers/adm-zip`. Accessed: October 20, 2023.

[120] N. Q. Uy and V. H. Nam. "A comparison of AMQP and MQTT protocols for Internet of Things". In: *2019 6th NAFOSTED Conference on Information and Computer Science (NICS)*. IEEE. 2019, pp. 292–297.

[121] VMware, Inc. *AMQP 0-9-1 Model Explained*. `https://www.rabbitmq.com/tutorials/amqp-concepts.html`. Accessed: October 23, 2023.

[122] VMware, Inc. *RabbitMQ Clustering Guide*. `https://www.rabbitmq.com/clustering.html`. Accessed: October 23, 2023.

[123] Docker Inc. *Docker Bridge Network Driver*. `https://docs.docker.com/network/drivers/bridge`. Accessed: October 25, 2023.

[124] A. MacGregor. *The Complete Guide to Docker Secrets*. `https://earthly.dev/blog/docker-secrets`. Accessed: October 26, 2023.

[125] J. J. Leider. *Vuetify*. https://vuetifyjs.com/en. Accessed: October 26, 2023.

[126] C. Slingerland. *Horizontal Vs. Vertical Scaling: How Do They Compare?* https://www.cloudzero.com/blog/horizontal-vs-vertical-scaling. Accessed: October 26, 2023.

[127] Google, Rancher Labs, and Cloud Native Computing Foundation. *Kubernetes*. https://kubernetes.io. Accessed: October 26, 2023.

[128] R. Powell. *Docker Swarm vs Kubernetes: how to choose a container orchestration tool*. https://circleci.com/blog/docker-swarm-vs-kubernetes. Accessed: October 26, 2023.

[129] N. Marathe, A. Gandhi, and J. M. Shah. "Docker swarm and kubernetes in cloud computing environment". In: *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*. IEEE. 2019, pp. 179–184.

[130] A. M. Beltre, P. Saha, M. Govindaraju, A. Younge, and R. E. Grant. "Enabling HPC workloads on cloud infrastructure using Kubernetes container orchestration mechanisms". In: *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. IEEE. 2019, pp. 11–20.

[131] K. Chodorow. *Scaling MongoDB: Sharding, Cluster Setup, and Administration*. "O'Reilly Media, Inc.", 2011.

[132] MongoDB, Inc. *MongoDB Atlas*. https://cloud.mongodb.com. Accessed: October 26, 2023.

[133] T. Sharvari and K. Sowmya Nag. "A study on modern messaging systems-kafka, rabbitmq and nats streaming". In: *CoRR abs/1912.03715* (2019).

[134] F5, Inc. *Nginx*. https://www.nginx.com. Accessed: November 9, 2023.

[135] The Apache Software Foundation. *Apache*. https://httpd.apache.org. Accessed: November 9, 2023.

[136] M. Boers. "Designing effective graphs to get your message across". In: *Annals of the rheumatic diseases* 77.6 (2018), pp. 833–839.

[137] N. C. Zakas. *ESLint*. https://eslint.org. Accessed: November 8, 2023.

[138] J. Long et al. *lint-staged*. https://prettier.io. Accessed: November 8, 2023.

[139] Typicode. *Husky*. https://typicode.github.io/husky. Accessed: November 8, 2023.

[140] A. Okonetchnikov. *lint-staged*. https://github.com/lint-staged/lint-staged. Accessed: November 8, 2023.

[141] N. Shtein. *Exit Codes In Containers & Kubernetes – The Complete Guide*. https://komodor.com/learn/exit-codes-in-containers-and-kubernetes-the-complete-guide. Accessed: October 22, 2023.