Tracing Dataflow Entities from Requirements to Software Architecture

Master's Thesis of

Hoang Hai Tran

At the KIT Department of Informatics
KASTEL – Institute of Information Security and Dependability

First examiner:	Prof. Dr.-Ing. Anne Koziolek
Second examiner:	Prof. Dr. Ralf Reussner
First advisor:	M.Sc. Sophie Corallo
Second advisor:	M.Sc. Dominik Fuchß

10. March 2025 – 10. September 2025

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Abstract

Traceability between requirements and software architecture models is essential for maintaining security-critical systems, as it ensures that implementations remain consistent with requirements and compliant with security constraints. Existing approaches such as SWATTR focus primarily on trace links to components, while dataflow-oriented requirements have not been considered so far. This thesis extends traceability approaches to security-related dataflows by leveraging large language models (LLMs) for entity extraction and trace link recovery, and by extending the underlying metamodel with dataflow representations. As a case study, requirements from the open-source EVerest framework for electric vehicle charging were annotated with SecLan element types and used to construct a gold standard. This dataset provides the evaluation basis and contributes a valuable resource, as labeled security-related requirements are otherwise scarce. The evaluation shows that GPT-4.1 can extract dataflow-related elements with promising accuracy. For components, a recall of 0.92 and an F2-score of 0.86 were achieved, clearly outperforming the SWATTR baseline (recall 0.80, F2-score 0.44). Across all dataflow-related element types, the extraction reached an overall recall of 0.78 and an F2-score of 0.73. For trace link recovery, GPT-4.1 surpassed SWATTR when provided with manually extracted elements. For tracing components, GPT-4.1 achieved a precision of 0.88 and a recall of 0.88, while SWATTR produced worse results (precision 0.77 and recall 0.73). At the same time, GPT-4.1 proved more sensitive to automatic extraction, while SWATTR was more robust under such conditions. In the final step, trace links were created for all annotated dataflows in the gold standard and additional valid flows were discovered beyond it. These findings indicate that LLM-based approaches can extend traceability beyond component-level links and provide a foundation to introduce tracing of dataflows. While challenges remain, especially regarding the reliable extraction of complete dataflows, the results demonstrate the potential of LLMs to complement heuristic approaches in creating detailed trace links for security-related requirements.

Zusammenfassung

Die Rückverfolgbarkeit zwischen Anforderungen und Softwarearchitekturmodellen ist eine wesentliche Grundlage für die Wartung sicherheitskritischer Systeme, da sie sicherstellt, dass Implementierungen konsistent mit den Anforderungen bleiben und Sicherheitsvorgaben einhalten. Bestehende Ansätze wie SWATTR konzentrieren sich in erster Linie auf Trace Links zu Komponenten, während datenflussorientierte Anforderungen bisher nicht berücksichtigt wurden. Diese Arbeit erweitert bestehende Traceability-Ansätze um sicherheitsrelevante Datenflüsse, indem Large Language Models (LLMs) für die Extraktion von Entitäten und die Erstellung von Trace Links eingesetzt sowie das zugrunde liegende Metamodell um Datenflussrepräsentationen ergänzt werden. Als Fallstudie wurden Anforderungen aus dem Open-Source-Framework EVerest für das Laden von Elektrofahrzeugen mit SecLan-Elementtypen annotiert und zur Erstellung eines Goldstandards genutzt. Dieses Datenset bildet die Grundlage der Evaluation und stellt zugleich eine wertvolle Ressource dar, da annotierte sicherheitsrelevante Anforderungen bislang nur in sehr geringem Umfang verfügbar sind. Die Evaluation zeigt, dass GPT-4.1 Datenfluss-bezogene Elemente mit vielversprechender Genauigkeit extrahieren kann. Für Komponenten wurde ein Recall von 0,92 und ein F2-Wert von 0,86 erreicht, womit der SWATTR-Baseline-Ansatz (Recall 0,80, F2-Wert 0,44) deutlich übertroffen wurde. Über alle Elementtypen hinweg erreichte die Extraktion einen Recall von 0,78 und einen F2-Wert von 0,73. Bei der Trace-Link-Erstellung übertraf GPT-4.1 SWATTR, wenn manuell extrahierte Elemente genutzt wurden. Für die Verknüpfung von Komponenten erzielte GPT-4.1 eine Präzision von 0,88 und einen Recall von 0,88, während SWATTR geringere Werte erreichte (Präzision 0,77, Recall 0,73). Gleichzeitig erwies sich GPT-4.1 bei automatisch extrahierten Eingaben als anfälliger, während SWATTR in diesen Fällen robuster war. Im letzten Schritt konnten Trace Links für alle im Goldstandard annotierten Datenflüsse erstellt werden. Zusätzlich wurden weitere gültige Flüsse entdeckt, die bisher nicht annotiert waren. Diese Ergebnisse zeigen, dass LLM-basierte Ansätze die Rückverfolgbarkeit über reine Komponenten-Verknüpfungen hinaus erweitern und eine Grundlage für die Einführung von Datenfluss-Tracing bieten können. Trotz bestehender Herausforderungen, insbesondere bei der zuverlässigen Extraktion vollständiger Datenflüsse, verdeutlichen die Resultate das Potenzial von LLMs, heuristische Ansätze zu ergänzen und detaillierte Trace Links für sicherheitsrelevante Anforderungen zu ermöglichen.

Contents

List of Figures

1 Introduction

Traceability between software artifacts plays a crucial role in software engineering because it enables developers to better understand relationships between requirements, design decisions, and implementation artifacts, leading to fewer errors and effort during system evolution. Traceability not only reduces the effort of maintenance activities such as change management, but also increases their accuracy. By making explicit the connections between artifacts, developers can more reliably identify the correct elements to modify [15]. Together, these benefits ensure consistency and alignment between requirements, design, and implementation artifacts [39].

Although traceability provides benefits across domains, security-critical domains benefit in particular, as trace links make it possible to check whether security requirements are properly addressed within a system. For example, a requirement may state that the system must provide secure communication with external services. With proper traceability, this requirement can be directly linked to the communication component in the software architecture model (SAM), and further to the corresponding implementations in the source code. This makes it easier for developers to check if the SAM or source code aligns with the requirement.
An example of a security-critical domain is electric vehicle (EV) charging, where systems involve security-sensitive interactions such as authentication and payment data exchange. For this, standards like `ISO 15118` and `OCPP`, that define handling of credentials and financial transactions are commonly used [27, 19]. Compliance with and usage of such standards are often described in the requirements of respective systems. Requirements like these typically affect several sub-systems, including the vehicle, the charging station, and backend services, and therefore require implementation across system boundaries. Establishing detailed trace links for such requirements is therefore particularly valuable, as it ensures that security constraints regarding data storage and dataflow are consistently upheld across the system and that modifications do not introduce vulnerabilities.

Ideally, traceability should be ubiquitous, meaning it should be built into the software engineering process [7]. Through this idea, tool-supported approaches are used to generate trace links automatically during the normal development process instead of manually creating trace links after development. The effort and cost of maintaining

trace links are largely reduced, since traceability information is continuously collected, analyzed, and presented to support relevant tasks. For example, a developer implementing a new user story can immediately access automatically captured trace data to understand its potential impact on related code, test cases, or requirements. Embedding traceability in the development process can minimize overhead, increase accuracy, and provide immediate benefits across projects, especially in security-related domains.
Several approaches have been proposed to automate the creation of trace links. Early work focused on information retrieval techniques, which calculate textual similarity between artifacts such as requirements and source code [7, 1]. Other traceability approaches like SWATTR also make use of heuristics to recover trace links from sentences to components [17, 16]. While effective in some contexts, these methods often struggle with inconsistent terminology across artifacts. More recently, pre-trained Large Language Models (LLMs) have been applied to TLR, showing promising results for linking requirements to goals, source code, and architectural components [12, 13, 9].
This thesis focuses on extending the heuristic trace link recovery approach, SWATTR, to incorporate security-related aspects. The goal is to support tracing sensitive dataflows to software architecture elements by making use of the existing framework. In order to do so, as a first step, dataflow entities have to be extracted from requirements. By leveraging pre-trained LLMs for named entity recognition [13], additional dataflow-related element types can be included without the need for hand-crafted heuristics. In a second step, the used metamodel is extended with elements representing dataflows, making it possible to create trace links between requirement-level dataflows and their counterparts in the software architecture model. On this basis, the connections between respective trace artifacts can be established with the support of LLMs, avoiding the need to rely solely on traditional heuristics.

This thesis aims to address the following research questions:

1. How accurately can dataflow entities from natural language requirement texts be extracted?
    a) How accurately can dataflow-related elements (components, data, entity, nodes) be extracted from requirements.
    b) How accurately can dataflows be extracted from requirements.
2. How accurately can trace links between data flow elements from requirements and architecture models be created?
    a) How accurately can dataflow-related elements be traced to architecture models?
    b) How accurately can dataflows be traced to architecture models?

By answering these research questions, this thesis aims to extend existing traceability approaches to incorporate security-related dataflows, thereby enhancing the automation and reliability of security requirement verification in software models.

As a case study, this thesis builds on the EVerest framework for electric vehicle charging. EVerest is an open-source platform that defines modular components for managing charging processes and backend communication. Building on prior work, security-related requirements have been elicited from EVerest, which provide the evaluation basis for the tasks described above. A gold standard was created for this purpose, which is also part of this thesis.

Chapter 2 introduces the foundations of this thesis, followed by Chapter 3, where related work on named entity extraction and trace link recovery is reviewed. Chapter 4 explains the creation of the gold standard and provides details on the annotated element types. Chapter 5 presents the approach developed in this work, starting with the extraction of elements and continuing with trace link generation and integration into SWATTR. Chapter 6 describes the evaluation of the different approaches, including comparisons with SWATTR. Finally, Chapter 7 summarizes the results and outlines possible future work.

2 Foundation

This chapter introduces the technical foundations required for the methods and evaluations in this thesis. Section 2.1 outlines trace link recovery with SWATTR. The Palladio Component Model is presented in Section 2.2, followed by dataflow analysis in Section 2.3. Section 2.4 introduces the EVerest framework, which serves as the case study. The SecLan system model is described in Section 2.5, providing the element types used for annotation. Section 2.6 summarizes large language models and prompt engineering, and Section 2.7 defines the evaluation metrics applied throughout this work. Together, these foundations provide the basis for the approach and evaluations presented in the following chapters.

2.1 Trace Link Recovery

Through previous work, the automated trace link generation framework formerly named Software Architecture Text Trace Link Recovery (SWATTR) has been developed, with Architecture Documentation Consistency (ArDoCo) acting as a collection containing different approaches, including SWATTR [17, 16]. In SWATTR, Trace Links are recovered from the architecture documentation and architecture model. Overall, the approach can be separated into different steps depicted in Figure 2.1. ArDoCo

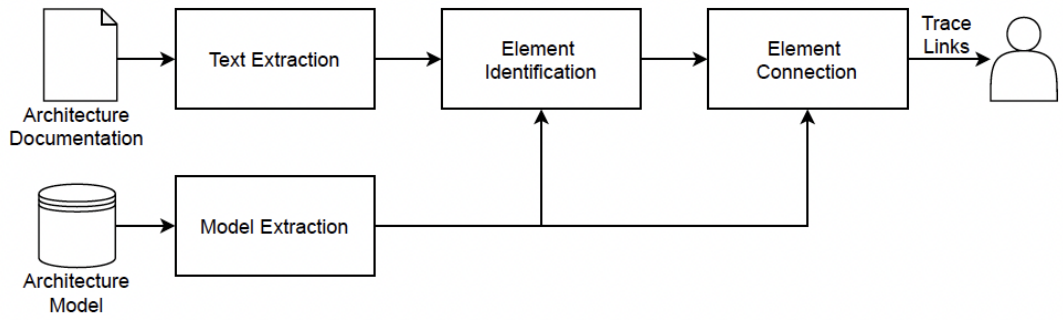

Figure 2.1: ArDoCo Trace Link Recovery Steps [17]

follows a structured, multi-stage process to establish trace links between architecture documentation and model elements. The key steps in this process include:

1. Text Extraction: The system processes architectural documentation to identify elements such as named components and types. Natural Language Processing (NLP) techniques are employed to extract and classify named elements or types of elements. In order to diminish the number of missed trace links in later stages, the classification aims for a higher recall.
2. Model Extraction: In parallel, the architecture model is analyzed to extract structured component information, including element names, types, and relationships. This structured data serves as the reference for creating trace links.
3. Element Identification: The extracted text and model elements are used to identify corresponding elements. Although the *Element Identification* step is independent of the actual model, it can access metamodel information, e.g., architecture element types. In this stage, also called *Recommendation Stage*, analyses are made to identify patterns accompanying element mentions like type-name and name-type. Informants can create *Recommended Instances* as potential matches, which will be later used to create trace links.
4. Trace Link Creation: Different trace links between documentation and model elements can be established once potential matches are identified. Comparing results from different agents makes calculating confidence for each trace link possible. When the confidence is high enough, a trace link is created.

Each step uses different agents that provide different analyses, based on information provided by informants. Most of the actual computation is done by the informants, that write their results into the shared data repository. The agent itself mainly provides the structure of the stage, defining which informants are executed and in what order.
Figure 2.2 illustrates this for the text extraction stage. The *InitialTextAgent* groups informants that analyze nouns, dependency arcs, and separated names in the documentation, with their findings written into the text state. The subsequent *PhraseAgent* builds on this state and uses its own informant to extract compound noun structures, generating noun mappings that extend the text state for further processing.
This modular design ensures that detailed analyses remain within the informants, while the agent serves as the integration point. As a result, extending the TLR approach to support new entity types, such as dataflow elements, can be achieved by adding new informants or replacing agents.

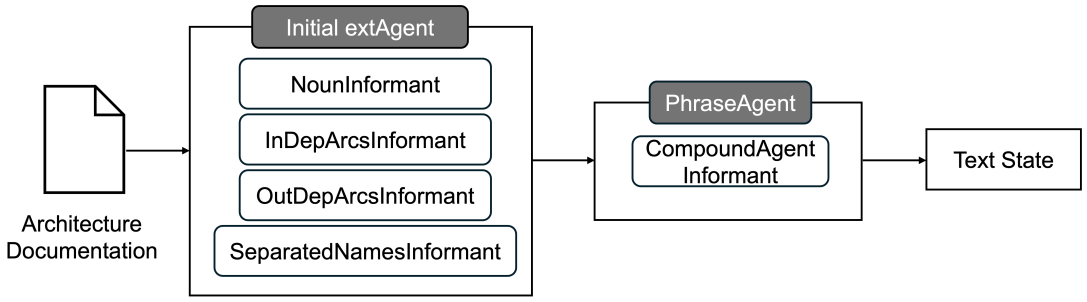

Figure 2.2: Text Extraction

2.2 Palladio Component Model (PCM)

The Palladio Component Model (PCM) is a domain-specific modeling language designed for early performance predictions of component-based software architectures [4, 35]. It allows software architects to assess response times, throughput, and resource utilization before implementation, helping to identify bottlenecks and optimize architecture. To do this, PCM provides different modeling capabilities offering different views on the system, like the component repositories, service effect specification, assembly diagrams, resource environments, allocation diagrams, and usage models.
In the component repository, different entities can be specified, like interfaces, data, types, and components. Part of the components are service effect specifications (SEFF). These make it possible to describe resource demands and calls to required services, provided by other components. In those calls, the input and output of the actions can be specified, making it possible to describe the state of input and output variables.
Different instances of the components can be composed into a system architecture. Those instances can be modeled in an assembly model. This model describes how the components are assembled by specifying which required interfaces are provided by which component. Through this, external calls to interface methods in the SEFFS can be delegated to specific components.
The usage model describes interactions with the system. To do so, different usage scenarios are modeled between actors and the system's provided services. These provide entry points of interaction with the system where parameters can be specified in the service calls. In order to describe this view on the system, the usage model references parts described in the assembly model and the repository.
In the resource environment, different component instances can be allocated to resources. Those resources can represent different physical locations. By combining these various model types, a wide range of performance and reliability analyses, including data flow analysis, can be supported.

2.3 Dataflow Analysis

Data Flow Analysis Framework (DFA) is an approach that enables automatic analysis of dataflow from architecture models, proposed by Boltz et al. [5]. During the DFA dataflows are extracted from the architecture models. These derived dataflows can then be used for analyses regarding different specified constraints that the system has to meet. An example of such a constraint is shown in Requirement 39 of the EVerest specification (see Figure 2.3). It states that sensitive information, such as tokens or payment data transferred to cloud systems, must not be written into logs stored on the charging station. This constraint can be described as a forbidden flow from the data entities *Token* or *Payment Info* to the node *Logs*. The DFA then can check whether the model align with the specified constraint.

Central part of the DFA is the Data Flow Diagram (DFD). The DFD metamodel that is used aligns with the unified DFD notation proposed by Seiferman et al. [37], which can be represented by transpose flow graphs (TFG). Each graph is made up of nodes, which are connected through flows of data. Characteristics of nodes and data are described by labels specified in the Data Dictionary, which can be referenced from the PCM models. The extraction of the flow graph from PCM models requires iteration through different models. In this process, references to elements from other model types have to be resolved. The entry point for each flow is the usage scenario in the usage model.

2.4 EVerest

EVerest [41] is an open-source framework for charging applications with loosely coupled modules. Different charging standards and usage scenarios can be supported by configuring the modules, depending on concrete needs. For example, a public charging station that provides plug and charge requires different modules for the authentication and protocols to communicate with the car, than a simple wall charger at home
The EVerest modules communicate with each other through the Message Queueing

Figure 2.3: Dataflow Constraint Requirement 39

Telemetry Transport (MQTT) protocol. Modules implement different parts that are necessary for EV charging, e.g., hardware drivers, protocols, and authentication. Additionally, EVerest specific tools are provided, like an admin panel to choose and configure used modules.
In order to support as many usage scenarios as possible, the number of modules is relatively high, with 29 modules. These modules are modeled in PCM as components, containing 143 SEFFs in total. Prior to this work, a first modeling effort was done, as other model types besides the component repository have been created, in an effort to model the EVerest system. The models have been created based on available requirements and documentation, and from reverse-engineering the source code.
As part of the master's thesis of Marettek [26], 93 design-level security requirements have been elicited as a result of questionnaires and interviews with EVerest software developers. Based on these requirements, as part of a practical course, a dataflow analysis for EVerest has been conducted. Out of 93 requirements, 19 held descriptions relevant to the dataflow analysis regarding constraints for the dataflows. These constraints are described in natural language in the requirements and had to be formalized using a Domain Specific Language (DSL) to check for dataflow violations. Different patterns were identified that generalize the different constraints, which helps with the reusability of the formalization. Those patterns contain different elements. For example, specific data with status, such as sensitivity, shall not flow to specific locations or components. This means that requirements may also describe dataflows that are not represented in PCM, but rather a state that should not occur. Also, some might require specific actors to participate in order to be authorized. One pattern that describes that data with a specific status should not flow to a defined set of components can be seen in Figure 2.4.

In general, the dataflow constraints can be described by different elements of the requirement text, such as data, actors, location, status of data and status of location.

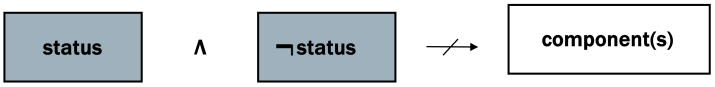

Figure 2.4: Dataflow constraint pattern

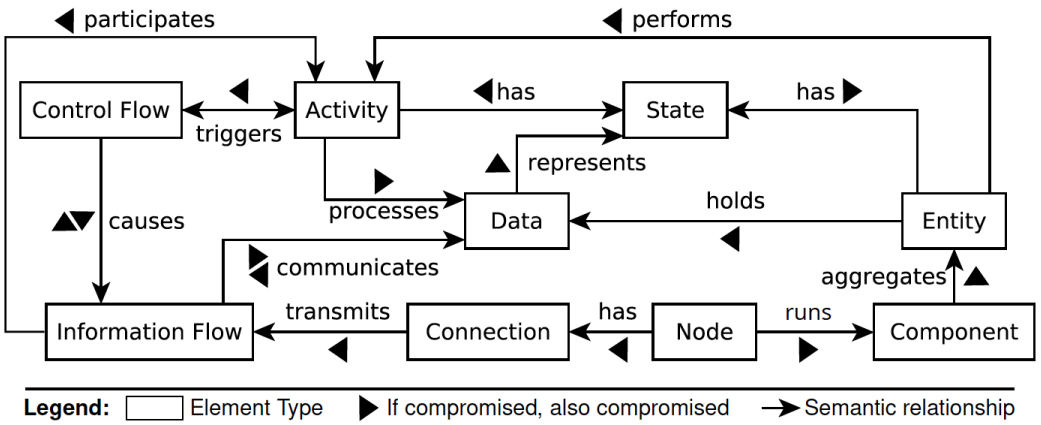

Figure 2.5: SecLan system model [32]

2.5 SecLan Model

The SecLan model is a conceptual framework with the purpose of bridging the gap between security design and implementation [32]. Through SecLan, it is possible to understand and manage relationships between implementation-level security in the form of code and security design, which is mostly specified through Domain-Specific Languages (DSL).

At its core, the SecLan model introduces a meta-model consisting of elements, which make up software systems such as *components*, *entities*, *nodes*, and *data*. This *system model* is shown in Figure 2.5, containing each element type and their respective relationships between each other. *Components* represent functional building blocks that can run on *nodes*, while *entities* represent actors or software objects, which are aggregated by *components*. *Data* elements represent information that is exchanged in the system. By explicitly modeling these elements and their interactions, for example as part of an *information flow*, SecLan provides a foundation for understanding dataflows. In addition, SecLan supports the modeling of *states* and *activities*, which allow the specification of the conditions under which dataflows occur. In the *system model* it is defined, that *states* are represented by *data* values.

Other sub-model that are part of the SecLan model are the *security model*, *SecDSL Description*, and *SecAnalyzer Description*. The *security model* describes three fundamental security concepts, *security objectives*, *threats*, and *weaknesses*. The *SecDSL Description* contains descriptions of common elements of security DSLs. The *SecAnalyzer Description* sub-model describes the security checks, which are possible through static analyzers.

Part of the description covers the purpose of each check and the weaknesses it is supposed to detect. The SecLan model itself defines how these concepts relate to each other.

2.6 Large Language Models

Large Language Models (LLMs) are advanced AI systems designed to process and generate human-like text [28]. LLMs have evolved from traditional statistical methods to sophisticated neural network-based architectures. This progression includes pre-trained language models (PLMs) and large-scale LLMs [28]. Their fast development is driven by transformer architectures, enhanced computational power, and vast training datasets [14]. Nowadays, transformer-based LLMs are used in various use cases of natural language processing, including information retrieval tasks, chatbots, and coding assistants [14].
Early models like T5 and GPT-3 have shown the ability to perform tasks without fine-tuning, introducing the concepts of zero-shot and few-shot learning [6]. These models undergo pre-training on massive datasets using self-supervised learning, in order to efficiently process natural text sequences. Their performance can then be further refined through fine-tuning on task-specific data, which is particularly relevant for narrowly defined tasks. At the same time, LLMs show their ability for generalization across tasks of different domains, excelling in applications involving natural language text [28]. Prompts are used as input for the most commonly used LLMs. In practice, prompts are used as primary input, which function as natural language instructions. The design of such prompts plays an important role to reach the desired output. This leads to different systematic design approaches referred to as *prompt engineering*.

Sahoo et al. present various prompt engineering techniques, which provide a structured approach to guiding LLMs toward desired outputs [36]. Rather than modifying the model's core parameters, prompt engineering enhances accuracy by adjusting the instruction text. The choice of technique depends on the specific use case, with methods ranging from zero-shot and few-shot prompting to chain-of-thought prompting and beyond.

A possible task where different prompting strategies can be applied would be classifying whether a requirement is a security requirement.

The *Zero-shot* prompt is built as the base-prompt, where the LLM is asked directly:

Prompt 1: Zero-Shot Prompt
Classify the following requirement as a security requirement or not
Requirement: “The system must log all failed login attempts.”
Answer with ’yes’ or ’no’

In a *few-shot* setting, additional examples are provided to guide the model. Those examples establish the expected result format and additional context, which can improve the accuracy of the model compared to *zero-shot* prompts [38].

Prompt 2: Few-Shot Prompt
Example 1: “The system must encrypt user passwords.”
Answer: yes

Example 2: “The application should support multiple languages.”
Answer: no

Classify the following requirement as a security requirement or not:
Requirement: “The system must log all failed login attempts.”
Answer with ’yes’ or ’no’

Finally, this can be extended with *chain-of-thought prompting*, where the model is encouraged to provide reasoning steps before answering.

Prompt 3: Chain-of-Thought Prompt
Classify the following requirement as a security requirement or not.
Let’s think step for step. Provide a short reason, on how you came to that conclusion

Requirement: “The system must log all failed login attempts.”

Answer with reasoning + ’yes’ or ’no’

The results achieved vary depending on the combination of prompting techniques and the chosen model, and it is difficult to define a single optimal configuration that performs similarly across all tasks [38]. Given the wide range of available techniques and their possible combinations, effective prompt design often depends on the individual developer. This highlights the need for a more systematic approach to crafting effective prompts, ensuring consistency and efficiency in optimizing LLM performance.

DSPY introduces a systematic approach to prompt generation by programmatically specifying the given input, desired output, and task requirements [18]. This streamlines the prompt engineering process and provides a good baseline, which can be adjusted depending on desired results. Since traditional text-based prompts can yield varying results depending on the input data or the specific LLM used, DSPY offers a tool-based solution that enhances generalization across different LLM pipelines. By systematically applying various prompt engineering techniques based on the task at hand, DSPY improves consistency, adaptability, and efficiency in leveraging LLMs for diverse applications. DSPY also provides optimization of prompts when sample input data and desired results are provided. This process refines the prompt structure to enhance model performance while reducing the reliance on manual prompt design.

2.7 Evaluation Metrics

Standard metrics are applied to evaluate the classification task performance in this thesis. Commonly used standard metrics are applied to evaluate the performance of the classification tasks in this thesis. This enables comparability with different variants used in this work and with other work. The central measures are *precision*, *recall*, and their combinations into F1 and F2.

complemented by both micro- and macro-averaged variants. These metrics allow for a balanced assessment across classes, which is crucial in the presence of class imbalance that is typical for requirement texts.

Precision and Recall

TP denotes the number of true positives, FP the false positives, and FN the false negatives. Precision P and Recall R are defined as:

$$P = \frac{TP}{TP + FP}, \tag{2.1}$$

$$R = \frac{TP}{TP + FN}. \tag{2.2}$$

The precision value measures the proportion of correctly identified items compared to all positive identified items. On the other hand, Recall measures the proportion of correctly identified items among all relevant items, including the number of items missed out of the FP. In trace link generation tasks, recall is particularly important, since

it measures how many of the relevant links have been retrieved. In practice, filtering out irrelevant trace links is preferable to manually searching for missing ones.

F-Measure

To combine Precision and Recall into a single score, the F_β-measure is used, defined as:

$$F_\beta = (1 + \beta^2) \cdot \frac{P \cdot R}{\beta^2 \cdot P + R}. \tag{2.3}$$

Commonly used is the harmonic mean F_1, where $\beta = 1$:

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R}. \tag{2.4}$$

For tasks where recall shall be weighted more than precision, β can be increased. In this thesis, the F_2 score is also reported ($\beta = 2$):

$$F_2 = 5 \cdot \frac{P \cdot R}{4 \cdot P + R}. \tag{2.5}$$

This formulation weights recall twice as high as precision, which reflects the importance of identifying as many relevant entities and links as possible.

Micro and Macro Averaging

Two aggregation strategies can be applied to measure performance across different runs or across element types: *micro-averaging* and *macro-averaging*.

Micro-averaging: Metrics are calculated globally by summing over all classes:

$$P_{\text{micro}} = \frac{\sum_i TP_i}{\sum_i (TP_i + FP_i)}, \quad R_{\text{micro}} = \frac{\sum_i TP_i}{\sum_i (TP_i + FN_i)}. \tag{2.6}$$

Micro-averaged F then becomes:

$$F_{\beta,\text{micro}} = (1 + \beta^2) \cdot \frac{P_{\text{micro}} \cdot R_{\text{micro}}}{\beta^2 \cdot P_{\text{micro}} + R_{\text{micro}}}. \tag{2.7}$$

Results of the micro-averaged measures are dominated by frequent classes and reflects overall correctness across the dataset.

Macro-averaging: Metrics are computed per class and then averaged:

$$P_{\text{macro}} = \frac{1}{N}\sum_{i=1}^{N} P_i, \quad R_{\text{macro}} = \frac{1}{N}\sum_{i=1}^{N} R_i, \tag{2.8}$$

$$F_{\beta,\text{macro}} = \frac{1}{N}\sum_{i=1}^{N} F_{\beta,i}. \tag{2.9}$$

Here N denotes the number of types or folds. This results in macro-averaging, weighting all types or runs equally.

3 Related Works

In this chapter, the research related to trace link recovery (TLR) of dataflows from requirements is discussed. The first part reviews existing approaches for *named entity recognition* (NER), as this thesis uses dataflow-related entity extraction as the base for creating tracelinks to dataflows. The second part discusses approaches for *trace link recovery*, contrasting traditional information retrieval techniques with recent advances based on large language models (LLMs). This chapter gives an overview of the state of the art and highlights the research gap addressed in this work.

3.1 Named Entity Recognition in Requirements

The literature review by Kolahdouz-Rahimi et al.[20] shows that heuristic NLP methods are the most commonly used for requirements formalization. Maltempo et al. [25], for example, attempt to extract multi-word named entities from requirement texts using NLP techniques and heuristic rules to derive a hierarchical model. Their approach relies on processing kernel sentences, which must follow a simplified sentence format. However, since natural language requirements vary depending on the author, system, or organization, this thesis aims to classify the elements within these requirements without requiring prior structuring. This ensures transferability of the results to other projects, which most commonly utilize natural language requirements.
Pakhale [31] provides a general overview of *named entity recognition* (NER) approaches, where different methods are presented ranging from early rule-based systems to modern transformer-based architectures, including domain-specific models such as ViBERTgrid and BioBERT. While ViBERTgrid [23] addresses specific challenges of financial and legal documents, BioBERT [22] specializes in retrieving information from biomedical language. Similar to these approaches, Ray et al. [40] propose an approach for the NER in aerospace requirements, aeroBERT-NER. Through the fine-tuning of BERT, a specialized model is created that can handle aerospace-specific terminology. This required a annotated aerospace corpus, containing 1432 requirements.
While specialized models achieve superior performance compared to LLMs that are fine-tuned for general-domain tasks [40], their adaptation to specific domains often requires extensive fine-tuning or hybrid methods, which is difficult to implement when only

limited annotated data is available. This directly relates to this thesis, as security-related requirement engineering is also a domain with scarcely labeled data.

The work of Malik et al. [24] focuses on requirements documents and proposes a supervised learning approach for extracting requirement-specific entities. They define ten entity categories that are contained in software specification texts (e.g., *API*, *GUI*, *HARDWARE*, *PLATFORM*). As the supervised-learning approaches used require annotated data, requirements had to be manually annotated with the defined entities as labels. Those requirements were collected from DOORS, a requirement management tool created by IBM. In this process, more than 3000 sentences were extracted from the requirements and annotated. Using models such as ML-CRF, C-MEM, and BiLSTM-CRF, they demonstrate that traditional machine learning methods can capture recurring patterns in requirements texts. However, their defined entity set differs significantly from the SecLan elements used in this thesis, and the approach relies on a large manually annotated dataset, which is not available for the SecLan entities. This highlights the need for alternative approaches that can operate effectively without extensive labeled data.

In contrast, Marettek proposes an approach for NER in her master's thesis [26], where general-use pre-trained models are utilized. Her approach classifies different SecLan elements from elicitated EVerest requirements using GPT. The results indicate that fine-tuning GPT 3.5 turbo leads to more accurate classification compared to prompt engineering with GPT 4, suggesting that fine-tuned models are better suited for precise extraction tasks. For the evaluation, a 5-fold cross-validation was performed, where four folds were used for training and one for evaluation. In addition, ten percent of the training data was withheld from fine-tuning, resulting in approximately 67 training samples per fold. Previous work, such as Oliver et al. [29], suggests that fine-tuning becomes more reliable with around 200 labeled examples. While fine-tuning on a smaller dataset can still yield acceptable results, it increases the risk of reducing generalization ability, since the limited training data has a high influence on the model. Consequently, the transferability of the results to other projects cannot be guaranteed. For this reason, the focus of this work lies on zero- to few-shot prompting, utilizing chain-of-thought prompting for entity classification in natural language requirements.

3.2 Trace Link Recovery

Trace Link Recovery (TLR) aims to identify and maintain relationships between different software artifacts [7]. Before the emergence of large language models (LLMs), most semi-automated approaches were based on Information Retrieval (IR) techniques [7]. Classical IR methods such as Vector Space Models (VSM) [1], Latent Semantic Indexing

(LSI) [2], or Latent Dirichlet Allocation (LDA) [3] rely on textual similarity and were widely applied to link requirements with design or code artifacts. LSI and LDA are extensions of VSM, where dependencies between terms and documents are considered, to represent latent structures. TAROT [10] proposes the use of consensual biterms extracted from requirement texts. These biterms can be used to refine the calculation of IR values, improving the recovery of trace links between software artifacts. While classical IR-based approaches can produce effective results in generic requirement-to-code TLR, they are limited due to possible term mismatches through different software artifacts [7]. To bridge this gap, the usage of LLMs could be beneficial in order to find links between elements that do not have consistent naming.

Recent research therefore has explored LLMs for trace link recovery for various software artifacts. Hassine [12] proposes an approach to recover trace links between security requirements and Goal models. GPT-3.5 turbo was used to create links between requirements and security-related goals. This method was evaluated on a dataset of 42 requirements for a virtual interior designer application and achieved promising results, with an F_1 score of 0.879.

To address the limited input size of LLMs, which makes it infeasible to provide the entire project context, Hey et al. [13] propose the usage of Retrieval-Augmented Generation (RAG) to retrieve the most likely candidate links. Their work embeds requirements into a vector representation. For each source requirement, the most similar target requirements are retrieved, and the requirement pairs are sent to a LLM for link classification. The best results were achieved with chain-of-thought prompting using GPT-4o. This approach can also be extended to a broader range of artifacts, including source code, requirements, architecture documentation, and architecture models. Fuchs et al. [9] transform these artifacts into textual representations first, which can be used for RAG. Evaluation of the SAD to SAM TLR showed that chain-of-thought prompting outperformed the keep-it-simple-stupid (KISS) strategy. The KISS prompt is similar to a simple zero-shot prompt. It could also be observed that the average performance of the classification decreased when multiple features were combined, for example, when including interface and usage information in addition to component names. While the approach was effective, the baseline approach ArDoCo was not outperformed, achieving $F_1 = 0.458$ and $F_2 = 0.589$.

LLMs can be used in various use cases in the TLR context, like Fuchs et al. [8] proposed the usage of LLMs to create a simplified SAM to bridge the gap between source code and SAD. The LLM is used to extract component names from SAD. This approach produced similar results to state-of-the-art TLR approaches using manually created SAM and outperforms TLR approaches that don't require SAM. In this work closed source LLM from OpenAi performed better than Llama-based models.

These works show the possibilities in automated trace link recovery using LLMs. However, existing work primarily focuses on linking requirements to other textual artifacts (e.g., other requirements, source code, or goals) or on extracting structural entities such as component names, while software architecture document to software architecture model TLR is mostly focused on components. Since components are part of dataflow entities, existing TLR approaches like SWATTR provide a valuable foundation for the tracing of dataflows. This thesis aims to apply LLM-based methods to identify and link additional dataflow-related elements besides components described in requirements to their corresponding architectural model elements.

4 Gold Standard Creation

A reliable gold standard dataset is needed to evaluate named entity extraction of dataflow entities from requirements and for the recovery of trace links. Without such a dataset, the quality of extracted entities or the correctness of generated trace links cannot be measured in a meaningful way, ensuring transferability to other projects. This labeled data is also useful for training purposes, since some approaches in named entitity recognition benefit from annotated examples.
A suitable dataset for the use in this thesis has to fulfill three aspects. First, it must be aligned with an existing software architecture model (SAM), so that trace links can later be evaluated against a concrete target. Second, the underlying requirements need to include security-related topics, especially dataflows, since these are central to the focus of this thesis. Third, the entities labeled in the requirements have to match the SecLan element types, so that the dataset is consistent with the conceptual model introduced in section 2.5.
As there is a lack of publicly available labeled datasets that would match the needs of this thesis, such a gold standard has to be created beforehand. The EVerest project is chosen as the basis for creating the gold standard. It provides a set of elicited requirements that explicitly include security-related aspects, which is valuable since such requirements are usually not made publicly available. These have been elicited together with Pionix developers as part of Marettek's master's thesis [26]. In addition, a PCM model of EVerest has been created, with a focus on describing dataflows of the system. This makes EVerest a suitable candidate, as it combines the availability of security-related requirements with a SAM that can be used as a target for trace link evaluation. Additionally, since EVerest is an open source project, the source code is available. This not only enables future extensions of TLR towards the implementation level, but also supports further research that requires access to security-related context beyond requirements and architecture.
The gold standard was created by the author of this thesis, together with two other annotators, in order to reach an inter-annotator agreement. For this, the requirements must be labeled separately, and differences must be discussed and resolved. Inter-annotator agreements are supposed to increase the overall quality of the gold standard, handle ambiguities in labeling, and create a more objective labeled data set.

4.1 Annotation Process

The annotation process followed a defined guideline to ensure consistency across annotators. The main objective was to identify entities in natural language requirements that correspond to SecLan elements.

A key concept in the annotation is the *acceptance window*, defined by a *long sequence* and a *short sequence*. The *long sequence* represents the longest acceptable term associated with a label. Those terms may include additional information or articles describing the element. The corresponding long sequence is labeled again if the same element occurs multiple times within a requirement.
A *short sequence* refers to the minimal required term that still contains enough information to understand the meaning of the labeled element. As an example in the requirement text "This can be ensured by the EvseSecurity module and by" (ID 28), a component was labeled with the *long sequence* "the EvseSecurity module" and *short sequence* "EvseSecurity".
Another concept used is *references* and *coreferences*. References are used when the text does not explicitly mention a concrete element but refers to it in a more general way, often by grouping multiple elements together. For instance, phrases like "the system" or "all critical modules" are annotated as references, since they describe several components without naming them individually. Coreferences are interchangeable names or pronouns in the text that refer back to previously mentioned components. They must point to either references or short sequences of components, such as in “these modules” or when using an acronym after the full name was introduced.

The labeling was done by three annotators with computer science backgrounds: one Bachelor student, one Master student, and one PhD researcher. The initial annotation of the requirements was performed independently by the annotators. Afterwards, all three annotators met to review the results, discuss disagreements, and resolve ambiguities. In cases where not all three annotators reached the same conclusion, even after reviewing the annotation guideline, a majority vote decided how the disagreement should be solved. Through this process of inter-annotator agreement, a consolidated version of the dataset was created, which leads to the final gold standard used in this thesis.

4.2 Annotation of SecLan Elements

The annotation guideline used is based on the definitions of SecLan elements (cf. section 2.5). During annotation, several ambiguities and corner cases were discovered, which were addressed by extending the definitions with concrete examples. In the following, the annotation procedure and additional fields for each element type are

described in detail, as part of the extended annotation guideline. For each element type, the original definition as proposed by Peldszus et al. [32] is presented, followed by the extensions to the definitions made during the annotation process, the annotation fields used, and an example.

Data – SecLan - Definition

"A central element type in software security is *Data*, which conveys information through a collection of values." [32]

The *data* definition is extended to include composed data objects such as personal information, addresses, or credit card data. Certificates are also explicitly added, since they occur in EVerest requirements and are particularly relevant in security contexts. Each data element is annotated with a long and a short sequence. The long sequence covers the full span of the requirement text, while the short sequence reduces it to the minimal meaningful term. Both sequences together form the acceptance window for evaluation, which ensures that different phrasings are handled consistently.

Running Example – Data Annotations

"The EvseManager component on the charging station must verify authentication tokens provided by the CSMS, log invalid tokens in the system log, and forward valid tokens to the Auth module for further processing."

Legend: long sequence short sequence

Figure 4.1: Example with annotated data labels

Entity – SecLan - Definition

"Data can be held by Entities, which can be a physical actor, software object, or external system such as a database." [32].

The definition of *entity* was expanded to explicitly include software libraries, protocols, external systems, and physical actors. **Entities** were annotated whenever the text referred to elements that can hold states or expose interfaces, but are not independently deployable architectural units.
This distinction ensured that protocols were annotated as entities (e.g., "OCPP protocol"), while modules implementing these protocols remained annotated as components (e.g., "OCPP module"). Similarly, libraries such as "libocpp" or "libevse" were treated as entities rather than components.

In the example shown in Figure 4.2, the phrase “CSMS” (Charging Station Management System) was annotated as an entity, since it represents an external system that interacts with internal components. Here, the long sequence was “the CSMS”, while the short sequence was simply “CSMS”. This complements the component annotations "ChargingManager" and "Auth module", and together they form the basis for the annotated information flows.

Running Example – Entity Annotations

“The EvseManager component on the charging station must verify authentication tokens provided by the CSMS, log invalid tokens in the system log, and forward valid tokens to the Auth module for further processing.”

Legend: long sequence short sequence

Figure 4.2: Example with annotated *entity* labels

Activity – SecLan - Definition

"For realizing a system’s behavior, an *Entity* performs an *Activity* which processes *Data* or communicates with other *Activities*" [32].

Activities may also be realized as manual actions carried out by system actors such as users, described in behavioral models, or implemented as functions at the source code level. *Activities* were extended to cover manual user actions and software-related functions. Updates and described functions were explicitly included as valid activities, even if not tied to a concrete implementation. In addition, statements about ownership relations of data or entities (e.g., “the module stores certificates”) were also annotated as activities.

Not every relation in the text was considered an activity. Statements that only express compliance, fulfillment of requirements, or general design claims (e.g., “complies to ISO 15118”, “fulfills all requirements from …”) were not annotated as activities, since they describe design context rather than an executable action. Similarly, ambiguous formulations such as “This affects system parts …” were only annotated if a concrete activity could be identified.

Each annotated activity containes additional fields to capture its context. These fields describe the *actor* performing the activity, the *activity* itself, the *objects* involved, further *additions*, possible *negations*, references to corresponding *model identifiers*, and optional *notes*. This ensured that activities were annotated not only as textual spans but also as structured elements that can be mapped to architecture models.

Special care was required when handling negations (e.g., “must not forward tokens”). In such cases, the negation was explicitly stored in the activity annotation. In contrast, statements about the absence or presence of elements (e.g., “the charger has no certificate”) were considered states rather than activities. This distinction was refined during the annotation process.

In the running example in Figure 4.3, besides *“verify”*, the verbs *“log”* and *“forward”* can also be annotated as activities. These fields don’t have their own acceptance window, as they refer to other annotated elements. For example the *object* "authentication tokens" refers to the *data* element in as seen in Figure 4.1.

Running Example – Activity Annotations

“The EvseManager component on the charging station must verify authentication tokens provided by the CSMS, log invalid tokens in the system log, and forward valid tokens to the Auth module for further processing.”

Legend: Actor Activity Object Addition

Figure 4.3: Example with annotated *activity* labels and color-coded fields

State – Definition

"*Entities* and *Activities* can have a *State* which is generally represented by *Data*." [32]

For *state*, the guideline was refined to also include the state of *data* and *node*. In addition, technical execution states (e.g., a crashed server) and specific configurations were considered valid states, provided they were explicitly mentioned in the requirement text.
States are generally understood as attributes of an object or as roles assigned to an actor. They can further describe properties of elements and provide additional attributes or circumstances that specify them in more depth. States can also describe restrictions or limitations, for example by narrowing the validity of data, entities, or activities.

Not every descriptive relation was considered a state. Statements describing software design or architectural relations to standards (e.g., “A implements B”, “C is included in process D”, “E is defined in ISO 15118”) were not annotated as states, since they capture design details rather than runtime conditions.
States restrict the interpretation of elements and are essential for determining the

system's behavior. Each state has to refer to an *object*, that can be an *activity*, *entity*, *data*, *component* or *node*. In the running example illustrated in Figure 4.4, the adjectives *valid* and *invalid* describe the data object *tokens* and were therefore annotated as *state*. The states always refer to the *short sequence* of a object.

Running Example – State Annotations

"The EvseManager component on the charging station must verify authentication tokens provided by the CSMS, log invalid tokens in the system log, and forward valid tokens to the Auth module for further processing."

Legend: Object State

Figure 4.4: Example with annotated *state* labels

Control Flow – Definition

"The execution of *Activities* is orchestrated by *Control Flow*."[32]

A typical pattern for control flows is a conditional statement such as "if X happens, Y must be executed". Only cases with a direct relation between initiator (*source*) and recipient (*target*) were annotated.

Each annotated control flow contains fields for the *action* verb, the *source* initiating the flow, and the *target* receiving it. Both *source* and *target* could be *components*, *entities*, or *nodes*. In addition, *references* were annotated in cases where the text described a control flow but did not explicitly mention either source or target (e.g., "in this case, processing must stop"). *Coreferences* were also considered, for example when a subsequent sentence referred back to an already annotated control flow with phrases such as "this action".

In the running example illustrated in Figure 4.5, the *source* "EvseManager" validates tokens and forwards them to the *target* "Auth module" for "further processing" (*action*). Similar to the *state*, the acceptance window for the different fields is defined by the acceptance windows of the prior labeled elements.

Information Flow – Definition

"When *Data* is exchanged between *Activities* (indirectly also between *Entities*, i.e., receiving data can be seen as an *Activity*), this can form an *Information Flow*" [32].

Running Example – Control Flow Annotations

"The EvseManager component on the charging station must verify authentication tokens provided by the CSMS, log invalid tokens in the system log, and forward valid tokens to the Auth module for further processing."

Legend: Source Target Action

Figure 4.5: Example with annotated *control flow* relation

The guideline was extended to define, that storing or persisting data within the same component is separated from *information Flows*, unless the data was explicitly transmitted to another element. This extension ensures that information flows are distinguished from data states. Purely descriptive access control information, such as stating that a component has permission to read or write, was not annotated as an information flow, since no concrete transfer of data was described. Furthermore, at least one type of data had to be explicitly mentioned in order for a *information flow* to be annotated.

The labeling of contained *information flows* was an important factor in the creation of the gold standard, as these were needed for the later evaluation. Each annotated flow contains additional fields to describe the *source*, *target*, the transmitted *data*, and the form of *transmission*.

In the running example, two information flows can be annotated. The first flow, illustrated in Figure 4.6, originates from the entity "CSMS" (*source*) and is directed to the component "EvseManager" (*target*), transmitting the data element "authentication tokens". The second flow goes from the component "EvseManager" (*source*) to the component "Auth module" (*target*), transmitting the data element "valid tokens". The verbs "provided" and "forward" were annotated as the form of transmission.
The acceptance window for dataflow is defined by the acceptance windows of the prior labeled elements.

Component – SecLan - Definition

"Multiple software Entities can be combined to a Component that encapsulates specific functionality."[32].

For *component*, the guideline is extended to clarify that architectural elements such as subsystems, frameworks, or architectural layers, can also be annotated as *components*, as long as they can be independently deployed.

Running Example – Information Flow Annotations

"The EvseManager component on the charging station must verify authentication tokens provided by the CSMS, log invalid tokens in the system log, and forward valid tokens to the Auth module for further processing."

Legend: Source Target Data Transmission

Figure 4.6: Example with annotated *information flow* relations

Mentions that do not explicitly refer to a component are marked as references, which is also the case when multiple components are addressed collectively without naming them separately. If both a general term and specific component names are provided, only the specific mentions are annotated, while the common term is ignored. References to protocols (e.g., *OCPP protocol*) are not mapped to components. Similarly, libraries such as *libocpp* or *libevse* are excluded, since they do not represent deployable units. Only system-internal components are annotated, while external systems such as the MQTT broker, trusted platform module, or CSMS are annotated as entities.

Running Example – Component Annotations

"The EvseManager component on the charging station must verify authentication tokens provided by the CSMS, log invalid tokens in the system log, and forward valid tokens to the Auth module for further processing."

Legend: long sequence short sequence

Node - SecLan – Definition

"A *Component* can be deployed on a *Node*, which is a physical device executing software."[32]

In contrast to *components*, which represent logical software units, *nodes* represent the underlying physical infrastructure. Annotating nodes is important to capture the system deployment view and to connect software architecture with its execution environment. *Nodes* include both complete physical devices and their physical interfaces. For example, "ethernet ports" was annotated as a *node*. This illustrates that not only large devices like servers but also ports and network interfaces are considered *nodes* when explicitly mentioned in the requirements.

As with other categories, both *long sequences* and *short sequences* are annotated to identify the acceptance window of a node. As illustrated in the Figure 4.7, the "charging station" would be a *node*, as the *component* "EvseManager" is probably deployed on it.

Running Example – Node Annotations

"The EvseManager component on the charging station must verify authentication tokens provided by the CSMS, log invalid tokens in the system log, and forward valid tokens to the Auth module for further processing."

Legend: long sequence short sequence

Figure 4.7: Example with annotated *node* label

Connection - SecLan – Definition

"Communication between *Activities* or *Entities* is established either through physical *Connections* between *Nodes* or internally within a *component*." [32]

Each annotated connection contained additional fields to describe its role in the system. The *source* identifies the *node*, *entity*, or *component* that initiates the *connection*, while the *target* refers to the element receiving the connection. The field *via* captures the physical connector or medium over which the source and target are linked (e.g., ethernet port, USB, local socket). This structured annotation ensures that *connections* can be consistently aligned with both *nodes* and *information flows*.

Although explicit mentions of connections were relatively rare in the EVerest requirements, they play an important role in linking nodes and supporting the traceability of information flows. Figure 4.8 shows an example of a sentence with a connection. In this case, the *node* "charging station" is annotated as the *source* and the *entity* "CSMS" is annotated as the *target*. The annotated elements were traced to an existing EVerest software architecture model, if possible. For this, the element names were manually compared to elements of the PCM to find respective *ModelIDs*. In the PCM component repository, we looked for *components* and *data* model elements, while for *nodes*, the resource environment was searched. Supposedly, a module is mentioned in the requirements, but only its interfaces exist in the architecture model. In that case, it was still traced to the closest component associated with the interface.

Example – Connection Annotations

"The charging station maintains active connections to the CSMS in order to exchange charging session data."

Legend: Source Target via

Figure 4.8: Example with annotated *connection* relation

4.3 Resulting Gold Standard

The annotation results for each SecLan element type are merged for the individual requirements, so that for each requirement, a single JSON file exists that contains all annotated element types in one place. The resulting dataset represents the gold standard used in this thesis.

The overall JSON structure is organized into two parts:

- **Requirement metadata**, which stores attributes such as requirement identifier, text, author, security objective, and a confidence score. These were taken from the original elicited requirements [26].
- **Element annotations**, which are grouped by SecLan element types (e.g., component, data, state). Each group contains the specific text sections, where the elements are described, as well as additional properties needed to describe the element as specified in section 4.2.

The resulting gold standard dataset contains labeled SecLan elements for 93 requirements of the EVerest project. The dataset contains 1261 annotated elements across all types: component, data, state, activity, entity, node, information flow, control flow, and connection. Table 4.1 shows the distribution of annotations per category. While most of the *components* could be mapped to a specific architecture element (86 out of 97), this task was more difficult for *data*, as data elements are often more abstract and their names often don't match a specific datatype, making them hard to trace. The high number of annotated states can be explained by their role in describing nearly every other element type. States are used to qualify or restrict components, entities, activities, and data elements, whereas only control flows and information flows are typically not associated with explicit states. As a result, states occur in almost every requirement, making them by far the most frequent annotation category in the dataset.
In total, 197 annotations were marked as *references*. These are cases where the requirement text did not explicitly mention a concrete element but only referred to it

Element Type	Total	With ModelId	References
Components	97	86	93
Data	67	37	34
Nodes	38	22	5
Entities	109	21	18
States	589	-	6
Connections	34	-	2
Dataflows	36	-	15
Activities	229	-	10
ControlFlows	62	-	14
Total	**1261**	126	197

Table 4.1: Distribution of annotated elements in the gold standard.

indirectly (e.g., “the system”, “these modules”). References often occur for components, which shows that architecture is often described in unspecific terms. The presence of references shows the inaccuracies that come with natural language requirements. Since references cannot be mapped directly to architectural elements, they require contextual interpretation.

Table 4.2 shows in how many of the 93 requirements each element type are labeled. States appear in 90 requirements, reflecting their importance in describing conditions for data and activities. Activities are annotated in 84 requirements, while components occur in 52 requirements. In contrast, connections and dataflows are much less frequent, as explicit descriptions of communication channels or complete end-to-end flows are rare. Dataflows that were fully specified with *target*, *source*, and *data* were relatively rare (18), as often either *target* or *source* was not explicitly mentioned in the requirement.

Element Type	Requirements with at least 1 Annotation
Components	52
Data	34
Nodes	27
Entities	57
States	90
Connections	20
Dataflows	24
Activities	84
ControlFlows	34

Table 4.2: Coverage of annotated elements across requirements.

The resulting gold standard contains annotations for all SecLan element types and therefore provides a good base for named entity recognition. The number of explicitly annotated dataflows is relatively low, which can make classification for this type more difficult. At the same time, many components and data elements could already be linked to the architecture model through model identifiers, which at least enables the evaluation and training of trace link recovery for these types.

4.4 Threats to Validity

Several threats to validity have to be considered for the created gold standard. First, a potential bias in labeling may arise from prior work with the EVerest project, since the annotators already had expectations regarding existing components. This prior knowledge could have influenced labeling decisions, although the effect was mitigated by using multiple annotators and resolving disagreements through discussion.

Second, mapping the annotated elements to the SAM could only be performed on a best-effort basis. In many cases, no model identifier could be assigned to an annotation. This can either occur because a corresponding model element does not exist, or because the element exists in the model but was not explicitly recognized during annotation. As a result, traceability coverage may be incomplete. On the other hand, this limitation also creates an opportunity to detect inconsistencies in the SAM, e.g., not modeled software architecture elements.

Third, a degree of subjectivity remains, particularly for the annotation of *short sequences*. Even though the acceptance window reduced this risk, annotators occasionally differed in their judgment of the minimal sufficient span. Similarly, the initial annotation guideline was not fully specified in all aspects, which required clarifications and iterative refinements of the guideline during annotation. While this increased consistency over time, it still reflects a potential source of variation in early annotations. The limited number of annotators (one Bachelor student, one Master student, and one PhD researcher) also constrains the diversity of perspectives. Although this group provided a useful balance of experience, it still leaves the possibility that alternative interpretations may have been overlooked.

Finally, the representativity of the annotated classes is limited by the dataset itself. The EVerest requirements were elicited in a real-world project and so naturally reflect this system. Consequently, some SecLan element types are more frequently represented than others, which affects the balance of the dataset. Moreover, the strong focus on a single project domain limits the external validity of the gold standard. The evaluation

results derived from this dataset cannot be directly generalized to other domains of requirements engineering without further validation on different datasets.

5 Extraction and Tracing of Dataflow entities from Requirements

Requirements often describe how data is exchanged between different elements of a system, for example, between components or physical devices. Such descriptions offer an opportunity to extend traceability beyond structural elements and include dataflows as trace artifacts. Since requirements can contain descriptions of how data is transmitted between system elements, they provide an opportunity to extend traceability to dataflows. This chapter introduces the approach for classifying dataflow entities in requirements, which forms the foundation for establishing trace links between software architecture documents (SAD) and SAM. The classification was integrated into the existing SWATTR approach (cf. section 2.1). This chapter describes which entities are classified for dataflow traceability, the strategy used to perform this classification, and the extensions required to integrate it into the existing SWATTR framework. Furthermore, it outlines the role of large language models (LLMs) in the classification process and compares different prompting strategies to evaluate their effectiveness.

5.1 Traceable Dataflow Entities

Dataflow entities are combinations of other entities: nodes that share data across edges along them. For the dataflow analysis on PCM (cf. section 2.3), dataflows are represented as dataflow graphs. As dataflows have no explicit representation in PCM, they have to be derived from usage scenarios, system behavior captured in SEFFs and deployment information. In PCM, each action, such as external calls between components, start and end nodes, or internal data processing, is mapped to a vertex in the graph Boltz et al. [5]. The incoming and outgoing pins of these vertices correspond to the transmitted data types. The dataflows derived from PCM contain *components* as processing and interaction units, *nodes* as the execution context for these *components*, *entities* representing external participants, protocols, or actors, and *data* as the information exchanged along the edges. The *nodes* are derived from the resource environment model, which describes the deployment of components on physical nodes(c.f. section 2.2). These make it possible to group components by their execution context. This is particularly relevant

when dataflow constraints describe dataflows at the level of physical devices rather than individual software components. In the following, the term *dataflow-related elements* is used for components, nodes, entities, and data. For tracing these, we use their respective SecLan elements from the gold standard. Another element type from SecLan that is relevant are information flows. These information flows could describe small subsections of the dataflow graph in PCM, consisting of a minimal graph between a source and a target, which exchange data. For this thesis, instead of the term information flow, dataflow is used, referring to the Seclan element. Besides these elements, SecLan also describes other elements that occur in security-related requirements. However, not all are relevant for dataflows, and thus, they are neglected in the tracing approach. For example, SecLan *states* are not further used, although they are partly represented in dataflow graphs through annotated characteristics (e.g., confidentiality labels or encryption states). Although *states* provide additional information about underlying elements, like *data* or *components*. They are not used as trace artifacts, as they are not modeled separately from the software model artifacts that they describe. Similarly, SecLan elements like *activities*, or *control flows* are also excluded.
In summary, while the SecLan Model describes an extensive set of entity types that can be annotated in security-related requirements, not all of these are equally relevant for dataflow trace link recovery.

The selection of dataflow-related elements, which can be traced to PCM architecture elements, is illustrated in the example Figure 5.1. Several elements can be classified and traced in this requirement. *Components* are the "EvseManager" and the "Auth module", which are both represented as components in the EVerest repository model. The *entity* "CSMS", which represents an external actor, is modeled in the EVerest PCM as a resource container. The mentioned *node* is the "charging station", providing the deployment context, which can also be traced to a resource container. Finally, the *data* exchanged are the "authentication tokens", which can be matched to the "Token" datatype, contained in the repository model. The adjectives "valid" or "invalid" correspond to the *state* of the tokens. Although they can be described in SEFFS by setting the state of a variable, they can not be used as trace artifacts by themselves. Rather, they further describe the token, but they don't identify it, making it part of the element, like by "authentication". Similarly, the verb "forwards" describes an *activity*, which explains system behavior but is not required for structural trace links. The dataflow between "EvseManager" and the "Auth module", sending "authentication tokens", can be traced by its contained elements, as each of the elements can be traced by itself.

Example – Dataflow-related Elements

"The EvseManager component on the charging station must verify authentication tokens provided by the CSMS, log invalid tokens in the system log, and forward valid tokens to the Auth module for further processing."

Legend: Entity Component Data Node

Figure 5.1: Example with annotated dataflow-related elements

5.2 Entity Extraction

The following section describes the approach for the extraction of dataflow entities from requirements. Through this approach, this thesis aims to replace the in SWATTR used heuristics focused on the extraction of components from architecture documents, with a method to extract dataflow-related entities. For this step, LLMs are utilized to extract phrases and classify them by element type. This approach makes use of the semantic understanding of LLMs to detect dataflow-related elements in natural language requirements without requiring intermediate preprocessing steps. The extraction of entities is done for each requirement separately, which means that the LLM does not carry knowledge of other requirements. When giving out extraction results, the model has to adhere to a predefined format to make further processing possible through parsing. This format is determined by the classification mode used.

Two classification modes are implemented and compared to each other. Named entity recognition from natural language text can be approached as a set of independent classification tasks or as a single joint task where all entity types are identified. Both alternatives offer different advantages, so evaluating them side by side provides insights into which strategy is better suited for the extraction of dataflow entities.

Single classification: Each dataflow-related element type (e.g., components, data, node) is classified individually in separate runs, through their respective informants. This allows more fine-grained control, through individual prompts and potentially better results per type, as the LLM only has to focus on one dataflow-related element type. Similarly, dataflows are extracted in a separate run. As the *dataflow related elements* are extracted separately, the LLM does not have to return the type of each extracted phrase, but a simple JSON array as output is sufficient. This can be seen for the extraction of components in Figure 5.2. The extracted dataflows are represented as JSON

Requirement

"The EvseManager component on the charging station must verify authentication tokens provided by the CSMS, log invalid tokens in the system log, and forward valid tokens to the Auth module for further processing."

Output: ["EvseManager", "Auth"]

Figure 5.2: Single class annotation for component

objects. As illustrated in Figure 5.3, each object contains the fields "source", "target", and "data", which describe the origin of the flow, its destination, and the transferred data.

Joint classification: All *dataflow related element* types are extracted in a single run. This allows the LLM to consider the relationships between the element types, which could help to classify them correctly. In this mode, dataflows are extracted as a follow-up question based on previously identified entities. This approach is expected to increase precision since the LLM will classify one extracted phrase with only one type. In the single classification approach, the same phrase contained in one requirement might be extracted as a component from one informant and as an entity by another informant.

The expected result, as shown in Figure 5.4, is formatted as a JSON object containing the extracted elements and their types. The dataflow extracted afterwards follows the format defined in the entity-wise classification. For both classification modes, prompt design plays a major role in reaching desired results from the LLM. The `dspy` [18] framework was used to create the initial prompt. The prompt created through DSPY

Requirement

"The EvseManager component on the charging station must verify authentication tokens provided by the CSMS, log invalid tokens in the system log, and forward valid tokens to the Auth module for further processing."

Source Data Target

Output:
[{"*src*":"EvseManager", "*target*":"Auth", "*data*": ["authentication token"]}]

Figure 5.3: Annotation for dataflow

Requirement

"The EvseManager component on the charging station must verify authentication tokens provided by the CSMS, log invalid tokens in the system log, and forward valid tokens to the Auth module for further processing."

Component Data Entity Node

Output:
[{"*element*":"EvseManager", "*type*":"Component"}, ...]

Figure 5.4: Joint class annotation for component

was only used as a template, and further optimization was not performed through the framework. For this, DSPY already provided a prompt that could be used especially for named entity recognition tasks. As a foundation, a zero-shot prompt was created, consisting of a system message and a user message. In the system message, the task and the expected output are defined. Although it is not necessary to use the system message, as they are not treated much differently from user messages [33], the structure was still used to organize the interactions and not have everything in the user message itself. The user message contains the required text and the extraction instructions. Early tests showed that by using the initial prompt as shown in Prompt 5.2, the used LLM had no problem adhering to the defined format. Regardless, the results for the dataflow entity extraction, produced by the LLM, were not satisfactory, which suggested that the task had to be further defined. For this purpose, the prompt was extended by type definitions to clarify disambiguities and define the dataflow entities and possible relationships between them. The additional definitions, shown in Prompt 5.2, were added to the system prompt. The definitions were taken from the annotation guideline to provide the prompt with descriptions that are not specifically tailored to the Everest project. In order to incorporate chain-of-thought prompting, the LLM is supposed to produce *reasoning* for the extraction results, as displayed in Prompt 5.2. The *reasoning* is not used afterwards, and only the predicted elements are collected for later trace link creation. Based on the created prompt, few-shot prompting is applied, where the model is provided examples in addition to the requirement text. In this work, the examples are taken from the gold standard. The requirement text is put into a user message, while the expected return value of the LLM is put into an assistant message. These messages can be chained to each other, depending on how many examples should be provided. To add examples to the prompt before the extraction runs, the prompt has to be built, except for the last user message.

Prompt 4: System Prompt

```
Your input:
 ‘requirement_text‘ (str): technical requirement
Your output:
 ‘results‘ (list[result]): extracted elements with {reference, type}
Format:

[[ ## requirement_text ## ]]
{requirement_text}

[[ ## reasoning ## ]]
{reasoning}

[[ ## results ## ]]
[
  {"reference": "...", "type": "..."}
]

[[ ## completed ## ]]
Objective: Given ‘requirement_text‘, produce ‘reasoning‘, ‘results‘.
```

Figure 5.5: System Prompt Joint classification

Prompt 5: System Prompt

```
Possible entity types:
 **Data**: values or information (e.g., attributes)
 **Component**: deployable software unit (e.g., database, framework)
 **Entity**: actor or object holding data (e.g., user, protocol)
 **Node**: physical or virtual device
```

Figure 5.6: System Prompt element type definition addition

5.3 Trace link recovery

The existing approach is extended to incorporate the TLR of dataflow entities in SWATTR. The existing core for tracing architecture components to natural language

sentences can be reused in this process. Based on the entity extraction approach (cf. section 5.2), which provides SWATTR with extracted dataflow entities for further linking, the next steps towards trace link recovery for dataflow entities include the meta model extension and the connection generation.

The metamodel of PCM used in SWATTR is a simplified submodel of PCM and does not fully capture all software architecture element types, as only components and interfaces are parsed from the repository model. To represent the dataflow-related elements, details from the PCM are added to the existing metamodel. This ensures that additional architecture items are added to the pool of traceable units. Each item is represented by its name and unique modelId, which are later on used for the connection generation. This work added datatypes, resource containers, usage scenarios, assembly contexts, and SEFFS to the metamodel (cf. section 2.2). While datatypes and SEFFs were already part of the repository, resource environment, assembly model, and usage model had to be added as input. From these architectural items, datatypes and resource containers can be directly traced to the dataflow-related elements, data and node (cf. section 5.1). Through this extension, it is possible to trace each dataflow-related element: component to PCM component, data to PCM datatype, nodes and entities to PCM resource container. By tracing these it is already possible to trace dataflows through its contained elements source, target and data, as the dataflow-related elements represent these elements. For future usage, the metamodel was also extended to represent simplified dataflow graphs represented in PCM. The dataflows contained in the PCM model itself were extracted similarly to the dataflow graph extraction (c.f. section 2.3), but are represented in the metamodel as a simplified version, which should be sufficient for TLR purposes. The usage model, assembly model, and SEFFs contained in components have to be parsed for this purpose. The usage model provides the entry points for the data flow extraction, from there on, references are resolved through the model to build a dataflow graph. Instead of a graph, the participating components and parameters are collected in a dataflow object, in order to enable tracing to the dataflow graph through its contained elements. To illustrate the process of the dataflow extraction from PCM, the following example is taken from the EVerest PCM model. The usage scenario "Reserve Charger" contains an `EntryLevelSystemCall`, which points to both a provided role in the assembly model and an operation signature in the repository. From there, the assembly context refers to the encapsulated component "EvseManager", and the SEFF corresponding to the referenced operation signature can be identified. Inside the SEFF, an `ExternalCallAction` is found that calls another service and passes the variable `reservation` as input. The called service is then resolved back through the assembly model to the providing component "Authentication", whose SEFF may again contain further external calls. This process is repeated until no further calls are found and the flow terminates in a component without outgoing calls. From this, the already

existing components "EvseManager" and "Authentication" and the data "reservation" can be collected and stored in a dataflow object, for later tracing.

With the metamodel now representing all architecture items that shall be traced, the next step for the TLR is the connection generation. The purpose of the connection generation is to find trace links from architecture items to extracted elements from requirements. By utilizing LLM for this, no additional heuristics have to be implemented for the new element types that are supposed to be traced (data, node, entity). For this task, the LLM is provided with the extracted element from the requirement and the architecture model context. Based on this information, the model should return the correct architecture items that can be linked to the extracted phrase. This architecture model information is provided to the LLM through the prompt. As providing the PCM as a whole to the prompt is not an option, due to token limitations and possible diminishing performances, only relevant parts of the model should be sent to the LLM. This selection is made through the types of elements that were extracted. For elements of the type components, all PCM component architecture items are selected. Similarly, the PCM datatypes are selected for data, and the PCM resource containers for nodes and entities. These architecture items are then parsed to JSON objects containing the name and modelId of the architecture item. The list of JSON objects is then added to the user message, together with the extracted element. This is illustrated together with the expected output in Prompt 5.3. Tracing dataflows relies on the previously traced dataflow-related elements and the extracted dataflow from the requirement. In order for this to work, the extracted dataflow-related elements have to be traced first. If the dataflow is traceable, then a subset of those traced dataflow-related elements is contained in the dataflow as source, target or data. Through this, some of the extracted dataflows are filtered out if the source or the target could not be traced.

5.4 Integration into SWATTR

The extension of SWATTR for dataflow entity classification is built on the existing modular pipeline but skips the text extraction stage. In the original architecture shown in Figure 2.1, requirements are first preprocessed by multiple informants that extract nouns, compound terms, and dependency structures. These results are then passed as *text state* to the *recommendation stage*, where potential element candidates are created. In this work, this first stage was skipped. Instead, the requirement texts are directly forwarded to the *recommendation stage* without prior preprocessing. The reasoning behind this decision is that LLMs can already process raw text input effectively, making separate noun and dependency parsing redundant. This reduces complexity while leveraging the contextual understanding of LLMs.

Prompt 6: User Prompt

```
[[ ## extracted_element ## ]]
{EvseManager}
[[ ## model_elements## ]]
[{ModelElement:"EvseManager", ModelId:"VtwOsJBbEe6OM"},...]
Respond with the corresponding output fields,
starting with the field ‘[[ ## reasoning ## ]]‘,
then ‘[[ ## tracelink ## ]]‘
```

Output:
{"extractedElement":"EvseManager", "ModelElement":"EvseManager",
"ModelId":"VtwOsJBbEe6OM"}

Figure 5.7: Example element connection

A new *LlmRecommendationAgent* was introduced, which orchestrates several LLM-based informants. These informants replace the classical NLP-based informants and generate `RecommendedInstances` directly from the raw requirement texts.
The following informants were implemented: `LlmComponentInformant`, which detects mentions of components in the requirement text and classifies them accordingly. `LlmDataflowInformant`, which identifies dataflow relationships and their roles (e.g., source, target). `LlmDatatypeInformant`, which classifies datatype mentions. `LlmNerInformant`, which performs a combined named-entity recognition for multiple elements such as components, data, nodes, entities, and dataflows.
Each informant produces `RecommendedInstances`, which are written into the `RecommendationState`. This state collects all identified entities in a format compatible with the existing pipeline. Subsequent steps in trace link creation can operate on the new entity types in the same way as they would on components. For simple elements such as components or nodes, standard `NounMappings` are sufficient. For dataflows, however, the underlying noun mappings were extended with a type field to distinguish `SOURCE`, `TARGET`, and `DATA`. This is realized through the subclass `TypedMapping`, which allows multiple mappings to be grouped into a single `RecommendedInstance` of the type `DATAFLOW`. In this way, dataflows can be represented, while maintaining compatibility with the existing `RecommendationState`.

Each informant communicates with an LLM via the `ChatLanguageModelProvider`. This utility class provides the setup of chat-based models and supports different platforms (e.g., OpenAI, Ollama). It manages platform selection, model name, and configuration parameters such as temperature, and builds the corresponding model instance using

environment variables for authentication. Through this, the *informants* remain independent of a specific LLM implementation while supporting later extension for additional platforms.

The platform (`ChatPlatform`) and model variant (`ChatModelName`) are configurable within each informant. The configuration is passed through the pipeline from the *runner* to the *agent*, and finally to the individual *informants*. This design enables seamless experimentation with different LLM providers and models without having to make changes to the individual informants.

Prompts are configured at the *runner* level and stored in the `DataRepository`. Each informant retrieves its task-specific prompt from the repository and combines it with the requirement text to construct structured chat messages. This ensures that prompts are managed centrally, allowing them to be reused, versioned, or replaced for comparative experiments.

The construction of messages and the parsing of results regarding LLMs are handled by the `LlmUtility` class. This helper provides static methods to format prompts as `SystemMessage`, `UserMessage`, and `AiMessage` objects, and to extract structured results from model outputs. Depending on the task, the results are parsed using regular expressions or JSON and mapped to `NounMappings` or `TypedMappings`, which are then written into the `RecommendationState`.

By delegating model instantiation to the `ChatLanguageModelProvider`, prompt management to the `DataRepository`, and prompt/response handling to the `LlmUtility`, the informants themselves remain lightweight. Their task is reduced to orchestrating the flow of data: retrieving prompts, constructing messages, invoking the LLM, and writing the parsed results into `RecommendedInstances`. This separation of concerns ensures modularity and reusability across different LLMs, prompt strategies, and extraction tasks.

For the extension of the metamodel a new PcmExtendedExtractor has been created, to parse additional model elements to ArchitectureItems. While Components were extended by SEFF descriptions, datatypes, and resource containers were added as architecture items as well. Additionally, a DataflowExtractor was added to propagate through the PCM and find modeled dataflow graphs. This extractor was not used for tracing the architecture items yet, and provides an opportunity for future work, to incorporate traceability to dataflow graphs. The architecture items, that were added as Endpoints are PCM Components, PCM Datatypes, PCM Resource Containers and PCM Interfaces. The PcmExtendedExtractor is used in the ModelExtraction phase, which stores the extracted architecture items in the shared datarepository for later stages.

For the last step of the trace link recovery pipeline, the *Element Connection* stage was extended to utilize LLMs. A new *LlmConnectionAgent* and *LlmConnectionInformant* were implemented. Similar to the *RecommendationStage*, the *LlmConnectionInformant* relies on the `ChatLanguageModelProvider` to interact with a configured model. For each dataflow-related element, one request is sent to the LLM, providing the extracted elements together with candidate architecture items. The model output is then parsed, and the results are added as trace links. For dataflows themselves, the tracing process depends on the individual dataflow-related elements. Once both source and target of a dataflow have been traced successfully, a new trace link is created by aggregating the trace links of source and target, thereby establishing the trace link for the dataflow as a whole.

6 Evaluation

This chapter concerns itself with the evaluation of trace link recovery (TLR) for dataflow-related elements. As an intermediate evaluation, the extraction for those elements from natural language requirements can be assessed in order to address the first research question. The goal is to determine to what extent large language models (LLMs) can replace heuristics in SWATTR and improve the coverage and precision of dataflow-oriented traceability. Also part of the evaluation is the assessment of different classification strategies, prompt strategies, and models in order to find an optimal configuration. By utilizing the EVerest gold standard (cf. chapter 4) and the software architecture model, the evaluation aims to provide findings that are transferable to other projects and domains. The following sections introduce the evaluation techniques and results. First, the *extraction of dataflow-related elements*, *entities*) from requirements is evaluated (*eval1*). Based on these results, the *extraction of dataflows from requirements* is assessed (eval2). To address the second research question, the *trace link recovery of dataflow-related elements* is evaluated (eval3). Lastly, the results of *the trace link recovery of dataflows* are assessed (eval4) to address the second part of the research question.

6.1 Evaluation Methods

For the evaluation, a 5-fold-cross-validation was used. Through this method, the gold standard is shuffled and separated into five folds of similar size. The resulting folds can each be used separately for evaluation, while the remaining four folds are used for training. As there are 93 requirements, the fold sizes cannot be equal. Instead, five folds with 18 requirements each were created. The remaining three requirements were each assigned randomly to one fold, resulting in two folds with 18 requirements and three with 19. The difference in training pool size can be ignored, as only a small amount is used for the LLMs. For each fold, a separate evaluation is done, counting the true positives (TP), false negatives (FN), and false positives (FP), to calculate the precision, recall, F1-score, and F2-score. These values can be used to determine the macro and micro averages.

For each fold, the training data of the remaining folds are used for the construction of few-shot prompts. For few-shot prompts, it can be commonly observed that the

performance of the used model increases initially with the amount of examples provided. However, after a certain number of examples, the performance decreases [21]. This means that not every training sample available should be added to the prompt. For a first evaluation to assess the effect of few-shot prompting, four examples are added to the prompt. First, a random selection is tested, and then a selection is made to create a representative example set, with a limited number of examples used. For such a selection, different strategies were applied.

One strategy used is the deterministic selection of few-shot examples. In the gold standard, each requirement is represented by its set of annotated entity types and by the tokens from its requirement text. Textual similarity between requirements is estimated using the Jaccard similarity. The Jaccard similarity measures the overlap between two sets A and B and is defined as

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

with values from 0 for no overlap and 1 for identical sets. For this evaluation, A and B represent the token sets of two requirements. As the similarity measures are ranked, an additional threshold does not have to be defined. This measure is used to ensure diversity in the chosen examples. The evaluations use three different modes to select few-shot examples: joint-class mode, single-class positive mode, single-class balanced mode, or random mode.

Random mode The training pool is shuffled, and four requirements are taken from it by creating a subset of the collection. For the shuffling a seed is set, for reproducibility. This mode serves as a simple baseline strategy, without explicitly considering balance or diversity.

Joint-class mode The goal in this mode is to select four examples that cover all relevant element types at least once. The selection proceeds in two phases:

1. *Coverage phase:* Starting with an empty set S of selected requirements and uncovered types $U = T$, where T is the set of all element types, requirements are then added iteratively using a greedy set cover strategy. In each step, the requirements in the training data are compared to each other, based on the number of uncovered types they contain. The requirement with the highest gain is added to S, and the covered types are removed from U. This is repeated until all types are covered or $|S| = 4$.
2. *Diversity phase:* If fewer than four examples have been selected after coverage, the remaining examples are filled with requirements that are most dissimilar to

the already chosen examples, ensuring variety in the selected set. For this, the Jaccard similarity is used.

Single-class positive mode In this mode, only a single element type t (e.g., component) is considered. The pool of examples is restricted to the ones containing at least one annotation of t. From this pool, four requirements are chosen to be as diverse as possible, again using Jaccard similarity. This mode provides the model with multiple varied examples of the same type.

Single-class balanced mode A single target type t is considered, but a balanced set of positive and negative examples is selected. The pool is split into positives (requirements containing t) and negatives (requirements without t). From each subset, two examples are selected. The first one is chosen randomly, and the second by selecting the most dissimilar to the already chosen requirement, utilizing the Jaccard similarity. The two subsets are then combined into the final example set. This ensures that the model observes positive and negative cases with equal frequency.

If several requirements achieve the same gain, the one covering the rarest types in the dataset is preferred. This approach ensures that the selection of examples not only covers all element types but also is reproducible and diverse.

LLM selection

For the evaluation, two different Large Language Models (LLMs) were used: GPT and LLaMA. Both models are transformer-based pre-trained LLMs that can process natural language input and generate structured output. They received identical inputs (system message, few-shot examples, requirement text) and were instructed to output the same JSON format for extracted entities. As a result, GPT and LLaMA could be used interchangeably in the recommendation stage without modifications to the surrounding pipeline. While GPT was accessed through the OpenAI API, LLaMA was hosted via *Ollama*.
GPT was selected for this thesis as it represents the state-of-art of proprietary LLMs, and consistently achieves high performances across different tasks [30]. LLaMA, in contrast, was chosen as an open-source alternative that can be hosted locally. This is especially valuable for security-related topics, as no confidential data has to be shared externally. In addition, local deployment avoids reliance on proprietary APIs and offers a more cost-efficient setup. Through this selection, both ends of the current LLM models can be covered: closed-source, high-performance models on one hand, and open-source, efficient models on the other hand.

This thesis aims to make use of the latest GPT model available. At the time of writing, this would be GPT-5, which represents the latest iteration of the GPT family. However, the current API of GPT-5 does not allow controlling the *temperature* parameter. *Temperature* is a parameter that is used to configure the randomness of a model. Johns et.al [34] suggest to use a temperature of 0.0 to achieve optimal results in problem solving tasks. Their study showed, that when a higher temperature was used, most mistakes made where caused by the model not adhering to the defined output format. This phenomena could also be observed in the few test runs, where GPT-5 was used. Not only did it sometimes deviate from the defined output schema, but it could also be observed a considerable number of times, where terms were extracted which were not part of the provided requirement. Since deterministic and reproducible outputs in strict JSON format are essential for tasks such as entity extraction and recommendation generation, GPT-5 was not used in this work.
Instead, GPT-4.1 was selected as the primary model used, as it is the latest iteration of the GPT-4 line (released in April 2025) and provides strong performance while supporting temperature adjustment [30]. In addition, the smaller variant GPT-4.1-mini was used to evaluate how performance changes when a less resource-intensive model is used.

For LLaMA, the LLaMA 3 8B model was selected. The LLaMA 3 family, released by Meta in 2024, includes models of different sizes [11]. The 8B variant was chosen as it still achieves competitive results on standard benchmarks [11], although it is the smallest model.
The ability to host it locally makes it the choice for experimentation in research contexts where reproducibility and limited computational resources are important factors.
Larger LLaMA variants, such as the 70B and 405B models, achieve higher benchmark scores but require substantially more resources. By selecting LLaMA 3 8B, the evaluation simulates the scenario where results must be obtained under limited computational budgets, complementing GPT as a high-end proprietary variant.

6.2 Element Extraction

This section addresses the evaluation of the first research question. This research question can be separated into the classification of dataflow-related elements and the classification of dataflows themselves. This distinction is possible because dataflows are conceptually different from the elements they contain. Element types like *nodes*, *entities*, *components*, and *data* exist independently and can be classified separately. A *dataflow*, in contrast, is defined as a relation between these elements, assigning roles such as *source*, *target*, and *data*. Therefore, its correct identification requires not only

extracting the individual elements but also capturing their roles within the flow, making it a much more abstract classification task.

To evaluate the *extraction of dataflow-related element* (eval1), the results that the LLMs provide for the extraction are checked against the gold standard. The acceptance windows defined in the gold standard can be used for the matching. The results are assessed for each dataflow-related element type separately. A TP is counted if a predicted phrase and its assigned element type match an entry of the same type in the gold standard. Such a match is achieved if the phrase lies within the boundaries of the acceptance window. This means that it has to contain the short sequence but should not exceed the long sequence annotated. An FN is counted when a gold standard entry exists, but no corresponding prediction matches the acceptance window criterion. In this case, the element was present in the gold standard but was missed by the model. An FP is counted when a predicted phrase with its assigned type does not match any corresponding entry in the gold standard for that requirement. This includes cases where either the phrase boundaries lie outside the acceptance window or the predicted type does not correspond to the annotated type. True negatives can not be counted in named entity recognition tasks, as the absence of a prediction can not be mapped to the absence of a gold standard entry. They are also unnecessary to calculate the metrics precision, recall, F1-score, and F2-score (c.f. section 2.7). The macro averages are calculated for each element type across the folds, determining the stability of the used approach across the different folds. The micro averages are calculated for each element type separately to observe the overall performance for each type. Additionally, the macro averages across all element types are determined, in order to give a performance measure regardless of type representation in the requirement pool. For this, the TP, FN, and FP for each element type across all folds are collected. For the evaluation different configurations of prompting strategies and models are tested. The *joint-classification* (JC) and *single-classification* (SC) modes are evaluated against each other, using zero-shot prompting. For both modes, the effects of few-shot prompting are tested. For the few-shot examples, different selections of the training pool are evaluated. This means for SC mode, few-shot examples containing only entries with the respective element types (*Positive*) and a balanced training pool with two examples containing the element type and two examples without (*Balanced*). The best configuration is then tested with different models.

The results that the LLM provides regarding extracted dataflows are assessed for the evaluation of the *extraction of dataflows* (eval2) from requirements. As described in the gold standard, a dataflow contains *source*, *target*, *data*, and *transmission*. The *transmission* does not have to be identified, as it is not used for the tracing. A TP for the extraction of dataflows is counted when the LLM recognizes all three values correctly. The *source* and *target* in the dataflow refer to short sequences, references,

or coreferences of the element types component, node, and entity. Similarly, the *data* value refers to an entry in the data annotations. Through this, it is possible to utilize the acceptance window of each element type to assess a match. An FN is counted when no predicted dataflow matches an entry in the gold standard. Partial matches are also counted as FN, for example, when only two of the three values are predicted correctly. A predicted dataflow is determined a FP, if such a dataflow does not exist in the gold standard. In this strict matching, a nearly correct dataflow leads to an FP and an FN. Similar to eval1, the precision, recall, F1-score, F2-score, and their micro and macro averages are calculated for the dataflow element. For the dataflow extraction zero-shot and few-shot prompts are tested against each other. Also evaluated is the effect of providing the extracted dataflow-related elements as additional information on the extraction of dataflows.

6.3 Trace link recovery

The *trace link recovery of dataflow-related elements* (eval3) is evaluated based on correct trace links created. All recovered trace links are compared to an entry in the gold standard. A trace link is represented in the gold standard through the annotated *modelID* of a labeled element. For each requirement, the predicted trace links are compared to the trace links that should be contained. A TP is counted when the extracted trace links matches a gold standard entry. This means that the extracted term has to match the acceptance window of an element, and the modelIDs have to be equivalent. An FN is counted when the requirement holds an entry for a labeled element, where a modelID is annotated, but such a trace link has not been extracted. FP may indicate a missing annotation in the gold standard. These cases where a predicted trace link does not match the modelID and acceptance window of any element type are counted as FP. Based on these values, the macro-averaged precision, recall, F1, and F2 can be calculated for each element type across all folds. The macro average across all element types is also determined to evaluate TLR performance, considering less represented element types. First, the trace link recovery with the gold standard as recommended instances is tested, with both the LLM and using the SWATTR heuristics. This is followed by the evaluation of the trace link recovery integrated with the extraction of data-flow related elements used in eval1.

For the evaluation of *the trace link recovery of dataflows* (eval4), the extracted dataflow trace links were supposed to be matched against completely defined dataflows in the gold standard where all three roles *source*, *target* and *data* are annotated to *modelIds*. As no such dataflows exist and generally only 18 dataflows exist, where all roles are annotated, the evaluation of *the trace link recovery of dataflows* (eval4) is done

qualitatively. Another important factor is, that the gold standard only contains five dataflows, where source and target are traced to a architecture element. Through this evaluation, the data flow tracelinks are analysed case by case by the author to assess the value of extracted data flow tracelinks. A TP is counted if one of the five entries matches the extracted dataflow. For this evaluation only matching of the source and target is assessed, regardless on how the data is traced. Another case where a TP is counted, is when the elements of the traced dataflow have meaningful traces to the model, while the dataflow is described in the requirements text. For this assessment the author will use the definitions of the annotation guideline. Such a TP, which was not labeled in the gold standard indicate, missing labels for the traces. An FP is counted when the trace links lead to unrelated architecture items or when the traces are correct, but the extracted is not a dataflow in the requirement text. These decisions are done subjectively by the author, and thus the personal decisions are documented to provide some degree of transparency. A FN is counted, when a gold standard dataflow entry exists with with source and target annotated to a architecture item, but the trace link recovery completely missed it.

6.4 Evaluation Results

6.4.1 Element Extraction

For the evaluation of the extraction of dataflow-related elements different approaches are compared to each other leading to a final approach which is compared to the base line. For the evaluation of different methods GPT 4.1 is used. For *eval1*, two different approaches were compared: the *single-class* (SC) prompt mode and the *joint-class* (JC) prompt mode. In the SC prompt mode, the LLM is only tasked to extract one element type. Meanwhile, in JC mode, the LLM is tasked with extracting all elements in one go. As a base of comparison, the modes were compared to each other using zero-shot prompts.

As illustrated in Figure 6.1, the JC method consistently outperforms the SC approach across almost all evaluated element types. For *components*, JC achieves a higher median F2-score of 0.82 and a tighter distribution, indicating a more stable performance. While the recall for extracted components was similarly high (SC 0.83 vs. JC 0.86), the precision was 12% points higher when using JC (0.70), which aligns with the assumption made that using the JC will lead to fewer FP, as no phrases are labeled as multiple element types.
The difference is even more pronounced for *data* and *nodes*, where SC shows substantially lower scores and higher variance, whereas JC maintains both higher medians and

Figure 6.1: Evaluation of extraction of dataflow-related elements: zero-shot comparison of single class and joint class

better overall consistency. Especially the extraction of data elements benefited from the usage of the JC method, as the recall was higher by nearly 17% points (0.90), while the precision significantly was higher too with 0.58 compared to 0.27. For the extraction of *nodes* the recall values achieved by the different methods are closer to each other (JC 0.76 vs. SC 0.78), but the JC method could significantly increase the precision by 42% points (0.59). The resulting F2-scores are closer to each other for *entities*. While SC does provide better recall by 14% points (0.57), the precision of the JC method was significantly better.

The overall performance gain for the *extraction of dataflow-related elements* of the JC method is reflected in the higher macro-averaged scores across all metrics. Macro precision improves substantially from 0.337 in the SC mode to 0.596 in JC, while macro recall remains similarly high (SC 0.733 vs. JC 0.740). This shows that both approaches can capture a comparable proportion of relevant elements, while the JC approach reduces the number of FP. The improvement is further reflected in the macro F1-score (SC 0.43 vs. JC 0.64) and the macro F2-score (SC 0.55 vs. JC 0.69), both of which show the superior balance between precision and recall obtained by the JC method. These macro values reinforce the findings of the per-class analysis, demonstrating that the joint class configuration provides a more effective overall strategy for element classification.

The expected higher recall of SC could not be confirmed, except for *entities*. Additionally, the SC takes up considerably more resources, as for each requirement and each element

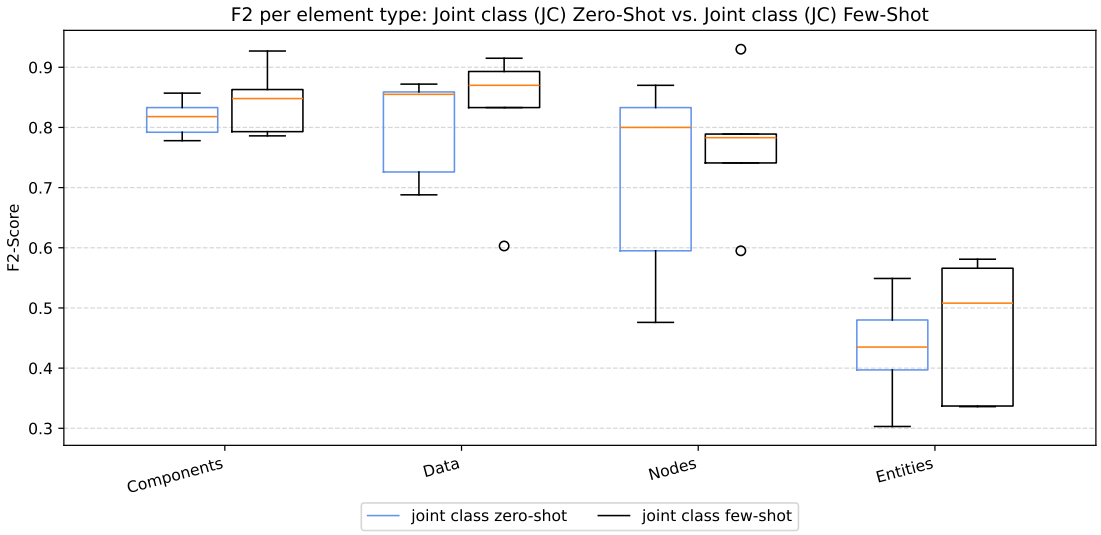

Figure 6.2: Evaluation of extraction of dataflow-related elements: Comparison of joint class using few-shot against zero-shot

type, one call to the LLM has to be made. This means for 93 requirements, 372 requests are sent, which also takes a longer time to finish in sum.

Few-Prompt

In order to evaluate how few-shot prompting affects the *extraction of dataflow-related elements*, four examples are randomly selected (random mode) from the training pool. The examples are parsed into the expected LLM output for the SC and JC modes to construct few-shot prompts. Results showed that using additional examples mostly benefited both SC and JC modes. This can be seen for the JC in Figure 6.2. The JC method with few-shot prompting achieves overall higher F2-scores compared to the zero-shot variant. For *components*, few-shot reaches a slightly higher median F2-score (FS 0.85 vs. ZS 0.82) and a larger maximum, indicating consistent but moderate improvements. This is also reflected in the recall, which only slightly improved when using few-shot prompting (0.88). The effect is more pronounced for *data*, where few-shot shows a median F2-score of 0.87 compared to 0.85 in zero-shot, while achieving a tighter distribution, reflecting greater stability across folds. The macro recall for *data* stayed relatively high with a value of 0.89. For *nodes*, the results are mixed: zero-shot maintains a slightly higher median F2-score (0.80), while few-shot achieves higher top values but greater variance. At the same time, the recall for the extraction of nodes was improved significantly, with a macro average recall of 0.85 compared to the zero-shot

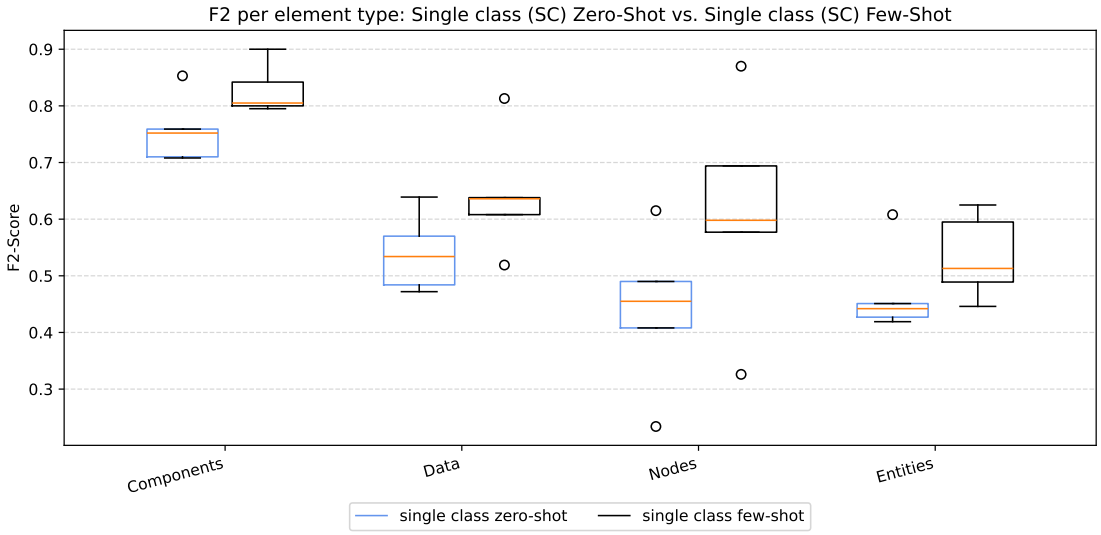

Figure 6.3: Evaluation of extraction of dataflow-related elements: Comparison of single class using few-shot against zero-shot

approach achieving 0.76, resulting in an improvement of 9% points. For *entities*, few-shot achieves a median F2-score of 0.51 compared to 0.44, achieving a higher maximum value, while the variance is similar for both approaches. These results demonstrate that few-shot prompting generally provides more robust performance, with particularly strong benefits for recall of the node extraction when using JC mode. Compared to these results, the SC mode benefited more from few-shot prompting as shown in Figure 6.3. For *components*, the median F2-score increases from 0.75 in zero-shot to 0.8 in few-shot, with higher top scores of 0.9. This result is reflected in both higher average recall for *component* extraction with 0.88 and precision 0.68, compared to the zero-shot approach with an average recall of 0.83 and precision of 0.58. The effect of few-shot prompting is more pronounced for *data*, where a median of 0.64 is achieved compared to only 0.53 in zero-shot. The few-shot approach combined with the SC mode achieved for the extraction of *data* elements a similar average recall (0.74), while increasing precision significantly by 13% points (0.42). For *nodes* few-shot increases the median F2-score to around 0.58 and reached a higher maximum value with 0.87. This is reflected in the considerably higher precision for the extraction of *nodes*, which increased by 18% (0.35), while average recall for the extraction of *nodes* stayed the similar with(0.77). Finally, *entities* also benefit from few-shot prompting, which improves the median F2-score by 7% points (0.51), while achieving the same average recall (0.57), while increasing the precision (0.43). These results demonstrate that few-shot prompting improves the single-class approach over all dataflow-related element types, with particularly large benefits towards precision. Through few-shot prompting both modes JC and SC

now achieve more similar results than compared to the zero-shot approach. Only in the average recall for the extraction of *entities* the SC approach holds a higher value with 0.57. While for *components* bot JC and SC combined with few-shot prompting achieved similar results, the recall for *data* and *nodes* are still significantly higher (see Table 8.2).

Example selection

On this basis, it is evaluated if the selection of examples given to the LLM in the few-shot approaches affects the results. For the SC mode, the LLM was provided only by examples regarding each element type (single class positive mode). Results showed that only the recall of the *component* extraction increased, reaching an average of 0.91. Meanwhile the recall for all other extractions decreased resulting in a macro average recall of 0.75. When the examples were chosen through the balanced mode, the recall increased by 6% points for the extraction of *data* (0.8), maintaining a similar precision (0.4) compared to the few-shot approach, where the examples are chosen randomly. The increase was smaller for *nodes*, but still achieving the best recall yet for SC (0.81). Overall, the selection of the examples did not have much of an impact, but with a balanced example set, GPT 4.1 in SC mode could achieve better results then with the single class positive mode, across all dataflow-related element types besides for *components*.
For the JC mode, examples were now chosen deterministically, ensuring a representative training set. Through this, the JC few-shot approach achieved higher recalls for the extraction of components and data (components 0.91, data 0.94), compared to the approach where examples are taken randomly out of the training pool. At the same time the recall for both the extraction of *node* and *entity* stayed nearly the same, with a decrease for *entities* of 5% points (0.41). This means the configuration achieving the best overall results is JC mode utilizing few-shot prompting with a representative example pool, where all element types are represented.

Model selection

In order to assess the effect of the choice of LLM model, the JC mode with few-shot prompting and selected examples using GPT-4.1 is compared to GPT-4.1 mini and LLaMA 3 8B. As illustrated in Figure 6.4, GPT-4.1 achieves the highest and most stable performance across folds, with F2-scores ranging from 0.67 to 0.79 and a median of approximately 0.71. In comparison, GPT-4.1 mini yields lower and more variable results, with F2-scores between 0.57 and 0.73 and a median of about 0.65, indicating a noticeable performance gap to GPT-4.1. This is also reflected in the recall across all element types. Especially for *data* where the recall decreased by 16% points to 0.78 and *node*

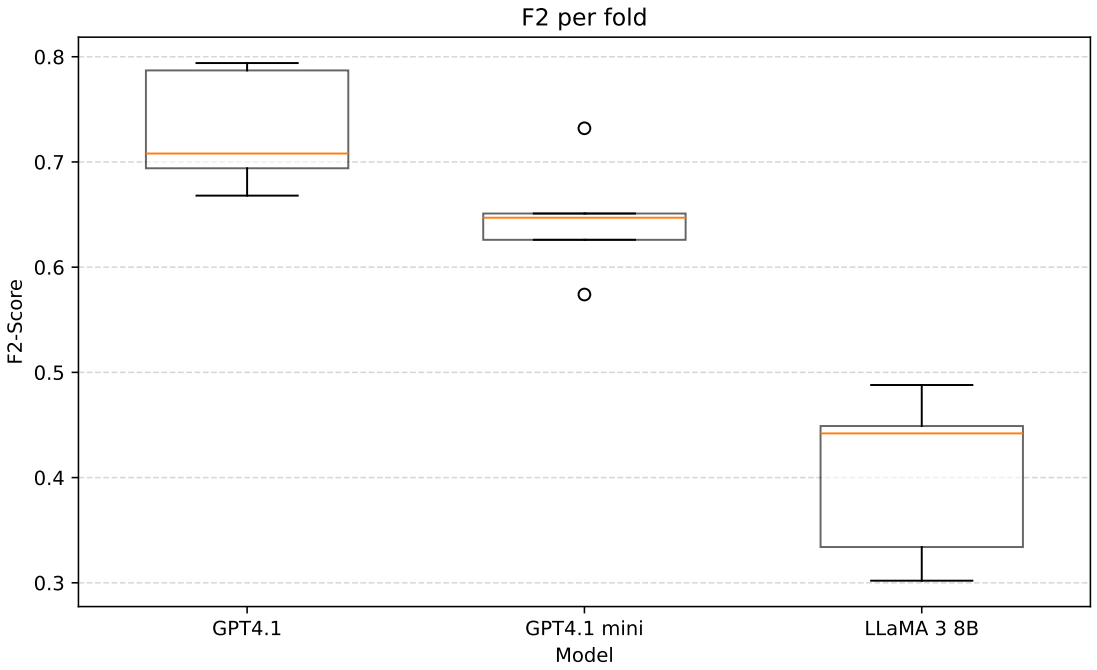

Figure 6.4: Dataflow-related element extraction model comparison using JC and det few-shot prompts

with a decrease of 13% points to 0.72. LLaMA 3 8B performs significantly worse, with F2-scores only between 0.30 and 0.49 and a median around 0.44, making it clearly less usable for this task, as the macro average recall decreases significantly across all dataflow-related element types by 40% points to 0.38. Overall, these results highlight that GPT-4.1 provides the most effective and reliable performance, while GPT-4.1 mini offers a weaker alternative and LLaMA 3 8B is not suited for this task, although it did process the requests the fastest. When comparing against the element extraction of SWATTR as a baseline, results show an increase in correctly created recommended instances when using the JC mode with a deterministic selected few-shot example using GPT 4.1. As the SWATTR approach focuses on achieving a high recall and filtering out FP later in the pipeline, the recall values are compared in the following section. These are shown in Figure 6.5, where the best performing LLM approach achieves consistently higher recall compared to the SWATTR baseline for most element types. For *components*, GPT-4.1 reaches a median recall of 0.91, clearly outperforming SWATTR, which remains at 0.80. The difference is even more pronounced for *data*, where GPT-4.1 achieves nearly perfect recall across folds, while SWATTR achieves a median of about 0.70. For *nodes*, SWATTR shows high variability, ranging from 0 to 1, whereas GPT-4.1 achieves a consistently high recall. In contrast, the extraction of *entities* remains a challenging task for both approaches, with low recall values overall and only marginal improvements

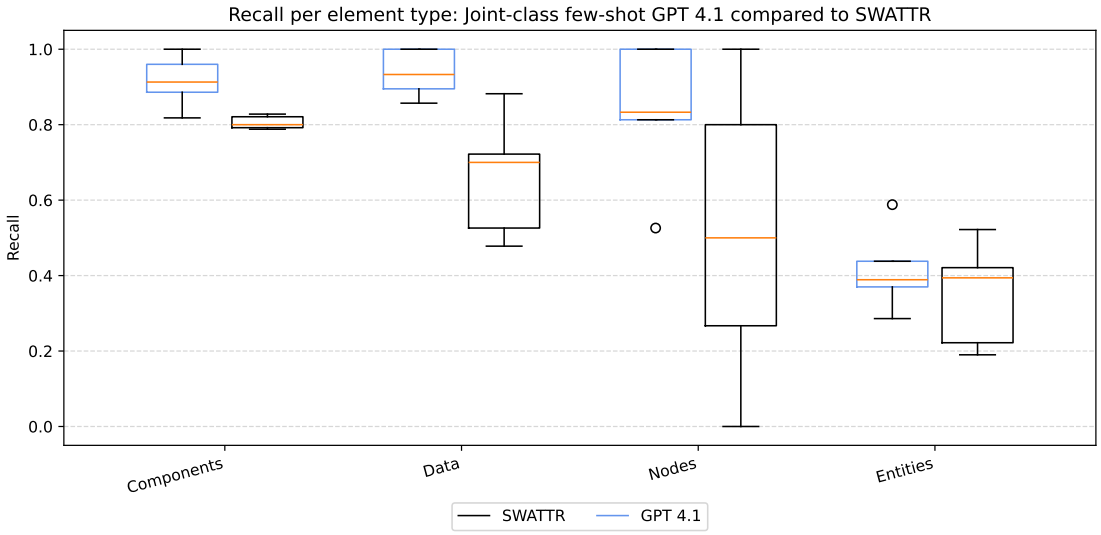

Figure 6.5: Recall SWATTR and GPT 4.1

observed for GPT-4.1. Overall, these results indicate that GPT-4.1 substantially improves recall over the baseline for components, datatypes, and nodes, while performance for entities remains similar. All results regarding eval1 can be found in the Table 8.2.

6.4.2 Dataflow Extraction

For the evaluation of the *extraction of dataflows* (eval2), the focus will lie on micro-averaged values, as they provide a more stable estimate of overall performance by pooling predictions across folds. Macro-averaged results are included for completeness, but show high variance due to fold imbalance, since some folds contain only very few complete dataflows.

As shown in Table 6.1, the few-shot configuration increases recall compared to zero-shot, both in the macro average from 0.35 to 0.45 and in the micro average from 0.46 to 0.54. Macro precision remains low for both approaches, with 0.16 for few-shot and 0.19 for zero-shot, indicating that the LLM frequently predicts dataflows that do not exist, resulting in a high number of false positives. This is also reflected in the F2-score, which weights recall more strongly, increasing from 0.346 to 0.396 in the micro setting. While precision is largely unchanged, recall and F2 clearly benefit from few-shot prompting. Since in requirements tracing recall is typically prioritized over precision, the few-shot configuration provides a more usable result.

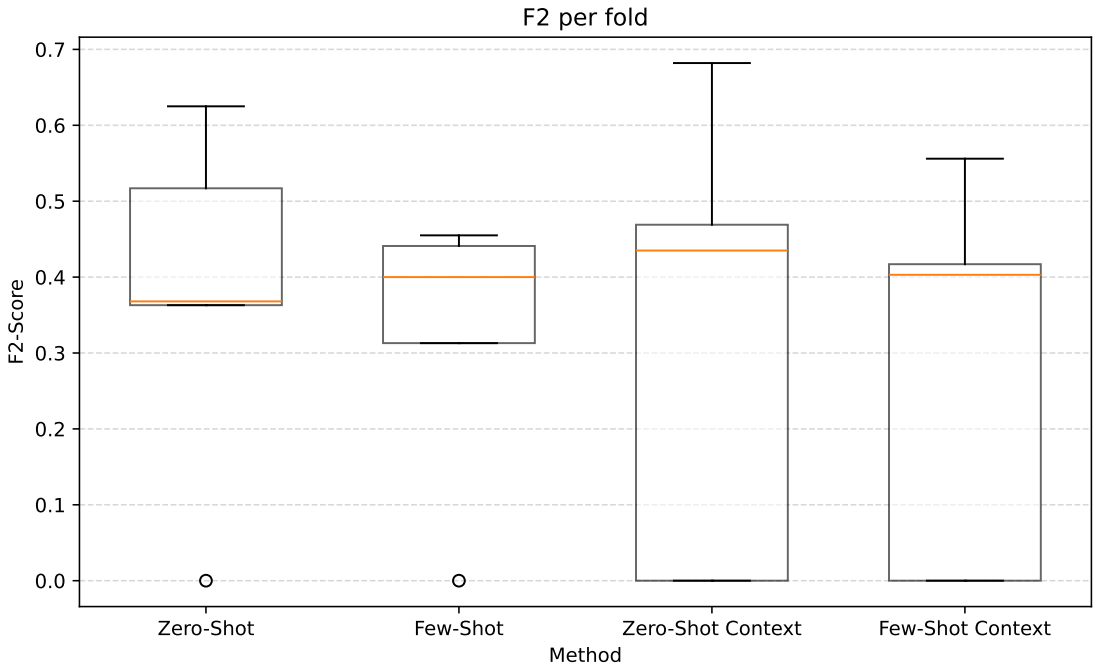

Figure 6.6: Dataflow extraction method comparison using GPT 4.1

Compared to these results, using additional extraction context of dataflow-related elements for the dataflow extraction showed mixed effects. In the zero-shot setting, context increases precision from 0.192 to 0.31 (macro) and from 0.175 to 0.31 (micro), while recall remains similar (0.35 vs. 0.33 macro, 0.458 micro in both cases). This yields a higher F1 (0.37 micro) and F2 (0.42 micro), indicating that context improves zero-shot extraction by reducing false positives. In contrast, the few-shot with context approach fails to replicate the recall gains observed without context. Recall drops from 0.45 to 0.31 (macro) and from 0.54 to 0.42 (micro), while precision remains low. This leads to lower F1 and F2 values compared to the plain few-shot approach. The distribution of F2-scores across folds further shows the varying performance, regardless of method used, as illustrated in Figure 6.6. While Zero-Shot and Few-Shot show relatively tight interquartile ranges with outliers at 0, the additional context results in much higher variance. In particular, zero-shot enriched with context shows the widest spread, ranging from near-zero to 0.68. The few-shot prompt with context also shows increased instability compared to the basic few-shot, with several folds dropping to very low performance.

Overall, these results suggest that additional context can support zero-shot prompting, especially improving precision, but in the few-shot configuration used, it reduces recall and affects overall performance. Regardless of the approach tested, the performance

Table 6.1: Evaluation results for dataflow extraction. Macro = mean across folds, Micro = pooled across folds.

Method	P		R		F1		F2	
	macro	micro	macro	micro	macro	micro	macro	micro
Zero-shot	0.192	0.175	0.353	0.458	0.248	0.253	0.302	0.346
Few-shot	0.162	0.191	0.453	0.542	0.232	0.283	0.322	0.396
Zero-shot + context	0.308	0.306	0.330	0.458	0.306	0.367	0.317	0.417
Few-shot + context	0.226	0.227	0.313	0.417	0.246	0.294	0.275	0.357

of dataflow extraction using GPT-4.1 remains similarly low, highlighting the difficulty of this more abstract task for LLMs compared to the *extraction of dataflow-related elements*.

6.4.3 Trace Link Recovery of Dataflow-Related Elements

As illustrated in Figure 6.7, the performance of trace link recovery strongly depends on whether the gold standard elements or automatically extracted dataflow-related elements were used. When evaluated with the gold standard annotations as input, GPT4.1 outperforms SWATTR across almost all element types. For *components*, GPT4.1 achieves both higher precision by 11% points (0.88) and recall by 15% points (0.88), resulting in a clear improvement in the F2-score (GPT-4.1 0.879 vs. SWATTR 0.737). This can also be seen in the distribution of F2-scores across folds for tracing components. Here, GPT-4.1 achieved a higher median (0.77) compared to SWATTR (0.52), with the maximum value being lower than the minimum value of GPT-4.1. As the heuristics used in SWATTR were not adjusted for the additional element types, the performance across those new types was expectedly worse. The difference is particularly high for *nodes*, where GPT4.1 reaches a recall of 0.96 compared to 0.33 for SWATTR, leading to more than double the F2-score (GPT-4.1 0.90 vs. SWATTR 0.36). Similarly, for *entities*, GPT4.1 shows a strong recall advantage (GPT-4.1 0.81 vs. SWATTR 0.08), though at the cost of a low precision. The extraction of *data* remains challenging for both methods, but GPT-4.1 provides a modest gain in F2 (0.247 vs. 0.067). On the macro level, these improvements translate into a substantially higher F2-score of 0.635 for GPT4.1 compared to 0.312 for SWATTR, reflecting a more balanced recovery of dataflow-related elements.

The results change when using automatically extracted elements from *eval1* as input. Here, both methods show a clear drop in performance due to error propagation from the extraction step. SWATTR maintains slightly higher macro precision (0.366 vs. 0.292), but GPT4.1 demonstrates considerably stronger recall (0.736 vs. 0.257). This difference

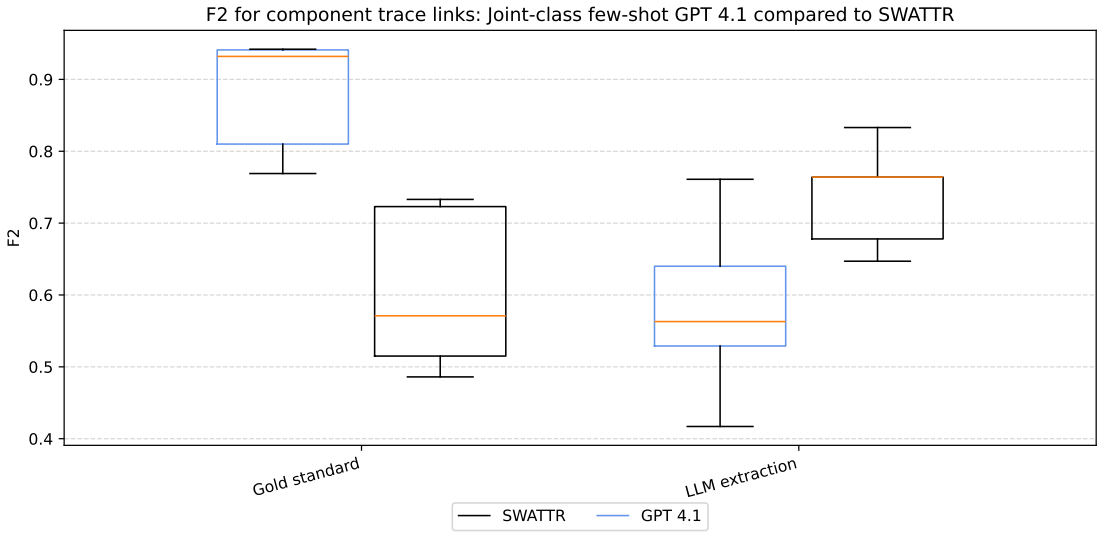

Figure 6.7: Component trace link F2-scores for SWATTR and GPT-4.1, using the gold standard and the LLM extraction from eval1

is most visible for *nodes* and *entities*, where GPT4.1 reaches recall values above 0.90, compared to 0.40 and 0.03 for SWATTR. The low precision of GPT-4.1 for *components* (0.30) leads to a large number of false positives, which negatively affects the overall F2-score. SWATTR shows a more stable distribution of F2-scores, with a higher median (0.76) compared to SWATTR (0.56). While GPT4.1 provides a higher recall by 19% points (0.79), the low precision is reflected in the F2-score. On the macro level, GPT4.1 achieves a higher F2-score (0.545 vs. 0.265), indicating that despite noisier input, GPT4.1 can still recover a larger portion of true links, but at the expense of precision.

Overall, the eval3 shows that GPT4.1 is capable of surpassing the heuristic baseline when accurate gold standard elements are available. However, once element extraction errors are introduced, GPT4.1's advantage shifts towards recall, while SWATTR maintains stronger precision and stability. This indicates the sensitivity of LLM-based approaches to propagated errors. All results regarding eval3 can be found in Table 8.2.

6.4.4 Trace Link Recovery of Dataflows

For the *trace link recovery of dataflows* (eval4), the extracted dataflow-related elements from eval1 and the dataflows from eval2 were used to create trace links. In eval2, GPT-4.1 with the few-shot setup (without extraction context) extracted 47 candidate dataflows, which, after the tracing step, were reduced to 18 flows where both source and target could be mapped to elements of the EVerest model.

Table 6.2: Results trace link recovery of dataflows (eval4)

	TP	FP	FN	Precision	Recall	F_1	F_2
Eval4	8	9	0	0.47	1.00	0.64	0.82

In the gold standard, five dataflows are fully labeled with source and target model IDs (Req. 7, 34, 64, 15, and 8). For all of these, the LLM was able to trace both source and target correctly. For example, in Req. 7 the dataflow from "CSMS" to the "OCPP modules" was correctly traced to the architecture elements *ChargingStationManagementSystem* and *OCPP*. Req. 15, describing a dataflow from "EvseSecurity" to the "EV" including "private keys", was also counted as correct, as traces for source and target were correct, although the data was traced to an unrelated element.

From the remaining 13 extracted dataflows, three represented valid dataflows that were not part of the gold standard annotations (for example, Req. 70 describing an "API module" sending "session events" to an "external system"). These extracted dataflow trace links were deemed as traced correctly, showing that the approach can also identify dataflows that were overlooked in labeling. The remaining nine cases were false positives. They consisted mainly of requirements describing storage relations rather than actual data transfer (e.g., Req. 71 describing the "charging station" storing "certificates" in the "TPM"), descriptions of control flow such as authentication results (e.g., Req. 47 describing the "Auth module" signaling to the "EvseManager", that it can proceed after authentication), or cases where source or target were traced to unrelated architecture items.

Overall, this results in eight true positives, nine false positives, and no false negatives. From these values, the micro precision over all folds calculated is 0.47, recall 1.0, with an F1-score of 0.64 and an F2-score of 0.82. While recall reaches 1.0, as all annotated gold standard dataflows were recovered, precision remains low, indicating that many extracted candidates are not valid dataflows. These results suggest that the approach is capable of identifying correct dataflows when both endpoints are clearly modeled, but struggles with differencing between dataflows and storage or control flow. The found trace links and notes regarding eval4 can be found in Table 8.3.

6.5 Discussion and Threats to Validity

In this section the results of the prior evaluation are discussed, followed by possible threats to validity. In the *extraction of dataflow-related elements* (eval1), the prompt configurations showed clear differences in performance. The joint-class (JC) approach

consistently outperformed the single-class (SC) approach for all configurations. The assumption that the recall would be higher when using SC mode could not be confirmed, as JC achieved a higher macro recall by 2% points for both methods' best configurations (0.78), while at the same time reaching a 16% points higher precision (JC 0.63).
The use of few-shot prompting mostly affected precision, while recall values were maintained. In JC mode the recall only increased slightly (0.74 to 0.77), while precision improved from 0.58 to 0.62. The effect of few-shot prompting was more pronounced in SC mode, where macro precision increased by 15% points (0.47), leading to significantly higher F2-scores for all element types. The effect of example selection was less critical, with only minor variations observed depending on whether examples were chosen randomly, single-class positive, or balanced.
Model choice played a significant role. While GPT-4.1 provided the most stable and effective results with a macro F2-score across all element types of 0.73, GPT-4.1 mini reached only 0.65, and LLaMA 3 8B dropped to 0.40, making it unsuitable for this task despite its faster runtime.
When compared against the recall-oriented SWATTR baseline, the LLM-based approach showed improvements. SWATTR achieved strong recall for *components* with a value of 0.80, but GPT-4.1 in JC mode combined with few-shot prompting outperformed it with a recall of 0.92. For the other element types, SWATTR performed worse, as expected, since its heuristics were designed primarily with component extraction in mind.
The extraction of the other element types using the best-performing LLM approach showed similar performances, except for entities. For entities, only a recall of 0.41 could be achieved with a precision of 0.6. This probably stems from the more abstract definition of entities provided through the annotation guidelines, which leads to the LLM struggling with extracting entities. Although explicitly defined, that entities may be protocols or libraries, many of them were missed out (e.g., "libevsesecurity", "powermeterdriver", "ocpp1.6"), and were extracted as components or nodes. This may indicate that the definition provided to the LLM is not detailed enough to make the distinction between the element types clear. This means that although the LLM extracted the correct elements, it mislabeled a significant portion of them, leading to FN for element types where they are missing and FP for types where they occur. In summary, the LLM approach used still mostly showed promising results for the extraction of *component*, *data* and *nodes*. The results for *entities* remained mixed, regardless of the method used.

The evaluation of the *extraction of dataflows* (eval2) showed that overall performance remains low, regardless of the chosen configuration. Few-shot prompting mainly increases the average recall to a micro average of 0.54, while precision remains largely unchanged (0.19). This leads to a micro average F2-score from 0.39. While few-shot prompting provides more usable results by extracting more relevant dataflows than zero-shot, many false positives remain. Although often the source and target of the

dataflow could be extracted correctly, most of the times the LLM struggled with labeling the correct data elements.
The use of additional extraction context of dataflow-related elements showed mixed effects. In the zero-shot setting, context increases precision considerably by 11% points (micro 0.31) while recall remains low (micro 0.46), leading to higher F1 (0.37) and F2 (0.42) scores. This suggests that context helps to reduce false positives when no examples are provided. In contrast, in the few-shot configuration, context reduces recall substantially by 12% points (micro 0.42) without compensating for improvements in precision, resulting in lower F1 and F2 values overall. The distribution of scores across folds further highlights the instability introduced by context. While zero-shot and few-shot prompting alone show relatively tight distributions, adding context results in much wider spreads, with some folds dropping to near-zero performance. As fully specified dataflows were relatively rare in the gold standard, the evenly distribution of them across the folds could not be guaranteed, they the requirements were shuffled randomly, resulting in folds, where only a small number of dataflows exist. Because of this, the micro-averaged results give a better overview of the dataflow extraction performance. Taken together, these results suggest that context can improve zero-shot extraction by reducing false positives, but when combined with few-shot prompting, it reduces recall and overall performance. Across all methods, the extraction of complete dataflows with GPT-4.1 remains challenging. Compared to the extraction of individual dataflow-related elements (eval1), results for eval2 are significantly lower, highlighting the difficulty of extracting dataflow relations from requirement text.
In the *trace link recovery of dataflow-related elements* (eval3), GPT-4.1 consistently outperforms the SWATTR baseline when using the gold standard as input. For *components*, the F2-score improves from 0.74 with SWATTR to 0.88 with GPT-4.1. For tracing *nodes*, GPT-4.1 also achieves a high F2-score close to 0.90, while it struggles with the tracing of *entities* and *data*. For entities, recall is relatively high (0.80), but precision remains low, leading to many false positives. For both methods, tracing performance is poor for data (e.g., GPT-4.1 F-2 0.247). Tracing element types other than components with SWATTR leads to poor results, which is expected, as the heuristics were primarily designed for component tracing.

When integrating the extracted elements from eval1, performance decreases significantly for both methods. For *components*, GPT-4.1 still achieves high recall (0.79), but precision drops strongly (0.30), resulting in many false positives and a lower F2-score (0.58). SWATTR, in contrast, handles the input better, filtering out more false positives and showing less performance degradation (F2-score 0.61). This highlights the dependency of the trace link recovery on the quality of the underlying extraction and shows that while GPT-4.1 can achieve superior results with high-quality inputs, heuristic-based approaches like SWATTR remain more stable when confronted with noisy element extractions.

For the extraction of tracelinks for dataflows (eval4), the results showed that the LLMs can be used to create trace links to their contained elements. Through tracing, many of the false positive dataflows, which were extracted in eval2, were filtered out, as only dataflows where source and target could be traced to a concrete architecture item were returned. Through this, it was possible to identify all five dataflows annotated in the requirements with modelIDs, and in addition, three more were found, which were not yet labeled in the gold standard. After tracing the dataflow-related elements and aggregating them to the dataflows extracted, some of the mistakes that the LLM made in eval1 and eval2 were mitigated. As many of the mistakes in eval1 did not stem from text mentions of elements not being extracted at all, but from wrongly labeling them, the results could still be used for tracing the source and target. This comes from the fact, that source and target are not typed and a dataflow can occur between component, node and entity. Because the dataflows in eval4 were filtered based on whether both source and target could be traced, many of the false-positive dataflows from eval2 describing unconcrete dataflows were filtered out. The results show that the LLM-based approach can recover dataflow trace links when both source and target are explicitly modeled, and even identify additional valid flows not yet included in the gold standard. However, these findings should be interpreted with caution, as the gold standard only contains five annotated dataflows with modelIDs, which limits the representativeness of the results.

Different threats to validity may affect the results of this work. One threat comes from the cross-validation setup. The folds were not completely balanced, with three folds containing 19 requirements and two folds containing 18 requirements. While this does not strongly affect the overall results, as four requirements were used from the training pool each, it may lead to minor differences in macro-averaged values. Another threat is that the prompts were tested and adjusted specifically with GPT-4.1, which may have introduced a bias toward this model and limited the neutrality of the prompt design. When using few-shot prompting, the choice of examples may also affect performance. This effect was mitigated to some extent by the 5-fold cross-validation, as different requirements were included in training and evaluation across folds. Finally, the annotation guidelines used during the annotation of the EVerest requirements were adjusted iteratively, and the same definitions were also used to design the prompts. This overlap may have influenced the evaluation, since both annotation and extraction are based on similar assumptions about the definition of dataflow-related elements.

Another threat to validity in this evaluation arises from the limited coverage of the gold standard, which only contains five fully labeled dataflows. This makes the recall appear high for the trace link creation for dataflows, as all five dataflows were found, while the actual ability to generalize to a larger set of requirements cannot be assessed. Another threat, especially relevant for the qualitative evaluation in eval4, comes from

the dataflow definition. Several requirements describe storage or control flow rather than explicit data transfer, and classifying these as false positives was done by the author alone, which definitely introduces subjectivity, as the evaluation involved manual judgment in assigning TP, FP, and FN labels. These limitations reduce the generalizability of the findings and suggest that the reported metrics should be interpreted as indicative rather than absolute.

7 Conclusion and Future Work

This thesis focused on extending existing traceability approaches towards security-related requirements, particularly the tracing of dataflows. The goal was to evaluate how large language models can be utilized to extract dataflow entities from requirements and create trace links to architecture models. As a base for evaluation, a gold standard was created for EVerest, where the requirements are fully labeled with SecLan element types (cf. section 2.5). This gold standard provides a valuable information source beyond the scope of this thesis, as labeled security-related requirements are considerably scarce.
The evaluation of the dataflow-related element extraction showed that GPT-4.1 can reliably identify components and nodes with high recall and precision, while entities and data elements remain more difficult. The extraction of complete dataflows was considerably harder, with both recall and precision being low. This indicates that the task remains challenging for LLMs. For trace link recovery of the dataflow-related elements, the results showed that GPT-4.1 outperforms the SWATTR approach when high-quality element annotations are available, particularly for components and nodes. However, when integrating automatically extracted elements, performance decreased strongly. SWATTR handled the noisy input more robustly, showing the dependency of LLM-based recovery on the quality of the extracted elements. In the final evaluation for the creation of traces to dataflows, combining the results of previously used steps, the used approach was able to recover all five dataflows annotated in the gold standard and found four further valid flows that were not labeled. At the same time, nine false positives were observed, mainly due to dataflows being traced that did not match the definition (cf. chapter 4). These results indicate that LLMs can identify valid trace links for dataflows, but that the distinction between different relation types is still a major source of errors.

This thesis provides different points where future work could be valuable. As the gold standard is limited in size, especially regarding dataflows, testing it across other projects is necessary for more representative evaluations. Future work could also investigate improved prompting strategies and fine-tuning. As the gold standard was finished relatively late, it was not viable to use automated prompt optimization.
The evaluation showed that LLM-based and heuristic approaches have complementary strengths. While LLMs achieved higher recall, the heuristic approach used in SWATTR was more stable under noisy input. Combining both heuristics with LLM-base extraction

may help in balancing out recall and precision of extracted trace links. As a foundation for integrating traceability for dataflows, the extracted dataflow trace links only directly link the source and target to the respective architecture items. Future work could build on this by moving beyond direct source and target tracing towards graph-based tracing, where relevant parts of the dataflow graph are also traced. In this way, traceability could be extended to entities that are not explicitly mentioned in the requirement text but are still involved in the modeled dataflow. In summary, this thesis demonstrates both the potential and the current limitations of LLM-based approaches for extending traceability towards security-related dataflows. This foundation provides a starting point for future research on combining LLMs and heuristics to achieve more accurate and robust traceability in security-critical domains.

8 Appendix

Table 8.1: Evaluation results dataflow-related element extraction, macro for each type across all folds

	Component				Data				Node				Entity				macro			
Method	P	R	F1	F2	P	R	F1	F2	P	R	F1	F2	P	R	F1	F2	P	R	F1	F2
Zero-Shot																				
SC	0.583	0.829	0.676	0.756	0.269	0.731	0.391	0.540	0.169	0.779	0.273	0.441	0.282	0.574	0.374	0.470	0.326	0.728	0.428	0.551
JC	0.698	0.860	0.763	0.816	0.575	0.896	0.694	0.800	0.594	0.760	0.661	0.715	0.466	0.431	0.440	0.433	0.583	0.737	0.640	0.691
Few-shot random																				
SC	0.683	0.878	0.766	0.828	0.422	0.743	0.537	0.643	0.349	0.773	0.474	0.613	0.429	0.577	0.486	0.534	0.471	0.743	0.566	0.654
JC	0.726	0.883	0.793	0.843	0.640	0.891	0.741	0.823	0.582	0.847	0.681	0.768	0.549	0.455	0.487	0.466	0.624	0.769	0.676	0.725
Few-shot deterministic																				
SC-positve	0.662	0.911	0.763	0.844	0.365	0.755	0.487	0.616	0.232	0.771	0.350	0.511	0.345	0.582	0.429	0.507	0.401	0.754	0.507	0.619
SC-balanced	0.633	0.887	0.734	0.817	0.402	0.801	0.528	0.660	0.288	0.814	0.417	0.582	0.376	0.530	0.431	0.482	0.425	0.758	0.528	0.635
JC deterministic	0.723	0.916	0.801	0.863	0.584	0.937	0.718	0.835	0.612	0.854	0.707	0.785	0.603	0.414	0.483	0.438	0.630	0.780	0.677	0.730
Models																				
JC few-shot det Llama 3	0.644	0.554	0.578	0.560	0.564	0.421	0.472	0.439	0.867	0.290	0.423	0.332	0.496	0.256	0.333	0.281	0.643	0.380	0.452	0.403
JC few-shot det GPT 4.1 mini	0.739	0.853	0.788	0.825	0.692	0.782	0.729	0.758	0.570	0.719	0.621	0.671	0.471	0.309	0.368	0.329	0.618	0.666	0.627	0.646
Baseline																				
SWATTR	0.162	0.806	0.268	0.444	0.076	0.662	0.136	0.259	0.022	0.513	0.042	0.093	0.054	0.356	0.094	0.167	0.079	0.584	0.135	0.241

Table 8.2: Evaluation results dataflow-related element extraction, macro for each type across all folds

	Component				Data				Node				Entity				macro			
Method	P	R	F1	F2	P	R	F1	F2	P	R	F1	F2	P	R	F1	F2	P	R	F1	F2
with goldstandard																				
SWATTR	0.775	0.730	0.749	0.737	0.167	0.059	0.086	0.067	0.800	0.334	0.419	0.359	0.100	0.080	0.089	0.083	0.460	0.301	0.336	0.312
GPT 4.1	0.875	0.882	0.876	0.879	0.276	0.241	0.257	0.247	0.798	0.956	0.856	0.907	0.212	0.813	0.332	0.508	0.540	0.723	0.580	0.635
with LLM extraction																				
SWATTR	0.654	0.595	0.622	0.606	0.000	0.000	0.000	0.000	0.610	0.398	0.444	0.412	0.200	0.033	0.057	0.040	0.366	0.257	0.281	0.265
GPT 4.1	0.298	0.794	0.425	0.582	0.133	0.255	0.172	0.213	0.466	0.964	0.621	0.785	0.272	0.933	0.409	0.601	0.292	0.736	0.407	0.545

Table 8.3: Qualitative evaluation of trace link recovery for dataflows (eval4), part 1

Req. ID	Dataflow	Trace Links	Notes
7	CSMS → OCPP modules: malformed input from the CSMS	CSMS → ChargingStationManagementSystem (PCM RC) OCPP modules → OCPP (PCM Component)	**TP** - Both endpoints correct, no data element traced
34	charger → CSMS: transaction related data, telemetry, information about updates	CSMS → ChargingStationManagementSystem (PCM RC) charger → ChargingStation (PCM RC) data: TransactionRequest; Powermeter; FirmwareUpdateRequest / UpdateFirmwareRequest / SignedUpdateFirmwareRequest / FirmwareUpdateResponse	**TP** - Endpoints correct; "updates" mapped to multiple datatypes.
65	charger → CSMS: OCPP messages	CSMS → ChargingStationManagementSystem charger → ChargingStation	**TP** – Endpoints correct; no data

Table 8.4: Qualitative evaluation of trace link recovery for dataflows (eval4), part 2

Req. ID	Dataflow	Trace Links	Notes
15	EvseSecurity → TPM: private keys used for TLS	EvseSecurity module → EvseSecurity (PCM Component) TPM → TPM (PCM RC) data: GetCertificate-SignRequestResult	**TP** — Endpoints correct; (TPM added).
8	EV → ISO15118modules: malformed input	EV → Car (PCM RC) ISO15118 modules → EvseV2G (PCM Component)	**TP** — Endpoints correct; ISO15118 resolved via EvseV2G.
80	EVerest → log file: customer identification means, contract identification means, ID tags, EV MAC addresses, bank information	log file → UploadLogsRequest (PCM Datatype) EVerest → System (PCM Component); bank information → BankSessionToken EV MAC addresses → Evse (PCM Component)	**FP** — “log file” wrong traced; no "EVerest" wrong traced found.

Table 8.5: Qualitative evaluation of trace link recovery for dataflows (eval4), part 3

Req. ID	Dataflow	Trace Links	Notes
75	libevse-security → TPM: private keys, certificates	libevse-security → EvseSecurity (PCM Component) TPM → TPM (PCM RC); TLS → CloudSystem data: CertificateType; CertificateHashData	**TP (not in GS)** — Correct mapping; flow not annotated in GS.
71	charging station → TPM: security certificates	charging station → ChargingStation (PCM RC) TPM → TPM (PCM RC) data: CertificateType	**FP** — Storage (data-at-rest), not a dataflow.
34 (FP)	EVerest → payment provider: relevant information	EVerest → System (PCM Component) payment provider → CloudSystem	**FP** — EVerest is not System component

Table 8.6: Qualitative evaluation of trace link recovery for dataflows (eval4), part 4

Req. ID	Dataflow	Trace Links	Notes
36	EVerest → CSMS: authentication tokens	EVerest → System (PCM Component) CSMS → ChargingStationManagementSystem (PCM RC) data: Token	**FP** — EVerest is not System component
45	charger → log files: MAC addresses	charger → ChargingStation log files → UploadLogsRequest / UploadLogsResponse data: HardwareCapabilities	**FP** — "log files" traced wrong
33	OCPP → EVerest: start charging request	OCPP → OCPP (PCM Component) EVerest → System (PCM Component)	**FP** EVerest is traced wrong

Table 8.7: Qualitative evaluation of trace link recovery for dataflows (eval4), part 5

Req. ID	Dataflow	Trace Links	Notes
70	API module → external system: session events	API module → API (PCM Component) external system → CloudSystem (PCM RC)	**TP (not labeled in GS)** — Correct mapping
35	(charger → external system: health information	charger → ChargingStation (PCM RC) external system → CloudSystem (PCM Component) data: Diagnostics	**TP (not in GS)** — Correct mapping; not in GS.
0	EVerest → database: transaction information	EVerest → System (PCM Component) database → Store (PCM Component)	**FP** — EVerest traced wrong

Table 8.8: Qualitative evaluation of trace link recovery for dataflows (eval4), part 6

Req. ID	Dataflow	Trace Links	Notes
0	libocpp → SQLite database: transaction information	libocpp → OCPP (PCM Component) SQLite database → SqlLiteDatabase (PCM Component)	**FP** Storage relation, not a dataflow.
47	Auth module → EvseManager: authentication result, permission to charge	Auth module → Authentication (PCM Component) EvseManager → EvseManager (PCM Component)	**FP** — Control flow (permissions), not dataflow

Bibliography

[1] G. Antoniol et al. “Recovering traceability links between code and documentation”. In: *IEEE Transactions on Software Engineering* 28.10 (2002), pp. 970–983. DOI: 10.1109/TSE.2002.1041053.

[2] G. Antoniol et al. “Recovering traceability links between code and documentation”. In: *IEEE Transactions on Software Engineering* 28.10 (2002), pp. 970–983. DOI: 10.1109/TSE.2002.1041053.

[3] Hazeline U. Asuncion, Arthur U. Asuncion, and Richard N. Taylor. “Software traceability with topic modeling”. In: *2010 ACM/IEEE 32nd International Conference on Software Engineering*. Vol. 1. 2010, pp. 95–104. DOI: 10.1145/1806799.1806817.

[4] Happe Becker et al. “Evaluating Performance of Software Architecture Models with the Palladio Component Model”. In: *Model-Driven Software Development: Integrating Quality Assurance* (Jan. 2008). DOI: 10.4018/978-1-60566-006-6.ch005.

[5] Nicolas Boltz et al. “An Extensible Framework for Architecture-Based Data Flow Analysis for Information Security”. In: *Software Architecture. ECSA 2023 Tracks, Workshops, and Doctoral Symposium : Istanbul, Turkey, September 18–22, 2023, Revised Selected Papers. Ed.: B. Tekinerdoğan*. 17th European Conference on Software Architecture. ECSA 2023 (Istanbul, Türkei, Sept. 18–22, 2023). Vol. 14590. Lecture Notes in Computer Science. 46.23.03; LK 01. Springer, 2024, pp. 342–358. ISBN: 978-3-031-66325-3. DOI: 10.1007/978-3-031-66326-0_21.

[6] Tom B. Brown et al. “Language Models are Few-Shot Learners”. In: *CoRR* abs/2005.14165 (2020). arXiv: 2005.14165. URL: https://arxiv.org/abs/2005.14165.

[7] Jane Cleland-Huang et al. “Software Traceability: Trends and Future Directions”. In: *Future of Software Engineering, FOSE 2014 - Proceedings*. May 2014, pp. 55–69. DOI: 10.1145/2593882.2593891.

[8] Dominik Fuchß et al. “Enabling Architecture Traceability by LLM-based Architecture Component Name Extraction”. In: *22nd IEEE International Conference on Software Architecture (ICSA 2025)*. 22nd IEEE International Conference on Software Architecture. ICSA 2025 (Ottensee, Dänemark, Mar. 31–Apr. 4, 2025). 46.23.01; LK 01. 2025. DOI: 10.1109/ICSA65012.2025.00011.

[9] Dominik Fuchß et al. *LiSSA: Toward Generic Traceability Link Recovery through Retrieval-Augmented Generation*. Tech. rep. 46.23.01; LK 01. Institute of Electrical and Electronics Engineers (IEEE), 2025. DOI: 10.5445/IR/1000178348.

[10] Hui Gao et al. "Using Consensual Biterms from Text Structures of Requirements and Code to Improve IR-Based Traceability Recovery". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE '22. ACM, Oct. 2022, pp. 1–1. DOI: 10.1145/3551349.3556948. URL: http://dx.doi.org/10.1145/3551349.3556948.

[11] Aaron Grattafiori et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.AI]. URL: https://arxiv.org/abs/2407.21783.

[12] 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.

[13] Tobias Hey et al. *Requirements Traceability Link Recovery via Retrieval-Augmented Generation*. Tech. rep. 46.23.01; LK 01. 2025. 16 pp. DOI: 10.5445/IR/1000178589.

[14] Yunpeng Huang et al. *Advancing Transformer Architecture in Long-Context Large Language Models: A Comprehensive Survey*. 2024. arXiv: 2311.12351 [cs.CL]. URL: https://arxiv.org/abs/2311.12351.

[15] Khaled Jaber, Bonita Sharif, and Chang Liu. "A Study on the Effect of Traceability Links in Software Maintenance". In: *IEEE Access* 1 (2013), pp. 726–741. DOI: 10.1109/ACCESS.2013.2286822.

[16] Jan Keim et al. "Detecting Inconsistencies in Software Architecture Documentation Using Traceability Link Recovery". In: *20th IEEE International Conference on Software Architecture (ICSA)*. 20th IEEE International Conference on Software Architecture. ICSA 2023 (L'Aquila, Italien, Mar. 13–17, 2023). 46.23.01; LK 01. Institute of Electrical and Electronics Engineers (IEEE), 2023, pp. 141–152. ISBN: 979-83-503-9749-9. DOI: 10.1109/ICSA56044.2023.00021.

[17] Jan Keim et al. "Trace Link Recovery for Software Architecture Documentation". In: *Software Architecture. ECSA 2021*. Ed. by Stefan Biffl et al. Vol. 12857. Lecture Notes in Computer Science. Springer, Cham, 2021. DOI: 10.1007/978-3-030-86044-8_7. URL: https://doi.org/10.1007/978-3-030-86044-8_7.

[18] Omar Khattab et al. *DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines*. 2023. arXiv: 2310.03714 [cs.CL]. URL: https://arxiv.org/abs/2310.03714.

[19] Silke R. Kirchner. "OCPP Interoperability: A Unified Future of Charging". In: *World Electric Vehicle Journal* 15.5 (2024). ISSN: 2032-6653. DOI: 10.3390/wevj15050191. URL: https://www.mdpi.com/2032-6653/15/5/191.

[20] Shekoufeh Kolahdouz-Rahimi, Kevin Lano, and Chenghua Lin. *Requirement Formalisation using Natural Language Processing and Machine Learning: A Systematic Review*. 2023. arXiv: 2303.13365 [cs.CL]. URL: https://arxiv.org/abs/2303.13365.

[21] Catherine Kosten, Farhad Nooralahzadeh, and Kurt Stockinger. "Evaluating the effectiveness of prompt engineering for knowledge graph question answering". In: *Frontiers in Artificial Intelligence* Volume 7 - 2024 (2025). ISSN: 2624-8212. DOI: 10.3389/frai.2024.1454258. URL: https://www.frontiersin.org/journals/artificial-intelligence/articles/10.3389/frai.2024.1454258.

[22] Jinhyuk Lee et al. "BioBERT: a pre-trained biomedical language representation model for biomedical text mining". In: *CoRR* abs/1901.08746 (2019). arXiv: 1901.08746. URL: http://arxiv.org/abs/1901.08746.

[23] Weihong Lin et al. "ViBERTgrid: A Jointly Trained Multi-Modal 2D Document Representation for Key Information Extraction from Documents". In: *CoRR* abs/2105.11672 (2021). arXiv: 2105.11672. URL: https://arxiv.org/abs/2105.11672.

[24] Garima Malik et al. "Named Entity Recognition on Software Requirements Specification Documents". In: *Canadian AI 2021*. https://caiac.pubpub.org/pub/v0la26hg. Canadian Artificial Intelligence Association (CAIAC). June 2021.

[25] Giuliana Maltempo et al. "Multi-Word Entity Extraction And Rich Relationship Identification To Derive Conceptual Models From Natural Language Specifications". In: (2024). DOI: 10.29327/1407529.27-5.

[26] Debora Maria Marettek. "Elicitation and Classification of Security Requirements for EVerest". Master's Thesis, supervised by M.Sc. Sophie Corallo and Dr.-Ing. Tobias Hey. First examiner: Prof. Dr.-Ing. Anne Koziolek, Second examiner: Prof. Dr. Ralf Reussner. MA thesis. KIT Department of Informatics, KASTEL – Institute of Information Security and Dependability, May 2024.

[27] Marc Mültin. "ISO 15118 as the Enabler of Vehicle-to-Grid Applications". In: *2018 International Conference of Electrical and Electronic Technologies for Automotive*. 2018, pp. 1–6. DOI: 10.23919/EETA.2018.8493213.

[28] Humza Naveed et al. *A Comprehensive Overview of Large Language Models*. 2024. arXiv: 2307.06435 [cs.CL]. URL: https://arxiv.org/abs/2307.06435.

[29] Michael Oliver and Guan Wang. *Crafting Efficient Fine-Tuning Strategies for Large Language Models*. 2024. arXiv: 2407.13906 [cs.CL]. URL: https://arxiv.org/abs/2407.13906.

[30] OpenAI et al. *GPT-4 Technical Report*. 2024. arXiv: 2303.08774 [cs.CL]. URL: https://arxiv.org/abs/2303.08774.

[31] Kalyani Pakhale. *Comprehensive Overview of Named Entity Recognition: Models, Domain-Specific Applications and Challenges*. 2023. arXiv: 2309.14084 [cs.CL]. URL: https://arxiv.org/abs/2309.14084.

[32] Sven Peldszus et al. "Can I Check What I Designed? Mapping Security Design DSLs to Code Analyzers". In: to be published.

[33] Yanzhao Qin et al. *SysBench: Can Large Language Models Follow System Messages?* 2024. arXiv: 2408.10943 [cs.CL]. URL: https://arxiv.org/abs/2408.10943.

[34] Matthew Renze. "The Effect of Sampling Temperature on Problem Solving in Large Language Models". In: *Findings of the Association for Computational Linguistics: EMNLP 2024*. Association for Computational Linguistics, 2024, pp. 7346–7356. DOI: 10.18653/v1/2024.findings-emnlp.432. URL: http://dx.doi.org/10.18653/v1/2024.findings-emnlp.432.

[35] Ralf Reussner et al. *The Palladio Component Model*. Tech. rep. 14. Karlsruher Institut für Technologie (KIT), 2011. 193 pp. DOI: 10.5445/IR/1000022503.

[36] Pranab Sahoo et al. *A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications*. 2024. arXiv: 2402.07927 [cs.AI]. URL: https://arxiv.org/abs/2402.07927.

[37] S. Seifermann et al. "A unified model to detect information flow and access control violations in software architectures". In: *Proceedings of the 18th International Conference on Security and Cryptography, SECRYPT 2021, Virtual, Online, 6 July 2021 - 8 July 2021*. 18th International Conference on Security and Cryptography. 2021 (Online, July 6–8, 2021). 46.23.01; LK 01. SciTePress, 2021, pp. 26–37. ISBN: 978-9897585241. DOI: 10.5220/0010515300260037.

[38] Sonish Sivarajkumar et al. *An Empirical Evaluation of Prompting Strategies for Large Language Models in Zero-Shot Clinical Natural Language Processing*. 2023. arXiv: 2309.08008 [cs.CL]. URL: https://arxiv.org/abs/2309.08008.

[39] Fuyu Tian et al. "The Impact of Traceability on Software Maintenance and Evolution: A Mapping Study". In: *Journal of Software: Evolution and Process* 33.10 (2021), e2374. DOI: 10.1002/smr.2374. URL: https://doi.org/10.1002/smr.2374.

[40] Archana Tikayat Ray et al. "Development of a Language Model for Named-Entity-Recognition in Aerospace Requirements". In: *Journal of Aerospace Information Systems* 21.6 (2024), pp. 489–499. DOI: 10.2514/1.I011251. eprint: https://doi.org/10.2514/1.I011251. URL: https://doi.org/10.2514/1.I011251.

[41] "What Is EVerest — EVerest documentation". In: (). URL: https://everest.github.io/nightly/.