Automated Unit Testing of Solidity Smart Contracts in an Educational Context

Batuhan Erden

11.12.2023, Master's Thesis in Informatics

Chair of Software Engineering for Business Information Systems (sebis)
Department of Computer Science
School of Computation, Information and Technology (CIT)
Technical University of Munich (TUM)
wwwmatthes.in.tum.de

Outline

1. Motivation
2. Problem Statement
3. Research Questions
4. Comparison of Test Runner Frameworks
5. Developing a Learning Platform
6. Results and Analysis
7. Summary

Motivation

- Rising prevalence of blockchains
- Blockchain integration in educational curricula (e.g., BBSE[1] course at TUM)
- Shift towards decentralized computing
- Ethereum and Solidity smart contracts
- Cruciality of testing smart contracts for vulnerabilities
- Growing focus on smart contract testing in education
- Encouraging student engagement in exercises
  - BBSE statistics: ~800 students registered, yet only a few exercise downloads
  - Enhancing student participation through gamification strategies (leaderboards, *gas* usage tracking)

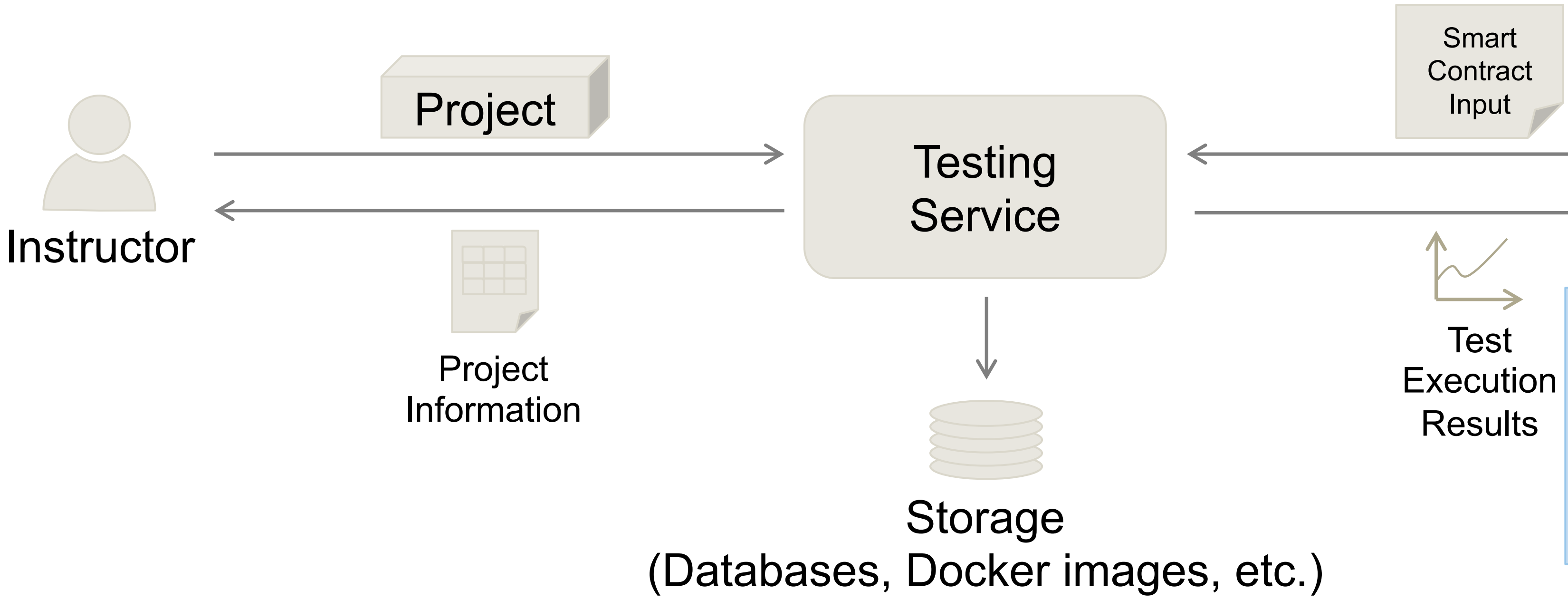
Gas as a Unit of Work. Dannen **[1]** describes *gas* in Ethereum as a unit of work, which quantifies the computational effort required for operations and transactions, where the total fee incurred is calculated by multiplying the total amount of gas used by the price paid for the gas.

[1] *Blockchain-based Systems Engineering*

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

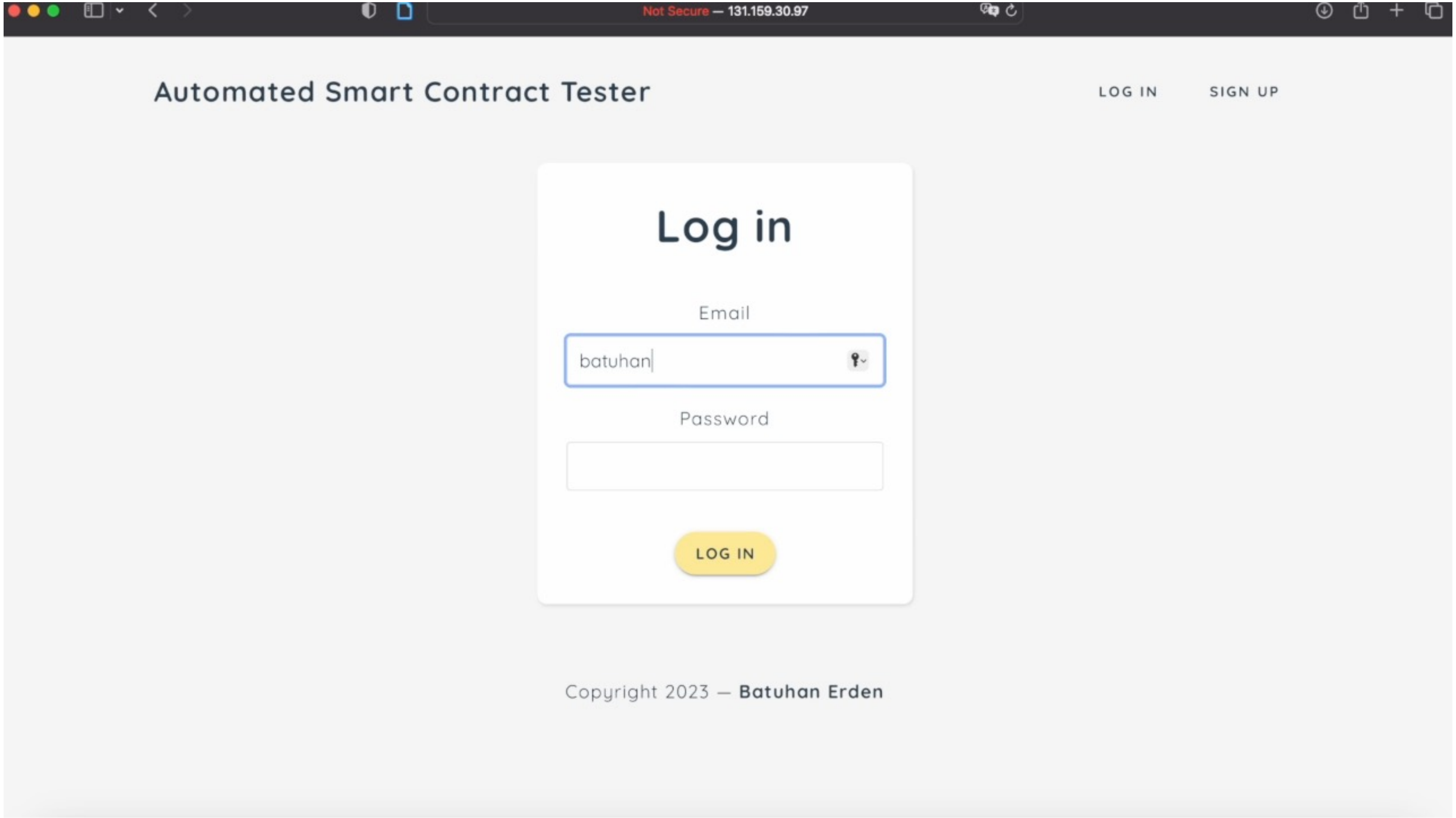
Problem Statement

We claim that this testing service will significantly contribute to the technological developments in educational settings, aiding students in creating more secure and reliable smart contracts before deploying them in critical applications.



- Ability to test their contracts
- Enhanced learning experience
- Elimination of the requirement to create their own tests
- Smooth and hassle-free user experience

Demo


Not Secure — 131.159.30.97
Automated Smart Contract Tester
LOG IN
SIGN UP
Log in
Email
batuhan
Password
LOG IN
Copyright 2023 — Batuhan Erden

Problem Statement – Main Challenges of Smart Contract Testing

- Unlike the execution of traditional programs, limited execution time and resources (e.g., set by metrics like gas usage) are involved.
- A blockchain is simulated.
- The blockchain is immutable; therefore, the deployment of reliable code is crucial.
- Smart contracts can be highly complex due to their self-executing nature.
- The significance of scalability is crucial in educational contexts, especially when the system experiences its highest loads during peak hours.
- Test runner frameworks are required to test smart contracts.

Research Questions

RQ1 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?

- Requirements Engineering
- Use Case Definition

RQ2 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?

- Literature Review
- Comparative Analysis

RQ3 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?

RQ4 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?

- Design
- Implementation
- Testing

Test Runner Frameworks – Most Commonly Used Ones and Overview

Test runner frameworks *are the software frameworks or platforms that facilitate the development, deployment, and testing of smart contracts on blockchain platforms.*

TRUFFLE

Hardhat

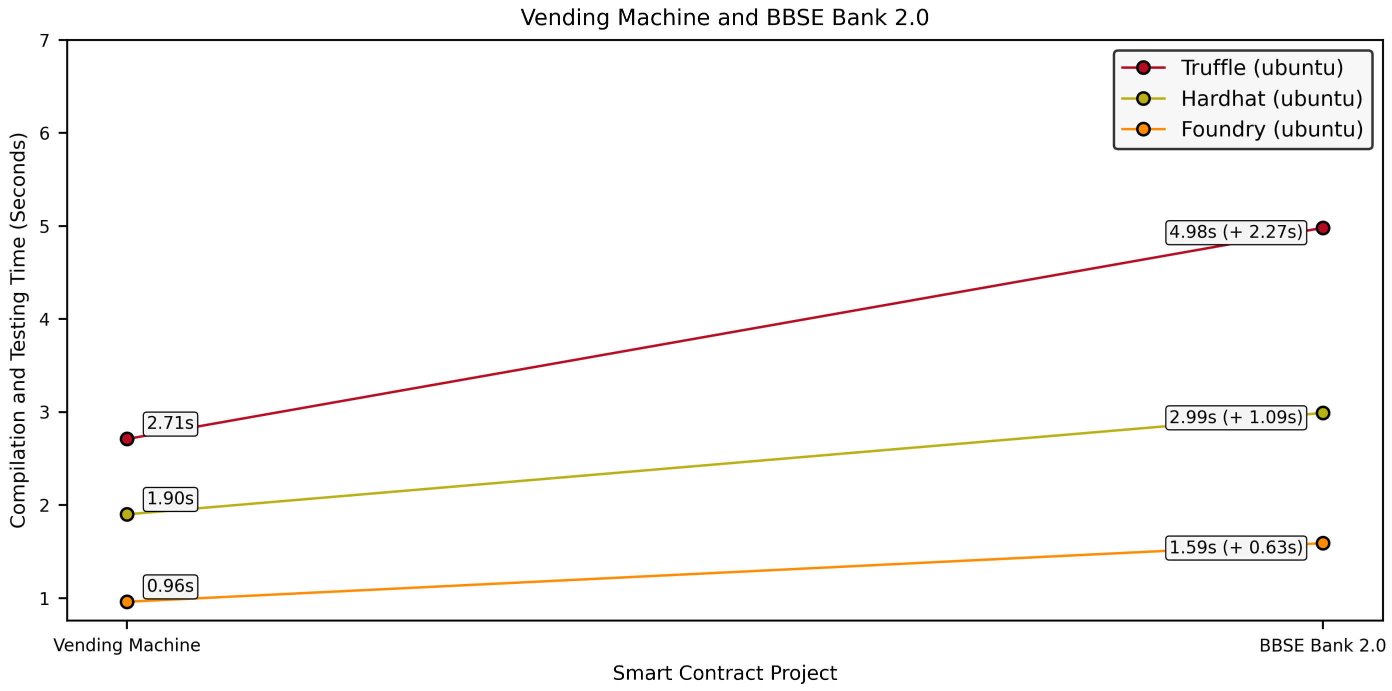
- Truffle was the initial smart contract framework.
- Hardhat followed later and rose to become a major competitor.
- Foundry is emerging as a rising star, distinguished by its remarkably swift testing speed.

Test Runner Frameworks – Key Comparative Factors

- Development experience (installation, setup, and documentation)
- Testing capabilities
- Test result reporting capabilities (e.g., accurate gas consumption results)
- **Performance**
- **Containerization capabilities**

Test Runner Frameworks – Performance Results

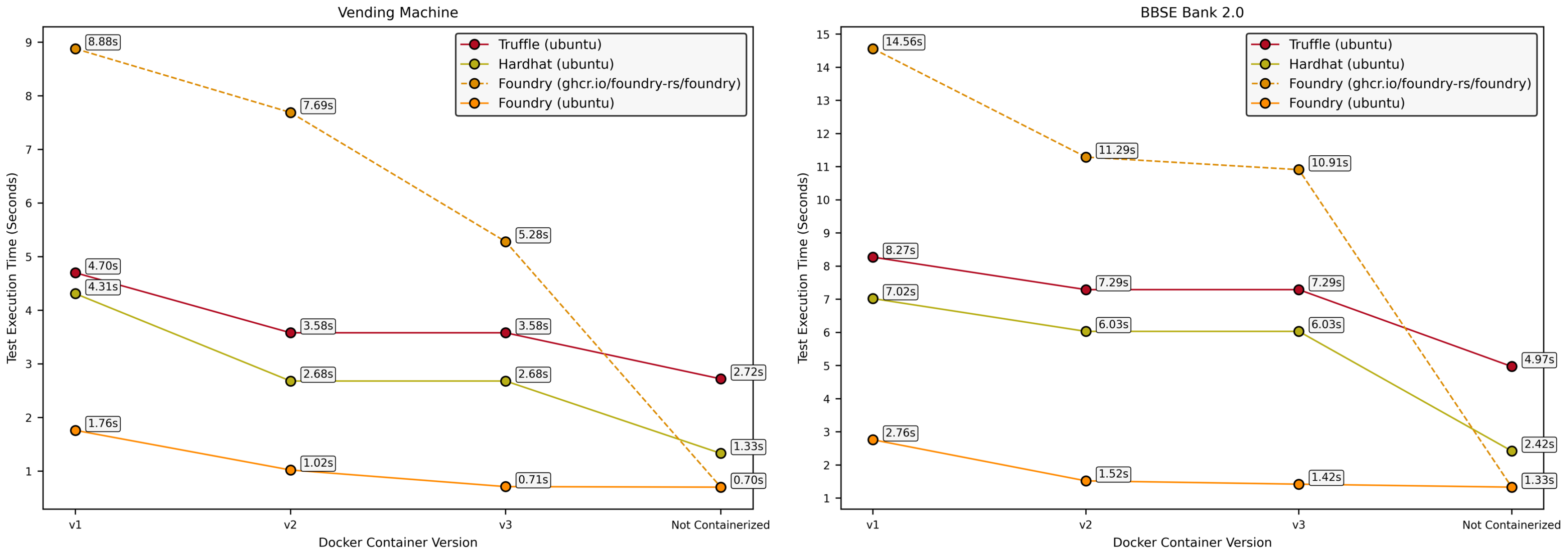
Compilation & Test Execution Times of Frameworks



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

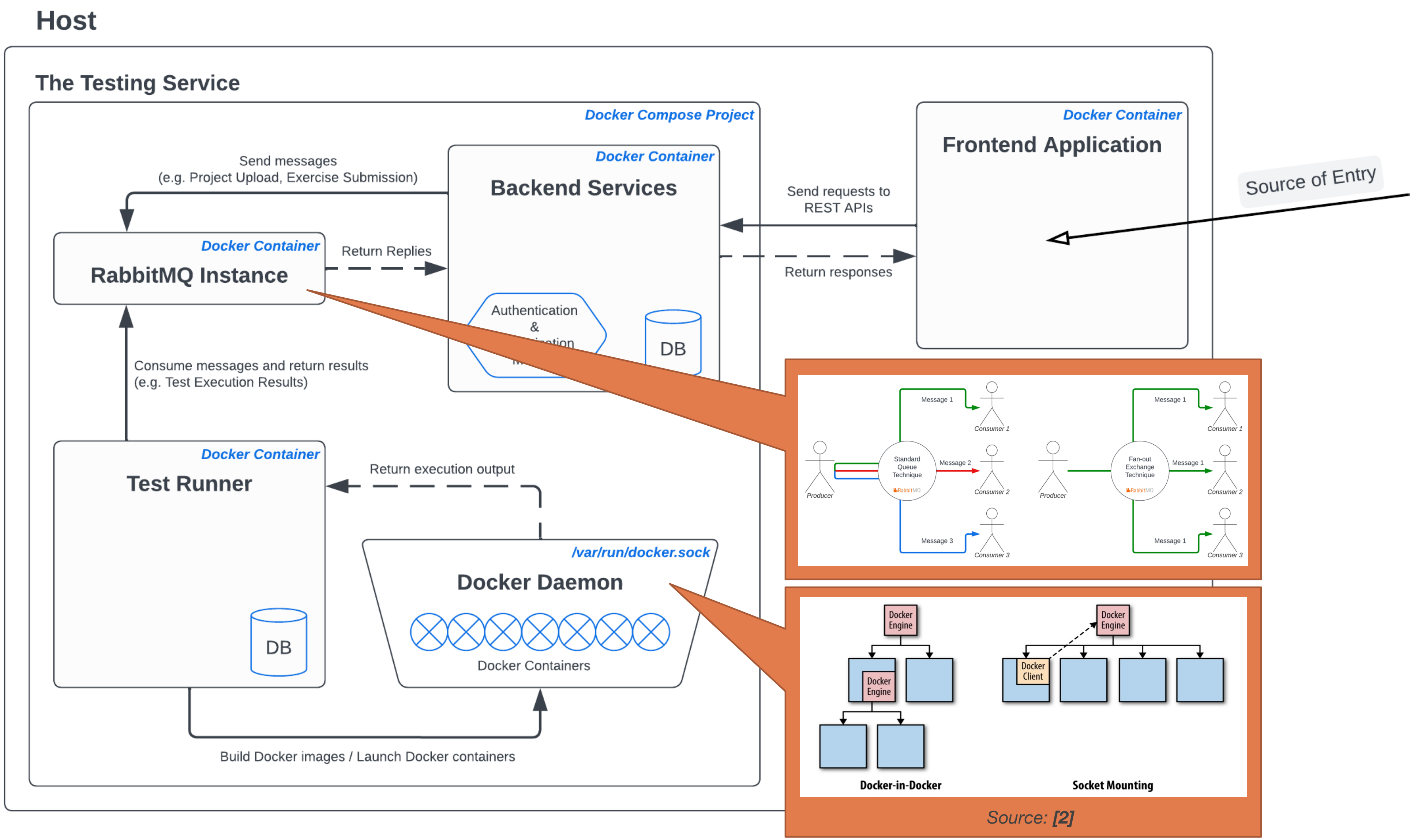
Test Runner Frameworks – Performance Results (Containerized)

Test Execution Times with Containerization Versions



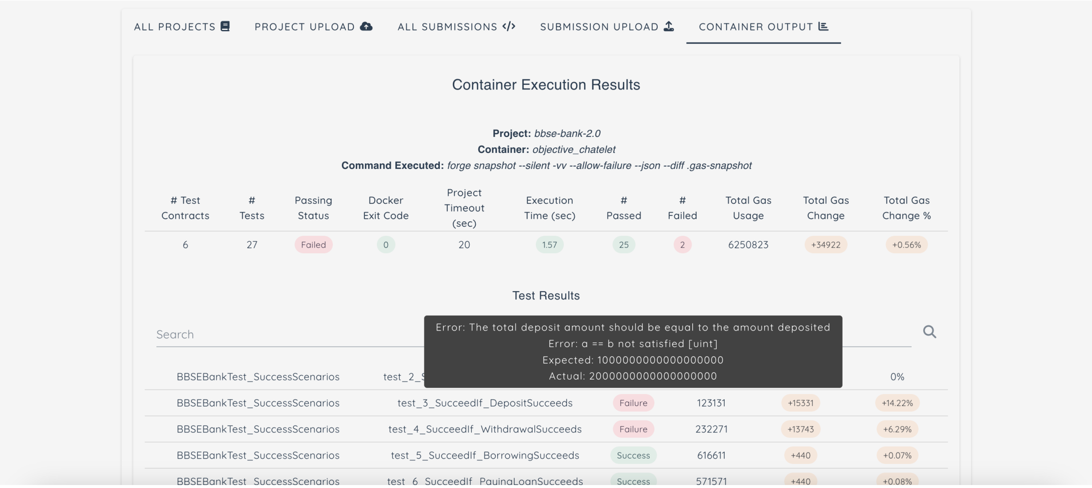
Project	Framework	Containerization Version v1	v2	v3	Not Dockerized
Vending Machine	Truffle (ubuntu image)	4.70s	3.58s	-	2.72s
	Hardhat (ubuntu image)	4.31s	2.68s	-	1.33s
	Foundry (ubuntu image)	1.76s	1.02s	0.71s	0.70s
	Foundry (foundry-rs image)	8.88s	7.69s	5.28s	0.70s
BBSE Bank 2.0	Truffle (ubuntu image)	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

Developing a Learning Platform

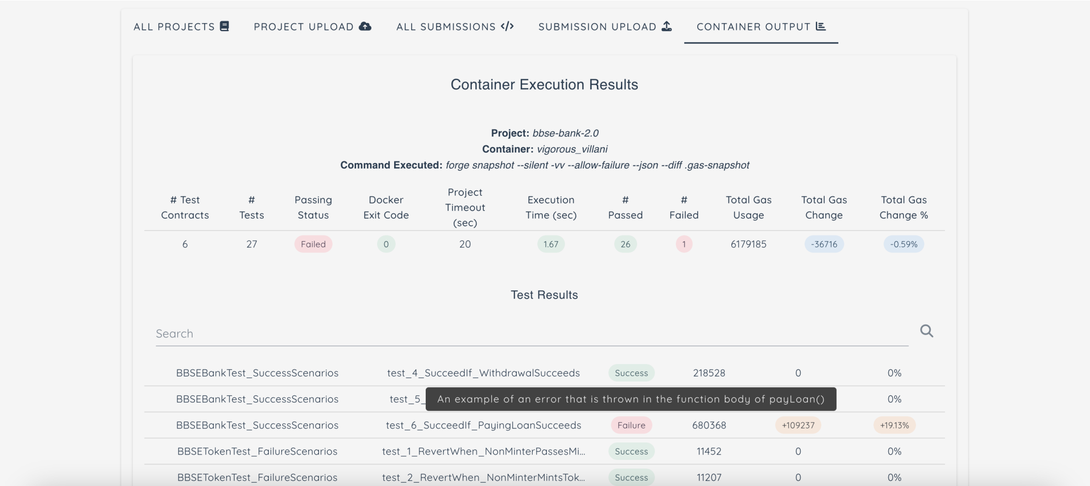


[2] A. Mouat. Docker. O'Reilly Japan, Incorporated, 2016.

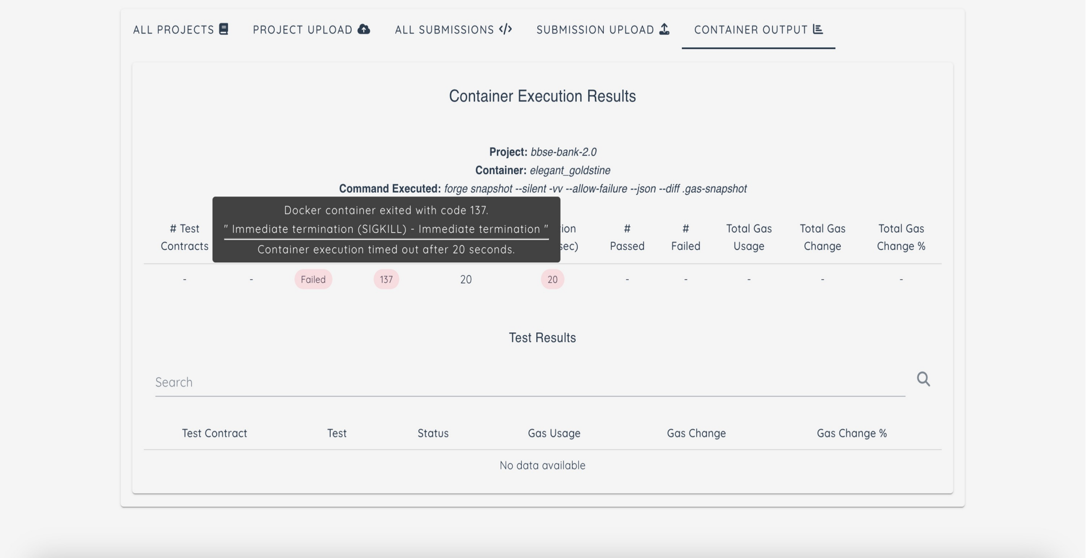
Results and Analysis – Security and Stability



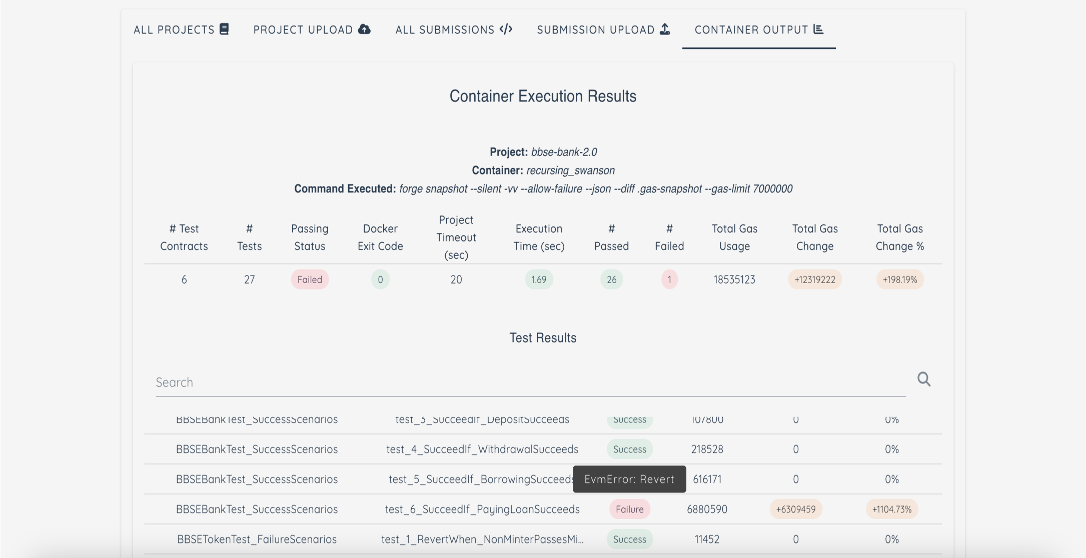
Test Execution with Failing Tests



Test Execution with Contract Error



Test Execution Timeout



Test Execution with Excessive Gas Usage

Results and Analysis – Efficiency and Performance

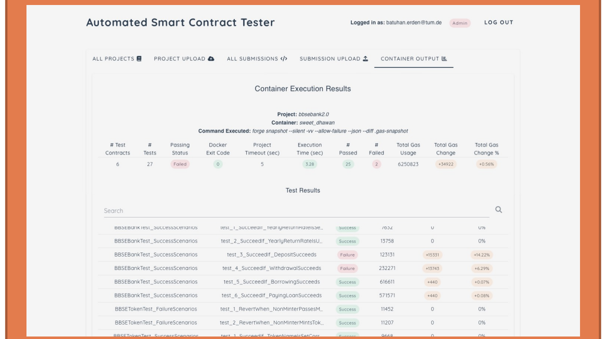
Total Processing Time for Simultaneous Execution of All Submissions

Number of Simultaneous Submissions	Hardware Setting	
	10-core CPU [2]	2-core CPU [3]
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

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

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

Summary



RQ1

What are the requirements for educational unit testing?

Empowering student development through tailored exercises within the core use case.

RQ2

What is the status quo in automated smart contract testing?

- Contrasting smart contract testing against traditional program testing to highlight differences.
- Identification of Foundry as the optimal framework for smart contract testing, based on various key comparative factors:
  - Usability
  - Development experience
  - Features
  - Performance
  - Containerization capabilities

RQ3 & RQ4

- What do we have to consider regarding security and stability when using a testing tool in a way that is not entirely intended?
- How can a learning platform giving feedback through automated smart contract unit testing be developed?

Development of a robust testing service aligned with the core use case requirements:

- Ensuring security
- Guaranteeing stability
- Optimizing efficiency
- Providing horizontal scalability

Batuhan Erden
batuhan.erden@tum.de

Technical University of Munich (TUM)
TUM School of CIT
Department of Computer Science (CS)
Chair of Software Engineering for Business
Information Systems (sebis)

Boltzmannstraße 3
85748 Garching bei München

Background – A Simple Smart Contract

A Simple Smart Contract to Deposit/Withdraw Funds

```
pragma solidity ^0.4.17;

contract SimpleDeposit {

    mapping (address => uint) balances;

    event LogDepositMade(address from, uint amount);

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

    function deposit() public payable minAmount(1 ether) {
        balances[msg.sender] += msg.value;
        LogDepositMade(msg.sender, msg.value);
    }

    function getBalance() public view returns (uint balance) {
        return balances[msg.sender];
    }

    function withdraw(uint amount) public {
        if (balances[msg.sender] >= amount) {
            balances[msg.sender] -= amount;
            msg.sender.transfer(amount);
        }
    }
}
```

*Source: **[3]***

[3] 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.

Background – An Example of a Solidity Test Case

Example Solidity Test Case – Verifying Successful Deposit

```
// 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");
}
```

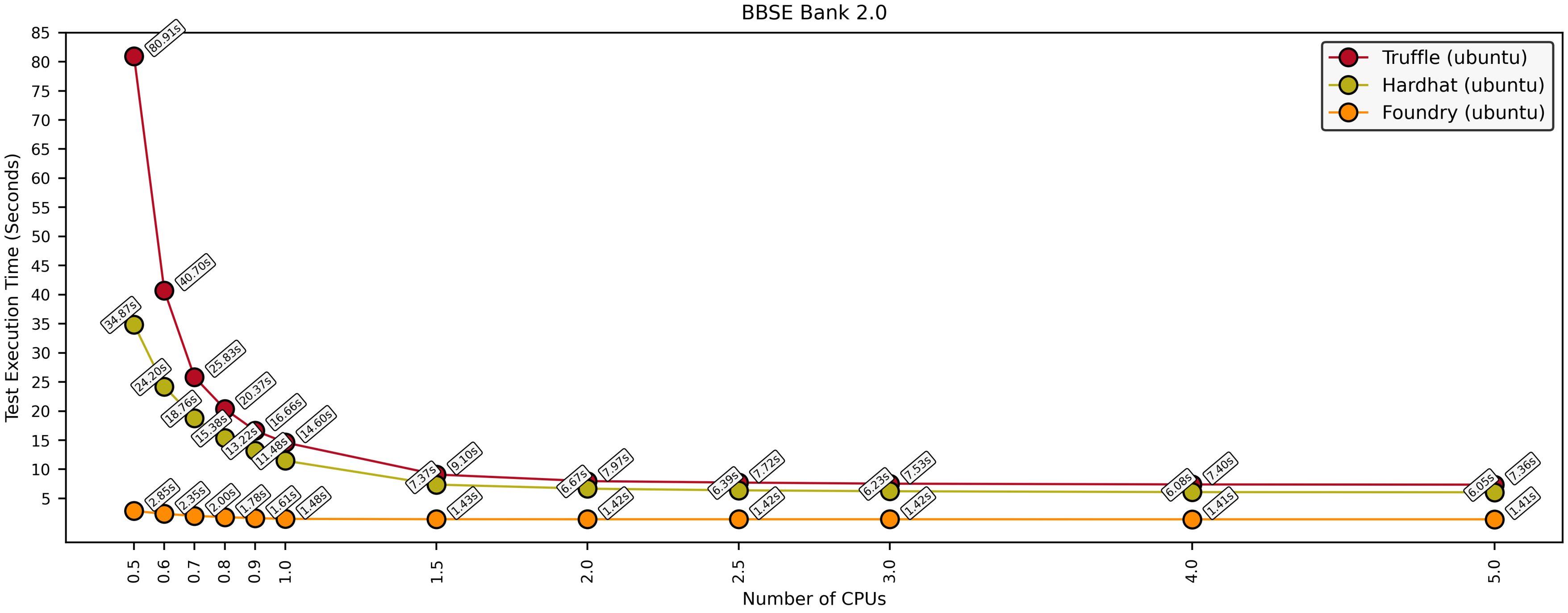
Test Runner Frameworks – Containerization Capabilities (Image Sizes)

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

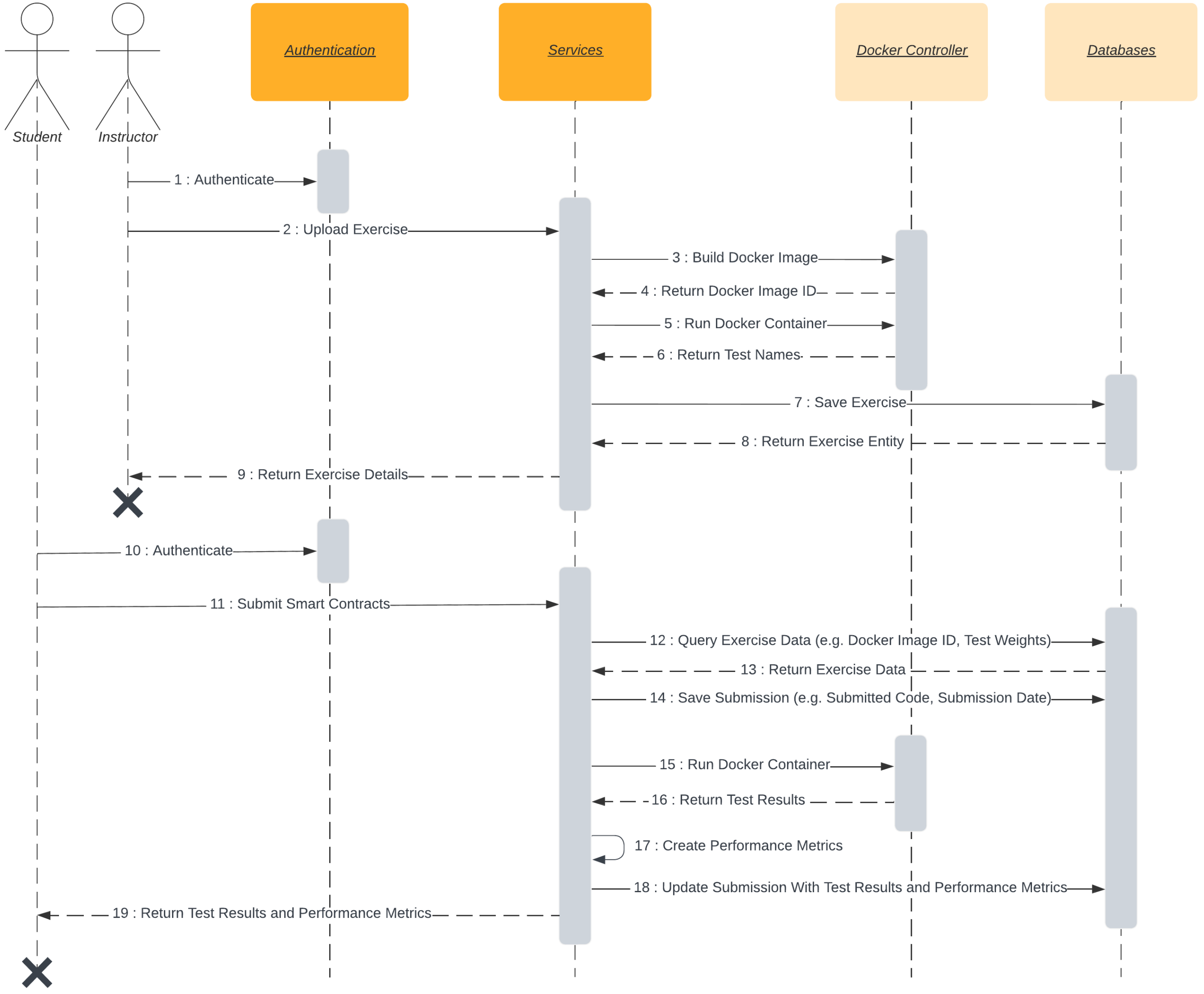
Test Runner Frameworks – Containerization Capabilities (Scalability)

BBSE Bank 2.0 - Test Execution Times with CPU Core Counts



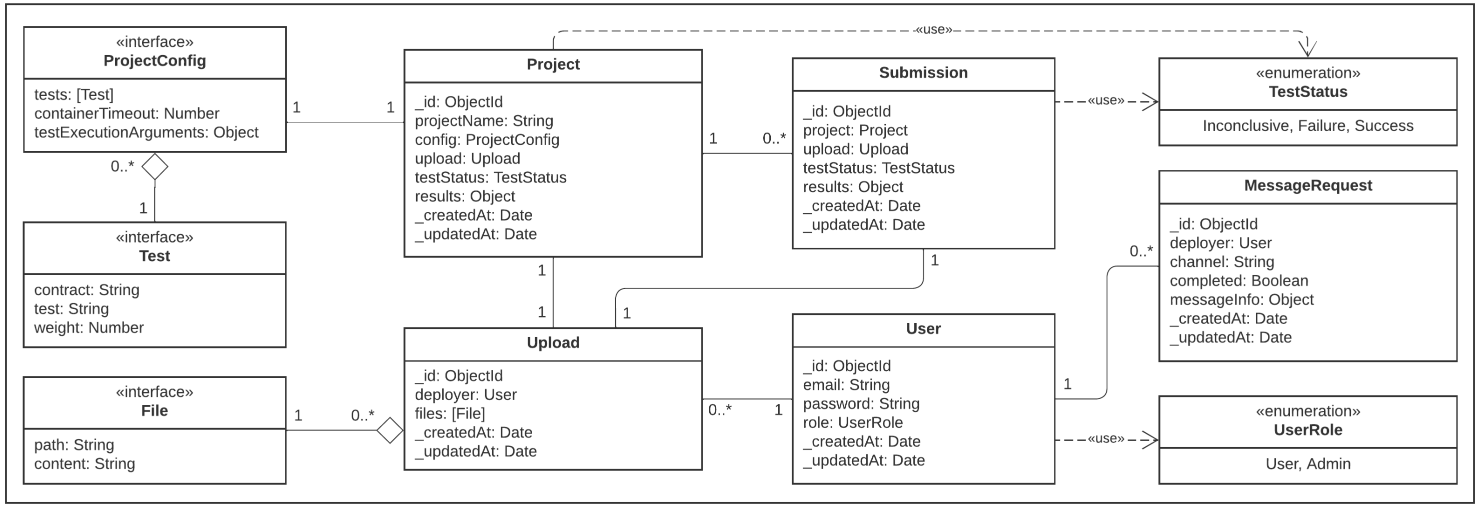
Framework	CPU Core Count									
	0.5	0.6	0.7	0.8	0.9	1.0	1.5	2.0	2.5	3.0
Truffle (v3)	80.91s	40.70s	25.83s	20.37s	16.66s	14.60s	9.10s	7.97s	7.72s	7.53s
Hardhat (v3)	34.87s	24.20s	18.76s	15.38s	13.22s	11.48s	7.37s	6.67s	6.39s	6.23s
Foundry (v3)	2.85s	2.35s	2.00s	1.78s	1.61s	1.48s	1.43s	1.42s	1.42s	1.42s

Developing a Learning Platform – High-Level Flow

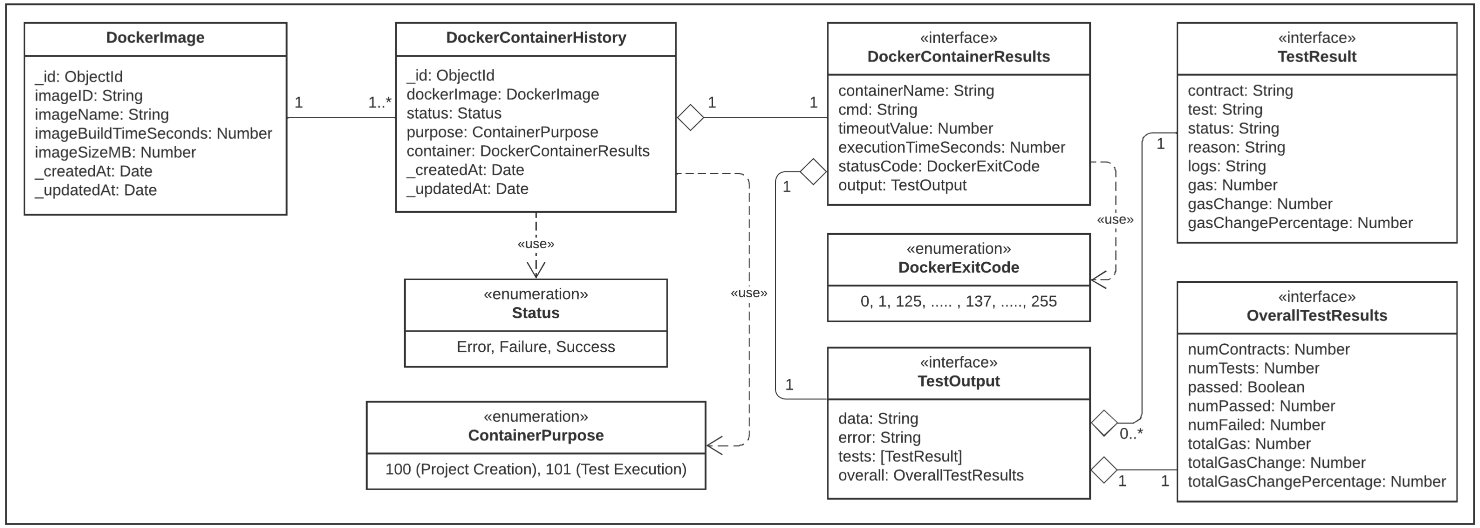


Developing a Learning Platform – Data Model

Backend Services



Test Runner



Developing a Learning Platform – Dockerfile for Project Image Creation

```
# 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"]
```

Developing a Learning Platform – Horizontal Scalability