Gottfried Wilhelm
Leibniz University Hanover
Faculty of Electrical Engineering
and Computer Science
Institute of Practical Computer Science
Department Software Engineering

Automatic tracing of security-critical requirements in software projects through the development of a plugin

Master Thesis

in course Computer Science

by

Jan-Marc Paßlack

Examiner: Prof. Dr. Kurt Schneider
Second Examiner: Dr. Jil Klünder
Supervisor: Alexander Specht

Hanover, 28.10.2024

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 28.10.2024

Jan-Marc Paßlack

Zusammenfassung

Die heutige Softwareentwicklung ist ein komplexes Gebiet, in welchem unabhängige Teams an verschiedenen Komponenten eines größeren Systems arbeiten. Eine korrekte Implementierung aller Anforderungen stellt eine signifikante Herausforderung für Entwickler dar, insbesondere im Kontext sicherheitskritischer Anforderungen. Um dieses Problem anzugehen, wurde in dieser Arbeit ein Plugin für Visual Studio Code (VS Code) entwickelt. Es verwendet ein Large Language Model (LLM), um halb-automatisches Tracing zwischen Anforderungen und der Implementierung im Code zu ermöglichen.

Das Plugin analysiert den Code, zerlegt ihn in Funktionen, paart jede Funktion mit den gegebenen Anforderungen und sendet es an das LLM, welches bewertet, wie gut jede Funktion die jeweilige Anforderung erfüllt. Ergebnisse werden mithilfe von Webtechnologien in der Seitenleiste von VS Code angezeigt. Die Benutzer können sehen, welche Funktionen mit einer Anforderung in Verbindung stehen, zusammen mit einer Bewertung und einem Kommentar des LLMs.

Ein Datensatz mit sicherheitskritischen Anforderungen und entsprechenden Implementierungen in verschiedenen Programmiersprachen wurde verwendet, um das LLM zu evaluieren. Einige Implementierungen wurden so variiert, dass sie unvollständige Funktionalitäten aufwiesen, Variablennamen zufällig Buchstaben enthielten und Kommentare entfernt wurden, um die Robustheit des LLMs zu testen. Die Ergebnisse zeigen, dass das LLaMA-Modell in Kombination mit einem bestimmten Paarungstyp von Anforderungen und Implementierungen sehr genaue Übereinstimmungen ergab. Außerdem hatten das Vorhandensein von Kommentaren und die Wahl der Programmiersprache keinen eindeutigen Einfluss auf die Ergebnisse.

Abstract

Automatic tracing of security-critical requirements in software projects through the development of a plugin

Todays software development is a complex landscape, in which independent teams work on different components of a larger system. The correct implementation of all requirements represents a significant challenge for developers, especially in the context of security-critical requirements. To address this issue, a plugin for Visual Studio Code (VS Code) was developed in this thesis. It integrates a Large Language Model (LLM) to enable semi-automatic tracing between requirements and code implementations.

The plugin analyses the code, breaks it down into functions, pairs each function with given requirements, and sends it to an LLM, which evaluates how well each function meets the corresponding requirement. Results are displayed in a custom sidebar using web technologies. Users can see which functions are related to a requirement, along with a score and comment provided by the LLM.

A dataset containing security-critcal requirements and corresponding implementations with different programming languages were used to evaluate the LLM. Some implementation sets were varied to have incomplete functionalities, randomised variable names and removed comments to test LLM robustness. The results show that the LLaMA model in combination with a certain pairing type of requirements and implementations produced very accurate matches. Furthermore, the presence of comments and the choice of programming language had no clear impact on the results.

Acronyms

AI Artificial Intelligence. 22

API Application Programming Interface. 9, 11, 14, 29, 34, 35, 36, 39, 40, 45, 46, 48, 52, 59, 60, 61

BDD Behavior-Driven Development. 18

CC Common Criteria. 15, 16

CSS Cascading Style Sheets. 10, 35, 39, 61

CSV Comma-separated values. 12, 25, 29

DOM Document Object Model. 10

FN False Negative. 14, 56, 58

FP False Positive. 14, 56, 58

GORE Goal-oriented Requirements Engineering. 16

GWDG Gesellschaft für wissenschaftliche Datenverarbeitung Göttingen. 48, 52

HeRa Heuristic Requirements Editor. 15

HTML Hypertext Markup Language. 10, 12, 35, 39, 61

IDE Integrated Development Environment. 21, 27, 36, 59, 60, 61

JS JavaScript. 10, 11, 39, 40, 61

JSON Java Script Object Notation. 12, 25, 29, 44, 46, 54

LLaMA Large Language Model Meta AI. v, vii, 2, 53, 58

LLM Large Language Model. v, vii, 2, 8, 9, 16, 17, 21, 22, 23, 24, 25, 26, 28, 30, 40, 45, 48, 51, 52, 53, 54, 56, 58, 59, 60, 61, 63, 64

LUH Leibniz Universität Hannover. 52

NLP Natural Language Processing. 7, 8, 24

NPM Node Package Manager. 11

TDD Test-Driven Development. 18

TN True Negative. 14, 56, 58

TP True Positive. 14, 56, 58

TS TypeScript. 11, 12, 29, 40

UI User Interface. 23, 34, 36, 40

UML Unified Modeling Language. 4, 16

UMLsec Unified Modeling Language Security Extension. 16

URI Uniform Resource Identifier. 13

URL Uniform Resource Locator. 13

URN Uniform Resource Name. 13

VS Code Visual Studio Code. v, vii, 2, 7, 13, 27, 36, 37, 38, 39, 40, 41, 44, 46, 47, 48, 52, 59, 60, 61

Contents

1 Introduction **1**
- 1.1 Problem . . . 1
- 1.2 Approach . . . 1
- 1.3 Results . . . 2
- 1.4 Structure . . . 2

2 Fundamentals **3**
- 2.1 Tracing . . . 3
- 2.2 Programming Languages . . . 6
  - 2.2.1 C# . . . 6
  - 2.2.2 Go . . . 6
  - 2.2.3 Python . . . 7
- 2.3 NLP . . . 7
- 2.4 LLM . . . 8
- 2.5 Web Technologies . . . 9
  - 2.5.1 HTML . . . 10
  - 2.5.2 CSS . . . 10
  - 2.5.3 JavaScript . . . 10
  - 2.5.4 Electron . . . 11
- 2.6 Formats . . . 12
  - 2.6.1 JSON . . . 12
  - 2.6.2 CSV . . . 12
  - 2.6.3 URI . . . 13
- 2.7 Tools . . . 13
  - 2.7.1 Git . . . 13
  - 2.7.2 Yeoman . . . 13
  - 2.7.3 Postman . . . 14
- 2.8 Evaluation Metrics . . . 14

3 Related Work **15**
- 3.1 SeqReq . . . 15
- 3.2 LLM for Goal Models . . . 16
- 3.3 CodeBERT . . . 16

3.4 LLMSecGuard . . . 17
3.5 Format of the Requirements . . . 18
3.5.1 Gherkin Format . . . 18
3.5.2 Following a Cookbook . . . 19

4 Concepts **21**
4.1 Brainstorming . . . 21
4.1.1 Ways of tracing . . . 21
4.1.2 Forms of Interfaces . . . 23
4.1.3 Alternative Visual Representations . . . 25
4.1.4 General Properties . . . 26
4.2 Development Enivronments . . . 27
4.2.1 Text Editor . . . 27
4.2.2 Sublime Text . . . 34
4.2.3 Visual Studio Code . . . 36
4.3 Final Concept . . . 40

5 Implementation **43**
5.1 Backend . . . 44
5.1.1 Extracting Functions . . . 44
5.1.2 Tracing . . . 44
5.1.3 LLM . . . 45
5.1.4 Manual . . . 46
5.2 Frontend . . . 46
5.3 Debugging . . . 48

6 Evaluation **51**
6.1 Procedure . . . 51
6.2 Prompt . . . 54
6.3 Data . . . 55
6.4 Results . . . 56

7 Discussion **59**

8 Conclusion **61**

9 Future Work **63**

A Appendix **65**

Chapter 1

Introduction

Cyber-attacks have become a major threat to the public and private sectors, causing the loss of sensitive data and billions of dollars in damages [1]. One reason for this is that in today's software development landscape, ensuring that all requirements are implemented correctly is a significant challenge for developers [2]. Large systems may contain many components, which may even be maintained by independent teams especially in agile development [3][4]. To prevent information from beeing lost or misinterpreted, tracing between software requirements and their corresponding implementations can be used. This is a essential step in ensuring that the software meets the required specifications [5][6]. As documentation is time-consuming, it is not the focus of the developers, but rather the implementation itself [7].

1.1 Problem

The presented situation shows that there is a need for some kind of automated tool to take the effort out of manually tracing requirements. Even more so when it comes to security-critical requirements. In today's environment, it is crucial to ensure the correct implementation of these requirements, such as encryption, access control and validation. Security vulnerabilities can be caused by mismatches between requirements and their implementations, where security-critical features are either overlooked or only partially implemented [2][8][6].

1.2 Approach

An automated tracing tool could not only reduce the manual effort required by developers, but also improve the accuracy and reliability of the results. The potential of this type of automation has already been recognised [9].

To address the mentioned problem, the development of a plugin will be presented in this thesis. To assist developers as automatically as possible, it will run in the background of the development environment and analyse the code without any user interaction. An interface should be intuitive and easy to use, minimising the time needed to add requirements and view analysis results. Developers may not use the tool if the effort required to use it is too high. The use of LLMs in software development is already widespread [10]. Therefore, the use of an LLM to identify the relationships between requirements and even complex code implementations is a particular focus.

1.3 Results

The plugin was developed for VS Code and semi-automatic tracing of requirements to the corresponding implementations within the code was achieved by using a buzzword list and an LLM. An evaluation was then carried out to analyse the accuracy of the LLM, taking into account the effect of the programming language, the completeness and complexity of the code and the way in which the pairs of requirements and implementations were created. The results showed that the LLM was able to achieve a high accuracy when matching the pairs under certain conditions. When the LLaMA model was used together with the pairing of all requirements and all implementations at once, the precision was *0.96*, the recall was *0.74* resulting in a f1-score of *0.84*. There was no clear difference visible between the f1-scores of commented and uncommented code or between different programming languages.

1.4 Structure

First, the necessary background information is provided in Chapter 2. This includes explanations of relevant concepts, such as the basics of tracing and the technology behind the plugin. Chapter 3 presents existing work in the area of tracing methods and LLMs. Then, in Chapter 4, several approaches to achieving automatic tracing and development environments are discussed. Moreover, the implementation of the plugin is described in Chapter 5. The evaluation of the LLM is then presented in Chapter 6 through conducting an field study to evaluate its accuracy. Furthermore, the limitations of the approaches are discussed in Chapter 7, as well as unresolved challenges and issues. The Chapter 8 summarises the results briefly. Finally, the Chapter 9 concludes with an outlook on future possibilities.

Chapter 2

Fundamentals

This chapter lays the foundation for the terminology and the methods that are necessary to understand the content of the thesis.

2.1 Tracing

The first step is to determine what is meant by the term requirements tracing. Over time, a number of definitions have been produced by developers from different perspectives. The following list shows four types of definitions [4].

- **Purpose-driven**: "...the ability to adhere to the business position, project scope and key requirements that have been signed off."
- **Solution-driven**: "...the ability of tracing from one entity to another based on given semantic relations"
- **Information-driven**: "....the ability to link between functions, data, requirements and any text in the statement of requirements that refers to them"
- **Direction-driven**: "..the ability to follow a specific item at input of a phase of the software lifecycle to a specific item at the output of that phase"

While the *purpose-driven* definition is more concerned with the business behind the project, the *solution-driven* definition focuses on the relationship between different parts of the system through their meaning. In addition, the *information-driven* definition is more about relating a variety of data sources to the requirements. The *direction-driven* definition emphasises tracking the progress of elements through the development process.

This variety of perspectives highlights the need for clear definitions of the terms associated with tracing. Within their work "Traceability Fundamentals" [11], Gotel et al. have provided comprehensive definitions of these terms. The ones that are used in this thesis are the following:

- **Trace Artifact**: "A *traceable* unit of data (e.g., a single requirement, a cluster of requirements, a Unified Modeling Language (UML) class, a UML class operation, a Java class or even a person). A *trace artifact* is one of the trace elements and is qualified as either a source artifact or as a target artifact when it participates in a *trace*. The size of the *traceable* unit of data defines the *granularity* of the related *trace*."
  - **Source Artifact**: "The *artifact* from which a *trace* originates."
  - **Target Artifact**: "The *artifact* at the destination of a *trace*."
- **Trace Link**: "A specified *association* between a pair of *artifacts*, one comprising the *source artifact* and one comprising the *target artifact*. The *trace link* is one of the *trace elements*. It may or may not be annotated to include information such as the *link type* and other semantic *attributes*. This definition of *trace link* implies that the link has a *primary trace link direction* for *tracing*. In practice, every *trace link* can be traversed in two directions (i.e., if A tests B then B is tested by A), so the *link* also has a *reverse trace link direction* for *tracing*. The *trace link* is effectively *bidirectional*. Where no concept of directionality is given or implied, it is referred to solely as an *association*."
  - **Primary trace link direction**: "When a *trace link* is traversed from its specified *source artifact* to its specified *target artifact*, it is being used in the primary direction as specified. Where *link semantics* are provided, they provide for a way to "read" the traversal (e.g., A implements B)."
  - **Reverse trace link direction**: "When a *trace link* is traversed from its specified *target artifact* to its specified *source artifact*, it is being used in the reverse direction to its specification. The *link semantics* may no longer be valid, so a change from active to passive voice (or vice-versa) is generally required (e.g., if A replaces B then B is replaced by A)."
  - **Bidirectional trace link**: "A term used to refer to the fact that a *trace link* can be used in both a *primary trace link direction* and a *reverse trace link direction*."

- **Trace** (Noun): "A specified triplet of *elements* comprising: a *source artifact*, a *target artifact* and a *trace link* associating the two *artifacts*. Where more than two *artifacts* are associated by a *trace link*, such as the aggregation of two *artifacts* linked to a third *artifact*, the aggregated *artifacts* are treated as a single *trace artifact*. The term applies, more generally, to both *traces* that are *atomic* in nature (i.e., singular) or *chained* in some way (i.e., plural)."
  - **Atomic trace**: "A *trace* (noun sense) comprising a single *source artifact*, a single *target artifact* and a single *trace link*."
  - **Chained trace**: "A *trace* (noun sense) comprising multiple *atomic traces* strung in sequence, such that a *target artifact* for one *atomic trace* becomes the *source artifact* for the next *atomic trace*."
- **Trace** (Verb): "The act of following a *trace link* from a *source artifact* to a *target artifact* (*primary trace link direction*) or vice-versa (*reverse trace link direction*)."
- **Traceability**: "The potential for *traces* to be established and used. *Traceability* (i.e., *trace* "ability") is thereby an *attribute* of an *artifact* or of a collection of *artifacts*. Where there is *traceability*, *tracing* can be undertaken and the specified *artifacts* should be *traceable*."
  - **Requirements traceability**: "Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of ongoing refinement and itemtion in any of these phases)"
- **Tracing**: "The activity of either *establishing* or *using traces*."
  - **Manual tracing**: "When *traceability* is established by the activities of a human *tracer*. This includes *traceability creation* and *maintenance* using the drag and drop methods that are commonly found in current *requirements management tools*."
  - **Automated tracing**: "When *traceability* is established via automated techniques, methods and tools. Currently, it is the decision as to among which *artifacts* to create and maintain *trace links* that is automated."
  - **Semi-automated tracing**: "When *traceability* is established via a combination of automated techniques, methods, tools and human activities. For example, automated techniques may suggest *candidate trace links* or *suspect trace links* and then the human *tracer* may be prompted to verify them."

2.2 Programming Languages

Each programming language has its own features and possibilities. They differ from access to low-level system resources to the availability of libraries for encryption and authentication.

2.2.1 C#

Developed by Microsoft, the C# programming language is used to implement their .NET framework [12]. A framework is a platform that provides a set of libraries and tools to simplify the software development. Combining object-oriented and component-oriented paradigms, C# allows developers to create self-contained packages of a given functionality. The language is type-safe and has a unified type system, meaning that even primary types such as *integers* are derived from a common root called *object*. In addition, the design of C# focuses on versioning, allowing seamless integration with ever evolving libraries and components. The virtual and override keywords allow for the management of method overloading and interface member declarations, thereby ensuring backward compatibility as software evolves. C# is a general-purpose programming language that runs on a wide range of platforms and offers both efficiency and simplicity, making it an good choice for handling even complex development tasks[1]. Although it offers low-level functionality, the standard code written in C# is considered "verifiably safe", meaning that .NET tools can be used to ensure its safety. However, developers have the option to use "unsafe" code when necessary to access advanced features such as direct pointers or manual memory allocation. This allows them to manage system resources more effectively while maintaining flexibility.

2.2.2 Go

Go, also known as Golang, is an open source programming language that was started in 2008 [13]. In addition to the open source community, contributors include employees of Google. One of the main design goals was to create a language that works at scale, making it known for its simplicity and efficiency in handling very large and complex systems. With an easy-to-parse object-oriented syntax, it is easy to read and write code in Go. You can use existing libraries to extend the functionality of the language, or write your own to share with the community. It provides a high-performance concurrency model by using its lightweight implementation of threads, called goroutines.

Go provides built-in security tools that can automatically scan code for vulnerabilities using a curated database, enabling developers to identify potential security issues at an early stage[2].

[1] https://learn.microsoft.com/en-us/dotnet/csharp/
[2] https://go.dev/

It also includes cryptography libraries, making it a suitable option for developing secure applications. Moreover extensions are provided for popular development environments such as VS Code, enabling developers to efficiently analyse and improve their code through automated analysis.

2.2.3 Python

A similar syntax is used by Python, developed by Guido van Rossum [14]. Python is a capable, high-level, object-oriented programming language that allows developers to create a wide range of applications. Like many modern languages, it is cross-platform and open source, with one of the largest communities contributing to its growth. Python has an extensive collection of libraries, many of which can be written in other languages such as C or C++, further enhancing its functionality and performance. Its simplicity and capabilities have made it one of the most popular languages in fields such as artificial intelligence, data science and web development.

2.3 NLP

Natural Language Processing (NLP) is the automatic processing and analysis of human language, helping machines to interpret and generate text [15]. To achieve this, NLP systems use machine learning models that are trained on large datasets of data [16], which is shown in Figure 2.1. Using a deep understanding of language, NLP systems are able to understand both the semantic and syntactic elements of text. At the syntactic level, NLP works only with the structure and basic rules of the language. This enables tasks such as machine translation, text summarisation and question answering. On the other hand, semantic analysis allows NLP systems to understand the deeper meaning of a text by extracting information that is not directly written. An example of this is shown in Figure 2.2, where the system interprets the meaning behind the words. The top sentence is a positive statement. In contrast, the syntax of the lower sentence is positive, while the semantic meaning is negative, which can be detected by some NLP systems. This is crucial for tasks such as sentiment analysis, where the system interprets the emotional tone or intent behind the words. NLP enables many of today's technologies, including search engines such as Google, where it helps fetching relevant results based on the user's query.

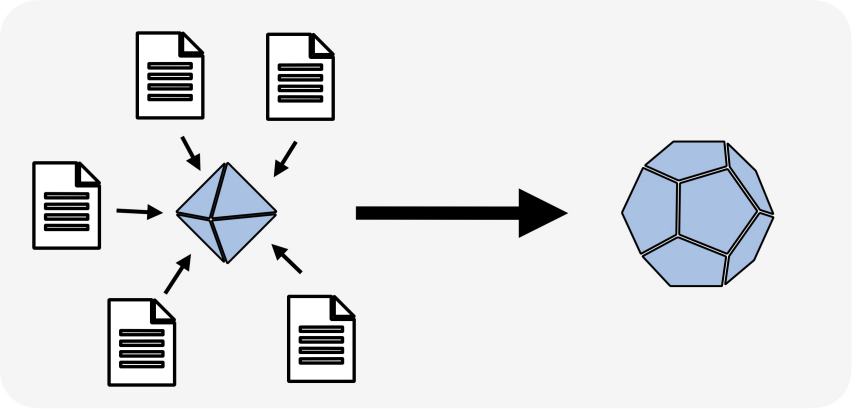

Figure 2.1: First step: NLP model learning process

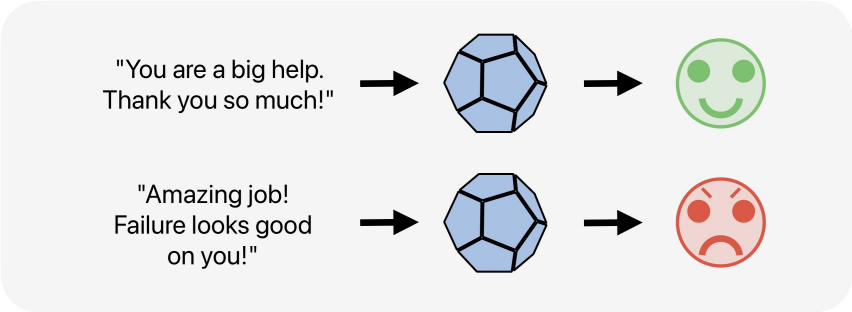

Figure 2.2: Second step: NLP model usage (sentiment analysis)

2.4 LLM

An LLM is a neural network that can transform input text data into output text data, making it ideal for NLP tasks such as translation [17]. The high performance of the LLM is achieved by scaling up the model size, training on larger datasets and using a greater amount of computing power. A larger model has more parameters, allowing it to process more complex data at the cost of more computing resources [18]. Chain-reasoning, problem solving, and following instructions are just some of the capabilities of an LLM.

First, you need to create a prompt that describes the task you want to perform [19]. The composing structure and the content of the prompt is called prompt engineering, and it has a big impact on the quality of the output. A prompt may contain a solved example for the given task, which is called a *1-shot* prompt. Correspondingly, a *0-shot* prompt is a prompt that contains no solved examples, and a *few-shot* prompt contains a few solved examples.

The Application Programming Interface (API) from OpenAI[3], the company behind the LLM ChatGPT, offers some strategies for getting better results. One is to write clear instructions, providing all the necessary details and context, and using delimiters to separate the instructions from the data contained in the prompt. An example would be: "Solve the calculation in brackets: (2*2)". It is also recommended that the model is given the steps to solve the problem. It is also important to provide the model with reference text to specify the output. For example, you can write a statement such as "Output the result in the following format: solution: <number>".

Going further, there are several properties of an LLM that can be adjusted to improve the output. One is the temperature, where a higher temperature results in more randomness, while a lower temperature results in more deterministic output. You can instruct the LLM in a system prompt to instruct it, while user prompts is contains a certain task.

Figure 2.3 shows an example of a conversation with an LLM, in this case ChatGPT-4o. The user prompt at the top requires logical reasoning, which the model is able to do by giving a correct answer at the bottom.

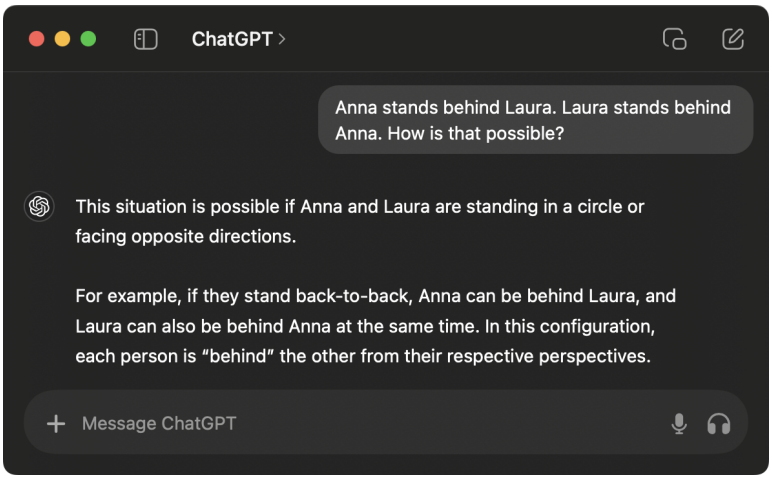

Figure 2.3: Reasoning capabilities of a LLM (ChatGPT-4o)

2.5 Web Technologies

Web technologies are the fundamental building blocks of modern web development. They enable the creation of websites and applications that are dynamic, visually appealing and easy to use. This section provides an overview of used web technologies and frameworks.

[3]https://platform.openai.com

2.5.1 HTML

The Hypertext Markup Language (HTML) is the most widely used standard for creating the structure and content of web pages [20]. It provides a framework for putting text, images, buttons, links and other elements into a structured document that can be displayed by web engines. HTML uses a system of tags and elements to define headings, paragraphs, lists and multimedia content, allowing developers to create hierarchies and layouts of their content.

DOM tree

One of the core functions of HTML is to define the Document Object Model (DOM) tree, which represents the objects in a tree-like structure that allows CSS and JS to manipulate it to create dynamic and interactive elements. Each element in the DOM tree is represented by a node, which has technical parent-child relationships.

2.5.2 CSS

Cascading Style Sheets (CSS) is a stylesheet language that allows developers to control the layout and appearance of their web applications [20]. While HTML provides the basic structure and content, CSS defines the visual style, such as colours, fonts, spacing and positioning. The CSS syntax is rule-based, with selectors targeting HTML elements and declarations specifying the properties to be applied. There are several types of selectors, including element, class and ID selectors, which are used to apply styles to specific elements or groups of elements. The cascading structure of CSS allows a hierarchy of styles to be applied, with more specific rules taking precedence over more general ones. For example, an ID selector will override a class selector, while a class selector will override an element selector. Styles can also be set to react to external conditions such as different screen sizes, orientations, or device types, which is a crucial aspect of responsive web design. In addition, CSS includes animations and transitions for adding dynamic visual effects to elements without the need for JS.

2.5.3 JavaScript

Alongside HTML and CSS, JavaScript (JS) is crucial for web development in order to manipulate the DOM tree in real time. JS is a flexible and widely used programming language that allows interactivity and dynamic behaviour. This means that without having to reload the entire page, you can respond to user input and update elements on the page. It is also event-driven, so it is possible to listen for clicks, keystrokes or mouse movements and then execute code.

TypeScript

A disadvantage of JS is that it is not type safe, which can lead to errors that are difficult to debug[4]. To solve this problem, TypeScript (TS) has been developed as a superset of JS, adding syntax for types. With TS you can define the types of elements such as variables, objects and function return values. This allows you to catch errors during development rather than at runtime, making the code easier to develop.

Node.js

Node.js is a high performance, free, open source, cross-platform JS runtime environment[5]. It is built on the same core engine as Google Chrome. Because Node.js lets you run JS code outside the browser, it makes it possible to use JS to develop software such as servers and scripts. Its event-driven, asynchronous architecture can handle many tasks simultaneously. This makes Node.js an good choice for developing scalable network applications that can effectively manage a high volume of concurrent connections.

Node Package Manager

Node Package Manager (NPM)[6], hosted by GitHub, is the world's largest software repository and serves as the default package manager for Node.js. It helps developers easily manage and share both private and public packages, making code reuse more efficient. NPM is designed to increase the productivity and security of JS development by providing a central platform for discovering, installing, and managing dependencies.

2.5.4 Electron

Aimed at building native cross-platform desktop applications using web technologies, Electron is an open source framework developed by GitHub[7]. Based on Chromium and Node.js, it uses the native system API to access the file system and display system notifications. This approach dramatically reduces development time by allowing the same code base to be used across all platforms. Installing and updating applications is also simplified by using the Electron framework. Well-known examples of applications built with Electron include 1Password, Discord, Signal and Dropbox.

[4]https://www.typescriptlang.org
[5]https://nodejs.org
[6]https://www.npmjs.com
[7]https://www.electronjs.org/

Angular

Within the Electron framework, another framework called Angular can be used to build web applications[8]. Maintained by Google, it provides a set of tools and libraries to create dynamic and interactive web applications from small to large scale. It is based on a single page application architecture, which means that the browser does not need to reload when the user navigates to another subpage. A big advantage of Angular is the two-way data binding, which allows to use the same data from the TS file to be used in the HTML file. This makes tasks such as form validation and data manipulation easier to implement.

2.6 Formats

In software development, the selection of appropriate standardised data formats is important to ensure efficient data exchange and consistency. This section outlines the formats used in this project's objects and evaluation database.

2.6.1 JSON

The Java Script Object Notation (JSON) format was developed with the goal to establish a lightweight syntax for data exchange format [21]. The main advantages of JSON are the simplicity of its structure and the interoperability between different programming languages that use it.

JSON Schema

To ensure consistency of the data, the JSON schema has been used [22]. This schema defines the structure of the JSON file and the data types of each element. Furthermore, the schema itself is also written in JSON format and can be used to validate the JSON file.

2.6.2 CSV

When it comes to tabular databases, the Comma-separated values (CSV) format has been a common choice for decades [23]. A CSV file consists of a series of lines, each line representing a single entry. The first line of a CSV file is often used as an optional header, containing the names of the columns to provide context for the data. Values within each entry are separated by a special character, usually a comma, which acts as a delimiter. Each row in the file must contain the same number of columns. Overall, this structure makes CSV files easy to read and write, and therefore quite popular.

[8] https://angular.dev

2.6.3 URI

Sometimes data is located in places that need to be explicitly known to other parties. A Uniform Resource Identifier (URI) can be used to achieve this [24]. This is a sequence of characters that identifies a resource, either abstract or physical. They consist of several components such as scheme, authority, path, query and fragment. They can also be classified as either or both of the Uniform Resource Name (URN) and Uniform Resource Locator (URL) types. URLs are well known for their use on the World Wide Web.

2.7 Tools

When developing software, it is important to have the right tools to support the development process. This section introduces the tools used in this project.

2.7.1 Git

Having a reliable place to store your code is essential, as it allows you to store and track changes. Git is a free and open source distributed version control system designed to efficiently handle everything from small to large projects[9]. Another key advantage over other systems is Git's branching model, which allows independent feature development by creating separate branches. This allows developers to work on new features, bug fixes or experiments without affecting the main code base. These broad features, combined with its relative ease of use, have contributed to Git's widespread adoption across the software development industry.

2.7.2 Yeoman

To simplify the process of starting new projects, Yeoman is a tool that provides a system for using code generators[10]. It allows developers to quickly create a new project from a predefined template, saving time and ensuring consistency by creating configuration files, folder structures and dependencies. There are a number of generators available for different types of frameworks, such as Angular and React.

Extension and Customization Generator

The Extension and Customization Generator[11] can be used to create a VS Code extension. Inside the terminal, the generator asks a series of questions to determine the type of extension to be created.

[9] https://git-scm.com
[10] https://yeoman.io
[11] https://www.npmjs.com/package/generator-code

It then automatically generates the folder structure and configuration files required for the extension.

2.7.3 Postman

When testing or developing APIs, Postman is a useful tool that can be used to send requests to a server and receive responses[12]. You can create and save different requests, organise them into collections, and even share them with your development team. Postman also provides automated testing, where you can write tests to validate the responses to your requests. It is a great tool for external debugging and testing of the API before using it in the application.

2.8 Evaluation Metrics

In an empirical evaluation, the terms *precision*, *recall* and f1-score are commonly used to measure the performance of a system. They range from 0 (worst) to 1 (best) and are calculated on the basis of True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN) values. A TP is a instance correctly identified as positive, while a FP is a instance that was incorrectly identified as positive. The values for TN and FN are defined similarly. The following formulas for the metrics are adapted from the paper [25]:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2.1}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{2.2}$$

$$\text{f1-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.3}$$

In the case of this thesis, the values for TP, TN, FP, and FN are calculated based on the following definitions:

- TP: The number of pairs between requirements and implementations that are related and are identified as related.
- TN: The number of pairs between requirements and implementations that are not related and are identified as not related.
- FP: The number of pairs between requirements and implementations that are not related but are identified as related.
- FN: The number of pairs between requirements and implementations that are related but are identified as not related.

[12] https://www.postman.com

Chapter 3

Related Work

While this thesis focuses on the traceability of security requirements to source code, there are already some approaches and tools that address the tracing of security requirements during other stages of development. Furthermore, since the construction of requirements is a critical part of the development process, the way requirements can be formulated is also discussed in this chapter.

3.1 SeqReq

In a rapidly changing technology landscape, software developers often lack sufficient security expertise [8]. As software systems become more complex, it is easier to overlook security issues. The problem starts with the fact that security requirements are not always clear or well explained by stakeholders, who are often unaware of potential threats. This makes it difficult to understand and document these requirements. The paper [8] presents SeqReq, which helps developers to trace security requirements throughout the system design process. The tool combines three techniques to ensure that security is considered from the start.

1. The Heuristic Requirements Editor (HeRa) helps to dentify security issues early by analysing requirements and descriptions. The software has an editor that helps you find potential security risks using keywords and patterns.

2. Common Criteria (CC) provides a framework for improving security requirements. The framework provides a set of predefined security classes, such as identification and encryption, to help develop secure components.

However, CC can be difficult at first for those without the right knowledge, as it uses a domain-specific language and requires an understanding of security concepts. To help people without in-depth security expertise understand, CC uses a step-by-step method to guide developers to meet the security requirements.

3. The Unified Modeling Language Security Extension (UMLsec) provides a framework for tracking security requirements from the beginning to the end of the design process. Therefore, these requirements are represented in UML design models. This allows developers to check that the design reflects the requirements and that no important security features are overlooked.

3.2 LLM for Goal Models

In their paper, Hassine focuses on establishing trace-links between goal models and specific requirements [26]. The rise of LLMs brings great potential, revolutionising automated traceability by solving previous challenges and opening up new possibilities. Goal-oriented requirements provide a well-defined structure with precise syntax and semantics, allowing the formal documentation of desired goals and their dependencies. This approach is a cornerstone of Goal-oriented Requirements Engineering (GORE), which provides a systematic framework for creating, analysing and managing requirements to meet stakeholder objectives. A key component of GORE is the focus on non-functional requirements, such as security. The LLM was asked to link requirements to objectives, while providing the rationale for its decision. The evaluation tested a set of 42 requirements covering both security and non-security aspects. The results were very promising, demonstrating the capabilities of the LLM approach.

3.3 CodeBERT

To reduce the time developers spend searching through large code bases and writing documentation, CodeBERT can help with its ability to link natural language and programming languages [27]. It was the first large-scale, multilingual, pre-trained model to capture the semantic links between natural language descriptions and code. One of its applications is in generation tasks, such as the creation of code documentation. By analysing code, CodeBERT can generate descriptive documentation that explains functionality in natural language. Another important capability of CodeBERT is its ability to perform tasks that involve understanding code. For example, in natural language code search, CodeBERT can retrieve relevant code snippets based on semantic similarity to a search query.

3.4 LLMSecGuard

For security requirements to be met, they must be correctly implemented in the source code. The use of LLMs to generate code is widely spread in software development [10]. However, their output code may contain errors, which is particularly problematic in the security domain. In addition, some developers lack the expertise to identify bugs and refine the input to improve the generated code. This is where the open source framework *LLMSecGuard* can assist, a tool that benchmarks the security properties of LLMs and improves the security of the generated code. In the benchmarking process, an LLM is given existing prompts with expected outcomes. The rating depends on the accuracy of achieving the expected outcomes. Moreover, the workflow of the second functionality of *LLMSecGuard*, the generation of secure code, is shown in Figure 3.1. Starting with a user prompt, the prompt agent passes it to the LLM which generates the code. The prompt agent then sends the code to the security agent, which passes it to an external code analysis engine. If vulnerabilities are found, the prompt agent is instructed to reformulate the prompt to address them. This process is repeated until no vulnerabilities are found or the maximum number of iterations is reached. At the end, the improved code is returned with an analysis of its security properties.

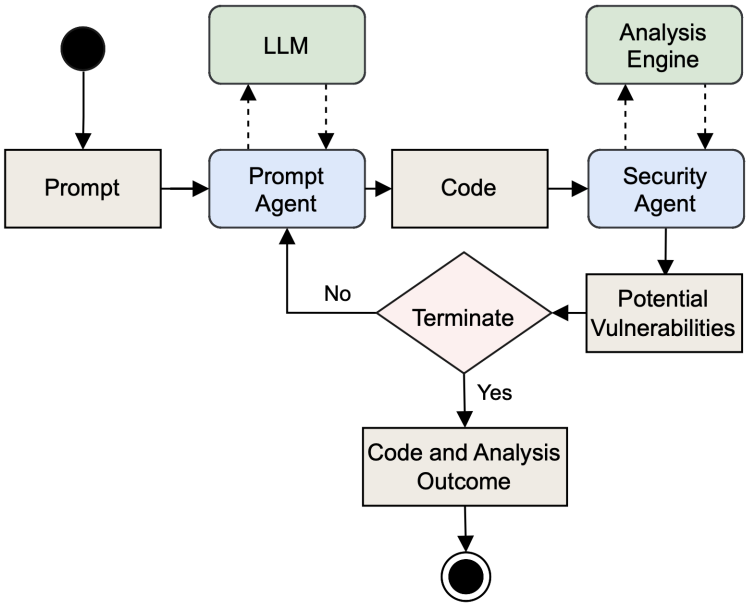

Figure 3.1: LLMSecGuard's secure code generation workflow [10]

3.5 Format of the Requirements

The way requirements are constructed is critical to achieving complete and understandable requirements. Incomplete, unclear and constantly changing requirements are a major cause of defects in software projects [2][28]. There is a lot of effort required to produce good requirements. As a result, particularly agile developers may be tempted to skip writing requirements altogether [29]. The reviewed literature is divided on the best way to write requirements, which will be the input for the plugin. On the one hand, César dos Santos et Vilain claim that natural language formulations can be confusing and suggest using a precise and structured way like the following Gherkin format [2]. On the other hand, Shostack argues that while a structured format may be appropriate as a guideline for writing general requirements, it may not be the best choice for security requirements [29]. The reason for this is that security would rarely be the main focus of a specific feature.

3.5.1 Gherkin Format

In Test-Driven Development (TDD), an existing test is used as the basis for the implementation of a new feature. Behavior-Driven Development (BDD) is achieved by incorporating acceptance testing into TDD. The IEEE defines acceptance testing as *formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system* [30]. Consequently, requirements are created by writing the behaviour and expected outcome of the system. The Gherkin language follows this approach by writing requirements in a structured way, while keeping the effort relatively low [2].

It consists of three main parts: the feature, the scenario and the steps[1]. Most lines start with a predefined keyword, which is crucial for the structure. After the keyword, the text can be written in natural language with few restrictions. The first line with the keyword *Feature* should contain an abstract description of the feature. This can be followed by one or more scenarios with the keyword *Scenario*. Each scenario should include a description of the scenario followed by a list of steps. A step starts with one of the keywords *Given*, *When*, *Then*, *And* and *But*. These are used to express the context, the user's actions and the expected outcome of the scenario. Apart from these main parts, there are many more keywords and possibilities to structure the requirements in Gherkin. For example, a *Rule* is used to group together scenarios that follow the rule. In addition, *Background* can be used to define steps that are executed before the scenarios of one feature to reduce redundancy with writing the same information in the *Given* steps of each scenario.

[1] https://cucumber.io/docs/gherkin/reference

Some secondary keywords such as “|“, “@“ and “#“ are used to create tables, tags and comments, which helps with the readability and organisation of the requirements. An example of a Gherkin format can be seen in Figure 3.2. It covers the conversion of a text file to PDF and DOCX formats.

```
Feature: File Conversion

   Background:
      Given a file named "document.txt" in the source directory

   Scenario: Convert a text file to PDF
      When I convert the file to "PDF" format
      Then a file should be created in the destination directory
      And be named "document.pdf"

   Scenario: Convert a text file to DOCX
      When I convert the file to "DOCX" format
      Then a file should be created in the destination directory
      And be named "document.docx"
```

Figure 3.2: A requirement in Gherkin format

3.5.2 Following a Cookbook

As the name suggests, this approach follows the concept of a cookbook, where you can look up the recipe for a particular dish. You can use it as a base and tweak it to your liking until you are happy with the result. In the context of security requirements, you can take a “straw-man“ requirement and adapt it to your specific needs. For example, in Figure 3.3 you can see a security requirement for a login feature, which starts broadly and is specified further in the following steps. This approach is still based on natural language and is even easier to write, because you don’t have to follow a strict structure like in Gherkin, and you can evolve the requirement as you need to.

```
1. The user can login into an account
2. The user can login into his account with his credentials
3. The user can login into his account with his username
4. The user can securely login into his account with his username and password
5. The user can securely login into his account with his username and password and two-
factor authentication
```

Figure 3.3: Example of the usage of a cookbook for writing security requirements

Chapter 4

Concepts

While the related work focused on creating requirements traceability between security requirements to artefacts such as design models and goal models, this thesis focuses on establishing traces between the source artefacts *security requirements* to the target artefacts *code implementations*. In this chapter, the concepts that form the basis of the plugin are outlined. First, a brainstorming presents different methods of tracing requirements to their implementations, involving a list of buzzwords and the use of an LLM. Afterwards, some forms of interfaces with different focuses are discussed. The development environment sets the circumstances for the plugin and determines the way it is implemented. Therefore, environments ranging from a simple text editor and to an advanced Integrated Development Environment (IDE) will be compared. At the end, the final concept of the plugin will be described.

4.1 Brainstorming

At the start of the project, several concepts were developed about how the plugin could be implemented and what features it could have. These features were developed without focusing on the concern of the feasibility of the implementation.

4.1.1 Ways of tracing

Two possible approaches for identifying pairs of requirements and implementations in the code are to use a list of buzzwords or to use an LLM.

Buzzword List

A list of buzzwords contains common terms within the requirements and the implementations. It can be specified whether the implementation term should be contained in a function name, a variable or a comment. The system then checks whether there is a word in the requirement that matches a term in the buzzword list, and then searches for the corresponding pair of terms in the code. This method allows the system to identify pairs of requirements and implementations. To ensure that all relevant buzzwords are included, the list may need to be updated regularly. The effort required to maintain a list containing multiple programming languages can be considerable. An example of the process of the buzzword list is illustrated in Figure 4.1. The term “Validate“ for the requirement is matched with term “validate“ for the implementation.

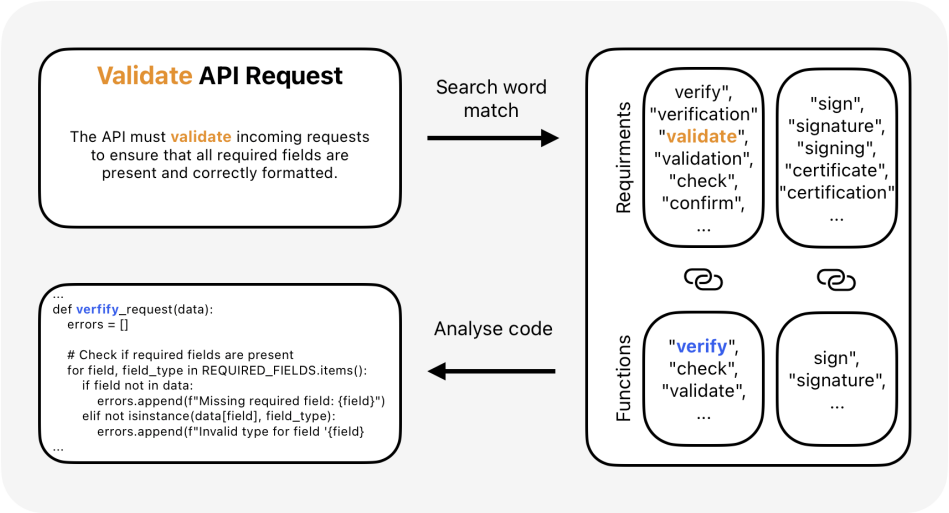

Figure 4.1: Process of the buzzword list concept

LLM

Another concept is to use the capabilities of Artificial Intelligence (AI) with the use of an LLM to identify the pairs. First, the LLM is given a *system prompt*, which contains the instructions on what to do. Afterwards, the *user prompt* contains the requirement and the code. It is to be determined if the structure of the prompts, requirements and code has any effect on the result. As output, the LLM would give feedback on how well the requirement and code match. This could be categories such as “perfect match“, “good match“, “possible match“ or “no match“ or a score from 0 to 10.

For better understanding, a comment should be provided by the LLM to explain why the match is rated as it is. Ideally, even code structures that contain no comments or are complex should be understood by the LLM. There could be problems when the code imports existent packages or external functions as they are not part of the code itself. Also the use of multiple programming languages may be challenging. All this could affect the quality of the results. Furthermore, implementations that are incomplete, incorrect, or just seem to be similar to the requirements but are actually not related to them could be a difficulty for the LLM. The concept is shown in Figure 4.2. Both the requirement and code are given to the LLM, which then outputs if they are related. In this case the requirement "Validate API Request" matches the code implementation well, resulting in a positiv rating.

A custom neural network could be trained and fine-tuned on a large collection of pairs between security requirements and their implementations. Although a great amount of effort is required to train the network, the results could be more accurate and reliable than the results of the LLM.

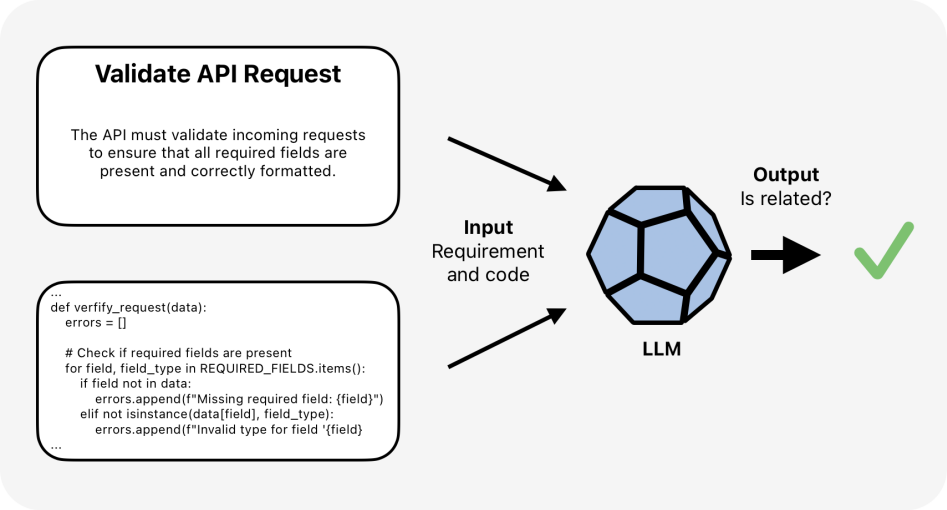

Figure 4.2: Process of the LLM concept

4.1.2 Forms of Interfaces

Besides integrating semi-automatic tracing, the plugin should have an intuitive User Interface (UI). Two different ideas for the interface were considered, ranging from a search system to a data manager.

Search System

The first idea for the interface was to create a project-wide search system where the user could enter any number of keywords. Then the system should be able to find the exact or similar matches in the code base. Perhaps the keywords could be separated by a delimiter to search for multiple keywords at once, where one or all of them must be present in the results. A natural language phrase could also be used as input and the system would search for the code that matches it. For this to work, the system may need to use NLP to transform and understand the query.

The search should be available at all times, perhaps via a shortcut or a search bar in the interface. The results should be displayed as a list in descending order of relevance, which is determined by how well the code matches the query. When you click on a result, the system should jump to the corresponding code and highlight the matching parts. The search query should be cached, so that the user can go back to previous search results if necessary. For frequently used queries, the user should be able to save them as favourites. In addition, some sort of search history would make it easier for the user to go back to previous searches. To make searching more convenient, search results could be filtered by aspects such as the programming language.

Data Manager

Another idea for the interface was to create a manager where the user could enter the requirements, which would all be stored in a database, rather than having to enter a search query everytime. This is similar to the concept of a book, where the code is the content and the plugin is like the table of contents to quickly find the right chapter. All requirements should be presented in a list, where the user can see the current status of each requirement with an appropriate colour. This status could indicate whether implementations have been found or whether the implementation of the requirement is complete. Like in the search system, some functionality to filter, sort and search the requirements could be integrated. When entering a new requirement, there should be two input fields for a title and a description. There could also be an overall progress bar at the top so that the user can quickly see how many requirements have been implemented.

Clicking on a requirement should display all the information about it. In addition to the data already mentioned, the implementations found should be listed. For each implementation, details such as the file name and line number should be displayed, which can be clicked on to jump to the corresponding code. In the case of using an LLM for tracing, the quality of the match with a short explanation should be displayed. When a possible implemenation is found by the system, the developer should be able to accept or reject it.

If an implementation matches several requirements, there could be a link to the other requirements to enable a *reverse trace link direction*. In the background, the system should be able to scan the code base for new implementations and update the status of the requirements accordingly. This scan could be triggered manually or automatically when the code changes. After the scan, the user should be notified of any implementations found and be able to review them. When the user starts writing code, they could tell the plugin that they are working on implementing a specific requirement.

In addition, if the automatic tracing system does not find an implementation, the user should be able to add it to a requirement manually. The user should be able to edit and delete requirements if necessary. Changing the requirement would have a direct impact on the previously found relationships to implementations in the code. Even a single word change could result in a different implementation being required. Therefore, all implementations should be flagged for review for the developer, who can confirm or deny the existing trace-link. The user should be able to see the history of the requirement, so that they can see the content of the requirement before it was changed, with a brief comment why it was changed. Another reason why a requirement might be marked as changed is if the implementation has been changed. Therefore, the system should be able to track the changes in the code and update the status of the requirement accordingly. Within the editor, the implementations could be marked with colours or tags to indicate the trace-link to a requirement.

For time saving purpose, it should be possible to import and export to commonly used formats such as CSV or JSON. This allows the user to build on existing requirements management software solutions without the need for redundant manual data entry. To further use the capabilities of the LLM, the user could be able to ask the system for assistance in implementing the specific requirement. This could range from a comment on what is missing to a complete part of code implementing the functionality. Besides being stored locally, the data could be synchronised with an internal database to allow multiple developers to collaborate. This would reduce the risk of data loss and the effort of data entry could be shared among team members. Concluding, this concept could be used to assist during the development of software, or even to check that all the requirements of an existing software project have been implemented.

4.1.3 Alternative Visual Representations

Besides displaying the search results or the data in the database as a list, there are other ways of presenting the data. Since the relationships between requirements and implementations need to be shown, a graph or a flow might be a good alternative. Mockups will be provided later in Figure 4.7.

Connected Graph

A connected graph is a collection of nodes and edges, where the nodes could represent the requirements and implementations and the edges the *bidirectional trace links* between them. The thickness of the edge could indicate how confident the LLM is with the found trace-link. Using the data manager as the underlying system, the user could search for a keyword. The path and nodes of the corresponding requirements and implementations could be highlighted, while completely unrelated nodes could be hidden. By clicking on a node, the user could see all the information about the requirement or implementation and jump to the corresponding code. Colours could be used to indicate the current status of the requirement or implementation. The result of many data points would be *chained traces* of nodes and edges, allowing the user to see the big picture of the project.

Flow

Focusing more on the concrete relationships between individual requirements and their implementations, a flow might be another appropriate representation. Starting with the requirements on the left, the associated implementations are displayed on the right. The flow could be divided into sections, with each section representing a requirement with its implementations. Similar features, such as the thickness of the edges or the display of information when clicking on a node, could be implemented as in the graph representation.

4.1.4 General Properties

Regardless of the type of tracing and interface, the plugin should contain some properties that are important for the user experience. Since the plugin tries to reduce the effort of manual tracing, it should be intuitive and easy to install and use. The data should be presented in a clear and structured way so that the user can quickly find the information they are looking for. In addition, the time needed to analyse the code should be as short as possible and never directly affect the developer's work. Data should be stored persistently so that the user does not lose the information when the plugin is closed. The plugin should run on all widely used operating systems to reach as many developers as possible. It should also work with the most commonly used programming languages. To meet the needs of different types of developers, the plugin should be customisable, for example by integrating keyboard shortcuts.

4.2 Development Enivronments

The choice of the development environment is a crucial step for this plugin, as it will determine the circumstances under which the plugin is developed. In the following section, several development environments will be presented, ranging from text editors to more advanced IDEs including Sublime Text and VS Code. An IDE is a software application that facilitates the development of software by providing a range of additional tools and features.

4.2.1 Text Editor

Starting with one of the most basic environments, a text editor is an application that allows the user to view, write and delete text. Each operating system comes with a pre-installed text editor, such as *Notepad* on Windows or *TextEdit* on MacOS. Both are shown in the Figure 4.3. They have a simple interface with a text area and a few buttons for basic functions such as saving or opening a file, but there is no native support for plugin integration. As a result, the plugin must be developed as a standalone application that runs parallel to the text editor. This provides the most freedom in developing the plugin, but may also require more effort to implement.

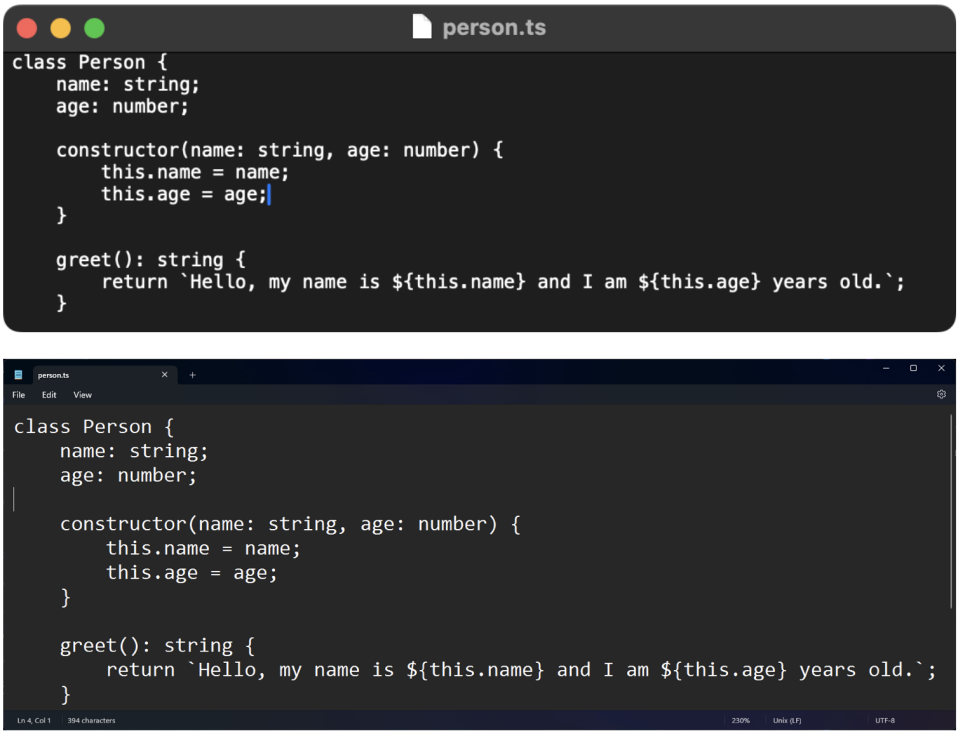

Figure 4.3: Interfaces of TextEdit (top) and Notepad (bottom)

External Application

Creating a standalone application that is cross-platform compatible can be achieved by using the Electron framework. This simplifies the development process by giving you access to the file system and allowing you to use the native user interface elements of the operating system, such as displaying notifications. Furthermore, Electron itself allows the integration of web frameworks such as Angular to build the user interface. To stay within the scope of the project, the root directory of the codebase needs to be defined by the user. Therefore, a page containing some settings should be implemented where the user can select the root directory. This page should also contain other customisation options, such as shortcuts or the appearance of the plugin.

Following the concept of a *search system*, submitting the search query could trigger a scan of the code withing the project. File after file would be searched for the keywords, and the results would be displayed in a list of rectangular card elements, perhaps in a grid view to fit more cards on the screen. Each entry would contain the file name and line number of the match. For results suggested by the LLM, the explanation of why the implementations is related to the requirement should also be displayed. In addition, the certainty of the match could be indicated by colouring the background of the card. The user could then click on a card to open the corresponding file in a new text editor. In case the search is entered frequently, a button should be displayed next to the input field to save the search as a favourite. Another element next to the input field would be the filter button, which opens a drop-down menu with options to filter the results by a specific programming language or minimum confidence level. A sidebar could display the history and favourites of searches, allowing the user to quickly return to previous searches. On the other hand, the *data manager* concept would be more complex to implement. The setting would be similar to the search system, but the rest of the interface would be quite different. An essential part is the functionality to add a new requirement. Clicking on the *Add* button somewhere in the corner of the interface would bring up a new card. This card would contain input fields for the title and description of the requirement. There should also be a file input field for importing requirements from an existing database file. After some requirements have been added, they should be displayed again as cards in a list.

By clicking on a card, the user should see all the information about the requirement in a new view. In addition to the title and description, the status of the requirement should be displayed, with different icons and colours indicating the degree of completion. The found implementations should be displayed in a similar way to the previous search system concept.

Similar to the process of adding a requirement, the user should be able to manually add an implementation with the file path and line numbers. If necessary, the user should be able to edit the requirement by clicking on an edit button. This would open the same view as when adding a new requirement, but with the existing data already filled in. In addition to the title and description, there should be a field for a comment explaining why the change is needed. The user can also accept or reject the implementation by clicking on the appropriate buttons. Accepting a proposed implementation should change the colour of the card to green, while rejecting it should remove it from the list. The information about rejected implementations should be stored in the database so that it does not reappear after the next scan.

Speaking of the database, there are a few things to consider. A simple format like CSV or JSON could be used to store the data, but using a real database management system has some advantages. For example, the requirements and implementations are linked, which could be represented by a relation within the database. It is also easier to modify specific data entries such a system, rather than having to modify the whole file with CSV or JSON. The database would run as a separate process on the computer, using Node.js to create an API. This API could be called by the TS code to send request and receive responses from the database. The database could be synchronised with another database on a server for collaboration with other developers.

The *VisJS* library[1] could be used to visually represent the data as a graph or flow. This library provides a wide range of options for creating a graph, such as the colour of the nodes or the thickness of the edges. The flow could be represented as a tree using the more hierarchical style. As this view would be more complex and take up more space, it should overlay the whole interface or be displayed in a new window. Some features should be implemented to allow the user to interact with the graph or flow, such as clicking on a node to see more information.

When the plugin is finally developed with all its features, publishing the plugin is simplified by the Electron framework. Compatible tools such as *Electron Forge* allow you to create a single executable for all operating systems at once, without having to worry about the individual implementation of the installation process.

[1] https://visjs.org

In Figure 4.4, a mockup of the plugin interface is shown. Here the search system is implemented with LLM as the tracing method. The user has entered *Message encryption* as a search query and all results are displayed as cards in a list view. Each card contains the file name and line number of the match, as in the first case *cryptography.py* and *line 122*. While the first two cards are marked with a green background, the third card is marked with a yellow background and the fourth card is marked with a red background. This indicates the certainty of the match, where green means a very good match, yellow means a possible match and red means an unlikely match. To the left of the input field is a filter icon and to the right is a favourite icon. At the bottom of the interface, the user can press the settings icon to open the settings view or the graph icon to open the graph view. On the left side of the interface, recent searches such as *Authentication Algorithm* and *Data Validation* are displayed as a history. Below the history, the user can see the favourites, in this case *Data Encryption*. In the top right corner of the sidebar there is an icon to toggle the display of the sidebar.

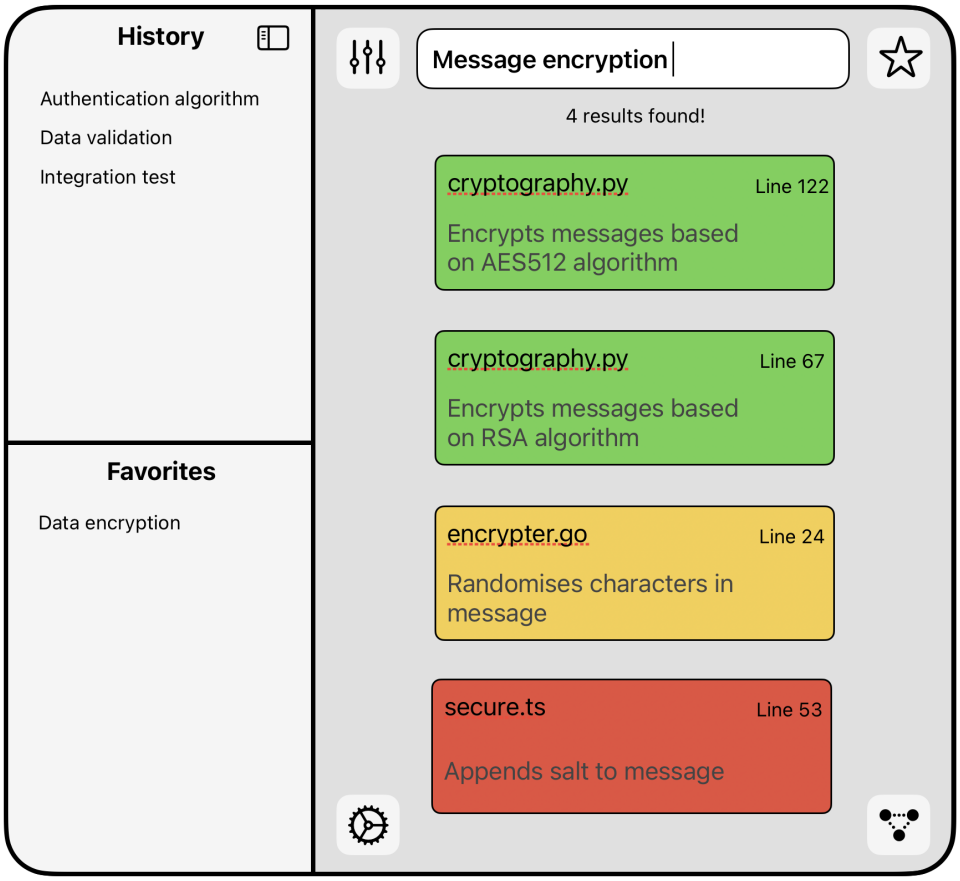

Figure 4.4: Mockup of the search system interface

The data manager displays all available requirements in its overview, as shown in the mockup in Figure 4.5. The interface buttons for opening the settings, activating filters and opening the graph view are placed in a similar way to the search system. In the top right corner there is a button for adding a new requirement. In the centre of the interface, the user can see all the requirements as cards in a grid view. Each card contains the requirement's title, description and status. The *Message Encryption* and *SQL Injection Prevention* cards have a status with a checkmark and a green background, indicating that the requirement is already fully implemented. Otherwise, the two cards *Password Complexity Enforcement* and *API Rate Limiting* have a status with a hammer symbol and an orange background, which means that the implementation is partially done. Finally, the *Session Timeout* and *Failed Login Lockout* cards have a circle status with a blue background, indicating that no implementation has yet been found.

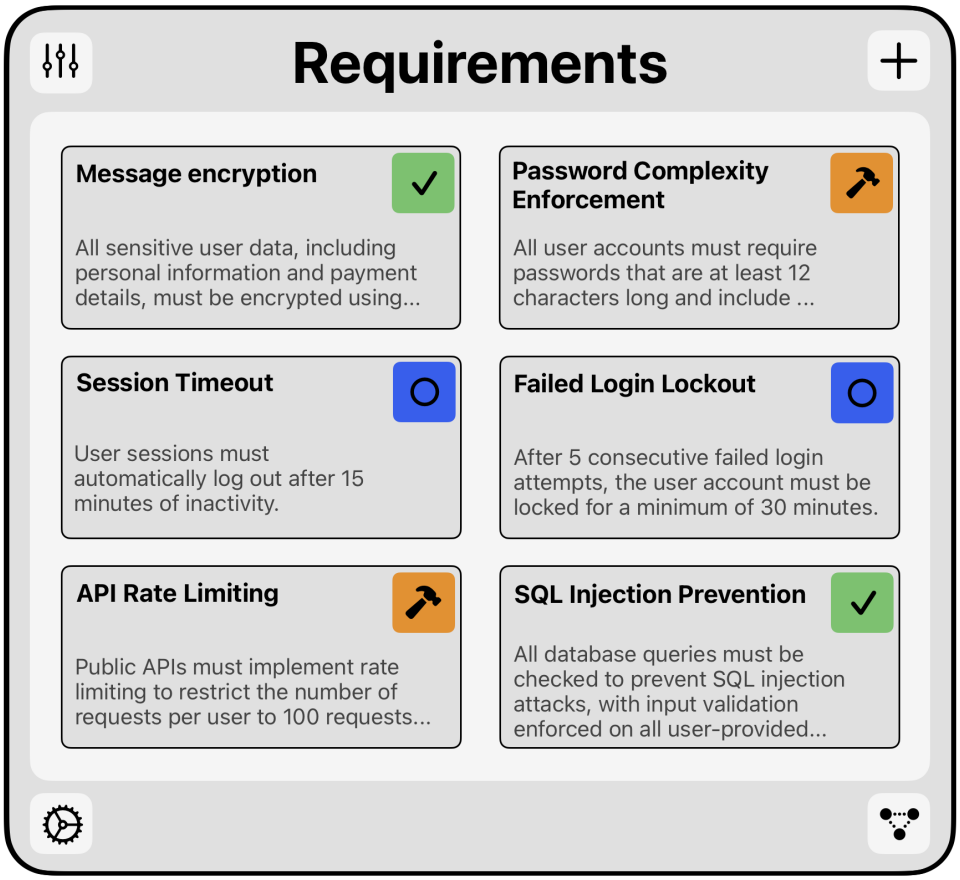

Figure 4.5: Mockup of the data manager interface - requirements overview

The mockup of the detail view of the Data Manager is shown in Figure 4.6. In this case, the user has clicked on the *API Rate Limiting* card within the overview of Figure 4.5. As in the overview, the requirement's title, description and status are displayed. The user can click buttons at the top to return to the overview, delete or edit the requirement and manually add a new implementation. All implementations are listed below in a grid view, similar to the search system mockup. At the bottom is the history of the requirement, where the user can see how the requirement has changed over time. In this case, the difference between the current version and the previous one is that the description has been made more detailed by changing the rate limit from *500* to *100*.

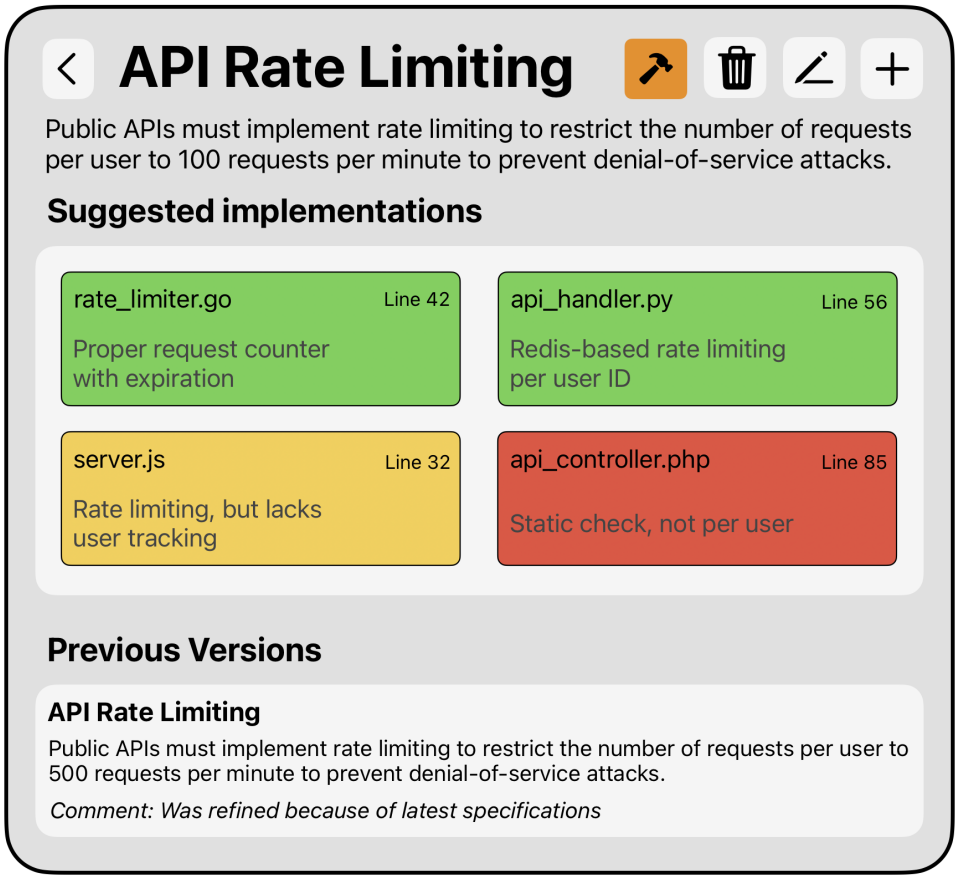

Figure 4.6: Mockup of the data manager interface - detail view

When the user clicks on the graph icon in the bottom right corner of the search system or data manager, a graph view is opened. This is shown as a mockup in Figure 4.7. The graph view displays the trace-links between requirements and implementations as a graph. Large nodes represent requirements such as *API Rate Limiting*, *Session Timeout* and *SQL Injection Prevention*. Moreover, small nodes represent the implementations such as *rate_limiter.go* and *server.js*. Suggested implementations are dotted edges, while confirmed implementations are solid edges. The thicker edge from *API Rate Limiting* to *rate_limiter.go* compared to *server.js* indicates that the first trace-link was created with a higher certainty. Files that can implement multiple requirements, such as *server.js*, can be easily identified by having multiple edges. The bottom half shows the flow view, which is a more hierarchical representation of the traces. This shows the trace-link between the *message encryption* requirement and its confirmed implementations, such as *main_hasher.py*.

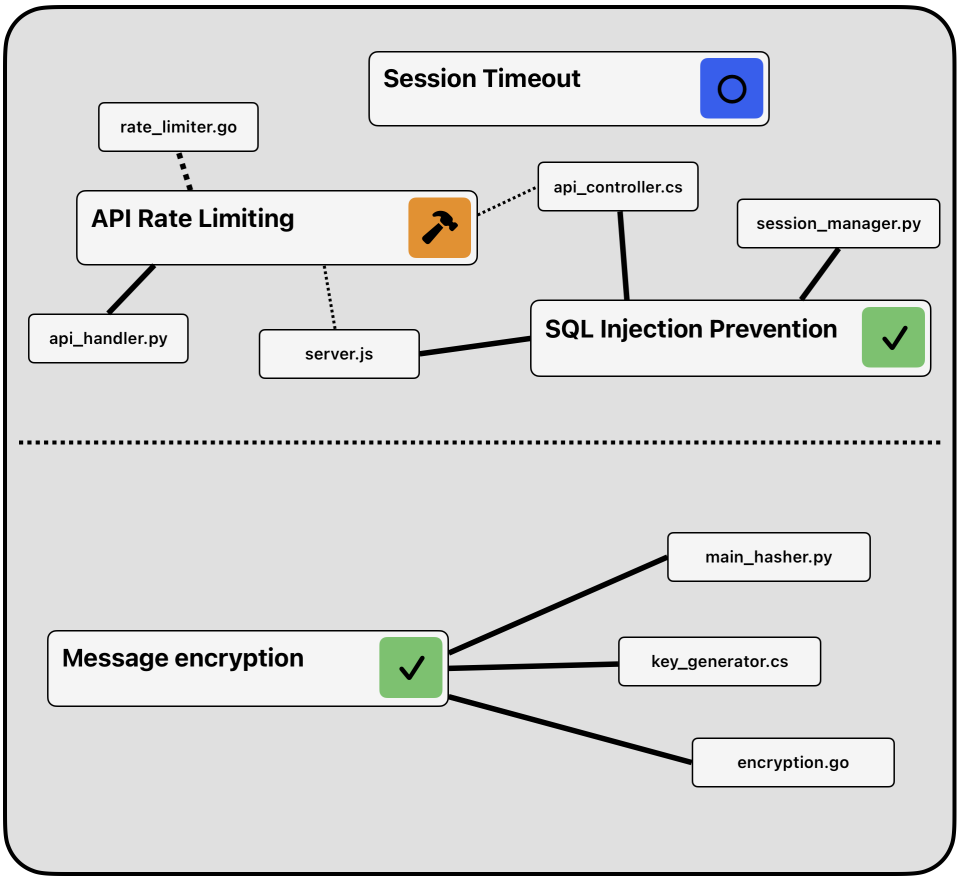

Figure 4.7: Mockup of the graph (top) and flow (bottom)

4.2.2 Sublime Text

The previous development environment doesn't have an API that can be used by a plugin. Therefore, the features of the plugin have to be implemented using external tools, which limits and complicates the development process. This changes if the plugin can communicate directly with the development environment using some sort of API. Sublime Text[2] is a text editor that provides this functionality as well as other more advanced features.

User Interface

The UI of Sublime Text is shown in the Figure 4.8. It is divided into several parts, starting on the left with the sidebar (1). This contains the File Explorer, which displays the files and folders of the current project in a tree structure. The central editor area (2) contains the contents of the files. The two files are opened side by side in a split view to make it easier to access the information in both files at once. At the top of the editor area (3) are tabs for each open file displayed, allowing you to quickly switch between recently used files. The status bar at the bottom of the UI (4) shows information about the current file, such as the programming language used or the line and column number where the cursor is located. Finally, a minimap to the right of an editor area (5) shows a preview of the whole file, making it easier to navigate through the file. All these elements can be customised in the preferences.

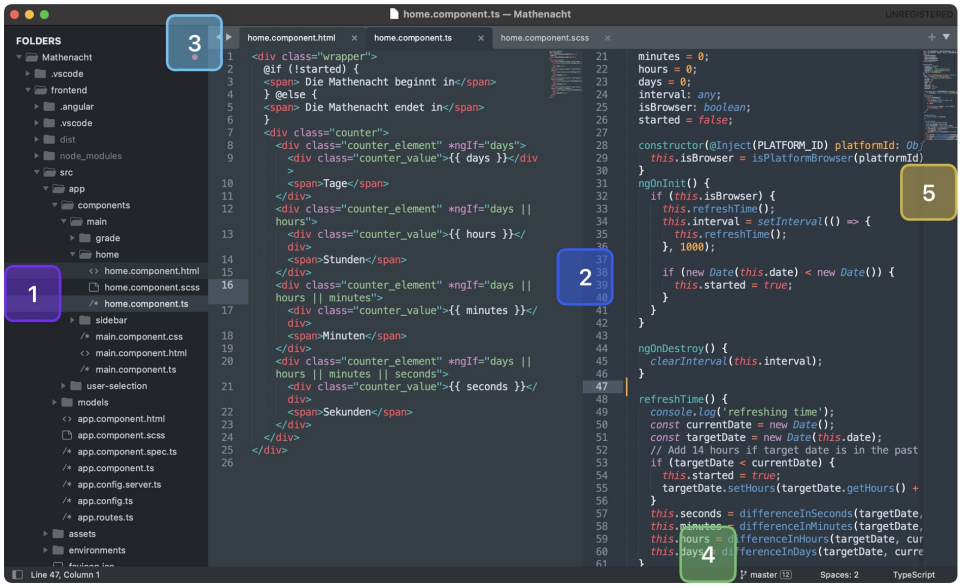

Figure 4.8: Interface of Sublime Text

[2]https://www.sublimetext.com

API

In addition to this interface with many features, Sublime Text provides an API to extend its functionality. All plugins are written in Python and rely on the integrated Python environment. This provides almost all the standard Python modules, with a few exceptions. One of the most important features of the API is the ability to access the folder and files of the current project. This allows the contents of the files to be read, edited and deleted. In addition, event listeners can be used to respond to user actions. These include *file events* such as creating, saving or closing a file, or *window events* such as opening, moving or closing a window. Another feature of the API is the ability to create commands that can be executed by the user. These commands can be triggered by defined events or by clicking a button in the menu. To display information to the user, the plugin can use sheets. These are small windows that can be displayed on the side of the interface. The content of these sheets is limited to text, images and HTML elements. Sublime Text provides its own HTML and CSS engine to render the sheets. This is a subset of the normal HTML and CSS syntax. Changing Sublime Text settings is also possible using API. This allows the plugin to change the appearance of the interface, such as the colour scheme. When user input is required, the plugin can use input panels. These are small windows that accept plain text or the selection of items from a list as input.

Once the plugin has been developed, it needs to be packaged. This must be imported into the environment by placing it in a specific Sublime Text directory. The plugin runs in a different process from the editor itself, which should make it more stable and secure.

Finally, Sublime Text's API provides some features that could be useful for the plugin. For example, the plugin could react to events such as saving a file to trigger a scan of the code, or jump directly to the line of an implementation. Also, access to the current file and text selection could be used to simplify adding a new implementation to a requirement. Also, the ability to render a rich user interface is rather limited with the custom HTML and CSS engine, which could make implementing the plugin more difficult. As a result, the rest of the plugin needs to be developed as a standalone application that runs in parallel with a plugin in Sublime Text. The general concepts of such an interface have been described in the previous section of the text editor and shown in the mockups in Figures 4.4, 4.5, 4.6 and 4.7.

4.2.3 Visual Studio Code

Much more advanced features are provided by the IDE Visual Studio Code (VS Code) [31]. Its cross-platform capabilities make it an adaptable tool for developers regardless of their operating system. VS Code offers a wide range of integrated debugging tools. These tools streamline the identification and correction of errors, speeding up the coding process. Another key advantage of VS Code is its extensive API. It provides far more possibilities for developing plugins, called extensions here. This has led to a large community of developers who have created a wide range of extensions that can be downloaded directly from the marketplace. In this section, the UI and API of VS Code are presented in detail.

User Interface

The VS Code interface is divided into several parts, as shown in the Figure 4.9. The main ones are the activity bar (1), the primary sidebar (2), the editor groups (3), the panel (4) and the status bar (5) [3].

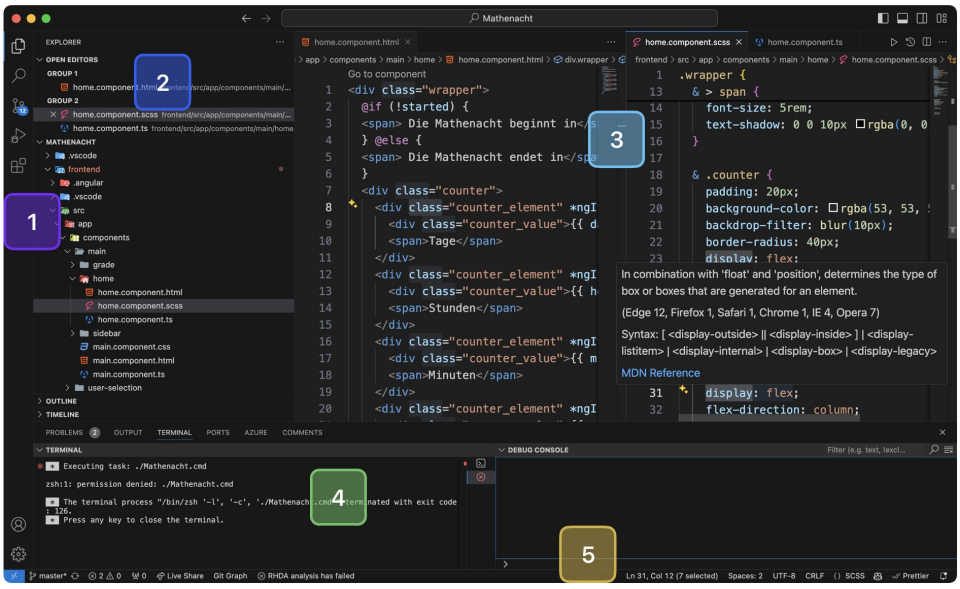

Figure 4.9: Example of the interface of VS Code

[3] https://code.visualstudio.com/docs

The leftmost section contains an activity bar (1) that displays icons for the different views. Clicking on the icons displays the corresponding content in the primary sidebar (2). At the top of the bar is the File Explorer, which provides a tree view of all the files and folders in the current workspace. A workspace can be a single file, a folder, or a collection of folders. These do not necessarily have to be in the same directory on the file system, but can be distributed across multiple locations. The Search view allows users to search for text within the files and folders in their workspace. Users can enter simple keywords or use more complex search patterns using regular expressions. For example, the pattern *requirement[0-9]* matches any string beginning with the word *requirement* followed by a single digit between *0* and *9*. In addition to searching, you can also replace all instances of a pattern with another string. There is also a source control view, which allows you to manage your code using Git. This shows the changes made to the code since the last synchronisation with the repository. This includes additions, deletions and modifications to files. Once you have made the changes you want, you can commit to the repository, add a description of the changes, and commit to the server. The Run and Debug view, located below the source control view, allows users to run and debug their code. It allows the creation of run configurations, which define how the code is executed. Once configured, users can initiate the execution of their code and view the output in the terminal area. If this functionality is not sufficient for the user's needs, they can install extensions from the Marketplace, which are listed in the activity bar at the bottom. There you can browse and add a variety of additional features to VS Code, all for free. The last section at the bottom is the account view, which displays account information and allows users to sign in and out of their account. The account is used to synchronise personal settings and additional extensions across devices, ensuring a consistent experience across platforms without the need to reconfigure settings on each device. An example of the File Explorer, Search View and Git integration is shown in the Figure 4.10.

The Editor Groups (3) form the main area of the interface, where you can view the contents of the files you are currently working on. It is possible to have several files open at once, which are displayed as small tabs at the top of the Editor Area. To view multiple files at once, you can split the editor area into multiple groups, as shown in the Figure 4.9. These can be arranged horizontally or vertically with an adjustable size, allowing you to make the best use of your screen space. Within each editor pane, the contents of the file are displayed, depending on the file type. As well as simple text files, VS Code allows you to view a wide range of file types, including images, videos and PDFs. Opening some file types may require the installation of additional extensions.

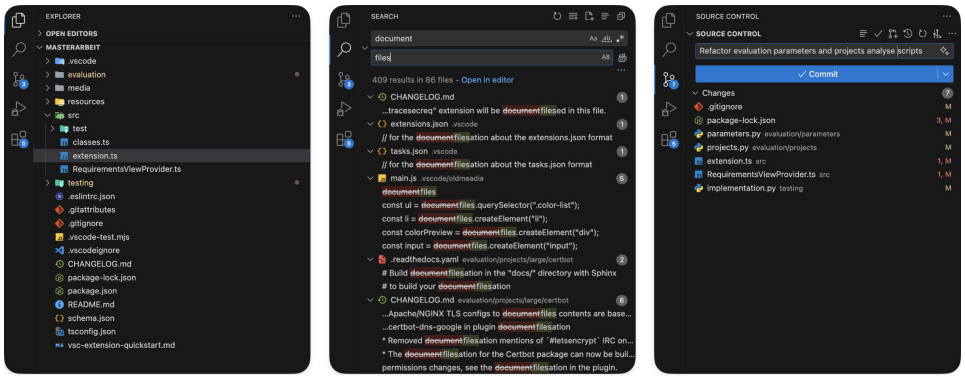

Figure 4.10: Examples of the activity and primary side bar

In addition, VS Code offers a variety of features that go beyond the basic display of file contents. For example, text files have syntax highlighting, which uses colour to differentiate between code components, making them easier to identify. There is also code completion, which provides suggestions for the code you are writing. Code snippets allow you to insert predefined blocks of code using a simple keyword. For example, type *for* and press *tab* to insert the structure of a for loop. The software offers a variety of built-in tools, including the ability to select and rename variables in your file, navigate to another file where a particular function is defined, and view error descriptions by hovering over them, and view error descriptions by hovering over highlighted errors in your code.

Below the editor area is the panel area (4) by default. It contains several features, such as the integrated terminal, which allows you to execute commands in the shell, as shown in the Figure 4.9. Then there is the output view, which can show you the output of the commands or the results of the execution of your code. The problems view lists all the errors and warnings in your code, grouped by file, and can be further filtered.

At the bottom of the interface is the status bar (5), which is also shown in the Figure 4.9. Here is some compact information about the current state of VS Code, such as the language mode of the file you are working on, the line and column number of the cursor position. According to the design guidelines, information here should be displayed in a very compact way, so that it does not take up too much space. In addition to the standard elements, each plugin can add its own information to the status bar.

Apart from the main interface elements, there are a few other features worth mentioning. If there is something important, such as a compilation error or the loading status of a plugin, a notification will appear in the bottom right-hand corner of the interface. Shortcuts allow to quickly perform commands with just a few keystrokes. For example, open, save and close the current file.

If the shortcut for a command is unknown, you can open the Command Palette and search for the desired command. The Quick Open feature allows to search for files in the workspace and open them directly. You can also search for symbols in the code, such as functions, classes or variables, and jump to their definition. These search functions are shown in the Figure 4.11.

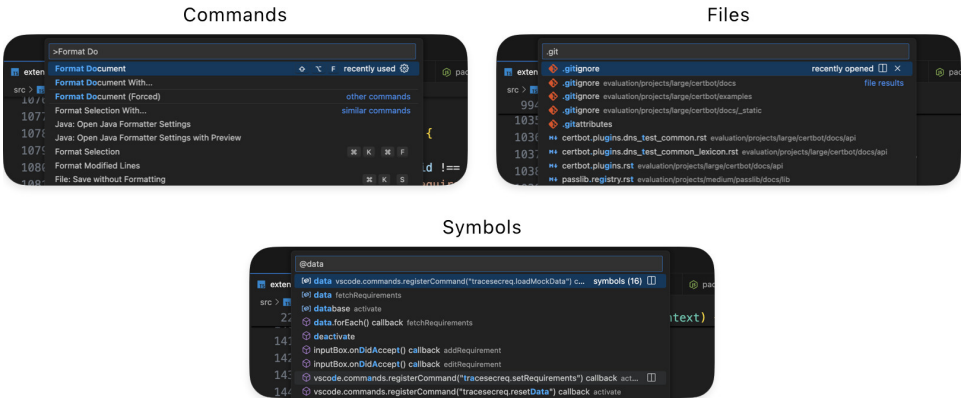

Figure 4.11: Examples of different search types

VS Code is highly customisable to adapt to different developers. The theme of the interface can be changed to a light or dark mode with various colour schemes Furthermore the key bindings can be customised. You can also use the Preferences view to fine-tune a number of features, such as changing the font size or formatting the code when saving a file.

API

But one of the most imporant feature of VS Code for the plugin is the API for developing extensions. Using this, you can add for example support for other code highlighting in many programming languages. Moreover, you can also get new visual themes for the user interface, or additional tools for debugging the code. Most of these plugins are developed by the community, which has created a huge number of plugins for many different use cases. Microsoft provides extensive documentation and tutorials on how to develop a plugin for VS Code.

Almost every feature of the API that Sublime Text has is found here. This includes command creation, event listeners, input panels, file access and more. But a big difference is the ability to create an advanced, directly integrated user interface within the primary sidebar or editor area. The API provides a file explorer-like tree structure that can be used to display information. Going further, HTML, CSS and JS build a web view and allow you to create a rich user interface.

There is a lot of freedom in how the interface is designed, but the API provides access to the colour theme of the user interface, so the plugin can adapt to the common look of VS Code. To start development, you will need to have some software already installed. In addition to VS Code itself, you will need the code version control system Git. Then you need Node.js, the JS runtime engine. This comes with NPM, which is used to install Yeoman and the extension and customisation generator. Once these are installed, you can start developing the plugin.

The backend of the plugin is mostly written in TS and the core of the plugin is the *package.json* file. This defines the name, version, description, author, licence and other metadata of the plugin. Then there are the menu items, which are small buttons with an icon at the top that trigger a custom command when pressed. Other items such as the icon and title in the activity bar are defined. Also listed are the *activationEvents*, which are the events that trigger the activation of the plugin. With 27 different events available, e.g. when a file with a certain language is opened, when a certain command is executed or when you open VS Code itself, the plugin can be activated in many different ways. This activation event is handled in another important part of the backend, the *extension.ts* file, which manages the core behaviour of the plugin. When the plugin is activated, the *activate* function is called, which can be used to register commands to perform actions on the plugin. It also registers event listeners that are triggered when a specific event occurs, such as when a file is opened or saved. These triggers can be used to execute existing commands or any other desired action. Persistent data can be loaded from storage when the plugin is activated.

A mockup of the plugin interface in VS Code is shown in the figure 4.12. On the left is an overview of the requirements. Each requirement is displayed as a card with the same information as in the standalone application. By clicking on a card, the user can see the detailed view of the requirement, which is displayed on the right. While the information displayed is again similar to the standalone application, a key difference is that the plugin interface is right next to the editor area containing the code.

4.3 Final Concept

This chapter has presented several concepts of how the project could be implemented. As a form of interface, the data manager concept seems more suitable, as it contains more features than the search system. VS Code was chosen as the development environment, as it provides a very extensive API for plugin development. In particular, the ability to create a custom UI within the application is a huge advantage. Furthermore, both methods of a buzzword list and LLM are considered for tracing. The buzzword list is easier to implement, but the LLM will probably achieve better results.

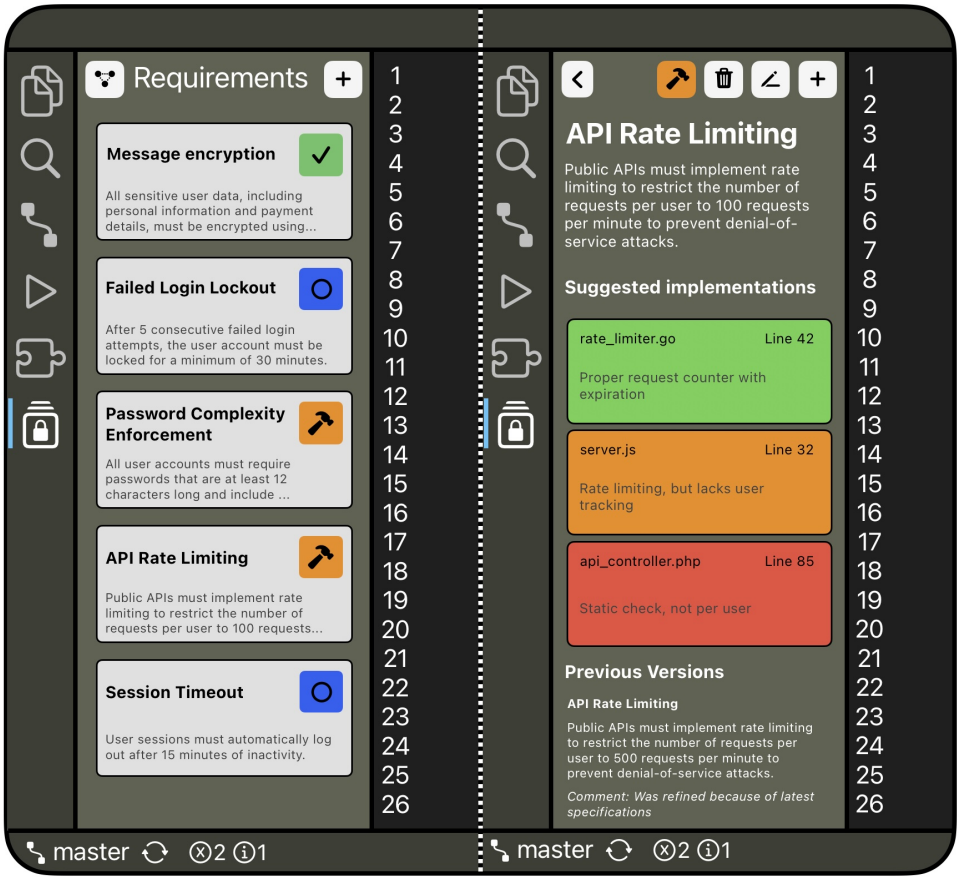

Figure 4.12: Mockup of the interface of the plugin in VS Code

Although, a fine-tuned neural network could provide even more accurate results, no sufficient data was found to possible train it, so it was not further considered.

Chapter 5

Implementation

The most essential features have been implemented in a prototype. The structure of the plugin is shown in Figure 5.1. It will be explained in the following sections.

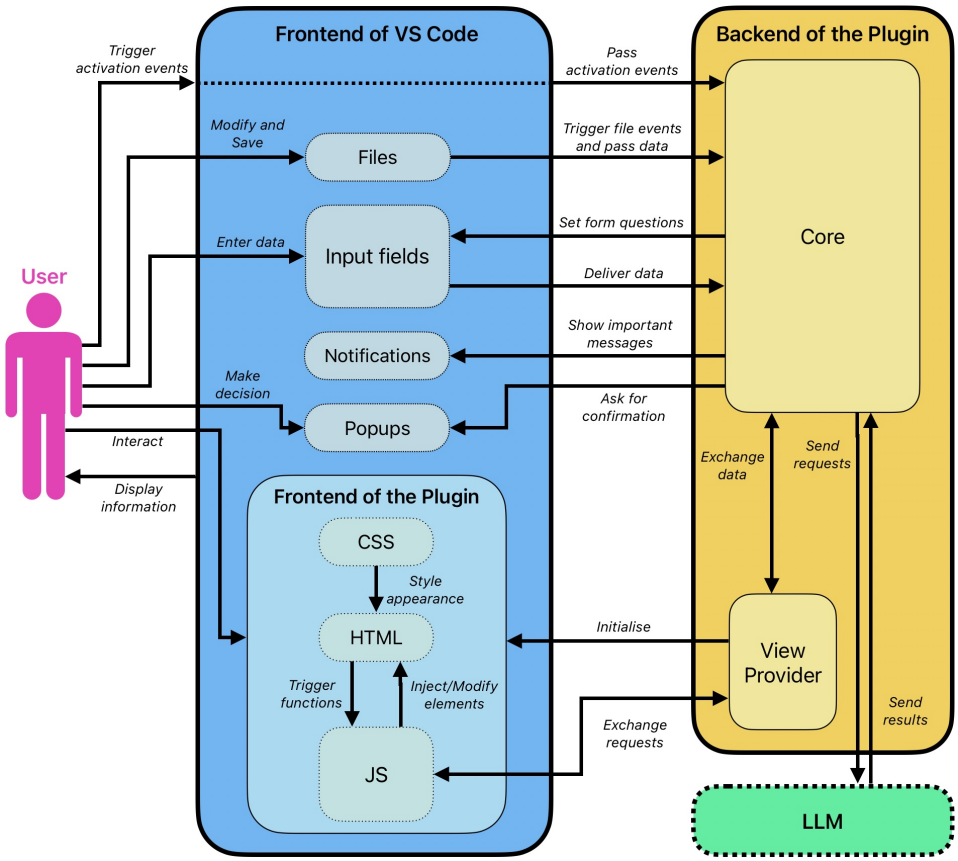

Figure 5.1: Structure of the plugin

5.1 Backend

Most of the functionality of the plugin is in the backend. The plugin's database contains data in JSON format, which is stored as a *workspace state*. The dataset is structured as an array of objects, with each object representing a specific request. Each object contains a title and description provided by the user, as well as several plugin-specific properties. These are an identifier to reference the requirement, a status to indicate whether the requirement is implemented, the date of the last change to the requirement, and a history of changes to the requirement. It also contains an array of implementation objects. Each implementation object has an identifier, a file path, a line number, a score and a comment. Finally, the actual schema is shown in Figures A.2 and A.3.

When the plugin's activation event is triggered, the database is loaded. *onStartupFinished* was chosen as the trigger because it ensures that the plugin can be used immediately without affecting the startup of VS Code. In addition, a number of commands have been registered on activation. Some of the commands are used to add, edit, delete and confirm requests, while another command initiates a refresh of the view. The event to evaluate the current file is set to be triggered by the user when saving a file. This eliminates the need for the user to take any action and ensures a fully automated process. Finally, the view provider for the web view is initialised. Within this provider, the technical details of the web view are defined, including storage locations and startup settings. In addition, communication between the backend and the frontend is set up by creating message listeners and functions to send messages.

5.1.1 Extracting Functions

For the prototype, only the analysis of Python files was implemented. When the save event is triggered, the backend extracts all the functions from the current file. This is achieved by reading the file line by line and looking for the function definition keyword in Python, which is *def*. Once the start of a function is identified, the backend then reads the following lines until the end of the function is reached. This end is indicated by a combination of the indentation level, the content and the current line. Some edge cases are also taken into account, such as the end of the file.

5.1.2 Tracing

The next step is to compare the extracted functions with the specified requirements.

Buzzwords

The first approach is to establish a trace-link between the functions and the requirements using a list of buzzwords. This requires a list of pairs of terms containing the most common words in the requirements and function definitions. Since the relevant part of the code is the function definitions, the list contains the most commonly used words in the function definitions of the security packages. Particular attention has been paid to the cryptography packages, as they are most likely to contain security-critical functions. Only methods were used, but classes and properties etc. could also be used.

First, the most common and prominent ones were searched for. These were *PyCryptodome*, *Cryptography*, *PyNaCl*, *PyOpenSSL*, *Fernet*, *Keyczar*, *M2Crypto* and *asn1crypto*. The relevant documentation was then downloaded and all files in *.rst* format, a common plain text format used for documentation, were extracted. Then all function definitions were extracted and any function names not directly related to cryptography, such as *copy* or *exchange*, were manually filtered out. Functions that served the same purpose, such as encryption, were grouped together. As the requirements are written in natural language, multiple variants of a term in the requirement, such as *encrypt* and *encryption*, were also added to the list. Finally, out of 88 functions in the cryptography packages, a set of 14 pairs containing 44 functions and 57 natural language terms were created. A part of it is shown in Figure A.1.

5.1.3 LLM

An alternative approach is to use an LLM to identify related requirements and implementations. In this instance, the extracted functions are compared to the requirements one by one using a request send to the LLM. The prompt for the LLM is constructed from a system prompt and a user prompt. While the system prompt remains consistent for each comparison, the user prompt is constructed by pairing a requirement with a function. The request is then passed to the LLM for processing, which can take some time. The GWDG (Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen)[1] provides access to a variety of pre-trained models. Through their API, the *Llam-3.1-8B-Instruct* and *Llam-3.1-70B-Instruct* can be accessed. Since the smaller model is faster, it was chosen for the initial implementation. The content of the system prompt and structure of the user prompt is detailed in Chapter 6 in Figure 6.2. A difference to here is, that the LLM is instructed to evaluate how well the requirement matches the function, by assigning a score between 0 and 10. The score of 0 indicates no match, while a score of 10 indicates a perfect match. In addition, the LLM is asked to provide a brief comment on the score assigned.

[1] https://chat-ai.academiccloud.de

This response is then transformed into a JSON object, which is then processed by the backend system. If the score is equal to or greater than 1, indicating a potential correlation between the specified requirement and the associated function, the result is added as a recommended implementation and stored in the database alongside the original requirement.

5.1.4 Manual

In the case that the semi-automatic tracing doesn't find a related implementation, the possibility to manually add it to the requirement was implemented as a backup. This was be achieved by using the VS Code API, which provides a method of getting the current selection in the editor. When the process is initiated by pressing a button, the plugin adds the first line of the selection as a new implementation to the current requirement.

5.2 Frontend

In terms of the frontend, the user interface of the plugin, several approaches were taken to determine the optimal approach for presenting the requirements. The first approach was to use the standard VS Code building blocks to display the requirements. One such approach is the tree view API [2], which effectively displays information in a tree structure. However, it is challenging to display requirements in a tree structure as they are not hierarchical, but rather a list of requirements. The number of ways to display information in this format is limited, as only text and icons can be inserted. In addition, a way was needed for users to interact with the system, such as approving or rejecting a proposed implementation. This could be achieved using buttons in the navigation bar at the top of the primary sidebar, but the usability is not optimal.

As a result, the Web View API was selected as the most suitable option, which offers greater control over the design of the user interface. While this flexibility is beneficial, it has also introduced a higher level of complexity. As the web view runs in an *iframe* container, there is a degree of isolation, which presents a challenge in terms of communication between the web view and the rest of the plugin backend. One solution for data transfer between these components is to use messages via the VS Code API. Each message consists of a name and a payload, which can be any data that can be serialised to JSON. In this particular case, the payload was used to transfer the requirements and implementations to the web view. As these were objects containing strings, numbers and arrays, they could be easily serialised and deserialised. Since the access to local files from the web view is restricted for security reasons, access must be explicitly granted.

[2] https://code.visualstudio.com/api/extension-guides

However, a unique URI must be exchanged between the backend and the frontend to allow access to local files. To further enhance security, the use of scripts in the web view is disabled by default. Nevertheless, to facilitate communication with the backend and dynamic content changes in the web view, it is necessary to enable scripts. Upon receiving data from the backend, the script creates html elements and populates them with the data before appending them to the DOM tree.

The goal was to maintain a subtle design for the user interface. It was important to ensure that the colour scheme of the plugin matched that of VS Code. This was achieved by using the predefined colour scheme of VS Code for the background and text. The web view is accessible to the user via the VS Code sidebar, where the plugin is listed. Clicking on the icon will open the primary sidebar displaying the web view. The user interface has two main components. The first is the requirements list, as shown in Figure 5.2. The navigation bar at the top of the sidebar (1) contains an option to add a new requirement. When the button is clicked, a new input field (2) appears where the user can enter the required data. The first input field is for the title, while the second input field is for the description. Once the input has been submitted, the new requirement is added to the database and displayed in the list of requirements (3). Each requirement is displayed in a card, containing the title, the description, the date of creation and the current status. By default, newly created requirements are in the status 'open'. Each status has its own colour and icon for easy identification. When the evaluation of the current file is triggered, a notification appears in the bottom right-hand corner of the window (4).

When a requirement is selected, the second main component is displayed, providing detailed information about the selected requirement, as shown in Figure 5.3. The navigation bar (1) contains four different buttons as shown. The button on the far left of the interface is used to delete the selected requirement. To prevent accidental deletion, a confirmation pop-up will appear asking the user to confirm their intention to proceed. The second button allows the user to edit the requirement. When the button is clicked, two input fields for the title and description are displayed in sequence. A third input field prompts the user to comment on the changes for future reference. Once the changes have been submitted, the requirement is updated in the database and the view is refreshed to reflect the new data. The third button allows the user to approve the requirement. When the button is selected, a pop-up window appears asking the user to confirm the action. Upon confirmation, the status of the requirement is updated to *finished* and the view is refreshed. Finally, the fourth button allows the user to add the first line of the current selection in the editor as a new implementation. The title and status of the requirement are displayed below the navigation bar (2). To return to the list of requirements, the user should click on the Back button located to the left of the title.

The requirement's description is displayed in a separate section below, next to the creation date (3). The implementations are then displayed in a list of cards (4). Each card displays the first line of the proposed implementation, which the user can select to open the file at that point in the editor. The score and comment provided by the LLM are also displayed. The card displays a colour gradient from green to red indicating the score. The highest score is green and the lowest score is red. An icon in the top right corner of the card indicates that the implementation has been suggested by the AI. When the user hovers the mouse over the card, two buttons appear, allowing the user to either approve or reject the suggestion (5). If the implementation is rejected, it is deleted from the database. Otherwise if the implementation is approved, the score is replaced by a checkmark and the card is coloured blue, similar to when a user adds an implementation. All implementations are sorted by score, from the highest at the top to the lowest at the bottom. Above the highest proposed implementation are those that have already been approved or manually added by the user. As a result of editing the requirement, all suggested implementations are deleted and the previously approved ones are marked as to be reviewed in grey with a question mark. The user can re-evaluate the questionable implementations by re-approving them if they are still valid or rejecting them if they are not.

5.3 Debugging

During the development of the plugin, there were often situations where the code needed to be debugged. Once the project is built, the plugin will start in debug mode and a new instance of VS Code will be opened. The debug mode is similar to the standard version of VS Code, but here the plugin is installed and can be tested. As VS Code is built on top of Electron, the debugging process is similar to web development. Opening the developer tools allows the user to view the DOM tree and console output. As a web view was developed to display the plugin, it was possible to inspect all the elements and make direct changes to the styling. The console output shows log messages, errors and warnings. Raw data was required for numerous test runs, and manually creating and deleting requirements was a time-consuming process. To streamline this process, two commands were added to the plugin: one to clear all data and another to create a set of requirements and implementations. Postman was used to test the GWDG API and with its help the request limit was identified.

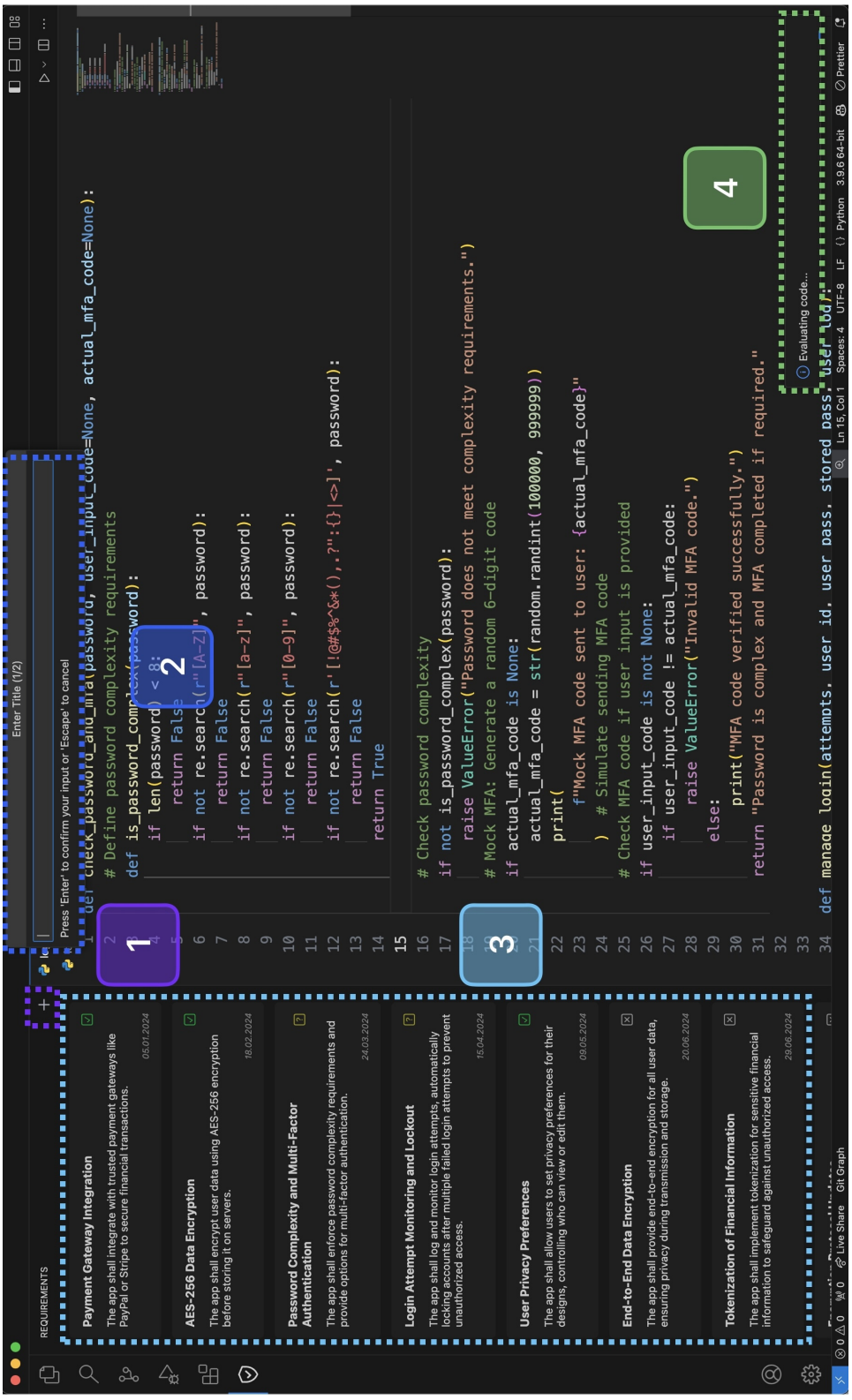

Figure 5.2: Interface of the requirements overview

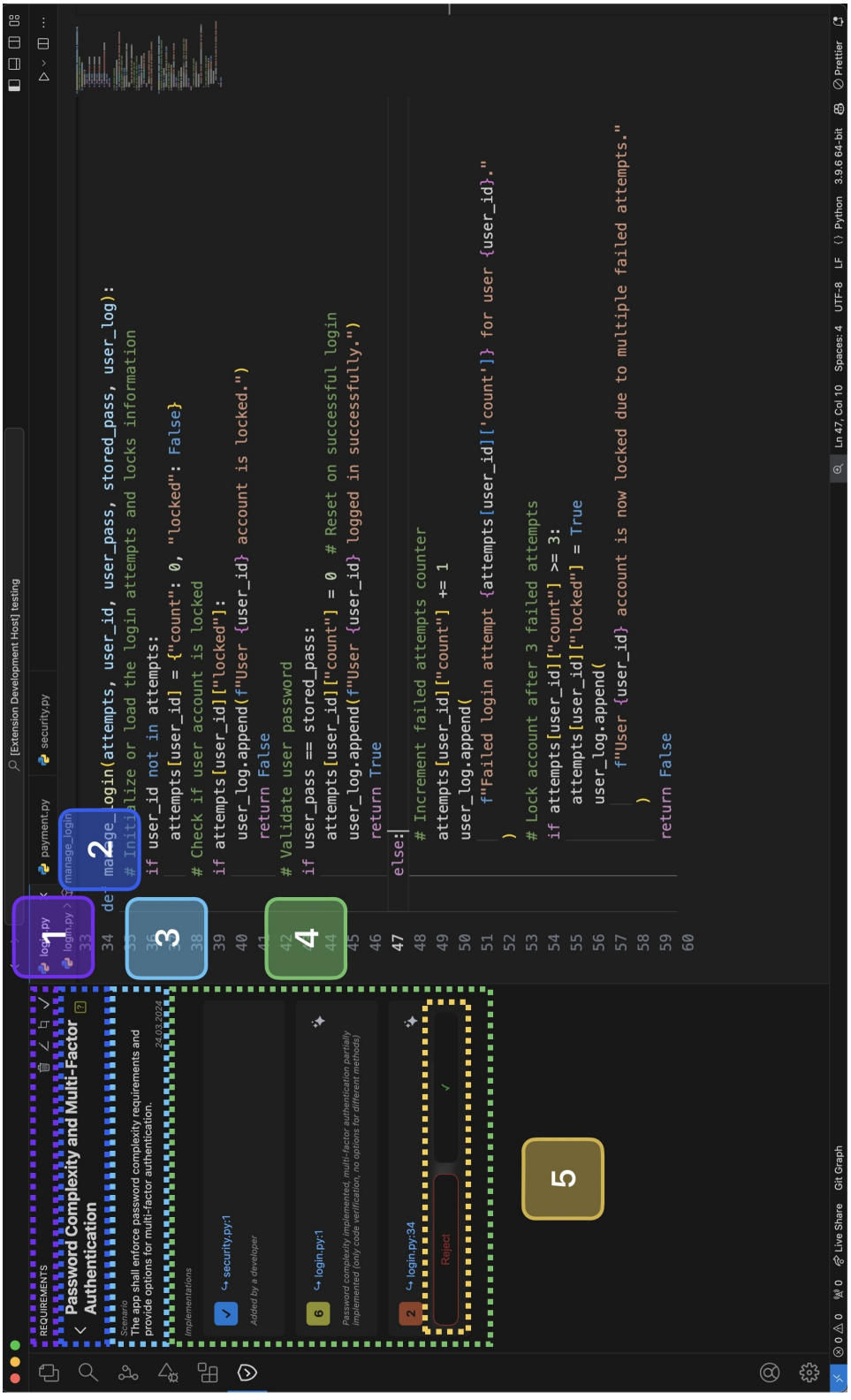

Figure 5.3: Interface of the detail component

Chapter 6

Evaluation

Both concepts of tracing, the buzzword list and the LLM, were implemented in the previous chapter and worked. In small test runs, the LLM showed more promising results than the buzzword list in identifying the correct implementations. Therefore, this evaluation focuses on assessing the ability of an LLM to accurately identify corresponding requirements and code implementations. During the evaluation, a set of requirements will be compared with different types of implementations. These types should cover normal and difficult situations for the LLM to evaluate. After presenting the evaluation procedure and outlining the dataset created and used, the results are presented and discussed.

6.1 Procedure

The following questions are to be addressed at the end of the evaluation:

- **Programming language**: RQ1: How does the choice of programming language (Python, C#, or GO) affect the LLM's ability to match requirements with implementations?
- **Implementation Type**: RQ2: How accurately does the LLM match requirements to fully implemented functions, and does the accuracy of the matches decrease with incomplete implementations? RQ3: How does the use of randomized, nonsensical variable names affect the LLM's ability to identify the correct matches between a requirement and an implementation ?
- **Comments**: RQ4: To what extent does the presence or absence of comments in the code influence the LLM's capability to accurately identify matching requirements?

- **Pair Type**: RQ5: Does the accuracy of the matching change when the LLM evaluates a pair of multiple requirements and implementations simultaneously, compared to pairs of single entities?
- **LLM**: RQ6: How does the choice of LLM model impact the accuracy of the matching?

A part of Hassine's set of requirements [26], shown in Figure A.4, contains several security-related requirements covering a broad range of issues. The following four requirements were selected as the basis for creating the set of corresponding implementations. Since several varieties will be created for each requirement, the inital amount was kept small to keep the evaluation manageable.

- The app shall integrate with trusted payment gateways like PayPal or Stripe to secure financial transactions.
- The app shall encrypt user data using AES-256 encryption before storing it on servers.
- The app shall enforce password complexity requirements and provide options for multi-factor authentication.
- The app shall log and monitor login attempts, automatically locking accounts after multiple failed login attempts to prevent unauthorized access.

First, Python functions were created from the requirements using ChatGPT-4o through LuhKI, an interface provided by the LUH. To keep the style largely similar, these implementations were translated to GO and `C#`. Next, the set was duplicated and the second half of the implementations were removed to form an incomplete set of implementations. To further determine if the LLM really understood the code or just checked for words that appeared in the functions, the initial set was duplicated again and all characters of variable names were replaced with random characters. All code required for the functionality was retained. These three sets were duplicated again and the comments were removed from the functions. A total of 72 implementations were created in this way. Furthermore, the type of requirement-implementation pairing was varied by using a prompt for each pair (1x1), for all requirements with one implementation (Nx1), for one requirement with all implementations (1xM), and for all requirements with all implementations (NxM). The API of GWDG has a tight request limit (198 requests per hour, 1000 per day and 2999 per month). As a consequence, the evaluation data was processed on the computer cluster at the Software Engineering Group at LUH. To connect to the cluster and write the code for the evaluation, the *Remote - SSH* extension for VS Code was used.

Since LLaMA and CodeLlama showed good results in understanding the code [32], these two LLMs were used for the evaluation. Both were publicly available for research purposes through Hugginface, a website that provides data and tools in the field of machine learning[1]. Since code analysis requires logical reasoning, the temperature was set at 0.01 to give the most deterministic results. It was also planned to use models with different parameter amounts to see the influence of model size on the results. Unfortunately, running the higher models resulted in insufficient memory errors on the cluster. Since the requirments are written in natural language, they have a similar structure like those that are produced following a cookbook concept. In a small comparison to corresponding requirements that are written in the Gherkin format, the natural language ones were better matched than the Gherkin ones. Furthermore, adding more requirements to the dataset would have increase the complexity of the evaluation even more. Therefore, the evaluation was limited to the 12 security-related requirements in Figure A.4. A visual representation of the evaluation process explained earlier is shown in Figure 6.1. It shows the different stages of how a request is formed and sent to the LLM until it outputs the results.

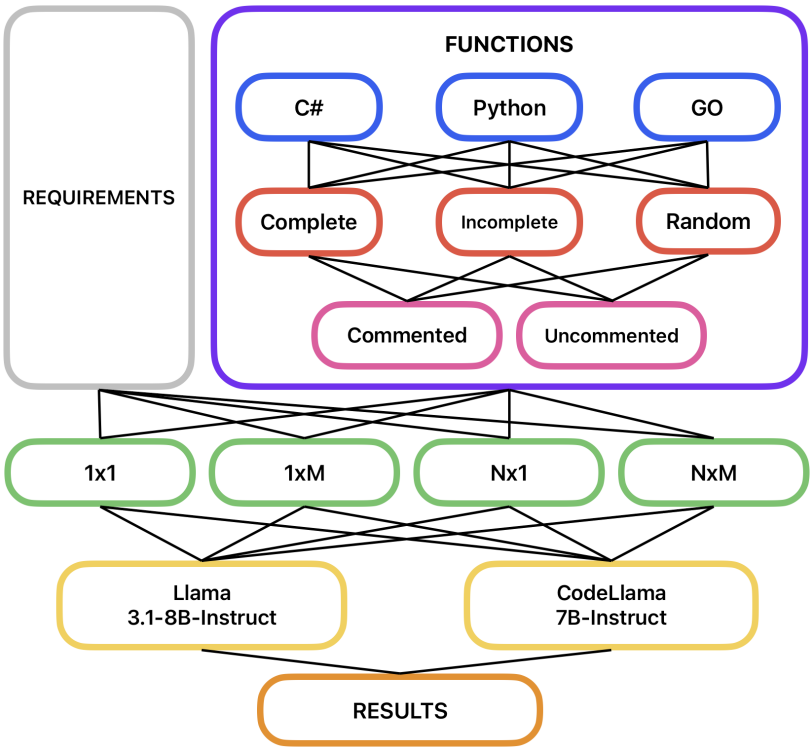

Figure 6.1: Evaluation procedure

[1]https://huggingface.co

6.2 Prompt

To retrieve an evaluation within the response by an LLM, a system prompt must be crafted to instruct the LLM on what to do. The prompt for the *1xM* pair type is shown in figure 6.2.

```
1  You are an expert software engineer that gets a requirement and many
   functions.
2  Each input will be structered like "REQUIRMENT-{ID}: {requriement}
   IMPLEMENTATIONS: [IMPLEMENTATION-{ID}: {implementation}|||
   IMPLEMENTATION-{ID}: {implementation}|||.....]".
3  Form pairs of the requirement with each implementation.
4  For each pair your evaluation will be "true" or "false", depending if
   the requirement is implemented in the function.
5  "true" should be given if the requirement is implemented in the
   function.
6  "false" should be given if the requirement is not implemented in the
   function.
7  Your output should be in json format, with the following values:
8  [{"REQ": <requirement-id>, "IMP": <implementation-id>, "evaluation":
   <true or false>}]
9  Here are the data types: "REQ" and "IMP" are integers, "evaluation" is a
   boolean.
10 Output only this json data, don't add explanations or comments after it!
```

Figure 6.2: System prompt for the *one requirment - many implemenations* (1xM) pair type

The first line presents the context to the model. It then defines the input format that the model should use to create pairs. Lines 4-6 tell the model how to evaluate the pairs. Moreover, Lines 7 and below deal with the output format, which should be JSON with certain data types. This is crucial, because an unstructured output would make it difficult to evaluate the results. As the model tends to describe its output afterwards, it is instructed not to do so in the last line. Nevertheless, the model does not just output a JSON object from time to time. An attempt was made to use a library called *Jsonformer*[2] to generate only JSON objects from the output, but it couldn't be made to work with the current setup. So the output is additionally filtered for an array of objects and if it is not present, the output is discarded. It was also intended to include a comment for each evaluation to see the reasoning behind the decision.

[2]https://github.com/1rgs/jsonformer

Unfortunately, the amount of output tokens increased too much, resulting in discarded data, so the comment was removed before the evaluation.

6.3 Data

To visualise the test data set, an example of a requirement is shown below, along with the corresponding implementation types. The requirement is *The app shall encrypt user data using AES-256 encryption before storing it on servers*. The generated implementation in Python is shown in the Figure 6.3. In this case, the implementation is complete and commented. An incomplete implementation is also shown in Figure 6.4. Here the second half of the implementation code has been removed. Finally, Figure 6.5 shows a commented implementation with random characters. For each implementation there is also a commented and uncommented version.

```python
def encrypt_user_data(user_data, encryption_key):
    # Ensure the provided encryption key is 32 bytes long for AES-256
    if len(encryption_key) != 32:
        raise ValueError("Encryption key must be 32 bytes long for AES-256 encryption.")
    # Generate a random 16-byte initialization vector (IV)
    iv = os.urandom(16)
    # Create a Cipher object using AES algorithm, CBC mode, and the given key and IV
    cipher = Cipher(algorithms.AES(encryption_key), modes.CBC(iv), backend=default_backend())
    # Prepare the padder to pad the user data to a multiple of block size (16 bytes for AES)
    padder = padding.PKCS7(algorithms.AES.block_size).padder()
    # Pad the user data to ensure proper block alignment
    padded_data = padder.update(user_data.encode()) + padder.finalize()
    # Initialize the encryptor object
    encryptor = cipher.encryptor()
    # Encrypt the padded data
    encrypted_data = encryptor.update(padded_data) + encryptor.finalize()
    # Return the IV and encrypted data (both are needed for decryption)
    return iv, encrypted_data
```

Figure 6.3: Example of an implementation - Type *complete* and *commented*

```python
def encrypt_user_data(user_data, encryption_key):
    if len(encryption_key) != 32:
        raise ValueError("Encryption key must be 32 bytes long for AES-256 encryption.")
    iv = os.urandom(16)
    cipher = Cipher(algorithms.AES(encryption_key), modes.CBC(iv), backend=default_backend())
    padder = padding.PKCS7(algorithms.AES.block_size).padder()
```

Figure 6.4: Example of an implementation - Type *incomplete* and *uncommented*

```python
def yuiopa(saöfads, wenrglnwig):
    # Ensure the provided encryption key is 32 bytes long for AES-256
    if len(wenrglnwig) != 32:
        raise ValueError("Encryption key must be 32 bytes long for AES-256 encryption.")
    # Generate a random 16-byte initialization vector (IV)
    klane = os.urandom(16)
    # Create a Cipher object using AES algorithm, CBC mode, and the given key and IV
    hasdhf = Cipher(algorithms.AES(wenrglnwig), modes.CBC(klane), backend=default_backend())
    # Prepare the padder to pad the user data to a multiple of block size (16 bytes for AES)
    dfnsse = padding.PKCS7(algorithms.AES.block_size).padder()
    # Pad the user data to ensure proper block alignment
    bnpirwbi = dfnsse.update(saöfads.encode()) + dfnsse.finalize()
    # Initialize the encryptor object
    jlblasv = hasdhf.encryptor()
    # Encrypt the padded data
    bvoqqib = jlblasv.update(bnpirwbi) + jlblasv.finalize()
    # Return the IV and encrypted data (both are needed for decryption)
    return klane, bvoqqib
```

Figure 6.5: Example of an implementation - Type *random* and *commented*

6.4 Results

A total of 6912 evaluations should be created per iteration. Since the results were uncomplete due to broken output and not all single pairs of requirements and implementations weren't constructed by the LLM, the test data was evaluated three times to mitigate the risk of missing data. After the three executed iterations, a total of 20736 (3*6912) evaluations should have been received. At the end, only 16.186 evaluations were obtained, which is about 78% of the expected outcome. In the following, the results are presented in Figures 6.6 and 6.7.
It should be noted that the tables are sorted by the categories *Implementation Type*, *Commented*, *LLM*, *Programming Language* and *Pair Type*. In each category the best f1-score is highlighted in blue and the worst in red. For example, in the case of *Implementation Type - Complete*, 421 evaluations were TP, 63 were FP, 3760 were TN and 992 were FN. Of the 484 evaluations that were identified as related, 421 were correctly identified, which results in a precision of 0.87. Furthermore, of the 1413 evaluations, where the implementation was related, only 421 were correctly identified by the LLM, resulting in a recall of 0.45. This gives a total f1-score of 0.3. Within the category *Programming language*, the results differ only slightly between them. Here, the GO got the best f1-score of 0.47 while Python got the worst with 0.43. So *RQ1* can be answered that the choice of programming language has no clear influence on the results. Surprisingly, the best f1-score of the *Implementation Type* was not the *Complete* with 0.45 but the *Incomplete* with 0.53. *Complete* has a better precision, but *incomplete* has a better recall.

Figure 6.6: Results of the evaluation - Part 1

	Implemention Type			Commented		LLM	
	Complete	Incomplete	Random	Yes	No	Llama	CodeLlama
TP	421	264	506	676	515	715	476
FP	63	146	169	186	192	262	116
TN	3760	3324	5048	6631	5501	9607	2525
FN	992	318	1175	1377	1108	834	1651
Precision	0,87	0,64	0,75	0,78	0,73	0,73	0,8
Recall	0,3	0,45	0,3	0,33	0,32	0,46	0,22
F1-Score	0,45	**0,53**	*0,43*	**0,46**	*0,44*	**0,56**	*0,35*

Figure 6.7: Results of the evaluation - Part 2

	Programming Language			Pair Mode			
	Python	C#	GO	1x1	1xM	Nx1	NxM
TP	371	581	239	224	356	205	406
FP	123	162	93	92	151	129	6
TN	3700	6039	2393	2856	5606	2732	938
FN	886	1162	437	446	1047	892	100
Precision	0,75	0,78	0,72	0,71	0,7	0,61	0,99
Recall	0,3	0,33	0,35	0,33	0,25	0,19	0,8
F1-Score	*0,43*	0,46	**0,47**	0,45	0,37	*0,29*	**0,88**

To adress *RQ2*, the LLM made many FN evaluations and therefore the f1-score is lower. Matches with *Incomplete* implementations had instead an increased f1-score. Since the difference between the f1-score of *random* implemenations with *complete* ones is quite small, *RQ3* can be answered that the use of randomized, nonsensical variable names has no clear influence on the results. It is assumed that the LLM doens't rely on variable names to make its evaluations. Adding comments to the functions has only a minimal positive effect on the results, with a better f1-score of 0.46 compared to 0.44, which answers *RQ4*. For the category *Pair Type*, the best results by far were obtained with the *all requirements - all implementations (NxM)* mode with a f1-score of 0.88. One reason for this could be that the total number of extracted evaluations was quite low compared to the other modes. Alternatively, the LLM could see the bigger picture of all the data and therefore evaluate the pairs better. The worst results were obtained with the *all requirements - one implementation (Nx1)* mode with a f1-score of 0.29. Answering *RQ5*, the accuracy of the matching differes strongly between different pair types, while the best results were obtained with the (NxM) type. A clear difference can also be seen between the results of the *LLM*s *CodeLlama* and *LLama*. Here LLaMA has a better f1-score of 0.56 compared to 0.35 of *CodeLlama*. One reason for this could be that *CodeLlama* is a smaller model with 7B parameters, while *LLama* has 8B parameters. So the answer to *RQ6* is that the choice of LLM model clearly impacts the accuracy of the matching. When iterating over all combinations possible between the categories, there were 14 combinations with a f1-score of 1 and 6 combinations with a f1-score of 0. In total 26 combinations had no evaluations at all due to errorneous and uncomplete output of the LLMs. Despite the fact that *Incomplete* implementations were rated best on average in their category, all f1-scores of 0 were obtained with this type of implementation. For example the combination *C#*, *Incomplete*, *Commented*, *LLama*, *1x1* had a TP of 0, FP of 16, TN of 176 and FN of 0. In this case, the LLM did not make any true positive matches and therefore gets a f1-score of 0. On the other hand, the combination of *CSharp*, *Complete*, *Randomised*, *CodeLlama* and *NxM* had a f1-score of 1 with a TP of 6, FP of 0, TN of 66 and FN of 0. This shows the LLM was able to make correct evaluations even if random characters are used.

Taking the best performing pair type (NxM) and the LLM (Llama-3.1-8B-Instruct) into account in combination with complete and incomplete implementations with comments or without, the TP was 108, FP was 4, TN was 315 and FN was 37. This results in a precision of 0.96, a recall of 0.74 and a f1-score of 0.84. These results are promising and show that the LLM can give reliable results in when using the right configuration and model.

Chapter 7

Discussion

The capabilities of the API of VS Code are quite extensive and have led to many features being implemented. Some big advantages were the ability to react directly to events and user input in the IDE and the freedom to create a custom webview to display the information in the desired form. An overview of all requirements of a single requirement was created to quickly view the implentation status to identify requirements that are not yet implemented. In the detail view, a user can see all confirmed and suggested functions that implement the requirement, with a score and comment given by the LLM.

The results of the evaluation showed that comments are not necessary for better matchings between requirements and implementations and that the LLM can handle different programming languages equally well. Furthermore, the LLM *Llama-8B-Instruct* in combination with the (NxM) pairing of requirements and implementations produced very promising results with a precision of 0.96, a recall of 0.74 and a f1-score of 0.84. Even though the precision is quite high, a human should still check the results to ensure that the trace-links are valid. Moreover, since the recall shows that not all trace-links are found, a human must also create the missing trace-links manually. This is especially important with security-critical requirements, where each one must be implemented correctly. Concluding, in this case semi-automatic tracing between requirements and code implementation was successfully established.

Threats to Validity

A major challenge was to instruct the LLM to generate an accurate evaluation of how well different code implementations match the requirements. To achieve this, prompt engineering was a crucial step. It was also a challenge to instruct the LLM to use a consistent output format for later processing. In the end, it was achieved to get results in a almost desired form, but even multiple lines of instructions *not* to comment the output could not prevent the LLM from commenting the output anyway.

The *Jsonformer* tool could assist here to get a more consistent output format. An attempt was made to use it, but it wasn't possible to integrate it into the existing evaluation functionalities. Overall, further development of the prompt and output format could lead to even better results. However, the current results are promising and show that the LLM can give reliable results in some situations. The dataset used in this thesis was relatively small due to the exponential amount of combinations created from it. Again, the output size of the responses caused many data points to be discarded, which could have reduced the meaningfulness of the results. A dataset that contains more requirements and implemenations could address this issue. In addition, the implementations were generated by an LLM because of unsufficient porficencies in the used programming languages. This could led to less meaningful results than if they had been created by a humann, as real developers may write code differently. Only three programming languages were used, so it is not clear how well the LLM would work in other languages. Different LLMs, or ones of the same type as those used, but with more parameters, could have been used to get better results. Moreover, the differences between the use of the Gherkin format and the natural language for requirements were only briefly evaluated, which could be further investigated. While only zero-shot prompts were used in this thesis, using examples in the prompt could lead to even better results. Since no big dataset was found, the approach of training a custom neural network was not further followed in favor of working with a pre-trained LLM. Nevertheless, it would be interesting to see how well it would perform compared to the LLMs. There could be even more concepts of establishing trace-links between requirements and code that were not considered in this thesis. Furthermore, since security-critical requirements and code are handled, the data sent through the LLM should not be used for any other purposes, which might led to a security risk. This can be ensured by running the LLM on an internal server or even locally on the user's computer. If the plugin is not user-friendly or requires a lot of effort to use it, developers are unlikely to adopt it. Looking at the buzzword list for creating trace-links between requirements and code, it seems unlikely that it will be able to compete with the LLMs. In addition, a complete list of buzzwords might require a lot of effort to create and maintain. A study would be needed to support these claims. Since it is a plugin for VS Code, it is not compatible with other IDEs. Adding or removing code before a function shifts the function's line number accordingly, which in turn affects the validity of trace-links. This is handled by the plugin by removing all trace-links to that file and re-evaluating the file from the beginning. In this case, an intelligent algorithm could be implemented to update only the line numbers of the affected trace-links. Using the API to its fullest could take the plugin even further.

Chapter 8

Conclusion

The problem of tracing requirements back to their implementation in code is a common problem in software development, leading to poorly implemented requirements. This is particularly important in security-critical software, where correct and complete implementation of requirements is essential. As an approach to this problem, a prototype of a plugin has been developed that is directly integrated into VS Code and uses an LLM to allow semi-automatic tracing between requirements and their implementation in code. VS Code was chosen as the platform for the plugin because it is a widely used IDE and the API for creating any kind of plugin is quite extensive. It opens up even more features to extend the capabilities of the plugin beyond what is currently developed. To establish the trace-links between the requirements and the code, the backend of the plugin first analyses the code and then divides it into functions. These functions are then sent in combination with the requirements to an LLM to determine how well a function implements a requirement, if at all. The frontend of the plugin displays all the information in a custom web view in the sidebar of VS Code, created with HTML, CSS and JS. It contains two main views, an overview of all requirements and a detail view of a single requirement. Within the detailed view, the user can see all the functions that implement the requirement, with a score and comment given by the LLM.

The capabilities of several LLMs were evaluated to determine how well they perform in creating accurate trace-links between requirements and code. This was done by creating a dataset containing security-critical requirements and their implementations in Python, C# and Go using an LLM. Each implementation set was further diversified by removing comments, making the implementation incomplete, or randomising variable names to see how well it understood even difficult code that would be challenging even for a human analyser. These requirements and implementations were paired in four different ways, such as one-to-one and one-to-many, to see how much the LLM could handle at once while still giving accurate results.

The results of the evaluation showed promising outcomes when using the right model and pairing of requirements and implementations. There was no clear difference with the presence or absence of comments or the choice of programming language.

Chapter 9

Future Work

As this plugin is a prototype, there are many opportunities to extend the plugin's capabilities in future work. Within the editor, the plugin could be further integrated into the development process. For example, each function that implements a requirement could have a link to the requirement displayed above the function. When the user has implemented a requirement, a notification could be displayed to indicate that the requirement has been implemented. A synchronisation feature with data from other developers could also be implemented to reduce redundant work. It could be logged how long and who worked on which implementation. This could be used to see all the developers responsible for the implementation and how long it took to implement the requirement. The trace-links created between requirements and code could be integrated with other tools to show the overall progress of the project. Alternatively, the plugin could receive other linked artefacts, such as mockups, to be displayed in the requirements detail view, providing a more comprehensive view of how the requirement is to be implemented. In addition, the plugin could be used to generate tests for the requirements and automatically run them against the code. This could help to ensure that the requirements are implemented correctly and that the code works as expected. The LLM could even be used to generate the implementations for the requirements, or to finish partially implemented functions. If the requirement has changed, this could also be used to indicate what needs to be changed in the code. In these cases, the plugin could use the LLMSecGuard framework to generate safe code suggestions. To increase the security of the plugin, the data being stored and processed could be encrypted and the LLM could be run locally on the user's computer or run on an internal server. Furthermore, a local instance would reduce latency and work even when the user is offline. This would require the LLM to be small and efficient enough, or a powerful enough computer. The developed plugin was based on the presented data manager approach.

Some features of the search system could also be integrated into the plugin, such as a search bar. The other types of visualisation, such as the graph and flow view, could also be integrated into the plugin. Some features that aren't implemented yet are filtering and sorting of requirements in the overview.

The capabilities of LLMs could be tested even more to identify potential weaknesses or methods to achieve better results. This would include testing other LLMs with many more parameters, such as *Llama-3.1-405B-Instruct*. Also, the essential part of prompt engineering could be further investigated to find a more appropriate prompt for evaluating a requirement-implementation pair. In addition, the approach of training a custom neural network specifically designed to identify the relationships between requirements and implementations could also be followed to see if it performs better than the mentioned LLMs.

It would be interesting to see the plugin in action and the real impact it would have on the fulfilment of requirements. A user study could be conducted where the plugin is used in a real project from start to finish. This could also be compared to the use of other tools for automatic or semi-automatic tracing of requirements to code. Since the LLMs used are not only trained on security-critical code, it would be interesting to see how well they perform on all kinds of requirements. Also, with some modification, the plugin could be used to analyse a completed project to see how well the requirements were implemented.

Appendix A

Appendix

Figure A.1: Part of the buzzword list used in the plugin

```
[
    {
        "functions": [
            "encrypt",
            "encrypt_at_time",
            "encryptor",
            "encryption_builder",
            "encrypt_and_digest"
        ],
        "buzzwords": ["encrypt", "encryption", "secure", "protection"]
    },
    {
        "functions": [
            "decrypt",
            "decrypt_at_time",
            "decryptor",
            "decrypt_and_verify"
        ],
        "buzzwords": ["decrypt", "decryption", "decode", "decoding"]
    },
...
]
```

Figure A.2: JSON Schema for the requirements - Part 1

```json
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "title": "Security Requirements",
    "description": "A collection of security requirements for a software project",
    "type": "array",
    "items": {
        "type": "object",
        "properties": {
            "id": {
                "description": "The unique identifier for a requirement",
                "type": "string",
                "format": "uuid"
            },
            "title": {
                "description": "The title of the requirement",
                "type": "string"
            },
            "description": {
                "description": "A detailed description of the requirement",
                "type": "string"
            },
            "status": {
                "description": "The status of the requirement",
                "todo": "change new to open, found to in progress, confirmed to finished",
                "type": "string",
                "enum": ["open", "in_progress", "finished"]
            },
            "date": {
                "description": "The date the requirement was last modified",
                "type": "string",
                "format": "date"
            },
            "implementations": {
                "description": "Possible implementations of the requirement",
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {
                        "id": {
                            "description": "The unique identifier for a implementation",
                            "type": "string",
                            "format": "uuid"
                        },
```

Figure A.3: JSON Schema for the requirements - Part 2

```
                        "filePath": {
                            "description": "The name of the file where the requirement is
implemented",
                            "type": "string",
                            "format": "uri"
                        },
                        "lineNumber": {
                            "description": "The line number in the file where the requirement
is implemented",
                            "type": "integer"
                        },
                        "score": {
                            "description": "The score of the implementation",
                            "type": "number",
                            "minimum": 1,
                            "maximum": 10
                        },
                        "comment": {
                            "description": "A comment on the implementation",
                            "type": "string"
                        }
                    }
                }
            },
            "history": {
                "description": "A list of changes to the requirement",
                "type": "array",
                "items": {
                    "type": "object",
                    "properties": {
                        "date": {
                            "description": "The date of the change",
                            "type": "string",
                            "format": "date"
                        },
                        "title": {
                            "description": "The current title of the requirement",
                            "type": "string"
                        },
                        "comment": {
                            "description": "A comment of the change",
                            "type": "string"
                        }
                    }
                }
            }
        },
        "required": ["id", "title", "description"]
    }
}
```

Figure A.4: Requirements by Hassine [26] for the evaluation

The app shall integrate with trusted payment gateways like PayPal or Stripe to secure financial transactions.
The app shall encrypt user data using AES-256 encryption before storing it on servers.
The app shall enforce password complexity requirements and provide options for multi-factor authentication.
The app shall log and monitor login attempts, automatically locking accounts after multiple failed login attempts to prevent unauthorized access.
The app shall allow users to set privacy preferences for their designs, controlling who can view or edit them.
The app shall provide end-to-end encryption for all user data, ensuring privacy during transmission and storage.
The app shall implement tokenization for sensitive financial information to safeguard against unauthorized access.
The app shall regularly update encryption protocols to maintain strong data security standards.
The app shall use machine learning algorithms to detect patterns indicative of fraudulent behavior.
The app shall notify users and administrators of suspicious activities and provide guidance on how to secure their accounts.
The app shall regularly undergo security audits and penetration testing to identify and address potential vulnerabilities, ensuring enhanced app security.
The app shall employ the principle of least privilege, granting access permissions to users based on their roles and responsibilities to minimize the risk of unauthorized access.

Acknowledgements

I would like to thank my family and friends for their ongoing encouragement and support. Furthermore, I would also like to thank my supervisor Alexander Specht for his guidance and time spent every week throughout the course of this thesis.

Bibliography

[1] Garima Bhardwaj et al. "Cyber Threat Landscape of G4 Nations: Analysis of Threat Incidents & Response Strategies". In: *2021 2nd International Conference on Intelligent Engineering and Management (ICIEM)*. IEEE. 2021, pp. 75–79.

[2] Ernani César dos Santos and Patrícia Vilain. "Automated Acceptance Tests as Software Requirements: An Experiment to Compare the Applicability of Fit Tables and Gherkin Language". In: *Agile Processes in Software Engineering and Extreme Programming*. Ed. by Juan Garbajosa, Xiaofeng Wang, and Ademar Aguiar. Cham: Springer International Publishing, 2018, pp. 104–119. ISBN: 978-3-319-91602-6.

[3] Nils Brede Moe, Torgeir Dingsøyr, and Tore Dybå. "Understanding Self-Organizing Teams in Agile Software Development". In: *19th Australian Conference on Software Engineering (aswec 2008)*. 2008, pp. 76–85. DOI: 10.1109/ASWEC.2008.4483195.

[4] O.C.Z. Gotel and C.W. Finkelstein. "An analysis of the requirements traceability problem". In: *Proceedings of IEEE International Conference on Requirements Engineering*. 1994, pp. 94–101. DOI: 10.1109/ICRE.1994.292398.

[5] J.H. Hayes, A. Dekhtyar, and J. Osborne. "Improving requirements tracing via information retrieval". In: *Proceedings. 11th IEEE International Requirements Engineering Conference, 2003.* 2003, pp. 138–147. DOI: 10.1109/ICRE.2003.1232745.

[6] Patrick Rempel and Parick Mäder. "Preventing Defects: The Impact of Requirements Traceability Completeness on Software Quality". In: *IEEE Transactions on Software Engineering* 43.8 (2017), pp. 777–797. DOI: 10.1109/TSE.2016.2622264.

[7] Rashidah Kasauli et al. "Requirements engineering challenges and practices in large-scale agile system development". In: *Journal of Systems and Software* 172 (2021), p. 110851. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2020.110851. URL: https://www.sciencedirect.com/science/article/pii/S0164121220302417.

[8] Siv Hilde Houmb et al. "Eliciting security requirements and tracing them to design: an integration of Common Criteria, heuristics, and UMLsec". In: *Requirements Engineering* 15.1 (2010), pp. 63–93. DOI: 10.1007/s00766-009-0093-9. URL: https://doi.org/10.1007/s00766-009-0093-9.

[9] Thazin Win Win Aung, Huan Huo, and Yulei Sui. "A literature review of automatic traceability links recovery for software change impact analysis". In: *Proceedings of the 28th International Conference on Program Comprehension*. 2020, pp. 14–24.

[10] Arya Kavian et al. "LLM Security Guard for Code". In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. EASE '24. Salerno, Italy: Association for Computing Machinery, 2024, pp. 600–603. ISBN: 9798400717017. DOI: 10.1145/3661167.3661263. URL: https://doi.org/10.1145/3661167.3661263.

[11] Orlena Gotel et al. "Traceability Fundamentals". In: *Software and Systems Traceability*. Ed. by Jane Cleland-Huang, Orlena Gotel, and Andrea Zisman. London: Springer London, 2012, pp. 3–22. ISBN: 978-1-4471-2239-5. DOI: 10.1007/978-1-4471-2239-5_1. URL: https://doi.org/10.1007/978-1-4471-2239-5_1.

[12] Anders Hejlsberg et al. *The C# programming language*. Pearson Education, 2008.

[13] Jeff Meyerson. "The Go Programming Language". In: *IEEE Software* 31.5 (2014), pp. 104–104. DOI: 10.1109/MS.2014.127.

[14] KR Srinath. "Python - the fastest growing programming language". In: *International Research Journal of Engineering and Technology* 4.12 (2017), pp. 354–357.

[15] Erik Cambria and Bebo White. "Jumping NLP Curves: A Review of Natural Language Processing Research [Review Article]". In: *IEEE Computational Intelligence Magazine* 9.2 (2014), pp. 48–57. DOI: 10.1109/MCI.2014.2307227.

[16] Ming Zhou et al. "Progress in Neural NLP: Modeling, Learning, and Reasoning". In: *Engineering* 6.3 (2020), pp. 275–290. ISSN: 2095-8099. DOI: https://doi.org/10.1016/j.eng.2019.12.014. URL: https://www.sciencedirect.com/science/article/pii/S2095809919304928.

[17] Yongchao Zhou et al. *Large Language Models Are Human-Level Prompt Engineers*. 2023. arXiv: 2211.01910 [cs.LG]. URL: https://arxiv.org/abs/2211.01910.

[18] Yifan Yao et al. "A survey on large language model (LLM) security and privacy: The Good, The Bad, and The Ugly". In: *High-Confidence Computing* 4.2 (2024), p. 100211. ISSN: 2667-2952. DOI: https://doi.org/10.1016/j.hcc.2024.100211. URL: https://www.sciencedirect.com/science/article/pii/S266729522400014X.

[19] Laria Reynolds and Kyle McDonell. "Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm". In: *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI EA '21. Yokohama, Japan: Association for Computing Machinery, 2021. ISBN: 9781450380959. DOI: 10.1145/3411763.3451760. URL: https://doi.org/10.1145/3411763.3451760.

[20] Danny Goodman. *Dynamic HTML: The definitive reference: A comprehensive resource for HTML, CSS, DOM & JavaScript*. O'Reilly Media, Inc., 2002.

[21] T. Bray. *RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format*. USA, 2017.

[22] Felipe Pezoa et al. "Foundations of JSON Schema". In: *Proceedings of the 25th International Conference on World Wide Web*. WWW '16. Montréal, Québec, Canada: International World Wide Web Conferences Steering Committee, 2016, pp. 263–273. ISBN: 9781450341431. DOI: 10.1145/2872427.2883029. URL: https://doi.org/10.1145/2872427.2883029.

[23] Johann Mitlöhner et al. "Characteristics of Open Data CSV Files". In: *2016 2nd International Conference on Open and Big Data (OBD)*. 2016, pp. 72–79. DOI: 10.1109/OBD.2016.18.

[24] T. Berners-Lee, R. Fielding, and L. Masinter. *RFC 3986: Uniform Resource Identifier (URI): Generic Syntax*. USA, 2005.

[25] Cyril Goutte and Eric Gaussier. "A Probabilistic Interpretation of Precision, Recall and F-Score, with Implication for Evaluation". In: *Advances in Information Retrieval*. Ed. by David E. Losada and Juan M. Fernández-Luna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 345–359. ISBN: 978-3-540-31865-1.

[26] Jameleddine Hassine. "An LLM-based Approach to Recover Traceability Links between Security Requirements and Goal Models". In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. EASE '24. Salerno, Italy: Association for Computing Machinery, 2024, pp. 643–651. ISBN: 9798400717017. DOI: 10.1145/3661167.3661261. URL: https://doi.org/10.1145/3661167.3661261.

[27] Zhangyin Feng et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. 2020. arXiv: 2002.08155 [cs.CL]. URL: https://arxiv.org/abs/2002.08155.

[28] Talha Javed, Manzil e Maqsood, and Qaiser S. Durrani. "A study to investigate the impact of requirements instability on software defects". In: *SIGSOFT Softw. Eng. Notes* 29.3 (May 2004), pp. 1–7. ISSN: 0163-5948. DOI: 10.1145/986710.986727. URL: https://doi.org/10.1145/986710.986727.

[29] Adam Shostack. *Threat modeling: Designing for security*. John Wiley & Sons, 2014.

[30] IEEE SA. "IEEE Standard for Software Verification and Validation Plans". In: *IEEE Standards* (1986). IEEE 1012-1986.

[31] Alessandro Del Sole and Del Sole. *Visual Studio Code Distilled*. Springer, 2019.

[32] Claudio Curto et al. "Can a Llama Be a Watchdog? Exploring Llama 3 and Code Llama for Static Application Security Testing". In: *2024 IEEE International Conference on Cyber Security and Resilience (CSR)*. 2024, pp. 395–400. DOI: 10.1109/CSR61664.2024.10679444.