POLITECNICO DI TORINO

Master's Degree in COMPUTER ENGINEERING

Master's Degree Thesis

Smart Contract Analysis and Visualization Software

Supervisors

Prof. Valentina GATTESCHI

Eng. Emanuele Antonio NAPOLI

Candidate

Lorenzo GANGEMI

December 2024

Summary

The thesis presents the design and development of a software for simplifying the reading, editing and security analysis of smart contracts. The system is based on a combination of atomic microservices and a modern frontend. The tool provides an easy to use environment, error resistant and ready for people with limited skills in programming. The main part of the project is a network of 3 services, each one responsible for a specific function. The Codec takes care of transforming the smart contract code into a JSON structure, and vice versa. The Auditor executes a security analysis with Slither and returns critical information that can help the user to avoid common vulnerabilities. The Assistant relies on Large Language Models (LLMs), specifically through integration with ChatGPT APIs, to generate descriptions and highlight the functional relationships within the elements. The Backend For Frontend pattern was used for managing communication and data flow between services and the frontend. This one has been developed in Flutter for increasing cross platform and device compatibility. It takes care of visually representing the elaboration results inside a navigable grid of elements, where each of them can be investigated and edited. While the tool successfully demonstrates the potential for integrating multiple technologies to simplify smart contract development, it highlights areas for improvement. Particularly prone to this is the frontend, which can be further improved to fully abstract the code from the user. Overall, this thesis contributes to the field by showcasing integration of microservices, LLMs and visualization techniques in modern technologies.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

The rapid evolution of blockchain technology and the fast adoption of decentralized application caused a significant increase in smart contract development. These self-executing software are integral part to their blockchain ecosystem. They allow automated transactions and reduce the need for intermediaries. However, the complexity of smart contract development represent a considerable challenge, especially for users who are not familiar to the software engineer world. Errors in a common piece of software can be eventually fixed, leading to a software patch. The same can't be done for smart contracts, and usually those errors lead to significant financial losses or security vulnerabilities. It is crucial to provide tools that support the development process, reduce the risk of mistakes and improve the understanding of relations and functionalities in the code-base.

This thesis aims to create a system that facilitates the process of reading, editing and analyzing smart contracts. The tools is designed to be accessible to a large user base, including those with less technical expertise, by providing a user-friendly graphical interface, relying on a network of micro services for intensive tasks and make use Large Language Model (LLMs). By abstracting the direct interaction with the code, the tool improves the overall security and reliability of decentralized applications.

1.2 Objectives

To achieve the result, the tool makes use of three key features. A Graphical representation of the smart contract, the integration with a Large Language Model and a security analysis method. Each of these components plays a role in addressing the challenges identified in the problem statement.

1.2.1 Graphical Representation

This feature aims to simplify smart contract interactions by abstracting the complex code into a visual format. This approach reduces the load associated with understanding and editing, particularly for users who may not be proficient in programming.

A micro services have been developed to take care of splitting the contract in its smallest parts. Those parts are then assembled into a data structure which can be interpreted by other services or by the front end. No information is lost in the process, and this allows the structure to be converted back into code if needed. The conversion is still possible even after modification to the structure.

The front end is able to parse the result of this elaboration into a nested object. This is later represented into a grid, where every element takes a graphical form. The user can then access the element details and modify them. Furthermore, the results of those changes, can be converted back into smart contract code, for later usage or distribution.

This not only helps in identifying potential issues or inefficiencies within the contract but also facilitates easier modification and debugging. The graphical interface provides a more accessible entry point for less experienced developers, by broadening the user base and encouraging the adoption of smart contract technology.

1.2.2 LLM integration

Another critical component in reaching the tool's objectives is the integration of a Large Language Model. Such technology is employed by the project to generate descriptions of the entities identified within the smart contract code. This feature is particularly valuable for users who may struggle to understand the contract logic and interactions.

By automatically producing human-readable explanations and identifying functional relationship within the contract, the LLM integration improves the user's comprehension of the code and reduces the risk of unwanted implementations. This can also provide a big aid in debugging and refining the contract to keep it aligned with the business logic and operational requirements.

The thesis makes use of ChatGPT APIs to implement this feature. The APIs are accessible through an API key which needs to be provided by the user through the front end interface. The key is later received and used by a specific micro service that integrates the necessary logic.

The interaction with the model can lead to unexpected results. To align them with a defined structure, the projects relies on the use of Function Calls. This tool allows to force the LLM to provide the answer in a specified JSON format, instead of providing plain text.

Once the expected results are obtained, the element description are nested in the same structure that represents the smart contract code elements, and the functional relations are stored in a specific array which is also sent back to the BFF.

1.2.3 Security Analysis

Given the irreversible nature of blockchain transactions, ensuring the security of smart contracts is a topic which is important to take care of. In classic software development, not reducing the risks can lead to damage and financial losses, but the problems can usually be tackled by releasing a patch containing the fix for the cause. Once a smart contract is deployed on the blockchain, if it behaves in an unexpected way, there's no way to directly replace the deployed code to fix the problem.

The project incorporates a static security analysis feature, which makes use of Slither to automatically detect and report vulnerabilities in the code. This component is essential for preventing common issues.

A specific micro service is responsible for receiving the plain smart contract code and the structure, performing the analysis and integrating the results within the structure. The vulnerabilities discovered are then graphically shown to the user, who can promptly take care of them.

1.3 Thesis structure

Each chapter is divided in different sections which are going deeper in details.

- Chapter 2 - State of the art: The first section explores Solidity, the programming language used for building smart contracts on the Ethereum blockchain, and the classic development environment. It starts by analyzing its basic elements and the aspects that can make it difficult to work with. The chapter proceeds with an analysis of existing low code or no code platforms that are trying to abstract the difficulties. Each of them provides different approaches to the problem. It then move to focus on the technologies and tools used in the project. It explores the the language recognition tool used to split the contract in elements, the current state of the ChatGPT LLM, and the security analysis tool Slither.

- Chapter 3 - System Architecture and Design: The first part of this chapter explains the decisions taken while building the whole system. It focus on the general design, the way communication protocols are integrated, and the reasons behind the programming language choices. The following sections are much more focused on every single element of the architecture.

1. The Codec, which takes care of encoding and decoding the smart contract. Details on the language recognition and the obtained JSON structure are provided.
2. The Auditor service, with explanation on how it runs Slither and how it gather the analysis results.
3. The Assistant, with focus on the use of ChatGPT and function calling.
4. The BFF, the central point in the architecture, responsible for communication between the front end and the services.

- Chapter 4 - Front end: This chapter starts with a section on the Data Layer, explaining how the elaboration results are interpreted. It then moves to the visual interface and the grid representation of elements.
- Chapter 5 - Results: Within the results chapter the thesis display what was achieved and provides examples of the software utilization with real smart contracts.
- Chapter 6 - Conclusions: In the final part, the results are compared with the initial problem statement and possible future development are identified.

Chapter 2

State of the art

2.1 Smart Contract Development

2.1.1 Definition and Overview

The term "Smart Contract" was coined by Nick Szabo in 1996. With this term, he refers to an actual contract enforced by physical property instead of law. He envisioned them as digital protocols that could facilitate, verify or even enforce negotiations and execution of agreements.

Smart contracts are self-executing agreements with the terms written and enforced by software. These entities are able to execute the agreed-upon rules and obligations without the need for intermediaries.

The advent of blockchain technology, specifically with the launch of Ethereum in 2015, turned this concept into a practical reality.

A blockchain can be defined as a decentralized and distributed ledger that records transactions across a network of computers.

The blockchain itself is composed of nodes, which are instances of the software that maintains a copy of the ledger, validating transaction and ensuring the network integrity. Each node operates independently, but works in concert with other nodes to achieve decentralisation and distribution.

Unlike traditional centralized systems, where a single server or a group of servers controls the entire network, these nodes collectively manage and govern the blockchain.

When a transaction is initiated, it is broadcast to all the nodes in the network. Each of them verifies the transaction according to the network consensus rules. Once the majority have validated the transaction, it is bundled into a block with other transactions. No central authority can alter the transaction history, but it this the collective agreement of nodes to maintain the integrity. In summary, a blockchain node is a critical element that represent the decentralized nature of the

blockchain.

Deploying

Deploying a smart contract is a complex task that transforms the contract from a piece of code into an immutable entity that is stored and operates on the network. The process involves several steps:

1. **Writing**: The code is first written by the engineer, typically in a programming language designated for that, such as Solidity for Ethereum. The code defines rules, functions, data structures and all that represent the contract's expected behavior.

2. **Compiling**: The Ethereum Virtual Machine (EVM) or the equivalent runtime environment of the chosen blockchain, can't directly understand the high level code. The contract must then be converted into byte code by a compiler. The byte code is the actual code that will be stored on the blockchain.

3. **Broadcasting**: After compiling, the developer initiates a deployment transaction to the blockchain network. This transaction contains the byte code and is sent from the developer's wallet to the blockchain.

4. **Validating**: The network nodes starts to pick up the transaction. They validate it to ensure it complies with the network's rules. After this process is completed, the transaction is included into a new block which is later appended to the blockchain. This makes the smart contract permanently recorded and globally accessible.

When these steps are completed, the smart contract is successfully included in a block on the blockchain. Derived from the sender's address and the transaction's nonce, a unique address it is assigned to it. This serves as a permanent identifier, allowing the users and other contracts to interact with it.

Deploying a smart contract is a multi-step process that transform written code into an entity on the blockchain, embedding all the logic into immutable rules that can be executed as needed.

Interacting

Interaction with a smart contract are executed though transactions that trigger a contract's function, starting a chain of activities.

The responsibility of its execution lies with the user initiating the transaction. The costs of this operation, known as "gas fees", are calculated based on the computation resources required to execute the code. The gas fee compensates the

network's miners or validators for processing and validating the transactions. This act as an incentive and reward for keeping the network working and secure. Trying to execute a transaction that trigger smart contract code still has a cost, therefore if the gas provided is not sufficient, the transaction fails but the user still loses the fees associated with the attempted execution.

Here's how an interaction with a smart contract tipically works:

- **Calling functions**: The interaction starts by sending a transaction to the contract's unique address on the blockchain. Within the transaction, the user can specify which function to invoke. The transaction can result in changes to the contract's internal state, such as updating variables or triggering more actions. The most common outcome consist of transferring tokens or issuing events.
- **Reading data**: Other than calling functions that alter the contract's state, the user can access and read the data stored. Many contracts expose "pure" functions that allow this operation. These read-only functions do not require a a transaction to be executed, therefore they do not have any gas fee cost.
- **Interacting with other contracts**: When executing smart contract code, this entity behaves as all the other entities on the blockchain and nothing is stopping the contract to execute transaction which are pointed to another smart contract. This is a feature that enables powerful interactions within the ecosystem, allowing for ore complex systems to exist, such as decentralized finance protocols (DeFi).
- **Events**: A contract can emit events during its execution. These objects are recorded in the blockchain's log data and can be monitored by external applications or users. Developers can setup listeners to catch specific events and react to those. An example can be the completion of a transaction or the triggering of a specific condition. This allows a better and clearer interaction with the contract.

Interacting with a deployed smart contract involves sending transactions to its address, reading data and responding to events. Direct interactions require technical knowledge, but user interfaces and tools have been developed to make this complex interactions more accessible.

2.1.2 Solidity

As blockchain technology evolved, particularly on platforms like Ethereum, there was a need for a specialized programming language that could efficiently handle the handle all the step previously indicated. Solidity was build to meet this need. Before,

creating smart contracts was a complex error-prone process that was often requiring developers to write low level code or use general-purpose existing languages, that were not optimized for the blockchain environment. These approaches were also prone to errors and security vulnerabilities, given the decentralized nature of the network which requires immutability and trustless execution.

Solidity is a smart contract tailored, high-level, statically-typed programming language. It's syntax is similar to general-purpose languages such as JavaScript, making it easier to approach by developers familiar with modern web development. Solidity takes care of abstracting much of the complexity around blockchain interactions, allowing developers to focus on writing contract logic rather than dealing with the blockchain mechanics.

Here's some key problems that Solidity solves:

- **Ease of use**: It simplifies the development process by providing clear and structured ways to define the behaviour of contracts. The code is concise, easier to read, maintain and audit.
- **Security**: It includes features to help avoid common issues, such as reentrancy attacks and integer overflows. The language supports the use of modifiers and exposes visibility specifiers, which control how functions and state variables can be read or modified.
- **Deterministic**: It is crucial that smart contracts execute consistently across all the nodes. Solidity grants its deterministic nature, producing the same output regardless of where and when it is executed.

Main components of a Solidity Contract

Several key components works together to define the contract's structure, behaviour and interactions. Understanding them is crucial for writing effective code.

- **Pragma Directive**: Every Solidity file has to begin with a pragma directive which specifies the version of the Solidity compiler that should be used to compile it. This to prevent compatibility issues between different versions of the compiler.

```
pragma solidity ^0.8.26;
```

- **Contract Declaration**: This component represent the core of the code by declaring the actual contract. A contract in solidity is analogous to a class in object-oriented programming. It contains state variables, methods, events and can be instantiated.

```
contract ExampleContract{
    ...
}
```

- **State Variables**: State variables are the way a contract can store data that is persistent and remains on the blockchain. These elements represents the contract state.

```
address public owner;
uint256 public totalSupply;
bool public paused;
```

- **Functions**: Define the behaviour of the contract. They are block of code that execute specific logic, such as updating a state variable. Functions are usually paired with a visibility indicator and often with modifiers.

```
function updateContractSettings() public onlyOwner {
    ...
}
```

- **Constructor**: The constructor is a special function which is executed once when the contract is deployed. It is used to initialize the state or set up initial conditions for the contract.

```
constructor(uint initialSupply) {
    totalSupply = initialSupply;
}
```

- **Modifiers**: Used to modify the behaviour of function by enforcing conditions. They are usually used for verifying access to a certain functions based on some criteria.

```
modifier onlyOwner() {
    require(msg.sender == owner, "Not the owner");
    _;
}
```

- **Events**: Events are used to log information on the blockchain. This allows smart contracts to communicate with the external world, as those signals can be seen by external applications.

```
event Transfer(address indexed from,
                address indexed to, uint amount);
```

- **Structs and Enums**: Custom data types which represent a group related set of variable. A set of named constants when talking about Enums.

```
struct User {
    address addr;
    uint balance;
}

enum Status { Pending, Completed, Cancelled }
```

- **Inheritance**: Solidity supports inheritance by allowing contracts to inherit properties and functions from other contracts. This feature promotes code reuse and modularity.

```
contract MyToken is ERC20 {
    // Inherited functionalities from ERC20
}
```

- **Libraries**: Similar to contracts, libraries are meant to collect reusable code. They cannot hold state or be deployed, but they can linked to another contract.

```
library SafeMath {
    function add(uint a, uint b) internal pure returns (uint)
    {
        uint c = a + b;
        require(c >= a, "Addition overflow");
        return c;
    }
}
```

2.1.3 Frameworks

As the complexity and adoption of this technology have grown, the need for specialized development environment has become apparent. Frameworks guides the user development process, reducing the errors rate and proving tools that simplify tasks such as testing, deployment and debugging. Without them, developers would need to manage multiple details manually, from compiling to ensuring best practices are followed. Frameworks are another time an abstraction over many of these tasks, allowing developers to focus more on the logic and functionality of the contract, rather the underlying complexity. They often include utilities specifically made to solve each key problem.

Foundry

Foundry is a modern development framework designed specifically for Ethereum smart contracts. It has gained popularity among developers and it now covers the whole development life cycle.

It is known for its compilation and testing speed. This usually results in shorter feedback loops during development, allowing rapid iteration and debugging.

Another notable feature is its integrated fuzz testing capabilities. Fuzz testing consist of generating random inputs to test the smart contract risk of falling in unexpected behaviours or edge cases. This helps in identifying potential security issues before deployment.

This framework allows the developer to fork any Ethereum network in the local environment, allowing local testing of contracts without the need to interact with an online running blockchain.

Foundry represent a robust and feature rich framework that extends the capabilities of the developer making it faster and reducing the risk of errors.

Truffle

Another well known framework is the Truffle suite. It provides a set of tools that support the entire development process, from writing and testing code, to deploying contracts on various Ethereum networks. It is valued for its ease of use and extensive ecosystem. It offers a powerful development environment by integrating with smart contract compilation, linking and deployment.

One of Truffle's main features is the built-in support for automated testing. It allows developers to write and execute tests for smart contracts using familiar libraries like Mocha and Chai. This increase the test coverage and ensures correctness and security.

Another noticeable feature is its network management capabilities. Truffle can

manage deployments across many different Ethereum network, from local development one, to public testnets and the mainnet. This includes direct integration with Infura.

Truffle comes with other tools, such as Ganache for local blockchain simulations or Drizzle for front end development. This improves the overall development experience by providing a full stack solution for DApp development.

2.2 Low-code/No-Code Tools

Low and No code platforms are software development environments that allow the user to build applications with minimal or zero manual coding. These usually provide an intuitive interface, with dran-and-drop features or ready to use components.

These tools provide a way to accelerate the development process by reducing the amount of manual coding required. They are usually composed by a range of tools and templates to abstract away the complex logics. For developers it means they can focus more on logic and design, but these platforms can be useful also for people with less coding experience.

Key features include:

- **Faster development cycles**: By automating the coding process, developers can build application much more quickly, which is especially important when prototyping in fast-paced environments.
- **Ease of Use**: The visual interfaces and pre-built components make these platforms accessible to a wider range of users, including those who may not have a background in software development.
- **Integration Capabilities**: Many of these tools come with built in connectors to other popular services or APIs.

The following analyses a list of Low-code and No-code platforms which offers blockchain and smart contract related features. Some of the tools listed here, are extracted from "Blockchain Application Development Using Model-Driven Engineering and Low-Code Platforms: A Survey" [1].

2.2.1 DappBuilder.io

DappBuilder[2] is a no-code platform defined as a marketplace for smart contracts and blockchain applications. It stands out for its user friendly interface and the guided app-building process.

It allows the user to choose between dApps and Smart Contract. The main difference is that a dApps features also client code generated from the smart contract Application Binary Interface.

DappBuilder requires the user to login with a personal account and to access with a web browser extension wallet, like Metamask[3].

Once the user select a category, it can choose between different available templates. The following step allows the user to fill a form with data, which are later integrated in the code. Once the form is filled, the front end interface is also generated. This gives the user a better idea of the final purpose.

When the dApp is ready, it can be deployed on the blockchain. The choice is between the main Ethereum net, and the Rinkeby testnet. At this moment, the Rinkeby testnet is no longer maintained, therefore the user can't actually deploy there.

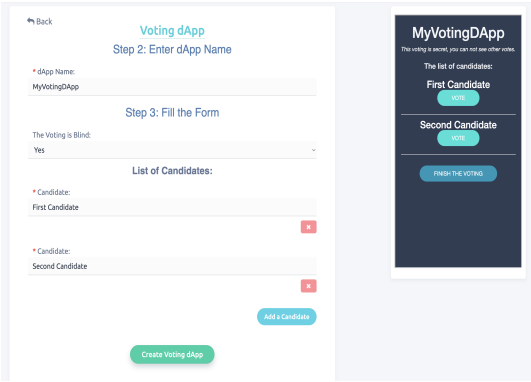

Figure 2.1: DappBuilder - Smart contract editor

Strong Features:

- **User Friendly**: DappBuilder is designed with accessibility in mind, offering a relative simplified interface and relying on templates to guide the user. It has a low entry barrier and it enables a broad range of individuals to engage with decentralised apps creation.

- **Templates**: The platform makes use of pre-built templates to cover common use cases, like voting system, lotteries or multi signature wallets. This feature is particularly valuable for accelerating the development process and ensuring best practices.

- **Client code generation**: DappBuilder offers client side code generation which is build starting from the smart contract abi file. This is especially useful for streamlining the development process and reducing the amount of

manual coding. It makes it easier to create fully functional prototypes in a short amount of time.

2.2.2 DamlHub

DamlHub[4] is a platform for building full decentralised application in the Daml Language[5]. Daml is an high level language used for building workflows and executing verified transactions to be inserted into a DAG, Directed acyclic graph, inside a blockchain node. All of this is build on the Canton blockchain.

DamlHub requires to login into the platform. It then provides an interfaces similar to the DappBuilder one, where the user can choose between templates to start with. After the template selection, more parameters become available, and the user can customise the application behaviour. When the editing is completed, the smart contract can be deployed to the Canton network. The client code ready for being used is generate too.

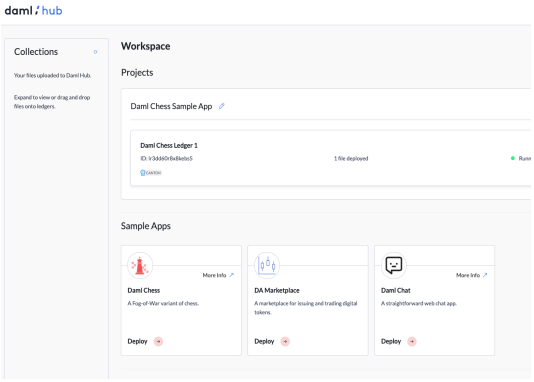

Figure 2.2: DamlHub - Template selection interface

The platform presents logics similar to DappBuilder, but those are applied to a less known blockchain, for more specific use cases.
Strong Features:

- **Templates and GitHub integration**: The strongest feature of DamlHub is the vast amount of choice in the template selection. It also make use of GitHub integrations for both saving the current work before deploying it, or importing other templates.

2.2.3 AuditWizard

AuditWizard[6] defines itself as an all-in-one web3 solution that comes packed with all the security tools for evaluating smart contracts. The developer has to log in to

use the software. The main interface consist of a web based IDE. On the left side a project can be imported, even directly from Github. The controls will then display the folder structure, and the user can navigate between files.

On the left side a selection of tools is displayed. It starts with an AI assistant that can answer question about the project currently open. Then it integrates different vulnerability scanners, such as Slither[7]. It follows with testing and graph generating tools.

AuditWizard really represent a complete suite of tools for smart contract and decentralised app building, improving the developer capabilities and limiting the chance of errors and bugs.

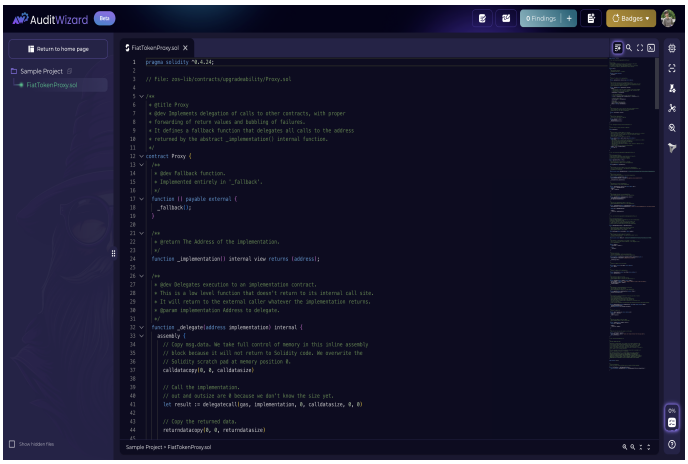

Figure 2.3: AuditWizard - IDE interface

For sure it places itself as software suited for people who are already in the development context, and it doesn't want to be an accessible platform for less technical people.
Strong Features:

- **Code editor**: The platform embodies many enhanced capabilities around a simple smart contract editor. It does not abstract the plain code in ways that could limit the user, but instead it supports the code writing.
- **Security analysis**: AuditWizard provides many tools for performing vulnerabilities analysis, such as Slither, 4naly3er[8] and Aderyn[9]. These integrations enable seamless and efficient security checks by leveraging well-established tools, reducing the need for manual code reviews and minimizing the risk of critical vulnerabilities.
- **AI Assistant**: The LLM support represent an important tool, especially when interacting with contracts that are imported from outside. The user can

interact with the AI to get explanations, feedbacks and tips.

2.2.4 Create-web3-dapp

Create-web3-dapp[10] is a CLI tool provided by Alchemy[11]. It allows the user to start decentralized app creation directly from the terminal. When the user executes it, a web3 app based on NextJs is built. This is going to be compatible with blockchains based on the EVM, as Ethereum, Polygon[12], Optimism[13] and Arbitrum[14]. It can be considered a no-code tool, but it also reduces the accessibility to the bare minimum, as it is intended to be used by developers and engineers
Strong Features:

- **CLI versatility**: It offers a command line interfaces which is very simple but powerful. It allows developers to rapidly build and configure a working web3 app. The CLI provides options to customize the app configuration, as selecting the favorite blockchain, integrating specific smart contracts and setting up dependencies

2.2.5 Nftify.network

Nftify[15] provides a solution for a very specific problem. It is a no-code tool that allows the user to create a custom NFT marketplace. It can generate all the required smart contracts and client side code. It can be considered very efficient in it reaching is goal, as it does not focus on any other result.
Strong Features:

- **Being specific**: The focus ensures that all the functionalities, the tools and the models are perfect for NFT marketplaces. This allows the user to reach his goal faster and with less difficulties, compared to more generic no-code platforms. Nftify provides dedicated support for minting NFTs, managing royalties and wallet integrations.

2.2.6 Supporting Reuse of Smart Contracts through Service Orientation and Assisted Development

The following paper[16] and related software was analyzed beside all the other tools listed. The paper looks at the problem of smart contract reuse and contract interactions, it states that some infrastructural elements that are considered useful in classic software development do not yet have an equivalent in blockchain technology. The project code is open-source and available at Guida's GitHub as SolidityRegistry[17] and SolidityEditor[18].

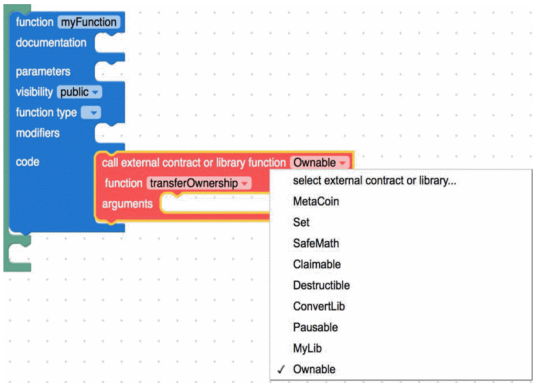

Figure 2.4: Supporting Reuse of Smart Contracts through Service Orientation and Assisted Development - Smart contract editor interface

The result of this paper consist in the introduction of multiple innovation to the state of the art of blockchain development. The project that comes with the paper introduces:

- Smart contract descriptor model and format for the abstract description and publication of reusable contracts.
- Smart contract registry for search and discovery of contracts for both human user and software agents.
- Visual development environment for model-driven development of smart contracts.

2.3 Positioning the project

The scope of this section is to determine what should be the features to implement in the project to result innovative into the current panorama.

From the comparison, emerges that each software is providing its set of features in which it tries to excel. None of them is providing all available features, but some of them like Audit Wizard is getting close by introducing a very large set of tools. The use of templates and the client code generation are quite common between no-code systems, while security analysis and AI assistant are more rare. Almost none of them introduces a more complete graphical abstraction, except for the use of templates. Only the paper "Supporting Reuse of Smart Contracts through Service Orientation and Assisted Development"[16] implement an actual graphical interface through the use of Blockly[19] library.

To represent an innovation compared to the state of the art, the software have to implement:

- An actual graphical interface that abstract the code complexity, by keeping the tool accessible to a broad range of users.
- The support of a LLM model, like ChatGPT, for describing the complex logics.
- Security analysis tools to reduce the possibilities of bugs and vulnerabilities.

2.4 Technologies

In this section are listed and described the three main tools that the project will make use of for obtaining the expected functionalities.

2.4.1 Language Recognition

ANTLR

Another Tool for Language Recognition [20], is a powerful tool for building parsers and interpreters. It is especially useful for interpreting programming languages and data format. It has established itself as a key technology in the language processing panorama, providing developers the ability to define and implement custom languages, domain-specific languages (DSLs) and data parser with ease. ANTLR generated parser are known for their robustness, with a built-in error handling system.

It allows to generate both the lexer and the parser from a single grammar file. This approach makes the development process easier by allowing to maintain the language's syntax and semantics in one place. The grammar files used by ANTLR are intuitive and easy to read, making it closer to a natural language.

Abstract Syntax Tree

The tool supports automatic AST generation, providing an easy way to move through the structure of the code. This feature is extremely useful for compilers and interpreters.

Listener and Visitor Patterns

ANTLR provides both listener and visitor interfaces. Those are design patterns used to process the parser tree. Listeners allows to react to specific parsing events, while visitors provide a method to walk though the tree.

Error handling

Error reporting and recovery capabilities are included with the tool. Whenever a syntactical issue is found, ANTLR provides a feedback useful for debugging and refining the grammars.

The project relies on ANTLR to split the contract in its individual elements, such as functions, state variables, modifiers and events.

2.4.2 LLMs

Large Language Models represent the latest improvement in the field of natural language progressing. These models are trained over vast datasets of text, from books, articles, websites and more. They are able to generate human-like text, understand context and perform a range of language related tasks.

LLMs are highly versatile and can be fine-tuned depending on the task they are used for.

This tools can be used to generate explanations for complex concepts and even assist in programming by suggesting code snippets or identifying errors in the code.

GPT-3.5

One of the most well known LLMs is ChatGPT[21], from OpenAI. This tool comes in multiple models, which change in the way they are trained and in their capabilities. For the purpose of this project, we are taking in consideration GPT-3.5, as it is a well-balanced model that offers good processing abilities while being computationally efficient. It provides robust performances for generating explanations and assisting with smart contract development. Its capabilities are sufficient to meet the project's needs without increasing the resource demands and therefore the cost associate with the use.

Function Calling

A powerful feature introduced with these models is function calling. It allows the model to return the computation result as a defined structure, with the objective of providing it as a function parameter.

This method allows to avoid having to extract data from a textual response which may vary in format on each interaction.

This capability expands the utility of the model from merely generating text to performing actions based on the text it generates.

In the project context, function calling can be used to automatically provide information on the smart contract, while abstracting the LLM interaction in a

micro service, and obtaining data in a structured way that can be integrated with the whole system in a friction less way.

Interaction

To integrate ChatGPT into a software application, the developers usually use the APIs provided by OpenAI. These allow the software to send request to the model and receive back the generated responses.

The API key must be obtained from the OpenAI console. This key is unique, as it is used to authenticate request and ensure that only authorized applications can interact with the model.

Each request is sent though HTTP to the ChatGPT API endpoint. These requests include a payload with the input prompt that the model will process. The key is included in the request header as a bearer token, to authenticate the call.

Through the OpenAI console is it possible to monitor the usage, limit the use to a specific threshold or to a specific model.

The project relies on GPT-3.5 to produce:

- Descriptions for each element resulting from the Codec split operation.
- Links between the element which represent the functional relations.

2.4.3 Security Analysis

Static analysis is a crucial process in software development. In the context of smart contracts, security vulnerabilities can have severe financial and operational consequences.

Static analysis examines the code without executing it, unlike dynamic analysis. This allows for early detection of bugs, vulnerabilities and logical flaws before the deployment process.

By analyzing the contract's codebase, these tools can identify patterns and coding practices that could lead to issues, such as reentrancy attacks, not handheld exceptions, integer overflow or improper access controls. The immutable nature of a deployed smart contract, makes this essential to avoid costly irreversible errors.

Slither

Slither[7] is a leading static analysis tools designed for Solidity. It is an open-source framework, released under GNU Affero General Public License v3.0. It is popular for its ability to detect a wide range of security vulnerabilities with high accuracy and efficiency.

- **Vulnerability Detection**: Slither excels at identifying security issues commonly found in solidity smart contracts, such as reentrancy vulnerabilities, uninitialized storage variables, improper inheritance structures, and unchecked return values.
- **Code Optimization**: It provides suggestions for code optimization by identifying redundant code, unused variables and inefficient code that could be improved. These suggestions actually reveal useful also when trying to optimise gas costs.

Slither operates by parsing Solidity code into an intermediate representation which then is analysed by a set of predefined rules and patterns. The final result of the operation is a report, which can be request in a JSON format to better integrate it with other solutions.

The project makes use of this tool by analysing the smart contract code and providing the report result to the user.

Chapter 3

System Architecture and Design

The chapter focus on the system architectural choices, providing a detailed overview of the design principles employed during the development of the project. It outlines the structural components of the system, their interactions, and the reason behind the chosen architecture. The chapter also dives deeper into design considerations including the modularity, scalability and security. By the end of it, readers will have a complete understanding of how the system is organized and how its components work together to achieve the project's objectives.

3.1 Overall System Architecture

The system is designed with a microservices architecture, emphasizing modularity, scalability and fault tolerance. It is composed of three primary microservices, each developed in Golang, responsible for distinct functionalities that all together enable the smart contract processing and analysis workflow. These services interact with a Backend-For-Frontend (BFF) layer, which serves as the intermediary between they and the client application.

With reference to the image 3.1. The graph shows the relations between each microservice, the BFF and the client. The light blue color is highlighting the functionalities offered by each service. Beside the elements names, the reference to their section. Server and clients that shares the same schema are highlighted with the same color. For instance:

- **Yellow**: The Codec gRPC[22] server and client.
- **Purple**: The Auditor gRPC[22] server and client.

- **Green**: The AI Assistant gRPC[22] server and client.
- **Dark gray**: The REST server and client.

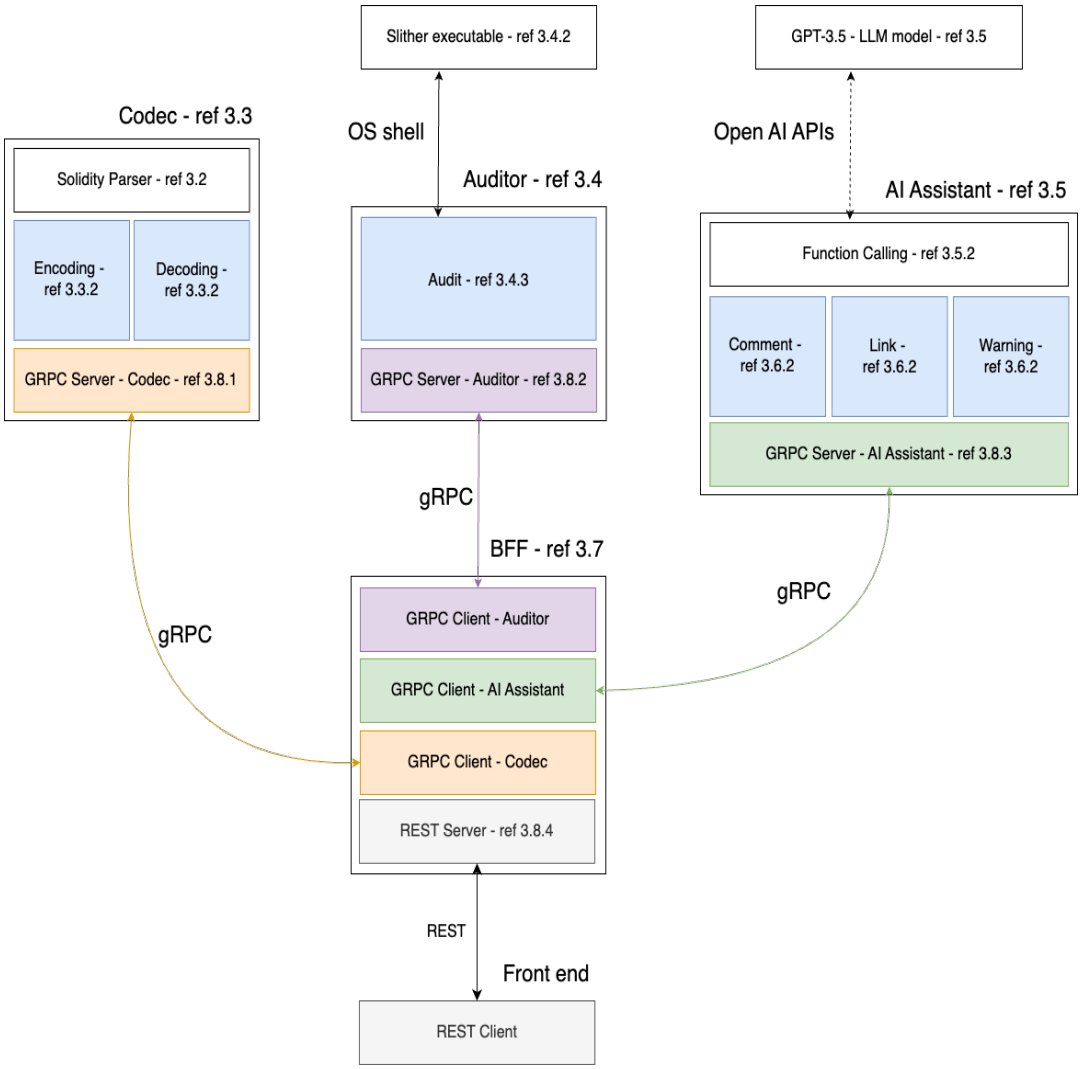

Figure 3.1: Overall System Architecture

Although the system is not currently deployed, its architecture is designed to support this scenario. Each component can be deployed independently, making it easier to scale depending on the demand. For instance the Codec microservice could require more computational resources, or the AI Assistant might be scaled based on the demand for LLM processing tasks.

3.1.1 Microservices

The following are the three microservices implemented, seen from the architecture point of view.

Codec

The Codec microservice serves as the main layer for language recognition within the system. It acts as the entry point for processing the smart contracts, converting them into a structured format that be later manipulated by other components. The Codec exposes a gRPC[22] server, enabling efficient and reliable communication with the BFF service. It is designed to be stateless, allowing to scale horizontally as needed.

Auditor

The Auditor microservice takes the role of a dedicated security analysis component. It communicates with the BFF through its own gRPC[22] server, receiving parsed smart contract data from the Codec via the BFF. It performs the analysis with complete independence, ensuring that security checks are isolated and do not interfere with other processes. The separation of concerns enhances the modularity and allows the service to scale as needed.

AI Assistant

The AI Assistant microservice is positioned as an auxiliary service that enhances the system's capabilities. It operates by interacting with the BFF through its gRPC[22] server, where it receives structured data from the Codec through the BFF. The AI Assistant's integration via gRPC[22] ensures that its language processing tasks are seamlessly incorporated into the system's workflow.

3.1.2 Language choices

Selection of Golang and Flutter

The decision to utilize Golang for the backend and Flutter for the frontend of this project was primarily driven by my prior experience and proficiency with both technologies. Having worked extensively with Golang in previous projects, I developed a good knowledge of its syntax, concurrency model, and optimization techniques. Similarly, my familiarity with Flutter simplified the creation of a an intuitive user interface. This familiarity allowed me to focus on implementing the core functionalities of the tool without the overhead of mastering new technologies.

Technical Advantages of Golang

Golang was chosen for the backend implementation due to its suitability for microservice architectures and its support for gRPC[22] code generation. Golang's static typing and memory management contribute to the reliability and maintainability of the codebase. The seamless integration with gRPC[22], further enhances Golang's capabilities by enabling efficient communication between distributed services. This is particularly positive for the project's microservice infrastructure, ensuring that the Codec, Assistant, Auditor, and BFF server can interact and perform their respective tasks with minimal overhead.

Technical Advantages of Flutter

Flutter was selected for the frontend development due to its powerful cross-platform capabilities, which allow for the creation of applications that run smoothly on both desktop and web environments from a single codebase. This cross-platform support not only reduces development time but also ensures consistency in the user experience across different devices and platforms. Flutter's customizable widgets and its rendering engine provide the flexibility needed to design the central grid for the client. The ability to maintain a unified codebase for multiple platforms also simplifies maintenance and updates, contributing to the efficiency and scalability of the project.

3.1.3 API protocols

gRPC[22]

gRPC[22] is a modern open-source protocol derived from base Remote Procedure Call (RPC). It enables efficient, low-latency communication between distributed services. It was developed by Google and it relies on Protocol Buffers (protobufs) files as a way of defining the interfaces. Developers can describe service methods and message structures in a language-agnostic way.

It was chosen for this system due to its superior performance, strong typing, and advanced features like bi-directional streaming. These qualities make it an ideal choice for the internal communication between microservices in a distributed architecture, ensuring that the system is both efficient and scalable.

REST

For communications between the Server microservice and the frontend, RESTful APIs have been defined. This approach facilitates interoperability and integration across different platforms and technologies.

The need for high performances is no more needed as it was for gRPC[22], in this system REST was selected for its simplicity and compatibility.

OpenAPI[23] is a widely adopted specification for defining APIs in a standardized, language-agnostic manner. It allows developers to clearly describe the endpoints, request and response structures, and authentication methods, facilitating automatic generation of documentation and client code. The following endpoints were defined to describe the possible interactions.

3.2 Solidity Parser

The Solidity Parser package is a custom library developed in Golang and built for abstracting the logics to transform the smart contract Solidity code into a JSON structure and back. It serves as a support and an utility for the microservices within the system. This process makes all the other operations possible, providing the ground for analysis, editing and integrations between services. It also supports the parsing back from the JSON structure into Solidity code, ensuring that all the changes and the improvements applied by the tool can be translated back in the codebase.

3.2.1 Language Recognition

ANTLR[20] (ANother Tool for Language Recognition) is a powerful parser generator, it is open source and it is used widely for reading, recognize and execute operations over text files that follow a defined structure. It is well known for its ability to generate parsers in many programming languages. Also Golang is supported, and this makes it perfect for the project requirements. ANTLR[20] uses a grammar specification to define the syntax of a language, allowing to create the lexer and the parser that can operate on code written in that language. It's architecture supports abstract syntax tree (AST) which are required for the transformations that are needed.

3.2.2 The need for a custom parser

The complexity and variability present in Solidity smart contracts create the need for a parsing solution that is tailored to the specific requirements of the system. Existing parsers usually produce abstract syntax trees (ASTs) that are generalized and may not align with data structures required for our application's functionality. To extract and represent the core elements of smart contracts in a way that facilitates the processing, it was important to develop a custom parser. This parser is designed to not only capture the basic components of a smart contract but also

to add to them metadata, including unique identifiers and the properties that will be later evaluate by other services.

By making use of ANTLR in combination with an existing Solidity grammar, we implemented custom logics over the listeners that traverse the AST. This approach allows to build data structures that are optimized for the tool operations.

3.2.3 Implementation of the Solidity Parser

ANTLR[20] act at the core of the Solidity Parser package, allowing the project to define a custom parser that can encode and decode the Solidity smart contract following the project needs. The lexer takes care of tokenizing the Solidity code, splitting it in tokens, each representing a single syntax element of the language. These are then provided to the parser, which uses them to build the abstract syntax tree that reflects the code structure. The parser is extended by the implementation of listeners and visitors. These tools are triggered every time a node in the AST is visited, both when entering and getting out from it, and allow to implement custom logics. Here, a structure that follows the project requirements takes shape, the Solidity Parser creates each element that is going to compose the JSON later used by the microservices.

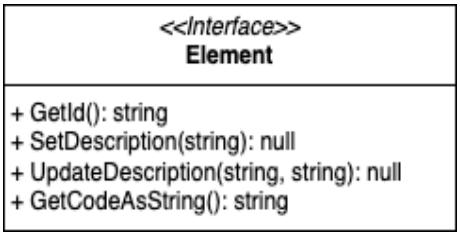

Figure 3.2: Solidity Parser - Element interface

Within the packages, each of the element composing the resulting structure have been defined as a single structure, with its properties and methods. An interface `Element` provides a common ground which forces each item to implement the following methods:

- **GetId**: return the item's ID.
- **SetDescription**: set the item's Description.
- **UpdateDescription**: upon receiving an ID to search and a Description string to set, the code perform a check on the element itself to see if the IDs are the same. If yes, the Description is set and the method returns. Otherwise, the search continues within the children properties that are also implementing the Element interface.

- **GetCodeAsString**: similarly to the previous one, it runs a recursive operation to the item and to its children. Each element lies down its portion of code using a template. The result of the recursion is then returned.

The UML in **figure 3.2.3** contains all the Elements and their relations.

3.2.4 Exposed Methods of the Solidity Parser

To facilitate the conversion between the source code and its corresponding JSON, the Solidity Parser package provides two methods:

```
ParseSmartContract(sourceCode string) (*listener.SourceUnit, error)
```

The first one receives as input a string containing the Solidity code and returns a `SourceUnit` object, which is the structured representation derived from the parsed AST.

```
GetCodeFromSourceUnit(sourceUnit listener.SourceUnit) (string, error)
```

The second one performs the inverse operation. It takes a `SourceUnit` object as parameter and reconstruct the Solidity code as a string. This method goes through the JSON structure using the information provided within the `SourceUnit` to build a working Solidity code.

3.3 Codec

The Codec microservice act as a core component in the whole architecture. It is responsible for the first analysis phase by handling the encoding and decoding functionalities, providing the interface to transform the code into the JSON structure and back. The Codec act as the only point to perform this operation, granting that the process is safe and consistent, simplifying future analysis and processes performed by other services. This separation of concern allows it to scale and optimize its process if needed, in an independent way.

3.3.1 Initialization and Startup

Upon initialization, the microservice performs a series of startup procedures. The first step involves loading configuration parameters from an `.env` file, which includes specifying the port on which the gRPC[22] server will listen for incoming requests. This approach allows flexible configuration and improves the security of sensitive information by keeping it separate from the source code.

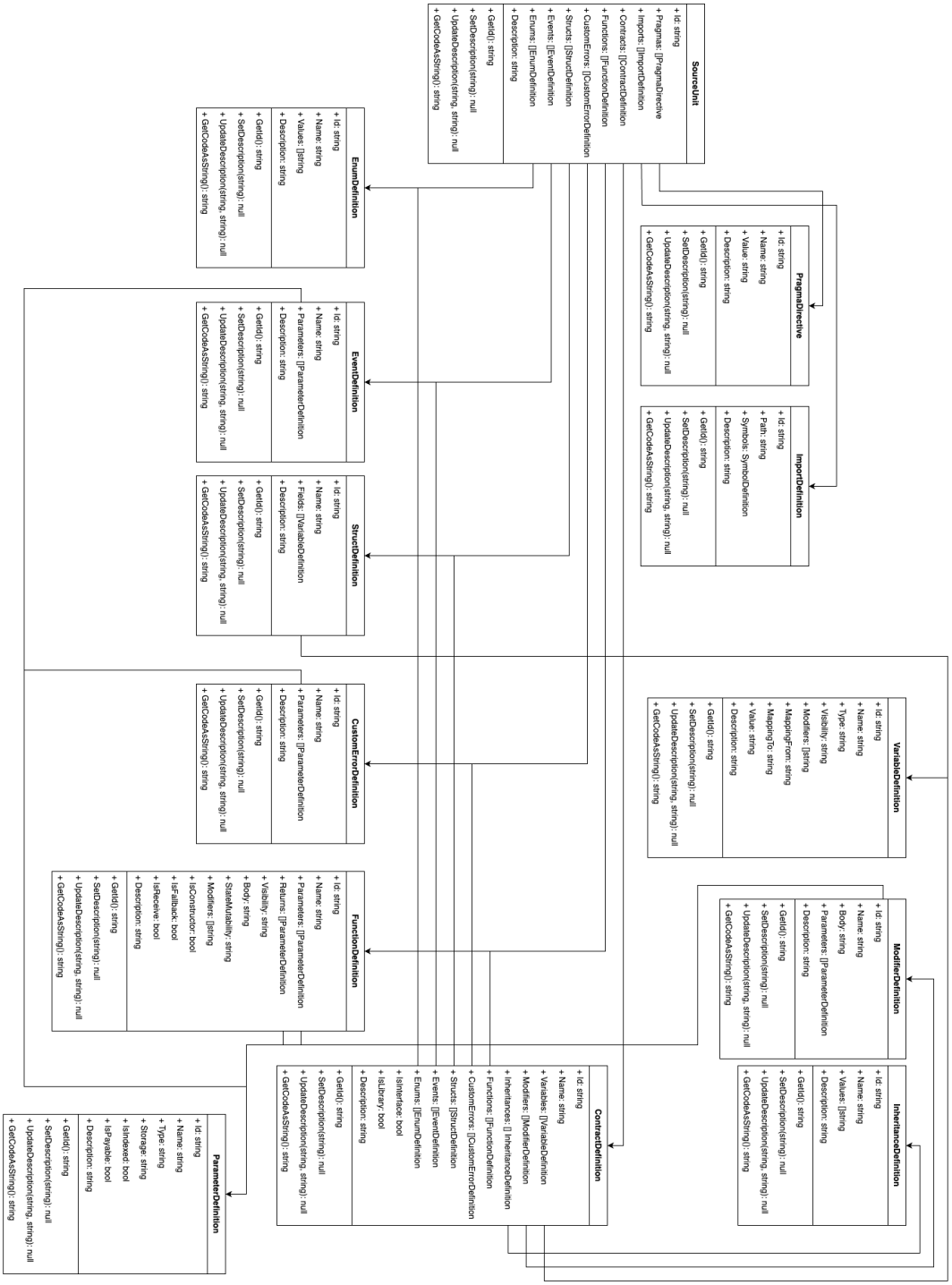

Figure 3.3: Solidity Parser - resulting structure UML

Following the configuration setup, a series of self-tests to verify the integrity and functionality of its capabilities are executed. This testing phase involves reading a sample Solidity file bundled with the service, executing the Solidity compiler (`solc[24]`) to compile the contract, and then performing both encoding and decoding operations on the compiled contract. These tests are required to ensure that the Codec can transform accurately the Solidity code into JSON and rebuild the starting source code without errors. The successful result of these tests confirms that the services is working as expected. Only after all these checks, the Codec starts the gRPC[22] server, making its services reachable to the other entities in the system.

3.3.2 Functionalities

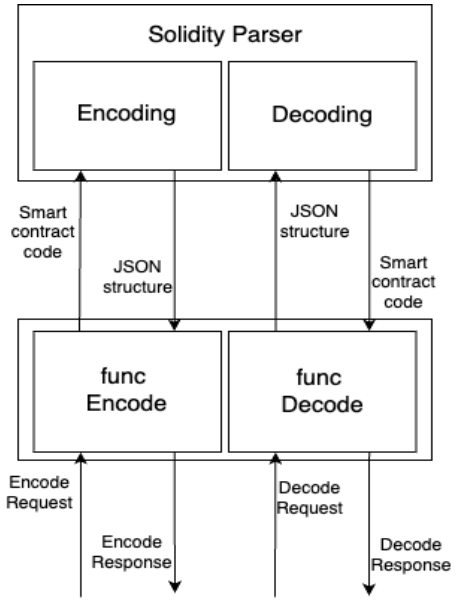

Figure 3.4: Codec - Functionalities

The primary functionalities of the Codec revolve around handling encoding and decoding requests via its gRPC[22] interface. Once the server is operational, the Codec exposes two main methods that facilitate transformation between Solidity code and JSON structures:

```
Encode(ctx context.Context, req *service.EncodeRequest) (*service.EncodeResponse, err
```

This method, `Encode`, accepts an `EncodeRequest` containing the Solidity source code as input and returns a `EncodeResponse` object, which contains the structured JSON format of the smart contract.

```
Decode(ctx context.Context, req *service.DecodeRequest)
(*service.DecodeResponse, error)
```

The `Decode` method expects a `DecodeRequest` object parameter. From that it extracts the JSON structure and reconstructs the original Solidity code.

3.4 Auditor

The Auditor services is an important component that takes part in the system architecture as the one responsible for increasing security and making the smart contract more reliable. Its functionality is to execute static analysis on the Solidity code using [7]. The result of this operations allows to identify potential vulnerabilities and warnings, providing the developers insights and tips to improve their smart contracts.

3.4.1 Initialization and Startup

Similarly to the other microservices, the Auditor undertakes a series of startup procedures to ensure being ready. The first step involves loading configuration parameters from an `.env` file, which includes specifying the port on which the gRPC[22] server will listen for incoming requests.

The Auditor then initializes an instance of `AuditorUtils`, a utility class designed to interface with the Slither[7] tool. This instance is responsible for executing audits and processing the results obtained from Slither[7], abstracting the complexities of running static analysis. To verify the functionalities, the microservice runs a self-test by executing an audit on a sample Solidity file included with the service. This test grants that Slither[7] is correctly integrated and that the Auditor can accurately interpret the analysis results. Successful completion of these tests confirms that everything is fully working. The Auditor proceed to start the gRPC[22] server, making its auditing services available to the system.

3.4.2 Slither integration

The microservice integrates Slither[7] to perform the security analysis. This integration is performed by including the Slither[7] executable, specifically compiled for the operating system on which the service is deployed, within the project's directory structure. After receiving a smart contract for analysis, the Auditor generates a temporary file containing the provided code. It then invokes Slither[7] through the operating system's shell to analyze this temporary file, resulting in a JSON report that contains the various vulnerabilities and warnings identified. It

subsequently parses the JSON report to extract the data, which is then formatted and returned to the user.

3.4.3 Functionalities

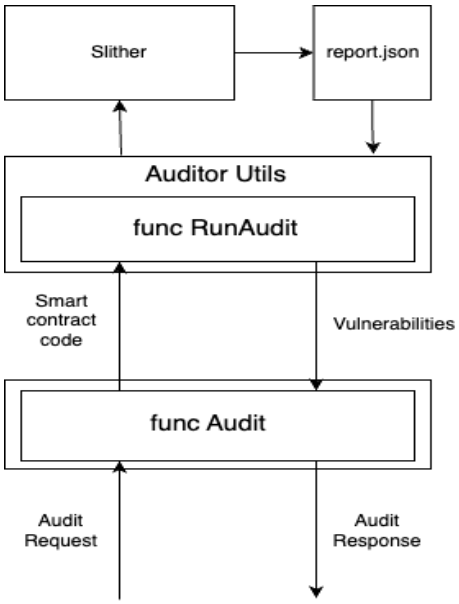

Figure 3.5: Auditor - Functionalities

The only functionality of the Auditor microservice consist in handling an audit requests via its gRPC[22] interface. It exposes a single method:

```
Audit(ctx context.Context, req *service.AuditRequest)
(*service.AuditResponse, error)
```

This method, `Audit`, is designed to accept an `AuditRequest` containing both the Solidity smart contract code and the JSON structure. The Auditor utilizes the `AuditorUtils` instance to execute Slither[7], performing a static analysis of the provided smart contract. Slither[7] examines the code for a wide range of potential vulnerabilities, including reentrancy attacks, integer overflows, and other common security flaws specific to Solidity contracts.

After it completes the analysis, the results are processed into an `AuditResponse`, containing a detailed list of identified vulnerabilities and warnings. This response is then returned, enabling developers to promptly address and rectify the highlighted issues.

3.5 LLMs

Large Language Models (LLMs) are a significant advancement in natural language processing. Models like GPT-3.5[21] from OpenAI are trained on a large amount of text data, allowing them to generate human-like text, always depending on the input they received. LLMs are composed of deep levels of neural networks with many parameters, allowing them to go further that simply generate text. They can be a valid tool for translations, summaries, analysis of many kinds, and even writing code.

This project will not focus on LLMs, but it will dive deeper in how it is possible to leverage on their capabilities.

ChatGPT[21]

ChatGPT[21] is a perfect example of an LLM built specifically for conversation. It's main ability is understanding the context, keeping the flow and providing accurate responses and information. All of these makes it a powerful tool for apps that require to interact dynamically with the user. Even more, it can be customized with extension for specialized activities, like in the project case, assisting in software development.

3.5.1 Integration

Integrating ChatGPT[21] into the project was done through the use of the `go-openai[25]` library, an open-source client for OpenAI's API. This library provides a convenient interface for developers to interact with OpenAI's models, enabling functionalities such as sending prompts, receiving generated text, and managing conversational contexts. In the context of this project, `go-openai[25]` is utilized within the Assistant microservice to harness the LLM capabilities for adding information to the JSON structures produced by the Codec microservice. The integration involves configuring API keys, setting up client instances, and defining request parameters to tailor the model's responses according to the application's requirements. The support for asynchronous operations and error handling in `go-openai[25]` ensures that the integration is efficient and resilient.

3.5.2 Function Calling

An useful feature of ChatGPT[21] that significantly shows its utility within the Assistant microservice is `function calling[26]`. This method allows developers to define specific functions that ChatGPT[21] can invoke during the conversation, enabling the model to perform controlled actions based on user inputs or contextual triggers. In this project, function calling[26] is employed to automate tasks such

as commenting on smart contract elements, establishing functional relationships between components, and identifying areas for improvement in smart contract functions.

The implementation of function calling[26] involves several steps:

1. **Definition**: Developers specify the functions that ChatGPT[21] can call, defining the names, parameters, and expected outputs.

2. **Triggering**: Based on the analysis of the JSON structure, certain conditions or keywords can prompt ChatGPT[21] to invoke the corresponding functions. This ensures that the model's responses are not only contextually relevant but also actionable.

3. **Handling Responses**: Once a function is called, the the service can process the output, integrating the results back into the JSON structure.

This feature ensures that the interactions with ChatGPT[21] are not only conversational, but structured as the code expects, aligning the model's capabilities with the project's technical requirements. It also removes the need for extracting data from the response using Regex expressions.

Here's the definition of the function call that expects the element descriptions, in the Assistant service:

Function calling example

```
// Define the schema for the description item
item := JSONschema.Definition{
    Type: JSONschema.Object,
    Properties: map[string]JSONschema.Definition{
        "id": {
            Type:        JSONschema.String,
            Description: "The ID belonging to the element currently described, e.g. 058f661c-2c5e-4eed-ab92-72b102f19ee7, a47acf50-0907-4663-bb1f-38117a2a42f6",
        },
        "description": {
            Type:        JSONschema.String,
            Description: "The description of the element, e.g. \"Function to deposit 1 ether into the contract, updating the balance and potentially setting the winner.\", \"Function for the winner to claim all ether in the contract.\"",
        },
    },
    Required: []string{"id", "description"},
}

// Define the schema for the parameters
```

```
params := JSONschema.Definition{
    Type: JSONschema.Object,
    Properties: map[string]JSONschema.Definition{
        "items": {
            Type:  JSONschema.Array,
            Items: &item,
        },
    },
}

// Define the function for setting the description by ID
f := openai.FunctionDefinition{
    Name:        "set_description_by_id",
    Description: "Set the description to the element with the given ID.",
    Parameters:  params,
}

// Define the tool that uses the function
t := openai.Tool{
    Type:     openai.ToolTypeFunction,
    Function: &f,
}
```

In this example, the function is defined and made available to ChatGPT[21]. When the user requests descriptions for on a smart contract, ChatGPT[21] invokes the function, providing the necessary parameters. The function then processes the input and returns a comment, which is integrated into the JSON structure by the Assistant microservice.

3.6 Assistant

The **Assistant** microservice is responsible for enriching the structured JSON data generated by the Codec microservice. By interfacing with ChatGPT[21] LLM, the Assistant adds contextual information and meaningful annotations to the smart contract elements, improving their comprehensibility and utility. The primary functionalities of the Assistant include generating comments for JSON elements, defining functional relationships between contract components, and identifying functions that may need improvements. This support enables developers to efficiently analyze and optimize their code.

3.6.1 Initialization and Startup

Similarly to the other microservice, the Assistant initialized itself by loading loading configuration parameters from an `.env` file. Following the configuration setup, the

Assistant proceeds to instantiate and launch the gRPC[22] server, thereby making its services accessible to other microservices and the frontend. There's no need for testing the LLM interactions, as the OpenAI API key will be provided in each request, and it's not yet available to the service.

3.6.2 Functionalities

The Assistant microservice is built to handle three types of requests.

- **Providing descriptions**
- **Linking and commenting elements**
- **Adding warnings over functions that can be improved**

Each of these request, performs some operation on the JSON structure coming from the Codec. A distinct aspect of the Assistant is that every incoming request needs to have the OpenAI API key within itself. This allows more users to use the microservice at the same time. This design choice requires the creation of a new instance of the OpenAI client for each received request, allowing secure and isolated interactions with the model.

Comment

The **Comment** request scope consist of generating descriptive annotations for individual JSON elements representing various components of the smart contract. Upon receiving a Comment request, the Assistant utilizes a predefined setup prompt made to instruct ChatGPT[21] to analyze the provided code snippets and generate meaningful descriptions. The updated JSON structure, now with these descriptions, improves the readability and comprehension of the smart contract elements.

Link

The **Link** request focuses on defining functional relationships between different elements within the smart contract. For instance, it identifies scenarios where a function modifies the value of a variable, establishing clear connections that represent the contract's operational flow. The Assistant employs a specialized setup prompt that guides ChatGPT[21] to recognize and articulate these relationships, creating a network of interconnected elements.

Warning The **Warning** request is responsible for identifying functions within the smart contract that could be optimized or improved. By analyzing the code, the LLM flags specific functions and returns their identifiers. This identification

of potential improvements contributes to the overall quality and efficiency of the smart contracts being developed.

3.6.3 Setup Prompts

Un aspetto critico della funzionalità del microservizio è l'uso di **chiamata di funzione[26]** insieme ai prompt di configurazione per interagire efficacemente con ChatGPT[21]. Ogni richiesta utilizza un prompt distinto che inizializza il contesto e fornisce istruzioni specifiche affinché LLM esegua la chiamata di funzione desiderata.

A critical aspect of the microservic's features is the use of **function calling[26]** with the configuration promt. This allows to interact in an efficent way with ChatGPT[21], as every request uses a distinct prompt that initialize the context and provide clear instructions for the LLM to execute the expected function call.

These are the steps the follows every time the Assistant receives a request:

1. **Client Initialization**: The service creates a new instance of the OpenAI client, using the unique **OpenAI API key** provided with the request. This ensures that the following interactions with ChatGPT[21] are isolated for the user.

2. **Setup Prompt loading**: The Assistant loads the correct **Setup Prompt** depending on the request type. These files contains instructions and context details needed for ChatGPT[21] to later execute the function calls.

3. **Function Calls execution**: All the relevant data are sent from the service to ChatGPT[21] using a second message in the same conversation. The prompt used for this operation, **appendix ??**, contains two placeholders which are replaced at runtime with the smart contract code and the JSON structure. ChatGPT[21] processes the input based on the instructions and invokes the correct function calls.

4. **Integrating Responses**: The Assistant integrates the responses from ChatGPT[21] back into the original JSON structure or in the response object, ensuring that the are then available for further processing by other microservices or the frontend.

The technique used for writing the prompt is inherited from the COSTAR methodology, but it was modified following a long list of trials and errors. The current structure of the prompts follows this four steps.

1. **Objective**: First we define the scope of the operation. The model is informed of what we are trying to accomplish and how it should act. It's important

to also specify that we are expecting results to be returned through function calling[26].

2. **Input Example**: Then, an example of the input data is provided. In the following prompt files, the example usually contains the Solidity Smart Contract's code, and the JSON structure.

3. **Instruction Steps**: The next part consist in listing the steps that the model needs to follow in order to perform the operation. This act as guidelines that it should follow to maintain consistency across multiple requests.

4. **Output Example**: Finally, an example containing the output is provided. In particular the prompt lists some possible function calls, based on the input example previously provided.

To illustrate the implementation of function calling[26] within the microservice, below are the descriptions of the setup prompt files used for each request type.

Comment Setup Prompt

The full prompt is available in the **appendix .1**.

First the prompt contains a small paragraph, introducing the LLM model to the files it will receive. It introduces the model to what it will receive, how it should implement the descriptions, and how it should respond using the function calls.

It's important to specify to the model that this prompt is only a setup, and the actual request will come later.

Comment Setup Promp - Function call

```
1 THIS PROMPT IS ONLY A SETUP FOR LATER REQUEST.
2 DO NOT PERFOM FUNCTION CALLS AS RESPONSE TO THIS FIRST PROMPT.
3 ANSWER ONLY WITH "OK" IF EVERYTHING IS CLEAR.
```

It follows an example of a smart contract code, and the JSON correlated. This example contains a good number of different elements, which can be then referred for telling the model what kind of response it should provide.

The third part of the prompt explains the steps to follow for obtaining the expected result. The model is requested to go though each element and generate a detailed description based on both the solidity code and the context provided in the JSON.

The last part of the prompt contains some possible function calls expected from the example provided.

Comment Setup Promp - Function call

```
1 For the deposit function:
```

```
2
3 # set_description_by_id("577d831c-0350-4789-82fc-6ad4b93e7841", "
    Function to deposit 1 ether into the contract, updating the
    balance and potentially setting the winner.")
```

Finally, some details were added, containing recommendation for the LLM to ensure the coverage of the contract when performing the function calls.

Link Setup Prompt

The full prompt is available in the **appendix .2**.

As the previous one, this prompt contains a small paragraph introducing the LLM model to the files it will receive and the actions it needs to perform. It explains the model to identify the relations between elements, ensuring none is missed, and for each of those it should perform a function call.

The prompts contains a step by step guide for the model that lists the possible types of relations, the properties of each of those and how to identify them.

Then It continues with an example smart contract and JSON, followed some possible function calls expected from the example provided.

Link Setup Promp - Function call

```
1 # set_relation('577d831c-0350-4789-82fc-6ad4b93e7841', 'eb5025c8
    -9650-4a23-8ec2-b0f873af2417', 'The function is increasing the
    value of the variable balance depending on the import received.',
    'set')
```

Warning Setup Prompt

The full prompt is available in the **appendix .3**.

This prompt starts with a small paragraph introducing the model to the files it will receive and the actions it needs to perform. The LLM should identify the functions which can be improved, and then it should provide a list of IDs though the function call.

As usual, it follows an example smart contract, the JSON and some possible function calls expected from the example provided.

Link Setup Promp - Function call

```
1 # set_warning('a47acf50-0907-4663-bb1f-38117a2a42f6')
```

3.7 BFF

The Backend For Frontend (BFF) microservice serves as a central layer within the project's architecture. It acts as the hub that handles communication between the

frontend application and the microservices. The primary reason for implementing a BFF lies in its ability to abstract backend interactions specifically to the needs of the frontend, simplifying client-server communication and optimizing the performances. By encapsulating the complexities of interacting with multiple microservices, the BFF takes care of data aggregation, error handling and request orchestration. Changes to backend services can be managed independently without necessitating significant modifications to the frontend codebase.

3.7.1 Components

The BFF microservice is composed of several components, each one with its specific role:

- **gRPC[22] Clients for Microservices**: The BFF includes dedicated gRPC[22] client instances for each of the three primary microservices. These clients are responsible for sending requests to and receiving responses from their respective microservices, enabling communication within the system.
- **REST Server**: The BFF exposes a RESTful API to the frontend application, acting as the entry point for client requests. This REST server handles incoming HTTP requests, computes the data by performing appropriate gRPC[22] calls to the microservices, and then formats the responses to be sent back to the frontend.

By maintaining separate client instances for each microservice, the BFF ensures organized and efficient communication channels, while the REST server provides a unified interface for frontend interactions.

The Task object

A fundamental aspect of the BFF microservice is the introduction of the "task" concept. A task represents a single analysis request initiated by the frontend, encapsulating all relevant data and maintaining the state of the analysis throughout its lifecycle. This abstraction allows the system to handle multiple concurrent analysis requests, ensuring that each task operates in isolation without interference from others. By associating each task with a unique identifier, the BFF can track the progress, manage the state, and handle the results of each analysis independently.

3.7.2 Task Analysis Workflow

The elaboration workflow in the BFF microservice is built to handle the uploading and the anlysis of Solidity smart contracts in an asynchronous way. It is composed of the following steps:

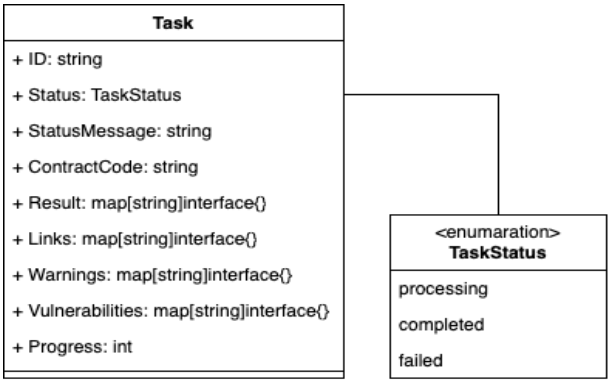

Figure 3.6: BFF - Task Object

1. **Task creation**: as the user uploads a Smart Contract through the frontend, the BFF creates a new instance of a Task object, which receives a unique identifier. The Task responsibility is to be associated and maintain the state of the following analysis requests , allowing the system to track the progress independently from other tasks.

2. **Chain of Microservice Requests**: The BFF orchestrates a sequence of gRPC[22] requests to the microservices, each corresponding to a specific stage of the analysis process. The progression through these stages is as follows:

   - **Encoding (30% Progress)**: The BFF sends the smart contract code to the Codec microservice to transform it into a structured JSON format.
   - **Generating Descriptions (45% Progress)**: The Assistant microservice is invoked to add descriptive comments to the JSON elements.
   - **Computing Links (65% Progress)**: The Assistant microservice further analyzes the JSON structure to define functional relationships between different contract components.
   - **Computing Warnings (75% Progress)**: The Assistant microservice identifies functions within the smart contract that could be optimized or improved, providing a list of warnings.
   - **Auditing (100% Progress)**: The Auditor microservice performs a static analysis using the Slither[7] tool to identify potential vulnerabilities and security issues within the smart contract.

3. **Task State Management**: After each microservice request, the BFF updates the task's state with the results and the current progress percentage. This continual updating allows the system to monitor the analysis's advancement and provides real-time feedback to the frontend.

4. **Error Handling**: If any step within the chain of microservice requests fails, the BFF immediately sets the task's status to `failed`, ensuring that the client is informed of any issues encountered during the analysis.

5. **Completion**: Upon successful completion of all analysis stages, the task's status is updated to `completed`, and the final results, including the JSON structure and identified vulnerabilities, are sent back to the frontend. The frontend can then present these results to the user.

3.8 APIs design

3.8.1 Codec

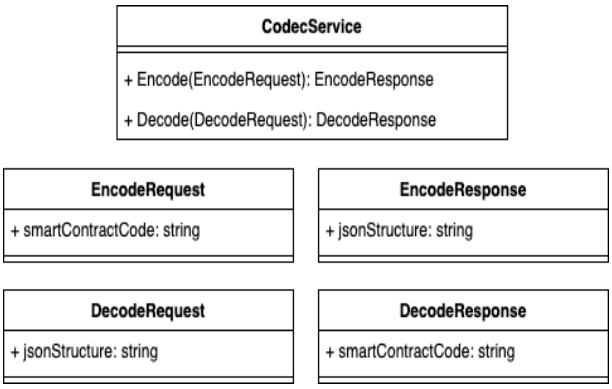

Figure 3.7: APIs design - Codec Protobuffer

The service's functionality is defined using a profobuf file, which describes the structure of the service and the messages it can handle. The Codec has a very simple interface, it receives the solidity code and returns the JSON structure. It can eventually perform the inverse operation.

`Figure 3.7` visually represent the content of the protobuffer file, which is available in its full content as **appendix .5**.

The `service` keyword defines a gRPC[22] service named `CodecService`, which consists of two calls:

- **Encode**: Takes an `EncodeRequest` message as input and returns an `EncodeResponse` message. It is designed to handle the conversion of a smart contract from code format into a structured JSON format.

- **Decode**: Performs the inverse operation. It takes a `DecodeRequest` message and returns a `DecodeResponse` message, handling the conversion process from JSON back into smart contract code.

3.8.2 Auditor

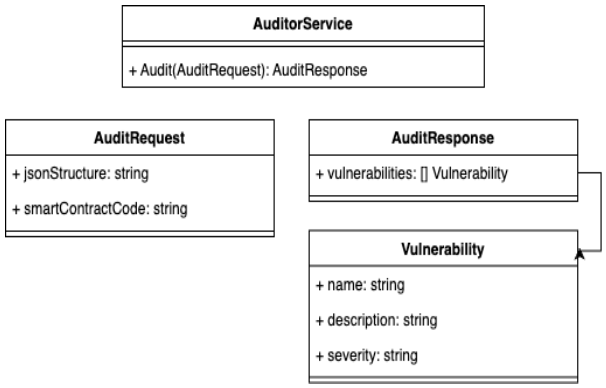

Figure 3.8: APIs design - Auditor Protobuffer

The Auditor is responsible for analyzing the contract code and returning the list of vulnerabilities found. The gRPC[22] implementation is similar to the Codec one, but with a single call available.

`Figure 3.8` visually represent the content of the protobuffer file, which is available in its full content as **appendix .5**.

The file defines an `AuditorService`, which consists of only one call:

- **Audit**: Takes an `AuditRequest` message as input and returns an `AuditResponse` message. The response contains the list of vulnerabilities found, with details like the name, a description and the severity.

3.8.3 AI Assistant

The AI Assistant service has to perform two similar operation. First it takes care of commenting each element in the JSON structure obtained by the Codec, and then it produces the functional connections between elements.

`Figure 3.9` visually represent the content of the protobuffer file, which is available in its full content as **appendix .5**.

The file defines an `AiAssistantService`, which consists of three calls:

- **Comment**: Takes an `CommentRequest` message as input and returns an `CommentResponse` message. Given the JSON structure and the contract code, it comments the elements and returns a JSON structure with the comments added beside each item.
- **Link**: It takes a `LinkRequest` message and returns a `LinkResponse` message. Similarly to the previous, it takes the structure and the code, it generates the links and returns them as JSON.

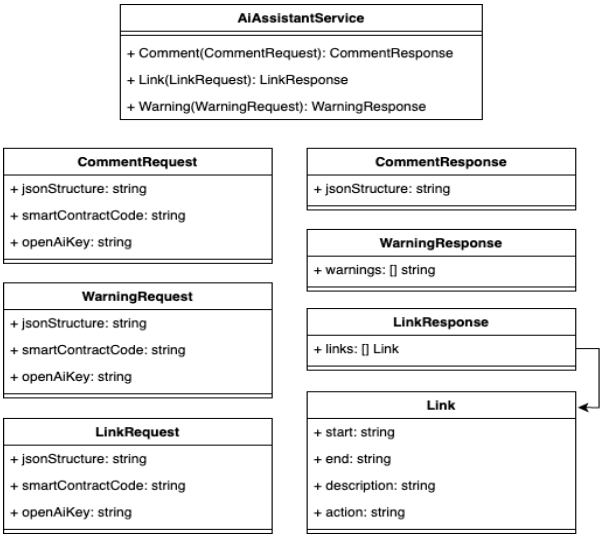

Figure 3.9: APIs design - AI Assistant Protobuffer

- **Warning**: Expects a `WarningRequest` message and responds with a `WarningResponse` message. Using the JSON structure and the code, it returns the possible functions that could be improved.

3.8.4 Client

For communications between the Server microservice and the frontend, RESTful APIs have been defined. This approach facilitates interoperability and integration across different platforms and technologies.

The need for high performances is no more needed as it was for gRPC[22], in this system REST was selected for its simplicity and compatibility.

OpenAPI[23] is a widely adopted specification for defining APIs in a standardized, language-agnostic manner. It allows developers to clearly describe the endpoints, request and response structures, and authentication methods, facilitating automatic generation of documentation and client code. The following endpoints were defined to describe the possible interactions.

Through the following sub sections, refer to the full Client APIs Schema at **appendix .6**.

Upload the smart contract

The `/upload` endpoint is designed to perform the submission of Solidity smart contract files for analysis within the system. Implemented as a POST request, this endpoint accepts multipart form data comprising two essential components: the

file and the `openAiKey`. The file parameter expects the Solidity source code in a binary format. The `openAiKey` parameter requires the user's OpenAI API key, needed by the Assistant microservice.

As the endpoint receives a valid request, it starts the analysis workflow by sending the uploaded file to the microservices for encoding, assistance and audit. This process happens in an asynchronous way, this means that the upload request from the frontend, is immediately resolved with a possible status 200, returning a JSON with the `taskID`. This `taskID` works as the unique identifier which allows the client to refer to the previous request when asking for progress and getting the results of the analysis back though the APIs.

Retrieving the results

The `/tasks/{taskId}` endpoint is designed to allow clients to query the status and retrieve the results of a previously submitted smart contract processing task. Implemented as a `GET` request, this endpoint requires a path parameter `taskId`, which serves as the unique identifier for the specific task.

Upon receiving a valid request, the endpoint responds with a `200` status code, indicating a successful retrieval of the task information.

- **id**: A string that uniquely identifies the task.
- **status**: A string representing the current state of the task, which can be one of `processing`, `completed`, or `failed`.
- **progress**: An integer indicating the percentage of task completion, providing real-time feedback on the processing status.
- **statusMessage**: A string offering additional contextual information about the task's current status.
- **result**: An object containing the outcomes of the processing task.
- **vulnerabilities**: An object detailing the results from the Slither[7] static analysis.
- **links**: An object that maps the connections between different elements.
- **warnings**: An object listing the identifiers of functions within the smart contract that can be improved.

The frontend application employs a polling mechanism to periodically send `GET` requests to this endpoint using the `taskId` obtained during the file upload phase. This continuous polling enables the frontend to update the user interface in

real-time, reflecting the current progress of the task and displaying the analysis results once the processing is complete.

Here's a visual representation of the polling process.

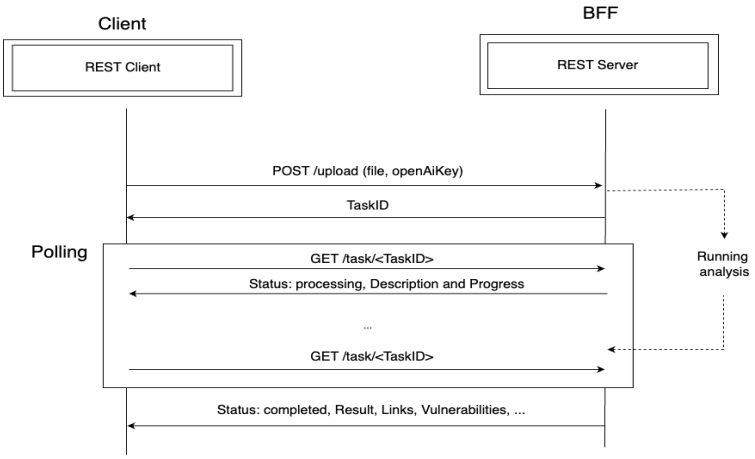

Figure 3.10: APIs design - Task Polling Process

Chapter 4

Frontend

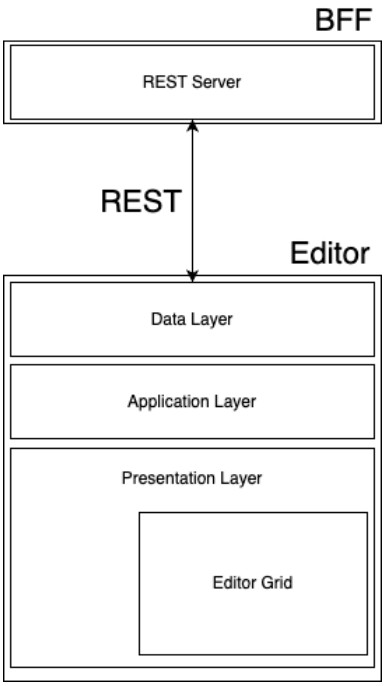

Figure 4.1: Frontend architecture

The Frontend act as the user facing component of the system, providing an intuitive and interactive interface through which users can interact with the underlying network of microservices. It is developed as both a desktop and web application using Flutter. It is designed to facilitate interactions, display analysis results, and offer a user-friendly visualization of the Solidity smart contract code, without the need for direct coding.

Central to its functionality is its ability to communicate effectively with the

BFF, managing requests and presenting the data from responses. To achieve this, the frontend architecture is divided into three primary layers: the Data Layer, the Application Layer, and the Presentation Layer.

- **Data Layer**: Utilizes the chopper[27] package to generate the API client from the OpenAPI schema .6, providing communication with the BFF. This layer also defines multiple data models to parse and manage the responses received from the backend.
- **Application Layer**: Implements the Bloc state management [28] technique to handle the application's state, manage data flow, and respond to user events. This layer acts as an intermediary between the data layer and the presentation layer, ensuring that the user interface remains synchronized with the data and business logic.
- **Presentation Layer**: Comprises various pages and widgets that render the application's intereface based on the current state. The main component of this layer is the custom Editor Grid widget 4.4. This widget provides a dynamic interactive grid interface, allowing users to manipulate and visualize smart contract elements.

The Editor Grid 4.4 widget offers custom responsive grid system, where users can navigate using mouse interactions, reposition elements through drag functionality, and visualize connections between different components via line connections.

4.1 Data layer

The Data Layer acts as the bridge between the frontend app and the microservices. It is responsible for handling all the data related operations, ensuring the communication, data fetching and data manipulation. By isolating these mechanisms from the rest of the client, this layer is improving maintainability, making the client scalable and testable, and overall increasing separation of concerns. This abstraction simplify the development process and makes it easy to integrate new functionalities in the future.

4.1.1 API Integration

The frontend uses the Chopper package [27], an HTTP client generator for Dart and Flutter. This package streamlines the process of creating API clients by automatically generating the necessary classes and methods based on a predefined OpenAPI schema document.

The process begins with the definition of the OpenAPI [23] schema, which describes the available endpoints, request parameters, and response structures of the backend services. The full schema is available in the appendix .6. The frontend generates strongly-typed API client classes that correspond to these endpoints. This results in a set of models that encapsulate the details of HTTP requests, handling serialization and de serialization of data seamlessly.

Through the API client, the frontend can initiate new tasks, monitor their progress, and retrieve the results of analyses. The task object retrieved from the Task API contains a result property, which holds the JSON structure resulting from the analysis performed by the backend microservices.

4.1.2 Repository Pattern Implementation

The Repo (Repository) classes are responsible for abstracting the data access logic from the rest of the app, by handling API request, fetching data and managing responses coming from the backend. By encapsulating these operations, the frontend ensures the logics related to a specific type of data is stored in one place, improving reuse and allowing making changes easily.

These classes perform the following key functions:

- **Performing Requests**: Using a Chopper client generated from the open API schema, these classes initiate various types of requests, as creating a task or fetching the task results.
- **Waiting for Responses**: The repos handle the asynchronous operations, ensuring the frontend waits and reacts in the correct way to the incoming data, eventually updating the user interface.
- **Returning Results**: After receiving the responses, these classes are responsible of transforming raw data into actual models that the application can use.

4.1.3 VisualElement Abstraction

The JSON structure obtained represents a complex hierarchy of smart contract components, each corresponding to various elements such as functions, variables, and properties. To effectively manage and interact with these elements within the frontend, the application parses the JSON into a series of Visual Elements. These are designed using an abstract class called VisualElement, which enforces a consistent interface and behavior across all implementations.

The VisualElement class defines a set of abstract methods and properties that each concrete Visual Element must implement:

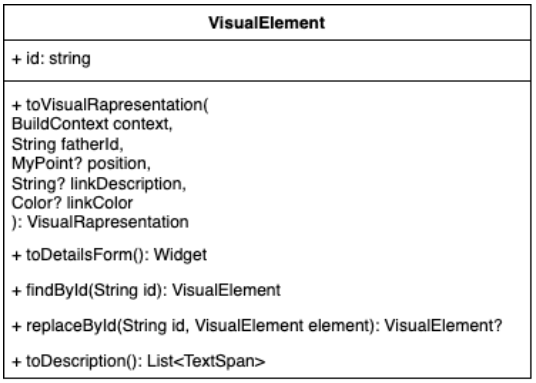

Figure 4.2: Data layer - Visual Element

- **id**: A unique identifier for the Visual Element.
- **toVisualRapresentation**: Transforms the Visual Element into a visual representation suitable for display within the UI, accepting parameters such as context, position, and link details.
- **toDetailsForm**: Generates a detailed form widget including information such as the element description and its properties.
- **findById**: Recursive search for a Visual Element within the hierarchy by its unique identifier.
- **replaceById**: Replaces a specific Visual Element identified by its ID with another element.
- **toDescription**: Provides a list of text objects that describe the Visual Element, facilitating rich text descriptions within the UI.

This abstraction allows for a flexible and extensible system where various smart contract components are represented as subclasses of VisualElement. For example, a Function is a Visual Element that implements the abstract methods to provide its specific visual representation, description, and interactive capabilities within the grid interface.

Parsing JSON into Visual Elements

Upon receiving the JSON structure from the Task API, the frontend application parses this data into a hierarchy of Visual Elements. Each element in the JSON corresponds to a specific component of the smart contract and is instantiated as an implementation of VisualElement. This parsing process involves mapping

the JSON properties to the appropriate Visual Element class, ensuring that each component's data is accurately represented.

4.2 Application Layer

The Application Layer acts as the middle man between the Data Layer and the Presentation Layer. Its main function is to manage the app state, introduce the business logic and handle the communication between the user interface and the data sources. The main role of this later is orchestrating the data and events flows, ensuring that the frontend reacts to the backend responses and to the user interactions.

The frontend makes use of the BLoC state management pattern [28] to manage the application's state. BLoC is a library for Dart and Flutter which helps splitting business logic from the presentation logic by introducing various state related elements. Thanks to BLoC, the application layer can elaborate in an efficient way the events, handle changes in the state and emit new states as responses to user actions and data changes.

4.2.1 Code BLoC

At the core of the Application Layer is the main code_bloc. This class is responsible for managing a series of operations essential to the smart contract analysis workflow. The key responsibilities of code_bloc include:

- **Selecting the Smart Contract File**: Allows the selection of a smart contract file from the user's file system. See graph 4.3.
- **Submitting the File for Analysis**: Initiates the analysis process by submitting the selected smart contract file to the microservices. This action triggers the creation of a new Task (refer to graph 4.4).
- **Polling for Task Status**: It continuously monitor the analysis status by sending periodic requests to the BFF. This mechanism allows to frontend to stay aware of the progress status and therefore it can update the user interface accordingly.
- **Updating the State**: Processes the results received from the repository and updates the state to reflect the analysis status. This includes handling successful results as well as managing errors in case of failures during any stage of the process.

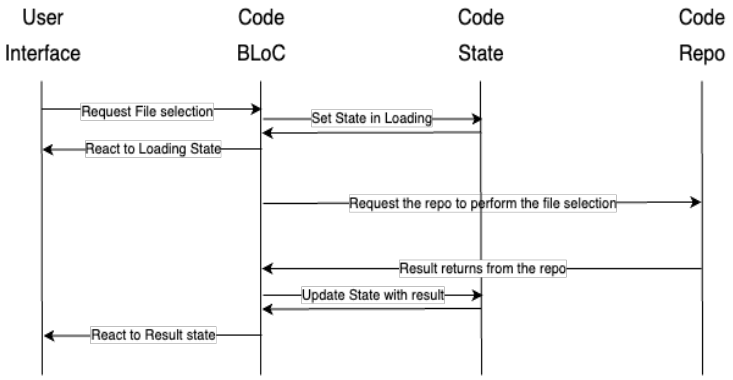

Figure 4.3: Application Layer - File selection process

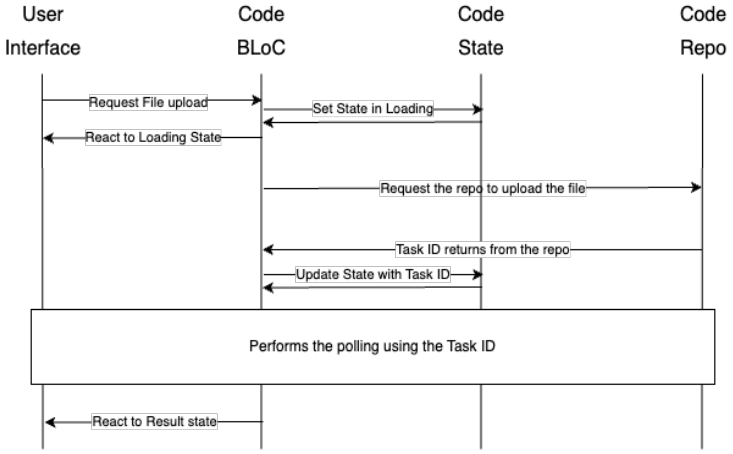

Figure 4.4: Application Layer - File upload process

4.3 Presentation Layer

The Presentation Layer is responsible for rendering the user interface and managing interactions. This layer reads the application's state and renders the visual interface accordingly. By separating the UI components from the business logic and data handling, the Presentation Layer ensures that the interface remains responsive, intuitive, and adaptable. In this project, the Presentation Layer is organized into distinct pages, each with specific functionalities:

4.3.1 Main Pages of the Application

The frontend comprises three primary pages: the Code Page, the Contract Page, and the Settings Page. Each of these pages serves a unique purpose, facilitating

different aspects of the smart contract analysis workflow.

Code Page

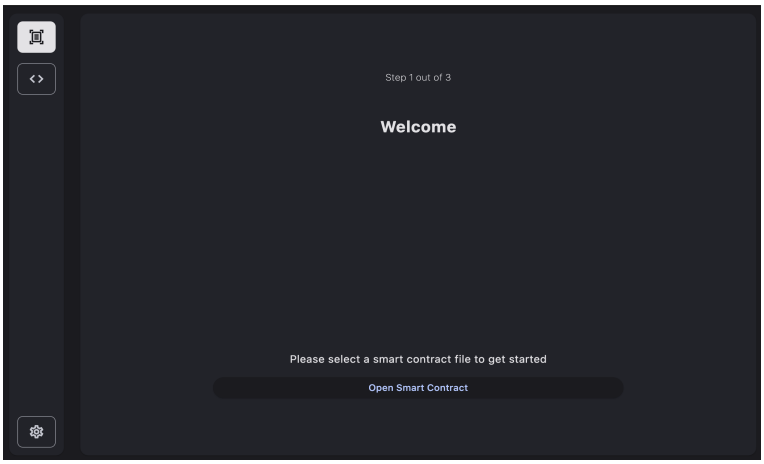

Figure 4.5: Presentation Layer - Code page welcome screen

The Code Page is the main interface of the application, acting as the entry point for users to initiate and view the results of smart contract analyses. Upon launching the application, this page displays the option to load a smart contract file from the local file system 4.5. This file selection interface allows the users to upload their Solidity contracts for analysis.

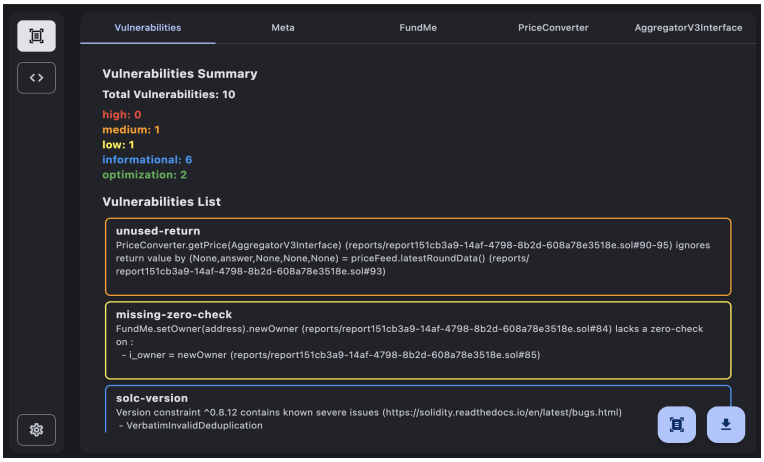

Figure 4.6: Presentation Layer - Code page analysis completed

Once a file is successfully loaded and the analysis is completed, the Code Page transitions to display the Editor Grid, a visual representation of the results 4.6. The

Editor Grid presents the smart contract's components in a grid format, allowing users to explore the relationships and attributes of various elements.

This dual functionality of the page, simplifies the user experience, guiding users from initiation to analysis within a single interface.

Contract Page

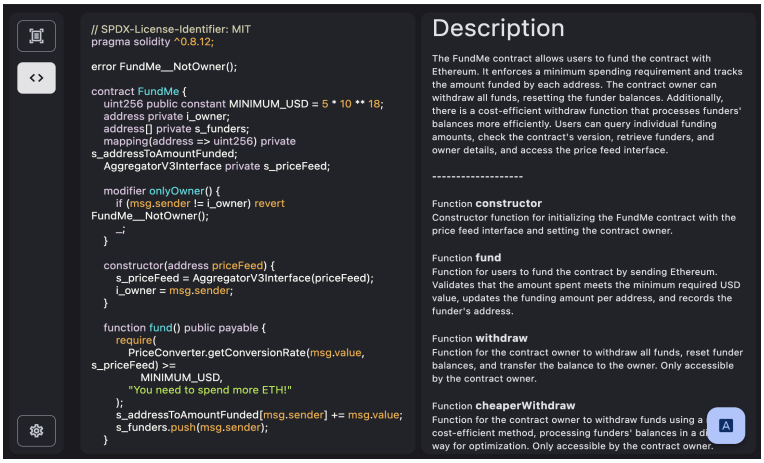

Figure 4.7: Presentation Layer - Contract page

The Contract Page offers a view of the smart contract's code alongside the description of its overall functionalities 4.7.

This page is designed to provide the user an immediate understanding of the structure and the scope of the smart contract. By showing the code and its description one beside the other, the page helps the user correlating specific segments of code with their functionalities.

This display format is beneficial for users who may not be familiar with Solidity syntax, as it bridges the gap between code and its real-world use. Additionally, the page serves as a centralized location for reviewing the contract's logic and ensuring that the implementation aligns with the intended design.

Settings Page

The Settings Page allows users to customize the app configurations to their specific preferences 4.8. This page provides the ability to set or update the OpenAI API Key, which is necessary for using the Assistant microservice's capabilities. Furthermore, the page offers theme customization options, enabling users to switch between Dark and Light modes.

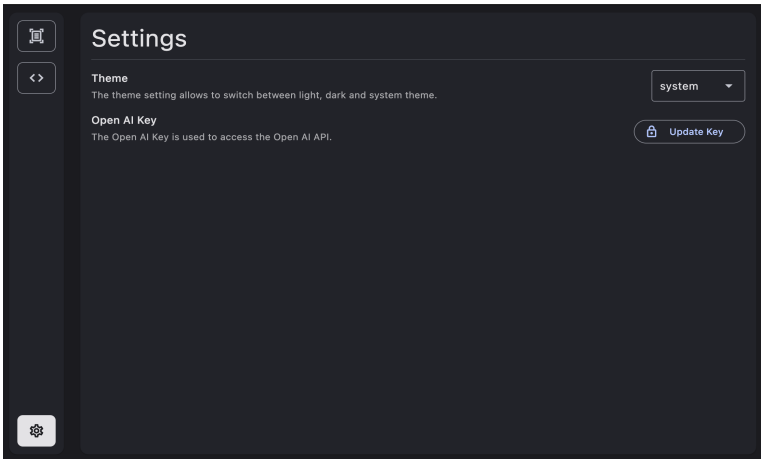

Figure 4.8: Presentation Layer - Settings page

4.4 Editor Grid

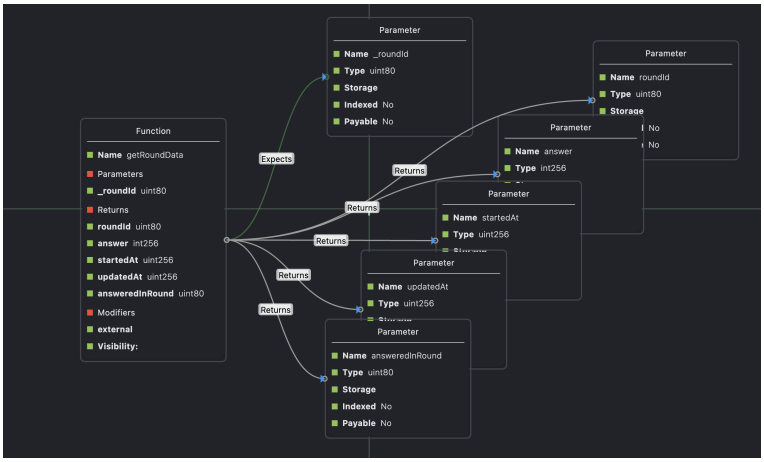

Figure 4.9: Editor grid - Elements relations

The Editor Grid is a custom made widget developed to act as the main interface to visualize and interact with the smart contract elements. The widget provides an intuitive environment based on a grid the allows users to explore and move around the contract's components without interacting directly with the code. This grid simplifies the comprehensions of the code architecture by showing the elements in a organized and navigable way. The Editor Grid provides a range of functionalities to enhance user interaction and facilitate comprehensive visualization of smart contract elements.

4.4.1 Element Display

The grid can display various elements by referring to specific points using Cartesian coordinates. This positioning ensures that each component of the smart contract, such as functions, variables, and properties, is accurately represented within the grid layout.

Each element can be re-rendered by changing its position coordinates. This mechanic allows for updates to the grid, ensuring that the representation remains consistent with modifications or rearrangements. As elements are added, removed, or repositioned, the grid adjusts to reflect these changes, maintaining an updated and accurate representation of the contract's structure.

4.4.2 Drag Operations

The Grid provides support for drag interactions by updating the elements coordinates according. The users can move as the like over to explore the content, while this mechanism moves the whole interface in real time.

Instead of just moving around, users con also move each single element by dragging it with the mouse. This ability to dynamically change items' position helps to visualize and better understand the flows and the dependencies. Moreover, it can be used to highlight specific relations or functionalities.

Thanks to this feature, the user experience is improved by a lot, allowing intuitive manipulation of elements. Based on user preference, the grid can be organized and customized.

4.4.3 Connection Lines

Each element can be related to another one by a parental or functional relationship. To represent those links visually, the Editor Grid renders connections lines between items. These lines are adapting and following when the elements are moved around, providing updated visual clues on dependencies and interactions. The image 4.9 is an example that displays how a function is expecting a specific parameter and it can return many values when its execution is terminated.

4.4.4 Interaction with Elements

One key feature of the frontend is the ability for users to interact with individual elements within the Editor Grid. This interaction provides a way for managing and customizing smart contract components.

Users can tap on any element within the Editor Grid to select and highlight it. After selecting an element, a side window automatically opens to display details about the component, including attributes such as code snippets, descriptions, and

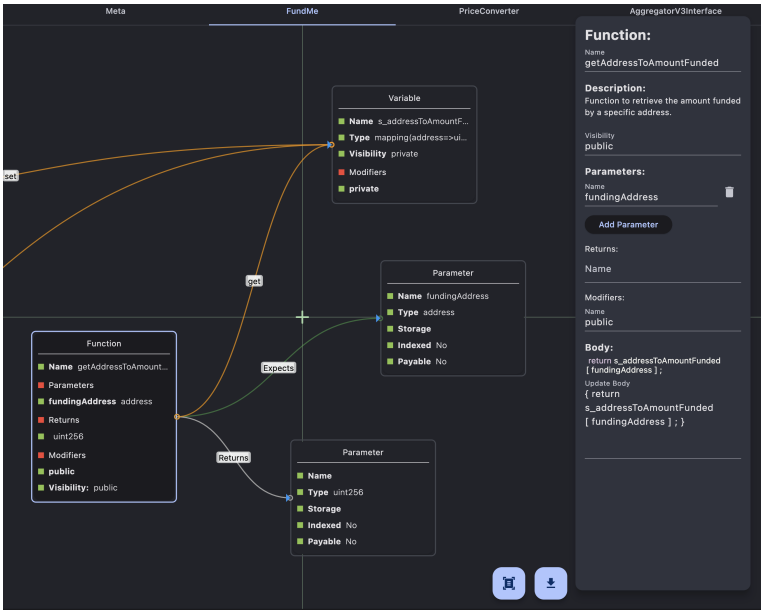

Figure 4.10: Editor grid - Function details

other relevant metadata This serves as a dedicated panel for viewing and editing the properties of the element 4.10

Certain properties of the element, such as the body, are editable directly within the side window. This functionality allows to modify specific aspects of the smart contract components without altering the entire structure.

After modifying the properties of an element, the properties are applied immediately, and the user can then initiate an Encoding Request. This process involves uploading the updated JSON structure to the BFF, which will later send it to the Codec microservice to regenerate the Solidity smart contract code based on the latest modifications.

This action results in a new smart contract file being downloaded and saved locally in the download folder. The file contains the smart contract's code with the changes applied.

Chapter 5

Results

To verify the effectiveness and results obtained by the tool, a survey was conducted targeting individuals with different levels of expertise in programming and blockchain technologies. The survey was divided into sections, starting from a general information questionnaire and following a task-based evaluation accompanied by the NASA Task Load Index (NASA-TLX) survey.

As for the evaluation of the user experience, we performed an a prior matched-pair Wilcoxon signed-rank analysis to obtain a power of 0.8 with an effect size of 0.5. The minimum number of user samples to achieve the above power results in the evaluation of at least 35 subjects.

To get the best focus level from the user and make the tests stand for a valid amount of time, a limit of five minutes was introduced for each of the tasks presented in the task-based evaluation section.

5.1 Survey Design

The initial portion of the survey collected background information from participants. This included data on age, educational level, current occupation, programming experience, and knowledge of blockchain technology. Collecting this information was needed to understand the user base and to evaluate their responses and feedbacks considering their background.

Following the first section, users were asked to interact with the software through a series of questions.

The participants received the smart contract file `MultisignWallet.sol` along with necessary instructions and an OpenAI API key to allow the use of the tool. The specific tasks involved answering the following questions:

1. Identify the types of the three parameters required by the `submitTransaction` function, listing them in order.

2. Determine the five types returned by the `getTransaction` function, listing them in order.

3. Identify all functions within the contract that emit the `SubmitTransaction` event.

4. Describe the outcome when `revokeConfirmation` is called with a `_txIndex` that does not correspond to any existing transaction.

Questions from 1 to 3 were proposed to evaluate graphical identification of basic smart contracts concepts, like function parameters, data types and events. A developer with experience in Solidity should be able to identify those values immediately, but a person that doesn't know the language syntax could find it more difficult. The last question requires knowledge on how the Solidity code behaves within both the function and the whole contract context. People that are not experts should find the description functionality offered by the tool to be helpful in explaining the logics.

It follows a section where participants were asked to answer specific questions using another smart contract, `FundMe.sol`, and the Remix (IDE). The questions are the following:

1. Identify which function invokes `getConversionRate` from the `PriceConverter` library.

2. Explain the purpose and functionality of the `latestRoundData` function.

3. Describe the implementation differences between the `withdraw` and `cheaperWithdraw` functions that contribute to the reduced cost of the latter.

4. Identify the vulnerability affecting the `setOwner` function and provide a corrected version of the function with the necessary modifications.

First question requires the user to understand the interaction of a smart contract with libraries. The second and third one are highly dependent on how much of the contract logic the user has understand. These are important questions to test the AI Assistant service functionalities. Finally the fourth question allows to test the vulnerability identification feature.

Each task-based section of the survey was followed by the administration of the NASA Task Load Index (NASA-TLX) survey. The NASA-TLX is a widely recognized tool for measuring perceived workload, including mental demand, physical demand, temporal demand, performance, effort, and frustration level. By integrating this survey after each set of tasks, the evaluation tried to compare the workload between the developed tool and Remix.

The tested user base was divided in two groups which were each tasked with a different version of the survey. The Survey-A was following the structure described previously. The Survey-B required the users to evaluate the developed tool in combination with `FundMe.sol` instead of `MultisignWallet.sol`, and doing the opposite when working with Remix. This aims to reduce the differences in the results due to some questions being slightly better suited for the tool or Remix.

5.2 Survey results

Table 5.1 contains general information collected from survey participants. The table represents each user with an ID, which will also appear in the other tables. The middle columns present the user skills that can be considered important for the survey, with possible values between 1 and 5. The last cell of each record refers to the survey type, which can be either A or B depending on the survey assigned. The results present exactly 18 participants for each group.

From the table, we can notice an average age of 27.6 with a valid score of 3.1 in programming experience. It is common for users to have good skills in the computer science field, but less in more specific blockchain topics.

From Table 5.2 the results of the questions about `MultisignWallet.sol` can be read. Users of group A were asked to provide answers by using the tool developed in the thesis, while group B was using the Remix.com IDE. Each cell contains a value from 0% to 100%. Considering a reference time of five minutes required to answer each question, this score is the ratio between this value and the actual time spent by the user. 0% means the user was able to provide a valid answer immediately, while 100% represent an answer given after the time limit. If the user answer was empty, or it was totally wrong, the resulting score would be considered 100% event if the time taken was less than the limit.

The last row compares the average results between group A and group B. This results in a comparison between the help that the two options are offering to the users to answer the same questions. We can deduct that the developed software allows the users to spend less time to answer all of the four questions. The first two tasks are presenting the biggest difference against Remix.com, suggesting that answering those is facilitated by the graphic interface provided in the tool. The last two questions are again performing slightly better with the tool, but the difference in time is less the the previous two.

The standard deviations represent the variability of performance within each group. The values are slightly higher for the group A using the thesis software, indicating greater variability in the results compared to Remix. This suggests that the software performance is less consistent, even if it leads to better averages.

The p-values indicate the statistical likelihood that the thesis software performs

ID	Age	Programming Experience	Time spent in programming-related activities each week	Experience in blockchain-related topics (e.g., smart contracts, Solidity programming, etc.)	Survey Type
1	35	5	4	1	A
2	22	3	4	2	A
3	30	4	4	2	A
4	34	4	3	1	A
5	26	4	2	2	A
6	26	5	5	4	A
7	27	1	1	5	A
8	26	4	3	2	A
9	25	4	4	3	A
10	29	1	1	3	A
11	27	4	4	1	A
12	32	2	2	1	A
13	31	2	2	2	A
14	34	5	5	4	A
15	29	4	4	2	A
16	27	1	1	1	A
17	27	1	1	1	A
18	27	1	1	1	A
19	27	1	1	1	B
20	25	3	3	1	B
21	27	1	1	1	B
22	29	5	5	5	B
23	30	3	2	2	B
24	27	3	3	2	B
25	41	3	3	2	B
26	25	4	4	3	B
27	24	4	3	2	B
28	29	2	3	1	B
29	26	1	1	2	B
30	22	1	1	1	B
31	19	3	1	1	B
32	23	5	2	5	B
33	27	4	3	2	B
34	25	4	3	1	B
35	30	5	4	5	B
36	22	3	3	1	B
AVG	**27.6**	**3.1**	**2.7**	**2.1**	-

Table 5.1: Survey results - General information

better than Remix. For task one and four, the values obtained are close to the threshold of p-value < 0.05, suggesting some limited evidence that the software outperforms the web IDE. Task two and three results in a much higher p-value that do not represent a valid evidence of a difference in performance.

ID	Identify the types of the three parameters required by the `submitTransaction` function, listing them in order	Determine the five types returned by the `getTransaction` function, listing them in order	Identify all functions within the contract that emit the `SubmitTransaction` event	Describe the outcome when `revokeConfirmation` is called with a `_txIndex` that does not correspond to any existing transaction
Survey Group A - Using the Software				
1	19.3%	13.08%	26.5%	30.45%
2	56.82%	59.89%	66.58%	49.03%
3	38.63%	32.47%	40.57%	45.54%
4	53.66%	56.54%	56.34%	63.68%
5	58.15%	62.5%	70.69%	74.49%
6	12.34%	16.25%	21.9%	23.47%
7	100%	86.78%	89.25%	92.63%
8	49.52%	44.08%	54.37%	59.68%
9	31.89%	28.75%	36.62%	41.9%
10	87.45%	90.84%	93.72%	100%
11	38.14%	18.52%	48.79%	54.37%
12	77.89%	82.63%	85.47%	88.35%
13	71.56%	76.72%	81.24%	86.49%
14	14.32%	10.87%	18.76%	26.39%
15	38.52%	34.27%	50.18%	46.89%
16	96.84%	100%	100%	84.75%
17	98.47%	99.12%	100%	97.84%
18	100%	100%	100%	100%
Survey Group B - Using Remix.com				
19	97.26%	100%	96.9%	100%
20	66.73%	77.39%	63.01%	60.91%
21	96.89%	100%	100%	95.68%
22	20.64%	10.42%	19.29%	15.75%
23	76.27%	66.55%	79.55%	68.56%
24	77.69%	61.17%	78.57%	73.37%
25	81.99%	78.97%	70.53%	83.45%
26	85.05%	81.94%	75.09%	72.27%
27	87.53%	62.2%	69.07%	74.17%
28	98.56%	100%	100%	100%
29	100%	100%	100%	100%
30	100%	100%	97.74%	100%
31	66.26%	85.75%	64.5%	81.39%
32	24.59%	13.19%	28.24%	31.26%
33	60.99%	78.67%	68.39%	78.6%
34	74.9%	64.36%	59.83%	100%
35	22.09%	24.77%	30.42%	91.9%
36	76.14%	59.72%	67.98%	77.08%
Software/Remix.com				
AVG	**57.97%**/72.98%	**56.30**/70.28%	**63.39%**/70.51%	**64.78%**/78.02%
STD	0.30/0.26	0.32/0.29	0.28/0.25	0.26/0.24
p-value	0.06	0.09	0.21	0.06

Table 5.2: Survey results - `MultisignWallet.sol`

Table 5.3 presents the results of tasks performed on the `FundMe.sol` file. The structure presented is the same as in the previous table, but here the survey group B is kept in the upper section to maintain that division between the thesis software and Remix.com.

By taking a look at those data some differences are showing up compared to the table 5.2. The average time for each task are higher, suggesting that `FundMe.sol` questions required more effort than those from `MultisignWallet.sol`. The first three tasks are scoring a better average time with the software, but the last question

ID	Identify which function invokes `getConversionRate` from the `PriceConverter` library	Explain the purpose and functionality of the `latestRoundData` function	Describe the implementation differences between the `withdraw` and `cheaperWithdraw` functions that contribute to the reduced cost of the latter	Identify the vulnerability affecting the `setOwner` function and provide a corrected version of the function with the necessary modifications
Survey Group B - Using the Software				
19	100%	81.62%	100%	100%
20	100%	100%	100%	100%
21	84.46%	83.08%	100%	100%
22	25.89%	39.79%	38.98%	36.74%
23	88.35%	100%	100%	100%
24	83.79%	86.81%	100%	100%
25	100%	81.9%	100%	100%
26	37.63%	37.75%	50.92%	32.54%
27	31.39%	49.76%	31.11%	28.65%
28	83.44%	88.62%	23.14%	100%
29	100%	100%	100%	100%
30	85.23%	80.65%	100%	100%
31	83.97%	82.74%	45.42%	100%
32	45.52%	39.08%	36.76%	34.17%
33	31.19%	27.44%	20.65%	47.59%
34	46.71%	39.93%	37.89%	57.91%
35	35.58%	49.19%	12.98%	40.1%
36	88.34%	83.79%	100%	100%
Survey Group A - Using Remix.com				
1	59.34%	69.29%	61.84%	45.85%
2	100%	100%	100%	100%
3	59.59%	53.8%	48.35%	39.73%
4	56.27%	61.61%	57.05%	33.19%
5	57.63%	62.69%	41.77%	59.2%
6	44.8%	57%	46.79%	41.25%
7	100%	100%	100%	100%
8	60.97%	50.6%	68.19%	53.63%
9	50.66%	41.49%	62.95%	62.54%
10	100%	100%	100%	100%
11	60.32%	65.3%	66.18%	52.24%
12	100%	100%	100%	100%
13	100%	100%	100%	100%
14	44.28%	60.89%	49.44%	38.64%
15	41.55%	52.08%	56.76%	58.77%
16	100%	100%	100%	100%
17	100%	100%	100%	100%
18	100%	100%	100%	100%
Software/Remix.com				
AVG	**69.53%**/74.19%	**69.56%**/76.38	**66.55%**/75.52%	76.54%/**71.39%**
STD	0.28/0.24	0.25/0.23	0.35/0.23	0.31/0.27
p-value	0.30	0.20	0.29	0.30

Table 5.3: Survey results - `FundMe.sol`

is actually showing a better value for Remix.com. This outcome could be due to the software not giving the user a proper edge for it. The question scores seem to be more related to the user skills in programming and in blockchain knowledge. Furthermore the tools is not easily providing help to answer the question as it was doing for other tasks.

The group using the thesis software tends to have a higher standard deviation, as for table 5.2, confirming a less consistent performance compared to Remix.

None of the p-values are below the common significance threshold (<0.05), with values that are relatively much higher, making it difficult to conclusively state the software superiority against Remix.

5.3 NASA-TLX results

The NASA-TLX survey allows to evaluate the user workload during the previously described tasks, over six different dimensions: mental demand, physical demand, temporal demand, performance, effort and frustration.

It is composed by two parts. The first one asks the user to provide a numerical score with a maximum value of 10, for each of the six dimensions. The second part allows to compute the weights by asking the user to choose between two dimensions the one that has impacted the workload the most. This is repeated for each possible pair, and the results are later normalized.

The graphs 5.4 show the NASA-TLX results. In the following order we have raw ratings, weight, adjusted ratings and overall ratings.

5.3.1 Raw Ratings

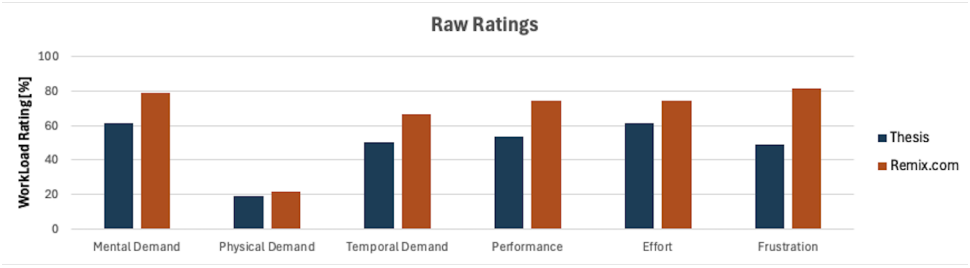

Figure 5.1: Survey results - NASA TLX - Raw ratings

- **Mental Demand**: the resulting score is on average lower using the software (61.67) compared to Remix (78.89). The web IDE does not provide direct guidance on how to answer the question, while the tools provided in the thesis software are more straightforward.

- **Physical Demand**: the users scored on average lower with the software (18.89) than with Remix (21.94). The difference is not relevant enough to consider it, and both the solutions are not really aiming to reduce the physical demand.

- **Temporal Demand**: the difference in temporal demand is more important between the software (50.56) and Remix (66.94). The user considers the thesis solution a less temporal demanding one, proving that the analysis and the graphic interface are providing information faster than the IDE.
- **Performance**: the resulting performance score are also marking an important difference close to a 21% in favor of the software. This can be explained by all the tasks accomplished in a small amount of time, and the immediate smart contract visual representation. the user feels more empowered by the solution.
- **Effort**: the results are higher by 13 points for Remix. This means the user is putting more effort when looking for the answer using Remix, especially if they don't know it already.
- **Frustration**: the difference in frustration scores is way higher, with Remix scoring 81.39 and the tool 49.17 (32% distance). Using a platform like Remix results in many trials and errors when looking for the task answer, especially from unexperienced users. On the other hand the thesis software is more straightforward in providing the information necessary for giving an answer.

5.3.2 Weights

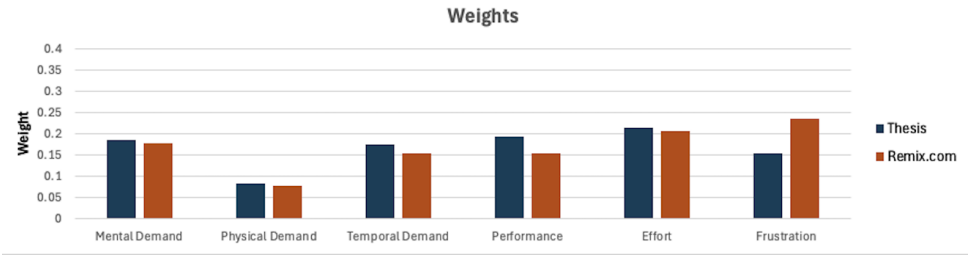

Figure 5.2: Survey results - NASA TLX - Weights

- **Mental Demand**: on average the weights score is close between the two solutions, with 0.18 for both Remix and the software.
- **Physical Demand**: the users score an average low weight for both the tools, placing it at around 0.08.
- **Temporal Demand**: on average the weights are 0.17 for the thesis software and 0.15 for Remix.
- **Performance**: the value is on average placed at 0.19 for the software, against 0.15 for Remix.

- **Effort**: the users score is on average at 0.21 for both the thesis and Remix.
- **Frustration**: the value is on average much higher on the Remix with a score of 0.24, while the thesis one is placed at 0.15.

5.3.3 Adjusted Ratings

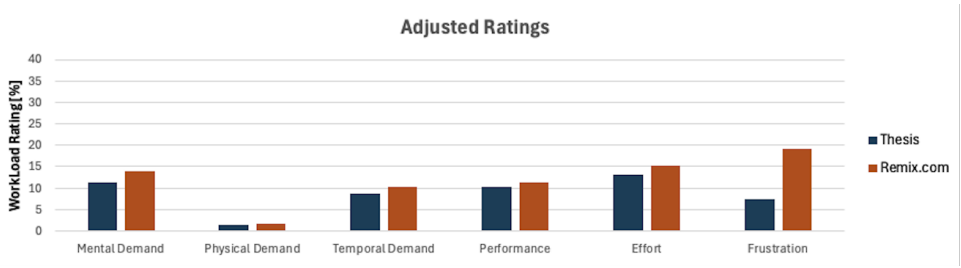

Figure 5.3: Survey results - NASA TLX - Adjusted ratings

- **Mental Demand**: the average score is around 3 points higher for Remix (14.02) than the software (11.30). This result confirms the raw data, telling that the software more guided approach is reducing the mental demand required from the user user, compared to Remix.
- **Physical Demand**: both Remix (1.51) and the thesis (1.53) are scoring on average very low on this dimension.
- **Temporal Demand**: on average Remix is scoring a value of 10.16, against the software value of 8.80. The thesis software is able to reduce the time needed to get to the task answer, thanks to the analysis and the graphic interface.
- **Performance**: the user score is on average slightly higher on Remix (11.35) then the software (10.33), highlighting the effect of the weight score, which reduced the difference between the two.
- **Effort**: on average the Remix score of 15.4 is higher than the thesis score of 13.19. This confirms the raw data by telling that the user require less effort when looking for the solution using the software, compared to Remix.
- **Frustration**: the difference in frustration score on average has increased by a lot, having a Remix score of 19.14, while the software is scoring 7.46. It is immediate to notice that the user is feeling much more frustration when working with Remix, trying to answer the task without a guided approach.

This result must be driven especially by people which don't have experience with the IDE and is interacting with a complex interface even when facing a relatively straightforward task.

5.3.4 Overall Ratings

The overall NASA TLX ratings are highlighting an advantage for the thesis software, with a score of 52.62, against Remix.com with 71.74. The software exposes its capacity of reducing mental demand, temporal demand, effort and frustration, reflecting its user-friendly, task-oriented design. Even tho the physical demand score remain comparable between the tools, the software ability to simplify access to information is drastically reducing the frustration. This proves its ability to provide a more efficient and less demanding experience, even for people that are not skilled in programming or smart contract development.

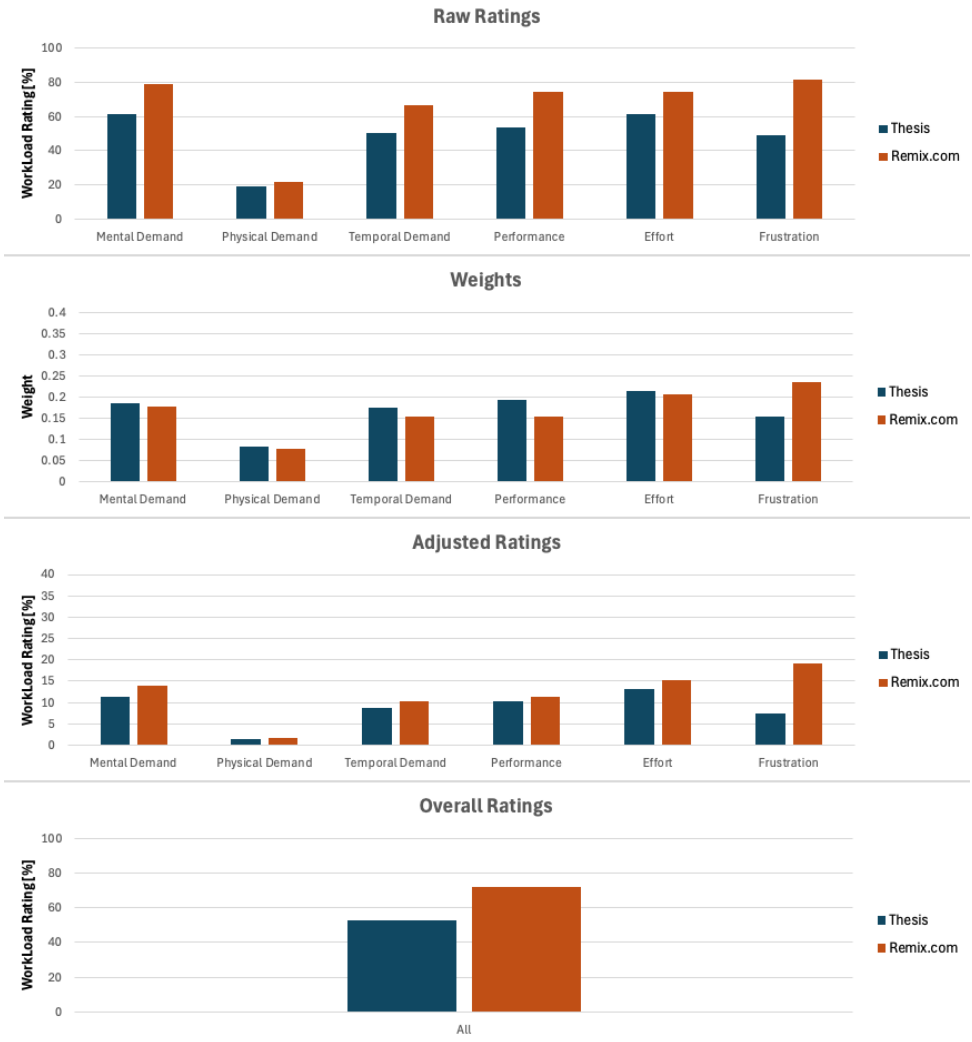

Figure 5.4: Survey results - NASA TLX - Overall ratings

Chapter 6

Conclusions

This thesis set out to address challenges in Solidity Smart Contract development by developing a system that simplifies the process of reading, editing, and performing analysis over smart contracts. The software met its initial objectives by integrating three key features: It provides a graphical representation, it integrates an LLM, and it performs security analysis.

The Graphical Representation was realized thanks to the realization of a microservices architecture that splits the smart contract into each of its fundamental components. Those are then assembled into a JSON format. This approach enables the frontend to render each element over a graphical grid, allowing users to interact with it regardless of their technical proficiency. The process concludes by providing the user the ability to convert back that graphical representation into functioning code.

The software integrates ChatGPT APIs to allow generating human-readable descriptions, represent functional relationships, and identify functions that can be improved. Thanks to the use of function calling, the system ensures that responses from the LLM are structured and follow the requirements, making it easier to integrate those results into existing data structures. This feature aids users in understanding complex code logics, and it also makes it easier to debug and optimize the contract.

Finally, the Security Analysis is provided through the use of Slither, a static analysis tool integrated into the Auditor microservice. This tool allows for the automatic detection and reporting of vulnerabilities. The result obtained is the reduction of the risks associated with the intrinsic irreversible nature of blockchain transactions. This approach to security ensures that the user can react to highlighted vulnerabilities before deployment.

The test conducted to evaluate the software against Remix.com resulted in positive results, with average performances better across tasks. On the contrary, the analysis revealed limitations in proving the software's superiority. The variability

in user responses and the lack of statistically significant p-values suggest that the observed results may not be robust. The users participating in the thesis survey presented a wide pool of programming skills and blockchain knowledge. Deciding to apply a value of 100% (failed) when tasks' responses were technically wrong, even if given within the time limit, could have resulted in an inflated standard deviation, potentially impacting the p-values calculated. Future evaluations could benefit from a more precise scoring system that is able to distinguish between levels of correctness.

The NASA-TLX results support the software's potential advantages over Remix, by highlighting reduced mental and temporal demands, as well as lower effort and frustration. These findings support the software as a valid solution to reduce cognitive load and improve productivity.

Overall, the system developed in this thesis bridges the gap between complex smart contract development and accessibility, by making use of other available technologies such as LLMs, Static Analysis tools, and a network of microservices, providing a solid structure for future enhancements.

Appendices

.1 AI assistant - Comment setup prompt

Listing 1: Setup prompt for comment process

```
The AI will receive a Solidity code as a string and a JSON
    representing a parsed structure of this code. The JSON contains
    details about each Solidity element, such as contracts, functions,
     and variables. Each element has a unique ID and a description
    field. The descriptions are initially empty and need to be
    populated by the AI. After generating the descriptions, the AI
    must conceptually prepare to call a specific function with the ID
    and the generated description as parameters, without actually
    performing the call.


THIS PROMPT IS ONLY A SETUP FOR LATER REQUEST.
DO NOT PERFOM FUNCTION CALLS AS RESPONSE TO THIS FIRST PROMPT.
ANSWER ONLY WITH "OK" IF EVERYTHING IS CLEAR.

Given the following example Solidity code and JSON structure:

Solidity Code:

// SPDX-License-Identifier: MIT
//
// https://cryptomarketpool.com/deposit-14-eth-game-in-a-solidity-
    smart-contract/

pragma solidity ^0.8.0;

// A game where the 14th person that deposit ether wins all the ether
     in the contract
// the winner can claim the 14 ether

contract EthGame {
    uint256 public targetAmount = 14 ether;
    address public winner;

    uint256 public balance;

    function deposit() public payable {
        require(msg.value == 1 ether, "You can only send 1 Ether");
        balance += msg.value;
        require(balance <= targetAmount, "Game is over");

        if (balance == targetAmount) {
            winner = msg.sender;
        }
    }
```

```
36
37      function claimReward() public {
38          require(msg.sender == winner, "Not winner");
39
40          (bool sent, ) = msg.sender.call{value: address(this).balance
        }("");
41          require(sent, "Failed to send Ether");
42      }
43
44      function getBalance() public view returns (uint256) {
45          return address(this).balance;
46      }
47  }
48
49  JSON Structure:
50
51  {
52      "contracts": [
53          {
54              "description": "",
55              "functions": [
56                  {
57                      "body": "{ require ( msg . value == 1 ether , \"
        You can only send 1 Ether\" ) ; balance += msg . value ; require (
         balance <= targetAmount , \"Game is over\" ) ; if ( balance ==
        targetAmount ) { winner = msg . sender ; } }",
58                      "description": "",
59                      "id": "577d831c-0350-4789-82fc-6ad4b93e7841",
60                      "isConstructor": false,
61                      "isFallback": false,
62                      "isReceive": false,
63                      "modifiers": ["public"],
64                      "name": "deposit",
65                      "stateMutability": "payable",
66                      "visibility": "public"
67                  },
68                  {
69                      "body": "{ require ( msg . sender == winner , \"
        Not winner\" ) ; ( bool sent , ) = msg . sender . call { value :
        address ( this ) . balance } ( \"\" ) ; require ( sent , \"Failed
        to send Ether\" ) ; }",
70                      "description": "",
71                      "id": "a47acf50-0907-4663-bb1f-38117a2a42f6",
72                      "isConstructor": false,
73                      "isFallback": false,
74                      "isReceive": false,
75                      "modifiers": ["public"],
76                      "name": "claimReward",
77                      "stateMutability": "",
```

```
                    "visibility": "public"
                },
                {
                    "body": "{ return address ( this ) . balance ;
    }",
                    "description": "",
                    "id": "058f661c-2c5e-4eed-ab92-72b102f19ee7",
                    "isConstructor": false,
                    "isFallback": false,
                    "isReceive": false,
                    "modifiers": ["public"],
                    "name": "getBalance",
                    "returns": [
                        {
                            "description": "",
                            "id": "db564ead-3440-43ca-9a5a-8
    caffab32da2",
                            "isIndexed": false,
                            "name": "",
                            "payable": false,
                            "storage": "",
                            "type": "uint256"
                        }
                    ],
                    "stateMutability": "view",
                    "visibility": "public"
                }
            ],
            "id": "d00c683a-45fa-43e9-beae-9ac284015391",
            "isInterface": false,
            "isLibrary": false,
            "name": "EthGame",
            "variables": [
                {
                    "description": "",
                    "id": "de63f8df-2cc7-4313-8b80-ed4b8e4bc287",
                    "mappingFrom": "",
                    "mappingTo": "",
                    "modifiers": ["public"],
                    "name": "targetAmount",
                    "type": "uint256",
                    "value": "14 ether",
                    "visibility": "public"
                },
                {
                    "description": "",
                    "id": "32c3f635-7b6d-44ae-8d6d-e5941843cf7a",
                    "mappingFrom": "",
                    "mappingTo": "",
```

```
                    "modifiers": ["public"],
                    "name": "winner",
                    "type": "address",
                    "value": "",
                    "visibility": "public"
                },
                {
                    "description": "",
                    "id": "eb5025c8-9650-4a23-8ec2-b0f873af2417",
                    "mappingFrom": "",
                    "mappingTo": "",
                    "modifiers": ["public"],
                    "name": "balance",
                    "type": "uint256",
                    "value": "",
                    "visibility": "public"
                }
            ]
        }
    ],
    "description": "",
    "id": "5084631a-533d-43e0-ac61-eabb5a5ee212",
    "pragmas": [
        {
            "description": "",
            "id": "602e5b30-ef56-4edb-ba2d-84a38ba92d24",
            "name": "solidity",
            "value": "^0.8.0"
        }
    ]
}

Instructions for the AI:

Parse the JSON Structure:

Identify each element (contracts, functions, variables, pragmas)
    within the JSON.
Generate Descriptions:

For each element, generate a detailed description based on the
    Solidity code and the context provided in the JSON. The
    description should include:
Function Signature: Explain the function name, parameters, visibility
    , state mutability, and return type.
Detailed Explanation: Step-by-step explanation of what the function
    does, including conditions, loops, assignments, and external calls
    .
Purpose: Explain the purpose and expected behavior of the function.
```

```
Conceptual Function Call:

After generating each description, conceptually prepare to call the function set_description_by_id with the parameters id and description, without actually performing the call.
The function signature is: set_description_by_id(string ID, string description).

Example Function Call Sequence:

For the deposit function:

# set_description_by_id("577d831c-0350-4789-82fc-6ad4b93e7841", "Function to deposit 1 ether into the contract, updating the balance and potentially setting the winner.")
For the claimReward function:

# set_description_by_id("a47acf50-0907-4663-bb1f-38117a2a42f6", "Function for the winner to claim all ether in the contract.")
For the getBalance function:

# set_description_by_id("058f661c-2c5e-4eed-ab92-72b102f19ee7", "Function to get the current balance of the contract.")

Ensure Coverage:
Ensure that each element in the JSON has a corresponding description generated and conceptually updated via the set_description_by_id function call.
```

.2 AI assistant - Link setup prompt

Listing 2: Setup prompt for link process

```
You are an AI assistant tasked with analyzing Solidity smart contract code and a corresponding JSON structure representing the components of the code. Your objective is to identify and generate a comprehensive list of relations between the elements of the code, ensuring no relation is missed. You must then perform a function call with the generated relations.

THIS PROMPT IS ONLY A SETUP FOR LATER REQUEST.
DO NOT PERFOM FUNCTION CALLS AS RESPONSE TO THIS FIRST PROMPT.
ANSWER ONLY WITH "OK" IF EVERYTHING IS CLEAR.

Here are the steps you need to follow:

1. **Input Structure:**
```

```
   - You will receive a Solidity smart contract code and a JSON structure.
   - The Solidity code can contain multiple contracts, functions, and variables.
   - The JSON structure provides detailed information about these elements, including their IDs, descriptions, types, and other properties.

2. **Types of Relations:**
   - **Structural Relations:** These occur when one element contains another element. For example, a contract contains functions and variables.
   - **Functional Relations:** These occur when one element interacts with another element. For example, a function sets the value of a variable or calls another function.

3. **Relation Properties:**
   - `start`: The ID of the element where the relation starts (taken from the JSON structure).
   - `end`: The ID of the element where the relation ends (taken from the JSON structure).
   - `action`: Describes in one word the action taken by the start element on the end element.
     - The action describes the specific interaction (e.g., 'set', 'check', 'transfer', 'call', etc.). Feel free to add many more!
   - 'description': Describes how the two elements are related and what kind of change their relation performs.

4. **Example Input:**
   - Solidity Code:
     ```
     // SPDX-License-Identifier: MIT
     //
     // https://cryptomarketpool.com/deposit-14-eth-game-in-a-solidity-smart-contract/

     pragma solidity ^0.8.0;

     contract EthGame {
         uint256 public targetAmount = 14 ether;
         address public winner;

         uint256 public balance;

         // Event declaration
         event DepositMade(address indexed player, uint256 amount, uint256 totalBalance);

         function deposit() public payable {
```
```

```
            require(msg.value == 1 ether, "You can only send 1 Ether
    ");
            balance += msg.value;
            require(balance <= targetAmount, "Game is over");

            // Emit the event
            emit DepositMade(msg.sender, msg.value, balance);

            if (balance == targetAmount) {
                winner = msg.sender;
            }
        }

        function claimReward() public {
            require(msg.sender == winner, "Not winner");

            (bool sent, ) = msg.sender.call{value: address(this).
    balance}("");
            require(sent, "Failed to send Ether");
        }

        function getBalance() public view returns (uint256) {
            return address(this).balance;
        }
    }
    ```
- JSON Structure:
    ```
    {
        \"contracts\": [
            {
                \"description\": \"\",
                \"functions\": [
                    {
                        \"body\": \"{ require ( msg . value == 1
    ether , \\\"You can only send 1 Ether\\\" ) ; balance += msg .
    value ; require ( balance <= targetAmount , \\\"Game is over\\\" )
     ; if ( balance == targetAmount ) { winner = msg . sender ; } }\",
                        \"description\": \"\",
                        \"id\": \"577d831c-0350-4789-82fc-6
    ad4b93e7841\",
                        \"isConstructor\": false,
                        \"isFallback\": false,
                        \"isReceive\": false,
                        \"modifiers\": [\"public\"],
                        \"name\": \"deposit\",
                        \"stateMutability\": \"payable\",
                        \"visibility\": \"public\"
                    },
```

```
87                      {
88                          \"body\": \"{ require ( msg . sender ==
    winner , \\\"Not winner\\\" ) ; ( bool sent , ) = msg . sender .
    call { value : address ( this ) . balance } ( \\\"\\\" ) ; require
     ( sent , \\\"Failed to send Ether\\\" ) ; }\",
89                          \"description\": \"\",
90                          \"id\": \"a47acf50-0907-4663-bb1f-38117
    a2a42f6\",
91                          \"isConstructor\": false,
92                          \"isFallback\": false,
93                          \"isReceive\": false,
94                          \"modifiers\": [\"public\"],
95                          \"name\": \"claimReward\",
96                          \"stateMutability\": \"\",
97                          \"visibility\": \"public\"
98                      },
99                      {
100                         \"body\": \"{ return address ( this ) .
    balance ; }\",
101                         \"description\": \"\",
102                         \"id\": \"058f661c-2c5e-4eed-ab92-72
    b102f19ee7\",
103                         \"isConstructor\": false,
104                         \"isFallback\": false,
105                         \"isReceive\": false,
106                         \"modifiers\": [\"public\"],
107                         \"name\": \"getBalance\",
108                         \"returns\": [
109                             {
110                                 \"description\": \"\",
111                                 \"id\": \"db564ead-3440-43ca-9a5a-8
    caffab32da2\",
112                                 \"isIndexed\": false,
113                                 \"name\": \"\",
114                                 \"payable\": false,
115                                 \"storage\": \"\",
116                                 \"type\": \"uint256\"
117                             }
118                         ],
119                         \"stateMutability\": \"view\",
120                         \"visibility\": \"public\"
121                     }
122                 ],
123                 \"id\": \"d00c683a-45fa-43e9-beae-9ac284015391\",
124                 \"isInterface\": false,
125                 \"isLibrary\": false,
126                 \"name\": \"EthGame\",
127                 \"variables\": [
128                     {
```

```
                        \"description\": \"\",
                        \"id\": \"de63f8df-2cc7-4313-8b80-
    ed4b8e4bc287\",
                        \"mappingFrom\": \"\",
                        \"mappingTo\": \"\",
                        \"modifiers\": [\"public\"],
                        \"name\": \"targetAmount\",
                        \"type\": \"uint256\",
                        \"value\": \"14 ether\",
                        \"visibility\": \"public\"
                    },
                    {
                        \"description\": \"\",
                        \"id\": \"32c3f635-7b6d-44ae-8d6d-
    e5941843cf7a\",
                        \"mappingFrom\": \"\",
                        \"mappingTo\": \"\",
                        \"modifiers\": [\"public\"],
                        \"name\": \"winner\",
                        \"type\": \"address\",
                        \"value\": \"\",
                        \"visibility\": \"public\"
                    },
                    {
                        \"description\": \"\",
                        \"id\": \"eb5025c8-9650-4a23-8ec2-
    b0f873af2417\",
                        \"mappingFrom\": \"\",
                        \"mappingTo\": \"\",
                        \"modifiers\": [\"public\"],
                        \"name\": \"balance\",
                        \"type\": \"uint256\",
                        \"value\": \"\",
                        \"visibility\": \"public\"
                    }
                ]
            }
        ],
        \"description\": \"\",
        \"id\": \"5084631a-533d-43e0-ac61-eabb5a5ee212\",
        \"pragmas\": [
            {
                \"description\": \"\",
                \"id\": \"602e5b30-ef56-4edb-ba2d-84a38ba92d24\",
                \"name\": \"solidity\",
                \"value\": \"^0.8.0\"
            }
        ]
    }
```

```
    ```

5. **Output Structure:**
   - The output should be a list all identified relations. Each relation should have the properties: `start`, `end`, `type`, and `action`. You must perfom a function call for each relation.

6. **Example Output:**

{
"start": "577d831c-0350-4789-82fc-6ad4b93e7841",
"end": "eb5025c8-9650-4a23-8ec2-b0f873af2417",
"description": "The function is increasing the value of the variable balance depending on the import received.",
"action": "set"
},
{
"start": "577d831c-0350-4789-82fc-6ad4b93e7841",
"end": "32c3f635-7b6d-44ae-8d6d-e5941843cf7a",
"description": "The function checks if the maximum import is reached, and eventually sets the winner",
"action": "set"
},
{
"start": "a47acf50-0907-4663-bb1f-38117a2a42f6",
"end": "32c3f635-7b6d-44ae-8d6d-e5941843cf7a",
"description": "The function checks wether the sender is the winner.",
"action": "check"
},
{
"start": "a47acf50-0907-4663-bb1f-38117a2a42f6",
"end": "eb5025c8-9650-4a23-8ec2-b0f873af2417",
"description": "The balance is transferred to the winner.",
"action": "transfer"
},
{
"start": "058f661c-2c5e-4eed-ab92-72b102f19ee7",
"end": "eb5025c8-9650-4a23-8ec2-b0f873af2417",
"description": "Provides the balance value to the sender.",
"action": "return"
}

7. **Instructions:**
- Parse the Solidity code to identify contracts, functions, and variables.
- Use the JSON structure to map each element to its ID and properties.
```
```

```
- Identify all possible relations between the elements.
- Ensure each relation is accurately described with its `start`, `end`, `type`, and `action`.
- Obtain the list of releations
- **Perform the function call for each of the generated relations.**

After generating the relations, you need to perform a function call for each relation. The function name is 'set_relation'. The function call should have the following format:

set_relation(start, end, description, action)


Here are some examples of function calls:

set_relation('577d831c-0350-4789-82fc-6ad4b93e7841', 'eb5025c8-9650-4a23-8ec2-b0f873af2417', 'The function is increasing the value of the variable balance depending on the import received.', 'set')
set_relation('a47acf50-0907-4663-bb1f-38117a2a42f6', '32c3f635-7b6d-44ae-8d6d-e5941843cf7a', 'The function checks if the maximum import is reached, and eventually sets the winner', 'check')


You must generate and call this function for each identified relation.
```

.3 AI assistant - Warning setup prompt

Listing 3: Setup prompt for warning process

```
You are an AI assistant tasked with analyzing Solidity smart contract code and a corresponding JSON structure representing the components of the code. Your objective is to identify and generate a comprehensive list of IDs of the elements of the code that can be improved. You must then perform a function call with the list of IDs obtained.

THIS PROMPT IS ONLY A SETUP FOR LATER REQUEST.
DO NOT PERFOM FUNCTION CALLS AS RESPONSE TO THIS FIRST PROMPT.
ANSWER ONLY WITH "OK" IF EVERYTHING IS CLEAR.

Here are the steps you need to follow:

1. **Input Structure:**
   - You will receive a Solidity smart contract code and a JSON structure.
```

```
   - The Solidity code can contain multiple contracts, functions, and variables.
   - The JSON structure provides detailed information about these elements, including their IDs, descriptions, types, and other properties.

2. **Example Input:**
   - Solidity Code:
     ```
     // SPDX-License-Identifier: MIT
     //
     // https://cryptomarketpool.com/deposit-14-eth-game-in-a-solidity-smart-contract/

     pragma solidity ^0.8.0;

     contract EthGame {
         uint256 public targetAmount = 14 ether;
         address public winner;

         uint256 public balance;

         function deposit() public payable {
             require(msg.value == 1 ether, \"You can only send 1 Ether\");
             balance += msg.value;
             require(balance <= targetAmount, \"Game is over\");

             if (balance == targetAmount) {
                 winner = msg.sender;
             }
         }

         function claimReward() public {
             require(msg.sender == winner, \"Not winner\");

             (bool sent,) = msg.sender.call{value: address(this).balance}(\"\");
             require(sent, \"Failed to send Ether\");
         }

         function getBalance() public view returns (uint256) {
             return address(this).balance;
         }
     }
     ```
   - JSON Structure:
     ```
     {
```
```

```
54          \"contracts\": [
55              {
56                  \"description\": \"\",
57                  \"functions\": [
58                      {
59                          \"body\": \"{ require ( msg . value == 1
    ether , \\\"You can only send 1 Ether\\\" ) ; balance += msg .
    value ; require ( balance <= targetAmount , \\\"Game is over\\\" )
     ; if ( balance == targetAmount ) { winner = msg . sender ; } }\",
60                          \"description\": \"\",
61                          \"id\": \"577d831c-0350-4789-82fc-6
    ad4b93e7841\",
62                          \"isConstructor\": false,
63                          \"isFallback\": false,
64                          \"isReceive\": false,
65                          \"modifiers\": [\"public\"],
66                          \"name\": \"deposit\",
67                          \"stateMutability\": \"payable\",
68                          \"visibility\": \"public\"
69                      },
70                      {
71                          \"body\": \"{ require ( msg . sender ==
    winner , \\\"Not winner\\\" ) ; ( bool sent , ) = msg . sender .
    call { value : address ( this ) . balance } ( \\\"\\\" ) ; require
     ( sent , \\\"Failed to send Ether\\\" ) ; }\",
72                          \"description\": \"\",
73                          \"id\": \"a47acf50-0907-4663-bb1f-38117
    a2a42f6\",
74                          \"isConstructor\": false,
75                          \"isFallback\": false,
76                          \"isReceive\": false,
77                          \"modifiers\": [\"public\"],
78                          \"name\": \"claimReward\",
79                          \"stateMutability\": \"\",
80                          \"visibility\": \"public\"
81                      },
82                      {
83                          \"body\": \"{ return address ( this ) .
    balance ; }\",
84                          \"description\": \"\",
85                          \"id\": \"058f661c-2c5e-4eed-ab92-72
    b102f19ee7\",
86                          \"isConstructor\": false,
87                          \"isFallback\": false,
88                          \"isReceive\": false,
89                          \"modifiers\": [\"public\"],
90                          \"name\": \"getBalance\",
91                          \"returns\": [
92                              {
```

```
                                \"description\": \"\",
                                \"id\": \"db564ead-3440-43ca-9a5a-8caffab32da2\",
                                \"isIndexed\": false,
                                \"name\": \"\",
                                \"payable\": false,
                                \"storage\": \"\",
                                \"type\": \"uint256\"
                            }
                        ],
                        \"stateMutability\": \"view\",
                        \"visibility\": \"public\"
                    }
                ],
                \"id\": \"d00c683a-45fa-43e9-beae-9ac284015391\",
                \"isInterface\": false,
                \"isLibrary\": false,
                \"name\": \"EthGame\",
                \"variables\": [
                    {
                        \"description\": \"\",
                        \"id\": \"de63f8df-2cc7-4313-8b80-ed4b8e4bc287\",
                        \"mappingFrom\": \"\",
                        \"mappingTo\": \"\",
                        \"modifiers\": [\"public\"],
                        \"name\": \"targetAmount\",
                        \"type\": \"uint256\",
                        \"value\": \"14 ether\",
                        \"visibility\": \"public\"
                    },
                    {
                        \"description\": \"\",
                        \"id\": \"32c3f635-7b6d-44ae-8d6d-e5941843cf7a\",
                        \"mappingFrom\": \"\",
                        \"mappingTo\": \"\",
                        \"modifiers\": [\"public\"],
                        \"name\": \"winner\",
                        \"type\": \"address\",
                        \"value\": \"\",
                        \"visibility\": \"public\"
                    },
                    {
                        \"description\": \"\",
                        \"id\": \"eb5025c8-9650-4a23-8ec2-b0f873af2417\",
                        \"mappingFrom\": \"\",
                        \"mappingTo\": \"\",
```

```
                        \"modifiers\": [\"public\"],
                        \"name\": \"balance\",
                        \"type\": \"uint256\",
                        \"value\": \"\",
                        \"visibility\": \"public\"
                    }
                ]
            }
        ],
        \"description\": \"\",
        \"id\": \"5084631a-533d-43e0-ac61-eabb5a5ee212\",
        \"pragmas\": [
            {
                \"description\": \"\",
                \"id\": \"602e5b30-ef56-4edb-ba2d-84a38ba92d24\",
                \"name\": \"solidity\",
                \"value\": \"^0.8.0\"
            }
        ]
    }
    ```
3. **Identify the warnings:**
   - Identifying functions that can be improved in a Solidity smart contract involves analyzing the contract for potential security risks, inefficiencies, and usability issues.
   - This process includes checking for common vulnerabilities such as reentrancy, ensuring adherence to best practices like the Checks-Effects-Interactions pattern, and verifying that the contract's logic is flexible and safe for long-term use.
   - By carefully reviewing the flow of funds, external calls, and state changes, we can spot areas that need improvement to enhance both security and functionality.

4. **Output Structure:**
   - The output should be a list of IDs of identified functions. Each output should only have the property: `id`. You must perfom a function call for each identified warning.

5. **Example Output:**

{
"id": "a47acf50-0907-4663-bb1f-38117a2a42f6",
},
{
"id": "058f661c-2c5e-4eed-ab92-72b102f19ee7",
}

6. **Instructions:**
```
```

```
- Parse the Solidity code to identify contracts, functions, and variables.
- Use the JSON structure to map each element to its ID and properties.
- Identify all possible function that could be improved.
- Ensure each warning is accurately described with the function `id`.
- Obtain the list of warnings
- **Perform the function call for each of the generated warning.**

After generating the warnings, you need to perform a function call for each warning. The function name is 'set_warning'. The function call should have the following format:

set_warning(id)


Here are some examples of function calls:

set_warning('a47acf50-0907-4663-bb1f-38117a2a42f6')
set_warning('058f661c-2c5e-4eed-ab92-72b102f19ee7')


You must generate and call this function for each identified warning.
```

.4 AI assistant - Input prompt

Listing 4: Input prompt

```
Solidity Code:
```solidity
<<SOLIDITY_CODE>>
```

JSON Structure:
```json
<<JSON_STRUCTURE>>
```
```

.5 GRPC - Protocol buffers

Listing 5: Codec protobuf

```
syntax = "proto3";

```

```
package codec;

option go_package = "codec_service/;codec_service";

service CodecService {
  rpc Encode(EncodeRequest) returns (EncodeResponse);
  rpc Decode(DecodeRequest) returns (DecodeResponse);
}

message EncodeRequest {
  string smartContractCode = 1; // Input smart contract code to be encoded into JSON
}

message EncodeResponse {
  string jsonStructure = 1; // Encoded JSON structure of the smart contract
}

message DecodeRequest {
  string jsonStructure = 1; // Input JSON structure to be decoded back into smart contract code
}

message DecodeResponse {
  string smartContractCode = 1; // Decoded smart contract code
}
```

Listing 6: Auditor protobuf

```
syntax = "proto3";

package auditor;

option go_package = "auditor_service/;auditor_service";

service AuditorService {
  rpc Audit(AuditRequest) returns (AuditResponse);
}

message AuditRequest {
  string jsonStructure = 1;     // The JSON structure of the smart contract to be audited.
  string smartContractCode = 2;  // The original smart contract code in its source format.
}

message Vulnerability {
```

```
  string name = 1;                // The name or identifier of the security check that was triggered.
  string description = 2;         // A detailed description of the vulnerability found.
  string severity = 3;            // The severity level of the vulnerability (e.g., low, medium, high).
}

message AuditResponse {
  repeated Vulnerability vulnerabilities = 1;  // A list of vulnerabilities detected during the audit.
}
```

Listing 7: AI Assistant protobuf

```
syntax = "proto3";

package ai_assistant;

// Specify the Go package where the generated code should reside.
// Adjust the path according to your module path and desired package structure.
option go_package = "ai_assistant_service/;ai_assistant_service";

service AiAssistantService {
  rpc Comment(CommentRequest) returns (CommentResponse);
  rpc Link(LinkRequest) returns (LinkResponse);
  rpc Warning(WarningRequest) returns (WarningResponse);
}

message CommentRequest {
  string jsonStructure = 1;      // The JSON structure of the smart contract to be commented on.
  string smartContractCode = 2;  // The original smart contract code in its source format.
  string openAiKey = 3;          // The OpenAI API key to use for generating comments.
}

message CommentResponse {
  string jsonStructure = 1;  // The JSON structure with added comments or explanations.
}

message LinkRequest {
```

```
26    string jsonStructure = 1;      // The JSON structure of the ↩
       smart contract to be analyzed for links.
27    string smartContractCode = 2;  // The original smart ↩
       contract code in its source format.
28    string openAiKey = 3;          // The OpenAI API key to use ↩
       for generating comments.
29  }
30
31  message LinkResponse {
32    repeated Link links = 1;  // The list of links found in the ↩
       smart contract.
33  }
34
35  message Link {
36    string start = 1;          // The Id of the element where the↩
        relationship starts.
37    string end = 2;            // The Id of the element where the↩
        relationship ends.
38    string description = 3;    // The description of the ↩
       relationship.
39    string action = 4;         // The action that the ↩
       relationship describes.
40  }
41
42  message WarningRequest {
43    string jsonStructure = 1;      // The JSON structure of the ↩
       smart contract to be analyzed for warnings.
44    string smartContractCode = 2;  // The original smart ↩
       contract code in its source format.
45    string openAiKey = 3;          // The OpenAI API key to use ↩
       for generating comments.
46  }
47
48  message WarningResponse {
49    string Warnings = 1;  // The list of warnings found in the ↩
       smart contract
50  }
```

.6 REST - OpenAPI schema

Listing 8: Client API schema

```
1  openapi: 3.1.0
2  info:
3    title: Smart Contract Processing API
4    description: API for uploading a file, processing it asynchronously
      , and receiving progress updates and result
```

```yaml
  version: 1.0.0
servers:
  - url: http://0.0.0.0:8080/api
paths:
  /upload:
    post:
      summary: Upload a file for processing
      requestBody:
        required: true
        content:
          multipart/form-data:
            schema:
              type: object
              required:
                - file
                - openAiKey
              properties:
                file:
                  type: string
                  format: binary
                openAiKey:
                  type: string
                  description: OpenAI API key

      responses:
        "200":
          description: File successfully uploaded
          content:
            application/json:
              schema:
                type: object
                required:
                  - taskId
                properties:
                  taskId:
                    type: string
                    description: Identifier for the uploaded task
  /export/{taskId}:
    post:
      summary: Post the new Source unit into the task
      parameters:
        - in: path
          name: taskId
          required: true
          description: Identifier of the task to get status for
          schema:
            type: string
      requestBody:
        required: true
```

```yaml
        content:
          application/json:
            schema:
              type: object
              properties:
                sourceUnit:
                  type: object
                  description: Source unit to be downloaded for the
    task
    responses:
      "200":
        description: File successfully uploaded
        content:
          application/json:
            schema:
              type: object
              required:
                - contractCode
              properties:
                contractCode:
                  type: string
                  description: Identifier for the uploaded task
  /tasks/{taskId}:
    get:
      summary: Get the status of a processing task
      parameters:
        - in: path
          name: taskId
          required: true
          description: Identifier of the task to get status for
          schema:
            type: string
      responses:
        "200":
          description: Task status retrieved successfully
          content:
            application/json:
              schema:
                type: object
                required:
                  - id
                  - status
                  - progress
                  - statusMessage
                properties:
                  id:
                    type: string
                    description: Identifier of the task
                  status:
```

```
                    type: string
                    enum:
                      - processing
                      - completed
                      - failed
                    description: Current status of the task
                  result:
                    type: object
                    description: Result of the processing task (dynamic structure)
                  vulnerabilities:
                    type: object
                    description: Result of the slither analysis (dynamic structure)
                  links:
                    type: object
                    description: Contains the connections between elements (dynamic structure)
                  warnings:
                    type: object
                    description: Contains the ids of functions which can be improved (dynamic structure)
                  progress:
                    type: integer
                    description: Percentage of completion of the task
                  statusMessage:
                    type: string
                    description: Additional information about the status of the task
    post:
      summary: Post the new Source unit into the task
      parameters:
        - in: path
          name: taskId
          required: true
          description: Identifier of the task to get status for
          schema:
            type: string
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                sourceUnit:
                  type: object
                  description: Source unit to be updated for the task
      responses:
```

```
146         "200":
147           description: File successfully uploaded
148           content:
149             application/json:
150               schema:
151                 type: object
152                 required:
153                   - taskId
154                 properties:
155                   taskId:
156                     type: string
157                     description: Identifier for the uploaded task
```

Bibliography

[1] Simon Curty, Felix Härer, and Hans-Georg Fill. «Blockchain Application Development Using Model-Driven Engineering and Low-Code Platforms: A Survey». In: *https://www.unifr.ch/inf/digits/en/* (Apr. 2022) (cit. on p. 12).

[2] *Dapbuilder*. URL: `https://dappbuilder.io` (cit. on p. 12).

[3] *Metamask*. URL: `https://metamask.io` (cit. on p. 13).

[4] *Daml Hub*. URL: `https://hub.daml.com` (cit. on p. 14).

[5] *Daml language documentation*. URL: `https://docs.daml.com` (cit. on p. 14).

[6] *Audit Wizard*. URL: `https://www.auditwizard.io` (cit. on p. 14).

[7] *Slither*. URL: `https://github.com/crytic/slither` (cit. on pp. 15, 20, 31, 32, 41, 45).

[8] *4naly3er*. URL: `https://github.com/Picodes/4naly3er` (cit. on p. 15).

[9] *Aderyn*. URL: `https://github.com/Cyfrin/aderyn` (cit. on p. 15).

[10] *Createweb3dapp.alchemy.com*. URL: `https://createweb3dapp.alchemy.com` (cit. on p. 16).

[11] *Alchemy*. URL: `https://www.alchemy.com` (cit. on p. 16).

[12] *Polygon*. URL: `https://polygon.technology/` (cit. on p. 16).

[13] *Optimism*. URL: `https://optimism.io/` (cit. on p. 16).

[14] *Arbitrum*. URL: `https://arbitrum.io/` (cit. on p. 16).

[15] *NftFy*. URL: `https://nftify.network` (cit. on p. 16).

[16] Luca Guida and Florian Daniel. «Supporting Reuse of Smart Contracts through Service Orientation and Assisted Development». In: *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*. 2019, pp. 59–68. DOI: `10.1109/DAPPCON.2019.00017` (cit. on pp. 16, 17).

[17] Luca Guida and Florian Daniel. *Solidity Registry*. URL: `https://github.com/LucaGuida/SolidityRegistry` (cit. on p. 16).

[18] Luca Guida and Florian Daniel. *Solidity Editor*. URL: https://github.com/LucaGuida/SolidityEditor (cit. on p. 16).

[19] *Blockly*. URL: https://developers.google.com/blockly (cit. on p. 17).

[20] *ANTLR*. URL: https://www.antlr.org (cit. on pp. 18, 26, 27).

[21] *ChatGPT*. URL: https://chatgpt.com (cit. on pp. 19, 33–37).

[22] *gRPC*. URL: https://grpc.io (cit. on pp. 22–26, 28, 30–32, 36, 40–44).

[23] *OpenAPI*. URL: https://www.openapis.org/ (cit. on pp. 26, 44, 49).

[24] *solc*. URL: https://github.com/ethereum/solidity/releases (cit. on p. 30).

[25] *go-openai*. URL: https://github.com/sashabaranov/go-openai (cit. on p. 33).

[26] *function calling*. URL: https://platform.openai.com/docs/guides/function-calling (cit. on pp. 33, 34, 37, 38).

[27] *chopper*. URL: https://pub.dev/packages/chopper (cit. on p. 48).

[28] *flutter bloc*. URL: https://pub.dev/packages/flutter_bloc (cit. on pp. 48, 51).