Recovering Trace Links In Software Architecture Documentation

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Jan Keim

aus Stuttgart

Tag der mündlichen Prüfung: 03. Februar 2025
Erste Gutachterin: Prof. Dr.-Ing. Anne Koziolek
Zweiter Gutachter: Prof. Dr. Andreas Vogelsang

Abstract

Introduction Software architecture is a critical element of software engineering. It influences the development, maintenance, and evolution of software systems and contributes significantly to their success. Knowledge about the architecture and the underlying Architectural Design Decisions (ADDs) can further improve good architecture and prevent fast software aging and erosion. ADDs play a pivotal role in shaping system architectures, yet existing taxonomies and frameworks for categorizing these decisions are often coarse, hindering detailed analyses. Architecture knowledge and ADDs are often documented in Software Architecture Documentation (SAD), which can have different forms. A common form is informal natural language text due to its approachable nature for communication. Another form is Software Architecture Models (SAMs), which are formal approaches like UML component diagrams or the Palladio component model. Besides communication, these can also be used to evaluate quality attributes like response times at design time. Architects must combine information from these different kinds of artifacts to gain a comprehensive understanding and manage their complex interrelationships. Trace links make the relationships between artifacts explicit, improving software quality and simplifying various activities such as impact analysis, change management, maintenance, and evolution. Thus software traceability is a key capability for successfully developing software. Traceability Link Recovery (TLR) is the process that allows software engineers to create these trace links between any uniquely identifiable entities in software engineering artifacts. However, manually creating and maintaining these trace links is time-consuming and error-prone. Automating TLR can make TLR viable and economically feasible. However, no approaches looked into TLR for software architecture, linking SADs and SAMs. One concrete application of trace links in software architecture is Inconsistency Detection (ID) between the artifacts. While not all inconsistencies are inherently problematic, unknown and unaddressed ones can affect, e.g., project understanding, negating the benefits of these artifacts. Two forms of inconsistencies between SADs and SAMs are Undocumented Model Elements (UMEs) and Missing

Model Elements (MMEs). UMEs are elements that are present in the SAM but not documented in the SAD. MMEs are architecturally relevant entities in the SAD that are not modeled in the SAM.

Contributions In this dissertation, I first explore ADDs in SADs, creating a fine-grained taxonomy. Using this taxonomy, I present an approach for the automated classification of ADDs. I then propose various approaches for automating Traceability Link Recovery (TLR) between various software artifacts, including SADs, SAMs, and code. First, I present the approach ArDoCo for TLR between SAD and SAM, which uses heuristics to detect architecture entities in SAD and compare them to entities in SAM. Then, I present the heuristic-based approach ArCoTL for TLR between SAM and code that combines heuristics and filters in a computational graph. The third TLR approach, TransArC, transitively bridges the semantic gap between SAD and code by combining the other two approaches. The last major contribution focuses on Inconsistency Detection (ID), specifically addressing UMEs and MMEs. It uses the recovered trace links and other intermediate TLR results to identify inconsistencies between SADs and SAMs. Additionally, the dissertation investigates the application of simple approaches based on large language models (LLMs) and compares their performance to the other approaches.

Results In the evaluation of the taxonomy for ADDs, I argue about its purpose and structure and demonstrate its application with a user study. The evaluation for the automated classification of ADDs shows mixed results. On the one hand, the results are promising with improvements over classical approaches, achieving an average F_1-score of 92% for identifying ADDs and average F_1-scores of 55% for multi-class and 50% for multi-label classification of the taxonomy's leaf-classes. On the other hand, these results require significant improvements for actual application. The evaluations of the TLR approaches use a benchmark dataset and compare the results to baseline approaches. The approach ArDoCo for TLR between SAD and SAM achieves an excellent average F_1-score of 82% with 87% precision and 82% recall. With an increased F_1-score of 26%, ArDoCo significantly outperforms the other baselines. The approach also outperforms the best version of the simple LLM-based approach that achieves an average F_1-score of 64%. The approach ArCoTL for TLR between SAM and code yields near-perfect results with an average F_1-score of 98%, significantly outperforming the baseline. The transitive approach TransArC for TLR between SAD and code using SAMs as intermediate artifacts achieves an average F_1-score of 82% with 87% precision and 81% recall. TransArC also significantly outperforms the baselines. This demonstrates that the intermediate artifact helps in bridging the semantic gap. The evaluation of the ID shows excellent results for detecting UMEs with an average F_1-score of 95%. The results for identifying MMEs are promising, with an average F_1-score of up to 49% in the best configuration, significantly outperforming the baseline. The simple LLM-based approach achieves mixed results that are equal or worse than the heuristic-based approach. For example, providing the ground truth trace links helps identify UMEs but reduces performance for MMEs.

Conclusion Overall, the evaluation results demonstrated the efficacy of the proposed approaches, which resulted in significant improvements over baseline methods. The findings highlight the potential of automated TLR and ID to enhance software architecture consistency, improve maintenance efficiency, and support better architectural decision-making throughout the software lifecycle. Future work should focus on improving and extending the approaches in different ways. A first priority is improving ID for MMEs by enhancing precision in identifying architecturally relevant entities, potentially by integrating LLMs into the heuristic-based methods. A second priority is advancing automated ADD classification. A reliable classification approach can enable the automated selection of inconsistency-checking approaches and new forms of inconsistency detection. A third priority is further exploring transitive TLR approaches. Future work should investigate the application of transitive TLR to additional artifact types and leverage architecture recovery methods to generate missing intermediate artifacts. These directions aim to refine the current approaches and broaden their applicability.

Zusammenfassung

Einleitung Softwarearchitektur ist ein kritisches Element der Softwaretechnik. Sie beeinflusst die Entwicklung, Wartung und Weiterentwicklung von Softwaresystemen und trägt wesentlich zu ihrem Erfolg bei. Wissen über die Architektur und die zugrunde liegenden Architekturellen Entwurfsentscheidungen (ADDs) kann dazu beitragen, eine gute Architektur weiter zu verbessern und eine schnelle Alterung und Erosion der Software zu verhindern. ADDs spielen eine entscheidende Rolle bei der Gestaltung von Systemarchitekturen, doch bestehende Taxonomien und Rahmenwerke zur Kategorisierung dieser Entscheidungen sind oft grobgranular, was detaillierte Analysen erschwert. Architekturelles Wissen und ADDs werden häufig in der Softwarearchitekturdokumentation (SAD) festgehalten, die unterschiedliche Formen annehmen kann. Eine gängige Form ist informeller Text in natürlicher Sprache, da diese Form der Kommunikation allen leicht zugänglich ist. Eine andere Form sind Softwarearchitekturmodelle (SAMs), die formale Ansätze wie UML-Komponentendiagramme oder das Palladio-Komponentenmodell nutzen. Neben der Kommunikation können diese Modelle auch zur Bewertung von Qualitätsmerkmalen wie Antwortzeiten in der Entwurfsphase verwendet werden. Architekt:innen müssen Informationen aus diesen unterschiedlichen Artefakten kombinieren, um ein umfassendes Verständnis zu erlangen und die komplexen Zusammenhänge zwischen ihnen zu verwalten. Verfolgbarkeitsverbindungen machen die Beziehungen zwischen den Artefakten explizit, verbessern die Softwarequalität und erleichtern Aktivitäten wie Auswirkungsanalyse, Änderungsmanagement, Wartung und Weiterentwicklung. Daher ist Software-Verfolgbarkeit eine Schlüsselkompetenz für die erfolgreiche Entwicklung von Software. Die Wiederherstellung von Verfolgbarkeitsverbindungen (Traceability Link Recovery, TLR) ist der Prozess, der es Softwareingenieur:innen ermöglicht, diese Verfolgbarkeitsverbindungen zwischen eindeutig identifizierbaren Entitäten in Softwareartefakten zu erstellen. Die manuelle Erstellung und Pflege dieser Verbindungen ist jedoch zeitaufwendig und fehleranfällig. Eine Automatisierung des TLR kann dessen Durchführbarkeit und Wirtschaftlichkeit erheblich steigern. Allerdings wurden bisher keine

Ansätze für TLR in der Softwarearchitektur untersucht, die SADs und SAMs miteinander verbinden. Eine konkrete Anwendung von Verfolgbarkeitsverbindungen in der Softwarearchitektur ist die Erkennung von Inkonsistenzen (Inconsistency Detection, ID) zwischen Artefakten. Während nicht alle Inkonsistenzen von Natur aus problematisch sind, können unbekannte und ungeklärte Inkonsistenzen das Projektverständnis beeinträchtigen und die Vorteile dieser Artefakte negieren. Zwei Formen von Inkonsistenzen zwischen SADs und SAMs sind Undokumentierte Modellelemente (Undocumented Model Elements, UMEs) und Fehlende Modellelemente (Missing Model Elements, MMEs). UMEs sind Elemente, die im SAM vorhanden, aber nicht im SAD dokumentiert sind. MMEs sind architekturell relevante Entitäten im SAD, die nicht im SAM modelliert sind.

Beiträge In dieser Dissertation untersuche ich zunächst ADDs in SADs und entwickle eine feingranulare Taxonomie. Mithilfe dieser Taxonomie präsentiere ich einen Ansatz zur automatisierten Klassifikation von ADDs. Anschließend schlage ich verschiedene Ansätze zur Automatisierung von TLR zwischen den verschiedenen Softwareartefakten SADs, SAMs und Code vor. Zunächst stelle ich den Ansatz ArDoCo für TLR zwischen SAD und SAM vor, der Heuristiken verwendet, um Architekturelemente im SAD zu erkennen und mit den Elementen im SAM zu vergleichen. Anschließend präsentiere ich den heuristikbasierten Ansatz ArCoTL für TLR zwischen SAM und Code, der Heuristiken und Filter in einem Rechenfluss kombiniert. Der dritte TLR-Ansatz, TransArC, überbrückt transitiv die semantische Lücke zwischen SAD und Code, indem er die anderen beiden Ansätze kombiniert. Der letzte Hauptbeitrag konzentriert sich auf die Inkonsistenzerkennung von UMEs und MMEs. Dabei werden die wiederhergestellten Verfolgbarkeitsverbindungen und andere Zwischenergebnisse des TLR genutzt, um Inkonsistenzen zwischen SADs und SAMs zu identifizieren. Zusätzlich untersucht die Dissertation den Einsatz einfacher Ansätze, die auf großen Sprachmodellen (LLMs) basieren und vergleicht deren Leistung mit den anderen Ansätzen.

Ergebnisse In der Evaluierung der Taxonomie für ADDs argumentiere ich über deren Zweck und Struktur und demonstriere ihre Anwendung in einer Benutzerstudie. Die anschließende Evaluierung der automatisierten Klassifikation von ADDs zeigt gemischte Ergebnisse. Einerseits sind die Ergebnisse vielversprechend und zeigen Verbesserungen gegenüber klassischen Ansätzen mit einem durchschnittlichen F_1-Wert von 92 % für die Identifikation von ADDs sowie durchschnittlichen F_1-Werten von 55 % für die Multiklassen- und 50 % für die Multilabel-Klassifikation der Blattklassen der Taxonomie.

Andererseits erfordern diese Ergebnisse erhebliche Verbesserungen für die praktische Anwendung. Die Evaluierungen der TLR-Ansätze nutzen einen Benchmark-Datensatz und vergleichen die Ergebnisse mit anderen Vergleichsansätze. Der Ansatz ArDoCo für TLR zwischen SAD und SAM erzielt einen hervorragenden durchschnittlichen F_1-Wert von 82 % mit 87 % Präzision und 82 % Ausbeute. Mit einem um 26 % erhöhten F_1-Wert übertrifft ArDoCo die anderen Ansätze signifikant. Der Ansatz übertrifft auch die beste Version des einfachen, auf LLMs basierenden Ansatzes, der einen durchschnittlichen F_1-Wert von 64 % erreicht. Der Ansatz ArCoTL für TLR zwischen SAM und Code liefert nahezu perfekte Ergebnisse mit einem durchschnittlichen F_1-Wert von 98 % und übertrifft die Vergleichsansätze deutlich. Der transitive Ansatz TransArC für TLR zwischen SAD und Code unter Verwendung von SAMs als Zwischenartefakte erzielt einen durchschnittlichen F_1-Wert von 82 % mit 87 % Präzision und 81 % Ausbeute. TransArC übertrifft die Vergleichsansätze signifikant, z. B. um 45 Prozentpunkte im durchschnittlichen F_1-Wert gegenüber dem nächstbesten Ansatz. Dies zeigt, dass das Zwischenartefakt hilft, die semantische Lücke zu überbrücken. Die Evaluierung der ID zeigt hervorragende Ergebnisse bei der Erkennung von UMEs mit einem durchschnittlichen F_1-Wert von 95 %. Die Ergebnisse für die Identifikation von MMEs sind vielversprechend und erreichen in der besten Konfiguration einen durchschnittlichen F_1-Wert von bis zu 49 %, womit sie die Vergleichsansätze deutlich übertreffen. Der einfache, auf LLMs basierende Ansatz liefert gemischte Ergebnisse, die gleichwertig oder schlechter als der heuristikbasierte Ansatz sind. Beispielsweise hilft die Bereitstellung der Verfolgbarkeitsverbindungen aus dem Goldstandard bei der Identifikation von UMEs im gleichen Maße wie beim nicht-LLM Ansatz, reduziert jedoch die Leistung bei MMEs.

Fazit Insgesamt haben die Evaluierungsergebnisse die Wirksamkeit der vorgeschlagenen Ansätze gezeigt, die zu erheblichen Verbesserungen gegenüber den Vergleichsansätzen führten. Die Ergebnisse unterstreichen das Potenzial von automatisiertem TLR und ID, um die Konsistenz der Softwarearchitektur zu verbessern, die Wartungseffizienz zu erhöhen und eine bessere architekturelle Entscheidungsfindung während des gesamten Softwarelebenszyklus zu unterstützen. Zukünftige Arbeiten sollten darauf abzielen, die Ansätze in verschiedener Hinsicht zu verbessern und zu erweitern. Eine erste Priorität ist die Verbesserung der ID für MMEs durch die Steigerung der Präzision bei der Identifikation architekturell relevanter Entitäten, möglicherweise durch die Integration von LLMs und anderen modernen Verfahren in die heuristikbasierten Methoden. Eine zweite Priorität ist die Weiterentwicklung der

automatisierten ADD-Klassifikation. Ein zuverlässiger Klassifikationsansatz kann die automatisierte Auswahl von Ansätzen zur Inkonsistenzprüfung und neue Formen der Inkonsistenzerkennung ermöglichen. Eine dritte Priorität ist die weitere Erforschung transitiver TLR-Ansätze. Künftige Arbeiten sollten die Anwendung transitiver TLR-Ansätze auf zusätzliche Artefakttypen untersuchen und Architektur-Wiederherstellungsmethoden nutzen, um fehlende Zwischenartefakte zu generieren. Diese Richtungen zielen darauf ab, die aktuellen Ansätze zu verfeinern und ihre Anwendbarkeit zu erweitern.

Contents

List of Figures

List of Tables

Acronyms

ArDoCo Architecture Documentation Consistency

ArCoTL ARchitecture-to-COde Trace Linking

TransArC Transitive links for Architecture and Code

STD Simple Tracelink Discovery

NoRBERT Non-functional and functional Requirements classification using BERT

UME Unmentioned Model Element

MME Missing Model Element

TLR Traceability Link Recovery

ID Inconsistency Detection

SAD Software Architecture Documentation

ADR Architecture Decision Record

OSS Open Source Software

NLP Natural Language Processing

IR Information Retrieval

ML Machine Learning

LLM Large Language Model

POS Part-of-Speech

WSD Word Sense Disambiguation

SAM Software Architecture Model

NER Named Entity Recognition

UML Unified Modeling Language

PCM Palladio Component Model

ADD Architectural Design Decision

RAG Retrieval Augmented Generation

GQM Goal-Question-Metric

MCC Matthews correlation coefficient

MAP Mean Average Precision

AP	Average Precision
JSD	probabilistic Jensen Shannon model Divergence
VSM	Vector Space Model
LDA	Latent Dirichlet Allocation
LSI	Latent Semantic Indexing
WMD	Word Mover's Distance
TFIDF	Term Frequency-Inverse Document Frequency
ANN	Artificial Neural Network
DNN	Deep Neural Network
RNN	Recurrent Neural Network
LSTM	Long short-term memory
SVM	Support Vector Machine
NB	Naïve Bayes
LR	Logistic Regression
RF	Random Forest

Part I.

Prologue

1. Introduction

Software engineering is the application of engineering to software, as the ISO/IEC/IEEE 24765 standard [109] and the Software Engineering Body of Knowledge (SWEBOK) [106] state. Thus, software engineering is the systematic, disciplined approach to developing, operating, and maintaining software. Software engineers have to focus in a disciplined manner on both product quality and process efficiency to deliver maintainable, high-quality software within time and budget constraints [2].

A central activity in software engineering is architecting the software, defined as "conceiving, defining, expressing, documenting, communicating, certifying proper implementation of, maintaining and improving an architecture throughout a system's life cycle" [110]. As noted by Medvidovic et al. [181], a robust software architecture facilitates well-engineered software systems, directly contributing to a system's success. According to them, this is due to good software architecture improving the system's development, maintenance, and evolution. Good software architecture facilitates development, maintenance, and evolution and influences many quality attributes such as performance, security, reliability, and usability [16].

Within my dissertation, I look into software architecture, particularly its documentation. For this, I will motivate the different ideas and approaches I will look into in the following sections. In Section 1.1, I introduce Architectural Design Decisions (ADDs) and will motivate my research concerning the kinds of ADDs in Software Architecture Documentation (SAD) and how to organize them in a taxonomy. Using the taxonomy and insights from its creation, I can then tackle Traceability Link Recovery (TLR) for software architecture in Section 1.3. Before I introduce TLR, I first introduce the different forms of SAD in Section 1.2. Then page 6 motivates why I plan to create approaches for TLR in software architecture, linking SAD, Software Architecture Models (SAMs), and code. Having trace links and using TLR enables different tasks, with one application being the detection of inconsistencies. In Section 1.4, I introduce

Inconsistency Detection (ID) and motivate my approach to detect certain inconsistencies based on TLR. I provide a concrete example for my approaches in Section 1.5 before I summarize my contributions in Section 1.6 and give an overview of the remainder of this dissertation in Section 1.7.

1.1. Architectural Design Decisions

During the design process, software architects face numerous critical ADDs that significantly impact both the software and its development process [106]. Consequently, decision-making is foundational to architectural design, often driven by the need to balance and optimize competing quality attributes, impacting everything from performance and scalability to maintainability and security. As software architecture comprises such non-trivial ADDs, their documentation is crucial for improved system evolution [4]. Documenting and preserving all the knowledge generated in the architectural design, including the rationale behind ADDs, in a Software Architecture Documentation (SAD) (see also [104, 109]) strengthens the advantages of a good software architecture and mitigates risks of rapid deterioration and software aging [208]. As Parnas emphasized, the success of a software system significantly depends on the quality of its documentation [209]. For example, according to Xia et al. [291], developers spend on average 58% of their time on comprehension, where SAD can support developers by reducing some of the required effort and, thus increasing the overall quality of the software.

There are existing taxonomies and ontologies that categorize these ADDs like Kruchten's ontology [146], the architectural design decision model by Jansen & Bosch [112], or the architectural design decision rationale framework by Falessi et al. [59]. However, these taxonomies are broad and abstract, complicating detailed analyses of the SADs. Based on this shortcoming, I phrase my first research question:

RQ 1 What kinds of fine-grained Architectural Design Decisions (ADDs) can be found in Software Architecture Documentations (SADs)?

The taxonomy can be used to enable or improve other tasks like automated TLR or automated ID that I present below. However, an automated approach is required to classify these design decisions. Yet, there is limited research in this direction. Consequently, I developed an approach for automated classification of ADDs in SAD. The accompanying research question is the following:

RQ 2 How well can my approach automatically classify these ADDs?

1.2. Software Architecture Documentation

To document ADDs, there are different forms of SADs. A common form is natural language text, due to its ease of use and flexibility. Slightly more structured natural language SADs include Architecture Decision Records (ADRs) [122, 32]. Each ADR file typically captures a single design decision and its rationale in natural language.

Templates and models also exist to help software architects structure and document the architecture. For example, the arc24 method [251] offers a systematic but flexible template to document ADRs. Similarly, ISO/IEC/IEEE 42010 [110] provides a semi-formal framework that suggests information items for ADRs and highlights key considerations for architects in identifying architectural decisions.

Other forms of documentation include formal Software Architecture Models (SAMs) and informal artifacts such as sketched diagrams of the software architecture. For example, the C4 model [27] provides a semi-formal approach for structuring and especially visualizing software architecture at four levels of abstraction: Context, Containers, Components, and Code. In addition, architects can use more formal SAMs like the Unified Modeling Language (UML) [44] and the Palladio Component Model (PCM) [231] to document, simulate, and evaluate the system at design time, assessing quality attributes such as performance.

In my dissertation, I will focus on natural language SAD, and throughout this work, I will use the term *SAD* to refer exclusively to natural language-based documentation. In contrast, I will use the term *SAM* for documentation through architecture models, such as those created with UML or PCM. Specifically, I refer to structural component diagrams, i.e., the system's development view (cf. Kruchten's 4+1 view model [145]) as structural architecture models. I will also use the term *formal artifacts* in this dissertation for code and architecture models like UML or PCM. The term *informal artifacts* is used for artifacts like SADs, sketches, and similar.

1.3. Traceability Link Recovery

Besides software architecture and its documentation, engineering and developing software involves the creation of various artifacts. These artifacts can include, among others, requirements specifications, low-level design documents, source code, test cases, and user manuals. Software development artifacts like these can cover different aspects of a system, just like SADs and SAMs do. For example, requirements specifications describe what the system should do, SADs and other design documents outline how the system will meet those requirements, source code implements the design, and test cases validate that the code works as expected. Consequently, the artifacts exist at different levels of abstraction, ranging from high-level business requirements to low-level implementation details.

According to Olsson and Grundy [200], software developers face various challenges due to the increasing complexity and volume of information related to software systems. As software projects grow, effectively managing interrelationships among various software artifacts, such as requirements, use cases, and test plans, becomes critical. The problem is that the relationships between these artifacts are not always apparent, and thus, software developers cannot fully utilize them. Making these relationships explicit improves software quality and simplifies processes such as impact analysis, change management, and system maintenance [168, 39, 84]. Thus, software traceability is a key capability for successfully developing software.

There are various examples where traceability is essential for successful software development. One example is the aerospace industry, specifically in developing avionics systems. In the aerospace industry, safety-critical software must meet stringent regulatory standards, such as DO-178C [296], which require complete traceability from requirements to code to tests. Traceability allows organizations to verify that every requirement is implemented and tested thoroughly. If an issue arises in an avionics system, traceability helps engineers quickly track down which parts of the code are affected by specific requirements or tests. Moreover, according to DO-178C [296], traceability should ensure that all safety-critical requirements are validated and verified, supporting the testing and review process needed to prevent any gaps in functionality that could lead to catastrophic failures. Avionics software often has a long lifecycle with many updates and modifications over time. Traceability helps maintain consistency and accuracy across updates, which is essential for systems expected to operate reliably for decades.

The automotive industry is another example of software traceability being critical. Similarly to the aerospace industry, automotive safety standards like ISO 26262 [107] require traceability throughout the development lifecycle for similar reasons. Automotive software must handle complex interactions between different vehicle systems. Traceability helps engineers perform impact analysis to assess potential failure points [60, 12], especially when components are reused or modified. The lack of trace links can cause issues, as Fucci et al. [70] show in their experience report. The company under investigation struggled with its product's quality due to the lack of trace links, and this lack of trace links hindered the investigation of this phenomenon.

Traceability Link Recovery (TLR) is the process that allows software engineers to create explicit trace links between any uniquely identifiable software engineering artifacts. This process helps maintain these trace links and enables engineers to use the resulting network to gain valuable insights into the software product and its development lifecycle [42]. Thus, creating and maintaining trace links can improve the software quality [229].

For example, Raţiu et al. state that TLR can help in synchronizing artifacts and keeping them consistent in collaborative software development [228]. Additionally, TLR can support and improve many critical software engineering tasks [40]. Examples of these tasks include the improved efficiency of software maintenance [167, 168]. This can be done by, e.g., lowering the effort and improving knowledge transfer and onboarding. Trace links can reduce the manual effort required to track and understand the impact of changes, improve consistency, and support quicker bug resolution. By making the maintenance process more efficient, TLR contributes to higher software quality, faster delivery, and more sustainable development over time. It can also support tasks like impact analysis and bug tracking, which are vital to software maintenance. For instance, Rath et al. [226] use TLR to localize bugs by relating requirements, bug reports, and source code files, including historical project data. For change impact analysis, Falessi et al. [60] also use trace links between requirements and source code and historical data to identify the classes that are impacted by a new requirement. TLR is also used in various approaches for system security (see [193, 191]), and trace links are used to demonstrate the safety of systems (see [192, 169, 178]). As discussed above, some standards even mandate traceability, such as the ISO26262 [107] about the functional safety of road vehicles or DO-178C [296] for avionics systems.

However, software engineers rarely create trace links between artifacts directly. Despite all the benefits, the main drawback of traceability is the time-consuming and error-prone process of manually creating and maintaining trace links [225, 56]. Further, incomplete knowledge can make it hard for the developers to link all relevant artifacts properly. The problems stem from the semantic gap between artifacts of different abstraction levels [22], e.g., requirements and code. In their survey, Ruiz et al. [234] identify further barriers to TLR in practice, including high costs, low perceived benefit, lack of guidance, and struggles with maintaining trace links. The authors suggest that addressing these barriers requires a multifaceted approach that includes improving organizational practices, enhancing tool usability, and fostering a culture that values traceability. According to the grand challenge of traceability by Gotel et al. [84], TLR needs to be purposed, cost-effective, configurable, trusted, scalable, portable, valued, and ubiquitous.

One way to improve traceability and make it more viable and economically feasible is to automate TLR. Many approaches have looked into automated TLR, but many approaches only look at trace links between requirements and code. However, as discussed before, software architecture is key to successfully developing, maintaining, and evolving a software system. Yet, few to no approaches look into TLR for software architecture (see also the related work in Section 3.2). Still, linking SADs to other software artifacts, such as other design artifacts or code, is beneficial for the above reasons. Further, according to the investigation by Ali et al. [5] about the current state of architecture consistency practices, the perceived high effort required to map systems to their intended architectures is one of the identified barriers. Here, automated TLR for software architecture can support architects by lowering the overall effort.

Consequently, I tackle TLR for software architecture in this dissertation. For this, I focus on three approaches for three kinds of artifacts (see Figure 1.1) commonly used by software architects: SADs, SAMs, and code. These artifacts and their relationships are exemplarily displayed in Section 1.5.

First, there is TLR between SAD and SAM that my approach ArDoCo tackles. Trace links between these artifacts can then help the architects with consistency between artifacts, support maintenance, facilitate auditing, and improve knowledge transfer and onboarding.

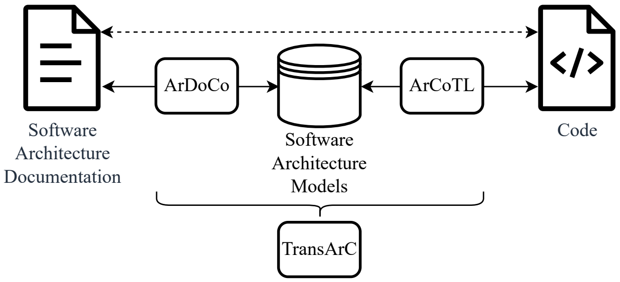

Software Architecture Documentation
ArDoCo
Software Architecture Models
ArCoTL
Code
TransArC

Figure 1.1.: Different kinds of artifacts with different approaches (in boxes with round corners) for TLR.

Then, I look into TLR between SAM and code that my approach, ARchitecture-to-COde Trace Linking (ArCoTL), tries to solve. Links between these artifacts can help architects check for architecture conformance, manage architectural drift, and facilitate impact analysis for code changes.

Third, I look into TLR between SAD and code. Because different types of artifacts like these cover different aspects and have different views on the system, these artifacts have different levels of abstraction, resulting in a semantic gap between them [22]. Automated approaches need to bridge this gap by capturing the underlying semantics. This task is inherently challenging, and approaches are prone to misinterpretation, leading to potential imprecision. Consequently, there is the idea to use intermediate artifacts that have a smaller semantic gap to the artifacts, allowing for easier pairing as some approaches have already suggested (cf. [194, 274, 191, 12]). My approach, Transitive links for Architecture and Code (TransArC), aims to bridge the big semantic gap between documentation and code using SAMs as they are usually semantically between SADs and code.

Lastly, Large Language Models (LLMs) are nowadays prominently present due to their capacity for understanding and interpreting complex, context-rich information. LLMs might be able to bridge the gap between, e.g., SADs and SAMs by interpreting architectural intent from text and aligning it with structured model elements, even when there are ambiguities or indirect connections. To assess whether a simple LLM-based approach can outperform ArDoCo, I also briefly explore the application of LLMs to this TLR problem.

Based on these four main focuses for automated TLR for software architecture, the research questions for TLR in software architecture are the following:

RQ 3 How well can trace links be automatically recovered between the different software architecture artifacts?

- **RQ 3.1** How well can my approach ArDoCo recover trace links between SAD and SAM, also compared to baselines?
- **RQ 3.2** How well can my approach ArCoTL recover trace links between SAM and code, also compared to baselines?
- **RQ 3.3** How well can my transitive approach TransArC recover trace links between SAD and code, also compared to baselines?
- **RQ 3.4** How does a simple approach based on LLMs compare to the other approaches for TLR between SAD and SAM?

1.4. Inconsistency Detection

An essential application of trace links in software architecture, as underlined by Ali et al. [5], is checking the conformance of architecture. Inconsistency Detection (ID) and maintaining consistency in software architecture can be challenging due to the diverse artifacts and documentation types required to capture various aspects of a project. These artifacts are often created and modified at different times and with varying levels of detail and formality [196]. According to Wohlrab et al., [290], common inconsistencies in these documents include mismatched specifications, contradictory rules and constraints, non-aligned patterns and guidelines, and inconsistent wording across artifacts.

While not all inconsistencies are inherently problematic [197], unaddressed ones can result in issues that affect project understanding and execution. Once an inconsistency is detected, developers can either address or tolerate it. Undetected inconsistencies, however, remove this choice, potentially leading to the erosion of key information and negating the benefits of these artifacts, particularly if they become outdated or contradictory [128].

This dissertation explores using TLR for automated ID within architectural documentation. Specifically, it focuses on two types of inconsistencies:

Unmentioned Model Elements (UMEs) and *Missing Model Elements (MMEs)*. *UMEs* are elements within the SAM, such as components, that are not documented in the SAD. When such elements lack documentation, critical information like their responsibilities or the design rationales behind them remains uncaptured. *MMEs* are inconsistencies that occur when the SAD includes architecturally significant elements that are absent in the SAM. This may happen, for example, when the SAD is prescriptive and describes components not yet implemented in the model. Outdated documentation, especially following refactoring or significant model changes, can also contribute to MME cases. Both kinds of inconsistencies are demonstrated in the example in Section 1.5. By applying trace links to detect and manage these inconsistencies, this work aims to enhance architectural consistency, preserving the reliability and utility of SADs throughout a project's lifecycle.

To detect these inconsistencies, I look into two adaptations of ArDoCo (Architecture Documentation Consistency). First, I plan to utilize the results of Architecture Documentation Consistency (ArDoCo), the trace links between SADs and SAMs, to detect UMEs. Second, I look into extending the ArDoCo and use its intermediate results to detect MMEs. Further, in the same fashion as for TLR, I want to explore the simple application of LLMs for this problem. Thus, the associated research questions are the following:

RQ 4 How well can inconsistencies between SADs and SAMs be automatically detected using trace links?

- **RQ 4.1** How well can my approach utilizing trace links identify Unmentioned Model Elements (UMEs)?
- **RQ 4.2** How well can my approach using results from TLR identify Missing Model Elements (MMEs)?
- **RQ 4.3** How does a simple approach based on LLMs compare?

1.5. Example

In this section, I want to introduce an example that serves multiple purposes. First, I want to introduce the different kinds of artifacts exemplarily. Secondly, the running example should demonstrate the relations between the artifacts that should be captured by trace links. Lastly, I want to highlight the inconsistencies between the artifacts, focusing on UMEs and MMEs. Figure 1.2 shows the example.

Figure 1.2a displays the SAD. The documentation contains different ADDs about the system. For example, the first sentence mentions the layered architecture pattern. The other sentences contain information about the system's structural architecture, including the existing components and their relationships. The third sentence highlights one of the difficulties of natural language in containing the coreference "it" that belongs to the controller in the second sentence. Overall, the SAD details the respective responsibilities.

Figure 1.2b shows a simple SAM. It displays the structural component diagram containing three components and their relationships in the form of required interfaces and provided interfaces (in lollipop notation). There are three key components: the *Controller* on the left, the *DataPersistence* on the bottom right, and the *Cache* that is on the top right, acting between the other two components.

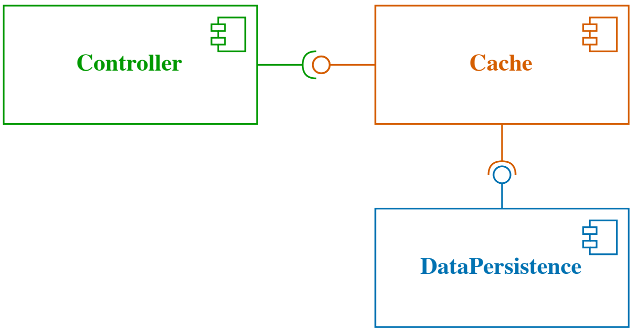
Lastly, Figure 1.2c is an abstract representation of the code. The code introduces two packages, namely `service` and `dataaccess`, each of which includes sub-packages such as `auth` and `preferences`. The `Controller` class employs the `Authenticator` to verify incoming requests, as mentioned in the SAD. The classes `Products` and `Users` serve as repositories, leveraging information from hidden classes within the `preferences` package to establish connections with a database.

The elements in the three figures are highlighted to show the different components. Elements that share the same color across the different figures, i.e., the different artifacts, must be linked via trace links. As such, the second and third sentences are associated with the *Controller* component and the classes within the `service` package, due to the references to the terms "controller" and "it". Similarly, there are trace links between the third sentence, the *DataPersistence* component, and the classes in the `dataaccess` package.

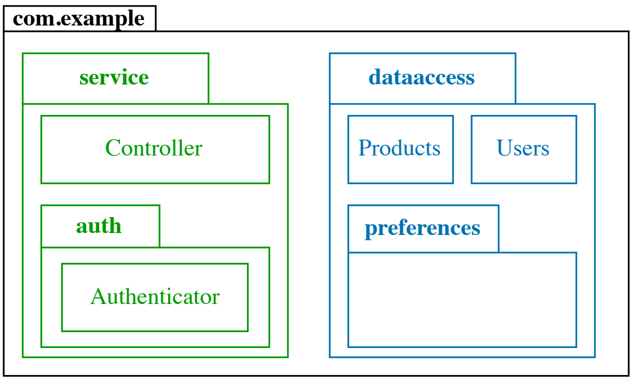
This simple example highlights a few challenges for TLR. The description of the *controller* component is necessary to properly link it to the whole `service` package instead of only the `Controller` class. Another challenge is the different naming of the "persistence" component in the SAD to the `dataaccess` package in the code. Using the *DataPersistence* component in the SAM can help bridge the gap as it is closer to the other two.

The system adheres to layered architecture.
The **controller** receives incoming requests and verifies them.
Then, **it** answers requests querying the **persistence** component.
The **Common component** contains utility functionality.

(a) The software architecture documentation



(b) The software architecture model



(c) The abstract representation of the code

Figure 1.2.: Example system with software architecture documentation, software architecture model, and code. For readability reasons, I use an abstract representation of the code instead of the actual source code in this example. Based on [124] and [123].

This example also contains inconsistencies. For example, there is an Unmentioned Model Element (UME) because the *Cache* component in the SAM in Figure 1.2b is not documented in the SAD in Figure 1.2a. Further, the *Common component* from the SAD is missing in the SAM, resulting in a Missing Model Element (MME).

1.6. Contributions

Following my research questions, I make various contributions with this dissertation. In the following, I will present my contributions. I will also evaluate and assess each of these contributions.

First, I look into Architectural Design Decisions (ADDs) in Software Architecture Documentations (SADs) with the following contributions.

C 1.1 *(Taxonomy)*: A fine-grained taxonomy for design decisions in SADs.

C 1.2 *(Automated Classification)*: An approach for automated classification of sentences in SADs based on the taxonomy.

Based on the analysis of SADs for the taxonomy creation, I developed several approaches for TLR in software architecture and contributed the following.

C 2.1 *(Intermediate Representations)*: A model for intermediate representations of the code and model artifacts to allow the approaches to be independent of concrete programming or modeling languages.

C 2.2 *(TLR SAD-SAM)*: An approach for TLR between SADs and SAMs (ArDoCo).

C 2.3 *(TLR SAM-Code)*: An approach for TLR between SAMs and code (ArCoTL).

C 2.4 *(Transitive TLR SAD-Code)*: An approach for transitive TLR between SAD and code that combines the approaches ArDoCo and ArCoTL (TransArC).

C 2.5 *(LLMs for TLR)*: An Large Language Models (LLMs)-based approach to recover trace links to assess the performance and compare it to the other approaches.

Lastly, I will use the trace links for Inconsistency Detection (ID), resulting in the following contributions.

C 3.1 *(ID UME)*: An approach that extends ArDoCo to detect Unmentioned Model Elements (UMEs), i.e., elements in the SAM that are missing in the SAD.

C 3.2 *(ID MME)*: An approach using ArDoCo to detect Missing Model Elements (MMEs), i.e., architectural elements in the SAD that are missing in the SAM.

C 3.3 *(LLMs for ID)*: An LLM-based approach to detect inconsistencies (UMEs and MMEs) to assess the performance and compare it to the other approaches.

1.7. Outline

In the remainder of Part I, I first present the foundations for this thesis in Chapter 2. These include machine learning, information retrieval, natural language processing, knowledge bases, software architecture, traceability link recovery, inconsistency, and evaluation metrics. Then, Chapter 3 discusses related work.

Part II contains the contributions. The part starts with the taxonomy for design decisions in software architecture documentation in Chapter 4. After presenting the taxonomy, I evaluate it in Section 4.3. Furthermore, I present and evaluate an approach to use the taxonomy for an automated approach to classify ADDs in Section 4.4.

I then focus on TLR in Chapter 5. I first present the required intermediate representation of artifacts in Section 5.1. Thereafter, I present the three approaches for TLR, starting with TLR between SADs and SAMs, continuing with TLR between SAMs and code, and finishing the chapter with the transitive approach for TLR between SAD and code. After presenting the approaches, I evaluate them in Section 5.5. In this evaluation, I will also compare the approaches to simple, exploratory ones based on LLMs.

The usage of TLR for detecting inconsistencies is presented in Chapter 6, divided into the detection of unmentioned model elements in Section 6.1 and the detection of missing model elements in Section 6.2. An evaluation follows these contributions in Section 6.4, where I again also introduce and compare to a simple exploratory LLM-based approach.

Lastly, the epilogue in Part III presents the conclusion and future work.

Replication Package

I provide the code, baselines, evaluation data, and results in a replication package to ensure replicability [140]. The replication package can be found online at `https://doi.org/10.5281/zenodo.14216277`.

2. Foundations

This chapter provides the necessary foundations to understand this work.

I introduce software architecture, including models and documentation, in Section 2.1.

Techniques and approaches in the related work make use of Machine Learning (ML) that I briefly introduce in Section 2.2. Section 2.3 then covers the basics of Natural Language Processing (NLP), including basic terms, common processing tasks, word embeddings, word similarity measures, and LLMs. Section 2.4 introduces knowledge bases, ontologies, taxonomies, and knowledge graphs. Various approaches for different tasks often use them to improve understanding and to add context information for their analyses.

Section 2.5 introduces Information Retrieval (IR), a commonly used technique for TLR. After the introduction of all these foundational techniques for TLR, I introduce TLR itself in Section 2.6. As the goal of my thesis is also to use trace links to identify inconsistencies, I define inconsistency in Section 2.7.

In my evaluation, I use various metrics to assess the performance of my approaches. Section 2.8 introduces and explains these and other commonly used evaluation metrics.

2.1. Software Architecture

The ISO/IEC 10746 standard defines (system) **architecture** as “set of rules to define the structure of a system and the interrelationships between its parts”[105, 109]. Similarly, the ISO/IEC/IEEE 42010 standard defines *software architecture* as “fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution”[110, 109]. The architecture names fundamental parts of a system and defines the set of essential properties of a system. Architecture relies on context and cannot be understood in isolation. To determine the essential properties, it is important to know the environment and the *stakeholders*, i.e., the “individual[s], team[s], organization[s], or classes thereof, having an interest in a system”.

The **architectural structure** is the physical or logical layout of the components of a system design and their relationships [109]. *Components* in software architecture are entities “with discrete structure, such as an assembly or software module, within a system considered at a particular level of analysis”[108] and are “one of the parts that make up a system”[109], describing a “functionally or logically distinct part of a system”[109].

Different systems can be defined by shared patterns of structural organization or can be characterized by sharing structural and semantic properties [109]. In this case, these systems use the same **architectural style**. These styles include pipes and filters, layers, rule-based systems, and blackboards.

It is important to note that the architecture is an abstract conception of a system and may exist without ever being written down [110]. This also means that every system has an architecture, but the architecture might not be explicitly documented. The architecture documentation is then the **architecture description** that represents an attempt to express a shared conception of a system [110]. There are various ways to describe architecture, including modeling languages and natural language. Software architecture documentation provides a detailed explanation of the structure, design decisions, and interaction between components within a software system. It serves as a reference for both development teams and stakeholders, ensuring that everyone understands the system’s technical foundation and how its parts fit together. Proper documentation of a system’s architecture is crucial for long-term maintainability, scalability, and collaboration, particularly in large or complex systems. Architecture documentation is crucial in preventing the rapid deterioration of software systems by maintaining clarity and structure throughout the development lifecycle [208, 278]. It enhances the benefits of a well-designed software architecture by preserving knowledge and facilitating future maintenance and scalability. It is particularly important to document architectural design decisions, which are “an outcome of a design process during the initial construction or the evolution of a software system, which is a primary representation of architecture” [4].

Especially for describing software architecture, **views and viewpoints** are essential tools for managing complexity by separating concerns. According to ISO/IEC/IEEE 24765 [109], a view represents a system from a particular perspective, focusing on specific concerns of stakeholders, while a viewpoint is a template or pattern that defines the conventions for constructing a view. Philippe Kruchten's 4+1 architectural view model [145] extends this idea, organizing architecture into five distinct views to address different concerns. The *logical view* captures the system's functionality, the *development view* focuses on its structure in terms of software modules, the *process view* addresses performance and concurrency, and the *physical view* looks at the system's deployment on hardware. The "+1" is the *scenarios view*, which ties together the other views by illustrating how the system handles key use cases. Together, these concepts enable a structured approach to documenting, communicating, and reasoning about complex software architectures.

A common method for documenting architecture involves structural, component-based models, which provide a clear view of system components and their relationships, as noted by Tian et al. [268]. At its core, **architectural modeling** provides a high-level, abstract representation of software systems, describing the interactions between components, subsystems, and layers. This abstraction allows developers and stakeholders to understand system functionality, predict performance, and ensure scalability, reliability, and maintainability before committing to actual development. Various modeling languages and frameworks have been developed to provide standardized approaches for designing and analyzing these architectures. Examples include the *Unified Modeling Language (UML)* [44] and its *Components Diagrams*, and specialized component models like the *Palladio Component Model (PCM)* [231].

UML is one of the most widely used modeling standards in software engineering. Among its many diagrams and constructs, **UML Component Diagrams** are central to architectural design, specifically when representing component-based architectures. UML component models focus on depicting the structural relationships between components, the services they provide, and their dependencies. Components are modular, self-contained units of software functionality, which can be reused or replaced independently. By using interfaces, these components define clear contracts for communication, making the overall architecture flexible and modular. UML component diagrams also support visualizing ports, connectors, and deployment relationships, making them ideal for representing service-oriented or microservices architectures. An example component diagram is shown in Figure 2.1. While these diagrams can depict which components do interact and how they inter-

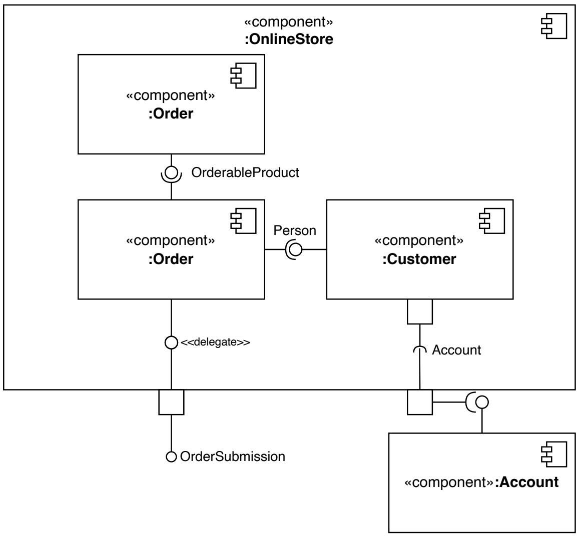


Figure 2.1.: Example UML component diagram

act, they do not directly contain information about non-functional properties such as performance or reliability. For these properties, other models or simulations are required.

One of these models for non-functional properties is the **Palladio Component Model (PCM)** [231]. PCM extends the concept of component-based modeling by focusing not just on the structure of a system but also on its performance, scalability, and reliability characteristics. PCM is designed for predictive, performance-driven software architectures, allowing engineers to simulate and analyze the behavior of a system early in the design phase. Unlike UML, PCM includes annotations for system performance, usage profiles, and resource demands, enabling quantitative analysis of non-functional requirements such as response times, throughput, and resource utilization.

PCM [231] uses a multi-view modeling approach, where the architecture is split into different view types based on the three viewpoints similar to Kruchten's 4+1 architectural view model [145]: *structural*, *behavioral*, and

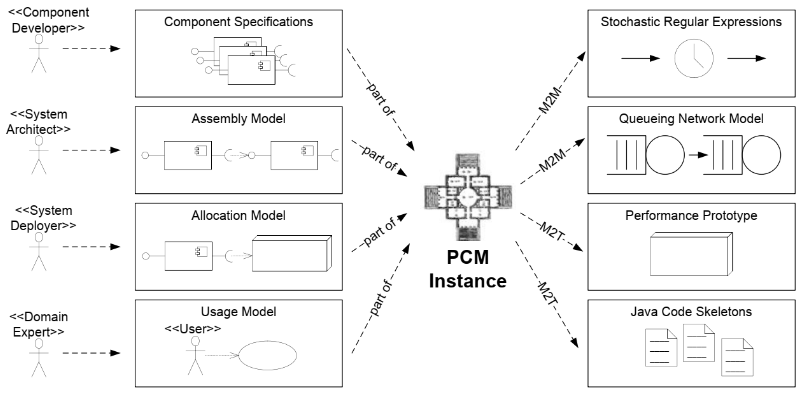


Figure 2.2.: Specification and Non-Functional Property Analysis with the PCM, by Koziolek et al. [141]

deployment. In the structural viewpoint, there is the *repository* view type (defining reusable components) and the *assembly* view type (showing the composition and interaction of components). The behavioral viewpoint contains *sequence diagrams* (for the functional execution semantics), the *service effect specification* view type (containing information about the extra-functional execution semantics), and the *usage model* view type (describing the behavior of the users or other systems interacting with the system). The deployment viewpoint then defines the *resource environment* view type (depicting hardware as resource containers and the links between them) and the *allocation* view type (describing which components instances are allocated on which resource containers). These views are then connected through probabilistic models and performance simulation techniques, allowing for detailed predictions of system behavior under various workloads or infrastructure configurations. Figure 2.2 shows some views and how an instance of a PCM can be used with model-to-model (M2M) transformations as well as model-to-text (M2T) transformations for further usage.

The integration with performance analysis tools makes PCM a powerful framework for architects aiming to ensure that systems meet their performance goals at design time. In contrast to UML, PCM provides more detailed insights into performance characteristics and is tailored for situations where systems need to be optimized for performance before deployment, such as enterprise or distributed systems.

In addition to formal modeling, software architecture documentation often comes in the form of **natural language descriptions**. These informal texts complement diagrams and models by providing thorough insights into the system's design, rationale, and evolution. Textual descriptions are essential for explaining design choices, outlining constraints, and delivering a narrative that conveys the responsibilities and interactions of various components.

For instance, documenting why specific architectural patterns, such as microservices, layered architecture, or event-driven architecture, were chosen helps future developers and stakeholders understand the design rationale. This knowledge can prevent misinterpretation and facilitate informed decisions when the system evolves. One effective method for documenting such decisions is through Architectural Decision Records (ADR)[1], which provide a structured way to capture architectural decisions, trade-offs, and the reasons behind them, ensuring consistency and enabling better future maintenance.

Comprehensive software architecture documentation typically covers several key areas (see [109, 146, 59, 52]). At a high level, the *System Overview* introduces the system's purpose, goals, and core functionalities, alongside the reasoning behind critical design decisions. This section sets the stage for understanding the system's architecture as a whole.

Another critical component is documenting the *Architectural Patterns and Styles* used. Here, the chosen patterns are outlined and justified, explaining how they meet the system's requirements and contribute to its overall qualities, such as maintainability or scalability. Additionally, detailed descriptions of *Components and their Interactions* are provided, illustrating their responsibilities, interdependencies, and, where relevant, the data flow between them. Such information is invaluable for understanding the system's structure and the role of each component.

The documentation often includes the system's *Technology Stack*, specifying the platforms, frameworks, libraries, and tools that the architecture depends on. This section also details the system's external dependencies and their versions, facilitating the management of future upgrades or changes.

[1] More information at `https://adr.github.io/`, last accessed 27.11.2024

Moreover, system qualities, also known as *Non-functional Requirements*, play a pivotal role in architectural design. Attributes such as performance, scalability, security, and availability are critical to the system's success. Documentation should explain how the architecture addresses these qualities through techniques like caching, load balancing, encryption, or other mechanisms.

Deployment-related information is equally important. Descriptions of the system's deployment model, including cloud architecture, server topology, and CI/CD pipelines, clarify how the system is set up in production environments. Such details ensure that deployment is reproducible and adaptable to future changes.

Moreover, it is crucial to maintain a history of architectural decisions, including trade-offs made, risks identified, and reasons behind specific choices.

Good and precise software architecture documentation can prevent software aging and the probability of hidden and lost knowledge, resulting in better software [208, 209]. Software architecture documentation should be clear, concise, and regularly updated to reflect system changes. It is vital to ensure that the documentation is accessible to a wide audience, from developers who need deep technical details to non-technical stakeholders who require a high-level understanding. By documenting the system's static structure and dynamic behavior, teams can foster a shared understanding that supports ongoing development, onboarding, and long-term maintenance.

2.2. Machine Learning

Machine Learning (ML) is a subfield of artificial intelligence (AI) and is defined as a field of study that develops algorithms that can learn from data. Mitchell defines *machine learning* more formally: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E." [189]. This means that ML develops techniques and algorithms to draw conclusions from (training) data to improve a prediction or classification.

There are three main categories of ML algorithms [189]: *Supervised learning*, *unsupervised learning*, and *reinforcement learning*. I will briefly introduce these three categories of ML algorithms in the following.

2.2.1. Supervised Machine Learning

In supervised learning, a model is trained on labeled data, meaning that each input is paired with a corresponding output. The model aims to learn the relationship between inputs and outputs to predict the output for new, unseen data. The training process involves feeding the model training data and continuously adjusting it based on errors between the predicted and actual outcomes. This training data consists of instances of inputs. The input is usually processed, and *features* are extracted, represented in a *feature vector*. For example, an approach can process a word of a sentence and extract the part of speech, position, and surrounding words as features. The input is processed by the algorithm to detect patterns and structures to predict an output value. Depending on the task, the output is either a nominal value, i.e., a class, or a continuous numeric value. Classification tasks like spam detection use nominal output values, and regression tasks, e.g., to predict prices, use numeric values.

There are various categories of classification tasks that are defined by the number of possible output values. *Binary classification* has two classes as outputs that the algorithm needs to sort the input into. For example, an algorithm needs to classify a person into the classes "sick" and "not sick/fit". The task is labeled as *multi-class classification* if there are more than two output classes. Lastly, an output can receive more than one label in so-called *multi-label classification*.

The success of a supervised ML approach is highly reliant on the training data. Important factors are the amount of training data, the distribution of features, and the distribution of output values.

One common problem is *overfitting*. It occurs when a model learns not only the underlying patterns in the training data but also noise and minor fluctuations that do not generalize well to new, unseen data. An overfitted model performs exceptionally well on the training set but poorly on the test or validation set because it essentially memorizes the data rather than learning general patterns. Overfitting is often caused by overly complex models (e.g., too many parameters or layers in Artificial Neural Networks (ANNs)) or insufficient training data. Techniques such as cross-validation, regularization, and pruning can help mitigate overfitting by encouraging the model to generalize better.

Another common problem is the case when one class dominates the other classes in a multi-class classification task. This can cause the model to focus heavily on this one class, influencing the results negatively. *Oversampling* and *undersampling* are techniques used to address such class imbalance in datasets. Oversampling involves increasing the number of examples from the minority class, often by duplicating existing samples or generating new synthetic ones. This helps balance the data distribution, ensuring the model doesn't become biased toward the majority class. In contrast, undersampling involves reducing the number of examples from the majority class to balance the dataset. While both techniques aim to balance class distribution, oversampling can increase the risk of overfitting (due to duplicated data). In contrast, undersampling can lead to the loss of valuable information from the majority class.

There are different suitable supervised ML approaches dependent on the problem at hand, including the available resources like training data and computational power. For example, with their recent advances, Artificial Neural Networks (ANNs) are powerful but require many training samples. For cases with fewer training samples, approaches like the Naïve Bayes (NB) classifier, Support Vector Machine (SVM), or Logistic Regression (LR) can be more useful. In the following, I briefly introduce these approaches.

2.2.1.1. Naïve Bayes (NB) classification

A NB classificator uses a probabilistic model to determine the output class. The classification is based on the simplified (naïve) version of Bayes theorem: The probability of a class c only relies on the independent features of the input. This simplified theorem can be realized by determining, for each feature, the probability that the feature occurs when the output should be the class c. The probabilities for each feature are then combined to determine the overall probability that the input should be labeled as c. The highest probability (argmax) for all classes then determines the output label.

2.2.1.2. Support Vector Machine (SVM)

SVMs try to differentiate the input in two hyperplanes [13]. Accordingly, SVMs use the training data to create a division of the training data such that the distance of the feature vectors of the inputs to the parting plane is maximized. The plane is described using support vectors, giving the approach the name. In their basic form, SVMs can only be applied to linear classification problems and require a binary classification task. With certain adaptations like the kernel-trick, SVMs can also be used for non-linear classification and multi-class classification tasks [13].

2.2.1.3. Logistic Regression (LR)

The LR classification is based on a regression curve. The resulting curve differentiates the input, similar to SVMs. This means that a statistical model is trained to estimate the linear relationship between the input and output by (empirically) determining the parameters of a function. LR can be used for binary and multi-class classification tasks. In contrast to NB, LR does not assume the independence of the features. While this can benefit the accuracy of the classification model, it also requires much more training data.

2.2.1.4. Artificial Neural Networks

Artificial Neural Networks (ANNs) are supervised ML approaches that use interconnected artificial neurons [189, 23]. An **artificial neuron** is a model of a biological neuron in the brain and depicts a mathematical function. Consequently, an artificial neuron receives one or more inputs, weights them, and combines them to produce an output.

An ANN can combine multiple neurons to model any mathematical function. Modern ANNs utilize up to billions of neurons. Typically, neurons are organized in layers. There are typically three types of layers: the input layer, which takes in the raw data, the hidden layers, which perform intermediate computations, and the output layer, which produces the final prediction or result. Each neuron in a layer connects to neurons in the adjacent layer. Hidden layers are where most of the learning and pattern recognition happens. A network with many hidden layers is referred to as a **Deep Neural Network (DNN)** [236], which can capture more complex patterns in data.

An **activation function** introduces non-linearity and complexity into the model. The activation function determines if a neuron should be activated, i.e., output a value, based on the input it receives. Popular activation functions include *ReLU (Rectified Linear Unit)*, which outputs the input directly if positive and zero otherwise, *Sigmoid*, which squashes inputs into a range between 0 and 1, and *Tanh*, which maps inputs to the range of -1 to 1. Activation functions enable the network to solve non-linear problems like image recognition and language processing [189, 23].

Backpropagation is the algorithm used to train ANNs. It works by calculating the gradient of the loss function, i.e., the error, with respect to each weight in the network by using the chain rule, moving backward from the output layer to the input layer. This process allows the network to adjust its weights to minimize the error. Backpropagation aims to optimize the model's performance by gradually reducing the prediction error across many training iterations, typically using gradient descent.

A **feed-forward neural network (FFNN)** is the simplest type of ANN architecture. In this model, information flows only in one direction—from the input layer, through the hidden layers, and finally to the output layer. The network does not have cycles or loops, meaning that once a prediction is made, the process ends. FFNNs are often used for tasks like classification and regression, where the data has no temporal or sequential component.

In contrast, **Recurrent Neural Networks (RNNs)** are designed to handle sequential data by maintaining a hidden state that captures information from previous inputs. Unlike feed-forward networks, ANNs can use their internal memory to process arbitrary sequences of inputs, making them ideal for tasks like time series prediction and natural language processing. However, standard ANNs struggle with long-range dependencies, as they have difficulty retaining information from earlier time steps over long sequences.

A specialized type of ANN is the **Long short-term memory (LSTM)** [101]. The goal of LSTMs is to address the problem of long-range dependencies in sequences. LSTMs include special units called gates that regulate the flow of information, allowing the network to retain or forget information over long time periods. These gates enable LSTMs to perform well on tasks that require learning from long data sequences, such as language translation, speech recognition, and stock price prediction.

The **encoder-decoder** architecture is commonly used in sequence-to-sequence tasks, such as machine translation. In this framework, the encoder processes an input sequence, such as an English sentence, and compresses it into a fixed-length vector called a context or hidden state. The decoder then takes this vector and generates an output sequence, e.g., the translated sentence in another language. This architecture is widely used in natural language processing and has evolved with the introduction of attention mechanisms for better performance. Modern LLMs like BERT or GPT base their architecture on the encoder-decoder architecture (see also Section 2.3.5).

To train ANNs, various **hyperparameters** need to be set. Hyperparameters specify the learning, distinguishing them from the parameters like the weights of neurons that are learned inside the network. Optimizing these hyperparameters is an important part of the training as hyperparameters can significantly influence the network's overall performance [66, 294]. In the following paragraphs, I will present the most common hyperparameters: neurons per layer, epochs, batch size, and learning rate.

The number of *neurons per layer* is a crucial hyperparameter that controls the network's capacity. Too few neurons may result in underfitting, while too many neurons may cause overfitting.

An *epoch* is one complete pass through the entire training dataset. The number of epochs is a hyperparameter specifying how often the learning algorithm will work through the entire dataset. A larger number of epochs allows the network to learn more thoroughly, but too many epochs can lead to overfitting.

The *batch size* refers to the number of training examples processed in one forward or backward pass during training. Smaller batch sizes make the model update weights more frequently, leading to potentially faster learning. However, larger batch sizes tend to provide more stable updates and make better use of hardware like GPUs.

The *learning rate* controls how much to adjust the weights of the network w.r.t. the error during each update. A small learning rate can result in a slower learning process. In contrast, a large learning rate may cause the network to converge quickly but risk overshooting the optimal weights, leading to instability or failure to learn. Properly tuning the learning rate is crucial for efficient training.

2.2.2. Unsupervised Machine Learning

In contrast, unsupervised learning is where the model is trained on data without labeled outcomes, meaning the algorithm is left to discover hidden patterns or structures in the input data on its own [81, 92]. Unlike supervised learning, no predefined target values or categories guide the training process. The model aims to identify relationships or groupings within the data, such as clustering similar data points, e.g., customer segmentation, or reducing data dimensionality, e.g., using principal component analysis. Unsupervised learning is often used for exploratory data analysis when the objective is to uncover insights without explicit guidance. This is crucial in applications like fraud detection in finance, where unusual transactions may signal fraudulent activity, or in cybersecurity, where it can help identify network intrusions. Unsupervised learning offers versatile tools for data exploration, pattern discovery, and anomaly detection in fields ranging from finance to healthcare, without requiring extensive labeled datasets.

One common approach in unsupervised learning is clustering, where the model aims to group similar data points. Popular clustering algorithms include k-means and Hierarchical Clustering. For example, k-Means works by partitioning the data into a pre-defined number (k) of clusters by minimizing the variance within each cluster. Clustering applications span various fields, including information retrieval (see Section 2.5), image analysis, and bioinformatics.

Another major unsupervised learning approach is dimensionality reduction, which seeks to reduce the number of features or variables in a dataset while preserving its essential structure. Techniques like Principal Component Analysis (PCA) [212] and t-distributed Stochastic Neighbor Embedding (t-SNE) [99] are commonly used. PCA transforms the data into a set of orthogonal components, i.e., uncorrelated, to simplify the dataset and highlight its most important features. T-SNE is particularly useful for visualizing high-dimensional data by projecting it into a lower-dimensional space. Dimensionality reduction is especially useful in fields like bioinformatics, where datasets are often extremely high-dimensional. It makes it easier to identify key trends or visualize complex relationships.

2.2.3. Reinforcement Learning

In reinforcement learning, an agent learns to make decisions by interacting with an environment, aiming to maximize cumulative rewards [258]. The agent operates in an environment, receives feedback through rewards or penalties, and adjusts its behavior over time based on this feedback. In the same fashion as unsupervised learning, there are no predefined correct input-output pairs. Instead, the agent learns through trial and error, continuously refining its strategy or policy to optimize its performance. Reinforcement learning is regularly applied in areas such as gameplay, robotics, and autonomous systems, where sequential decision-making is critical.

There are several approaches to reinforcement learning, each tailored to different types of environments and agent behaviors. One common method is value-based learning [258], where the agent estimates the value of different states or state-action pairs, such as in Q-learning [283]. In this approach, the agent learns a value function that predicts the expected cumulative rewards of future actions and selects the action that maximizes this value. Alternatively, policy-based methods like REINFORCE [287] directly learn the optimal policy without explicitly estimating value functions. In this case, the agent focuses on learning a mapping from states to actions, often using gradient-based techniques to improve its policy.

Another advanced approach is actor-critic methods, which combine elements of both value-based and policy-based strategies [139]. The actor updates the policy based on environmental feedback, while the critic evaluates the action taken by estimating its value. This hybrid approach is particularly useful in complex environments with continuous action spaces, where simpler methods may struggle to converge efficiently.

Reinforcement learning has found diverse applications beyond traditional areas like gaming. In robotics, it is used for tasks such as robot navigation, manipulation, and coordination, enabling robots to learn behaviors that are too complex to program manually [138]. In healthcare, RL algorithms help in personalized treatment planning and medical decision-making, optimizing patient care [298].

2.3. Natural Language Processing and Understanding

In our interconnected world with an ever-increasing amount of textual data, there is a demand for systems that can effectively comprehend, use, and interact with human language. To tackle this, NLP and natural language understanding (NLU) focus on developing algorithms and models that enable computers to understand, interpret, and generate human language. The fundamental premise of NLP and NLU lies in bridging the gap between human language and computer systems.

The complexity of human language presents many challenges that require a diverse range of NLP tasks. The following subsections introduce these tasks along with various terms, techniques, and similar concepts to provide a solid foundation for the rest of this work.

First, Section 2.3.1 introduces basic linguistic terms. This is a precursor to Section 2.3.2, which delves into common NLP tasks.

Notable significance has the technique of lexical word representation within a vector space and embedding words. This technique is introduced in Section 2.3.3. These techniques find application in LLMs like BERT or chatGPT, encapsulating the most recent advancements in comprehending and interacting with human language. Section 2.3.5 contains a brief introduction to LLMs.

2.3.1. Basic Linguistic Terms

A **natural language** refers to a language that has evolved naturally among humans for communication, such as English or German. These languages develop over time through social interaction and are shaped by culture, history, and human cognition [277, 117]. Natural languages are spoken or written and typically involve complex rules that regulate how to form meaningful expressions [36, 37]. Unlike natural languages, **artificial languages** are deliberately created for specific purposes, such as programming languages like Java, C, or Python or constructed languages like Esperanto. While natural languages are flexible, often ambiguous and open to interpretation, artificial languages are designed to be precise, unambiguous, and efficient, with clear syntax and semantics tailored to specific tasks.

Syntax is the study of the structure and rules that regulate how words are combined to form sentences in a language [36, 37]. In natural languages, syntax refers to arranging words and phrases to create well-formed, grammatically correct sentences. For example, the typical word order in English is subject-verb-object: “The fox jumped over the dog”. Changing the order can alter the meaning or render the sentence ungrammatical. Syntax also encompasses more complex rules, such as how subordinate clauses or prepositional phrases are integrated into sentences. In artificial languages like programming languages, syntax is equally important, as it dictates the structure of code that ensures programs run correctly. Following the correct syntax is essential for communication or function in both cases.

However, not all syntactically correct sentences contain a proper meaning, differencing it from the **semantics**, the meaning behind words, sentences, and phrases. For example, Chomsky shows with the sentence “Colorless green ideas sleep furiously” that there is a certain independence between syntax and semantics. In natural languages, semantics deals with how linguistic elements like words, phrases, and sentences convey meaning and encompasses the nuances of meaning that can be influenced by context, word choice, and cultural factors. For example, the word *Bank* can have different meanings depending on the context. Among others, a *bank* can be a financial institute, the building where the financial institute resides, or a seating accommodation. In artificial languages, semantics refers to the meaning or behavior that instructions, commands, or functions are designed to express or perform. While syntax ensures that expressions are well-formed, semantics ensures that those expressions make sense and do what is intended.

The underlying purpose in the context of a statement is based on the **pragmatics**. Pragmatics is a fundamental branch of linguistics that focuses on how context influences the interpretation of meaning in communication. Unlike semantics, which deals with the literal meaning of words and sentences, pragmatics examines how speakers use language in real-life situations, considering factors such as speaker intent, social norms, and the relationship between participants. For example, the sentence “You have to do it!” can either be an order by a superior or a friend’s recommendation. Pragmatics involves understanding implicit meanings, assumptions, and how language is shaped by cultural and situational contexts.

Syntax, semantics, and pragmatics are dependent on each other [199]. For example, the syntactic rules allow a semantic interpretation of phrases and sentences. Similarly, pragmatics require an understanding of the semantics.

2.3.2. Natural Language Processing Tasks

There is a variety of different tasks for NLP. These tasks aim to create an understanding of the input natural language text. Several foundational tasks look particularly into grammatical and syntactic structures to provide these to other tasks These other tasks should then be able to (better) interpret the structures to form an understanding of the semantics. As such, some tasks build upon or complement each other.

In the following, I will introduce some of these common tasks in an NLP pipeline. As such, these tasks also play a role in this thesis.

2.3.2.1. Tokenizing

Tokenization is usually the first task in an NLP pipeline [117]. The input natural language text is processed by splitting it into parts, and the resulting tokens can be processed by the following tasks. This commonly means that the text is split into words and punctuation characters. For example, the sentence "The quick brown fox jumps over the lazy dog." is split into the following tokens: [The, quick, brown, fox, jumps, over, the, lazy, dog, .].

Tokenization is often done into sub-word structures for LLMs (cf. Section 2.3.5). Language models like GPT [223] or RoBERTa [161] use byte-pair encoding (cf. [74, 243]) for that. Although the tokens are different, the idea is the same: creating a representation of the input text that is divided into (syntactic) parts to allow subsequent processing.

2.3.2.2. Sentence Splitting

Using the tokenized input, sentence splitting is responsible for dividing the input into structural parts [117]. These structural parts are usually sentences, hence the name. Sometimes, this task also regards paragraphs or similar structures, e.g., in markup languages like HTML. Usually, sentence splitting is done with rule engines that process the token sequences to detect sentence boundaries.

2.3.2.3. POS Tagging

Parts of Speech (POS) are foundational for the syntactical structure of a text. These POS include nouns, verbs, pronouns, prepositions, adverbs, conjunctions, participles, and articles. They provide useful information about

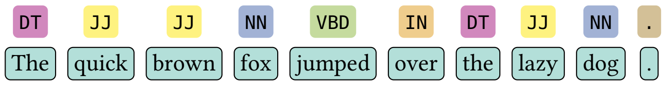


Figure 2.3.: Example sentence annotated with POS tags

Table 2.1.: Selected POS Tags Of The Penn-Treebank Tagset [264].

Abbreviation	Description
CC	Coordinating conjunction
DT	Determiner
IN	Preposition
JJ	Adjective
NN	Noun (singular, mass)
NNS	Noun (plural)
NNP	Proper noun (singular)
RB	Adverb
VB	Verb (base form)
VBD	Verb (past tense)
.	Sentence-final punctuation

sentences and their meaning [117], making it an important cornerstone for NLP.

Assigning labels to each word in the sentence corresponding to their POS is called POS tagging. There are slightly different tagsets for this task. One of the most prominent tagsets is the Penn Treebank [264]. An excerpt of its tagset is displayed in Table 2.1. Figure 2.3 shows an example with a tagged sentence.

Classic algorithms for POS tagging include Hidden Markov Models and Conditional Random Fields [117]. Modern approaches use ANNs (see Section 2.2.1) and transformer-based LLMs (see Section 2.3.5).

2.3.2.4. Lemmatizing and Stemming

Words can appear in different forms to indicate number, case, gender, tense, voice, and others. Declension and conjugation can impede the uniform treatment of a word. There are two different ways to counter such impediments:*lemmatizing* and *stemming*.

Lemmatizing determines the base form of a word, the so-called *lemma* [117]. For example, the verbs "went" or "has" are transformed into the lemmas "go" and "have", respectively. Similarly, the superlative "best" is transformed into "good". Simple algorithms for lemmatizing use a dictionary that contains the mapping between declension or conjugation and base form. However, modern solutions also use machine learning to cover edge cases and resolve potential ambiguities.

Stemming maps words to their stems, usually by cropping the word's suffix [217]. For example, the stem of the word "jumped" is "jump". As such, there are similarities to lemmatization. However, stemming can also reduce different words to the same stem, such as "university" and "universe". Both are reduced to "univers". Using a stemmer requires considering this behavior, as it can either cause issues in subsequent tasks or be used advantageously.

2.3.2.5. Stop Word Removal

Stop words usually contribute little to no (semantic) meaning to the text. Common stop words include articles like "the" or similar determiners, but can Standard lists of stop words often contain articles like "the" or similar determiners and prepositions like "on" or "off" that should only provide structural information to the sentence [117]. Stop word lists can also include auxiliary verbs like "be" or "have".

Stop words can (negatively) affect other NLP tasks or impede processing. Removing stop words is an easy way to eliminate their influence on these tasks and improve the overall results. Filtering stop words is most commonly done with stop word lists that contain all unwanted words. As such, carefully maintaining the used list is important to avoid accidentally filtering relevant words.

2.3.2.6. Parsing

Syntactical parsing analyzes the syntax of a sentence [117]. The goal of parsing is to gain insights into the grammatical structure and relations within the text. These insights can include contained phrases, other constituencies,

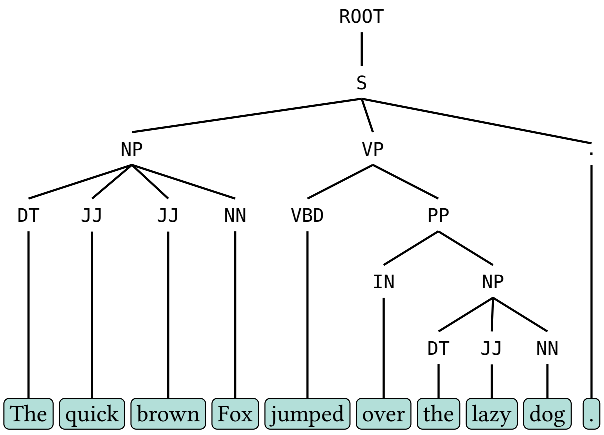


Figure 2.4.: Example parse tree with constituencies

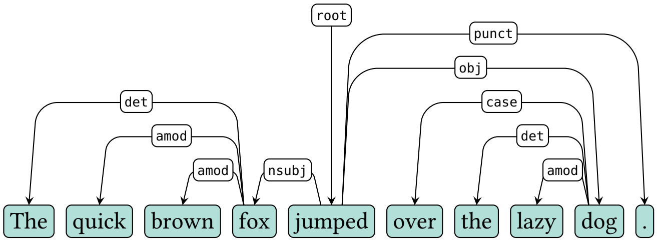


Figure 2.5.: Example sentence annotated with syntactic dependencies

and syntactical dependencies. As such, there are two different types of parsers: constituency parsers and dependency parsers, each with slightly different goals, advantages, and use cases.

Constituency parsers analyze the hierarchical structures of the input sentence to find the constituencies and create parse trees like the example in Figure 2.4. Algorithms to create these parse trees include the CYK algorithm by Cocke, Younger, and Kasami [121, 297] that uses context-free grammars.

Dependency parsers generate graphs that denote the dependencies between the tokens. As depicted in the example in Figure 2.5, dependencies include the subject of a verb or its objects. Moreover, the dependencies can show that a word is modified with, e.g., an adjective like the *amod* dependency between “fox” and “brown”.

Different sets of dependencies are used, like the Universal Dependencies [195] or the Stanford Typed Dependencies [47]. In the following, I introduce some common dependencies and dependencies that are relevant to this thesis, taken from or based on the Stanford Typed Dependencies [47] and the Universal Dependencies [195].

AGENT	The *agent* is the actor in a passive sentence doing the action. Example: “This man has been killed by the police”, killed ⟶ by.
APPOS	The *appositional modifier* of a noun (phrase) defines or modifies the noun (phrase). Example: “Bill, John’s cousin”, Bill ⟶ cousin.
COMPOUND	The *compound* relation denotes compound words that behave as single nouns. Example: “Apple juice”, apple ⟶ juice.
CONJ	A *conjunction* relation connects two words that are separated by coordinating conjunctions such as “and”, “or”. Example: “Bill is big and honest”, big ⟶ honest
IOBJ	The *indirect object* is the noun (phrase) that is the (dative) object of the verb. Example: “She gave me a raise”, gave ⟶ me.
NMOD	*Nominal dependents* of another noun (phrase) and functionally corresponds to an attribute or genitive complement. Example: “Toys for children”, toys ⟶ children.
NSUBJ	A *nominal subject* is a clause’s syntactic subject and agent. Example: “Clinton defeated Dole”, defeated ⟶ Clinton.
NSUBJPASS	Same as NSUBJ, but for passive clauses. Example: “Dole was defeated by Clinton”, defeated ⟶ Dole.
NUM	A *numeric* modifier of a noun and can be any number that modifies the meaning of the noun with a quantity. Example: “Sam ate 3 sheep”, sheep ⟶ 3.
OBJ	The (accusative) *object of the verb.* Example: “She gave me a raise”, gave ⟶ raise.

POBJ	The *object of a preposition* is the noun (phrase) following the preposition. Example: “I sat on the chair”, on → chair.
POSS	A *possessive modifier* preceding its nominal head. Example: “Bill’s clothes”, clothes → Bill.
PREDET	A *predeterminer* is the relation between the head of a noun phrase and a word that precedes and modifies the meaning of the noun phrase determiner. Example: “All the boys are here”, boys → all.
RCMOD	A *relative clause modifier* of a noun (phrase) is a relative clause modifying the noun (phrase). Example: “I saw the man you love”, man → love.

2.3.2.7. Word Sense Disambiguation

Natural language is ambiguous and can cause issues when automatically processing it [117]. These ambiguities play a major role in interpreting the semantics. One kind of ambiguity in natural language is the meaning or sense of a word, also called polysemy. For example, the word “mouse” can mean different things, like an animal or an electronic device. As such, the sentence “I need a new mouse” can have different meanings.

Word Sense Disambiguation (WSD) is difficult and always relies on the context. Firth described this with the statement: “You shall know a word by the company it keeps!” [67]. While the immediate context within the sentence is meaningful, sometimes, like in the given example, it is not enough, and a wider context, e.g., document-wise or domain-specific, needs to be considered.

State-of-the-art approaches for WSD use supervised machine learning like the approaches by Bevilacqua and Navigli [19] or by Barba et al. [14].

2.3.2.8. Coreference Resolution

Coreference resolution is a task in NLP that focuses on determining when different expressions in a text refer to the same entity. This involves identifying and linking words or phrases, such as pronouns, proper nouns, or noun phrases, that all point to the same *thing*. For example, in the text

"Alice went to the store. She bought milk," the pronoun "she" refers to "Alice." Coreference resolution helps improve text understanding by identifying relationships between different parts of a sentence or document, which is crucial for applications like machine translation, question answering, and summarization. The task can be particularly challenging due to linguistic ambiguities, such as when a pronoun might have multiple possible referents or when a noun phrase is vague. It involves both linguistic and computational techniques to interpret context and resolve ambiguities accurately. Modern coreference resolution systems ([116, 292, 133, 53]) often combine machine learning techniques with linguistic rules or use Large Language Models (see Section 2.3.5) to handle these complexities, leveraging context, syntax, and semantics to improve accuracy.

2.3.2.9. Named Entity Recognition

Named Entity Recognition (NER) is an NLP technique that identifies and classifies key information (entities) within a text. These entities can be names of people, organizations, locations, dates, or any other predefined categories such as products or events. For example, in a sentence like "Apple launched the new iPhone 16 Pro in California," an NER system would identify "Apple" as an organization, "iPhone 16 Pro" as a product, and "California" as a location. NER plays a critical role in extracting structured data from unstructured text, helping to organize large amounts of information in documents and other textual sources.

NER is widely used in tasks like information retrieval, text summarization, and sentiment analysis, and forms a foundational step in applications such as chatbots, search engines, and automated customer service. Recognizing entities helps systems understand the context and meaning behind the text, improving functionality in tasks like recommendation engines, machine translation, and sentiment analysis. NER combines techniques from machine learning, rule-based systems, and deep learning models to achieve accurate results, making it crucial for structuring and analyzing large volumes of textual data. Recent approaches like the approach LUKE by Yamada et al. [293] or ACE by Wang et al. [282] leverage transformer-based models such as BERT with document-context integration, achieving high performance on standard datasets like CoNLL-2003 [270] and OntoNotes [220, 219].

2.3.2.10. Further Tasks

Plenty of further NLP tasks cover different aspects of language and can be used for different purposes.

Topic modeling and labeling care about modeling topics such that texts can be analyzed according to the topics they cover. Topic labeling then uses these models to create a concrete, usually human-understandable, label for the topic. Approaches for these can be based on Term Frequency-Inverse Document Frequency (TFIDF) [86] as well as Latent Dirichlet Allocation (LDA) [24] or Latent Semantic Indexing (LSI) [49, 207]. Some approaches also use ontologies (cf. [284]). Recent approaches also use LLMs like BERTopic [86].

Semantic role labeling (SRL) tries to label phrases of the text with the role within their sentence, usually in combination with the sentence's predicate [176]. Example roles include the agent, the verb, and the patient (the target of an act). The general idea is to label "who" did "what" to "whom", "where", "when", and "how.". SRL's biggest benefit over dependency trees, for example, is its independence from syntax, as these roles are only dependent on semantics. This way, the meaning of the sentence can be interpreted more easily.

Other tasks, like sentiment analysis, are not directly relevant to this thesis, so I do not cover them here.

2.3.3. Word Representations and Embeddings

Word representations in the form of vectors or embeddings are critical for transforming textual data into numerical formats that machine learning models can interpret. The underlying idea is to map words or phrases into a vector space, where the relationship between the words can be captured based on their meaning or usage. One of the earliest methods for this is the Vector Space Model (VSM), which represents documents as vectors in a high-dimensional space. In VSM, individual words are encoded as vectors, and the similarity between documents can be calculated by measuring the distance between their corresponding vectors (see also Section 2.3.4).

One simple way to represent words as vectors is through one-hot encoding, where each word in a vocabulary is assigned a unique binary vector with all elements being zero except for one position corresponding to the word's index in the vocabulary. While simple, this approach produces highly sparse vectors and fails to capture any meaningful relationships between words, as all vectors are orthogonal, meaning they share no similarity.

More sophisticated techniques include the bag-of-words (BoW) model [91], which represents a document based on the frequency of words, disregarding the order in which they appear. Variants of BoW such as n-grams or bag-of-n-grams go a step further by capturing short sequences of words (like bigrams or trigrams), allowing some level of context to be preserved. This captures more meaning from phrases like “machine learning,” but still ignores word order or deeper semantic meaning.

The TFIDF weighting scheme is often applied to BoW representations to address these limitations. TFIDF evaluates the significance of words within a document by considering their frequency within that document and their frequency across a larger corpus. Words that occur frequently across many documents receive lower weights, while those that are less common but contextually important are assigned higher importance. By emphasizing contextually relevant words, TFIDF improves the performance of VSM, enhancing both document similarity measures and the quality of word representations for more nuanced textual analysis.

Recent advances in NLP with word embeddings like Word2Vec [184], fastText [183], GloVe [215], ELMo [216] and contextualized embeddings from models like BERT [51] shifted toward even more refined vector representations. These approaches follow the quote from Colin Firth from 1957: “You shall know a word by the company it keeps” [67]. The approaches create representations with small and dense vectors that capture semantic relationships between words by learning from large corpora. This enables models to understand word co-occurrences and more subtle language patterns, such as synonyms, antonyms, and contextual meaning shifts. This progression in word representation techniques has significantly advanced the capabilities of NLP systems, allowing for more sophisticated text understanding and processing.

2.3.4. Word And Document Similarity Metrics

Word and document similarity metrics are essential tools in NLP by enabling the comparison of textual data at various levels of granularity. Classical string similarity metrics like Levenshtein distance [156] and Jaro-Winkler

[289] focus on surface-level differences between two strings. Levenshtein measures the number of insertions, deletions, or substitutions needed to transform one string into another, while Jaro-Winkler gives higher similarity to strings that match earlier, making it useful for short words and names.

More complex approaches involve representing documents or words as vectors and calculating their similarity using metrics like Euclidean distance or cosine distance. Euclidean distance measures the straight-line distance between two vectors, while cosine distance focuses on the angle between vectors. The cosine distance is ideal for comparing document similarity when the magnitude is less relevant.

The Word Mover's Distance (WMD) [150] evaluates semantic similarity by computing the minimal cost to transform one document's word distribution into another's. This means that the WMD calculates the minimal distance of pairing vectors of one document with the vectors of the other document.

The Jaccard coefficient [111] compares sets of vectors, capturing the overlap between them, while Jensen-Shannon divergence (JSD) [73] measures the difference between two probability distributions, often used for comparing topics or word distributions.

LDA [24] is a generative statistical model to discover topics within a collection of documents. It assumes that documents are mixtures of various topics, and each topic is a distribution of words. LDA works by uncovering these hidden (latent) topics based on word patterns, using a probabilistic framework. The Dirichlet distribution governs the distribution of topics across documents and words across topics, ensuring that the generated topics and word allocations are both sparse and diverse. LDA can be used for document similarity by representing documents as vectors in a "topic space" rather than directly comparing their raw words.

Closely related is the LSI [49, 207]. The core of LSI is Singular Value Decomposition (SVD), which reduces the dimensionality of the document space to uncover latent structures in word-document matrices and capture semantic similarity. By projecting documents into this reduced space, the similarity between documents can be computed using cosine similarity measures.

These methods are widely used in tasks such as text classification, clustering, and information retrieval, where understanding the relationship between textual entities is key to improving performance. By combining classical metrics with vector-based approaches, we can capture both lexical and semantic similarities.

2.3.5. Large Language Models

Large Language Models (LLMs) have revolutionized NLP by leveraging the **Transformer architecture** introduced by Vaswani et al. in 2017 [276]. The Transformer is based on a self-attention mechanism that allows models to capture long-range dependencies within text efficiently. This architecture enables parallel token processing, dramatically improving training times compared to previous sequential models like RNNs and LSTMs. Using stacked encoder and decoder layers in the Transformer model laid the foundation for many state-of-the-art LLMs, enabling them to handle large-scale textual data and perform a wide range of language tasks, from translation to text generation.

Several pivotal LLMs have emerged from the Transformer framework, beginning with BERT and GPT. BERT (Bidirectional Encoder Representations from Transformers) introduced by Devlin et al. in 2019 [51], revolutionized the field by employing bidirectional attention to better understand the context of words in a sentence. This made BERT highly effective for various NLP tasks such as question answering and sentiment analysis. On the other hand, Radford et al.'s GPT (Generative Pre-trained Transformer) [222], introduced in 2018, adopted a unidirectional approach, focusing on text generation by predicting the next token in a sequence. While GPT-1 pioneered this autoregressive model, its successors, including GPT-2 [223] and GPT-3 [28], scaled up the number of parameters significantly, resulting in models capable of generating more coherent and contextually relevant text.

Transfer learning has played a critical role in the success of LLMs, allowing pre-trained models to be fine-tuned for specific tasks with much less data. Howard and Ruder's ULMFiT (Universal Language Model Fine-tuning) [102], introduced in 2018, was one of the first major approaches to transfer learning in NLP. It demonstrated that fine-tuning a pre-trained language model could significantly reduce the need for large task-specific datasets. This approach was further enhanced by models like BERT and GPT, which fine-tune on downstream tasks after being pre-trained on massive corpora.

Prompting techniques have since emerged as powerful methods for guiding LLMs toward specific tasks. *Zero-shot prompting* allows models to perform tasks without any task-specific examples, while *few-shot prompting* introduces a small number of examples to improve accuracy. *Chain-of-Thought prompting* further enhances model performance by breaking down complex tasks into smaller, more manageable steps. Additionally, *Retrieval Augmented Generation (RAG)* approaches combine LLMs with external retrieval mechanisms to improve accuracy by incorporating relevant knowledge from external sources during generation [78]. This innovation bridges the gap between the knowledge limitations of LLMs and real-world applications that require dynamic, up-to-date information. At the same time, RAG partly allows to deal with the input limitations of LLMs. Knowledge that is unknown to the LLM needs to be provided with the query. However, without pre-selecting the actual relevant parts, the prompts can become too long for the LLM to handle due to their limited input size requirements. RAG allows selecting relevant and necessary information beforehand to reduce the prompt length.

2.4. Knowledge Bases, Ontologies, Taxonomies, and Knowledge Graphs

In computer science, **knowledge bases** play a crucial role in storing, organizing, and providing access to structured information. They are designed to assist in decision-making, problem-solving, and other computational tasks by encapsulating relevant knowledge in a format that machines can easily interpret and use. Knowledge bases have become foundational in many domains, from artificial intelligence to semantic web technologies, and their influence is growing in the development of intelligent systems that often rely on knowledge bases to process, analyze, and reason with data more efficiently.

A popular example of a widely used knowledge base is WordNet, a lexical database for the English language [185]. WordNet organizes words into sets of synonyms (synsets) and defines their relationships, such as hypernyms (general terms) and hyponyms (specific instances), providing rich semantic information. WordNet is particularly influential in NLP, helping software systems understand the meaning of words and phrases in different contexts. It enables various applications, including machine translation, text analysis, and sentiment analysis, by enhancing the system's ability to understand the nuances of human language.

Closely related are **taxonomies**. They provide a hierarchical classification system that organizes information into categories and subcategories based on shared characteristics [263]. In knowledge bases, taxonomies help structure the relationships between different concepts or entities, making it easier for both humans and machines to navigate and understand the underlying data. By defining clear parent-child relationships (e.g., general to specific), taxonomies enable more efficient querying, retrieval, and reasoning over the stored knowledge. This hierarchical approach is foundational for various applications, including search engines, recommendation systems, and semantic web technologies, where the organization of knowledge is key to improve accuracy and relevance in data analysis and information retrieval.

Ontologies are a central concept in knowledge bases, especially when building systems that need to represent complex domains. An ontology defines a formal and explicit specification of concepts, entities, and their relationships within a specific domain of knowledge [88]. Gruber [87] defines ontologies as an explicit specification of a shared conceptualization. Ontologies not only define the terms used to describe a domain but also the rules and logic that describe how these terms relate to each other. In this way, they are similar to taxonomies but can use more complex relationships. This structured framework enables both humans and machines to understand, reason about, and manipulate complex knowledge in a meaningful way. Ontologies are often represented in formal languages, such as OWL (Web Ontology Language), which are designed to be machine-readable and used for tasks like data integration, semantic search, and automated reasoning. An ontology typically consists of several components. *Classes* represent the entities or objects within a domain like `Disease` or `Person`. *Instances* are the specific objects or entities that belong to a class like `COVID-19`. *Properties* define the characteristics or features of a class. For example, a `Disease` might have the property `Symptoms`. *Relationships* then describe how classes and instances are related. For example, the relation `affects` can show that a `Disease` affects a `Person`.

Ontologies also support logical reasoning through the use of axioms and rules, which define constraints and relationships in the domain. This allows intelligent systems to infer new knowledge by applying these rules. For example, if an ontology specifies that all female mammals produce milk and that a dog is a mammal, a reasoning system can infer that female dogs can produce milk, even if this specific fact was not explicitly stated. This means ontologies follow the *open-world principle*: just because something is not explicitly stated in the ontology, it does not imply it is false—there may be missing facts that are simply unknown. This makes them differ from closed-world systems and allows reasoning with incomplete data, making it ideal for dynamic and evolving domains like the semantic web.

Wikipedia and DBpedia are significant knowledge bases in the software engineering and AI communities. The open online encyclopedia Wikipedia is a crowd work where users maintain and extend the content for free. Wikipedia is the biggest encyclopedia with roughly 6.9 million content pages[2] in the English variant, making it a useful data source and knowledge base. By using ontologies, DBpedia creates a semantic representation of the Wikipedia content, linking entities and concepts together [11]. This structured information is particularly valuable for semantic web applications, enabling better data interoperability, querying, and integration across different platforms and datasets.

Knowledge graphs are also closely related to knowledge bases and ontologies, providing a practical and visual representation of structured knowledge. In essence, a knowledge graph is a type of knowledge base that organizes information into nodes representing entities, concepts, or objects, and edges representing relationships or connections between those entities. This graph-based structure makes it easy to visualize complex relationships and associations between data points, facilitating tasks like information retrieval, data integration, and reasoning.

Knowledge graphs build upon the principles of ontologies by using them as the underlying structure that defines the types of entities and the relationships between them [210, 57]. Ontologies provide formal semantics, defining entities and how they can be related. The knowledge graph uses this ontology to instantiate specific instances and their real-world relationships. Traditional knowledge bases focus on storing structured information, often in tables or hierarchical formats, while knowledge graphs emphasize the interconnectivity between entities, creating a network of knowledge. This makes knowledge graphs useful for NLP, recommendation systems, and the semantic web.

[2] According to `https://en.wikipedia.org/wiki/Special:Statistics?action=raw` on 27.11.2024

2.5. Information Retrieval

Information Retrieval (IR), as defined by Manning et al. [174], is the process of identifying and retrieving relevant materials, typically documents, from large, unstructured datasets. The main goal of IR systems is to fulfill an information need by locating content that best matches a user's query. These collections often consist of text-based information sources, such as web pages, books, or digital archives, and are often stored across large, computer-based systems. Unlike structured databases that organize information in a predefined manner, IR systems need to deal with more flexible, unstructured data that lacks a predefined schema, making retrieval both complex and important for applications such as search engines and digital libraries.

A key challenge in IR is balancing the retrieval of relevant documents with the retrieval of all relevant documents (i.e., precision versus recall, see also Section 2.8.1), as these systems must analyze vast amounts of data efficiently. The analyses are often based on similarity measurements that return the relevance of a given document to the query based on syntactic or semantic conformity. To improve accuracy, techniques such as term weighting, relevance feedback, and ranking algorithms are employed to determine which documents are most likely to meet a user's query. Recent advancements in IR have incorporated NLP (see Section 2.3) and ML techniques (see Section 2.2), enabling these systems to comprehend and rank documents based on their semantic content rather than merely matching keywords.

2.6. Traceability Link Recovery

In software engineering, traceability is the possibility of establishing traces between artifacts to relate information and establish relationships in these artifacts [85, 109]. The related information can then be used in various situations and for different tasks. This makes Traceability Link Recovery (TLR) an important task for software engineering. TLR contains specific vocabulary and terms that this section introduces.

According to Gotel et al., the term trace artifact corresponds to a "traceable unit of data (e.g., a single requirement, a cluster of requirements, a UML class, a UML class operation, a Java class, or even a person). A trace artifact is one of the trace elements and is qualified as either a source artifact or as a target artifact when it participates in a trace. The size of the traceable unit of data defines the granularity of the related trace." [85]. Similarly, they define that source artifacts are the artifacts from which traces originate, and that target artifacts are the artifacts at the destination.

A trace artifact also has a type, which, according to Gotel et al., is a label that characterizes trace artifacts that have the same or similar structure (syntax) and/or purpose (semantics) [85]. The trace granularity is the level of detail at which a trace is recorded and performed. This granularity is defined by the granularity of the source artifact and target artifact [85]. For example, trace links can be defined between a use case and a code class or more fine-grained between parts of a use case and on the method level in the code.

Following these definitions, a trace link is a "specified association between a pair of artifacts, one comprising the source artifact and one comprising the target artifact"[85]. Moreover, Gotel et al. state that trace links are effectively bidirectional. To show the pairs of associated trace artifacts in a system, trace links are often displayed in traceability matrices [85, 109]. In these matrices, the cells record all traced relations.

According to Gotel et al., traces can be atomic when they each comprise a single source artifact, a single target artifact, and a single trace link [85]. Chained traces, also known as transitive traces, are traces that combine multiple atomic traces in sequence, such that a target artifact for one atomic trace becomes the source artifact for the next atomic trace.

The activity to establish traces is called tracing and can either be done manually or automated [85]. Tracing is called horizontal when tracing artifacts are at the same level of abstraction, e.g., traces between all performance requirements. When tracing artifacts are at differing levels of abstraction, tracing is called vertical.

There are different types of trace links based on the structure, i.e., the syntax, and/or purpose, i.e., the semantics [85]. Example types include *implements*, *tests*, *refines*, and *replaces*, *derives*, *verifies*, *satisfies*, *allocates*. The most general types that are commonly used are *corresponds to* and *is (closely) related to*. In this work, if not stated otherwise, trace links usually correspond to the latter two types.

Automated software and system traceability covers different domains, use cases, and techniques [248, 12]. Common use cases include TLR between requirements and other requirements or code, between test cases and code, and between issues. Common techniques are IR-based techniques (see Section 2.5), ML-based techniques (see Section 2.2), and in general various techniques that make use of NLP (see Section 2.3).

2.7. Inconsistency

In this work, I look into the inconsistency between SADs and SAMs. While there is some intuitive understanding of inconsistency, there are slightly different understandings. As such, I define inconsistency for this work in the following.

ISO/IEC/IEEE 24765 [109] defines consistency as "degree of uniformity, standardization, and freedom from contradiction among the documents or parts of a system or component". A system is consistent if it is "without internal conflicts".

According to Kühn et al., [148], inconsistency arises when there is a lack of coherence among the various elements within a model or between multiple models. This means that certain statements, constraints, or assumptions within the model contradict each other or fail to align with established rules and relationships, leading to an overall conflict in the model's structure and behavior.

Inconsistency can be classified according to its impact on the model's functionality and validity. Functional inconsistency occurs when the model's operations or processes do not work as intended due to contradictory elements. Validity inconsistency, on the other hand, refers to scenarios where the model fails to meet the criteria or constraints that are necessary for its correctness or accuracy, often leading to incorrect conclusions or unreliable outputs.

There is also the concept of inconsistency types based on the origin or source of the discrepancies. For instance, syntactic inconsistencies arise from errors in the model's formal language or structure, such as violations of syntax rules. Semantic inconsistencies occur when the meaning or interpretation of elements within the model leads to conflicts, despite correct syntax. This distinction helps pinpoint the exact nature of the inconsistency and apply the appropriate resolution techniques.

The classification schema of Kühn et al. uses seven applicable properties of consistency relations: Abstraction, Metalevel, Position, Observed Property, Development Phases, Quantification, and Gradual Consistency. Derived from that, in this work, inconsistency means that entities and descriptions within the SAD contradict the corresponding parts in the model and vice versa. This also includes cases where one artifact states the presence of an entity, while the other does not contain such a statement. For example, the SAD mentions a certain component that is missing in the SAM, or the SAM contains one or more components that are never mentioned in the documentation. Under the open-world assumption, the absence of information about something does not imply its non-existence; it merely indicates that it has not been modeled. Therefore, these examples do not represent inconsistencies within this framework. In contrast, under the closed-world assumption commonly used in (formal) architecture modeling, anything not explicitly modeled is assumed not to exist. From this perspective, these examples would indeed be considered inconsistencies. These inconsistencies can be problematic if developers are unaware, but after being notified, they can also decide they are not problematic or postpone fixing the inconsistency. Consequently, *Inconsistency Detection* is important to realize that the system, including all its artifacts, has internal conflicts and, thus, is inconsistent.

To address inconsistencies between models, automated techniques for *consistency preservation*, also referred to as *model repair*, have been developed [136]. These methods commonly use *bidirectional transformations* [252], which maintain consistency by establishing relationships between models through predefined consistency relations. When an inconsistency is detected, these transformations automatically update the models to restore consistency. However, in this work, I focus on informal documentation rather than the consistency management of formal models. As a result, I do not address the issue of consistency preservation in this context.

2.8. Evaluation Metrics

Specific metrics are needed to evaluate the performance of a specific approach and compare the performances of different approaches. There are many metrics in general, so I will focus in this section on metrics commonly used for various tasks, including TLR and classification.

Table 2.2.: Confusion Matrix

		Gold Standard	
		Positive	Negative
Prediction	Positive	TP	FP
	Negative	FN	TN

The following sections cover metrics for nominal data in Section 2.8.1, statistical hypothesis tests to evaluate hypotheses and to check the significance of results in Section 2.8.2, and methods to examine inter-rater reliability in Section 2.8.3.

2.8.1. Metrics for Nominal Data

Many tasks allow us to define an expected outcome. There can be numerical outcomes, such as the expected response time of a system. For numerical values, an experimenter can calculate how much a single response deviates from the expected value. In cases like TLR, the expected outcome is nominal. For nominal data, a researcher can classify a result as either correct or incorrect. This also means that we can treat such a problem as a binary classification task. For example, for TLR, the task can be seen as a classification in which a decision is made for each possible trace link whether it is actually a trace link.

When treating a task as a binary classification task, one can use a confusion matrix like the one in Table 2.2 to sort the results into four categories: True Positives (TP), False Positives (FP), False Negative (FN), True Negative (TN). Each result can either be predicted as positive (e.g., is a trace link) or negative (e.g., is no trace link). Similarly, the gold standard defines the set of positive and negative results. True Positives, on the one hand, are correctly predicted positive results, as they are also contained in the set of positive results in the gold standard. False Positives, on the other hand, are results that are incorrectly predicted to be positive according to the gold standard. False Negatives are negatively predicted results that should have been predicted positively, while True Negatives are correctly predicted negatively.

Classification tasks can also be multi-class classification tasks. The difference to binary classification is that the confusion matrix has a bigger dimension

Table 2.3.: Confusion Matrix Of A Multi-class Classification Task

		Gold Standard		
		Class A	Class B	Class C
Prediction	Class A	AA	AB	AC
	Class B	BA	BB	BC
	Class C	CA	CB	CC

than 2x2. Table 2.3 shows an example for three classes. Each possible class has one row for the predicted class and one column for the correct class according to the gold standard. The cells show how each prediction relates to the gold standard. For example, a correct prediction of class A is sorted into cell AA, while an incorrect prediction is then either in cell AB or AC. While a multi-class classification is different from a binary classification, for the evaluation, the multi-class classification can be boiled down into binary classification for each class. This means that when looking at a certain class, like class A in the example, the confusion matrix can effectively be transformed into a 2x2 confusion matrix. For example, for class A, the cells AB and AC are combined as false positives, cells BA and CA are false negatives, and the remaining cells (BB, BC, CB, CC) are true negatives. This way, we can calculate metrics in the same way as in the binary classification. The difference is that these calculations must be done for many metrics for each class, as they do not summarize the overall results. To have summarized results, we need to aggregate these results, e.g., by averaging with the arithmetic mean.

With a confusion matrix, we can calculate various metrics. These metrics include Precision, Recall, F_β, Accuracy, Specificity, Φ, and the Mean Average Precision that I want to introduce in the following.

Precision can be calculated with the formula shown in Equation 1. It is the portion of correct predictions to the made predictions and, thus, shows the fraction of correct predictions. As such, precision ranges from 0 to 1.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{1}$$

Recall, as shown in Equation 2, is the proportion of correct predictions from all expected positive instances in the dataset. Thus, recall depicts the share of the actual predictions to the expected predictions, and recall values range from 0 to 1.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2}$$

In isolation, the significance of precision and recall is limited. Precision and recall are intertwined metrics. Using precision alone can result in an overly cautious approach, where only the most confident predictions are accepted, potentially missing important instances. Relying solely on recall might lead to a strategy that captures many relevant instances but at the expense of introducing more false positives.

The synergy between precision and recall lies in their ability to provide a comprehensive assessment of an approach's performance. Together, both metrics offer a balanced evaluation that takes into account both accurate predictions and comprehensive retrieval. Therefore, analyzing precision and recall in combination is essential to thoroughly understand an approach's effectiveness and efficiency, especially in IR and classification approaches.

A common way to combine precision and recall into a single score is the **$\mathbf{F_1}$-score** in Equation 3. The F_1-score is the harmonic mean between precision and recall, balancing the influence of both metrics. As a harmonic mean, the score ranges between 0 and 1 and tends more towards the lowest value the farther apart both values are. This reduces the influence of extreme values compared to, e.g., the arithmetic mean. Some example results are depicted in Table 2.4. In scenarios one and two, the F_1-score is 0.0 as one of the values is 0, which shows the difference to the arithmetic mean.

$$F_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3}$$

$$F_\beta = (1 + \beta^2) \times \frac{\text{Precision} \times \text{Recall}}{(\beta^2 \times \text{Precision}) + \text{Recall}} \tag{4}$$

While the F_1-score balances precision and recall, the **$\mathbf{F_\beta}$-score** in Equation 4 allows to emphasize either precision or recall. The F_β score generalizes the concept of the F_1-score by introducing the parameter β, a positive real number that controls the relative importance of precision and recall. As β increases to values above 1.0, recall is emphasized more, making the metric more sensitive

Table 2.4.: Example Results For F_1-score In Different Scenarios

Scenario	Precision	Recall	F1
Scenario 1	0.00	1.00	0.00
Scenario 2	1.00	0.00	0.00
Scenario 3	0.20	0.90	0.33
Scenario 4	0.40	0.60	0.48
Scenario 5	0.60	0.60	0.60

to false negatives. Conversely, as β decreases to values below 1.0, more emphasis is placed on precision, making the metric more sensitive to false positives. For example, a β of $\sqrt{2}$ weights recall twice as much as precision, while a β of $\sqrt{0.5}$ doubles the weight of precision.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{5}$$

Another commonly used metric is **accuracy**. As shown in Equation 5, accuracy takes all values in the confusion matrix into account. Accuracy sets the correct predictions, i.e., true positives and true negatives, in relation to all possible results. In contrast to precision and recall, accuracy also considers the true negatives. Accuracy provides a general view of the correctness of predictions across all classes. It's a simple and easy-to-understand metric, but it may not be sufficient in cases where class distribution is imbalanced or when the costs of different types of errors (false positives and false negatives) are different. A limitation associated with accuracy is its sensitivity to imbalanced datasets. In cases where one class significantly outnumbers the others, accuracy tends to be biased toward the dominant class's performance, often leading to misleadingly high scores. This occurs because the model can achieve seemingly high accuracy by simply predicting the majority class for most instances, even if it performs poorly on the minority class. For example, if 90% of cases need to be classified positively, a simple approach can classify every case as positive and achieve a high accuracy of 0.9 while misclassifying every case of the minority class.

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}} \tag{6}$$

Specificity is a further commonly used metric. Specificity measures the ability of an approach to avoid false positives w.r.t. the true negatives. In Equation 6, the similarity to precision is visible as the sole distinction being that specificity focuses on true negatives instead of true positives. It is particularly useful when the cost of false positives is high, and correctly identifying the true negatives is crucial. For example, in medical diagnosis, correctly identifying healthy patients (true negatives) can be as important as identifying those with a disease (true positives).

$$\Phi = \frac{\text{TP} \times \text{TN} - \text{FP} \times \text{FN}}{\sqrt{(\text{TP} + \text{FP})(\text{TP} + \text{FN})(\text{TN} + \text{FN})(\text{TN} + \text{FP})}} \tag{7}$$

The **Phi correlation coefficient** (Φ), is a statistical measure used to assess the association between two categorical variables in a 2x2 (confusion) matrix [45]. The Φ coefficient is also known as Matthew's correlation coefficient (MCC) [177] and is often used in the machine learning domain. The Φ coefficient quantifies the degree of dependence or relationship between the variables. The values range between -1 and 1. 1 indicates a perfect positive correlation, -1 is a perfect negative correlation, and 0 indicates no correlation, i.e., random predictions. Values closer to -1 or 1 indicate a stronger association, and values closer to 0 indicate a weaker association or independence. This correlation coefficient returns the same results as the Pearson correlation coefficient for two binary variables.

As shown in Equation 7, Φ is calculated from all four values in the 2x2 confusion matrix (cf. Table 2.2). Φ is a concise way to measure the relationship between two categorical variables. In our cases, the relationship is between the expectation of the gold standard and the prediction. Consequently, the Φ coefficient is suitable to capture the strength and direction of the association between prediction and expectation. The F_1-score, in comparison, has a slightly different meaning by assessing the prediction's accuracy in identifying positive cases. Both metrics provide valuable insights into different aspects of an approach's performance.

Table 2.5.: Example Results For F_1-score And Φ In Different Scenarios

Scenario	TPs	FPs	FNs	TNs	F_1-score	Φ
Sc. 1	78	10	10	2	0.88	0.05
Sc. 2	78	10	10	22	0.88	0.57
Sc. 3	50	50	50	50	0.50	0.00
Sc. 4	90	6	14	950	0.90	0.89

One benefit of the Φ correlation coefficient is its capacity to provide a balanced measure, which is especially useful when dealing with imbalanced datasets. As such, the metric is especially valuable when other metrics, such as accuracy or F_1-score alone, are less informative. For example, when there are many more true positives than true negatives, the F_1-score is high while the Φ coefficient is low: Scenario one in Table 2.5 shows 78 true positives, ten false positives, ten false negatives, and two true negatives. The resulting F_1-score in scenario two is 0.88, indicating good results. However, the Φ coefficient of 0.05 shows that there is a low correlation. This observation stems from the imbalanced dataset and the comparably low number of true negatives, especially with the number of false negatives. The results signify that while the classification may yield a good F_1-score, there is a limited association between prediction and expectation. The Φ coefficient is sensitive to variations in both positive and negative class associations and, as such, offers insight into the extent of dependence between both. The other examples in Table 2.5 show scenarios with different distributions and how they affect both metrics.

In conclusion, leveraging both metrics, F_1-score and the Φ coefficient, together facilitate a more nuanced evaluation of classification efficacy, accounting for different aspects of the classification process and yielding a more comprehensive characterization of the underlying relationships. Together, these metrics contribute to informed decision-making for practical applications.

The **Mean Average Precision (MAP)** is an evaluation metric widely used in the context of IR and TLR. The MAP quantifies the accuracy of retrieval systems, making it a crucial tool for assessing the performance of algorithms for recommendation systems and similar. As Equation 8 shows, the MAP is the mean of the Average Precision (AP) of every query (q). For example, in the context of TLR, a query can be a source artifact that needs to be traced

Table 2.6.: Example Average Precision Calculation

Rank (k)	rel(k)	TPs	$P(k)$	AP Contribution
1	1	1	1/1	1/1
2	0	1	1/2	0
3	1	2	2/3	2/3
4	0	2	2/4	0
5	1	3	3/5	3/5
6	1	4	4/6	4/6
$\text{AP}_q = (1 + 0 + \frac{2}{3} + 0 + \frac{3}{5} + \frac{4}{6})/4 = \frac{11}{15} \approx 0.74$				

to target artifacts. The AP calculates the precision of each item in a ranked list, i.e., the ranked prediction outputs of an algorithm or approach. For TLR, the ranked list can be the list of target artifacts for the given source artifact, ordered by confidence.

$$\text{MAP} = \frac{\sum_{q=1}^{Q} \text{AP}_q}{Q} \tag{8}$$

$$\text{AP}_q = \frac{\sum_{k=1}^{n} P(k) \times \text{rel}(k)}{\text{total number of relevant documents}} \tag{9}$$

Equation 9 shows the formula to calculate the AP. In this context, $P(k)$ denotes the precision at rank k. This means $P(k)$ is the ratio of true positives in the list up to the k-th entry divided by k. rel(k) yields a value of 1 if the item at rank k is relevant (i.e., a true positive); otherwise, it returns 0. When combined, these two elements constitute the numerator, effectively representing the sum of precisions at all ranks with true positives. This sum is subsequently normalized with the number of expected true positives. Table 2.6 shows an example for calculating AP.

2.8.2. Statistical Hypothesis Tests

Statistical hypothesis tests, also called significance tests, check the validity and significance of hypotheses. Because you cannot statistically prove the validity of a hypothesis directly, statistical hypothesis tests work with null and alternative hypotheses. For this, researchers need to formulate a null hypothesis that is checked with statistical tests. Additionally, researchers formulate an alternative hypothesis that is accepted if the null hypothesis is rejected. Usually, the original hypothesis is used as the alternative hypothesis, and the opposite hypothesis of the original one is used as the null hypothesis. This way, researchers can statistically prove the validity of their hypothesis.

Researchers must define a significance level (α) to accept or reject a hypothesis. The significance level is the predetermined threshold that researchers set to determine how much evidence they require to reject the null hypothesis. The significance level represents the probability of making a Type I error, which is the error of rejecting the null hypothesis when the hypothesis is true. It is important to select an appropriate significance level based on the study's context and the potential consequences of making Type I errors. Smaller significance levels reduce the chance of making a Type I error. Still, they may increase the likelihood of making a Type II error, which is the error of failing to reject the null hypothesis when it's false. In software engineering, the significance level is usually set at 0.05. This means researchers are willing to accept a 5% chance of incorrectly rejecting the null hypothesis even when it's true.

The p-value is calculated to assess if the null hypothesis is correct according to the significance level. The p-value quantifies the likelihood of getting the observed data if there is no real effect or difference, i.e., assuming the null hypothesis is true. Given the null hypothesis, low values suggest that the observed data is unlikely. If the p-value is below the defined significance level, researchers can reject the null hypothesis.

Hypothesis tests can take the form of either one-sided or two-sided tests to calculate the significance. Researchers employ a one-sided test when examining a specific direction, e.g., a positive or negative trend. If researchers do not focus on a particular direction, they must utilize a two-sided hypothesis test.

There are different statistical hypothesis tests, and the selection of a specific test depends on the assumptions that a researcher needs to make. One important assumption is about the data and how it is distributed. For example, Pearson's chi-squared test is applied to categorical data such as confusion matrices [214]. Pearson's chi-squared test is a generalization of the Φ metric (cf. Section 2.8.1). For continuous data that follows a normal distribution, researchers can apply one of the various variants of the t-test, like Student's t-test [255] that compares the averages of two groups.

A non-parameterized statistical hypothesis test for paired samples is the **Wilcoxon signed-rank test** [286]. Wilcoxon signed-rank test can be used to evaluate the significance of the results of an approach compared to a baseline approach or similar. The test is independent of the data distribution and does not need to assume, e.g., normal distribution.

To calculate the Wilcoxon signed-rank test, the difference for each pair is determined. All pairs are ranked according to the difference, not considering the sign. Then, the ranks are combined by adding positive ranks and subtracting negative ones. Equal pairs, i.e., with a difference of 0, are ignored. The resulting value is compared to its distribution under the null hypothesis. There are tables for the minimum resulting value needed to reject the null hypothesis for small numbers of test pairs (excluding equal pairs).

2.8.3. Inter-Rater Reliability

Inter-rater reliability, also known as inter-annotator agreement, is a research methodology concept that expresses the consistency and agreement between two or more independent raters when evaluating or coding a set of data or observations. The set of data or observations can be, for example, manual classifications according to a classification scheme. As such, creating gold standards for the metrics of nominal data in Section 2.8.1 also fall in this category.

In essence, inter-rater reliability assesses the degree to which different raters come to the same or similar conclusions. This measure is essential because it quantifies rating variability and how it can be attributed to the data's inherent nature rather than to inconsistencies among raters. A high inter-rater reliability score indicates that raters interpret the data consistently, minimizing the potential for subjective bias to affect the results. As such, a high inter-rater reliability score contributes to the overall robustness of a study.

There are various statistical approaches for calculating inter-rater reliability. Examples for these approaches include Cohen's kappa[43], Fleiss' kappa[68], Pearson's product-moment correlation r[213], or Spearman's rank correlation coefficient ρ[249]. Selecting a suitable approach depends on the kind of data, preconditions, and requirements.

A versatile measure of agreement is **Krippendorff's** α (**K**α) [143, 144]. The versatility of Kα stems from various properties: Kα can handle different kinds of data, including nominal, ordinal, interval, and ratio levels of measurement. Additionally, Kα can deal with missing data and can operate on small sample sizes. Lastly, Kα is not constrained by the number of observers or coders.

$$\alpha = 1 - \frac{D_o}{D_e} \tag{1}$$

As displayed in Equation 1, the calculation of Kα can be broken down into a calculation of the ratio of observed disagreement D_o to the expected disagreement D_e. A coincidence matrix is used to calculate the disagreement values. The matrix shows all values that can be paired from the canonical form of the reliability data in a symmetrical $v \times v$ square matrix, where v is the number of possible values of a variable. In the coincidence matrix, the diagonal represents the number of matches for any two coders for the given variable. As such, the disagreement is calculated from the values aside from the diagonal.

A Kα value of 1 indicates perfect reliability, and a value of 0 shows that there is no reliability at all. Values below 0 suggest systematic disagreements that exceed chance. There is no universally accepted minimum value for Kα that indicates statistical significance, as each case has different requirements. When mistakes are expensive, researchers must choose a high minimum Kα. A commonly used lower bound is 0.66, and a commonly accepted threshold is 0.8 [144]. According to the taxonomy by Landis and Koch [153], values below 0 can be seen as no agreement, 0.01 to 0.20 as none to slight agreement, 0.21 to 0.40 as fair agreement, 0.41 to 0.60 as moderate agreement, 0.61 to 0.80 as substantial agreement, and 0.81 to 1.00 as almost perfect agreement.

Table 2.7.: Abstract values in a matrix that shows the prevalence of the raters. c_{ab} are the numbers of answers where A answered a and B answered b. π_A and π_B are the sums of the c_{ab} in the row or column, normalized by the total number of answers.

		B		
		0	1	
A	0	c_{00}	c_{01}	$1 - \pi_A$
	1	c_{10}	c_{11}	π_A
		$1 - \pi_B$	π_B	

Several known statistical problems with metrics like Cohen's kappa and Kα [180] exist. For example, if the ratings are imbalanced, the resulting values are comparably low even though the raters have a high agreement. This is known as the Kappa paradox [63]. In such cases, the validity and significance of the metric can be limited. Gwet's AC1 [90] is a measure that is resistant to such a paradox that can be applied when there are two raters.

Gwet compares the possibility of raters performing a random rating (coin flip) to both raters agreeing on decisions. Equation 2 shows the formula of calculating Gwet's AC1.

$$g = \frac{a - \gamma}{1 - \gamma} \tag{2}$$

$$\gamma = 2 \times \pi(1 - \pi) \tag{3}$$

$$\pi = 0.5 \times (\pi_A + \pi_B) \tag{4}$$

α is the proportion of ratings where both raters agreed. γ (see Equation 3) is then the chance that one or both raters performed a random rating. The value for γ is approximated by the ratio of the variance of the prevalence to the maximum possible variance. This is done using π in Equation 4. As depicted in Table 2.7, π_A and π_B are the observed values for two raters, i.e., the portion of each rater answering positively. These values depict proportions of ratings, meaning they range from 0 to 1, normalized by the total number of ratings.

The resulting value for Gwet's AC1 can be interpreted in the same way as Kα with the taxonomy by Landis and Koch [153], ranging from no agreement for values below 0 up to an almost perfect agreement for values above 0.81.

3. Related Work

In this section, I look into work that is related to my contributions in this dissertation. Consequently, I look into work about design decisions in Section 3.1, about traceability link recovery in Section 3.2, and about inconsistency detection in Section 3.3.

3.1. Design Decisions

Design decisions are crucial to developing complex systems, influencing architecture, functionality, and long-term maintenance. Properly capturing, documenting, and analyzing these decisions can improve the clarity of the design process, reduce the risk of costly errors, and support future modifications or refactoring [208, 113]. As software systems grow in size and complexity, systematic approaches to documenting and analyzing design decisions become even more important to preserve institutional knowledge and ensure consistency and maintainability.

In Section 3.1.1, I focus on related work about documenting design decisions. The section explores the methods and practices used to record the reasoning behind design choices and reviews techniques and tools that support the formal recording of these decisions, emphasizing the importance of documentation.

Next, in Section 3.1.2, I examine the structured frameworks used to categorize and describe design decisions. Various models have been proposed to classify design decisions based on factors such as scope, impact, and abstraction level, not only for documenting purposes but also for further processing.

Consequently, Section 3.1.3 addresses advances in automating the identification and evaluation of design decisions. With the increasing use of machine learning, static analysis, and data mining techniques, it has become possible to retrospectively extract design decisions from code, issue trackers, and documentation. This section reviews tools and methodologies for automatic recovery and analysis, highlighting their potential to reduce manual effort and improve the quality of software designs.

Lastly, I conclude and discuss implications for my dissertation in Section 3.1.4.

3.1.1. Documenting Design Decisions

In their work, Jansen et al. [113] focus on the challenge of recovering and documenting ADDs in software systems after they have already been made. Software architecture documentation often focuses primarily on the system's structural aspects, i.e., components and their interactions. At the same time, the rationale behind design decisions remains undocumented or gets lost. The authors propose a method for systematically recovering and documenting these design decisions retrospectively to improve the system's quality and maintainability.

The approach ADDRA outlined by the authors emphasizes the use of a combination of recovery techniques and the tacit knowledge of the original architect to recover ADDs. The proposed method was validated through a case study, demonstrating its feasibility in real-world settings and showing that it can be used to improve software maintenance, system evolution, and communication among stakeholders.

The approach ADDRA is similar to their approach Archium [112, 114] using a similar architectural decision model. Still, Archium combines requirements, architecture, and implementation models into a single unified model for improved traceability.

While Jansen et al. [113] tackle the challenge of retrospective decision recovery, Tyree et al. present a template for documenting architectural design decisions tailored towards forward engineering [275].

The survey of the state of research and practice regarding the documentation of software design decisions by Alexeeva et al. [4] from 2016 provides a broader perspective on existing techniques and frameworks for documenting decisions proactively. Alexeeva et al. focus on the importance of capturing and maintaining the reasoning behind architectural and design choices during software development. They argue that while much attention is given to documenting software architecture, the rationale and trade-offs made during design are often overlooked. This leads to issues like knowledge loss, difficulties in maintaining software, and the inability to understand why certain decisions were made.

Alexeeva et al. also discuss the existing techniques and tools used to document architectural decisions, highlighting both strengths and limitations. They point out that while tools exist for design decision documentation, they are not widely adopted in industry, and only minimal support for brownfield development, i.e., legacy systems, exists. One reason is the perceived high cost of maintaining decision documentation, especially when development teams are pressed for time. The authors emphasize the need for lightweight, integrated tools to capture decision rationale in the normal software development workflow.

A significant contribution of this work is its analysis of the key challenges in design decision documentation. These challenges include the evolving nature of design decisions, difficulties integrating decision documentation with other software artifacts, and the need to ensure that the documentation stays relevant and up-to-date as the system evolves. The authors suggest that the lack of a standardized approach to documenting design decisions contributes to these problems and propose that further research is needed to develop methodologies that address these challenges.

In conclusion, Alexeeva et al. provide valuable insights into the importance of design decision documentation for software engineering, while highlighting the gaps in current research and practices. They advocate for future efforts in creating more practical, efficient, and automated solutions to support decision documentation, aiming to make it more accessible and beneficial for development teams in real-world settings. The paper calls for more empirical studies and collaboration between academia and industry to overcome the obstacles associated with this crucial aspect of software design.

To analyze design documentation in practice, Weinreich et al. conducted an expert survey in 2015 [285]. Their goal was to identify the potential kinds of architectural decisions, their drivers, and the sources of their documentation as used in practice. They asked software architects, developers, and persons in similar roles from 10 different countries in Europe and the US. The results reveal that while design decisions are typically captured, their rationale is often neglected. The study identifies personal experience, user requirements, and quality attributes as key decision drivers, and highlights the need for better decision documentation practices and support for knowledge recovery.

The authors suggest that decision-making could be improved by formalizing documentation processes and reducing bias from personal experiences. They recommend using patterns, classifying decisions based on requirements, and tailoring documentation practices to team size. The authors see potential future work in the area of recovering and maintaining architectural knowledge and improving decision documentation efficiency.

Making decisions is not something that only one person does on their own at a single point in time, as Hesse et al. state [95]. Therefore, they present a tool for documenting design decisions collaboratively and incrementally. They developed a decision documentation model to capture decisions and their relations. Decisions contain components that can be the decision's problem, solution, context, or rationale. Their tool, DecDoc, is built upon the knowledge management tool UNICASE [29] that supports modeling of, e.g., use cases and UML diagrams. The authors evaluate this tool in design sessions with professional software designers. According to their retrospective analysis of the decision-making process, the tool is feasible for documenting complex decision knowledge collaboratively and incrementally in DecDoc.

While all this work tackles the documentation and maintenance of ADDs, they do not yet consider or tackle consistency explicitly. Inconsistencies are only incidentally resolved in incremental processes, and no effort is expended to actively detect inconsistencies in these processes. Moreover, this work either does not consider the kinds of design decisions or does not organize these decisions into well-defined categories and structures in the form of taxonomies or similar. I discuss further shortcomings of the related work and implications for my work in Section 3.1.4.

3.1.2. Models and Taxonomies

There are several approaches that either model design decisions or define a classification schema or taxonomy for design decisions. These models and classifications can help process design decisions. These should help detect inconsistencies and improve TLR in my use case.

Kruchten [146] introduces an ontology of architectural design decisions in software-intensive systems. The ontology defines a framework for understanding the different dimensions of design decisions, offering architects a way to approach the complexities of software architecture. Kruchten categorizes design decisions into three main types: existence, property, and executive design decisions.

Existence design decisions define the presence or absence of architectural elements. Kruchten further divides these into three subcategories: structural, behavioral, and ban decisions. Structural and behavioral decisions are about creating and interacting system elements, respectively. Structural decisions focus on the system's composition, while behavioral decisions relate to how these elements interact. Ban decisions, in contrast, explicitly dictate that certain elements should be excluded from the design.

Property design decisions describe the traits or qualities of the system. Kruchten distinguishes between positively and negatively stated property decisions, referring to the former as design rules or guidelines and the latter as constraints. These decisions shape the system's non-functional characteristics like performance, reliability, and security.

Finally, executive design decisions involve high-level decisions influenced by the business context. These decisions impact the development process, the organization, and the people involved and extend to the selection of technologies and tools. Executive decisions are often driven by external factors like market demands, organizational priorities, or resource availability, and they play a critical role in aligning architectural choices with business goals.

In 2013, Miesbauer and Weinreich [182] conducted an expert survey with a detailed analysis of the types and distribution of design decisions made by software developers. They observed that existence decisions, which pertain to whether a particular element or feature should be included in a system, are the most commonly encountered by developers. In contrast, executive decisions involving high-level strategic choices, such as selecting major technologies or frameworks, accounted for only about a quarter of all design decisions. Furthermore, property decisions, which involve defining the characteristics or parameters of system components (e.g., setting specific technical attributes), only occur in around 10% of the overall decisions.

Miesbauer and Weinreich also categorized design decisions into four distinct levels based on their scope and impact: implementation, architecture, project, and business. Decisions at the implementation level deal with low-level choices related to the coding and technical aspects of the system's functionality. Architectural decisions shape the system's overall structure and define key components, their relationships, and how they interact. Project-level decisions concern management and organizational factors, such as scheduling, resource allocation, and team structures. Lastly, business-level decisions are at the highest level and focus on aligning the technical design with broader business goals, market strategy, and customer requirements.

Their work highlights the complexity and multi-layered nature of design decisions in software development, emphasizing that while developers frequently engage with lower-level technical decisions, higher-level business and strategic considerations play a critical, though less frequent, role.

Jansen et al. published the aforementioned work, introducing an architectural decision model [112, 114, 113]. In their work from 2005, Jansen and Bosch [112] regard software architecture by defining it as a composition of architectural design decisions made throughout the development process. Rather than viewing architecture solely as a static structure of components and connectors, they emphasize the importance of capturing the rationale, trade-offs, and dependencies behind design decisions. This decision-centric approach reflects the dynamic nature of software architecture, which evolves as new decisions are made and when previous ones are revisited. Focusing on these decisions aims to provide a more comprehensive and traceable understanding of how software architectures are shaped. Similarly, Hesse et al. introduces their mentioned decision documentation modeling [95].

To support this perspective, Jansen and Bosch introduce *Archium*, a tool that explicitly models the relationships between design decisions and the architecture itself. Archium traces how individual design decisions impact the overall architecture, tracking changes in the form of deltas and design fragments as they evolve. The authors argue that this method improves architectural transparency and reduces knowledge vaporization. Knowledge vaporization is a common issue where critical architectural knowledge is lost as teams change or projects evolve. By making architectural decisions explicit and traceable, their approach helps maintain the integrity of the architecture throughout the software lifecycle, facilitating better communication, documentation, and long-term maintenance.

Falessi et al. [59] present a framework that systematically associates design decisions with their underlying goals, justifications, and available alternatives. This framework's primary goal is to enhance the comprehensibility and traceability of the design process, facilitating better maintenance and evolution of complex systems. By explicitly capturing the rationale behind design choices, the framework enables developers and stakeholders to understand what decisions were made, why they were made, and how the available alternatives were evaluated. This structured approach to documenting design decisions supports knowledge transfer and reduces the risk of errors during system maintenance and future modifications.

The authors argue that many issues related to system evolution arise from a lack of transparency regarding the rationale behind key design decisions. The framework introduced by Falessi et al. addresses this gap and improves decision-making processes in software development by providing a formal method to capture design rationales. The framework also promotes the reuse of design knowledge by documenting the trade-offs and considerations that influenced past decisions. This, in turn, helps developers make more informed choices when adapting systems to new requirements or technologies, ultimately leading to more robust and maintainable software architectures.

In 2007, Zimmermann et al. [305] proposed a structured approach to managing architectural decisions in enterprise application development, drawing upon Kruchten's ontology and the use cases defined by Falessi et al. They formalize the decision-making process into three distinct phases: decision identification, decision-making, and decision enforcement. This approach aims to streamline how architects address complex decisions by creating a reusable framework that guides the decision-making process across multiple projects. Providing structure to decisions enables architects to handle the complexity and variability inherent in enterprise-level applications.

To manage these decisions effectively, Zimmermann et al. [305] group similar architectural decisions under shared topics. Each topic is then mapped to one of three levels of abstraction: the conceptual, the technology, or the asset level. The conceptual level deals with high-level architectural principles and design philosophies, while the technology level addresses selecting and evaluating technology stacks. The asset level focuses on concrete, reusable artifacts, such as software components or services, that can be leveraged across different projects. This structured categorization supports more consistent decision-making and also facilitates improved communication among stakeholders by clarifying how different decisions relate to one another.

Ultimately, the authors argue that this reusable decision model enhances both the efficiency and the quality of architectural decision-making in enterprise application development. By capturing and reapplying best practices through this decision model, organizations can reduce costs, mitigate risks, and accelerate the development of robust, scalable enterprise systems.

In their 2009 survey on architectural design decision models and tools, Shahin et al. [245] analyzed existing approaches for capturing and documenting architectural design decisions. The authors identified four key elements common to all nine models they examined: decisions, constraints, solutions, and rationales. These elements represent the fundamental building blocks used to describe and justify the design process, illustrating how architects arrive at a particular solution within a set of constraints.

Despite this commonality, the survey revealed that the models differ significantly regarding the specific terminology and focus. Some models emphasize the decision-making process itself, while others prioritize the rationale behind decisions or the resulting architectural solution. This variability reflects the diverse ways in which architectural knowledge is captured, structured, and communicated, depending on the model's intended application.

Shahin et al. concluded that, although the models are conceptually similar, their differences create interoperability and tool support challenges. The authors call for more standardized frameworks and tools to ensure consistency across different approaches, ultimately facilitating better decision-making and documentation practices in software architecture.

While some of this related work is close to mine, several shortcomings exist. I discuss these shortcomings and how the related work affects my work in Section 3.1.4.

3.1.3. Automatic Recovery and Analysis

Approaches that automatically identify or analyze design decisions have the benefit that they reduce or even omit the costly manual effort needed to recover design decisions using a domain expert.

The approach *RecovAr* proposed by Shahbazian et al. [244] aims to automatically recover architectural design decisions by leveraging software artifacts like issue tracker entries and data from the version control repository. The core idea behind *RecovAr* is to address the challenge of tracing and documenting design decisions in large-scale software systems, where architectural evolution often occurs incrementally and is recorded indirectly through changes in code and issue management systems.

First, the approach recovers the system's static architecture by analyzing various versions available in the version control system and constructing snapshots of the architecture at different points in time. By comparing these snapshots, it identifies architectural changes, mapping how the system evolves. The second phase involves linking issues from the issue tracker to the affected architectural elements, providing a traceable connection between system requirements, development activities, and the resulting architectural adjustments.

Finally, *RecovAr* synthesizes these insights into a decision graph, where architectural changes and issue-to-architecture trace links are integrated. This decision graph represents the architectural design decisions made throughout the system's evolution, making implicit decisions explicit and facilitating better understanding, maintenance, and future evolution of the system's architecture. The approach offers a systematic way to recover and document architectural design decisions, which can be invaluable for developers and architects in managing the complexities of evolving software systems.

Bhat et al. [20] also focus on automatically extracting design decisions from issue management systems. They use a machine learning-based approach to address this challenge. Building on Kruchten's classification schema [146], they train a classifier using a dataset of 1500 annotated issues. Their method employs SVMs to accurately detect the presence of existing design decisions, achieving a detection accuracy of 91.29% in their evaluation. Moreover, their classifier successfully classifies these decisions into the aforementioned subclasses with an accuracy of 82.79%, highlighting the effectiveness of machine learning techniques in structuring and identifying key design elements within issue management systems.

In addition to their machine learning model, Bhat et al. developed the *Amelie Decision Explorer (ADeX)* [21], a tool designed to further assist software architects in managing design decisions. *ADeX* can recover design decisions from natural language text by analyzing issue descriptions and comments within issue trackers. Apart from extraction, the tool also supports the decision-making process by suggesting potential solutions, presenting alternative options, and referencing similar past design decisions. This functionality provides architects with a comprehensive decision-support system, enhancing the overall quality and consistency of design choices in software development processes. By automating both the extraction and analysis of design decisions, Bhat et al.'s approach aims to streamline the often complex and time-consuming task of decision documentation and recovery in software projects.

Kleebaum et al. [137] also address the problem of identifying rationale in software development processes by focusing on issue tracking and version control systems, specifically targeting platforms like Jira. They propose a machine learning approach using a logistic regression model trained on GloVe embeddings [215] (see also Section 2.3.3). The goal is to classify textual data such as Jira issues, descriptions, comments, and commit messages into several critical categories for understanding the rationale behind software development decisions. These categories include identifying decision problems, design decisions, alternatives, and pro- and con-arguments. In their evaluation, the classifier can achieve an F_1-score of 39% for detecting design decisions, a significant but still challenging result given the complexity of natural language in technical contexts.

Similarly, Li et al. [158] employ machine learning techniques to extract design decisions automatically. In their work, they focus on developer mailing lists, particularly on identifying decision-related content in e-mails. Their approach utilizes an SVM-based classifier to detect whether a given sentence within an e-mail contains a design decision, achieving a promising F_1-score of 75.9%. This work is notable for focusing on mailing lists as a key source of decision-making documentation. These lists often contain important but informal discussions about design choices, trade-offs, and requirements.

Building on this foundation, Fu et al. [69] enhance decision classification through the use of ensemble learning methods and feature selection techniques. Their work extends beyond binary classification to a multi-class classification problem, aiming to categorize decisions into five distinct classes: design, requirement, management, construction, and testing decisions. By employing an ensemble of classifiers using NB, LR, and SVMs along with 50% of features selected, Fu et al. achieve a weighted F_1-score of 72.7%, demonstrating the effectiveness of combining multiple algorithms and feature selection for improving decision identification in mailing list discussions.

3.1.4. Conclusion and Implications

My taxonomy aims to support and improve tasks such as TLR and, in particular, consistency checking in software architecture documentation. While some discussed related work, especially regarding models and taxonomies in Section 3.1.2, helps by differentiating the kinds of design decisions, the approaches are not fully aligned with these purposes. These approaches classify ADDs broadly and help for certain purposes like improving communication or identifying assets that can be reused across different projects. Yet, they are too broad or focus on abstraction levels that are unsuitable for my intended purposes. For example, the approach by Shahin et al. [245] can only partly help for TLR and is too vague for concrete inconsistency analyses. Effective consistency analysis requires distinguishing between different types of design decisions. For instance, verifying the presence of a component can typically be done by querying the architectural model, as components are often treated as first-class entities. In contrast, layering is not usually explicitly modeled as an entity, meaning that the structure of the model must first be examined to ensure compliance with layering principles. Kruchten's ontology [146] is the work that closest fits my purposes. However, the categories are still too broad, as concepts like structural existence combine subsystems, layers, partitions, and components, all needing different analyses to check their consistency. Still, Kruchten's ontology is a good starting point for my taxonomy (see Chapter 4).

The studies about automatic recovery and analysis from Section 3.1.3 highlight the potential of machine learning to automate the extraction of decision-related information from unstructured communication platforms, such as issue trackers and mailing lists. This ability can be crucial as decision-making plays a central role in software development processes. Automating the extraction of this information can enhance the documentation and comprehension of software design choices over time, contributing to better-informed decision-making. These contributions are also important in addressing the challenge of bridging human decision-making with machine-understandable documentation, particularly in the context of software engineering tools.

While many of the discussed approaches could be applied to classify architectural design decisions presented in this work, direct comparisons or usage are not feasible due to differences in scope and input types. *RecovAr* by Shahbazian et al. [244] uses historical data in combination with information from issue trackers to identify ADDs, focusing on making implicit decisions explicit. Similarly, the related work classifies issues (cf. [137, 20]) or analyze mailing lists (cf. [158]). I focus on software architecture documentation, and all these other artifacts have different forms and amounts of information available that can be used for classification. Additionally, the taxonomies employed in these studies differ from my own, aside from the shared goal of identifying the existence of design decisions. Taxonomies can affect the choices of what techniques can and should be used and, e.g., which features might positively or negatively impact the classification. For example, the classifiers of the ensemble approach by Fu et al. [69] require a careful selection of features, and the authors showed that in their case of classifying decisions in mailing lists into five distinct classes, the best results are achieved if 50% of the features are selected. This selection can differ for my taxonomy and for software architecture documentation. Consequently, a research gap exists for automated classification of ADDs in software architecture documentation into fine-grained classes. Nevertheless, most underlying techniques from the presented approaches are adaptable and can be candidates for integration into the automated classification approach outlined in Section 4.4.

3.2. Traceability Link Recovery (TLR)

In their publications, Borg et al. [25], Souali et al. [248], Aung et al. [12], and Pauzi et al. [211] review the literature about TLR, each with a different focus. These publications say automated TLR covers different domains, use cases, and techniques.

Borg et al. [25] recognize IR techniques as one of the major techniques for TLR and conclude that many approaches are based on typical IR techniques. According to them, from their viewpoint in 2014, IR techniques dominated the trace link recovery scene for more than a decade. The pioneering work was made by Antoniol et al. [9] in 1999 along with their subsequent work, e.g., from 2002 [8]. Borg et al. reveal that from 1999 onwards, there is an increasing number of IR-based TLR publications. According to the authors, most approaches cover TLR between requirements, followed by TLR between requirements and code and between requirements and tests.

In the example-driven work by Souali et al. [248], the authors identify different domains that use TLR. Moreover, they identify several commonly applied techniques to manage traceable items, including IR techniques, ontologies, graphs, and models. However, the authors use a broad definition of managing that includes not only the (automated) recovery of trace links but also manual modeling of trace links and even knowledge modeling.

Aung et al. [12] systematically review the literature on TLR for software change impact analysis. As a result, they identify IR-based approaches, heuristic-based approaches, machine learning-based approaches, and deep learning-based approaches. They also found out that most publications focus on unidirectional TLR and that most researchers evaluate their approaches using experiments, in contrast to case studies, with open-source datasets.

Pauzi et al. [211] focus on the application of NLP techniques for TLR. In their systematic literature review, they categorize the approaches into three tiers based on the amount of NLP involved in the presented approaches. Approaches of the first tier only perform basic complexity tasks to process text to analyze the text syntax. The authors identify the techniques Part-of-Speech (POS) tagging, stemming, lemmatization, tokenization, stopword removal, regular expressions, keyphrase extraction, and TFIDF. The second tier consists of approaches that perform basic to intermediate NLP tasks, including word embeddings and topic modeling. Consequently, the approaches utilize more semantics. The identified approaches of this category use LSI and LDA, embeddings, VSMs, topic modeling, translation (language), NER, and document, sentence, and word similarity. For the third tier, approaches need to perform advanced tasks and use techniques such as deep learning and ANNs, extending the second tier further. In particular, more recent publications can be found in the second and third tier.

Based on these literature reviews and my contributions, I look into related work about TLR in three major sections. First, Section 3.2.1 covers information retrieval-based approaches, including those that use VSMs and LSI. Second, Section 3.2.2 covers machine-learning approaches and those approaches that use deep learning, with a focus on approaches that use, e.g., ANNs, genetic algorithms, or language models. Lastly, Section 3.2.3 looks into approaches that use intermediate artifacts, as I also use intermediate artifacts in my transitive approach. In Section 3.2.4, I discuss the related work, how my work relates to it, what the distinctions are, and what I can infer from it for my work.

3.2.1. Information Retrieval-based Approaches

Apart from mostly automated approaches that follow below, there are also approaches with a strong manual component, like the approach presented by Tang et al. [262]. The presented approach supports TLR between requirements and architecture design by using an ontology. Specifications and architectural artifacts can be defined manually in the ontology and are documented in a semantic wiki. If developers manually manage their requirements and the design, they can make use of the semantic media wiki and different queries to find different corresponding trace links.

One of the most prominent publications for automated TLR is the IR-based work by Antoniol et al. from 2002 [8]. In their work, Antoniol et al. recover trace links between code and natural language documentation such as design documents, requirements, or manuals. The authors propose an automated approach to recover these trace links by analyzing both the source code and the documentation. They use two different models, a probabilistic IR model and a VSM. The probabilistic model computes the probability that a document is related to a source code component using approximations and estimated unigram probabilities. For the VSM, they use the TFIDF to generate the vectors and calculate the similarity between two artifacts by calculating the cosine similarity between the corresponding vectors. In their experiments, Antoniol et al. apply their method to case studies, showing how their approach can effectively identify trace links. Despite their promising results with high recall, the authors also acknowledge that precision is low, not all links can be perfectly recovered, and human judgment is still necessary to verify the results.

VSMs are a popular technique for TLR [80, 171]. The problem with VSMs is that they are designed for text-based data focusing on natural language texts. As such, they cannot be applied as efficiently on source code and their application on SAMs is very limited. When applied to source code, VSMs can only reflect some parts of the code semantics and disregard, for example, structural information like inheritance. To resolve these issues, several approaches try to reuse structural information to capture contexts in source code [147, 204, 301].

For example, Panichella et al. [204] integrate structural information with IR-based methods. They also regard structural links as transitive for TLR purposes. This means that if there is a trace link between a documentation artifact D and a source code artifact S1 that has a structural link to another source code artifact S2, a second trace link is introduced between the D and S2. Structural information, such as function calls and inheritance in source code, complements the textual similarities identified by IR methods like the VSM and probabilistic Jensen Shannon model Divergence (JSD). The authors introduce a feedback mechanism wherein software engineers verify candidate links suggested by IR-based methods. Structural information is applied only after a user manually confirms that these links are correct, ensuring that additional links are derived without introducing incorrect or irrelevant ones. This conditional use of structural data prevents performance degradation when poor initial IR results are blindly enhanced with structural connections. They call this method a User-Driven Combination of Structural and Textual Information (UD-CSTI). In their evaluation, they tested against traditional IR methods and a naive combination approach (O-CSTI) across three software systems. The results showed a significant performance improvement with UD-CSTI, particularly in reducing false positives and enhancing precision at lower recall levels, where precision improved by up to 20%. This method demonstrated an ability to better prioritize relevant links while reducing the manual effort required by engineers to look through irrelevant ones. However, the approach struggles with recall, especially in cases where links have low textual similarity. Overall, the approach was more stable and consistently outperformed the alternatives, highlighting the value of incorporating structural information, in particular with expert feedback. While the approach is a good starting point for TLR that involves code, its use and its transferability to artifacts like SAMs are unclear and questionable. This stems mainly from the different information details between code and SAMs. As such, the applicability on my intended use cases is limited. Moreover, the approach requires manual steps, which is a big distinction from my automated approach.

Kuang et al. [147] investigate the role of method data dependencies for TLR between software requirements and source code. They explore whether data and method call dependencies can support the recovery. While other research has looked into call dependencies, the authors mention that methods can also depend on shared data without directly calling each other. To use these dependencies, they introduce the Inverse Data Type Frequency classifier that is similar to the inverse document frequency. In an experiment with five software systems, they show that data dependencies can be just as important as call dependencies in supporting TLR and that combining both types of dependencies significantly improves the accuracy. The combination of method call and data dependencies led to a marked increase in precision and recall, with an average improvement of 18.88% over using call dependencies alone. The approach's robustness was also tested in scenarios with incomplete and erroneous traceability data, showing that it maintained its effectiveness even when faced with traceability inaccuracies. Again, the transferability to non-code artifacts like SAMs is unlikely, making the approach not applicable for such use cases. While the approach can be an option for TLR between SAD and code, the other, subsequent approaches are more promising.

The approach by Zhang et al. [301] uses synonyms and verb-object phrases from the natural language artifacts and structural information from the source code to calculate a similarity score. In the first step, the approach uses WordNet and calculates the maximum similarity of pairs of possible synsets for two words, using this as a similarity value. Then, in the second step, the approach extracts verb-object phrases from text and code and, for each pair of phrases, calculates a similarity based on the weighted similarity of the pair of verbs and the pair of objects in the corresponding phrase, using the similarity values from the first step. To calculate the similarity between requirements and code, the approach then creates a VSM where the values are based on weighted words or phrases. The weighting is done with TFIDF and the aforementioned similarity value from the second step. The resulting vectors can then be compared using cosine similarity. Afterward, the approach updates the values using structural information that increases similarity based on inheritance and call relations within the source code. In their evaluation of the four projects eTour, iBooks, SMS, and EasyClinic, the authors conclude that their approach is more effective than the approaches by Panichella et al. [204], Antoniol et al. [8], and Zou et al. [306]. Additionally, the authors state that improved results come from more accurately extracting terms and from the structural information. Due to the promising results, I use the approach as a baseline in my evaluation (see Section 5.5.3.1).

In contrast to these VSM-only approaches, Gethers et al. [80] combine the techniques VSM, Relational Topic Modeling (RTM), and JSD in their approach. The approach makes use of the empirical finding that certain IR techniques, such as VSM and JSD, while similar to each other, produce different and complementary trace links compared to newer methods like RTM. The core idea is, therefore, a combined approach that enhances accuracy. This hybrid approach exploits the strengths of each method, particularly RTM's ability to model both document content and relationships, which traditional methods miss. This hybrid method was evaluated through case studies on six software systems, and the authors report that the combination of RTM with canonical methods outperforms the use of any standalone technique. The combination led to statistically significant improvements, especially when the methods were orthogonal. Additionally, the study examined the impact of factors like artifact type and language, finding that the hybrid approach was robust across different scenarios, confirming the advantage of combining complementary IR methods. The results of subsequent approaches are more promising, but the underlying idea is interesting and I partly transfer the idea to my approach by employing multiple similarity metrics.

Lohar et al. [162] identify that there is a wide selection of algorithms with different configurations and features. They explore different combinations of IR techniques to recover trace links between various artifact types, such as requirements and code, using a pipe-and-filter architecture. The authors use a ML-based genetic algorithm to identify the best configuration based on a training set of validated trace links. This also means that for new projects, they always need initial trace links and expand the established links instead of recovering them all. In their evaluation covering multiple projects and domains, their approach achieved high MAP scores of up to 0.80 and 0.86 when recovering trace links between use cases or test cases and code. Nonetheless, the authors identified a strong dependency on configurations, projects, and artifacts, as the best results for each project were obtained with distinct configurations. Consequently, this can mean that there is likely no silver bullet for IR-based TLR.

One common challenge for IR-based approaches is the handling of semantically similar expressions like synonyms. Different methods and techniques have been used to address this problem, including the use of latent semantic indexing [175], incorporation of synonym coefficients for similarity calculations [93], word embeddings [35, 97], and the construction of semantic-relationship graphs [239, 238].

In 2003, Marcus et al. [175] present a method for recovering trace links between system documentation and source code using LSI. The method automatically identifies links by analyzing the semantic content of both the documentation and the source code. This approach leverages all comments and identifier names in the source code, avoiding the need for complex preprocessing, such as parsing or predefined vocabularies. By projecting documents into a reduced space with LSI, they calculate the similarity between source code and documentation using cosine similarity. The evaluation involved two case studies: the LEDA library and the Albergate system. For LEDA, LSI achieved a precision of 71% with 42.63% recall using a similarity threshold, and precision-recall trade-offs were observed across different thresholds. LSI outperformed previous methods, especially when higher precision was needed. Similarly, in the Albergate experiment, LSI demonstrated comparable performance to probabilistic and VSM-based approaches. The authors claim that the method proved to be flexible, efficient, and effective, even without using syntax-based parsing or language-specific tools. Other approaches like S2Trace by Chen et al. [35] or FTLR by Hey et al. [97] build upon insights from this work. However, this approach is targeted at source code and uses specific information from artifacts like comments in source code. This limits its use to my use case as, for example, SAMs do not contain this information.

Hayes et al. [93] incorporating synonym coefficients to improve similarity calculations. The approach expands traditional TFIDF methods by introducing a thesaurus-based approach, where pairs of synonymous terms are assigned similarity coefficients to account for vocabulary differences between documents. Their tool, RETRO, enhances the matching process by considering not just keyword frequency but also synonymous relationships between terms. The paper also evaluates LSI to capture conceptual similarities rather than just word matches and also provides a method that incorporates relevant feedback from users. The Evaluation of the methods shows that the synonym-enhanced TFIDF outperformed traditional TFIDF, particularly in balancing recall and precision. RETRO was assessed using data sets, including MODIS and CM-1, showing that synonym-enhanced methods and feedback mechanisms significantly reduced false positives and increased accuracy for TLR. Accordingly, the authors conclude that synonym coefficients improved tracing, but more work needs to be done in this direction.

In their approach S2Trace, Chen et al. [35] use sequential semantics in software artifacts and combine them with word embeddings to generate document embeddings. The authors criticize that other existing approaches rely heavily on word embeddings, which capture semantic relationships between terms in software artifacts but often overlook the importance of word order. The authors address this by mining sequential patterns that capture the lower-level semantics of terms and then combining them with word embeddings to generate document embeddings. These embeddings should better represent both the content and structure of software artifacts. The idea is that by learning from both the terms and the sequential relationships between them, S2Trace can identify more accurate trace links between requirements, improving over methods that use word embeddings alone. The evaluation of S2Trace is conducted on five public datasets from different domains: GANNT, CM1-NASA, eTour, iTrust, and EasyClinic. They compare the performance of S2Trace with three baseline approaches: LSI, Word2Vec, and WQI, a supervised ML method that uses learning to rank to improve the Word2Vec method by query expansion and weighting. The results demonstrate that S2Trace outperforms these baselines, particularly in precision and F-measure. The study shows that the integration of sequential patterns significantly improves trace link accuracy, and the use of Principal Component Analysis (PCA) further enhances the results by reducing redundancy in the document embeddings. Additionally, parameter sensitivity analysis shows the impact of minimum support threshold, distance threshold, embedding dimensions, and training epochs on the performance of S2Trace. This approach promises good results, and the techniques employed, such as sequential patterns, look promising. However, due to the nature of these models, sequential patterns have limited use for artifacts like formal software architecture models. Much effort is required to make sequential patterns applicable to models with no guarantee of success. Moreover, while S2Trace could outperform the three baseline approaches, Hey et al. [97] have shown the limits of S2Trace, significantly outperforming S2Trace with their approach.

Hey et al. [97] utilize IR-based metrics to establish trace links between requirements and source code classes using word embeddings in a more fine-grained approach. In FTLR, requirements are first broken down into individual sentences, and code elements are represented by public method signatures, enriched with the containing class name and accompanying documentation comments. Both types of artifacts undergo preprocessing, which includes stop word removal, lemmatization, and filtering based on word length. Each preprocessed artifact is then represented as a bag of embeddings, using pretrained word embeddings from fastText [183]. The trace link generation then follows a two-step process. First, fine-grained elements are mapped using the Word Mover's Distance (WMD) between their bag-of-embeddings. Next, the resulting element trace links are filtered based on a defined threshold. FTLR aggregates these mappings at the class level by applying a majority vote, linking a class to the most frequently mapped requirements across its methods. In their evaluation, the authors report an average F_1-score of 0.327 for this approach using the best-performing configuration. While the approach cannot be directly applied to all of our use cases, it is a good baseline candidate, in particular for TLR between SAD and code.

Schlutter et al. [239, 238] introduce in their work a novel approach that uses semantic-relationship graphs and spreading activation for TLR between requirements. The authors claim that traditional approaches face limitations either due to their reliance on large labeled datasets or their inability to capture semantic context. To tackle these issues, the authors present a fully automated approach that converts natural language requirements into a semantic relation graph, where terms and their relationships are represented as vertices and edges. Their approach then applies semantic search using spreading activation that propagates activation pulses over the graph, identifying trace links based on the relevance of connected nodes that are presented in a ranked candidate list. The approach was evaluated with the five common TLR datasets Infusion Pump, CCHIT-2-WorldVista, GANNT, CM-1, and WARC. In the evaluation, the authors adjust the graph structure and algorithmic parameters to optimize performance. The evaluation metrics included mean average precision (MAP), lag, and recall, focusing on providing engineers with ranked candidate lists of trace links. The method achieved promising results, with a mean average precision of around 50%, a Lag of approximately 30%, and a recall of 90%, depending on the dataset and configuration. These results demonstrate the approach's effectiveness, especially in ranking relevant trace links near the top of the candidate list, thus reducing manual effort in trace link identification. This approach promises good performance but focuses solely on natural language text, and their approach only supports English. It is unclear, how the applied methods, i.e., the semantic relation graphs and spreading activation, can be applied to models or code. The transformation of the input artifacts into the semantic relation graph is one of the key contributors to the approach's success. Therefore, the approach is not directly applicable to my use cases, and more research is required to apply it.

Gao et al. [77] claim that other related work does not properly treat context, especially related to terms that co-occur. Their approach, TAROT, introduces so-called biterms and uses them to improve TLR between requirements and code. Depending on the artifact, biterms have slightly different meanings. Biterms in natural language text refer to two terms that have a grammatical relationship within a sentence. Similarly, biterms in code represent any combination of two terms within identifiers, and code comments are treated just like text. Their approach identifies biterms in artifacts and compares the biterms in different artifacts to find so-called consensual biterms. These are weighted based on their frequency and location. Different IR models, including VSM, LSI, and JSD, are employed to create candidate trace links between requirements and code and weighted using global and local weights. In their evaluation on nine projects (iTrust, GANNT, Maven, Pig, Infinispan, Drools, Derby, Seam2, and Groovy), TAROT achieves, for example, a MAP of 0.62 for the iTrust project when using VSM. When TAROT is combined with the CLUSTER enhancing strategy (cf. [76]), the MAP improves to 0.73 using LSI. This approach is promising and, with some adjustments, can be applied to my use cases. The approach uses a different strategy to tackle TLR by focusing on biterms. To compare this approach to my contributions, I use the approach as a baseline in the evaluation (see Section 5.5.5).

Overall, the field of TLR has seen a variety of IR-based approaches, ranging from manual methods to fully automated systems. Early methods, like those presented by Antoniol et al. [8], rely heavily on probabilistic models and VSMs, demonstrating effectiveness in recovering links but struggling with precision. Later approaches, such as UD-CSTI by Panichella et al. [204], incorporate structural information from source code to enhance results, improving precision while reducing false positives. Innovations like the use of data dependencies by Kuang et al. [147] and the integration of semantic similarities from WordNet by Zhang et al. [301] further push the boundary of TLR. Hybrid approaches combining multiple IR techniques, such as the RTM-VSM integration by Gethers et al. [80], show marked improvements in accuracy. Despite advancements, challenges remain, particularly in handling semantically similar expressions and adapting to diverse artifact types, as seen in approaches using word embeddings and semantic graphs. Overall, while no silver bullet exists, combining complementary techniques and incorporating structural, semantic, and contextual information has proven essential in advancing TLR. In Section 3.2.4, I further discuss these works in relation to my contributions.

3.2.2. Machine Learning-based Approaches

As for most areas, the progress made by ML and language models has significantly advanced TLR approaches. There are different kinds of usages for ML. Some approaches use ML to find optimal parameters for IR-based approaches, e.g., by applying genetic algorithms. Other approaches directly use ML techniques like ANNs or LLMs to tackle TLR. For example, many more recent approaches try to understand and interpret the semantics more thoroughly, using NLP techniques as well as deep learning. Lastly, some approaches combine IR-based approaches and ML approaches. In the following, I present approaches from these categories.

Several approaches use ML techniques to improve IR methods. For example, recent approaches like the approach by Rodriguez et al. [233] and by Ghannem et al. [82] report good results by combining IR with a genetic algorithm for TLR between software requirements and source code elements.

Ghannem et al. [82] experimented with the *Non-dominated Sorting Genetic Algorithm (NSGA-II)* [48] using semantic similarity and maintenance history to recover trace links between requirements and the source code. For semantic similarity, they calculate a similarity score based on TFIDF and cosine similarity. For maintenance history, Ghannem et al. use the two measures frequency of change and recency of change. They base their approach on semantic similarity and change history by extracting the frequency and recency of changes from the source code history. Starting with an initial random population, they use NSGA-II to maximize the similarity score between individuals, favoring artifacts that changed recently and more frequently. In their evaluation, the authors apply their approach to the three projects, LEDA, Albergate, and eTour, and report promising results.

Rodriguez et al. combine the two IR metrics Jaccard similarity and cosine similarity, weighted by TFIDF, and use NSGA-II to explore and find trace links. The metrics assess the semantic overlap between the textual content of the requirements and the source code, and the evolutionary algorithm NSGA-II is adapted to iteratively explore the large solution space, represented as a population of candidate trace links, to converge on an optimal set of solutions that balances between the two IR metrics. According to the authors, this method enables the automatic generation of trace links without extensive manual intervention.

The reported evaluation results demonstrate promising performance, with the approach yielding average precision and recall rates above 75% across different datasets. For example, when evaluated on datasets such as Event-Based Traceability (EBT), Albergate, and eTour, the algorithm consistently performed well, with precision values typically higher than recall in larger datasets. Compared to other approaches, they can achieve much higher recall with values up to 77%. The paper emphasizes the potential of using NSGA-II as a more precise and complete alternative to traditional IR techniques, noting that this multi-objective optimization algorithm improves both the precision and recall of TLR when compared to methods that rely solely on individual IR measures. The authors claim that this success suggests that combining genetic algorithms with IR techniques offers a viable path to better automate TLR. However, one issue is the approach's randomness, which is highly reliant on probability and a good initial random population, which causes variation in the results. Keim et al. [129] could show that this variation can have major implications for the performance, resulting in many cases where the approach could not find any correct trace links. Moreover, the good performance could not be confirmed in subsequent studies [96, 129]. For example, Hey [96] could only achieve an F_1-score of 27% instead of the claimed 71.9%, much lower than other approaches.

Panichella et al. [205] use LDA in combination with genetic algorithms to find optimal parameters. To do so, they cluster documents by topics using LDA to determine the dominating topic of each document, i.e., the most prominent topic in a document. This way, they can find similar documents based on the topic-distribution from LDA and they can define the silhouette of each cluster. The authors then use a silhouette coefficient that represents how good a clustering is to predict the accuracy of the LDA. With a genetic algorithm, the authors then optimize the LDA configuration. With the best-located configuration, the approach can use LDA for TLR. In their evaluation on the two projects, EasyClinic and eTour, the authors evaluate up to 2,000 configurations. The authors show that there is a high variability that is dependent on the configuration and that their approach can significantly detect a configuration that is among the best-performing configurations.

Wang et al. [279] introduce a method for linking bug reports to source code and also make use of a genetic algorithm. Their approach systematically explores the space of potential VSM variants to identify a heuristically near-optimal composite model. The approach uses 15 slightly adapted variants of the TFIDF weighting to create VSM variants. The genetic algorithm then looks for a near-optimal composition of VSMs to create a composite model. This composite model can then take bug reports and files to return a ranked list. Further, the authors also enhance the approach with three components: version history, structure, and similar reports. However, for the approach to work, it needs training data. As such, it cannot recover trace links without already having some trace links at hand. In their evaluation, the authors use a third of the existing links as training data, and they argue that datasets that are too small will not lead to a (near-) optimal model. In a comparison with a simple VSM, the authors state significant improvements by their approach.

There are different problems regarding IR approaches, including polysemy of words, i.e., when terms have multiple meanings, as stated by Wang et al. [281]. Therefore, Wang et al. introduce an approach for TLR that makes use of WSD to resolve polysemy that uses ANNs. Their method builds on IR techniques and enhances accuracy by using an ANN to determine whether a term has the same meaning across different requirements. By leveraging coreference information, the ANN model refines term meanings, improving the precision of TLR without sacrificing recall. The authors define various models that use different features and techniques. The features include embedding features, syntactic features like POS tags, distance features like the distance between mentions, string matching features, and stakeholder features that denote, for example, if two requirements were created by the same person. For evaluation, the authors compared their approach with traditional LSI across two benchmark datasets (i.e., MODIS and CM-1) and six open-source projects, namely Airflow, Any23, Dashbuilder, Drools, Immutant, and JBTM. They observed that their method significantly outperformed LSI, particularly in improving precision. In the experiments, their approach ruled out an average of 22.63% of false positives caused by polysemy. Additionally, feature ablation studies showed that features like word embeddings and string matching were critical, while stakeholder features did not significantly improve the results. The approach demonstrated high potential for improving the precision of trace link recovery in requirements engineering.

Lam et al. [152] use a ANN in the IR technique called revised Vector Space Model (rVSM) that collects features on the textual similarity to link bug reports and source files. The idea is to tackle the lexical mismatch, which occurs when terms in bug reports differ from those in code. They use ANNs to learn to relate the terms in bug reports and code frequently appearing as pairs in (known) buggy cases. One ANN is used to learn the relationship between text from bug reports and textual parts from code, another ANN is used for the relationship between the bug report text and code tokens. Lastly, the approach uses another ANN to combine various features, including the output of the relevancy estimation using the ANNs, text similarity, and metadata to rank code files for a given bug report. Again, as the approach needs to train the ANNs, there is a need for training data. As such, they require (historical) data and also add negative samples, i.e., pairs of unrelated bug reports and code files. In their evaluation, the authors use the six projects AspectJ, Birt, Eclipse UI, JDT, SWT, and Tomcat. In their comparison with other approaches, the authors report that their approach performs best. One downside of the approach is the required time to train the ANNs for each project, taking up to two hours on their machine with an Intel Xeon E5-2650 CPU that has 32 cores at 2.00 GHz and 126 GB RAM.

Poshyvanyk et al. [218] employ Formal Concept Analysis (FCA) in conjunction with LSI to establish links between textual descriptions of software features or bug reports and the corresponding segments of source code. FCA is a formal method for identifying clusters of objects sharing common attributes. It offers a theoretical framework for analyzing hierarchical relationships among these clusters. The approach uses LSI to map natural language queries that are, e.g., describing software features or bugs, to the relevant code components. The results are ranked and clustered into a concept lattice using FCA, which structures and visualizes relationships between code elements. The concept lattice allows developers to explore the code more efficiently by navigating through relevant nodes and ignoring irrelevant ones. This clustering aims to reduce the size of search results, helping developers identify feature-implementing code elements faster, particularly in software maintenance tasks. As such, the approach is semi-automatic and aims to aid the developer in finding relevant parts instead of providing these directly.

This approach was evaluated on six open-source software systems: ArgoUML, Freenet, iBatis, JMeter, Mylyn, and Rhino, each with several hundred features and bugs. The experiments involved different configurations of concept lattices to assess the method's effectiveness. The results show that combining FCA with IR reduced irrelevant search results compared to standalone IR-based methods. This reduction in irrelevant results translated into less effort for developers to find the correct source code. The authors measured the approach's performance using precision, recall, and effectiveness metrics, confirming that the proposed method outperformed traditional IR-based concept location techniques in most configurations.

Next to approaches that use ML techniques to improve existing IR-based approaches, there are approaches that directly use techniques from ML, such as word embeddings and neural networks.

Guo et al. [89] present of one of those approaches that directly use ML techniques, using word embeddings and ANNs to recover trace links. First, word embeddings are trained on a domain corpus to capture word semantics in a vector space. Then, these word vectors are used as inputs for an ANN to learn the sentence-level semantics of software artifacts. The model compares the semantic vectors of two artifacts and outputs the likelihood that they are linked. This method addresses shortcomings in traditional approaches like VSM and LSI capturing deeper semantic associations between artifacts. In the evaluation phase, the authors trained and tested their model on a dataset from the Positive Train Control (PTC) domain, comparing it against standard techniques like VSM and LSI. By experimenting with different ANN configurations, they found that BI-GRU provided the best performance. The model was trained using a dataset that included manually validated trace links, and multiple configurations were tested to fine-tune parameters. The evaluation showed that the proposed approach significantly outperformed VSM and LSI, achieving higher MAP and better recall at high levels of precision. Moreover, the network's performance improved further when trained with a larger dataset, indicating its scalability.

Zhao et al. [302] propose the approach WELR that leverages word embeddings and a learning-to-rank (LtR) mechanism to improve semantic similarity calculations between software artifacts. WELR includes the use of word embeddings with query expansion and inverse document frequency (IDF) weighting to improve initial similarity scoring, and LtR to refine rankings and predict trace links. The system operates in two phases: preprocessing, where software artifacts are prepared and the word embeddings are learned, and the WELR phase, which applies the embedding-based similarity method and LtR to generate the ranked list of trace links. For evaluation, WELR was tested on the five public datasets CM1-NASA, GANTT, eTOUR, iTrust, and Easy-Clinic and compared against baseline methods like LSI and Word2Vec. WELR showed improvements in both precision and recall over these methods, with a 33.3% improvement in precision and 24.5% in recall over LSI. On average, the approach achieves a precision of 28.3% and a recall of 58.0%. The system performed better across various artifact types, including text-to-text and text-to-code TLR. Further improvements were observed after applying the LtR step, for example, with enhanced MAP scores, outperforming the baselines and the approach AML by Le et al. [155] and demonstrating WELR's effectiveness. However, more recent approaches like by Hey et al. [97] outperform their approach when compared on the same evaluation projects.

Rath et al. [227] use pre-trained classifiers to automatically identify and recommend missing trace links between commits and issues in projects where developers may forget to tag commits with issue IDs. The automated classifier is trained using features like the relationship between commits and issues based on stakeholder information, temporal constraints, and textual similarity between commit messages and issue descriptions. The evaluation of the method was performed on the six open-source projects Derby, Drools, Groovy, Infinispan, Maven, and Pig that are using Git and Jira. The classifier was tested in two scenarios: (1) recommending links to developers at commit time and (2) augmenting incomplete trace links in existing projects. In the first scenario, the classifier achieved an average recall of 96% and a precision of 33% for recommending relevant links. In the second scenario, it automatically augmented missing links with an average precision of over 89% across most projects. However, recall dropped to 50% in the augmentation task due to the high precision requirement. The Random Forest classifier performed best among the tested algorithms.

The issue with ML-based approaches is their dependence on training data, with some approaches requiring lots of data to perform decently. However, training data is often scarce. Approaches like the one by Rath et al. [227] even need existing trace links for a project. This makes the approach a lot less applicable as many projects do not have these trace links (cf. [234]). Thus, I have no reliance on training data for my approach.

Mills et al. also try to tackle this downside with an active learning approach [186, 187]. The approach called TRAIL (TRAceability lInk cLassifier) uses historical TLR data to train a ML classifier that predicts whether trace links between new or existing software artifacts are valid or invalid, thus, labeling it as a binary classification approach. The features used in TRAIL include IR-based features, query quality (QQ) features, and document statistics features. In summary, the approach uses 131 normalized features with 14 IR-based features, 42 pre-retrieval QQ metrics, 70 post-retrieval QQ metrics, and 5 document features. In the evaluation, TRAIL is tested on 11 datasets from six different software systems, which contain a total of 32,616 possible links, with only 7.97% valid ones. TRAIL is compared against seven popular IR techniques. The study finds that TRAIL significantly outperforms these traditional IR techniques across all datasets regarding precision, recall, and F-score, with improvements averaging over 26 percentage points. While these results are promising, the approach still requires initial trace links from the historical data of each project to train the models. The out-of-the-box performance for recovering trace links of unseen projects without initial trace links is unclear.

Advancements in NLP and especially the advent of LLMs with their advanced capabilities in processing natural language have opened new options for TLR.

As self-attention is a central component of LLMs and is one of the reasons for their success (see also Section 2.3.5), Zhang et al. [300] also try self-attention for TLR between software requirements and source code. The benefits of self-attention, as claimed by the authors, are the ability to capture the global connection in one step and that it can solve the long-distance dependence, all in parallel calculations, unlike ANNs. As such, the authors propose a hybrid method to recover semantic trace links, focusing on the use of word embedding and a self-attention mechanism for feature extraction and representation. The approach includes three main stages: preprocessing, feature representation, and the final TLR. In the preprocessing step, NLP techniques and abstract syntax trees (AST) are used to extract meaningful features from requirements and source code artifacts, including the comments. Word embeddings are then used to convert these features into vectors, and a self-attention mechanism is applied to capture contextual dependencies between words, forming text semantic vectors. Finally, cosine similarity between the vectors, adjusted by weighted contributions of code content and comments, is computed to establish trace links. In the evaluation, the authors conducted experiments using two datasets, eTour and iTrust and compared their method with traditional IR techniques such as VSM and LSI. The proposed hybrid method consistently outperformed VSM and LSI, showing higher precision and recall. The optimal configuration for the self-attention model was also determined experimentally, with the best results achieved using a word embedding dimension of 100 and five attention heads. The paper concludes that the use of semantic vectors significantly improves TLR compared to traditional approaches. While this approach is theoretically applicable to SADs and code, it cannot directly be applied to SAMs. Further, an extensive dataset containing enough examples for each kind of artifact is required to learn the self-attention model. As there is limited availability of openly accessible SADs and SAMs [52], this makes the approach hardly applicable for TLR in software architecture.

Instead of only using the self-attention mechanism of LLMs, some approaches directly use LLMs. As large language models can be used for transfer learning across tasks and projects, they are promising candidates for TLR tasks. Lin et al. [159] use the LLM BERT to generate trace links between source code and natural language artifacts. The authors see the advantage of LLMs in that fact that they can be pre-trained on a huge amount of unlabeled data in self-supervised training tasks. In the fine-tuning phase, the model can then be trained on smaller, labeled datasets. The authors propose a three-fold procedure of pre-training, intermediate-training, and fine-tuning. In the pre-training, a BERT model is trained on code, resulting in a so-called CodeBERT, just like in the approach by Feng et al. [65]. In the intermediate training, the CodeBERT is further trained for code search using labeled training examples. The resulting T-BERT model is then fine-tuned for TLR to predict if two artifacts are linked. The authors present three different kinds of architectures, TWIN, SINGLE, and SIAMESE. The TWIN architecture uses two BERT models to encode natural language and code artifacts and then combines the outputs of these models in a fusion layer, providing one vector into a neural classifier. The SINGLE architecture uses one BERT model and the artifacts are combined in one query, separated by separator tokens. The output is then pooled, i.e., averaged into one fused feature vector, and propagated to the classification header. The SIAMESE architecture is a hybrid between the other two architectures. It uses a single BERT model but passes the artifacts one after another into the model and concatenates the pooled output in the fusion layer into a joint feature vector that can then be used in the classification header. The evaluation on the CodeSearchNet dataset [103] and on the three projects Pgcli, Flask, and Keras shows promising results, outperforming classic methods such as VSM, LSI, and LDA. Moreover, the training data impacts the performance, which differs for each of the architectures. Lastly, the authors report that the intermediate-training improves performance.

A similar approach is used by Lüders et al., who treat TLR as a classification problem. In their research, Lüders et al. aim to trace issues with various link types [163] and automatically predict and classify the typed links [164, 165]. The authors investigate various link types that connect issues, such as *depends on*, *blocks*, and *relates to*, aiming to enhance the understanding and prediction of these links. By analyzing a large dataset from multiple open-source projects, they employ an LLM to predict link types with significant accuracy. Furthermore, the paper highlights the importance of accurately predicting link types to streamline the workflow in issue-tracking systems. For their approach, the authors propose a model that leverages textual and contextual features of issues to predict the type of link between them. This approach not only aids in better triaging issues but also helps in prioritizing and resolving issues more effectively. The study's results demonstrate that incorporating link-type predictions can significantly enhance the overall productivity and quality of software maintenance processes. In one of their works [164], Lüders et al. focus on classifying the link types. In the 2023 paper [165], Lüders et al. combine the detection and classification in an approach that predicts the presence of a link together with the link type. For their work, the authors use the LLM BERT, using the concatenation of the title and description of two issues as input. The pooled output is then used to classify the link type with the labels *Relate*, *Subtask*, *Clone*, *Block*, *Depend*, *Epic*, *Duplicate*, *Split*, *Incorporate*, *Bonfire Testing*, *Cause*, and *Non-links*. In the evaluation on 15 projects, the approach scores a mean F_1-score of 73% and a median F_1-score of 71%.

Overall, the presented approaches use various ML methods for TLR. Some approaches mainly used IR methods that they improved with the help of ML. Others created word embeddings and similar feature vectors and used the vectors as input for ANNs. The most recent approaches make use of LLMs and their language-understanding capabilities. All of the approaches try to tackle the underlying problems, such as different wording and abstraction levels and, overall, the semantic gap between artifacts, with varying effectiveness. The promising approaches among all these approaches are supervised, i.e., they require training data. Some approaches require project-specific training data in the form of preexisting trace links meaning that developers need to create enough trace link beforehand for these approaches to work. Other approaches require lots of training data to train or fine-tune models for specific kinds of artifacts or TLR tasks. In Section 3.2.4, I further discuss the limits of these approaches and the distinction between my use cases in software architecture and my approach.

3.2.3. Approaches Using Intermediate Artifacts

When trying to bridge larger semantic gaps like between requirements and code, many approaches weaken the semantic comparisons to factor in things like different abstraction levels and different vocabulary, including synonyms and polysemy. However, this can cause lower precision. Instead of bridging large semantic gaps, some approaches suggest using transitive trace links. This way, instead of bridging one large semantic gap, two (or more) smaller semantic gaps need to be tackled in a divide-and-conquer fashion, which can be two easier tasks.

The Connecting Links Method (CLM) by Nishikawa et al. [194] and by Tsuchiya et al. [274] focuses on establishing transitive trace links between two artifacts using a third artifact. The authors' idea is to use a transitive approach in cases where other methods can be challenging. Additionally, the approach can help understand the relationships between various artifacts like requirements, designs, source code, and test cases. CLM leverages a third artifact to recover trace links between two other artifacts, effectively addressing the limitations of methods like the VSM that struggle with certain link combinations. Consequently, when two trace links share the same element in the intermediate artifact, they are connected. The evaluation of CLM was conducted using the project EasyClinic. The paper demonstrates how CLM can recover links missed by VSM, showcasing its effectiveness in recovering trace links that involve weak or indirect relationships among software artifacts. In experiments comparing the two methods, CLM demonstrated a significant improvement in precision (0.95 compared to VSM's 0.55) for specific trace links. However, CLM was less effective in cases where VSM already performed well, suggesting that CLM's strength lies in handling more complex transitive trace links. These findings indicate that while CLM can be highly beneficial in certain contexts, it is not universally superior to VSM. Overall, the underlying idea is promising, and I transfer parts of this idea to software architecture artifacts. However, the approach has limitations in certain contexts that I plan to avoid.

In a similar approach, Rodriguez et al. [232] also propose the use of existing artifacts as intermediate artifacts for TLR in a transitive way. For instance, design artifacts are used to map requirements to subsystem requirements. The methodology introduces three families of techniques: *Direct*, *Transitive*, and *Hybrid*. The *Direct* families use methods like VSM and LSI to recover trace links without intermediate artifacts. The *Transitive* family applies various methods to recover trace links by establishing a path of trace links between the artifacts only using the intermediate artifacts. The *Hybrid* family combines both direct and transitive approaches to potentially further improve accuracy. In their evaluation on the five datasets Dronology, TrainControl, EasyClinic, EBT, and WARC, Rodriguez et al. compare the different families and several techniques for scaling (independent, global) and aggregating (Max, Sum, PCA) similarity scores. The evaluation demonstrates that, in most cases, the best transitive techniques significantly outperform the direct techniques ($\alpha \leq 0.001$). Although hybrid approaches often marginally outperform transitive approaches, this difference was not statistically significant. This suggests that while transitive methods offer significant performance gains, combining them with direct techniques provides limited additional benefit.

To transitively link source code and use cases, Berta et al. [18] use existing issue reports and commit history as transitive artifacts for TLR between use cases and source code to be able to identify and remodularize relevant use cases. For their approach, the authors propose various methods to identify similarities: analyses of synonyms, hypernyms, and hyponyms, sentence similarity between use cases and code, and semantic similarity between issues and commits. The system generates sentences from the code, such as method names and their associated variables, and compares them with use-case steps using the aforementioned semantic similarity methods. The evaluation using the OpenCart e-commerce system focuses on 16 use cases across 141 methods. The results show a precision of 75%, with recall being relatively low at 3.37%. The precision was highest when combining all three methods, though the recall remained a challenge. Methods based on sentence similarity performed less effectively, while the use of issues and commits showed promise, especially when combined with other methods. However, the performance heavily depends on the quality of issue descriptions and commit messages. Additionally, the approach was computationally expensive, taking up to an hour per use case. The authors note that while their method aids in remodularization, expert involvement is still required due to the limitations of precision and recall.

Many approaches often use only single measures, a shortcoming that is approached by Moran et al. [191] with a hierarchical Bayesian network. Their approach, COMET, uses the hierarchical Bayesian network to combine a set of textual similarity measures and estimate the likelihood that there is a trace link between a source artifact and a target artifact. The process involves four stages. First, the approach recovers trace links using IR and ML techniques such as VSM, LSI, JSD, LDA, and Non-Negative Matrix Factorization, as well as combinations of these. Second, the recovered links are reviewed by developers, and the model can continuously update and refine trace link predictions by learning from this feedback. In the third stage, the approach makes use of transitive relationships between artifacts through transitive trace links. Transitive links between two artifacts are established when two links from one artifact refer to the same element in another artifact. For example, if a source code is linked to a requirement and that code also connects to test code, COMET can infer a potential trace link between the requirement and the test code, even in the absence of direct textual similarities. The last stage constitutes a holistic model that combines the three underlying stages to infer the posterior for their Bayesian network, allowing for a more robust and context-aware approach to TLR. The authors evaluated the approach on the projects Albergate, EBT, LibEST, eTour, SMOS, and iTour and also performed an industrial case study. The results demonstrate that COMET outperforms several optimally configured baseline techniques, showing an improvement of approximately 5% in average precision across all subjects and up to 14% in the best cases. These results suggest that COMET is not only effective in controlled environments but also holds promise for practical applications, as demonstrated in the industrial case study with developers from Cisco Systems, who tested COMET's Jenkins plugin integration.

To extend their earlier work, TAROT [77], that I presented in Section 3.2.1, Gao et al. [75] make use of biterms, i.e., co-occurring term pairs, and combine them with intermediate artifacts to tackle TLR. The authors follow the idea that biterms indicate a consensus between two artifacts. As the semantic gap might be too big between certain artifacts because artifacts on different abstraction levels usually have different textual descriptions, intermediate artifacts can aid in bridging this gap. Overall, the approach follows the same idea from their earlier work [77], but makes use of intermediate artifacts. The biggest difference is that the approach now uses intermediate-centric consensual biterms. This means that biterms from the source artifact or from the target artifact also need to appear in the intermediate artifact. At the same time, biterms that are only present in the intermediate artifact are discarded. With the biterms, the candidate list is calculated in the same way as for TAROT [77]. The IR scores are then adjusted for the transitive links, differentiating inner- and outer-transitive links. Inner-transitive links are links within the same kind of artifact with the same level of abstraction, outer-transitive links are between different kinds of artifacts. Using paths of transitive links, the similarity scores from the candidate list are adjusted. In their evaluation with the datasets Dronology, WARC, EasyClinic, EBT, and LibEST, Gao et al. compare their transitive approach TRIAD with four baselines, including TAROT [77] and COMET [191]. The result shows partly significantly higher F_1-scores for TRIAD, but in some cases and configurations, TRIAD even performed worse in MAP. Overall, the authors conclude that the configuration that combines outer- and inner-transitive links works best in bridging the abstraction gap and recovering trace links. While the general idea of biterms, coming from TAROT, is promising, the authors only consider artifacts that are either natural language texts (e.g., requirements, specifications) or code (including test code, supporting only Java and C). Consequently, the application on architecture models is not defined, requiring further work to adapt the approach to SAMs. Due to the recency of the approach and the difficulties of extending the approach for SAMs, I do not use it or compare my approach to it. Additionally, while the approach uses transitive links, it does not apply specialized approaches for the respective TLR with the intermediate artifacts. Thus, I expect limited results. However, future work needs to explore this further and needs to compare TransArC with TRIAD. In this work, I still compare my approach to the TAROT approach that TRIAD is based upon.

In conclusion, using intermediate artifacts for TLR has proven to be an effective strategy for bridging semantic gaps between software artifacts. Techniques like the Connecting Links Method (CLM) [194, 274] and the approach by Rodriguez et al. [232] demonstrate how leveraging transitive relationships between artifacts can yield higher precision and better recovery of trace links. These methods enable more manageable connections by breaking down large semantic gaps into smaller, more tractable ones. Additionally, while transitive techniques generally outperform direct methods, hybrid approaches offer limited further improvement. Evaluations of various methods reveal that the effectiveness of these approaches depends on the quality and nature of the artifacts and the specific configuration of transitive link strategies. Overall, the use of intermediate artifacts holds significant promise, particularly in handling complex or indirect trace link relationships, though challenges such as computational expense and recall limitations remain.

3.2.4. Conclusion and Implications

There are numerous approaches, each offering distinct advantages and disadvantages. For example, IR-based approaches often excel in precision but tend to struggle with recall due to their limited ability to capture deeper semantic meaning. Some approaches try to tackle the problem more semantically, often by using ML techniques. However, these ML-based approaches regularly require training data that might be unavailable. Moreover, the approaches often soften the comparison criteria to recover more links to improve recall. However, this increased recall can come at the expense of precision, as broader matches are more likely to introduce noise.

Addressing my contributions regarding TLR between SAD and SAM as well as between SAM and code, there are key differences to existing work. Foremost, my approaches mainly aim to specialize in these architectural artifacts and their unique characteristics to maximize the performance for TLR.

Further, the existing related work has not dealt with SAMs yet. SAMs have different characteristics to other artifacts, thus they need different treatment. For example, they usually have less detailed information compared to other artifacts as they usually do not contain comments or similar, unlike code. At the same time, they are not based on natural language, so commonly used NLP techniques for, e.g., feature extraction also do not work. Therefore, approaches that tackle SAMs need to make the best use of the available information.

Similarly, while SADs are natural language-based, they usually contain other information or the information is differently presented. This can be an upside, e.g., because SADs often directly name architecturally relevant entities like components. This is why my approach ArDoCo aims to make use of such information, which is not directly done by related work. However, this can also be a downside. For example, naming is a common inconsistency [290], negating most of the advantages of having named entities. Therefore, an approach can use the names, but this problem needs to be considered, which has not been done. Respecting these characteristics in specialized approaches can potentially result in much better performance. However, many approaches do not consider such artifact-specific characteristics and try to tackle the problem with generic methods like VSM or machine learning. In other cases, approaches cannot use these characteristics properly because the characteristics of different artifacts mismatch. This regularly happens if the semantic gap between artifacts is large.

This is in line with a key observation from the related work: Many proposed TLR approaches perform well when the semantic gap between artifacts is small, but these methods often face challenges as the semantic gap widens This limitation is critical when dealing with heterogeneous artifacts such as source code, requirements documents, or architectural models, which often differ in terminology and abstraction level. As part of my thesis, I will explore how the use of intermediate artifacts can serve as a bridge to overcome larger semantic gaps and improve TLR for software architecture, linking SAD and code using SAMs as intermediate artifacts.

Relatively few existing approaches use intermediate artifacts and transitive links to address the semantic gap between artifacts. Those that do report promising results, suggesting that transitive methods can enhance TLR performance by incrementally bridging the semantic gap. Such methods typically rely on the assumption that intermediate artifacts share enough semantic similarity with both the source and target artifacts to support effective traceability. For example, an intermediate artifact like design documentation can serve as a conceptual link between high-level requirements and low-level source code.

Building on these transitive TLR approaches, I intend to adopt and adapt ideas for my specific scenario. By tailoring the transitive methods to the distinct characteristics of the intermediate artifacts in my domain, I aim to capitalize on their strengths. A notable limitation of many transitive TLR approaches is their reliance on the same or highly similar techniques for recovering trace links between both the source/intermediate and intermediate/target artifacts. This lack of differentiation between the stages might prevent the full potential of transitive TLR from being realized. In my opinion, the greatest advantage of transitive TLR is its ability to focus on pairs of artifacts that are more closely related, enabling the use of specialized techniques that are well-suited for each part.

This leads to a key distinction between my transitive approach, TransArC, and existing methods. As described in Section 5.4, TransArC aims to introduce more specialization by targeting the unique challenges presented by each pair of artifacts. By focusing on each distinct task, I can maximize the performance of TLR at each stage, rather than relying on a one-size-fits-all approach.

Moreover, none of the current approaches specifically tackle TLR within the context of software architecture, particularly for SADs or SAMs. Specifically, SAMs present unique challenges due to their higher level of abstraction and formal structure, which are not easily handled by traditional TLR methods. SADs and SAMs often encapsulate system-wide design decisions, patterns, and constraints that are essential for understanding how requirements are fulfilled by the system's architecture. Because of this, many existing TLR approaches are unsuitable, as they are not equipped to handle such formal models or highly abstract artifacts.

Thus, my approach seeks to bridge the semantic gaps through transitive methods and introduce tailored techniques capable of handling the specific challenges posed by architectural artifacts. By addressing the abstraction differences and formal nature of SADs and SAMs, my research aims to extend the applicability of TLR to areas that have been largely overlooked in previous studies.

3.3. Inconsistency Detection

Inconsistency Detection is a critical area of research in different disciplines and domains, especially in domains involving large-scale software engineering and knowledge representation. It addresses the challenge of identifying conflicts, contradictions, or irregularities within datasets, system specifications, or logical frameworks. Prior work in this area has developed various techniques, including rule-based approaches, machine learning methods, and formal verification tools, to automate the identification of inconsistencies. These approaches have been applied across diverse fields, such as linguistics, biomedical informatics, mechanical engineering, and software engineering, where inconsistency detection ensures the correctness and reliability of system behavior. In the following, I give an overview of different approaches for ID, with a main focus on ID for software engineering. In Section 3.3.1, I first look into loosely related work for problems in the software engineering domain. Then, in Section 3.3.2, I present related work that concerns ID between API or documentation and implementation. This is closely related to the work in Section 3.3.3 about detecting inconsistencies in requirements. In Section 3.3.4, I tackle related work specifically for software architecture, including architecture conformance checking. There are also other domains and research fields that I briefly look into in Section 3.3.5. Finally, in Section 3.3.6, I give a short conclusion about the related work for inconsistency detection and the implications for my work.

3.3.1. Complementary Work in Software Engineering

Some approaches are distantly related to inconsistency detection for software engineering. For example, Arora et al. [10] check if requirements written in natural language conform to predefined templates. The authors highlight that while natural language is widely used for specifying requirements due to its accessibility, it is also prone to ambiguity, which can complicate automated analysis. To mitigate these issues, requirements templates are employed to structure NL requirements, but manually checking for conformance to these templates can be tedious and error-prone, especially in dynamic environments where requirements frequently change. The motivation behind this work is to develop an automated approach that leverages NLP techniques to facilitate template conformance checking without relying on a complete glossary of terms. The authors propose a method for processing the requirements and pattern matchers, each representing a requirements template, using text chunking. The pattern matchers then enable the conformance check. In their evaluation, the authors conducted four case studies across different domains, assessing the accuracy and scalability of their solution. The results indicate that the automated conformance checking is robust and accurate, with the approach demonstrating a high level of precision and recall. At the same time, their approach scales linearly, suggesting that it can be effectively applied in large-scale projects. The biggest downside of this work is that this inconsistency checking only applies to cases where natural language texts are written for and with predefined templates, not for true natural language documentation.

In their work, Molenaar et al. [190] propose an approach called RE4SA that aligns requirements and architecture. The authors highlight that effective communication between requirements engineers and software architects is often problematic, leading to project failures. To tackle this issue, the idea is a model to link requirements, in the form of epic stories and user stories, and their architectural counterparts, in the form of modules and features. These trace links can have various types, including *refinement*, *abstraction*, *allocation*, and *satisfaction*. In their model, the authors include metrics to measure the alignment between the two sides. The model establishes explicit relationships between functional requirements and architectural components, aiming to reduce rework in later development phases and improve overall project outcomes. Essentially, the idea is to use trace links to avoid inconsistencies in later stages. To evaluate the effectiveness of the RE4SA model, the authors conducted two case studies. The results showed varying degrees of alignment between requirements and architecture, with metrics indicating areas for improvement. For example, approximately 13% of the requirements were found to be under-allocated, suggesting that the architectural components did not address some needs. In contrast, the architecture recovery case revealed that while most user stories were allocated to features, a significant number of features did not satisfy any requirements. The authors conclude that the RE4SA model and its associated metrics can significantly enhance communication and traceability between requirements and architecture, ultimately leading to better alignment and project success in agile development environments. In this work, the authors assume that the relationships and trace links are already modeled. The model then can be used to express and inspect the relationships further, e.g., to calculate how well the artifacts align.

Similarly to the previous work but with another purpose, Rempel et al. [230] propose traceability metrics in the context of agile development to use graph-based metrics to estimate the implementation risk, i.e., the error-proneness. This can be interpreted as assessing the risk of introducing inconsistencies. The authors highlight that current practices for assessing implementation risk rely heavily on subjective judgment, which can lead to suboptimal decision-making. They propose that the complexity of relationships among requirements, represented through a traceability graph, can serve as a quantitative indicator of implementation risk. This approach aims to enhance requirement prioritization and testing efforts, ultimately contributing to more successful project outcomes. In their evaluation, the authors showed that the metrics could reliably predict defect rates in unseen projects, achieving up to 97.82% accuracy in risk category assignments. Making the risk explicit is useful for reducing inconsistencies; contrary to my approach, this approach does not directly detect inconsistencies.

These previous approaches are all concerned with inconsistencies, and they try to avoid inconsistencies using different strategies. In contrast to my work, they do not directly detect inconsistencies. Similarly, Olsson and Grundy [200] introduce an approach for inconsistency management. The authors highlight that existing tools often provide limited support for traceability and consistency management, leading to manual and error-prone processes. The authors introduce a novel approach focusing on managing the fuzzy relationships between high-level software artifacts. They propose an abstract model for each type of software information, allowing for both explicit and implicit linking of elements across different representations. This linking facilitates traceability, visualization of cross-representational information, and change management for manually tracking inconsistencies. The manual inconsistency tracking differentiates this approach from my automated inconsistency detection approach. In terms of evaluation, the authors have developed a proof-of-concept prototype that allows for the association of requirements, use cases, and black-box test plans.

3.3.2. API, Documentation, and Implementation

A major area of current ID research that concerns automated approaches is between API or code documentation and implementation. The detection methods include static analyses [54, 304], dynamic analyses [261], machine learning [26, 206], and trace links [256].

While the following approach detects inconsistencies, they are concerned with a different type of inconsistency. My approach looks at software architecture, i.e., SADs and SAMs, and tries to identify inconsistencies regarding architecture elements like components. In contrast, the following work looks at configuration options, API documentation, code, and similar.

In their work, Dong et al. [54] use static analyses to address the challenges posed by configuration options in modern software systems. These options are often defined across various components, including source code, configuration files, and documentation, leading to potential inconsistencies that can result in misconfigurations. The authors highlight the difficulty of maintaining consistency as software evolves, particularly in large projects like Apache Hadoop, which has over 800 configuration options. The authors propose ORPLocator, a static analysis tool that automatically identifies where configuration options are read in the source code to tackle these inconsistencies. The approach involves locating option read points (ORPs) by analyzing the source code to find method calls that access configuration values and then inferring the corresponding option names. This method allows for a comparison between the identified ORPs and the documented options, facilitating the detection of inconsistencies. The evaluation of ORPLocator was conducted on multiple components of Apache Hadoop. The results demonstrated that ORPLocator successfully located read points for 93% to 96% of documented options across different modules, outperforming existing techniques. Additionally, the evaluation revealed four previously unknown inconsistencies between the documentation and the source code, underscoring the tool's practical utility in improving software reliability and consistency. The biggest difference of the approach compared to my approach for ID is its focus on a different kind of inconsistency by looking at configuration options.

Similarly to the approach by Dong et al., Zhou et al. [304] apply static analyses to tackle the issue of incomplete or inconsistent API documentation. While this also involves a different kind of inconsistency, it is closer to my approach than the previous work. Zhou et al. state that the motivation behind their work stems from the observation that even well-regarded APIs often contain documentation errors, which can lead to developer frustration and abandonment of APIs. Thus, the authors introduce DRONE (Detect and Repair of dOcumentatioN dEfects), a framework designed to automatically identify and repair defects in API documentation. DRONE particularly focuses on directives related to parameter constraints and exception handling. For this, their approach looks into four cases of parameter usage constraints, namely *nullness not allowed*, *nullness allowed*, *range limitation*, and *type restriction*. These are the kinds of inconsistency that the approach looks into, making its goal different from mine. Their solution combines program analysis, NLP, and constraint solving. The framework operates in four main steps: it extracts annotated documentation from the source code, parses the code to create an abstract syntax tree (AST) for analyzing control flow and exception handling, employs NLP techniques to identify relevant textual features in the documentation, and finally uses first-order logic to detect inconsistencies between the documentation and the code. The approach then uses documentation templates for the different kinds of constraints to suggest repairs to the user. The authors emphasize that their focus is on semantic defects rather than syntactic errors, aiming to enhance the quality and usability of API documentation. In the evaluation, the authors conducted experiments on the JDK 1.8 and Android 7.0 APIs. The results indicate that DRONE can effectively detect API documentation defects, achieving an average F_1-score of 79.9% across various libraries. Additionally, the framework's repair recommendations were assessed through user judgments, which showed that the suggested fixes were accurate and concise.

Tan et al. [261] also address inconsistencies between code and its API documentation. In their work, the authors highlight the challenges of automating the detection of these inconsistencies due to the complexities of NLP and the inherent ambiguities in comment text. The proposed approach, named @TCOMMENT, consists of two main components. The first component analyzes Javadoc comments to infer properties related to method parameters, particularly concerning null values and exceptions. These properties as goals for inconsistency detection are the main distinction to my proposed approach for ID. The second component generates random tests for the methods and checks the inferred properties against the actual method behavior. This dynamic analysis approach, built on the Randoop tool for random test generation [203, 202], allows for detecting inconsistencies that static analysis methods may miss. The authors argue that focusing on null-related properties is particularly relevant, as null pointer exceptions are a common source of bugs in Java applications. In the evaluation, @TCOMMENT was applied to seven open-source Java projects, resulting in the identification of 29 inconsistencies between Javadoc comments and method bodies. The evaluation demonstrated that @TCOMMENT could infer properties with high accuracy (97-100%) without relying on complex NLP techniques, largely due to the structured nature of Javadoc comments.

Kim and Kim [130] also focus on detecting inconsistent identifiers in API documentation and code. The authors present an approach that identifies three types of inconsistencies: semantic (different words for the same concept), syntactic (similar letter sequences), and POS inconsistencies. Again, these types of inconsistencies are the main distinction to my goals for ID. Their proposed approach involves creating a custom Code Dictionary that captures domain words and idioms from popular Java projects using NLP. This dictionary enhances the accuracy of inconsistency detection by filtering out acceptable identifiers, thus reducing false positives. Like many similar approaches, this work uses text similarity. For evaluation, the authors applied their method to seven Java projects. The results showed a precision of 85.4% and recall of 83.59%. Additionally, interviews with developers confirmed the prevalence of inconsistent identifiers in software projects and the tool's utility in enhancing detection accuracy. Overall, the study demonstrates that the proposed approach can significantly aid developers in maintaining code quality and improving software maintainability.

To detect inconsistencies between natural language comments and the corresponding source code, Panthaplackel et al. [206] apply ML algorithms. The paper proposes a deep-learning approach for just-in-time inconsistency detection. It aims to identify inconsistencies before they are committed to a codebase, thereby improving software reliability and developer productivity. To achieve this, the authors develop a deep-learning framework that correlates comments with code changes using ANNs and gated graph neural networks (GGNNs), capturing the syntactic structures of both comments and code changes. Each model is trained on a large dataset of comment-code pairs extracted from open-source Java projects. The authors emphasize the importance of detecting inconsistencies at the moment of code changes, as this allows for leveraging the previous version of the code, which is consistent with the comment, to assess the impact of the changes. In their evaluation, the authors demonstrate that their model significantly outperforms several baseline methods, including traditional approaches that do not consider code changes. They also conduct an extrinsic evaluation by integrating their inconsistency detection model with a comment update system, showcasing its utility in building a comprehensive automatic comment maintenance system. The results indicate that their approach not only effectively detects inconsistencies but also facilitates the automatic updating of comments, thereby enhancing the overall quality of software documentation. In contrast to the previous approaches, this approach is less specific and might be applied to software architecture. Their inconsistency detection requires a large training set, for which they can use the many available open-source projects. However, the required data is not available for software architecture.

Machine learning is also applied by Borovits et al. [26] in their work that focuses on detecting inconsistencies between the names and bodies of code units in Infrastructure-as-Code (IaC) scripts. The authors highlight that as software development cycles shorten and automation becomes prevalent, the quality of IaC scripts is crucial for maintainability and comprehensibility. To tackle this problem, the authors propose a novel approach that employs word embeddings and classification algorithms to detect name-body inconsistencies. The approach formulates the detection task as a binary classification problem, utilizing the abstract syntax tree of IaC code units to generate embeddings for both task names and bodies. The authors performed two experiments to assess the performance of their approach. They compared six machine learning algorithms, including Random Forest, SVM, and ANN models like Convolutional Neural Networks and LSTMs. The results showed that classical machine learning models and deep learning models achieved comparable performance in detecting inconsistencies, with metrics such as Matthews correlation coefficient (MCC), i.e., the Φ correlation, Area Under the Receiver Operating Characteristic curve (AUC-ROC), and accuracy indicating satisfactory results. Again, this approach requires training data to function properly, making it hardly applicable to software architecture artifacts. Moreover, the authors again look at a specific type of inconsistency, differentiating this approach from mine.

Stulova et al. [256] first create fine-grained trace links and use these to detect inconsistencies between source code and its accompanying documentation, particularly comments, during software evolution. The proposed solution, a tool named upDoc, aims to automatically detect these inconsistencies by mapping code changes to their corresponding documentation, thereby enhancing program comprehension and reducing potential errors. The authors' approach involves a multi-module architecture for upDoc, which includes parsing source code, mapping code elements to comments, extracting changes, analyzing these changes, and suggesting fixes. The tool employs NLP techniques to create fine-grained mappings between code and comments, i.e., mappings between AST nodes of the code and sentences of the comments. The authors emphasize the importance of semantic analysis in their approach. Semantic analysis enables the detection of more complex relationships between code and comments, thus improving the reliability of the inconsistency detection process. In their preliminary evaluation, the authors tested upDoc on a dataset of Java open-source projects, focusing on commits that were supposed to fix documentation inconsistencies. The results indicated that upDoc successfully identified improvements in code-comment mappings, with a significant percentage of cases showing increased similarity scores after documentation updates. However, the evaluation also revealed some limitations, such as instances where the tool failed to detect changes due to its current focus on method signatures alone. It is important to note that this work is only preliminary. The inconsistency detection is only outlined, and no implementation or evaluation of it exists yet. Moreover, in contrast to my work, this approach aims to identify inconsistent comments based on changes between commits. Thus, in contrast to my approach, the authors rely on historical data.

3.3.3. Requirements

Another major area of research about ID in software engineering is centered around requirements. Various approaches have been proposed to address inconsistencies in this context. Some approaches focus on maintaining consistency between different types of diagrams used in the development process, such as UML or other modeling languages, as highlighted by Egyed [55]. Another approach addresses the alignment between requirements and design specifications, ensuring that the transition from requirements to implementation is consistent, as explored by Kozlenkov [142]. Additionally, Gervasi [79] examines techniques for detecting inconsistencies within textual requirements themselves, a crucial activity to prevent misunderstandings and ambiguities early on.

Fantechi et al. [61] propose an approach for ID between several textual requirements. The authors state that existing NLP tools have attempted to address these issues, but they propose a novel approach that leverages the extraction of Subject-Action-Object (SAO) triples from requirement documents to identify potential inconsistencies and gaps in information. The approach involves a syntactic parsing of requirement documents to extract SAO triples that encapsulate the core interactions described in the text. By analyzing the distribution of these triples, the tool can assess the relevance of different document sections concerning specific functionalities. The SAO-based Content Analysis technique assigns scores to the extracted triples based on their alignment with predefined dictionaries related to the functionalities under investigation. This scoring system allows for identifying redundant or missing information, thereby highlighting areas that may lead to inconsistencies. The tool also features automatic dictionary generation to streamline the process of compiling relevant keywords for analysis. For their evaluation, the authors conducted experiments on a software requirements document, achieving a 63% accuracy in correctly extracted SAOs. The authors note that inaccuracies stemmed from parsing errors and output processing from the Link Grammar parser.

Kamalrudin et al. [118] also addresses the critical need for consistency in software requirements from the earliest stages of the requirements engineering process. The authors' approach involves a prototype tool that utilizes traceability techniques to manage consistency between textual requirements and essential use cases (EUCs), i.e., structured representations of functional requirements. The tool is designed to extract essential interactions from natural language requirements using an interaction pattern library and a tracing engine. This process allows for the automatic generation of EUC models. In this process, trace links are generated and the authors aim to create an environment where requirements engineers can easily navigate between different representations of requirements—textual, abstract interactions, and EUCs using these trace links while maintaining traceability and consistency. As such, the approach does not tackle consistency in an automated fashion but provides a tool for users to detect them more easily. In the evaluation, the authors conducted a preliminary study with eight software engineering post-graduate students to assess the tool's usefulness and ease of use. The results indicated that nearly all participants found the tool useful, with a significant percentage rating it as very useful. The ease of use was also positively received, with most participants agreeing that the tool was user-friendly. However, some participants expressed a desire for more complex inconsistency-checking features. Overall, the evaluation suggests that the tool effectively supports requirements engineers in detecting and managing consistency between different forms of requirements.

Ali et al. [6] address requirements that are expressed in formal models. The approach proposed in their paper involves developing automated analysis mechanisms to identify and analyze modeling errors in contextual requirements models, specifically through the use of contextual goal models. The authors introduce two primary analysis mechanisms: one for detecting inconsistencies in context specifications and another for identifying conflicts arising from simultaneous actions taken to meet different requirements. The approaches detect inconsistencies by using a formula defining the context and then applying SAT-based techniques to check consistency. In the evaluation section, the authors present a case study involving a smart-home system. The evaluation aims to assess the effectiveness of their automated analysis techniques in detecting inconsistencies and conflicts that might be overlooked by requirements engineers. The results indicate that the proposed mechanisms successfully identified numerous inconsistencies and conflicts. Additionally, the engineers found the modeling constructs required for the analysis to be generally understandable and manageable, although some aspects, particularly the specification of relations between contexts, were noted as challenging. Overall, the paper concludes that the proposed framework is a significant step toward ensuring the correctness of adaptive systems by addressing the complexities introduced by contextual variability. While this approach also addresses inconsistencies in (semi-) formal software artifacts, my work does not target requirements or use formal methods for logical validation. Instead, my approach focuses on the consistency between SAD and SAM, which are more structural and implementation-oriented. The work by Ali et al. focuses on contextual and goal-driven requirements; my work operates on the architectural level, dealing with mostly structural representations rather than logical consistency in formalized requirements. This makes the approaches complementary but distinct in scope and methodology.

Overall, the works referenced in this section primarily focus on identifying inconsistencies within requirements engineering or between requirements and other representations, such as diagrams or formal models. For example, Gervasi and Kamalrudin's approach concentrates on textual requirements, utilizing techniques like SAO triples extraction or traceability to align requirements with use cases. The presented approaches target the early stages of the development lifecycle, emphasizing requirements correctness, alignment, and traceability. While requirements focus on what the system should do, architecture documentation and models describe how it is structured and achieves those requirements. My approach identifies discrepancies in architectural elements, such as components, that are present in one artifact but missing or misrepresented in another, a concern not addressed in the reviewed works. Thus, my work bridges a critical gap in ensuring consistency beyond the requirements phase.

3.3.4. Architecture Level

When looking at related work that addresses ID on the architecture level, there are only a few related works.

Ali et al. [5] investigates the state of architecture consistency practices in software development in 2017, focusing on the challenges faced by practitioners and the requirements for effective tools. The authors conducted interviews with 19 experienced software engineers. The findings reveal that many practitioners rely on informal methods to maintain architectural consistency, often because justifying the need for formal approaches to management is challenging. The study identifies several barriers, including the invisibility of architectural inconsistencies to customers and the perceived high effort required to map systems to their intended architectures. Despite these challenges, practitioners believed in the utility of architecture consistency approaches for various software development activities, such as auditing and ensuring quality attributes. In evaluating existing tools, the authors found that current offerings often fail to meet practitioners' needs, particularly in supporting systems with diverse technologies and addressing non-maintainability architectural concerns. The study concludes with recommendations for researchers and tool vendors to enhance architecture consistency tools by incorporating features that address the complexities of modern software systems, support real-time inconsistency awareness, and facilitate the resolution of identified inconsistencies. Ali et al. suggest that further empirical studies are needed to better understand the impact of such tools on software development practices and to promote their industry adoption.

Regarding specific tools and approaches, there are approaches for ID between software specifications and implementations via architectural models [157], between component-and-connector models and behavioral models [38], and between automatically generated architectural decisions and *component-and-connector models* [166]. All these artifacts and the general goals of the inconsistencies analyses are different from my approach, but the techniques employed might be interesting when adapted to my use case.

Li and Horgan [157] address the challenge of keeping software specifications synchronized with their implementations. The authors propose a methodology called Workflow-Implementation-Synchronization (WIS), which consists of three main steps: representing the specification in a formal model, detecting inconsistencies between the specification and implementation, and updating the specification based on these inconsistencies. First, the approach requires the user to create a specification of the architectural design in a Specification and Description Language (SDL) called ArchiFlow. This manual step is a major downside and distinction to my automated approach for ID. The manually created specification and the implementation can then be compared to detect inconsistencies. The WIS method aims to reduce human involvement in maintaining synchronization, thus streamlining the maintenance process and ensuring that specifications remain reliable and up-to-date. In their evaluation, Li and Horgan executed two types of experiments: one in which inconsistencies were detected without intentional changes to the implementation and another in which changes were deliberately made to test the synchronization capabilities. The results demonstrated that the WIS method successfully identified inconsistencies and updated the specification model to reflect new features, proving its effectiveness. The authors conclude that their research offers promising avenues for reducing manual effort in software maintenance and enhancing the overall reliability of software systems.

Çiraci et al. [38] tackle inconsistencies that can arise between software architecture documentation and actual code implementation. Their approach employs a runtime verification strategy that leverages Prolog to automatically generate runtime monitors based on the architectural behavior models. The process involves three main phases. First, during the architecture phase, the architect manually defines component and connector diagrams and execution scenarios. Secondly, during implementation, the source code is analyzed to extract call graphs and to semi-automatically establish mappings between architectural elements and code. Lastly, at runtime, instrumentation aspects are generated to monitor the software system's execution and ensure that it adheres to the specified behavioral models. The evaluation of ConArch is demonstrated through a case study involving a Crisis Management System. The results indicate that the approach effectively detects inconsistencies between the architectural documentation and the implementation. The evaluation also revealed a runtime performance overhead of approximately 5 seconds, primarily due to the evaluation of conditions. This suggests that while the approach is effective, there is potential for optimization. In contrast to my automated approach for ID, this approach requires manual work in multiple steps. Further, the approach tackles another type of inconsistency, i.e., runtime behavior, which is different from the structural consistency that my approach tackles.

In their research, Lytra et al. [166] look at the challenge of maintaining consistency between ADDs and corresponding architectural designs, particularly component-and-connector (C&C) models. They propose a semi-automated approach that utilizes constraints for consistency checking and offers model fixes to help software architects manage these inconsistencies. For this, they use tools and templates to create semi-formal ADDs that then are combined with C&C models using manually defined mappings to create OCL-like constraints. The proposed approach integrates two existing tools: *ADvISE*[1] for modeling reusable ADDs and *VbMF* [273] for editing C&C views. When inconsistencies are detected, the system suggests possible fixes, which can either be automatically applied to the C&C models or require the architect's reconsideration of the ADDs. In their evaluation, the authors conducted a case study in the context of service-based platform integration, demonstrating the efficiency and scalability of their approach. They measured the time required for constraint validation and found that it increases linearly with the number of C&C view elements, remaining efficient even for larger models. The results indicate that their approach significantly reduces the manual effort needed to resolve inconsistencies, as it supports the generation of constraints and suggested fixes. While this approach tries to achieve similar goals to my work, it requires the creation of semi-formal ADDs using a specific tool in contrast to simple natural language documentation. It further requires predefined mappings to create OCL-like constraints. However, some inconsistencies cannot be found using such mappings, including UMEs and MMEs. They are characterized by the absence of mappings. Therefore, they address different kinds of inconsistencies, while my work aims at these existence decisions.

[1] `https://swa.univie.ac.at/Software_Architecture/research-projects/architectural-design-decision-support-framework-advise/`, last accessed 27.11.2024

The research field about architecture conformance is closely related to ID on the architecture level. This research field focuses on ensuring that software systems are implemented in alignment with their intended architectural design. Consequently, architecture conformance looks for inconsistencies between implementation and design, taking the design as the ground truth. Various methods and tools are designed to automatically detect violations of architectural rules. These rules can include component boundaries, allowed dependencies, or interaction patterns. There are two types of analysis tools, static and dynamic ones [267]. Static analysis tools that inspect source code to check conformance. Similarly, dynamic analysis tools monitor runtime behavior to ensure it matches architectural expectations. The defined architectural rules often need to be specified in some formal language like description logics [242] or controlled natural language (CNL) using templates [241].

Pruijt et al. [221] introduce HUSACCT, a static architecture conformance tool designed to enhance the verification of software architecture conformance to high-level design models. The motivation behind this research stems from the limited adoption of existing tools. The authors emphasize the need for tools to support semantically rich modular architectures (SRMAs), consisting of various module types and rules that govern their interactions. HUSACCT aims to provide extensive support for five common module types and eleven rule types, addressing the inadequacies found in current tools. The approach taken in HUSACCT involves defining intended architectures, mapping them to implemented software units, and validating their conformance through a series of checks. The tool allows users to manually create and maintain modular architectures with rich sets of module and rule types, ensuring that the rules applied are semantically appropriate for the module types. Regarding evaluation, HUSACCT has been tested through practical applications on both open-source and professional systems. Without presenting concrete examples or an empirical evaluation, the authors claim significant improvements in accuracy, performance, and usability, with processing times for large codebases drastically reduced. The authors state that the tool's extensive SRMA support has proven relevant in real-world scenarios.

Like Pruijt et al. with HUSACCT, Schröder and Riebisch [241] want to ensure that software architecture decisions are correctly implemented in the source code. The authors propose an ontology-based approach that utilizes a CNL for formalizing architecture rules. The CNL serves as a bridge between natural and formal languages, facilitating the creation of architectural rules that are both human-readable and machine-verifiable. The authors emphasize that this method enhances the clarity of architecture documentation and allows for automated verification of compliance with the defined rules, addressing the issues of ambiguity and inconsistency prevalent in traditional documentation practices. For evaluation, the authors apply their approach to the real-world software system TEAMMATES, demonstrating its effectiveness in detecting architecture violations. They compare the results of their conformance checking with those obtained from the HUSACCT tool [221], which serves as a ground truth. The evaluation shows that their approach can identify crucial architecture violations that other tools may miss, highlighting its flexibility in defining architecture rules and its capability to produce accurate violation reports. The authors conclude that their ontology-based approach significantly improves the documentation and validation of architecture rules, paving the way for more effective architecture enforcement in software development. The downside of this approach is the requirement to formalize the architectural rules manually. While the CNL allows the use of these rules as documentation that can also be understood by architects and developers, learning the CNL and formulating the rules requires additional effort.

Zhong et al. [303] present DOMICO, an approach for checking the conformance between domain models and their implementations in microservices architecture, specifically using domain-driven design (DDD). The authors argue that existing conformance-checking techniques do not adequately address the specific needs of DDD-based projects, thus motivating the development of DOMICO to formalize and automate the conformance-checking process. DOMICO is designed to verify the existence of pattern-related elements and their compliance with a set of predefined constraints derived from eight common DDD patterns. The approach utilizes reflexion analysis to identify discrepancies between the intended model and the implementation, categorizing them into convergence, modification, divergence, and absence. Additionally, it defines 24 compliance rules to detect violations of pattern constraints. The authors provide a detailed formalization of these patterns and their representations in both models and code, allowing for a systematic and automated checking process. The evaluation of DOMICO is based on a case study involving a real-world supply chain project. This study demonstrated a 100% accuracy rate in identifying inconsistencies between the domain model and its implementation. The results indicate that DOMICO effectively detects both discrepancies and compliance violations, thereby supporting the maintenance of architectural integrity in microservices. The authors conclude that DOMICO enhances the conformance-checking process and seamlessly integrates into the regular domain modeling workflow, providing significant benefits for practitioners in the field. The biggest difference to my approach is the focus on DDD for microservice architectures and its usage of domain models. The authors use common DDD patterns that do not necessarily exist in other projects.

To check conformance and alignment with reference architectures, Bucaioni et al. [30] introduce the concept of continuous conformance, which quantifies the degree of adherence of a software architecture to a designated reference architecture through a distance function. This approach allows for incremental, non-blocking checks and restoration tasks, facilitating the design process without interruptions. This process includes manually mapping the components and connectors of a concrete architecture to an abstract architecture (i.e., the reference architecture). The authors implement the continuous conformance concept through an assistive modeling tool designed to architect Internet of Things systems. This tool supports architects by automatically identifying and rectifying misalignments with reference architectures, enabling multi-level and partial architecture checks. The continuous conformance process is derived from extensive research, including systematic literature reviews and expert interviews. The tool is built on model-based techniques, allowing architects to specify various architectural elements while receiving real-time feedback on conformance. The authors conducted a case study using a real-world IoT scenario for the evaluation. They addressed two research questions: whether the tool meets the identified challenges and what its strengths and weaknesses are. The results demonstrated that the tool effectively identifies architectural violations and provides guidance for resolution, showcasing its utility in modeling and continuous validation. Expert feedback highlighted strengths such as the ability to perform partial compliance checking and customization of conformance checks. However, weaknesses were also noted, including concerns about manual input requirements and the tool's focus on architecture analysis rather than implementation. This approach consequently differs from my approach for ID by requiring manual input, e.g., for the mapping between the implemented architecture and its reference architecture. Furthermore, the approach focuses on checking if the reference architecture is correctly implemented, which is a different goal from checking the consistency of an architecture with its documentation.

Besides these approaches, there are tools that help developers analyze and control the quality of their software systems. These tools can also be used for software architecture conformance checks, especially for automated code checks. Examples for these tools include ArchUnit[2], CQLinq[3], Sonargraph[4], and jQAssistant[5]. However, these tools require manual effort to create queries or rules and the checks are usually project-specific, reducing their reusability. Still, these are valuable tools that might be combined with other novel approaches to generate these queries or rules. In my work, I do not focus on the architecture conformance of code, but future work might explore these options further.

3.3.5. Other domains

In model-driven software development, various approaches tackle consistency preservation. Many approaches use bidirectional transformations and consistency preservation rules [252, 253, 58]. However, model-to-model inconsistency with, often manually created, consistency preservation rules tackle slightly different problems. Moreover, these directional transformations have to deal with different difficulties, such as the propagation of transformations in transformation networks. These approaches generally tackle a different use case (i.e., keeping formal models consistent using manually defined transformation rules and semi-automated resolution) and problems (e.g., concurrent editing or cyclic dependencies). Apart from software engineering, other fields also employ ID. For example, there is ID for knowledge bases and ontologies, like the approach by Töpper et al. [272]. In linguistics, there are works that try to detect inconsistencies, e.g., in summarization [151] or for structural ambiguity [266]. In biomedical informatics, there is work that, for example, addresses the identification of annotation inconsistency [299, 170, 280, 34] in different use cases. In mechanical engineering, there are model-based processes that use similar approaches to the software engineering domain [64]. However, in all these fields, inconsistency has other notions for the underlying problems to identify and tackle inconsistency.

[2] `https://www.archunit.org/`, last accessed 27.11.2024

[3] `https://www.ndepend.com/docs/cqlinq-syntax`, last accessed 27.11.2024

[4] `https://www.hello2morrow.com/products/sonargraph`, last accessed 27.11.2024

[5] `https://github.com/jqassistant`, last accessed 27.11.2024

3.3.6. Conclusion and Implications

Overall, several approaches deal with some form of ID. Some approaches are closer to the problem that this work tackles, for example, the approaches that detect inconsistencies between natural language text-like requirements and code and the approaches for architecture conformance. Other approaches tackle more distant problems but employ techniques that can be useful, e.g., employing TLR to link the artifacts beforehand. However, there are few to none looking at SAMs or SADs. Additionally, many approaches use structured input instead of unrestricted natural language texts. Processing structured forms of inputs like formal requirements or controlled natural language differs from my case with unrestricted, informal SADs. An approach for ID between SADs and SAMs does not yet, to the best of my knowledge, exist.

Part II.

Contributions

4. Taxonomy for Design Decisions in Software Architecture Documentation

When developing a software system, developers make numerous design decisions that influence the development process, the resulting architecture, and the final software system. Design decisions play a critical role in shaping the architecture of software systems, impacting everything from performance and scalability to maintainability and security. Capturing these design decisions in documentation is crucial, as it makes the decisions readily accessible for various purposes, such as onboarding new team members or facilitating maintenance. As Parnas emphasized, the success of a software system significantly depends on the quality of its documentation [209]. However, the documentation of these decisions is often inconsistent, making it difficult for stakeholders to understand the rationale behind architectural choices.

In this chapter, I introduce a comprehensive taxonomy for design decisions in software architecture documentation that organizes these decisions into well-defined and fine-grained categories. Further, I detail the methodology and dataset I used for its development. As such, I tackle **RQ 1** in this chapter. This chapter is based on our publication at the joint workshops MSR4SA (Mining Software Repositories for Software Architecture) and SAEroCon (Software Architecture Erosion and Architectural Consistency) at the European Conference on Software Architecture (ECSA) in 2022 [125].

In software architecture documentation, different kinds of Architectural Design Decision (ADD) exist. Some decisions are about the existence of components, while others describe the development process or define design rules. Some ADDs play an important role in tackling Traceability Link Recovery (TLR) and Inconsistency Detection (ID), influencing these tasks significantly.

For TLR, knowledge about the types of ADDs can enhance the understanding of the problem domain. Concrete, the creation and application of the taxonomy helped me understand SADs and extended my knowledge about, e.g., existing styles, typical layouts, and varying abstraction levels. Thus, the taxonomy can help to grasp the intricacies of SADs for TLR. Moreover, the taxonomy might later improve the corresponding approaches. For instance, filtering for specific ADDs can enhance precision by allowing unrelated parts of the documentation to be bypassed. Additionally, a fine-grained taxonomy could enable the mapping of more specific design decisions to related artifacts in the development process, improving the accuracy of traceability links.

For ID, knowing the specific types of ADDs enables the application of more precise and targeted analyses to verify the consistency of certain ADDs. For example, verifying the existence of an entity like a component requires different checks than ensuring compliance with a layered architecture or rules prohibiting dependencies between microservices. Several existing taxonomies classify ADDs, such as Kruchten's ontology of design decisions [146], the architectural design decision model introduced by Jansen & Bosch [112], and the architectural design decision rationale framework by Falessi et al. [59]. However, these taxonomies are often too broad for at least one of our purposes: identifying inconsistencies. For example, there is the category behavioral design decision of Kruchten's ontology [146]. The category is broad and concerns different specific behavioral decisions. Some behavioral decisions are about relations like call dependencies or similar interactions between components. Other behavioral decisions discuss concrete functionality. Checking the consistency of the described call dependencies clearly differs from checking the availability of some documented functionality. For this reason, more fine-grained classes are necessary to achieve a higher level of precision in our analyses.

Furthermore, creating a fine-grained taxonomy can offer several additional benefits. For example, it can enhance clarity and communication. By categorizing design decisions into more specific classes, the taxonomy might enable clearer communication, where everyone precisely understands the decisions and their implications. Further, it might improve documentation quality, as awareness of the fine-grained classification can help organize documentation in a more structured manner, making it easier to navigate and comprehend, thus improving the overall documentation quality. Lastly, understanding the specific types of design decisions allows for more effective impact analysis when changes are made. This helps anticipate potential issues and mitigate risks associated with modifications to the architecture.

In **RQ 1**, I ask about the kinds of fine-grained Architectural Design Decisions (ADDs) that can be found in Software Architecture Documentations (SADs). I want to organize ADDs into well-defined and fine-grained categories to answer this. I first discuss the methodology and dataset used to create the taxonomy in Section 4.1. Then, in Section 4.2, I present the resulting taxonomy.

To effectively and efficiently apply the taxonomy and to use the classifications in automated approaches for ID, an automated classification approach is required. Although this is not the main focus of this dissertation, I present some exploratory studies for automated approaches in Section 4.4, tackling **RQ 2** about the automated classification of ADDs.

4.1. Methodology & Dataset

The process of creating the taxonomy follows the guidelines established by Ralph [224], incorporating Bedford's principles [17].

Ralph recommends the following steps when creating a taxonomy: Choosing a strategy, selecting a site, collecting data, analyzing data, conceptually evaluating the taxonomy, writing up, and peer review. Regarding possible strategies, he recommends secondary studies, grounded theory, and interpretive case studies. Often, researchers iteratively collect and analyze data to benefit from preliminary findings and intermediate results and gain further insights.

Bedford introduces the following principles for a taxonomy: *consistency*, *affinity*, *differentiation*, *exclusiveness*, *ascertainability*, *currency*, and *exhaustiveness*. *Consistency* dictates that the procedures for creating, modifying, and retiring categories should remain uniform. *Affinity* asserts that each category's definition should be derived from its parent category, with lower hierarchy categories being more specific. Additionally, sub-categories should be created in pairs or more (*differentiation*). However, there does not need to be a differentiation on all levels. All categories must be *exclusive*, ensuring no overlap, with each category possessing a distinct and well-defined scope. Names must be clear and easily understandable (*ascertainability*), and they should reflect

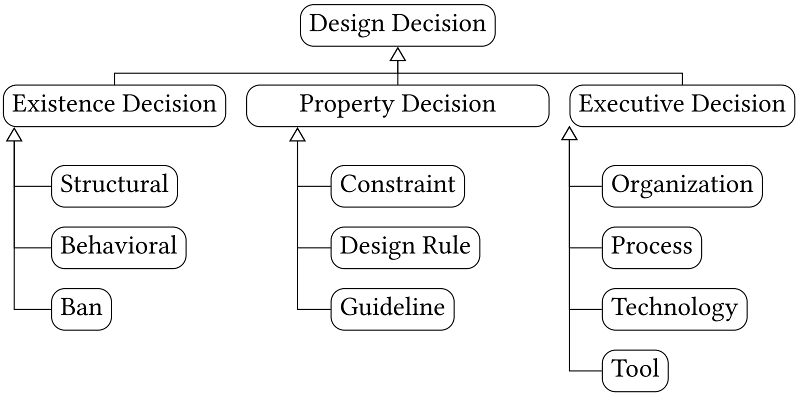


Figure 4.1.: Kruchten's Ontology of Architectural Design Decisions in Software-Intensive Systems [146]

the language pertinent to the domain (*currency*). Ultimately, categories must be *exhaustive*, covering the entire domain comprehensively.

Following these guidelines and principles, an initial taxonomy is created and then iteratively refined by classifying sentences from the documentation of 17 open-source projects. After each classification step, the gained insights are used to identify and address shortcomings in the taxonomy.

The initial taxonomy is influenced primarily by Kruchten's ontology [146], as depicted in Figure 4.1. It includes classes on *Existence*, *Property*, and *Executive* decisions, further enriched by insights from Jansen and Bosch [112], Bhat et al. [20], and Miesbauer and Weinreich [182]. For instance, *existence* decisions are the most common kind of ADDs [182] and should be subdivided to take (fine-grained) differences of these decisions into account. *Structural* decisions, based on Bhat et al.'s classification rules [20], are divided into internal structure decisions and those involving third-party systems and external dependencies such as libraries and plugins. Literature on software architecture and design (e.g., [44, 257, 259, 83, 231]) also informs these adaptations. Finally, the taxonomy is adjusted considering the specific requirements of the intended application areas.

The initial taxonomy is applied iteratively to SADs, and adaptations are made based on observations during classification. The first iteration involves a single case study as a pre-study. Subsequent iterations use three case studies

Table 4.1.: The 17 projects that were used for developing and evaluating the taxonomy. *#Lines* show the number of lines in each documentation file (without empty lines). The names link to the projects; these links can also be found in the replication package (see Section 1.7) and in Keim et al.'s supplementary material [126], along with the used documents.

Project	Domain	#Lines
CoronaWarnApp	Healthcare	369
TEAMMATES	Teaching	252
Beets	Media	125
ROD	Data Mgmt.	119
ZenGarden	Media	109
MyTardis	Data Mgmt.	100
SpringXD	Data Mgmt.	95
QMiner	Data Analysis	92
IOSched	Event Mgmt.	81
SCons	Software Dev.	79
OnionRouting	Networking	51
Spacewalk	Operating System	38
Calipso	Web Dev.	30
MunkeyIssues	Software Dev.	23
BIBINT	Science	22
OpenRefine	Data Mgmt.	21
tagm8vault	Media	16

per iteration, allowing for early identification of necessary adaptations and minimizing the risk of overfitting. Iterations continue until the taxonomy stabilizes, defined as no adaptations being required over two consecutive iterations, totaling six SADs without changes.

To apply the taxonomy, English SADs from open-source projects on GitHub were mined. This involved querying GitHub for "architecture". From the first 500 results, we randomly selected 50 projects. These were manually filtered based on criteria such as English documentation with acceptable text quality (i.e., mostly correct grammar and spelling), varying SAD sizes, and diverse domains. Overall, 17 case studies of different sizes and domains were used in the process, as listed in Table 4.1 (see also the dataset in the supplementary material [126]).

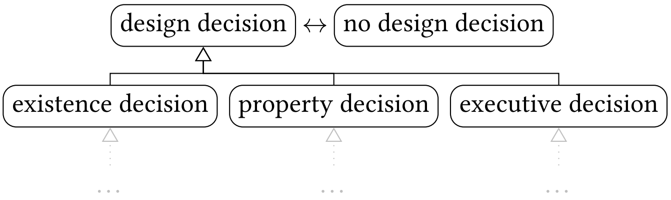


Figure 4.2.: High-level view on the taxonomy, based on Keim et al. [125]

After creating the taxonomy, we evaluate the taxonomy both argumentatively and empirically (see Section 4.3).

4.2. Resulting Taxonomy

In this section, I describe the taxonomy developed by applying the methodology outlined in Section 4.1 using the specified dataset (see also [125]). The high-level classes of this taxonomy are illustrated in Figure 4.2.

At the root of the taxonomy is the decision regarding the existence of an ADD. When an ADD is present, the subsequent child classes adhere to Kruchten's ontology [146], categorizing ADDs into *existence decisions*, *property* decisions, and *executive* decisions.

A clear distinction from Kruchten's ontology is the omission of *ban* decisions as a subcategory of existence decisions. We interpret *bans* as the negative version of ADDs, functioning as orthogonal properties to ADDs. Consequently, *bans* can also concern other categories, such as *executive* decisions. For instance, there might be *existence* decisions regarding the exclusion of a specific component or *executive* decisions concerning the prohibition of using a certain tool or process. Thus, our model does not include a separate class for exclusions.

Similarly, delayed decisions, which involve postponing decisions until a later time (e.g., when more information becomes available), represent another type of decision property not incorporated into this class hierarchy. These deferred decisions highlight the temporal aspect of decision-making, which is not explicitly modeled in our framework.

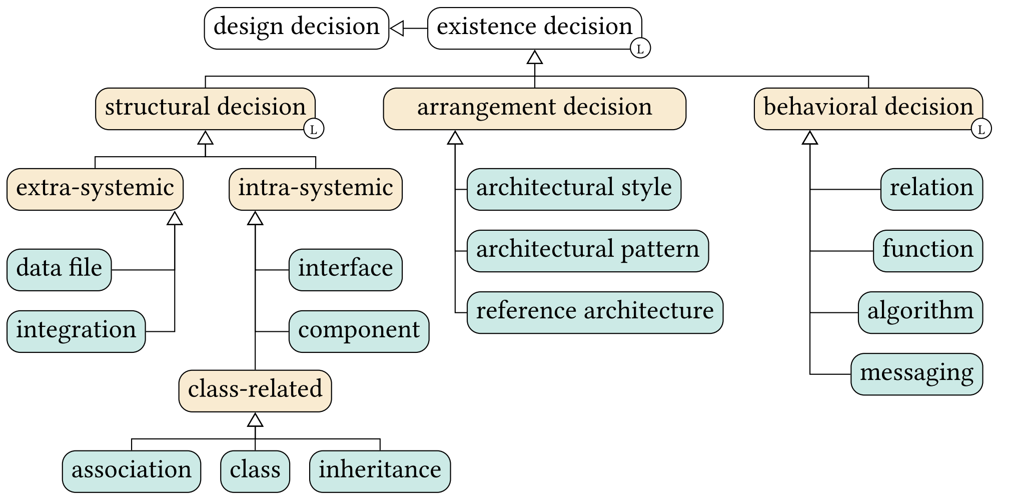


Figure 4.3.: Sub-categories of existence decisions with leaves highlighted in green and intermediate categories in orange. Categories derived from literature are labeled with Ⓛ. Taken from Keim et al. [125].

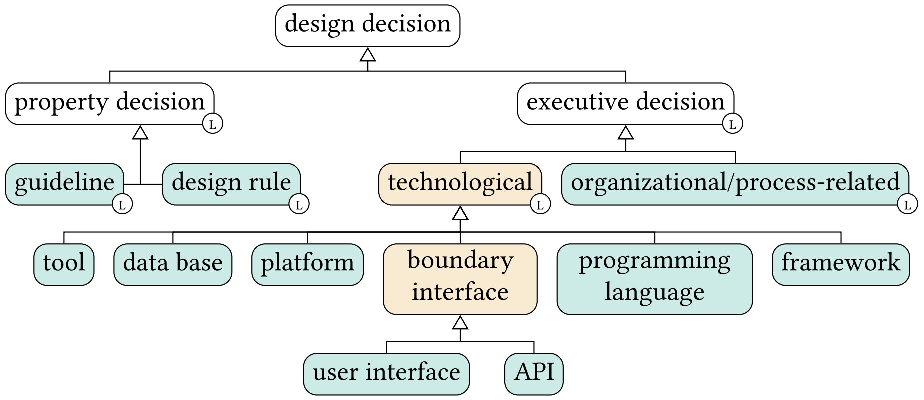


Figure 4.4.: Sub-categories of property decisions and executive decisions with leaves highlighted in green and intermediate categories in orange. Categories derived from literature are labeled with Ⓛ. Taken from Keim et al. [125].

In Figure 4.3 and in Figure 4.4, the lower levels of the taxonomy are displayed. The taxonomy divides *existence* decisions into *structural* decisions, *arrangement* decisions, and *behavioral* decisions.

Structural decisions can be further categorized into *extra-systemic* and *intra-systemic* decisions. We define *systemic* components as those parts of the executable system that are specifically developed for it. In contrast, *extra-systemic* components include all external influences on the system, such as libraries or external plug-ins. This distinction allows us to differentiate between external elements and those developed exclusively as part of the system. Such differentiation is crucial for consistency analyses, as external components are typically represented as external calls requiring different consistency checks than explicitly modeled internal components.

In the following, I provide detailed definitions and descriptions of the classes within the taxonomy developed by Keim et al. [125].

Existence decisions: State whether some software element or artifact will be present in the system's design or implementation [146].

structural decision: Break down a system into reusable subsystems and components.
extra-systemic: Decisions that use software elements beyond the system's borders to add them to the system under development or to feed in data.
data file: Non-executable files that provide information that can be transferred and shared between (sub-) systems.
integration: In this category fall externally developed software elements like libraries and plug-ins, but also reused components from other projects/systems that are not specifically developed for the current system under development.
intra-systemic: Decisions affecting the division of a system in subsystems and smaller units developed for this particular system.
interface: Provide "a declaration of a set of public features and obligations that together constitute a coherent service" [44]. Interfaces within a software system serve to bundle functionality together.
component: A unit of composition with contractually specified interfaces and explicit context dependencies [259]. We additionally recognize packages as such compositions. This category comprises decisions about components that are developed for the system under development, not imported components (cf. *integration*).

class-related: Deal with the development of software systems along classes of objects. Even if class-related decisions may not be architectural in general, a lot of architecture documentation contains them, so we include them in our taxonomy.
class: A set of objects with consistent properties and functionality [44]. One can make decisions about its existence, structure, and naming.
association: A set of links that are tuple(s) of values that refer to typed objects [44].
inheritance: Through *inheritance*, a child class adopts properties of its parent class. Design decisions in this category include the existence and design of inheritance hierarchies and determining (abstract) parent and child classes.

behavioral decision: Relate to how software elements are connected and how they interact to provide functionality [146].
relation: Decisions about connectors and dependencies between software elements that can be established, modified, or excluded. This determines the accessibility of the functionality of the elements.
function: Decisions about specific functionality of a software system. These *functions* are usually implemented as methods.
algorithm: Refer to a named sequence of operations to realize certain functionality for a specific problem. The procedures must be sufficiently general and state the general goal. Otherwise, it should be categorized as *function* or *messaging*.
messaging: Decisions about functionality concerning communication between software elements through method calls, sending messages, and data packages (cf. [100]). With messaging, the sender usually invokes some behavior at the receiver.

arrangement decision: Decisions for an *arrangement* of software elements in a known manner and under consideration of related principles. Decisions in this category will affect both the structure and behavior of the system.
architectural style: Provide solution principles for architectural problems that are independent of the given application and should be used throughout the whole architecture [235]. Examples are Layered Architecture and Client-Server architecture.
architectural pattern: A proven, reusable solution to a recurring problem. The scope is broader (concerning components) compared to design patterns (concerning code base parts) [231]. Compared to architectural styles, architectural

patterns are more specific to a certain problem. Examples for architectural patterns are Model-View-Controller and Domain Model.
reference architecture: Defines a set of ADDs for a specific application domain [231]. The AUTOSAR architecture is an example of the automotive domain.

Property decisions: State an enduring, overarching trait of quality of a system [146].

design rule: A rule, positively or negatively stated, expresses some trait or quality the system design must strictly fulfill. This is a combination of Kruchten's classes *design rules* and *constraint* [146]. This does not include observable quality attributes such as performance or reliability.
guideline: Recommended practices that improve the system's quality. They are less strict than design rules and usually not enforced.

Executive decisions: Relate to environmental aspects of the development process [146].

organizational/process-related: Sum up all decisions concerned with the development process, the methodological procedure, and the project organization.
technological: State the choice for or against numerous technologies that enable and support software development.
tool: Developer *tools* can support and automate the development process.
data base: Decisions referring to storing data in *databases* as well as decisions on query languages (e.g., SQL) and database technologies (e.g., MongoDB).
platform: Decisions that refer to an environment of software and hardware components that allow development, deployment, and execution.
programming language: An agreed *programming language* for a system or components leads to syntactical and semantic rules for writing code.
framework: Abstractions that allow the development of extensive applications. A framework can either be an architecture framework (cf. [110]) or a software framework like a web framework (e.g., Django).
boundary interface: Interfaces that are located on the system's boundary and enable the connection from and to other systems and technologies.
API: *Application programming interfaces* provide external functions that can be used in another software system. If an ADD is not about the functionality of communications but about used protocols like HTTP and TCP, we also

Table 4.2.: Distribution of design decisions in the dataset, taken from Keim et al. [125]

Taxonomy class					Primary	Secondary	Total
existence	structural	extra		integration	38	7	45
				data file	17	6	23
		intra		component	132	23	155
				interface	19	0	19
			class-r.	class	77	26	103
				association	65	12	77
				inheritance	19	3	22
	arrang.			architectural style	26	14	40
				architectural pattern	21	3	24
				reference architecture	4	2	6
	behavioral			function	271	43	314
				relation	56	14	70
				algorithm	48	6	54
				messaging	37	11	48
property				design rule	73	0	73
				guideline	8	2	10
executive				organizational/process-related	9	1	10
	technological			platform	72	15	87
				programming language	39	19	58
				framework	32	13	45
				data base	40	14	54
				tool	5	2	7
		boundary i.		API	51	12	63
				user interface	31	10	41
identified design decisions (in 1622 sent.)					1190	258	1448
sentences without a design decision							432

classify it into this class.
user interface: Located between the (human) user and the technical system, and allows the user to enter commands. There are different kinds of user interfaces, such as GUIs or CLIs.

Table 4.2 presents the distribution of decisions within the dataset. The dataset comprises 1622 sentences from 17 SADs. The sentences are classified based on the most prevalent primary ADD and, if present, the secondary ADD per sentence. In rare cases, there are more than two ADDs in one sentence. In such cases, the annotators decided which ADD are the most central ones for the sentence. Among the sentences in the dataset, 432 sentences do not have any ADD, while the remaining sentences contain a total of 1448 ADDs. Additionally, 258 sentences include a secondary ADD.

Overall, the majority of ADDs are *existence* decisions, with *function* being the most prevalent class. The second most common ADDs are *intra-structural* decisions, particularly concerning *classes* and *components*.

The rarest classes are *reference architecture* and *tool*. This scarcity is expected, as ADDs regarding reference architectures are only necessary when they are employed. In such cases, one single statement per documentation typically suffices. Likewise, tools also only need a few statements if and only if they are employed.

4.3. Evaluating the Taxonomy

In this chapter, I presented a taxonomy for design decisions in SADs to answer **RQ 1**. The purpose of the taxonomy is to classify design decisions in SADs to facilitate the identification of inconsistencies. The idea is that the different classes might need different approaches to assess the consistency of the statements with information from other artifacts.

In this section, I evaluate the taxonomy. I base the evaluation on the Goal-Question-Metric (GQM) approach by Basili et al. [15]. I define the goals (G) and derive corresponding questions (Q) for each goal to examine the fulfillment of the goals. To answer the questions, I use various metrics (M). In cases where my evaluation is qualitative instead of quantitative, I use argumentation (A) instead of metrics. I present the GQM plan in Section 4.3.1.

Then, I plan to fulfill the goals set in the GQM plan (see Figure 4.5) by answering the respective questions about the purpose (Question **Q1.1**), structure (Question **Q1.2**), and reliability (Question **Q1.3**) using the stated metrics in the corresponding sections.

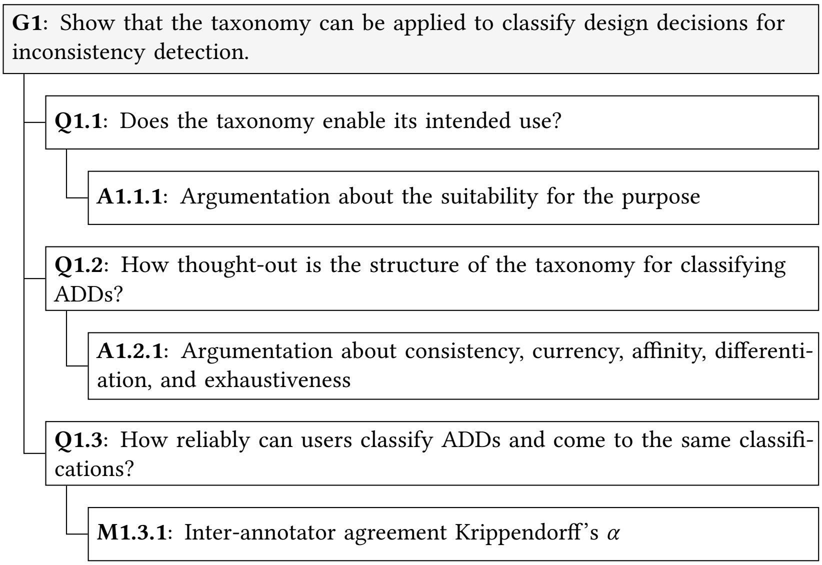

G1: Show that the taxonomy can be applied to classify design decisions for inconsistency detection.
Q1.1: Does the taxonomy enable its intended use?
A1.1.1: Argumentation about the suitability for the purpose
Q1.2: How thought-out is the structure of the taxonomy for classifying ADDs?
A1.2.1: Argumentation about consistency, currency, affinity, differentiation, and exhaustiveness
Q1.3: How reliably can users classify ADDs and come to the same classifications?
M1.3.1: Inter-annotator agreement Krippendorff's α

Figure 4.5.: Goals, Questions, and Metrics for Evaluating the Taxonomy for Architectural Design Decisions

This chapter is based on our publication at the joint workshops MSR4SA (Mining Software Repositories for Software Architecture) and SAEroCon (Software Architecture Erosion and Architectural Consistency) at ECSA in 2022 [125].

4.3.1. Evaluation Plan

To evaluate the taxonomy, I derive a GQM plan from the principles of Bedford [17] and the guidelines by Ralph [224]. Figure 4.5 shows the resulting GQM plan for the taxonomy.

In the evaluation, I aim to show that the taxonomy can be applied. Consequently, I consider questions about the purpose (Question **Q1.1**), the structure (Question **Q1.2**), and the reliability when applying the taxonomy (Question **Q1.3**).

To answer Question **Q1.1** about the purpose, I argue about the suitability for the purpose. Similarly, I answer Question **Q1.2** (structure) by arguing about consistency, currency, affinity, differentiation, and exhaustiveness. In Question **Q1.3** (reliability), I evaluate the usability, application, and handling of the taxonomy, in particular, the exclusiveness and ascertainability using the inter-annotator agreement Krippendorff's α (see also Section 2.8.3). A high value there can show that multiple annotators reliably come to the same conclusions when applying the taxonomy to classify SADs.

4.3.2. Evaluating the Purpose

In this section, I argue about the ability of the taxonomy to fulfill the intended use to answer Question **Q1.1**. The main purpose of the taxonomy is to enable consistency checks between SADs on one side and SAMs or code on the other side. The idea is that there is a need for different approaches to analyze different kinds of design decisions.

To enable the selection of different approaches for inconsistency detection, the taxonomy's classes need to be clearly differentiated and non-overlapping (i.e., exclusive). Furthermore, classes must capture concepts and design decisions that require different approaches. This necessity is evident for classes belonging to different upper categories. For instance, structural decisions can be validated for consistency using a structural perspective on the architecture, whereas executive decisions cannot be assessed similarly.

This principle also applies to classes within specific sub-categories. Approaches must treat components differently than interfaces or associations for intra-structural existence decisions, as each has different attributes and involved elements. In contrast, verifying inheritance structures requires the analysis of relationships between components or classes. For extra-structural existence decisions, integrating libraries or plug-ins clearly is represented differently, usually within some dependency management configuration, than non-executable data files, for which the existence of the concrete files have to be confirmed. While the arrangement decisions about architectural styles, patterns, and reference architectures look similar at first, they differ vastly and, thus, require different validation. On one hand, styles are used throughout the system, so their validation requires an extensive look into all parts of the system. On the other hand, architectural patterns are more specific to certain areas of the system. At the same time, reference architectures require alignment checks.

Similarly, behavioral decisions differ. Treating named algorithms requires external knowledge to capture all involved parts. Functions require the analysis of a certain behavior, while relations require an analysis of the relationships. Various software elements are involved in messaging, and a specialized consistency check needs to find the involved elements, identify their relationship, and confirm the message exchange. Often enough, certain message systems like message brokers or message-oriented middleware exists and their usage of the involved elements need to be confirmed. Property decisions differ by the enforcement level of the two classes and by the implications. For example, checking design rules requires analyzing a certain trait or quality.

Executive decisions mostly differ in how and where they can be checked. For example, organizational and process-related decisions can most likely not be analyzed using typical development artifacts. Further, checking the usage of a platform clearly differentiates from checking the usage of a programming language or a specific framework. Analyzing the use of certain databases and tools also requires specialized tests, as does providing APIs or user interfaces.

Overall, I assess that generic approaches will likely fail at ID without considering the specific classes of the taxonomy. Consequently, I conclude that the purpose of this taxonomy is well-defined and that the taxonomy is appropriately aligned with this purpose. Nonetheless, additional evidence must be gathered in future work by developing applications that utilize the taxonomy.

4.3.3. Evaluating the Structure

In this section, I discuss the proposed taxonomy's structure to answer Question **Q1.2**.

The taxonomy is based on related work and refines the ontology by Kruchten [146]. Kruchten's ontology is widely used in the community, serving as a foundation for the automated classification of design decisions. Consequently, the taxonomy can be regarded as consistent with existing work.

The primary distinction between our taxonomy and Kruchten's ontology is how it handles inclusive/exclusive wording. Instead of explicitly modeling ban decisions as a sub-category of existence decisions, our taxonomy treats inclusive/exclusive wording as an attribute of each design decision. This approach allows for exclusions to be made on various occasions and at fine-grained levels. For instance, it enables the modeling of negative executive decisions, e.g., bans on certain programming languages. Given that such situations arose in our case studies (e.g., project Zengarden, line 129: "C++ is not the ideal language"), we argue that our taxonomy is more comprehensive. One example of a negative decision in the dataset [126] includes line 26 in the project Calipso: "modifications to configuration [..] will not be applied across nodes in a cluster automatically".

In refining the taxonomy, we adhered to Bhat et al.'s guidelines for the manual identification of design decisions [20].

The taxonomy uses well-known and widely accepted terms within the software architecture domain to reflect the domain's language and to ensure currency (see Bedford [17]). Adding subclasses in a top-down fashion to existing classes ensures the affinity of each subclass to its parent classes. Moreover, each refined class has at least two subclasses to accomplish differentiation.

Regarding the taxonomy's exhaustiveness, we acknowledge the possibility that other case studies might reveal decisions necessitating additional classes. Nonetheless, we consider the taxonomy at least as exhaustive as related work because its upper classes are based on Kruchten's ontology. For these upper classes, related work ([20, 182]) has demonstrated their sufficiency for classifying design decisions.

Finally, the applied iterative approach follows the methodology of the National Information Standards Organization [7]. Even when dealing with heterogeneous documentation in terms of length and domain, this approach did not result in additional (sub-)classes.

4.3.4. Evaluating the Application

The evaluation of the applicability and reliability to answer Question **Q1.3** builds upon the guidelines by Kaplan et al. [119].

Table 4.3.: Inter-Annotator Agreement using Krippendorff's α (Kα) and Gwet's AC1 (GAC1) as well as Gwet's AC1 when not considering true negatives (GAC1$_{\text{noTN}}$).

Project	Kα	GAC1	GAC1$_{\text{noTN}}$
Calipso	0.917	0.991	0.832
Spacewalk	0.945	0.994	0.891
SpringXD	0.661	0.970	-0.066
Overall	0.841	0.979	0.552

Two software engineering doctoral researchers independently classified the design decisions in three randomly selected case studies from the dataset (Calipso, Spacewalk, and SpringXD). The subjects received only the information on the taxonomy provided in Section 4.2.

We use the resulting classifications to calculate Krippendorff's α (Kα) [143] to determine the inter-annotator agreement (see Section 2.8.3). The results are displayed in Table 4.3. The overall Kα across the three studies is 0.771. If the results are calculated per project and, on average, achieved a Kα of 0.841. The variance comes from the project SpringXD, which is seemingly more difficult to document. The documentation is more lengthy and consists of various parts discussing things unrelated to architecture. As such, we argue that the quality of the architecture documentation also influences the applicability and reliability.

Nevertheless, the resulting values for Kα show a reasonable agreement that exceeds the lower bound of 0.66 and approximates the commonly accepted threshold of 0.8 (cf. Krippendorff [144]). According to the taxonomy by Landis and Koch [153], the overall results show a substantial agreement, and the project-averaged results show almost perfect agreement. Given the taxonomy's size with 24 leaf classes, these results are very promising. Moreover, the annotators also identified a fitting class for each decision, showing that the taxonomy can likely be applied in cases where a refined classification of design decisions is beneficial.

We additionally calculate Gwet's AC1 (see Section 2.8.3) to assess the inter-annotator agreement. The overall Gwet's AC1 across all three studies is 0.979, showing very high agreement. The resulting value when averaging the results calculated per project is 0.985. According to Landis and Koch [153], this is an almost perfect agreement. Again, the lowest-performing project is SpringXD, where annotators disagree most due to many architecturally irrelevant statements.

However, there is an issue with calculating Gwet's AC1. Due to the multi-class property, there are a lot of true negatives that influence the calculation. While choosing not to select a class is reasonable and important, the high number of true negatives inflates the score.

When changing the calculation slightly by only considering the true positives, false positives, and false negatives, the score changes considerably, as Table 4.3 shows. For the project Calipso, the score changes from 0.99 to 0.83. For Spacewalk, the score drops by ten percentage points from 0.99 to 0.89. Considering these two projects, the agreement is still almost perfect.

The project SpringXD shows the biggest change. As discussed before, the annotators disagree due to many architecturally irrelevant statements. While one annotator skipped irrelevant statements, the other annotator still tried to apply the taxonomy. The annotators agreed in 60 cases, but one annotator labeled a class, whereas the other didn't in 65 cases. Consequently, this results in the score for Gwet's AC1 of -0.07, showing no agreement. This shows how important it is to clearly state the application area of the taxonomy to avoid such a scenario.

4.3.5. Discussion and Conclusion

The evaluation tackled **RQ 1** and demonstrates that the proposed taxonomy is well-aligned with its intended purpose, structurally sound, and generally applicable with a high degree of reliability. The taxonomy's differentiation between design decisions and its foundation on established frameworks contribute to its effectiveness. However, the evaluation also identifies areas for improvement, such as the need for additional classes in future case studies and the potential for discrepancies in application due to varying documentation quality.

The findings suggest that while the taxonomy shows promise, especially in terms of consistency and comprehensiveness, further validation in practical applications is necessary to confirm its utility and address identified limitations. Future research should focus on applying the taxonomy in diverse contexts and refining it based on the outcomes to enhance its robustness and applicability in software architecture.

However, it is important to note that there are several threats to validity, that I discuss based on the guidelines by Runeson and Höst [235].

We applied a common experimental design and metrics to reduce threats w.r.t. *Construct Validity*. Still, there is a potential bias in the selection of use cases. To reduce the bias, we selected projects from varying domains with different characteristics, including size, architectural styles, and architectural patterns.

Regarding *Internal Validity*, there can be several forms of bias, especially in the labeling process. During the labeling process, annotators labeled only the most prevalent design decisions and, additionally, the most obvious secondary, implicit decisions per sentence. In some cases, the decisions which decisions are primary and secondary can be biased. Moreover, in some rare cases, there are more ADDs in the sentences, and thus, there can be bias in the selection.

Regarding *External Validity*, we examined various publicly available case studies from different domains in the construction and evaluation. The goal of the selection is a representative selection, but there is the risk that not all facets and aspects of ADDs are contained. The dataset represents every class, but some classes only have a few representatives (see also Table 4.2). Classes like reference architecture naturally occur only once or a few times per documentation, consequently making them less represented compared to other design decisions, e.g., about components.

To ensure *Reliability*, we followed a rigorous approach for the labeling and construction of the taxonomy. With this approach, we aim to ensure consistency and repeatability. However, there is still potential bias in the project selection.

Despite our efforts, certain threats and biases remain. Further research should, therefore, aim to mitigate threats and try to validate our findings.

4.4. Automated Classification of Architectural Design Decisions

The taxonomy for Architectural Design Decisions that I introduced in Chapter 4 aims to classify sentences in SADs for the underlying design decision.

With the information about the design decision, developers can check if the design decision is correctly realized. For example, developers can identify if there is an inconsistency between the design decision and a SAM. For this, different design decisions require different checks, as previously discussed.

Therefore, an automated approach to detecting inconsistencies requires knowledge about the kind of design decision to be able to execute the appropriate approach. For instance, the precision of the approach for detecting MMEs might be improved if only sentences that contain related ADDs are considered. A fully automated approach also needs to classify the design decisions automatically.

In the following, I tackle **RQ 2** and present an exploratory approach to automatically classify architectural design decisions, as presented at the joint workshops MSR4SA (Mining Software Repositories for Software Architecture) and SAEroCon (Software Architecture Erosion and Architectural Consistency) at ECSA in 2022 [125].

The classification of ADDs in SAMs is a rather unexplored field of research. However, there are fields that are closely related to classification tasks. One of these fields is Requirements Engineering, where requirements are classified into different classes like *functional* and *non-functional* or a subclass of these classes. Therefore, I first performed a preliminary study on requirements classification before developing initial approaches for classifying ADDs.

4.4.1. Preliminary Study: Requirements Classification

In the preliminary study, we develop a fine-tuned version of the LLM BERT for classifying requirements. The resulting approach is named Non-functional and functional Requirements classification using BERT (NoRBERT). This approach and the preliminary study is published at the IEEE International Requirements Engineering Conference (RE) in 2020 [98].

The main research questions of the study are:

1. How does transfer learning perform in classifying requirements?
2. Does transfer learning improve the performance of classifying requirements on unseen projects?

3. To what extent do transfer learning approaches detect subclasses of functional requirements?

We apply the approach to four tasks to assess the performance. Task one is the binary classification of requirements into *functional* (F) and *non-functional* (NFR) requirements. The second task is the binary and multi-class classification of the four most frequent subclasses of non-functional requirements (i.e., usability, security, operational, and performance). In task three, the approach needs to classify all non-functional requirement subclasses. These subclasses are Availability (A), Fault Tolerance (FT), Legal (L), Look & Feel (LF), Maintainability (MN), Operational (O), Performance (PE), Portability (PO), Scalability (SC), Security (SE), and Usability (US).

Lastly, task four requires the binary classification of requirement aspects (*functional* and *quality*, including overlapping aspects). The performance of NoRBERT on each task is assessed and tuned using the metrics precision (P), recall (R), and F_1-score (F_1).

To train and evaluate NoRBERT for these tasks, we use the commonly used datasets PROMISE by Cleland-Huang et al. [41] for the first three tasks and the relabeled variant by Dalpiaz et al. [46] for the fourth task.

The evaluation of the approach uses three kinds of cross-validation techniques: *k-fold*, *project-fold (p-fold)*, and *leave-one-Project-out (loPo)* cross-validation.

For the k-fold cross-validation, we first perform a ten-fold cross-validation, evaluating the approach similar to other requirement classification approaches. Moreover, we apply five-fold cross-validation and repeat this validation five times to compare our results with the results of Abad et al. [1].

Besides cross-validation, we also evaluate using the p-fold and the loPo validation. In ten-fold cross-validation, data from one project can be included in both the training set and the test set. This can influence the results as project-specific properties can already be seen in training. These two variants aim to investigate the transferability of approaches on unseen projects. P-fold is a cross-validation variant also used by Dalpiaz et al. [46] that splits the dataset ten times into three projects as the test set and twelve projects as the training set. Similarly, the loPo variant performs cross-validation, where each run uses the requirements of one project as the test set and the requirements of the other projects as the training set, repeated for every project.

Table 4.4.: Functional/Non-Functional requirements classification on PROMISE NFR dataset. Bold values show the highest score for each metric per class. [98]

	F (255)			**NFR (370)**		
Approach (Parameters)	P	R	F_1	P	R	F_1
10-fold						
K. & M. (word feat. w/o feat. sel.)	**.92**	.93	**.93**	.93	.92	.92
K. & M. (500 best word feat.)	**.92**	.79	.85	.82	.93	.87
K. & M. (500 best feat.)	.88	.87	.87	.87	.88	.87
A. et al. (unprocessed data)	.84	.93	.88	.95	.88	.91
A. et al. (processed data)	.90	**.97**	**.93**	**.98**	.93	**.95**
D. & F. (word2vec, ep.=100, f.=50)	—	—	—	.93	.92	.92
NoRBERT (base, ep.=10)	.91	.90	.90	.93	.94	.93
NoRBERT (base, ep.=10, ES, US)	.88	.88	.88	.92	.92	.92
NoRBERT (base, ep.=10, ES, OS)	.91	.86	.88	.91	.94	.92
NoRBERT (base, ep.=16)	.89	.88	.89	.92	.93	.92
NoRBERT (large, ep.=10, OS)	**.92**	.88	.90	.92	**.95**	.93
p-fold						
NoRBERT (base, ep.=10)	.85	.51	.64	.74	.94	.82
NoRBERT (large, ep.=16)	**.89**	**.61**	**.73**	**.78**	**.95**	**.86**
loPo						
NoRBERT (base, ep.=10, ES)	**.88**	.20	.64	.73	**.95**	.83
NoRBERT (large, ep.=10, US)	.87	**.71**	**.78**	**.82**	.93	**.87**

Table 4.4 shows the results for the binary classification of requirements into functional and non-functional (task 1). The results are compared against the approaches by Kurtanovic and Maleej [149], by Abad et al. [1], and by Dekhtyar and Fong [50]. It performs comparably with an F_1-score of 90% for functional and 93% for non-functional requirements. For non-functional requirements, only the approach by Abad et al. outperforms NoRBERT, which requires manually provided dictionaries and rules to preprocess the dataset. In contrast, the BERT-based approach requires no manual preprocessing, making it more applicable to new, unseen datasets or projects. Similarly, for functional requirements, NoRBERT is only outperformed by those variants of the competing approaches that need manual feature selection or preprocessing.

Table 4.5.: Multi-class classification of all non-functional requirements subclasses on NFR dataset. 16, 32, and 50 indicate the used epoch number, *bin* binary and *mult* multi-class classification, and B and L the used BERT model (base/large). bin_{16} additionally uses OS and multiL_{32} ES. LDA and NB (Naïve Bayes) refer to approaches by Abad et al. [1] with preprocessed data (P) or without (UP). [98]

		A (21)			**FT (10)**			**L (13)**			**LF (38)**			**MN (17)**			**O (62)**			**PE (54)**			**SC (21)**			**SE (66)**			**US (67)**			
	Model	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1	A
10-fold	bin_{16}	**.93**	.62	.74	.50	.20	.29	.82	.69	.75	.79	.71	.75	.54	.41	.47	**.87**	.77	.82	.88	.78	.82	.70	.67	.68	.88	.88	.88	**.87**	.72	.79	.78
	multiB_{16}	.73	.76	.74	**1.0**	.20	.33	.91	.77	.83	.77	.79	.78	**.86**	.35	.50	.77	.79	.78	.87	.83	.85	.68	.71	.70	.77	**.92**	.84	.86	**.91**	**.88**	.79
	multiB_{32}	.75	.71	.73	.38	.30	.33	.91	.77	.83	.81	.79	.80	.60	.35	.44	.78	.82	.80	.90	.83	.87	**.76**	**.76**	**.76**	.84	**.92**	.88	.78	.87	.82	.79
	multiB_{50}	.77	**.81**	**.79**	.60	.30	.40	.91	.77	.83	.80	.74	.77	.70	.41	.52	.83	**.84**	**.83**	**.94**	.87	**.90**	.64	.67	.65	.78	**.92**	.85	.78	.85	.81	.80
	multiL_{32}	.70	.76	.73	.56	.50	.53	**.92**	**.85**	**.88**	**.82**	**.87**	**.85**	.58	.41	.48	.74	.77	.76	.91	**.89**	**.90**	.70	.67	.68	.86	.89	.87	.86	.85	.86	**.81**
	multiL_{50}	.80	.76	.78	.60	**.60**	**.60**	.91	.77	.83	.81	.79	.80	.62	**.47**	**.53**	.78	.84	.81	.92	.87	**.90**	**.76**	**.76**	**.76**	**.90**	**.92**	**.91**	.83	.88	.86	**.82**
5x5-fold	LDA(P)	.60	.95	.74	.02	.10	.03	.20	.47	.28	.85	.60	.70	.52	.70	.60	.70	.35	.47	.95	.70	.81	.57	.81	.70	.87	.87	.87	.76	.61	.68	.62
	NB(UP)	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	.45
	NB(P)	**.90**	**.90**	**.90**	**.90**	**.97**	**.93**	**1.0**	.75	**.86**	**.94**	**.94**	**.94**	**.82**	**.90**	**.86**	**.91**	.78	**.84**	**1.0**	**.90**	**.95**	**.83**	**.83**	**.83**	**1.0**	**.97**	**.98**	.77	**.97**	**.86**	**.90**
	multiB_{32}	.73	.77	.75	.65	.26	.37	.80	**.82**	.81	.77	.75	.76	.54	.40	.46	.79	**.82**	.81	.86	.81	.83	.68	.73	.70	.83	.89	.86	**.79**	.83	.81	.78
p-fold	bin_{16}	**.82**	.64	.72	.00	.00	.00	.64	.35	.45	**.85**	.45	.59	.38	.15	.21	**.72**	.55	.62	**.88**	.53	.66	.71	.64	.68	.80	.86	.83	**.73**	.74	**.74**	.65
	multiB_{32}	.71	.76	.74	.33	.05	.09	.79	.42	.55	.60	.55	.58	.40	**.41**	**.41**	.67	.81	.73	**.88**	**.70**	**.78**	**.73**	**.69**	**.71**	.83	.83	.83	.65	.79	.72	.70
	multiL_{50}	.77	**.88**	**.82**	**.38**	**.15**	**.21**	**.83**	**.58**	**.68**	.79	**.59**	**.68**	**.46**	.35	.40	.71	**.85**	**.77**	.85	**.70**	.77	.66	.64	.65	**.86**	**.94**	**.90**	.68	**.80**	.73	**.74**
loPo	bin_{16}	**.80**	.57	.67	**.50**	**.20**	**.29**	.83	.38	.53	**.86**	.47	.61	**.55**	.35	**.43**	**.80**	.63	.70	.88	.52	.65	.58	.52	.55	.78	.85	.81	**.69**	.60	.64	.65
	multiB_{32}	.71	.71	.71	.40	.20	.27	.40	.15	.22	.66	.55	.60	.41	**.41**	.41	.75	.76	**.75**	**.92**	**.67**	**.77**	**.71**	**.71**	**.71**	.75	.86	.80	.62	**.84**	**.71**	.69
	multiL_{50}	.75	**.86**	**.80**	.20	.10	.13	**.89**	**.62**	**.73**	.77	**.63**	**.70**	.50	.35	.41	.72	**.79**	**.75**	.88	.65	.74	.67	.57	.62	**.83**	**.96**	**.89**	.63	.81	**.71**	**.72**

Table 4.6.: Results of NoRBERT in comparison to the results reported by [46] for Task 4, the binary classification of classes in the relabeled PROMISE NFR dataset. Bold values represent the highest score for each metric per class.

			F (310)			Q (382)			OnlyF (230)			OnlyQ (302)		
	Approach	Parameters	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
75%-split	K. & M. reimpl.	(100 best features)	.80	.78	.80	**.91**	.90	.88	.86	.82	.89	.86	.75	.81
	K. & M. reimpl.	(500 best features)	.82	.80	.82	**.91**	.89	.87	.87	.87	.91	**.90**	.85	.87
	Dalpiaz et al.	(final 17 features)	.71	.76	.73	.77	.80	.92	.77	.83	.72	.71	.75	.78
	NoRBERT	(base, ep.=10)	.86	**.88**	.87	.90	.96	.93	.89	**.98**	**.93**	.87	**.93**	**.90**
	NoRBERT	(large, ep.=10)	**.92**	**.88**	**.90**	**.91**	**.99**	**.95**	**.92**	.93	.92	.82	.87	.85
10-fold	K. & M. reimpl.	(100 best features)	.82	.74	.77	.82	.91	.86	.82	.67	.73	.79	.81	.80
	K. & M. reimpl.	(500 best features)	.76	.68	.71	.79	.87	.82	.77	.63	.68	.74	.80	.77
	NoRBERT	(base, ep.=10)	.87	.84	.86	.92	**.97**	**.94**	**.92**	.89	**.91**	.85	**.85**	.85
	NoRBERT	(base, ep.=10, ES)	**.89**	**.86**	**.88**	.91	.96	**.94**	.91	.86	.88	**.87**	**.85**	**.86**
	NoRBERT	(large, ep.=10)	.86	**.86**	.86	**.93**	.95	**.94**	.90	**.90**	.90	.83	.75	.79
p-fold	K. & M. reimpl.	(100 best features)	.81	.70	.74	.75	.92	.82	.82	.52	.62	.75	.80	.77
	K. & M. reimpl.	(500 best features)	.75	.60	.66	.71	.88	.78	.75	.48	.57	.68	.79	.73
	NoRBERT	(base, ep.=10)	.86	.76	.81	.81	**.95**	.87	.85	.60	.70	**.79**	.86	.82
	NoRBERT	(large, ep.=10)	.86	**.79**	**.82**	**.87**	.94	**.90**	**.89**	**.77**	**.82**	.76	**.89**	.82
	NoRBERT	(large, ep.=10, ES)	**.87**	.77	**.82**	.84	**.95**	.89	.86	.72	.78	.78	.88	**.83**
loPo	NoRBERT	(base, ep.=10)	**.87**	**.76**	**.81**	.79	**.94**	.86	**.88**	.63	.74	.79	**.86**	.82
	NoRBERT	(base, ep.=10, ES)	**.87**	.73	.79	.83	**.94**	.88	**.88**	.72	.79	**.80**	.85	**.83**
	NoRBERT	(large, ep.=10)	.84	.75	.80	**.86**	.93	**.89**	.87	**.75**	**.81**	.79	**.86**	.82

For task 2, the binary and multi-class classification of the four most frequent NFR subclasses, NoRBERT also outperforms the comparative approach by Kurtanovic and Maleej [149] by more than three percentage points with a weighted average F1-score of up to 83% for binary classification and 87% for multi-class classification in the ten-fold cross-validation.

For the performance of classifying the subclasses of non-functional requirements (task 3), Table 4.5 shows the results for NoRBERT and compares the results to the approaches by Abad et al. [1] for the repeated five-fold cross-validation. The multi-class classifier performs quite well with outliers in those classes with small representation. On average, the multi-class classifiers also outperform the binary classifiers.

For the fourth task, the classification of functional and non-functional aspects on the relabeled NFR dataset by Dalpiaz et al. [46], we additionally apply a 75%-split. This split is used by Dalpiaz et al. and is a single, stratified split that divides the dataset into 75% train and 25% test split. We use it to compare our results with their approaches. Table 4.6 shows the results of the binary classifiers trained on the relabeled set for the fourth task. NoRBERT outperforms the other approaches in all settings. For example in the ten-fold cross-validation, NoRBERT outperforms the best competing model by ten percentage points on average.

In all four tasks, the approach showed that it can perform well on unseen projects and handle these cases better than the other approaches.

Overall, this pre-study gives good insights into the capabilities and limits of LLM approaches, especially for a BERT-based approach. While less training data is required, imbalanced training data or too few training samples still affect classification negatively.

4.4.2. Approach for Classifying Architectural Design Decision

We create classifiers based on the pre-study and use established supervised machine-learning approaches to explore the classification of ADDs according to the created taxonomy (cf. Chapter 4). We use the labeled sentences from the taxonomy-building process as the dataset for training. The dataset consists of 1622 lines from the 17 projects, with the distribution displayed in Table 4.2.

Table 4.7.: Results for the three ADD classification tasks: binary classification (Task 1), multi-class classification (Task 2), and multi-class multi-label (Task 3) classification. By Keim et al. [125]

	$LR_{trigram}$			DT_{BoW}			RF_{bigram}			BERT		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
Task 1	0.88	0.89	0.89	0.85	0.85	0.85	0.83	**0.97**	0.90	**0.90**	0.94	**0.92**
Task 2	0.45	0.45	0.43	0.31	0.32	0.30	0.35	0.35	0.27	**0.58**	**0.56**	**0.55**
Task 3	0.58	0.33	0.40	0.45	0.39	0.41	0.48	0.09	0.15	**0.68**	**0.43**	**0.50**

There are two kinds of classifiers. First, those that are based on classical machine-learning approaches. Second, there is a classifier that uses the LLM BERT in the same way as NoRBERT.

We experimented with different preprocessing steps, including stop word removal, lemmatization, transformation to lowercase, and combinations of these. Only lemmatization and transformation to lowercase positively affect the results. Further, we tried bag-of-words (BoW), TFIDF, and bi- or trigrams as vectorization techniques. Using the resulting vectors, we apply Logistic Regression (LR), Decision Trees (DTs), and Random Forests (RFs) as our classical machine-learning techniques, as these are commonly used approaches from literature (e.g., [20, 158]).

For the BERT-based classifier, we replicate the setup from the requirements classification task in the preliminary study (see Section 4.4.1). As such, we fine-tune a classifier using the BERT-base-uncased model with 16 epochs, a batch size of 8, and lowercasing as preprocessing.

To answer **RQ 2**, I evaluate the approaches using a random 5-fold cross-validation with three repetitions. The evaluation looks at three different classification tasks. The first task is binary classification, which classifies if a line contains a design decision. The second task is the multi-class classification of the kind of the most prevalent design decision. This means a classifier needs to label the input sentence with one of the leaf classes of the taxonomy. The third task is a multi-class multi-label classification of the various kinds of design decisions. This task represents the need to identify multiple decisions in one sentence as multiple decisions might be contained (see also Table 4.2).

For the second and third tasks, I report the weighted F_1-score. The weighted score allows for the dataset's imbalance to be considered.

Table 4.7 shows the results of the best-performing variants of the classical machine-learning approaches and the BERT approach.

The BERT approach performs best for all three tasks, according to the F_1-score. In the first task, the BERT classifier achieves an F_1-score of 92.1%. While the task is slightly different from existing work, the results are comparable to or better than the results by Bhat et al. [20] for identifying design decisions in issue trackers and by Li et al. [158].

The BERT classifier also yields good results for the second task, the multi-class classification. With an F_1-score of 55.2%, the BERT classifier outperforms the classical approaches by over 12.5 percentage points. These results are particularly promising given the number of classes and the comparably low size of the dataset.

In the multi-class multi-label classification task, multiple BERT classifiers are trained in a One-vs-Rest fashion: A binary classifier is trained for each label to determine if this label is present. This way, there can be multiple classifiers with positive predictions, realizing the multi-label aspect.

The performance for this more difficult setting decreases slightly to an average F_1-score of 50% with 68% precision and 43% recall. Despite the performance drop, the results are close to the results of the second task.

These classifiers directly label the input based on the leaf classes of the taxonomy that I call *flat classification* (cf. Figure 4.3 and Figure 4.4). Another approach is to label the input hierarchically. This means that labeling is a process that walks from the root of the taxonomy through the tree to the leaves. On each level, a classifier decides which path to take by labeling the input accordingly. For example, a design decision about a component is first labeled that it is a *design decision*, then that it is an *existence* decision, then a *structural* decision, then an *intra-systemic* one, and finally that it is a *component* decision. The idea is to have many specific classifiers that can focus on a more narrow problem than distinguishing all leaf classes. For this hierarchical classification, a multi-class classifier is trained for each branch in the taxonomy, with a total of 13 classifiers. This idea was realized by Janek Speit in his master's thesis [250] that I supervised. For the evaluation, we trained multiple supervised classifiers, classical ML approaches, and LLM-based ones. The classical ML approaches include LR, NB, SVM, and Random Forest (RF). The LLM-based approaches include RoBERTa [161], XLNet [295], and BertOverflow [260]. They are trained using fine-tuning in the same

Table 4.8.: Results of the five-fold cross-validation of the flat and hierarchical classification of ADDs

		Flat			Hierarchical		
	Approach	P	R	F_1	P	R	F_1
Classic	Logistic Regression (LR)	**0.31**	0.24	**0.27**	**0.27**	0.24	0.25
	Naïve Bayes (NB)	0.25	**0.25**	0.25	**0.27**	**0.29**	**0.28**
	SVM	0.25	0.23	0.24	0.25	0.24	0.25
	Random Forest (RF)	0.27	0.20	0.23	0.25	0.23	0.24
LLM	RoBERTa	**0.54**	**0.53**	**0.53**	0.43	**0.44**	**0.44**
	XLNet	0.46	0.44	0.44	**0.46**	0.42	**0.44**
	BERTOverflow	0.43	0.44	0.40	0.33	0.34	0.34
	GPT-3	0.52	0.46	0.49	0.41	**0.44**	0.42

fashion as the BERT-based classifier before. Additionally, there is a GPT-3 [28] classifier that is trained using prompt-completion and the fine-tuning API of OpenAI. In the evaluation, we compared the flat classification (i.e., directly classifying the leaf classes) and the hierarchical classification.

The evaluation showed mixed results, as shown in Table 4.8. Overall, there are improvements in some cases, for some classes, and some classification approaches. For others, the results are deteriorating. The experiments with classical ML approaches showed that the performance was overall equal between hierarchical and flat classification. Improvements are mostly not significant. The results range between 23% and 28% F_1-score for the flat classification with roughly equal precision and recall. For the hierarchical classification, the results range from 24% to 28% F_1-score.

The LLM-based classifiers perform better compared to the classical ML approaches. However, hierarchical classification often diminishes the performance. For RoBERTa, the flat classification with 53% F_1-score outperforms the hierarchical with 44% F_1-score. The same applies to BERTOverflow (40% vs 34% F_1-score). XLNet performs equally in both cases with an F_1-score of 44%. GPT-3 yields an F_1-score of 42% in the hierarchical variant and 49% in the flat one. In all cases, precision and recall are roughly equivalent.

Looking into some details, the biggest issue with the hierarchical variant is the probability of taking the wrong branch in one of the early classification levels. If a wrong branch is taken, there is no possibility of recovery. On each level, there is a chance for misclassification. Moreover, the classifiers perform better in more specific classes and underperform, particularly in the higher-level classes. These misclassifications on the higher levels create too many issues compared to the increased performance on the lower levels. In this case and setup, the benefits cannot outweigh the issues.

4.4.3. Discussion and Conclusion

Overall, the approaches for classifying ADDs are promising based on these exploratory studies. Despite the small dataset, the results are good, especially for identifying ADDs.

The results for identifying the type of ADD are also promising but need significant improvement. These promising results are insufficient for automated applications. The goal is to use the classification results for targeted analyses to verify the consistency of ADDs. However, the results are insufficient to reliably use automated classification for this task. A recall of 43% means that more than half of the design decisions are not detected, leaving less than half available for consistency checks. Additionally, a precision of 68% indicates that roughly a third of the detected design decisions are incorrect, potentially leading to erroneous analyses.

To enhance the approach, researchers can focus on improving either precision or recall using various methods. For instance, using multiple classifiers can improve precision by adding labels only for classes that most classifiers agree on, i.e., using the intersection. Alternatively, recall can be increased by using the union of all classifiers' outputs and adding all returned labels. This typically improves one metric at the expense of the other. A critical decision involves choosing between two goals:

1. Focusing on correct labeling (high precision) ensures subsequent analyses receive accurate labels and select appropriate analyses, but risks missing some labels and potential inconsistencies.
2. Focusing on comprehensive labeling (high recall) ensures no inconsistencies are missed but introduces the risk of incorrect labels and false positives, complicating inconsistency detection.

Multiple factors will likely improve performance in the future. More training data will help distinguish classes, particularly those with low representation in the current dataset. Additionally, larger and more advanced LLMs are expected to enhance performance on this task, as evidenced by the apparent scaling law identified by Kaplan et al. [120]. New methods could also improve results as emerging methods may enable LLM-based approaches to better capture underlying intentions, enhancing both precision and recall.

5. Recovering Trace Links in Software Architecture Documentation

Trace links can connect architectural elements in different architecturally relevant artifacts. These links are essential for understanding how and why specific architectural choices were made. They also play a critical role in maintaining consistency, ensuring architecture conformance, and facilitating system evolution. This chapter focuses on approaches for recovering for Traceability Link Recovery (TLR) in software architecture documentation, looking into **RQ 3**. In this dissertation, I look at three types of artifacts that are shown in Figure 5.1: informal Software Architecture Documentations (SADs), formal Software Architecture Models (SAMs), and code. There is a semantic gap between the artifacts that an approach for TLR needs to bridge.

The biggest semantic gap between the mentioned artifacts is between SADs and code. The direct relation between both kinds of artifacts is not strictly defined, but the abstraction level of SADs is usually higher. The main elements

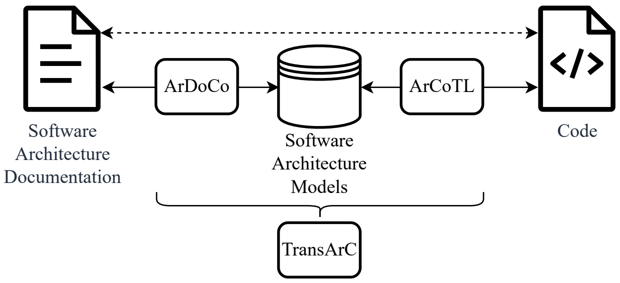

ArDoCo
ArCoTL
Software Architecture Documentation
Software Architecture Models
Code
TransArC

Figure 5.1.: Different kinds of artifacts with different approaches (in boxes with round corners) to recover trace links.

of code are classes and methods and SADs usually cluster the underlying ideas and responsibilities in components. Although components from SADs can be realized with single classes, they are regularly implemented with multiple classes that can be structurally organized in packages, modules, or similar. Moreover, code focuses on the behavior and execution semantics while SADs contain design decisions that can cover different parts of the code and behavior but also about, e.g., structure, guidelines, and organization (cf. Chapter 4).

SAMs are semantically in-between SADs and code. SAMs, particularly the structural component-based SAMs that I focus on in my work cover the structure similarly to code. At the same time, SAMs have a higher abstraction level and look primarily at components and interfaces just like SADs.

Linking artifacts like SADs and SAMs requires their existence. According to surveys with practitioners, a considerable fraction of practitioners already have SAMs: 86% of practitioners stated in the study by Malavolata et al. [172] that they use UML as an architecture description language. In the study by Tian et al., [268], 26.3% of the participating practitioners stated that they model or visualize architecture with UML. In addition to SAMs being already used, some approaches reduce the overhead to create SAMs by (semi-) automatically recovering them from code and deployment artifacts (cf. [62, 31, 154, 240, 132]).

SADs are prevalent in software projects, but their presence, quantity, and quality can vary significantly. While research has shown that open-source documentation positively impacts the adoption of Open Source Software (OSS) and their economics [3], only a few open-source projects have some documentation, as Ding et al. showed [52]. Their research also demonstrates that projects with more developers are more likely to have documentation, and industry and research OSS projects also have a higher likelihood of having SAD. This is likely attributed to increased costs for maintenance and evolution when documentation is missing or outdated [208]. Additionally, there are frameworks like the arc42 template [1] and the C4 model [2] that provide structured and efficient approaches for SADs.

[1] `https://arc42.org/`

[2] `https://c4model.com`

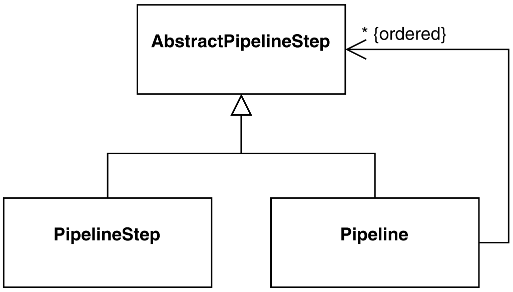


Figure 5.2.: The General Pipeline Definition

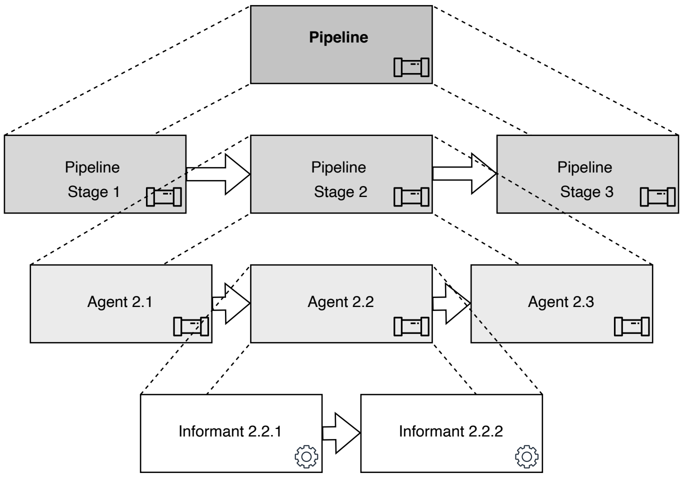


Figure 5.3.: The multi-level pipeline where a pipeline consists of different subsequent stages that themselves consist of various agents. Further, agents use several informants, which are the actual processing units.

This chapter presents my approaches to recover trace links between SADs, SAMs, and code.

The approaches all share the same underlying architecture that combines the *Chain of Responsibility* pattern and the *Pipeline* pattern. The general idea is to use a pipeline where each pipeline step uses various heuristics to process the data and extract necessary information. The pipeline definition, visible in Figure 5.2, uses the composite pattern to reflect the requirements. Each `Pipeline` contains an ordered list of `AbstractPipelineSteps` that each can be either another `Pipeline` or a concrete `PipelineStep` that performs calculations. This way, the approach can define pipelines on multiple levels, as depicted in Figure 5.3: The approach defines the pipeline at the highest level with its various processing steps, the so-called *stages*. Each stage is another pipeline comprising various *Agents*. An *Agent* serves the purpose of initiating the collection of specific information. *Agents* themselves are again pipelines that have a number of *Informants*. An *Informant* is an actual processing unit that, e.g., realizes a heuristic or performs calculations to extract certain information. A pipeline step (i.e., an *Informant*) stores results within a repository that can be universally accessed by all pipeline steps, similarly to a blackboard in the blackboard pattern. This way, each pipeline step and, thus, each heuristic can access the results of previous steps and provide its results for others.

One advantage of such a pipeline-based approach is the capability to use RESTful microservices for the actual processing `PipelineSteps`. In this case, microservices bring the benefit of outsourcing processing to dedicated processing servers that offer more computing power and might even be necessary to fulfill the hardware requirements, e.g., for certain NLP models or LLMs. Models can also be held in memory and do not need to be loaded each time, further reducing the processing time. Additionally, microservices allow easy mixing of programming languages, e.g., to add an approach or heuristic implemented in Python to an implementation of the framework in Java. One concrete example is a microservice implementation of the *Text Preprocessing* stage for TLR between SAD and SAM (cf. Section 5.2.3).

A shared concept between the different approaches is the requirement to load and process SAMs and code. There are different modeling languages, e.g., UML and PCM, to represent SAMs and different programming languages for code. The inputs are first transformed into intermediate representations, which treat them uniformly and independently of programming or modeling language within the approaches. Section 5.1 introduces the intermediate representations for software architecture models and code.

ArDoCo is tailored to recover trace links between informal, textual, natural language SADs and formal SAMs. The approach uses a pipeline and several heuristics to identify parts in the natural language text that are potential elements of the SAM, e.g., components. These potential elements are then compared to actual model elements, and if the similarity is high enough, a trace link is created. The use of NLP and IR techniques is well-known and many approaches use general techniques like VSM to recover trace links. The novelty of the approach is the specialization towards the artifacts to be able to create heuristics to first identify the relevant parts. Further, the approach touches uncharted territory as SADs and SAMs have not yet been tackled for TLR (see also Section 3.2). Therefore, another difficulty of the approach is the required analysis the problem domain to come up with reasonable heuristics. I present the details of the approach in Section 5.2.

The second approach, ArCoTL (ARchitecture-to-COde Trace Linking), is for TLR between SAMs and code. ArCoTL uses a computational graph that combines various heuristics to assess the similarity of artifacts and create trace links. The goal and challenge of the approach is to mimic how a human would analyze the artifacts to recover trace links between these artifacts. I explain the approach in Section 5.3.

The third approach TransArC (Transitive links for Architecture and Code) that I present in Section 5.4 combines the previous two approaches to tackle TLR between SADs and code (cf. Figure 5.1). The main idea is to reduce the semantic gap by dividing the problem into two smaller ones in a divide-and-conquer fashion. While other approaches have used intermediate artifacts for TLR (see Section 3.2.3), these approaches did not look into using different, specialized approaches to tackle different tasks involving the intermediate artifact. TransArC can use the two specialized approaches, ArDoCo for TLR between SAD and SAM as well as ArCoTL for TLR between SAM and code, and combines their results transitively.

The biggest challenge of these approaches is finding heuristics that are general enough to be applied to different projects that all can have slightly different abstraction levels and documentation styles. At the same time, the heuristics need to be specific enough to precisely recover the trace links. This requires cautious assessments of the trade-offs. Due to the different abstraction levels and styles, this is more than simply weighing precision and recall, as adding a heuristic to increase the recall for one project might negatively influence the overall performance for all other projects.

I assess the performance of these approaches in an evaluation in Section 5.5 to answer the research questions regarding TLR between SAD and SAM (**RQ 3.1**), TLR between SAM and code (**RQ 3.2**), and transitive TLR between

SAD and code (**RQ 3.3**). In this section, I also discuss the evaluation goals and the setup, including the used benchmark dataset in Section 5.5.2.

The advancements in LLMs, such as GPT-4, present a promising prospect for addressing several challenges in software engineering, including TLR. Leveraging the capabilities of LLMs in these areas can significantly enhance the efficiency of these tasks and the accuracy of the results. Specifically, the ability of LLMs to capture semantic relationships and contexts is beneficial for these tasks. The capabilities of LLMs to seemingly understand and effectively process text are useful in my application areas. For example, an LLM could better distinguish relevant and irrelevant parts of the documentation or better assess the necessary context for, e.g., disambiguation compared to current approaches. As such, future work should focus on experimenting with LLMs to use and integrate them for these problems. In exploratory studies, I initially assess the capabilities of LLMs. For this, I briefly present simple end-to-end approaches for TLR between SADs and SAMs in Section 5.5.6 and evaluate these, answering **RQ 3.4**.

5.1. Intermediate Representations of Artifacts

The approaches use intermediate representations, which I will introduce in this section. The idea of the intermediate representations is to be independent of the input programming or modeling languages. There are different modeling languages like UML and PCM and different programming languages for code. Handling these languages directly requires specialized adaptations to access the contained information. To partly avoid this, the approach transforms the various artifacts into an intermediate representation, allowing it to easily extract the required information uniformly.

Adapters can transform architecture model languages like UML component models or PCM repositories and programming languages into the intermediate representation. For each different language, there needs to be a specific transforming adapter. This requirement creates an initial one-time overhead to create the adapter. However, the intermediate models create high flexibility and extensibility for the approach.

Overall, the models are simple and contain only as much information as necessary for our purpose, i.e., TLR. The models are only intended to extract

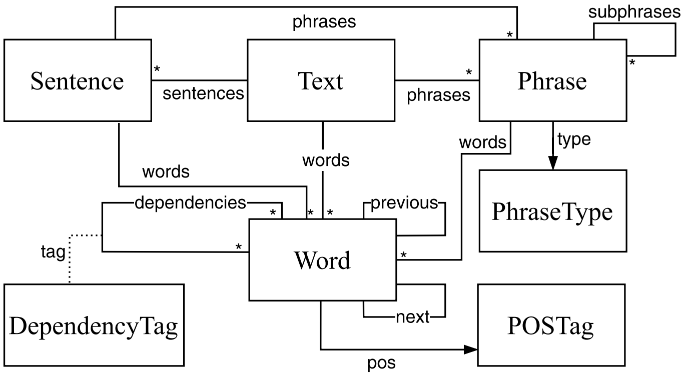


Figure 5.4.: Internal Representation of Text

parts of the information present in the languages and are a view of them. As the name states, their purpose is to represent artifacts. The transformations are not bidirectional, as there is no intention to reconstruct or update the original models from the intermediate representations alone. Still, in theory, there are bidirectional transformations imaginable to, e.g., update the original model(s) after something changes values in the intermediate model. However, the transformations are unidirectional, and intermediate models are so far read-only.

5.1.1. Internal Representation of Text

Apart from the more formal input artifacts like SAMs and code, we also need to represent the text of the input SADs. Figure 5.4 shows the used representation model of the text from the SADs.

A `Text` contains an arbitrary number of `Sentences`, each with their sentence number as their identifier. The `Text` also contains references to all contained `Words` and `Phrases`. In the same manner, `Sentences` contain references to all their contained `Words` and `Phrases`.

`Phrases` again contain references to all contained `Words` and additionally all sub-`Phrases`. A `Phrase` also has a distinct `PhraseType`.

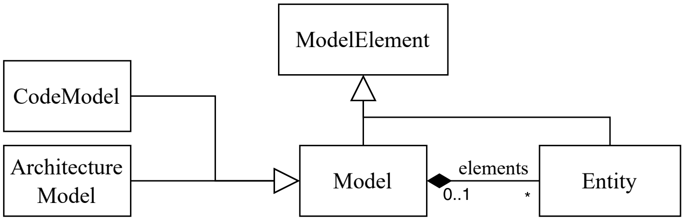


Figure 5.5.: General Model for the Intermediate Representations of Formal Artifacts

`Words` store their position within the sentence and their `POSTag`. Further, each `Word` contains references to the succeeding and preceding `Word`. Additionally, incoming and outgoing dependency relations to other words are stored as typed relations using `DependencyTags`.

5.1.2. Intermediate Representations of Formal Artifacts

The models for the formal input artifact types such as SAMs and code are based on the *Knowledge Discovery Metamodel (KDM)* [198]. The KDM is a metamodel for describing software artifacts. It represents the physical and logical elements of software, including their different levels of abstraction. KDM is powerful, and the representation is targeted to allow for many different kinds of analyses of existing systems. This generality also makes it very detailed, too sophisticated, and complex for our use case. Some parts of the KDM describe details that the approaches do not require at all. Some parts of the KDM even describe things that the used artifacts do not provide, like the *Data package* that represents persistent data and data storage. Accordingly, the intermediate representation only selects those parts of the KDM that are required for the purpose, i.e., the approaches and simplifies the model.

The intermediate representations are specialized for the corresponding kind of artifact. Representing SAMs is different from representing code. Still, there are certain parts that these formal artifacts share. These shared parts of the intermediate representations are depicted in Figure 5.5.

The `ModelElement`, based on the identically named class in the KDM, is the overarching class that defines that everything needs to have an identifier. The models themselves are represented by the class `Model` that is based on

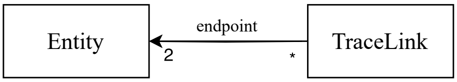


Figure 5.6.: Trace Link Model

the `KDMModel` class. In our case, we have two differentiations of the `Model` in the form of `CodeModel` and `ArchitectureModel`.

The class `Entity` is central to the intermediate representations. The `Entity` is based on the `KDMEntity` in KDM. It represents an artifact with a name and the inherited identifier from the `ModelElement`. With the `Entity` class as a shared class between the intermediate representations, we can define a trace link as depicted in Figure 5.6. A `Trace Link` consists of exactly two references to entities.

In this model, and this dissertation, there is no focus on relations as first-class entities. If the formal artifacts do not have relations as first-class entities, they are not modeled explicitly as entities. In theory, it is possible to extend the model and represent relations explicitly as (special) entities. These then can have, for example, relations to other non-relation entities. This then can enable more analyses that are targeted towards relations. However, as this dissertation does not focus on linking relations, the model does not explicitly model relations this way.

The shared model for formal models is then extended for the two kinds of formal artifacts that this dissertation covers: SAMs and code. The specializations for the two kinds of artifacts focus on defining the different kinds of contained entities. The following sections describe these specializations.

5.1.2.1. Software Architecture Models

The specializations for the intermediate representations of SAMs like PCM or UML are depicted in Figure 5.7.

Every class in the model is an `ArchitectureItem` that itself inherits from `Entity`, thus having a name and identifier. There are three kinds of `ArchitectureItems`, namely `Component`, `Interface`, and `Signature`.

As the name suggests, a `Component` represents components in the various architecture model languages. For example, an UML `Component` is represented

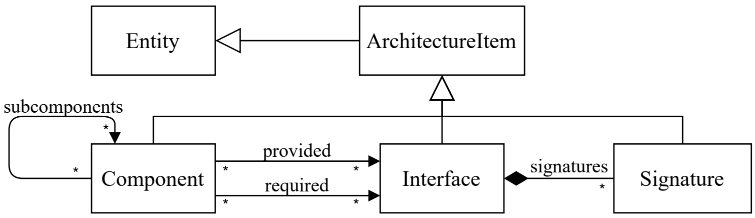


Figure 5.7.: Intermediate Representation for Architecture Models [123]

by this class. For PCM, this class represents both, a `BasicComponent` and a `CompositeComponent`. For `BasicComponents`, the `Component` simply has no sub-components. Sub-components are only set for `CompositeComponents`.

A `Component` can require or provide an `Interface`. Provided `Interfaces` are realized by the `Component`. Required `Interfaces` define the functionality that a `Component` needs.

An `Interface` contains arbitrarily many `Signatures` of methods. Due to the composite relation, a `Signature` can only exist in combination with an `Interface`.

5.1.2.2. Code

The intermediate model for code in Figure 5.8 is based on the source code package within the KDM [198].

The different classes in the code model inherit from `CodeItem` that itself is a specialized `Entity` and, thus, has a name and identifier. There are three kinds of source code elements: `Module`, `Datatype`, and `ComputationalObject`.

`Modules` are usually some kind of logical component of the system with a certain level of abstraction. A `Module` can contain `CodeItems` and there are three differentiations of `Modules`: `CompilationUnit`, `Package`, and `CodeAssembly`.

A `CompilationUnit` is a source file into which code is put. As such, a `CompilationUnit` contains a relative path to where the file is located on disk and its programming language. The `CompilationUnit` is partly based on the `InventoryModel` from KDM.

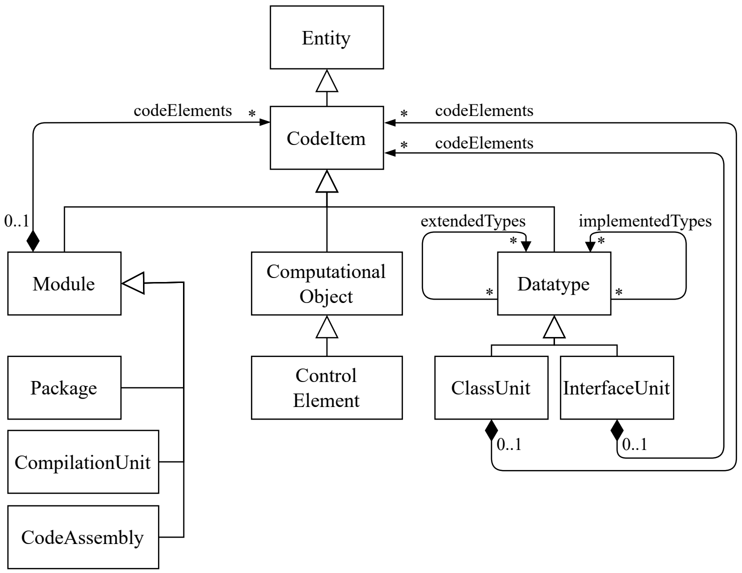


Figure 5.8.: Intermediate Representation for Code [123]

A `Package` is a logical collection of source code elements (i.e., `CodeItems`). `Packages` can also contain sub-`Packages` like it is common in Java.

A `CodeAssembly` contains source code artifacts that are linked to each other to make them runnable. For example, source code files and their headers are put together in a `CodeAssembly`.

There are two kinds `Datatypes`, namely `CodeUnit` and `InterfaceUnit`. A `CodeUnit` is something like a class in Java. It can contain other `CodeItems` like methods and inner classes. Similarly, an `InterfaceUnit` can also contain code elements like methods. The `implements` and `extends` relations from the KDM are present in the intermediate model as the relations `extendedTypes` and `implementedTypes`. A `Datatype` can implement an arbitrary number of `extendedTypes` relations to represent inheritance in object-oriented programming languages. In Java, only one such relation is possible, but for languages like C++ that support multiple inheritance, the model needs to support this. The construction around `extendedTypes` and `implementedTypes` also allows interfaces to extend other interfaces like it is common in Java. At the same time, interfaces can also extend classes which is possible in some programming languages like TypeScript. The KDM also contains several primitive datatypes like boolean. These are not realized within this model as they are not needed by the approaches yet. If the approaches are extended in future work with a thorough comparison of datatypes, then we need to extend the intermediate model with further sub-classes of the KDM.

There is currently only one kind of `ComputationalObject`, the `ControlElement`. The `ControlElement` represents callable parts with a given behavior like functions, procedures, or methods. In contrast to the KDM, there is no further distinction between `CallableUnits` and `MethodUnits`. In KDM, the difference of a `MethodUnit` is its containment within a class. This distinction is not used and, thus, unnecessary in this work. Further, this work does not use parameters, return types, or similar elements of the KDM; hence, they are not modeled.

5.2. Recovering Links Between Documentation And Model

This section will present the approach *ArDoCo* for TLR between SADs and SAMs. This section is mainly based on my publications at ECSA in 2021 [129] and at the IEEE International Conference on Software Architecture (ICSA) in 2023 [124].

To develop ArDoCo, I used the information from the taxonomy presented in Chapter 4. In particular, I used the project TEAMMATES[3]. Additionally, I looked at MediaStore[4], an example project for the Palladio approach [231]. For these projects, there were SADs and SAMs that I partly analyzed to derive the approaches. These are also part of the projects used in the evaluation (see Section 5.5.2). While I tried to generalize from these projects and tried to avoid tuning and overfitting on these projects, there are threats to validity. In Section 5.5.7, I discuss the threats to validity that the use of these projects for developing the approaches brings.

[3] `http://github.com/TEAMMATES`

[4] `http://sdq.kastel.kit.edu/wiki/Media_Store`

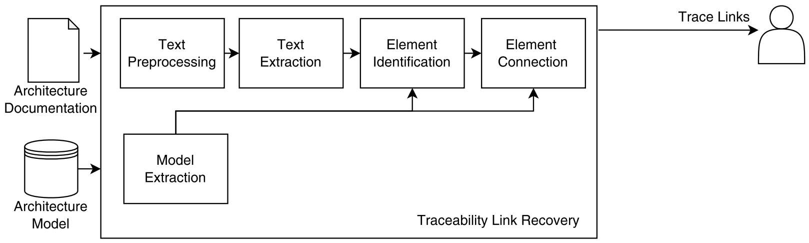


Figure 5.9.: Overview of the ArDoCo approach for TLR [129, 124] between SAD and SAM

The approach uses textual, natural language SADs as an input artifact. As for the other kind of input artifact, SAMs, the approach expects the development view of the architecture (cf. Kruchten's 4+1 view model [145]) as structural component-based software architecture models. Examples of input SAMs are the repository view and/or the system view of PCM [231] and UML component diagrams [44].

Figure 5.9 shows the pipeline with the different processing steps of ArDoCo.

The *Model Extraction* deals with extracting the intermediate representation of SAMs (cf. Section 5.1) and is further detailed in Section 5.2.2. The text is first preprocessed with a common NLP pipeline in the *Text Preprocessing* step that I present in Section 5.2.3. Then, the *Text Extraction* starts to analyze the text to identify relevant parts of the text. I explain the ideas and heuristics in Section 5.2.4. In Section 5.2.5, I go over the next pipeline step, the *Element Identification*, that refines the relevant parts and tries to identify potential architecture elements. These steps are comparable to NER but for architecture elements. While heuristic-based NER is not new, the novelty of this approach lies in the multi-step approach that uses architecture-specific heuristics. This means that each processing step refines the previous results. The biggest challenge is defining specialized heuristics to identify architecturally relevant elements in SADs. The last step in the TLR pipeline is the *Element Connection*, where the results of the previous steps are one final time refined and, lastly, the trace links created. The *Element Connection* is explained in Section 5.2.6.

The overarching philosophy of the approach is to expand promising results in each subsequent step. The approach should explore many possibilities and not discard viable ones too early to maintain a high recall, especially in the early steps. Therefore, the approach keeps options that are less likely early, refines them in each step, and filters out very unlikely ones in the end.

The approach needs to assess the similarity of words or phrases in various steps. The similarity assessment is detailed in Section 5.2.1. Lastly, I discuss the limitations of this approach in Section 5.2.7.

5.2.1. Similarity

Similarity is used on different occasions, for example, to cluster words or determine if a text element and a model element should be connected with a trace link. This similarity is based on assessing the similarity of single words, compound nouns, and phrases. As such, the similarity component is central to the approach.

The similarity component supports different similarity measures that can be enabled on preference like (normalized) Levenshtein similarity [156], Jaro-Winkler similarity [289], n-gram-based similarity, similarity based on the software-specific word similarity database SEWordSim [269], and vector-based similarity that uses vectors created from word embedding approaches, e.g., word2vec [184] together with cosine similarity.

The result values for each similarity measure must be interpreted individually as the meaning can be slightly different, and a shared threshold is too inflexible. Consequently, the user can set a specific threshold for each similarity measure that influences when the measure considers two inputs similar. Therefore, the similarity measure returns only a boolean that delivers an interpreted value indicating whether two inputs are similar.

Moreover, the similarity component supports the usage of multiple similarity measures. This has the benefit of combining the advantages of multiple approaches or weakening the disadvantages of a single approach. To combine the results of multiple similarity measures, there are two modes: *at-least-one* and *majority*. *At-least-one* returns a positive result (i.e., that the input is similar) if one of the applied similarity measures returns a positive result. *Majority* returns positive results if most of the applied similarity measures return positive results.

The binary return value simplifies the handling but limits the information that might be used by heuristics. For example, there is no difference between a similarity value being closely and convincingly above the threshold. Similarly, there is no information on how much a similarity value is below the threshold. Future work can explore further capabilities to better use and combine the values apart from the binary result.

The similarity component can assess the similarity of single words and also supports comparing lists of words, e.g., when looking at phrases or compound words. To do so, the parts of the lists are compared with the method described above. During this comparison, the order within the lists is not regarded. This is done to ensure that, for example, a slight reordering of words still returns high similarity. This way, the phrases “the database component” and “the component database” are regarded as equal.

In this approach, if not specified otherwise, the default method for assessing similarity is a combination of the normalized Levenshtein similarity [156, 33] and the Jaro-Winkler similarity [289]. Each metric must have a high similarity value (at least 0.9) to interpret the input as similar. We use the high similarity value to increase precision, and the value is empirically determined. The *at-least-one* strategy is applied to combine these metrics.

5.2.2. Model Extraction

The *Model Extraction* step consists of a component that transforms the input SAMs into the intermediate representation to have a simple representation of the input UML models [44] and PCM models [231]. As such, the model extraction step implements adapters to transform UML and PCM models into the intermediate architecture model that contains all the required information (cf. Section 5.1).

For UML, the adapter directly maps the corresponding concepts from UML into the intermediate artifact. The names and their relations are extracted for the different entities, like interfaces and components. The identifiers are also extracted from the underlying model so that they can later be used to precisely point to the corresponding model elements.

For PCM, names and identifiers are equally extracted. `BasicComponents` are mapped to the `Component` class in the intermediate representation without adding any `subcomponents` relations. `CompositeComponents` are also mapped to the `Component` class and the contained components are related with the `subcomponents` relation. `Interfaces` of PCM are mapped to the corresponding `Interface` class in the intermediate representation. Similarly, relations like `provided` or `required` relations between components and interfaces are mapped to their corresponding, usually equally named, parts.

5.2.3. Text Preprocessing

The first step within this pipeline step is loading and initially processing the text with common NLP processing. The approach uses Stanford CoreNLP [173] to process the following NLP tasks: tokenization, sentence splitting, part-of-speech tagging, syntax tree parsing, dependency tree parsing, and lemmatization (cf. Section 2.3.2). After the NLP task pipeline, the results are transformed into the internal representation for text (cf. Section 5.1.1). This way, the results of these preprocessing steps can be used in the various subsequent pipeline steps.

Loading the necessary models for this pipeline can be time-consuming. As such, having a second implementation of this *Text Preprocessing* stage as a (micro-) service is advised. Consequently, the user can either run the processing fully locally or transfer some processing to the (micro-) service "into the cloud".

5.2.4. Text Extraction

The text extraction step is the first step when processing the preprocessed text. The main idea of this step is to look at the text and identify important, i.e., architecturally relevant, parts for TLR and ID without any kind of influence or bias from the SAM.

In this step, the approach identifies nouns as important parts, so-called *mentions*, and attaches probabilities to these nouns based on the likelihood that the nouns are *names* or *types* of architecturally relevant entities. Some used heuristics like the *Nouns* heuristic create new *mentions*, others like both *Dependencies* heuristics refine existing *mentions*. It is important to note that these are only heuristics and only give indications but are not strict rules that are always correct. These heuristics are developed using observations after analyzing SADs (see also the taxonomy in Chapter 4).

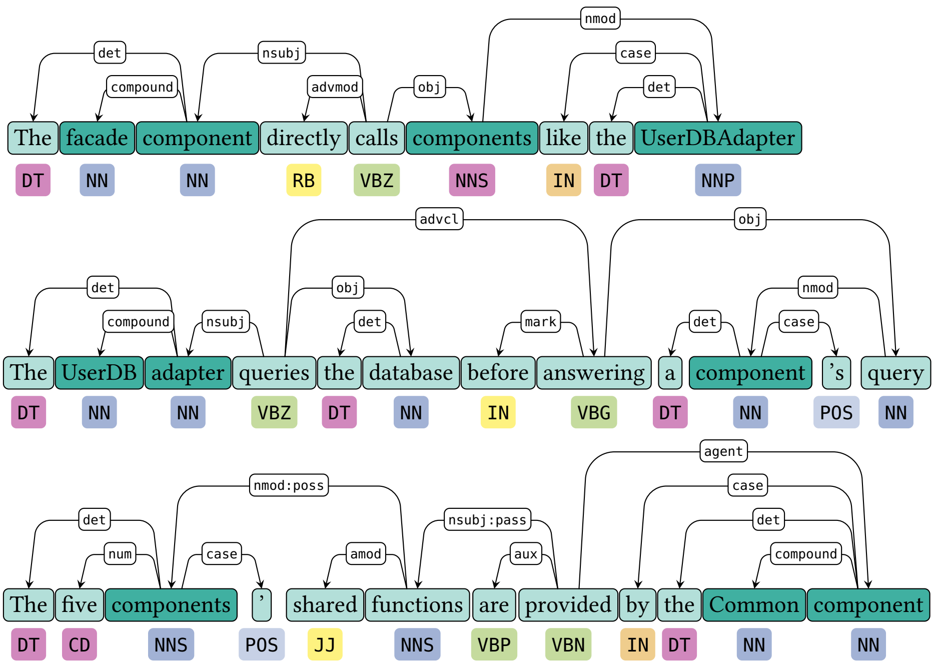

The facade component directly calls components like the UserDBAdapter
DT NN NN RB VBZ NNS IN DT NNP
det compound nsubj advmod obj nmod case det
The UserDB adapter queries the database before answering a component 's query
DT NN NN VBZ DT NN IN VBG DT NN POS NN
advcl det compound nsubj obj det mark obj det nmod case
The five components ' shared functions are provided by the Common component
DT CD NNS POS JJ NNS VBP VBN IN DT NN NN
nmod:poss det num case amod nsubj:pass aux agent case det compound

Figure 5.10.: Example sentence annotated with syntactic dependencies to be analyzed with heuristics. Names and types are highlighted with darker colors.

When creating *mentions*, the approach first identifies if a similar *mention* already exists using similarity metrics introduced in Section 5.2.1. If a highly similar *mention* already exists, the *mentions* are combined. As a result, *mentions* are effectively clusters of similar parts in the text that should be treated the same, just like coreferences (see also Section 2.3.2.8).

Nouns The *Nouns* heuristic extracts and classifies all nouns in the sentences. The approach focuses on nouns because architecturally relevant entities, either as *name* or *type*, are most certainly always nouns. If a noun occurs in plural, the heuristic makes it more likely that it is a type. This is based on observations that component names mostly have a singular word as a *name*, while a *type* can regularly appear in plural. For example, in the phrase “the components DatabaseAdapter and UserManagement”, the “components” clearly refer to the *type*, while the other two nouns each seem to be a *name*. Additionally, this heuristic filters the project name, as it is unlikely to be an actual architecturally relevant entity.

Incoming Dependencies For each identified *mention*, this heuristic examines the incoming dependencies in the dependency graph obtained by the dependency parser (see also Section 5.2.3, Section 2.3.2.6). The heuristic assesses the dependencies of each mention to check if the dependencies indicate a *name* or *type* and adjusts the corresponding likelihood accordingly. Example sentences are displayed in Figure 5.10. If there is an incoming dependency of a nominal subject (NSUBJ), possessive (POSS), or appositional (APPOS) modifier, the likelihood for both, *name* and *type*, is increased. These dependencies usually depict key parts of sentences and refer to words that play an important role. For example, a verb's nominal subject (NSUBJ) is regularly an architectural element, exemplarily shown in Figure 5.10. If there is an incoming dependency that indicates it is an object (OBJ, IOBJ, or POBJ), a nominal dependent (NMOD), or a passive nominal subject (NSUBJPASS), the likelihood for *type* is additionally increased. This stems from observations that types are often used in such sentence structures. In the second case, if there is an indirect determiner such as "a" or "an", the likelihood for *type* is further increased because named components are never indirectly referred to.

Outgoing Dependencies Similarly, as for *incoming dependencies*, the *outgoing dependencies* heuristic checks for outgoing dependencies obtained by the dependency parser (see also Section 5.2.3, Section 2.3.2.6); Figure 5.10 displays these dependencies. If the outgoing dependency is to a numeric (NUM) or a predeterminer (PREDET), the likelihood for *type* increases. Again, named components are usually directly and uniquely specified, making numerics or predeterminers unlikely. If the target is an agent (AGENT) or a relative clause modifier (RCMOD), the likelihood for *name* and the likelihood for *type* are both increased. Both kinds of dependencies mark architecturally relevant entities, but at the same time, do not necessarily differentiate *name* or *type*. Agents, for example, are the actors in passive sentences (see Figure 5.10), and actors are usually architecturally relevant in SADs.

Compound Terms Compound terms are a strong indicator for a *name* and *type* combination or for the *name* of architecturally relevant entities. As such, this heuristic uses various indicators to find compound terms and, if found, increase their likelihood for *name*. First, the heuristic looks at the phrases from the constituency parser and the dependencies, such as COMPOUND from the dependency parser (see also Section 5.2.3, Section 2.3.2.6). Moreover, this heuristic looks at casing such as camel case (e.g., "UserDBAdapter"), snake case (e.g., "user_db_adapter"), or kebab-case (e.g., "user-db-adapter").

Separators Following the same underlying idea of the *Compound Terms* heuristic, this heuristic looks at words that contain separators. In some cases, this heuristic even overlaps with the casing heuristic. This heuristic includes separators that are commonly used in software development like hyphens ("-"), colons (":"), double colons ("::"), and underscores ("_"). Concrete examples include "storage::entity" and "end-to-end". These separators are often used for names of software entities and, thus, are a good indication for a *name*. Consequently, the heuristic increases the likelihood for *name* if a word contains such a separator. Additionally, the heuristic also creates *mentions* and attaches likelihoods for the parts as these separators are also regularly used for embedded paths, package names, and module names. To incorporate cases where these, e.g., paths are not used uniformly, but parts, especially later parts, are identical.

5.2.5. Element Identification

The *Element Identification* step takes the output, namely the *mentions*, from the *Text Extraction* step in Section 5.2.4 and further processes and refines the *mentions*. For refinement, this step also includes information from the meta-model, e.g., about existing types. This step aims to identify potential architecturally relevant entities to create so-called *RecommendedInstances*. *RecommendedInstances* represent architecturally relevant entities, comparable to named entities in NLP (see also Section 2.3.2.9).

Similar to *mentions*, *RecommendedInstances* should also cluster all occurrences of the same underlying entity within the text. When creating a new *RecommendedInstance*, the approach first checks if an existing one with an equal or highly similar name exists. If there already is one, the existing *RecommendedInstance* is extended with the new occurrences. Otherwise, a new *RecommendedInstance* is created.

This pipeline step uses two heuristics: *Name-Type* and *Compound Terms.*

Name-Type This heuristic checks first if the *mentions* from the previous pipeline step have similar naming to the types extracted from the meta-model. If this is the case, the likelihood for *type* is increased accordingly.

After that, the heuristic looks at neighboring *mentions* and performs pattern matching. These patterns include adjacent occurrences of words that are most likely a *name* and a *type*. For example, this will detect the pattern *NAME-TYPE* in "the DatabaseAdapter component" or will detect the pattern *TYPE-NAME* in "the component DatabaseAdapter". If the heuristic detects such a pattern, it creates a *RecommendedInstance* for the occurrence.

The heuristic also resolves uncertainty about the classification of *mentions* into name or type to be then able to create *RecommendedInstances.* In cases where a *mention* has similar likelihoods of being a *name* or a *type*, the heuristic looks for co-occurrences with other *mentions* and patterns like those mentioned. This can help to decide if the *mention* is a *name* or *type* and can result in further *RecommendedInstances.*

Compound Terms This heuristic operates on the results of the *Compound Terms* heuristic of the *Text Extraction* (see also Section 5.2.4) and looks at compound terms. The heuristic creates a *RecommendedInstance* if a *mention* is a compound as previously identified in the *Text Extraction* step. For example, the "UserDB adapter" in the example in Figure 5.10 consists of two names likely connected as compound terms. In this case, both occurrences are combined into one *mention* and then further processed into a *RecommendedInstance.* Another example is the compound "facade component" in the example. There, a *name* and a *type* are combined. While the *NAME-TYPE* usually already identifies such cases, this heuristic can also look for cases where, for example, one of the two parts of the compounds is not correctly identified as *name* or *type.*

5.2.6. Element Connection

The *Element Connection* step has two responsibilities. The first responsibility is the final refinement step of the results of the previous processing steps. This step now also has access to the instances from the architecture model and, thus, has full insight into the model. As such, this step uses various heuristics to create and refine *mentions* and *RecommendedInstances* using the newly available information. The second responsibility is the actual mapping of *RecommendedInstances*, i.e., parts of the text that the approach identified as potential architecture elements, to the actual model elements. These mappings are then the wanted trace links.

5.2.6.1. Refinement of previous processing results

The refinement can be classified into two categories. First, some heuristics create and update existing *mentions* and *RecommendedInstances*. Second, we remove some results based on the available information.

Name-Type The *Name-Type* heuristic in the *Element Connection* is the same as before in the *Element Identification* (see Section 5.2.5). In contrast to before, the heuristic now also considers the instances from the model. This particularly means that instances' names are used to resolve ambiguous *mentions*. If a *mention* has a similar naming to an instance name, the likelihood for *name* is set to 100%. Then, it is analyzed if these changes result in new *RecommendedInstances* based on the previously introduced patterns.

Extraction-dependent Mentions This heuristic looks for parts in the text that are similar to instance names or types. This is necessary due to two main reasons. First, we do not want to miss *mentions* that slipped through the other heuristics but can easily be identified with this comparison. Second, we want to be more resilient when the preprocessing is erroneous. A common error is the mislabeling of the part of speech. As we focus on nouns, cases with mislabeling or where the instance name is not a noun can still be identified.

Reference To process all the *mentions* that were detected by heuristics like the *Extraction-dependent Mentions*, the *Reference* heuristic creates further *RecommendedInstances*. The general idea is to create *RecommendedInstances* if there are *mentions* similar to the model's instances. If a *RecommendedInstance* already exists for a *mention*, the confidence of the existing *RecommendedInstance* is increased.

Project Name This heuristic is a filter to remove *mentions* and *RecommendedInstances* based on the project name. The project name is rarely an actual component but rather refers to the whole system. However, the project name can be used similarly to model elements in text, potentially creating false positives. For example, describing a "button" in the GUI should not be confused with the "button" in the project name *BigBlueButton*.

5.2.6.2. Mapping of *RecommendedInstances* to model elements

In this last part, the *Connection Generator* compares the identified *RecommendedInstances* with the instances in the model and creates trace links. First, the approach iterates over the model instances and tries to find the most similar *RecommendedInstance*. If the confidence score for the identified *RecommendedInstance* is above a set threshold, a trace link is created. Second, the approach looks at the problem from the other direction. The approach goes through the list of *RecommendedInstances* and tries to find similar model instances, just as in the first step.

The procedure intends to look at links and their corresponding artifacts from both directions. The most similar instances from one direction might not cover everything, and some links may be missed. Overall, the approach tries to optimize and increase recall. Moreover, the goal is not to miss linking *RecommendedInstances* that might be overshadowed by other ones. This process also allows to have multiple mappings per element, further increasing recall. By factoring in the confidence score, we can ensure more precise links.

5.2.7. Limitations

There are some limitations to the concept of this approach. One of the biggest limitations is that relations are not taken into account. Currently, the approach does not consider relations between architecture elements, for one thing, to better create trace links, or, for another thing, to create trace links for the relations. The first reason is that relations between elements are rarely traced in other TLR tasks. A second reason is that relations were sparse in initial assessments of natural language SADs. However, relations are more present and central for some kinds of artifacts. For example, relations between elements are usually one of the main statements in diagrams, apart from the existence of elements. While I do not look at relations in this work, extending the approach to relations is possible. This mostly requires the creation of relation-specific heuristics for both, SAD and SAM. Further, it needs an approach to calculate the similarity of relations, especially if the relations only carry general names (e.g., simple *use* relations) or are even unnamed.

A second limitation lies in the used heuristics. Creating heuristics to identify architecturally relevant elements in SADs is challenging. The freedom of natural language allows a plethora of different wordings and sentence structures. There are no common rules that apply to all SADs. As such, the developed heuristics try to catch several cases but can never be universal. Potential future work is the creation of a big dataset of SADs to be able to create a ML-based approach that might deal with more cases.

Another core limitation of the approach is its reliance on similarity, which also applies to a certain degree to the abstraction levels. At multiple stages and in most heuristics, the approach uses similarity metrics. The approach relies on the reliability of the similarity metric. The similarity metric used needs to bridge the semantic gap between the entities and the artifacts they contain. The similarity metrics can be exchanged to make the approach more generally applicable, as discussed in Section 5.2.1. At the same time, the heuristics are designed with a certain abstraction level between SADs and SAMs in mind. The heuristics and similarity metrics will fail if the abstraction levels are too different. For example, assuming the SADs describes a component that should authenticate users by validating the password after hashing it. The architects wanted to reuse an existing component for hashing and another for password validation. Consequently, the SAM contains two components that interact instead of the one component described in the SAM. In such cases, similarity checks can struggle to identify both occurrences, especially if only one is expected. This gets even more complicated if only one responsibility is described but gets broken down into multiple responsibilities. A simple everyday example is *making coffee*. If one side breaks this task down into its several steps (e.g., *supply ingredients*, *grind beans*, *heat water*, etc.), existing similarity checks will most likely fail as they cannot detect that *making coffee* is the aggregation of the other steps. In software architecture, this can happen, for example, when a SAD describes multiple (related) responsibilities (e.g., *encryption*, *hashing*, *authentication*) but the SAM only represents the aggregated responsibility (e.g., *password storage*).

5.3. Recovering Links Between Model And Code

In this section, I will present the approach to recover trace links between SAMs and code that is named *ARchitecture-to-COde Trace Linking (ArCoTL)*. This section is mainly based on my publication at the IEEE/ACM International Conference on Software Engineering (ICSE) in 2024 [123] that is partly based on the master's thesis by Tobias Telge [265] that I supervised.

As the first kind of input, the approach uses SAMs in the same form as ArDoCo in Section 5.2. This means the approach expects structural component-based software architecture models. For the second kind of input, the approach expects the root location of the code files.

Just like ArDoCo in Section 5.2, the approach ArCoTL first transforms the input model and code into the intermediate representations as introduced in Section 5.1.2.1 and Section 5.1.2.2.

In the second step, ArCoTL uses various heuristics to find evidence of a link between an architecture and a code element. The general idea to recover trace links between SAMs and code is to mimic human behavior when looking for links. The approach has various components to implement the idea: Heuristics, aggregators, and filters. Section 5.3.1 covers the employed heuristics, and Section 5.3.2 tackles the aggregators and filters.

The approach uses a computational graph that is illustrated in Section 5.3.3 to execute the various components in classified order. The main challenge is to combine the heuristics, aggregators, and filters in such a way that actual trace links are favored while false positives are eliminated.

5.3.1. Heuristics

The approach uses heuristics to analyze the input artifacts and determine whether a link between two entities should exist. These heuristics were created based on considerations of how a human would analyze the artifacts to recover trace links. The goal is to mimic human performance. ArCoTL uses two types of heuristics: standalone heuristics and dependent heuristics.

Standalone heuristics operate directly on the input data. The heuristics calculate the similarity between entities of the artifacts to create a confidence score. Then they create a *mapping* that consists of the entity pair and the confidence score. The approach uses the following standalone heuristics:

Package	Compares package name with name of components
Path	Compares the path of a compilation unit with the names of components
Method	Compares method names with names of signatures
Names	Compares names of architecture elements with those of compilation units and datatypes

The heuristics are based on comparisons of certain entities. For these comparisons, the approach determines if one entity name contains parts of the other entity name. The approach identifies parts of an entity name based on word boundaries and camel casing, hyphenations, underscores, periods, and similar separators. Capitalization is ignored during the comparisons. The similarity score, i.e., the confidence in these comparisons, is the ratio of the parts that are contained in the other entity. Consequently, the comparisons are asymmetric. For example, “database” is contained in “DatabaseAdapter”, and the resulting similarity score is 50%. In the opposite direction, there is no containment, and, as a result, there is no similarity. This asymmetric comparison is crucial to increase precision and to have better control over containment relations. For instance, containment is important for heuristics like *Package* and *Path* where, e.g., a model entity name can be contained within a path.

Many programming languages use file system paths to encode package names. However, not all programming languages do this, and there can be slight differences for those who do. This is why the approach uses both heuristics, although they might be redundant. We exclude the package from the path to minimize redundant considerations of the package name in both the name and the path. For example, given the package `mediastore.persistence` and the path *ms-database/src/main/java/mediastore/persistence*, the approach only utilizes the beginning of the path, i.e., *ms-database/src/main/java* for the *Path* heuristic and the rest for the *Package* heuristic. This example also demonstrates the difference between both heuristics and why they are required: The starting of the path represents a folder that might indicate a component name and can also provide additional information when compared to the package name in this case. Such situations often occur if the project uses multiple modules, e.g., using maven modules.

Dependent heuristics operate both on the input and on the results of other heuristics to adapt the results. As such, the dependent heuristics take the mappings of one or more other heuristics along with the input artifacts as input. These heuristics then manipulate the *mappings* by changing the confidence score. They then output the adapted *mapping* list. There are the following dependent heuristics:

Hint Inheritance	Inherits results from other heuristics (mappings) along *extends*- and *implements*-relations.
Common Words	Checks if names differ only in common words or prefixes/suffixes (e.g., Test, Impl, I).
Ambig. Sub-packages	Detects ambiguously mapped sub-packages.
Component relations	Looks at relations between components to resolve ambiguity.
Interface provision	Checks if a *provide*-relation of the architecture exists in the source code.

The *Hint Inheritance* heuristic inherits mappings that include a class along the inheritance relations, e.g., to extending classes. The assumption is that there is a strong coupling between a class and its parent.

The *Common Words* heuristic adapts the results of the *Names* standalone heuristic. The idea is that two entities should still be similar or equal if they only differ in a certain set of common words, prefixes, or suffixes. For example, common words include “Test” to denote test classes, “Exception” for exception classes, and “Factory” for classes implementing a factory for another class. For prefixes and suffixes, the approach regards, for example, “Impl” that is used to indicate implementations of abstract classes. In such cases, the confidence scores of the corresponding mappings are increased.

The *Ambiguous Sub-packages* heuristic deals with cases where a component and its packages are included in another component’s package(s), thus creating an ambiguous mapping. For example, the package `dataaccess.preferences` can be mapped to both a *DataPersistence* and a *Preferences* component. This heuristic addresses this ambiguity and differentiates two cases. In the first case, the heuristic detects another location where the component (e.g., the *Preferences* component) is implemented. We then assume that there is no actual relation between the package (`dataaccess.preferences`) and said component. We reason that a component should not be implemented scattered in multiple places in the code. Therefore, the heuristic removes the mapping to the (*Preferences*) component. In the second case, if the approach does not detect a second location, we assume that the sub-package is likely the implementation of a component (here: *Preferences*), and the heuristic removes the mapping to the *DataPersistence* component.

The approach uses the heuristics *Component relations* and *Interface provision* to investigate relations between components and/or interfaces that might exist in the code. The heuristics adapt mappings and their confidence based on the existence or absence of such relations. This is important if multiple mappings exist between an architecture entity and code entities. In these cases, the heuristics can help to disambiguate and clarify. The heuristics might remove respective mappings if source code entities cannot be associated with relations in the architecture.

5.3.2. Aggregators and filters

To combine evidence in the form of *mappings*, the approach uses *aggregators* and *filters*.

Aggregators combine the output of two or more heuristics by combining their *mappings*. The goal of aggregators is to select a confidence score for each entity pair based on the confidences annotated by the heuristics. There are two categories of aggregators used to select the confidences: combiners and selectors.

Combiners are aggregators that, as the name suggests, combine the mappings generated by one or more heuristics about the same entity pair. The approach uses a *maximum* combiner that sets the confidence score of an entity pair to the highest confidence of mappings for this pair.

Selectors assess the mappings and, as the name suggests, select mappings based on specific criteria. Due to the nature of selections, selectors combine mappings and filter them. The approach uses two selection criteria, *best* and *first*. The *best* criterion looks at one fixed entity and looks at mappings that contain this entity. The best mapping according to the confidence score is selected for this entity. Consequently, one variant selects the best mapping for each model entity, and the other selects the best mapping for each code entity. The *first* criterion examines the order of the heuristics and selects, for one entity, the first mapping from an ordered list of heuristics. For each entity, mappings are assessed sequentially, and a mapping is chosen if its confidence is greater than 0 and no further mappings are assessed. The underlying idea is that the different heuristics have different importance, and as such, some kind of precedence must be factored in. The approach implements this precedence of heuristics by prioritizing heuristics based on the predefined ordered list.

Filters are special aggregators that remove mappings that contain negative evidence from dependent heuristics such as *Ambiguous Sub-packages*, *Component relations*, and *Interface provision*. Filters are crucial in refining the final results of our approach, increasing precision.

5.3.3. Computational Graph

The approach uses the computational graph depicted in Figure 5.11 to execute the heuristics and apply the aggregators and filters.

The entry points of the computational graph are the five standalone heuristics. The resulting mappings are then filtered with selectors using the *best* criterion. The approach first selects the best code entity for each architecture entity, i.e., the set of code entities that share the highest confidence for each corresponding architecture item. Then, the computation proceeds inversely, selecting the best architecture entity for the previously selected code entities. One exception to this is the *Names* heuristic for components. For this

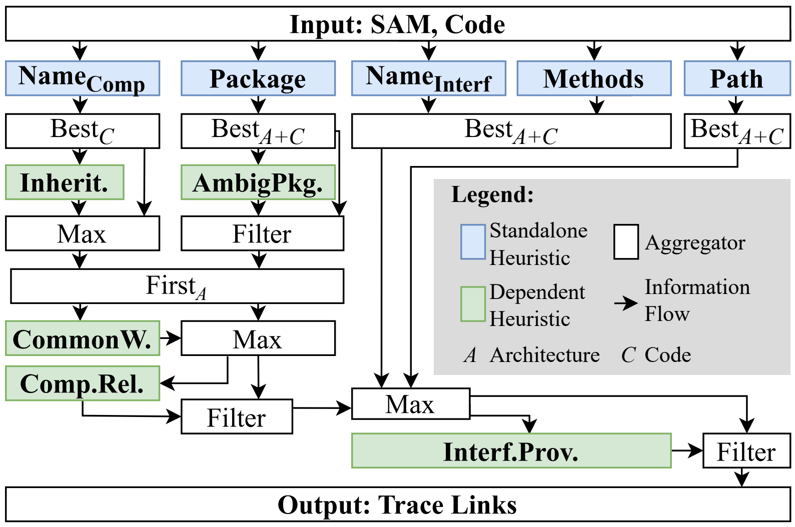


Figure 5.11.: The computational graph of ArCoTL, by Keim et al. [123]

heuristic, the filtering solely uses the best architecture entity for each code entity, not the inverse direction. Architecture entities can be implemented with one or more code entities, and thus, limiting architecture entities to only the best code entity will likely reduce recall. Generally, the approach applies both directions for more precise results and only one direction for broader exploration to potentially achieve higher recall.

The approach utilizes selectors with the *first* criterion when combining the standalone heuristics *Package* and *Names*, prioritizing package names. For this, we assume that package and component structures are likely to contain similarities. Still, names can be part of the package, and consequently, the approach first selects the results of the *Package* heuristic and, if no mapping is found, falls back to the *Names* heuristic.

After the standalone heuristics, the approach uses the various dependent heuristics to adjust confidence values and also to filter out unlikely mappings. The *maximum* combiners are used to select only those mappings with the highest confidence for each entity pair. The last refinement step uses the *Interface provision* heuristic to filter the results to increase the precision of the final results.

In the last step, the approach generates trace links for each mapping with a positive confidence value. If the target of a trace link is a package or path, the approach recursively resolves it. This means that the approach creates a trace link for each class contained in the package or path. This is done recursively if there are further packages or folders. The approach performs this resolution because of the intended granularity of links, i.e., model elements like components to classes.

5.4. Recovering Links Between Documentation And Code

In this section, I will present the approach to recover trace links between SADs and code that is named *Transitive links for Architecture and Code (TransArC)*. This section is mainly based on my publication at ICSE in 2024 [123].

When linking SADs and code, an approach needs to analyze the artifacts in a way that allows it to bridge the semantic gap between both artifacts. There is a comparably big semantic gap between those kinds of artifacts due to their vastly different levels of abstraction and different levels of detail. SADs contain design decisions about the entities and also about reasoning or architectural guidelines and patterns. In contrast, code contains entities along with execution semantics and low-level documentation. Moreover, the entities within SADs often encapsulate several code entities. For example, a component that is mentioned in the SAD can be implemented with multiple classes, potentially spread over multiple packages or modules. On top of that, common difficulties include inconsistencies like changed naming [290].

To bridge the semantic gap, the approach for TLR between SAD and code names TransArC uses a simple idea as shown in Figure 5.12. The idea is to reduce the semantic gap by using intermediate artifacts instead of directly tackling the bigger semantic gap. There is not one big semantic gap with intermediate artifacts but two smaller ones. The idea is that the intermediate artifacts are semantically between the other artifacts and, thus, are semantically closer to each other. This allows us to simplify the recovery of trace links as two easier problems must be tackled. For these two smaller problems, we can benefit from approaches that are specialized to the corresponding

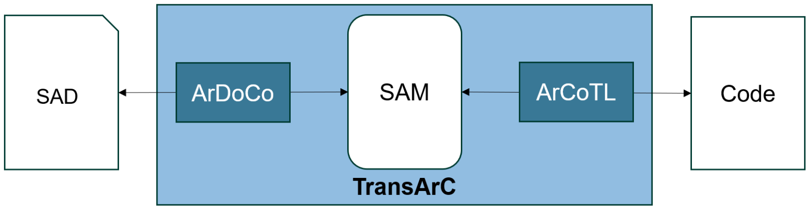


Figure 5.12.: The TransArC approach, by Keim et al. [123]

problems. In the case of TLR between SAD and code, we can use component-based architecture models like UML component diagrams as intermediate artifacts. With these SAMs as intermediate models, we can, on one side, use an approach for TLR between SAD and SAM and, on the other side, an approach for TLR between SAM and code.

Consequently, as shown in Figure 5.12, TransArC combines ArDoCo and ArCoTL: ArDoCo is used to recover trace links between SAD and SAM, ArCoTL is used for TLR between SAM and code. Using transitivity, the approach combines the trace links from ArDoCo that connect sentences in the SAD with model entities in the SAM with the trace links from ArCoTL connecting model entities in the SAM with entities like classes in the code.

This also updates the definition of a trace link for the TransArC approach: A transitive trace link is a connection between two entities with a third entity that is connected to each of the two entities, acting as a bridging intermediate entity.

This approach has one major downside: it is not regarding potential direct links between SAD and code that are not represented in any form in the intermediate SAM. For example, a class that is described in the SAD will not be regarded if it cannot be linked to the SAM. While there are cases where these situations can arise, in my experience, it is rather uncommon. SADs usually describe the system on a different abstraction level and do not go that much into implementation details. This is also because the idea and abstraction of architecture documentation do not directly include code-level information. However, there is also no strict specification that prohibits referencing code entities. This restriction can potentially be solved in future work by combining the approach with other approaches that try to map SADs or similar to code directly.

5.5. Evaluation

This chapter introduced three approaches that address three use cases of TLR.

The first case is TLR between SADs and SAMs. The presented approach ArDoCo takes SADs and SAMs as input, processes them in an agent-based pipeline approach, and outputs trace links (cf. Section 5.2).

The second case is TLR between SAMs and code. The approach ArCoTL (ARchitecture-to-COde Trace Linking) takes SAMs and code as input, transforms them into intermediate models, and calculates trace links in a calculation graph using various heuristics (cf. Section 5.3).

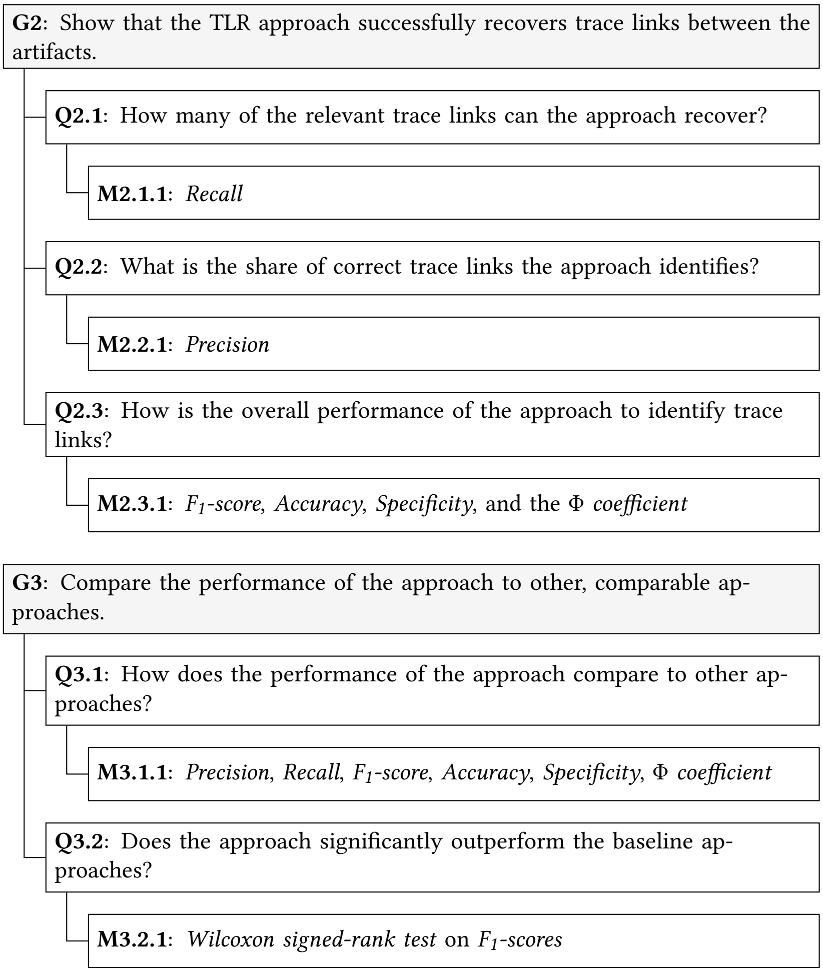
The third case is TLR between SADs and code. The approach TransArC (Transitive links for Architecture and Code) uses the previous two approaches and combines the results transitively to recover trace links between SADs and code using SAMs as the intermediate artifact (cf. Section 5.4).

This section evaluates these approaches. For the evaluation, I have several goals that the evaluation should achieve that I present in Section 5.5.1. I then introduce the datasets used for the evaluation in Section 5.5.2 before evaluating the different approaches in Section 5.5.3 for **RQ 3.1**, Section 5.5.4 for **RQ 3.2**, and Section 5.5.5 for **RQ 3.3**. I also briefly explore the application of LLMs to recover trace links in Section 5.5.6 and answer **RQ 3.4**. Lastly, in Section 5.5.7, I discuss the threats to validity for this evaluation.

5.5.1. Evaluation Plan

For the evaluation, I have several goals that I want to achieve. As such, I base the evaluation on the GQM approach by Basili et al. [15] and align it with the state-of-the-art as presented by Konersmann et al. [140]. For each contribution, I define these goals and derive corresponding questions for each goal to examine the fulfillment of the goals. I use various metrics and, for each question, I select fitting metrics that allow us to answer each question. I introduced and detailed the metrics that I use in Section 2.8.

Figure 5.13 shows the goals, questions, and metrics for the TLR approaches. The goals, questions, and metrics are identical for each different kind of use



G2: Show that the TLR approach successfully recovers trace links between the artifacts.

Q2.1: How many of the relevant trace links can the approach recover?

M2.1.1: *Recall*

Q2.2: What is the share of correct trace links the approach identifies?

M2.2.1: *Precision*

Q2.3: How is the overall performance of the approach to identify trace links?

M2.3.1: *F_1-score*, *Accuracy*, *Specificity*, and the Φ *coefficient*

G3: Compare the performance of the approach to other, comparable approaches.

Q3.1: How does the performance of the approach compare to other approaches?

M3.1.1: *Precision*, *Recall*, *F_1-score*, *Accuracy*, *Specificity*, Φ *coefficient*

Q3.2: Does the approach significantly outperform the baseline approaches?

M3.2.1: *Wilcoxon signed-rank test* on *F_1-scores*

Figure 5.13.: Goals, Questions, and Metrics for Evaluating the Traceability Link Recovery Approaches

case for TLR in this thesis, i.e., SADs to SAMs, SAMs to code, and SADs to code.

The initial goal (Goal **G2**) is to show that the approaches can appropriately recover trace links. Accordingly, the questions tackle the performance of

Table 5.1.: Taxonomy for assessing the performance of TLR approaches based on Hayes et al. [93]. The values for the F_1-score are derived from the precision and recall.

	Acceptable	Good	Excellent
Recall	0.60 - 0.69	0.70 - 0.79	0.80 - 1.00
Precision	0.20 - 0.29	0.30 - 0.49	0.50 - 1.00
F_1-score	0.30 - 0.41	0.42 - 0.60	0.61 - 1.00

the approaches to recover their corresponding trace links. To answer the questions, I primarily use the metrics *Precision*, *Recall*, *F_1-score*, supported by the metrics *Accuracy*, *Specificity*, and the Φ *coefficient* (see also Section 2.8). To assess the significance, I calculate the Wilcoxon signed-rank test on the F_1-scores.

To classify the performance of the approaches, I use the taxonomy established by Hayes et al. [93], as shown in Table 5.1. According to this taxonomy, a recall rate above 60% is considered acceptable, above 70% is deemed good, and above 80% is excellent. In contrast, precision does not need to be as high. Values above 20% are deemed acceptable, and those above 50% are considered excellent. This prioritization stems from the understanding that recall is generally more critical for TLR. Users can easily identify incorrect trace links when precision is lower, but identifying missing trace links due to lower recall is more challenging and time-consuming.

Consequently, some researchers prefer to deviate from the traditional F_1-score and use other β values in the generalized F_β measure (see Equation 4). For instance, Winkler et al. conducted a user study to determine an appropriate β value based on the time required to correct a defect, finding $\beta \approx 6.2$, which places significantly more weight on recall than precision [288]. In my research, I adhere to the F_1-score, as it remains the most widely used metric in the community. Additionally, assessing an accurate β is very time-consuming and error-prone and can be highly influenced by various biases.

5.5.2. Evaluation Datasets

For the evaluation, we need datasets that contain the necessary artifacts, i.e., natural language SADs, SAMs, and code, for different projects. The approaches focus on processing English documentation, and, as such, the dataset requires English SADs.

Additionally, the dataset needs to contain the gold standards for our objectives. This means that we need a gold standard for TLR between SAD and SAM, between SAM and code, and between SAD and code. Moreover, there is the need for gold standards to evaluate the identification of MMEs and UMEs in the ID between SAD and SAM.

In this section, I discuss the datasets that are used for the evaluation and their creation. This section is partly based on our publication at the joint workshops MSR4SA (Mining Software Repositories for Software Architecture) and SAEroCon (Software Architecture Erosion and Architectural Consistency) at ECSA in 2022 [71] as well as our publication at ICSE in 2024 [123].

Before creating my dataset, I analyzed existing datasets in the TLR community. A widely used dataset is the *CoEST*[5] repository. Currently, there are 15 projects with gold standards for TLR between requirements and source code. However, *CoEST* cannot be used in this case: The dataset is missing SADs, and some projects contain non-English language like Italian. Lastly, *CoEST* does not contain any SAMs.

Some missing artifacts can theoretically be created artificially, e.g., by manually deriving SAMs from code. This is highly undesirable because this means we have to create most of the artifacts that should be processed artificially and the gold standards between these.

Overall, I am unaware of a dataset that provides the required artifacts for our intended cases. Consequently, we created a benchmark dataset that can be used for these cases.

Using benchmarks in evaluation brings numerous benefits, as Sim et al. state [247]. Among these benefits are clearly defined standards and expectations, increased awareness of related work, and more frequent collaborations within a domain. Still, in a study of 153 full technical papers at the conference-series ECSA and ICSA between 2017 and 2021, Konersmann et al. [140] show that only 2.6% of papers state that they use benchmarks for evaluation. Instead, most papers use case studies and technical experiments (57%).

[5] `http://sarec.nd.edu/coest/datasets.html`

For the evaluations of the approaches, we create a benchmark dataset by mining publicly available studies from papers, following Kistowski et al. [134]. As a starting point, we use the overview by Konersmann et al. [140] of papers that made their (case) studies publicly available. Next, we mine public software repositories, select appropriate projects, transform the extracted data into uniform formats, and label them.

To find fitting projects for our cases, we looked for open-source projects that contain SADs. For this, we retrieved the list of open-source projects that Ding et al. [52] identified contain some form of SAD. Moreover, we analyzed the repositories of the *Lindholmen dataset* [94].

From all of these sources, we could not identify suitable projects that have both, an appropriate SAD and a presentation of the architecture, e.g., as a figure, diagram, or similar. According to Ding et al., [52], smaller projects usually lack architecture documentation. Large, successful projects are more likely to have maintained architecture documentation.

Lastly, we looked at other approaches and related work that use SAMs to identify potential candidates. From these projects, we selected those that fit our requirements: the projects need to have a SAM, a SAD, and the code needs to be mostly written in Java. The project needs to be written in Java because, currently, there is only a transformation into the intermediate representation for Java. The following listing shows a brief overview of the selected projects:

Table 5.2 shows some characteristics of the benchmark projects. The projects have different sizes in terms of their lines of code. *MediaStore (MS)* is the smallest project with only four thousand lines of code (kLOC). MS is also a model or example application that is not available on a platform like GitHub, which supports easy forking and contributing. We cannot clearly state the number of forks or contributors. Similarly, *TeaStore (TS)* is an example and scientific application developed for evaluating performance analysis tools. The other

[6] `http://sdq.kastel.kit.edu/wiki/Media_Store`
[7] `http://github.com/TEAMMATES`
[8] `http://bigbluebutton.org`
[9] `http://github.com/DescartesResearch/TeaStore`
[10] `http://github.com/JabRef/jabref`
[11] rounded kLOC for programming languages with most LOC (calculated via `cloc`)

MediaStore[6]	is a "model application built after the iTunes Store". Its architecture was used for exemplary performance analyses on SAMs.
TEAMMATES[7]	is an open-source "online tool for managing peer evaluations and other feedback paths of your students". TEAMMATES is used as a case study in several SAD-based approaches (cf. [129, 241]).
BigBlueButton[8]	is a non-scientific application that provides a web conferencing system focusing on creating a "global teaching platform".
TeaStore[9]	is a scientific application [135] that is used as a "microservice reference test application". Like MediaStore, it is used to evaluate architecture performance analyses.
JabRef[10]	is a tool to manage citations and references in bibliographies. It has features to collect, organize, cite, and share research work.

Table 5.2.: Characteristics of the projects in the benchmark with thousand lines of code (kLOC) of the languages, number of forks, and number of contributors (Contrib.)

Project	Languages (kLOC)[11]	Forks	Contrib.
MediaStore	Java (4)	N/A	N/A
TEAMMATES	Java (91), TypeScript (54)	$\approx 2.6k$	≈ 500
BigBlueButton	JavaScript (69), JSX (47), Scala (22), Java (21)	$\approx 5.8k$	≈ 180
TeaStore	Java (12)	$\approx 0.1k$	≈ 15
JabRef	Java (157)	$\approx 2.0k$	≈ 490

three projects, *TEAMMATES (TM)*, *BigBlueButton (BBB)*, and *JabRef (JR)*, are medium-sized projects with a reasonable number of forks and contributors.

All of these projects in the benchmark already contain a SAD that can be directly used for our evaluation purposes. For SAMs, we require the development view (cf. Kruchten's 4+1 view model [145]) of the system as structural architecture models. The models for MediaStore [254], TeaStore [135], and TEAMMATES [179] originate from other researchers. We reverse-engineered the models for BigBlueButton and JabRef (see also Fuchß et al. [71] and Keim et al. [124]) by using and transforming existing models within the software

Table 5.3.: Overview of the benchmark and the gold standards with the number of artifacts per artifact type, the number of trace links in the gold standard for each task and project, and the solution space.

Artifact Type / Task		MS	TS	TM	BBB	JR
SAD	#Sentences	37	43	198	85	13
SAM	#Model elements	23	19	16	24	6
Code	#Files	97	205	832	547	1,979
SAD-SAM	#Trace links	29	27	51	52	18
	Solution Space	851	817	3,168	2,040	78
SAM-Code	#Trace links	60	164	1,616	730	1,956
	Solution Space	2,231	3,895	13,312	13,128	11,874
SAD-Code	#Trace links	50	707	7,610	1,295	8,240
	Solution Space	3,589	8,815	164,736	46,495	25,727

architecture community (cf. [201, 115, 271]) as closely as possible. Most importantly, these models are not specifically created for the research that we want to evaluate.

Table 5.3 shows, for each project, the number of artifacts for each artifact type, the number of expected trace links, and the solution space. The solution space shows how many possible trace links exist, while the expected trace links show the number of actual correct trace links. This further shows how different the projects are. MS, TS, TM, and BBB have a comparable number of model elements. JR has the smallest SAM with only six model elements, roughly a third to a fourth of the others.

Despite having mostly similar number of model elements, the projects have distinct sizes of their SADs: The SADs of MS and TS are almost equally sized, and BBB's SAD has roughly double their size. At the same time, JR has only around a third as much architectural documentation despite being, code-wise, the largest project. Based on the size of the architectural documentation, TM is the largest project with 198 sentences.

The dataset is intentionally curated to include projects with various characteristics, such as different domains, sizes, and documentation-to-code ratios. This diversity aims to minimize potential confounding factors, thereby enhancing the robustness and validity of our research findings. However, there are still issues. Although our projects span various domains and sizes, they do not comprehensively represent all possible application variants. Consequently, certain types of projects or specific characteristics may be overrepresented or underrepresented in our dataset. This imbalance could potentially skew the evaluation results, limiting the generalizability and applicability of our findings to different projects and real-world scenarios.

Aside from the input artifacts, the evaluation also requires gold standards that contain the expected trace links between the artifacts. Unfortunately, there are no existing gold standards for our use cases and selected projects. Therefore, we created them through a rigorous process: Initially, at least two researchers independently created a gold standard for each given pair of artifacts. Then, these gold standards were then compared, and discrepancies were resolved through discussions, resulting in a merged gold standard. This process mitigates individual biases but does not eliminate all threats to validity. For instance, potential biases, such as researchers interpreting artifacts differently, can still impact the reliability of our findings to some extent.

The gold standard for TLR between SAD and SAM consists of mappings between the sentences in the SADs and the indicated model elements in the SAMs. To identify sentences, we assign consecutive sentence numbers as identifiers. For model elements, we use their UUIDs (Universally Unique Identifiers). Each sentence can refer to multiple model elements; conversely, each model element can be referenced in multiple sentences.

The gold standard for TLR between SAMs and the code consists of mapping the model elements of the SAMs to the corresponding relative paths of the source code files. For each model element, we select the best-fitting source code elements. Given the different levels of abstraction, each model element can map to multiple source code elements. For example, an interface in the model might correspond to multiple interfaces in the code. A concrete example is the interface *IDownload* in the model that needs to be mapped to the *IDownload* and *IDownloadCache* in the code.

Sometimes, the best mapping is at the folder or package level rather than the file level. For instance, a component might be implemented across an entire package, such as the component *DataPersistence* and the package `dataaccess` in the example in Section 1.5. In such cases, all contained elements are considered part of the component, and we trace them accordingly by creating a trace link between the model element and each code element within the package.

Like TLR between SAD and SAM, several classes can implement a given component, resulting in multiple trace links. Likewise, a single class can implement multiple components, particularly in low-quality code with mixed concerns.

The gold standard for TLR between SADs and code comprises the links between the sentences in the SADs and the corresponding source code files. To achieve this, we can combine the gold standards for SAD-SAM and SAM-code. However, we also need to identify cases where there is a direct link between the documentation and the code that is not represented in the intermediate artifact.

If a source code folder needs to be linked, we resolve the link to include the folder's contents corresponding to the relevant sentence in the SAD. Additionally, each sentence in the SAD can be linked to multiple classes in the code, and each class can be linked to multiple sentences in the SAD.

Table 5.3 shows the number of expected trace links in the gold standards for each artifact pairing aside the solution space (i.e., the number of possible trace links).

It is important to note that we do not create a semantic abstraction of the inputs (i.e., "semantic clusters") and then create links between these abstractions/clusters. For example, a collection of classes dispersed across the codebase might collectively represent the implementation of a specific component, such as a security component, but only when considered together. This clustering and combination of classes require deep knowledge of the code's intricacies and interactions. While this method is intriguing, it is rarely employed and may be explored in future work.

The resulting benchmark datasets with inputs and expected outputs, i.e., the gold standards, are also publicly available [72].

5.5.3. Recovering Links Between Documentation And Model

In this section, I will evaluate the approach ArDoCo for TLR between SAD and SAM (see Section 5.2), based on my publications at ECSA in 2021 [129] and at ICSA in 2023 [124]. The evaluation is designed to answer the respective questions from the GQM plan (see Section 5.5.1), i.e., Question **Q2.1**, Question **Q2.2**, and Question **Q2.3** to assess the performance of the approach and

Question **Q3.1** and Question **Q3.2** to compare the performance to other approaches. In this evaluation, I follow existing methods, e.g., by Hayes et al. [93], and the guidelines by Shin [246].

5.5.3.1. Methodology

To evaluate the approach for TLR between SAD and SAM, we use the projects and the contained gold standard from the dataset (see Section 5.5.2). We apply the approach and compare the results with the expected trace links from the gold standard.

For the comparison with other approaches (Question **Q3.1**), we look into several baseline approaches from the literature. As candidate approaches, we use two adapted approaches from the literature and a common baseline approach. The two approaches from the literature are approaches for TLR but are not created specifically for TLR between SAD and SAM. However, these approaches do not have strict requirements for the artifacts and can be easily adapted to the case. I selected these approaches due to the availability of a replication package, making them executable. Furthermore, the approaches could be adapted to the required use case and do not make artifact-specific assumptions that are required for the TLR process. Among the available and adaptable approaches, these approaches promised the best results according to the reported results. In the following, I first describe all three approaches before further detailing the setting in Paragraph 5.5.3.1.

Evolutionary Approach by Rodriguez and Carver The first approach by Rodriguez and Carver [233] considers the problem as an optimization problem using the Nondominated Sorting Genetic Algorithm (NSGA-II) [48] with the Jaccard similarity and weighted cosine similarity as fitness functions. To do so, they start with random pairings that should represent potential trace links. Each iteration generates a new population out of the best candidates from the previous population. Best candidates are kept and the rest of the population is created using mutation and crossover operators.

Information Retrieval Approach by Zhang et al. The second approach is by Zhang et al. [301]. The approach calculates the similarity between requirements and source code using synonyms, verb-object phrases, and structural information. A vector space model is employed, where vectors contain weights for unique words and phrases. These weights are determined by term frequency and inverse document frequency (see also Section 2.3.3). Terms are extracted from verb-object phrases in both requirements and code, and synonyms are resolved to unify these terms. When similar terms are identified, a trace link is established. Unlike the original approach, we adapt the verb-object phrase extraction for models and introduce a similarity threshold to enhance precision.

Baseline Approach: Simple Tracelink Discovery (STD) As a common baseline approach, we create an approach that we call *Simple Tracelink Discovery (STD)*. STD assumes that elements that should be linked have equal or similar naming. This assumption is commonly applied in Information Retrieval approaches. Therefore, the baseline approach utilizes standard Information Retrieval techniques, i.e., n-grams and word similarity methods. The approach extracts n-grams, specifically unigrams, bigrams, and trigrams, from the words in each sentence and model element, then compares these n-grams between the text and the model. The comparison is case-insensitive and employs the normalized Levenshtein distance (see Section 2.3.4), as defined by Charlet and Demnati [33], to assess n-gram similarity. If the similarity between two n-grams exceeds a specified threshold, a trace link is established between the corresponding sentence and model element.

Evaluation Setting We first perform an exploratory pre-study using only a subset of projects and only the metrics precision, recall, F_1-score. This first study aims to assess the performance of the baseline approaches. This way, we can select the best baseline approach before elaborately performing the time-consuming effort of finding optimal parameters and calculating all metrics on all projects. For this initial study, we used two smaller projects, MediaStore and TeaStore, and the bigger project, TEAMMATES.

We first perform a parameter optimization for all approaches with thresholds or similar parameters to retrieve the best possible F_1-score. For the approach by Rodriguez and Carver, we run experiments ten times and only select the best results to cover the random factor of the approach and to identify the maximum performance of the approach.

After the pre-study, we continued with the main study, adding the two missing projects, BigBlueButton and JabRef. Based on the results of the pre-study on the first three projects (see Paragraph 5.5.3.2), the primary study proceeds with using only the baseline approach STD. We decided to do so because it is the best-performing approach among the baselines, and there is no indication that the other approaches will perform significantly better for the remaining projects.

We use the metrics stated in the GQM plan (see Section 5.5.1) for the respective questions. These metrics are precision, recall, F_1-score, accuracy, specificity, and the Φ coefficient. To assess if the results are significant (Question **Q3.2**) compared to the baseline approach STD, we use the one-sided Wilcoxon signed-rank test on the F_1-score.

5.5.3.2. Results and Interpretation

In the following, I report the results of the pre-study in Paragraph 5.5.3.2 and the results of the main evaluation in Paragraph 5.5.3.2.

Results of the Pre-study Table 5.4 shows the results of the pre-study for the baseline approach STD, the approach by Rodriguez and Carver [233], the approach by Zhang et al. [301], and our approach ArDoCo. The evolutionary approach proposed by Rodriguez and Carver performs the worst among the evaluated approaches, showing significant shortcomings in both precision and recall. The primary issue with this approach lies in its inherent randomness. When the algorithm selects incorrect initial trace link candidates, it often fails to identify the correct trace links. This problem is evident in the evaluation, where most repeated runs yield results significantly worse than the reported values. The values presented in Table 5.4 represent the best outcomes from applying the approach ten times. In most other runs, the approach achieves F_1-scores as low as 0%.

The approach by Zhang et al. outperforms the method proposed by Rodriguez and Carver, achieving better overall results. Specifically, their approach demonstrates higher precision than recall, primarily due to the introduction of a threshold. Lowering this threshold improves recall but at the expense of precision. The reported values represent the optimal configuration (e.g., regarding the threshold) according to the F_1-score.

Table 5.4.: Results of the pre-study for TLR between SAD and SAM, using the baseline approach Simple Tracelink Discovery (STD), the ArDoCo approach, and the approaches by Rodriguez and Carver (R & C) and by Zhang et al. The best values per column are highlighted.

Project	**Metric**	**R & C**	**Zhang**	**STD**	**ArDoCo**
MediaStore	Precision	0.07	0.76	**1.00**	**1.00**
	Recall	0.32	0.52	0.39	**0.62**
	F_1-score	0.12	0.62	0.56	**0.77**
TeaStore	Precision	0.10	0.35	0.89	**1.00**
	Recall	0.20	0.28	0.36	**0.74**
	F_1-score	0.13	0.31	0.51	**0.85**
TEAMMATES	Precision	0.10	0.49	**0.76**	0.56
	Recall	0.15	0.30	0.45	**0.90**
	F_1-score	0.12	0.37	0.56	**0.69**
Average	Precision	0.09	0.53	**0.88**	0.85
	Recall	0.22	0.37	0.40	**0.75**
	F_1-score	0.13	0.44	0.54	**0.77**
Weighted Avg.	Precision	0.10	0.52	**0.86**	0.79
	Recall	0.19	0.34	0.41	**0.78**
	F_1-score	0.13	0.41	0.55	**0.75**

The best-performing baseline approach is the STD approach. STD achieves the highest precision across all projects and in both average (88%) and weighted average (86%) calculations. This indicates that the links identified by STD are almost always correct. This is due to the use of the normalized Levenshtein distance combined with a high threshold value that causes the approach to only create links if there is coherent naming in the SAD and the SAM. STD has the second-best recall values on average behind ArDoCo, performing comparably to or better than the other two baseline approaches. Although the recall is not perfect, it identifies a larger proportion of true links than the other approaches. According to the taxonomy by Hayes et al. [93] (see Table 5.1), the results of the STD approach fall on the high end of the spectrum for being classified as *good*. STD consistently achieves high precision, recall,

Table 5.5.: Detailed Results of ArDoCo for TLR between SAD and SAM Using the Metrics Precision (P), Recall (R), F_1-score (F_1), Accuracy (Acc), Specificity (Spec), and the Φ coefficient (Φ)

Project	**P**	**R**	**F_1**	**Acc**	**Spec**	**Φ**
MediaStore	1.00	0.62	0.77	0.98	1.00	0.78
TeaStore	1.00	0.74	0.85	0.99	1.00	0.85
TEAMMATES	0.56	0.90	0.69	0.97	0.98	0.70
BigBlueButton	0.88	0.83	0.85	0.99	0.99	0.84
JabRef	0.90	1.00	0.95	0.97	0.97	0.93
Average	0.87	0.82	0.82	0.98	0.99	0.82
Weighted Avg.	0.83	0.82	0.80	0.98	0.99	0.80

and F_1-scores, making it the most effective and reliable approach among the baseline approaches evaluated in the pre-study.

In this pre-study, ArDoCo demonstrates superior performance and outperforms all baseline approaches, achieving *excellent* results in both precision and recall.

Results of the Main-study After the pre-study, as described in the methodology in Section 5.5.3.1, we perform an extensive evaluation on all projects of the dataset (see Section 5.5.2) using all metrics stated in the GQM. Table 5.5 shows the results of the evaluation.

Based on the taxonomy by Hayes et al. [93] (see Table 5.1), the approach performs on average *excellently.*

The overall precision is *excellent*, except for TEAMMATES as an outlier. The issue with TEAMMATES arises because certain terms in the documentation are similar to component names but do not represent the components themselves. For example, a "processed image" is an object that should not be confused with the `ImageProcessor` component. Although there is a relationship between the two, with one being the output of the other, a trace link to the component should not be created every time the object is mentioned.

The recall is also generally *excellent*, with MediaStore being the worst-performing project, achieving a recall of 62%.

The results also show the difficulties that I mentioned earlier about the different abstraction levels and styles for different projects. This makes creating heuristics hard that are general enough on one side and precise enough on the other side for good performance. All projects have high precision except for TEAMMATES as the outlier. At the same time, all but MediaStore have high recall, making it the outlier in this regard. The shortcomings of ArDoCo in this context are twofold.

First, ArDoCo relies on similar naming conventions, and if the naming within the documentation deviates significantly from the naming within the SAM, the approach naturally fails to create the trace link. This issue highlights one of the main challenges in bridging the semantic gap between artifacts. Future work should address this problem by developing methods to create trace links when artifacts are semantically similar, even if their naming conventions differ.

The second shortcoming is handling coreferences, such as using terms like "it" in the text. The current approach does not perform extensive coreference resolution due to the unreliability of existing methods. Implementing regular coreference resolution methods can degrade performance. As a result of not using coreference resolution, the approach's effectiveness decreases when numerous coreferences are present in an SAD.

Based on the results for precision and recall, the resulting F_1-score is also *excellent*. While some projects like MediaStore and TEAMMATES show imbalances between precision and recall, others like TeaStore, BigBlueButton, and JabRef achieve a better balance.

Accuracy and specificity achieve fairly high values, close to 100%. The high values for accuracy and specificity across all projects indicate reliable identification of true negatives. However, these results are influenced by the ratio of possible trace links to actual trace links in the gold standard. This is common in TLR, as each source artifact typically links to only a few target artifacts, making the problem space much larger than the solution space.

On average, there are approximately 35 actual trace links in the gold standard, compared to around 735 possible trace links in the problem space. Consequently, only about 5% of the possible trace links need to be identified as trace links. This disparity slightly skews these metrics, as not detecting any trace links or recovering every trace link falsely would still result in specificity and

Table 5.6.: Comparison Of The Average Results For TLR Between SAD And SAM Of ArDoCo And The Baseline STD Using the Metrics Precision (P), Recall (R), F_1-score (F_1), Accuracy (Acc), Specificity (Spec), And The Φ Coefficient (Φ)

Approach	P	R	F_1	Acc	Spec	Φ
Average						
- Baseline	0.78	0.58	0.65	0.96	0.99	0.64
- ArDoCo	0.87	0.82	0.82	0.98	0.99	0.82
Weighted Average						
- Baseline	0.75	0.53	0.61	0.96	0.99	0.60
- ArDoCo	0.83	0.82	0.80	0.98	0.99	0.80

accuracy values of approximately 95%. Despite this, the high results close to 100% still confirm good performance.

Consistent with the other metrics, the Φ correlation also shows promising results with high values. The values range from 70% for TEAMMATES to 93% for JabRef, resulting in an average Φ correlation of 82% (weighted 80%). These values indicate that the output of the approach closely aligns with the expected output.

ArDoCo's performance varies across different projects, but are overall *excellent*. The average and weighted average metrics indicate robust overall performance, with ArDoCo showing an ability to maintain high standards across different projects. Still, there is room for improvement in balancing precision and recall in specific cases like TEAMMATES.

To assess the performance of ArDoCo compared to other methods, we conducted a detailed analysis against the best-performing approach from the pre-study, the STD approach. Table 5.6 shows the average results of both approaches. Detailed results for the baseline Simple Tracelink Discovery (STD) are displayed in the appendix in Table B.1.

ArDoCo consistently achieves higher precision than the baseline STD, both in terms of the average and weighted average results. Additionally, ArDoCo significantly outperforms the baseline STD in recall, indicating its superior ability to identify true positives. Consequently, the F_1-score for ArDoCo is substantially higher in both average and weighted average cases, underscoring the method's well-balanced performance between precision and recall.

The detailed results show that the baseline achieves strong results for some projects but lackluster performance for others. The results also showed differences in precision and recall for the different projects. For some projects the precision is (much) higher, for some other projects the recall. This phenomenon is similar to the results of ArDoCo, indicating the challenge the benchmark imposes due to different abstractions and styles.

While both approaches have high accuracy, ArDoCo demonstrates a slight advantage. The high accuracy observed in both approaches is primarily attributable to their high specificity, reflecting a low incidence of false positives. This high specificity also suggests that both approaches effectively identify true negatives, likely influenced by the dataset's low prevalence of actual trace links.

The Φ coefficient for ArDoCo is significantly higher than that for the baseline STD, indicating a stronger overall correlation between predicted and actual values.

In summary, ArDoCo shows superior performance across all evaluated metrics when compared to the baseline STD. This improvement is particularly significant in recall, F_1-score, and the Φ coefficient, suggesting that ArDoCo is more effective at recovering trace links with higher reliability and consistency.

Finally, the significance of these findings is confirmed by a one-sided Wilcoxon signed-rank test ($\alpha = 0.05$) on the F_1-score, which shows that ArDoCo significantly outperforms the baseline STD.

5.5.3.3. Discussion & Conclusion

ArDoCo's consistently higher precision and significantly better recall than the baseline STD are particularly noteworthy. Precision is critical in minimizing false positives, ensuring that the method identifies relevant instances accurately. The improvement in recall demonstrates ArDoCo's superior capability in detecting true positives, which is essential for comprehensive coverage of relevant instances. This balance between precision and recall is crucial, as it reflects the method's ability to perform well across different conditions without compromising one metric for the other. The improved F_1-score for ArDoCo, which harmonizes precision and recall, further emphasizes its robust performance. These good results are important as they can affect subsequent usage and processing, like with our approaches for ID (see Chapter 6).

The significantly higher Φ coefficient for ArDoCo strongly indicates its superior overall correlation between predicted and actual values. This metric is particularly valuable as it provides a holistic view of the method's performance, capturing both its strengths and weaknesses across the evaluation's various dimensions. A higher Φ suggests that ArDoCo is more aligned with the actual distribution of true positives and true negatives, enhancing its credibility as a reliable method in real-world applications.

These results suggest that ArDoCo offers a more effective and reliable approach than the baseline approach. The superior performance in recall indicates that ArDoCo could be particularly useful in applications where missing a true positive has significant consequences. The method's balanced performance, as highlighted by the F_1-score, also suggests that it can be applied in diverse settings with varying requirements for precision and recall.

However, the evaluation contains some threats to validity. I discuss these threats in Section 5.5.7.

Overall, these results position ArDoCo as a promising tool for future applications, offering a good mixture of accuracy, reliability, and comprehensive detection capability that outperforms existing approaches.

5.5.4. Recovering Links Between Model And Code

In this section, I will evaluate the approach ArCoTL for TLR between SAM and code (see Section 5.3). This section is based on my publication at ICSE in 2024 [123]. The evaluation is designed to answer the respective questions from the GQM plan (see Section 5.5.1), i.e., Question **Q2.1**, Question **Q2.2**, and Question **Q2.3** to assess the performance of the approach. Further, we compare our approach ArCoTL with a baseline to answer Question **Q3.1** and Question **Q3.2**. Again, I base the evaluation on existing evaluation methods, e.g., by Hayes et al. [93], and on the guidelines by Shin [246].

5.5.4.1. Methodology

In the same fashion as for TLR between SAD and SAM in Section 5.5.3, we use the projects and the contained gold standard from the dataset (see Section 5.5.2) to evaluate our approach ArCoTL. We use the approach to recover the trace links and compare the results with the expected trace links from the gold standard. Furthermore, we use the metrics precision, recall, F_1-score, accuracy, specificity, and the Φ correlation as planned in the GQM plan (see Section 5.5.1).

Unfortunately, to the best of our knowledge, there are no existing approaches for this task to use as baseline approaches against which to compare. In theory, it is possible to adapt approaches for architecture recovery. This requires an approach that can handle all the benchmark projects, including different architecture styles like layered architecture and client-server or microservice architectures. Moreover, the goal of architecture recovery is to create an architecture model. While this can result in trace links between the original artifacts and the created architecture model, there is still no link to the actual existing SAM. An approach that uses architecture recovery, therefore, needs to then link the created SAM and the existing SAM, resulting in another TLR problem. If the resulting model is at the same abstraction level, it likely covers roughly the same components. In this case, using name similarity and maybe including some structural information can yield good results. However, there is no guarantee that the extracted SAM has the same abstraction level. For example, approaches like the ones by Kirschner et al. [132, 131] often create SAMs that are close to the implementation. This is understandable as such approaches use implementation as the main source of recovery. Creating abstractions is hard, and automatically finding the exact abstraction level for the recovery is an open research challenge [131]. In such cases, the linking between the SAMs is non-trivial. As developing an approach for TLR between different SAMs is not the goal of this work, and as this introduces potential threats to validity, we decided against the creation of a baseline that uses architecture recovery methods.

To still have a baseline to compare against, we adapt the baseline approach Simple Tracelink Discovery (STD) used for TLR between SAD to SAM (see Paragraph 5.5.3.1). The approach extracts n-grams, specifically unigrams, bigrams, and trigrams, from each model element and the class names together

Table 5.7.: Results of ArCoTL for TLR Between SAMs And Code Using the Metrics Precision (P), Recall (R), F_1-score (F_1), Accuracy (Acc), Specificity (Spec), And The Φ Coefficient (Φ)

Project	**P**	**R**	**F_1**	**Acc**	**Spec**	**Φ**
MediaStore	0.98	1.00	0.99	1.00	1.00	0.99
TeaStore	0.98	0.98	0.98	1.00	1.00	0.97
TEAMMATES	1.00	1.00	1.00	1.00	1.00	1.00
BigBlueButton	0.94	0.96	0.95	0.99	1.00	0.95
JabRef	1.00	1.00	1.00	1.00	1.00	1.00
Average	0.98	0.99	0.98	1.00	1.00	0.98
Weighted Avg.	0.99	0.99	0.99	1.00	1.00	0.99

with their paths. Then, it compares the n-grams from the code and the model using the normalized Levenshtein distance (see Section 2.3.4).

For each project, we optimize the parameters of STD, like the required similarity threshold, to create a trace link. While this setting is not realistic because users cannot know the most optimal configuration for a project beforehand, we want to compare ArCoTL with the most optimal results of the baseline approach.

In contrast to ArDoCo, no projects from the evaluation dataset were used and analyzed to develop ArCoTL to come up with its heuristics.

5.5.4.2. Results and Interpretation

Table 5.7 presents the results of ArCoTL for TLR between SAMs and code.

The precision values across all projects are exceptional, ranging from 94% to 100%. The average and weighted average precision are both approximately 98% and 99%, respectively, indicating that the approach is highly effective at correctly identifying true positive trace links with minimal false positives.

Similarly, the recall values are consistently high, ranging from 96% to 100%, with both the average and weighted average recall being 99%. This suggests that the approach is very effective at capturing most of the true links, resulting in very few false negatives.

Table 5.8.: Comparison Of The Average Results For TLR Between SAM And Code Of ArCoTL And The Baseline STD Using the Metrics Precision (P), Recall (R), F_1-score (F_1), Accuracy (Acc), Specificity (Spec), And The Φ Coefficient (Φ)

Approach	P	R	F_1	Acc	Spec	Φ
Average						
- Baseline	0.21	0.32	0.20	0.99	0.99	0.22
- ArCoTL	0.98	0.99	0.98	1.00	1.00	0.98
Weighted Average						
- Baseline	0.08	0.31	0.10	0.99	0.99	0.13
- ArCoTL	0.99	0.99	0.99	1.00	1.00	0.99

With such high precision and recall values, the F_1-score is correspondingly strong across all projects, ranging from 95% to 100%. The average and weighted average F_1-score are 98% and 99%, respectively, further affirming the approach's effectiveness.

Both metrics score perfectly or near-perfectly (mostly 100%) in terms of accuracy and specificity, underscoring the method's reliability and the minimal occurrence of incorrect links. As in the evaluation results of TLR between SAD and SAM (see Section 5.5.3.2), the results here are influenced by the ratio of possible trace links to actual trace links in the gold standard.

The values for the Φ correlation coefficient are also notably high (95% to 100%), indicating a strong positive association between the predicted and actual trace links.

To compare the results of ArCoTL to the baseline, Table 5.8 shows the results of ArCoTL and the results of the baseline approach STD. Detailed results for the baseline STD are displayed in the appendix in Table B.2.

ArCoTL has a significantly higher precision compared to the baseline, and the recall of the baseline is quite low, while ArCoTL achieves near-perfect recall. Due to the low scores for both precision and recall, the baseline's F_1-score is also very low.

Both approaches have high accuracy and specificity, indicating that they are both good at correctly identifying non-relevant links. Still, these values are influenced by the high number of true negatives, as the Φ shows. The baseline has a much lower Φ coefficient compared to ArCoTL.

Overall, ArCoTL outperforms the baseline by a significant margin across all evaluated metrics, especially in precision, recall, and F_1-score. This significance is confirmed by a one-sided Wilcoxon signed-rank test ($\alpha = 0.05$) on the F_1-score. The results also show that the problem is not trivial despite the excellent, near-perfect results of ArCoTL.

5.5.4.3. Discussion & Conclusion

The results suggest that ArCoTL is highly effective in recovering trace links between SAMs and code across different software projects. The consistently high precision and recall, combined with perfect accuracy and specificity, highlight that the approach can identify the correct links with great accuracy while minimizing both false positives and false negatives. The strong Φ values further support the robustness and reliability of the TLR approach.

According to the taxonomy by Hayes et al. [93], the results of ArCoTL are *excellent*. High results are expected as SAMs and code are comparably closely related. The approach can utilize many of the similar structures that can be found in both types of artifacts. Ultimately, the semantic gap is relatively small between these artifacts. Still, the semantic gap is big enough to be non-trivial, as the results of the baseline approach show.

The small semantic gap and the obtained results are also reasons to link SAD and code transitively via SAMs in our approach TransArC (see Section 5.4). To effectively create transitive trace links, it is paramount that the employed approaches perform well for their respective task. For the TLR between SAM and code, we can summarize that our approach, based on the evaluation results, can help to reduce the semantic gap.

Nevertheless, it is important to consider the context in which these results were obtained. I discuss potential threats to validity in Section 5.5.7.

5.5.5. Recovering Links Between Documentation And Code

In this section, I will evaluate the transitive approach TransArC for TLR between SAD and code using SAMs as bridging artifact (see Section 5.4). This section is based on my publication at ICSE in 2024 [123].

As with the evaluation of the two TLR approaches before, this evaluation is designed to answer the respective questions from the GQM plan (see Section 5.5.1), i.e., Question **Q2.1**, Question **Q2.2**, and Question **Q2.3** to assess the performance of the approach. Further, this evaluation compares TransArC with state-of-the-art approaches to answer Question **Q3.1** and Question **Q3.2**. Again, the evaluation uses existing evaluation methods, e.g., by Hayes et al. [93], and is based on the guidelines by Shin [246].

5.5.5.1. Methodology

In this evaluation, we again use the benchmark dataset (see Section 5.5.2) to evaluate our approach. This corresponds to the five projects MediaStore, TeaStore, TEAMMATES, BigBlueButton, and JabRef with their SAD and their code together with gold standards that contain the expected trace links for this TLR task. For code, we look into Java code and Bash scripts, as all these approaches are mainly Java-based.

With the gold standards, we can evaluate the trace links that TransArC found with the expected trace links. For measuring the performance, we use the metrics precision, recall, F_1-score, accuracy, specificity, and the Φ correlation as planned in the GQM plan (see Section 5.5.1). The results allow us to assess the effectiveness of the TLR approach in maintaining consistency and traceability between SAD and implementation.

For the comparison with other approaches (Question **Q3.1**), we look into several baseline approaches from the literature. As candidate approaches, we use three different approaches from the literature: TAROT by Gao et al. [77], FTLR by Hey et al. [97], and CodeBERT by Feng et al. [65]. All of these approaches provide a replication package that allows us to easily adapt them for our use case, as these approaches are for TLR between requirements and code. These approaches can handle natural language input and code, and as such, they can address our specific TLR problem. However, they target slightly different natural language artifact types (method documentation and requirements). The results of these approaches can be negatively influenced by the characteristics and the different abstraction levels of SADs compared to method documentation and requirements, and thus, the approaches might perform worse if applied to SADs. For example, the approaches are developed assuming that the name of a code entity is less likely to be directly mentioned in the text (i.e., the requirement). This causes the approaches to look for any clue that a sentence in the text is related to a code entity, requiring much less certainty. However, SADs regularly name architecture entities that are somehow represented in the code. SADs as documentation of the solution structure are usually semantically closer to code than requirements are to code. Accordingly, the assumptions of the approaches can affect the results. Most likely, the approaches achieve good recall but are less precise.

To add another baseline that can handle SADs and that had SADs in mind during the development, we adapt the ArDoCo approach for TLR between SAD and SAM by interpreting the code as a model.

We optimize the required parameters for each approach to yield the best possible results.

In the following, I briefly describe the baseline approaches.

TAROT by Gao et al. TAROT is an approach by Gao et al. [77] that uses so-called biterms. In texts, biterms refer to two terms within a sentence with a grammatical relationship. In code, biterms represent any combination of two terms within identifiers, and code comments are treated analogously to text. Consensual biterms are the intersection of both biterm sets. These consensual biterms are subsequently weighted based on their frequency and location. The approach combines these biterms with different IR models, including VSM, LSI, and JSD, to create candidate trace links. To potentially improve results, TAROT can be combined with the CLUSTER enhancing strategy (cf. [76]).

To apply TAROT to our scenario, we interpret each sentence of the SAD as a requirement. The outcome of TAROT is a similarity matrix between code files and sentences of the SAD. We use a threshold to filter this similarity matrix to generate trace links.

We evaluate each of TAROT's different IR methods, i.e., JSD, LSI, and VSM. Since we aim to compare TAROT's results with our approach, we try to find the best threshold for each project and each IR method. To achieve this, we utilize the threshold that maximizes the F_1-score concerning each project and IR method, thus obtaining the most favorable results from TAROT.

FTLR by Hey et al. Hey et al. [97] propose the FTLR approach, which leverages IR-based metrics to retrieve trace links. FTLR distinguishes itself by adopting a more fine-grained technique compared to other methods. In this approach, artifacts are broken down into smaller units: requirements are split into sentences, while code elements are represented by public method signatures, supplemented by the containing class name and related documentation comments.

The preprocessing of both artifact types involves steps such as stop word removal, lemmatization, and word length filtering. The preprocessed elements are then represented as bags of embeddings (BoEs), using pre-trained word embeddings from fastText [183].

Trace links are generated through a two-step process. First, fine-grained elements are mapped based on the WMD of their corresponding BoEs. Next, a threshold is applied to filter the resulting element-level trace links. A majority vote then creates class-level trace links by associating each class with the most frequently mapped requirements across its methods.

To apply FTLR to our scenario, in the same fashion as with TAROT, we interpret the sentences of the SAD as the sentences of requirements. We apply each of FTLR's different modes of operation and select the best-suited mode. In our evaluation, the best mode uses the WMD as the similarity measure, taking method comments into account. Further, we use the results of the best possible threshold for each project for our comparison.

CodeBERT by Feng et al. CodeBERT by Feng et al. [65] is an LLM trained on finding the most semantically related source code for a given natural language description. We use CodeBERT and fine-tune it to our TLR problem, following the TraceBERT (T-BERT) approach by Lin et al. [160].

To apply CodeBERT in our scenario, we fine-tune the CodeBERT language model [65] for the Java code search task of the CodeSearchNet dataset [103]. In this dataset, there are pairs consisting of Java methods and their corresponding method documentation. The training task for the LLM is to predict whether a given method documentation belongs to a method implementation. Although slightly different, this task can be interpreted as similar to linking sentences in SADs to their corresponding source code classes. To fine-tune the model, we adapted the code provided in the replication package of T-BERT [160], as it includes support for applying and evaluating CodeBERT models

Table 5.9.: Results of TransArC for TLR between SADs and code

Project	P	R	F_1	Acc	Spec	Φ
MediaStore	1.00	0.52	0.68	0.99	1.00	0.72
TeaStore	1.00	0.71	0.83	0.98	1.00	0.83
TEAMMATES	0.71	0.91	0.80	0.98	0.98	0.79
BigBlueButton	0.77	0.91	0.84	0.99	0.99	0.83
JabRef	0.89	1.00	0.94	0.96	0.94	0.92
Average	0.87	0.81	0.82	0.98	0.98	0.82
Weighted Avg.	0.81	0.94	0.87	0.97	0.96	0.85

to the TLR task. In particular, we use the SINGLE architecture with online negative sampling, which performed best on the CodeSearchNet dataset. After training on the CodeSearchNet dataset, we use the fine-tuned model to predict links between SAD and code.

ArDoCode To add a baseline that is more geared towards SADs, we adapt ArDoCo to create trace links between SADs and code, calling it ArDoCode. ArDoCode interprets the intermediate code model as a kind of SAM. In the same fashion as for ArCoTL and, thus, TransArC, the approach (recursively) resolves packages or paths to create trace links to the contained elements, i.e., the classes within each package or path.

5.5.5.2. Results and Interpretation

Table 5.9 shows the performance of TransArC in the evaluation.

Across the projects, the precision scores are generally high, with values ranging from 71% (TEAMMATES) to a perfect 100% (MediaStore and TeaStore). The high average Precision (87%) indicates that the TLR approach effectively minimizes false positives, ensuring that most of the identified mappings between SADs and code are correct.

The results show a broader range of recall values, from 52% (MediaStore) to 100% (JabRef). The overall average recall (0.81) suggests that while the approach performs well, there is some variation in its ability to capture all relevant mappings, with potential room for improvement, particularly in projects like MediaStore, where recall is lower.

The results of ArDoCo for TLR between SAD and SAM partly transfer to TransArC, including the special characteristics. The approach yields the lowest precision for TEAMMATES and the lowest recall for MediaStore. However, the difference between the precision for TEAMMATES and the precision for other projects is not as considerable.

The F_1-scores range from 68% (MediaStore) to 94% (JabRef), with an average of 82%. The variation in F_1-scores across projects suggests that while the approach is robust, it performs better in some contexts (e.g., JabRef) than others (e.g., MediaStore), possibly due to differences in the complexity or structure of the SADs and codebases.

Accuracy and specificity are consistently high across all projects, with values close to 100%, indicating that the approach is highly reliable in overall classification. The average Accuracy is 98%, reflecting the approach's strong overall performance. The specificity values complement the high precision, showing that the approach effectively does not misclassify irrelevant instances as relevant. However, in the same way as with the evaluation results of the other TLR tasks in Section 5.5.3.2 and Section 5.5.4.2, the results here are influenced by the ratio of possible trace links to actual trace links in the gold standard.

The ϕ-Metric, which might reflect a more sophisticated or composite performance measure, varies from 72% (MediaStore) to 92% (JabRef), with an average of 82%. This metric reinforces the general trend observed in the F_1-scores, highlighting that while the approach performs well overall, certain projects may benefit from further optimization.

The average and weighted average metrics suggest that the TLR approach performs well across different contexts. The slight differences between the average and weighted average metrics (e.g., Precision: 87% vs. 81%, Recall: 81% vs. 94%) imply that certain projects, likely those with larger or more complex SADs and codebases, are influencing the overall performance. This indicates that while the approach is generally robust, its effectiveness can vary depending on the project size and complexity.

Table 5.10 shows the average results of TransArC and the other approaches for comparison. Detailed results for the baselines can be found in the appendix in Section B.3. The performance of the approaches varies significantly across the

Table 5.10.: Comparison of the Average Results for TLR between SAD and Code Using the Metrics Precision (P), Recall (R), F_1-score (F_1), Accuracy (Acc), Specificity (Spec), and the Φ coefficient (Φ)

Approach	**P**	**R**	**F_1**	**Acc**	**Spec**	**Φ**
Average						
- TAROT	0.15	0.44	0.22	0.75	0.71	0.10
- FTLR	0.15	0.43	0.21	0.72	0.68	0.08
- CodeBERT	0.23	0.44	0.27	0.83	0.83	0.21
- ArDoCode	0.27	0.78	0.37	0.82	0.81	0.37
- TransArC	**0.87**	**0.81**	**0.82**	**0.98**	**0.98**	**0.82**
Weighted Average						
- TAROT	0.19	0.63	0.29	0.58	0.44	0.06
- FTLR	0.19	0.59	0.28	0.57	0.46	0.03
- CodeBERT	0.28	0.53	0.36	0.76	0.74	0.23
- ArDoCode	0.47	0.92	0.62	0.87	0.83	0.59
- TransArC	**0.81**	**0.94**	**0.87**	**0.97**	**0.96**	**0.85**

metrics. Overall, the TransArC approach demonstrates the best performance by a wide margin in both average and weighted average comparisons.

The approaches TAROT and FTLR show similar, relatively low performance across all metrics. They achieve low precision (15%) and recall (43%-44%), resulting in F_1-scores of around 21%-22%. Their Φ coefficients are particularly low (8%-10%), indicating poor alignment between predicted and actual links. Even in the weighted average evaluation, both approaches fail to reach satisfactory levels, with low precision (19%) and recall (59%-63%), leading to F_1-scores around 28%-29%.

CodeBERT, a model based on pre-trained transformers, outperforms TAROT and FTLR. It achieves a precision of 23%, a recall of 44%, and an F_1-score of 27%. The approach exhibits better accuracy (83%) and specificity (83%) as well. The weighted average shows improvement in recall (53%) and F_1-score (36%), indicating better handling of diverse traceability cases.

ArDoCode, with its more targeted handling of SADs, achieves considerably better performance than TAROT, FTLR, and CodeBERT. It achieves an F_1-score of 37% in the average evaluation and 62% in the weighted average. Its recall (78%) and accuracy (82%) are notable, but the standout feature is the Φ coefficient of 37% (average) and 59% (weighted average), indicating significantly better alignment.

TransArC consistently outperforms all other methods across the board. In the average evaluation, it achieves high precision (87%), recall (81%), and an F_1-score of 82%. Accuracy (98%) and specificity (98%) are near perfect, while the Φ coefficient (82%) indicates very strong alignment. Even in the weighted average, it remains dominant, with a precision of 81%, recall of 94%, and F_1-score of 87%, reinforcing its robustness and reliability.

To evaluate the effectiveness of our approach compared to baseline methods, we apply Wilcoxon's signed-rank test to analyze the significance of the observed differences in F_1-scores. Our results indicate that TransArC achieves statistically significant improvements over the baselines at the 0.05 significance level. These gains are likely due to the wider semantic gap that the baseline approaches struggle to bridge. Furthermore, TAROT, FTLR, and CodeBERT are optimized for scenarios that, while similar, differ from the specific context addressed by our approach. This specialization likely contributes to the superior performance of TransArC.

5.5.5.3. Discussion & Conclusion

The results highlight that the ArCoTL approach for TLR between SADs and code is effective, particularly in terms of precision and accuracy. However, variability in recall and F_1-scores across projects suggests that further refinements could be made to improve its ability to consistently capture all relevant mappings without sacrificing precision. Future work could focus on enhancing Recall for projects like MediaStore while maintaining the strong performance observed in other cases like JabRef.

The evaluation results reveal that TransArC is the most effective approach for TLR between SAD and code. It consistently achieves the highest scores in every metric, demonstrating both precision and recall, leading to strong overall performance. The combination of our two specialized approaches demonstrates significant effectiveness. This transitive approach, therefore, enhances the practical applicability of TLR (cf. [93]). A comparison between ArDoCode and TransArC suggests that using intermediate artifacts plays a crucial role in bridging the semantic gap.

ArDoCode is the next best, particularly due to its high recall and moderate F_1-score, making it a viable alternative for scenarios where recall is prioritized. In contrast, approaches like TAROT and FTLR fail to deliver competitive performance and are not recommended for scenarios requiring accurate and reliable TLR. CodeBERT, while outperforming the other two methods, still lags behind ArDoCode and TransArC, suggesting that further fine-tuning or domain-specific adaptations are necessary for transformer-based approaches to be competitive in this context.

However, the results also highlight a certain project dependency, influenced by the project's characteristics. These characteristics include the similarity and consistency of the names in the two artifacts. Moreover, projects may use similar terms corresponding to different entities, making it hard to link them correctly. As a result, these factors influence the findings and other potential threats to validity. As such, I discuss these in Section 5.5.7.

5.5.6. Exploratory Application of LLMs to Recover Links

Recent advancements in LLMs offer a transformative approach to the challenge of TLR due to their abilities to handle the complex and frequently unstructured nature of software artifacts. LLMs, with their deep understanding of language and context, demonstrate a high level of accuracy in processing and interpreting extensive volumes of unstructured text. Their ability to capture nuanced relationships between terms and concepts across different domains makes them particularly suited for TLR.

In this section, I want to apply LLMs in an exploratory study to assess their performance and to compare it to our approach ArDoCo. For this, I employ an end-to-end approach that integrates SADs and SAMs with an LLM. In this methodology, I prompt the LLM to generate the trace links between the provided artifacts. I anticipate that the LLM will respond with the corresponding trace links. This allows me to evaluate the LLM and compare the results with the results of the ArDoCo approach (see Section 5.5.3). I detail this approach in Section 5.5.6.1 and then proceed with its evaluation in Section 5.5.6.2.

5.5.6.1. Approach

To enhance the prompting process, we explore various prompting methods, including zero-shot prompting, few-shot prompting, and Chain-of-Thought prompting. Zero-shot prompting aims to use the model's capabilities of understanding the input and its context to solve the task. Few-shot prompting, with a limited number of examples, aims to fine-tune the model's understanding and response accuracy. For this, we utilize small example scenarios comprising SAD and SAM, along with the trace links between these artifacts. Chain-of-Thought prompting seeks to leverage the logical flow and reasoning capabilities of the LLM by providing a step-by-step thought process for establishing trace links. The Chain-of-Thought prompting extends the few-shot case by using the same scenarios and artifacts augmented with reasoning to explain why the trace links should be established.

Besides prompting, we also experimented with an embedding-based variant along with RAG. The embedding-based variant embeds the sentences of the SADs and the model elements of the SAMs (see also Section 2.3.3). These embeddings are then compared using cosine similarity (cf. Section 2.3.4. The approach creates a trace link if the similarity is above a given threshold.

The approach uses a dynamic threshold that it calculates based on the similarity scores between the sentences of the SAD S and the model elements of the SAM M. For each model element $m_i \in M$, the approach first determines the sentences $s_j \in S$ with the highest and the lowest similarity scores among all sentences. Then, the approach calculates the threshold by applying a constant factor c on the difference between the lowest and highest similarity, normalizing it. The following equations show the calculation:

$$\begin{aligned} \mathrm{sim}_{max}(m_i) &= \max_{j \in |S|}(\mathrm{similarity}(m_i, s_j)) && (1)\\ \mathrm{sim}_{min}(m_i) &= \min_{j \in |S|}(\mathrm{similarity}(m_i, s_j)) && (2)\\ t_{m_i} &= \mathrm{sim}_{max}(m_i, s_j) - (\mathrm{sim}_{max}(m_i, s_j) - \mathrm{sim}_{min}(m_i, s_j)) \times c && (3) \end{aligned}$$

In the RAG variant, there is the possibility to adapt the aforementioned prompting approaches to only include a subset of sentences of the SAD. For this, the approach first creates an embedding for each sentence in the SAD (see also Section 2.3.3). The approach then uses these embeddings to retrieve the most relevant sentences for a model element based on the cosine similarity and the dynamic threshold. For each model element, the approach

Table 5.11.: Weighted Average Results for TLR using the LLM-based approaches on the benchmark dataset.

Approach	P	R	F_1
ArDoCo	**0.83**	0.82	**0.80**
Zero-shot	0.34	0.38	0.35
Few-shot	0.38	0.60	0.43
Chain-of-Thought	0.56	0.56	0.53
Embedding	0.48	0.56	0.48
RAG + Zero-shot	0.44	0.85	0.55
RAG + Few-shot	0.34	**0.90**	0.47
RAG + CoT	0.52	**0.90**	0.64

then provides these most relevant sentences to the LLM as solution space, reducing it from all sentences to only these relevant ones. There are again the three prompting strategies zero-shot, few-shot, and Chain-of-Thought.

5.5.6.2. Evaluation

In our experiments, we evaluate the performance of each variant and prompting method. For this, we use the same dataset described in Section 5.5.2 and apply the metrics precision (P), recall (R), and F_1-score (F_1). Using these metrics, we compare the results of these exploratory LLM-based approaches with the ArDoCo approach.

We also experimented with OpenAI's GPT-3.5 and GPT-4. As GPT-4 performs better in our experiments, I only report the results using GPT-4 as LLM.

Table 5.11 shows the weighted average evaluation results. Overall, the results of the LLM-based approaches are worse than the ArDoCo baseline. Chain-of-Thought prompting outperforms the two other prompting techniques with and without RAG.

The embedding-based variants perform better than the variants that directly use the LLM. This might indicate that the LLM cannot focus clearly enough when presented with too many options. Yet, the semantic information to relate the two artifacts seems to be, to a certain degree, present within the LLM.

Examining the prompting techniques, the results show that adding examples helps keep the LLM on track. Chain-of-Thought prompting further guides the LLM and improves the results.

Overall, the results when applying LLMs for TLR are promising for an exploratory study but require further refinement. These simple LLM-based approaches could not consistently outperform the baseline approach, ArDoCo. Further research is necessary to determine the exact potentials and limitations of LLMs and to identify the best ways to apply them. This requires more thorough investigations of the different LLMs, LLM-based approaches, and techniques. A hybrid approach combining LLMs with other methodologies could be particularly beneficial.

One significant issue encountered during evaluations is the maximum token limit an LLM can process. In the used GPT-4 version, this limit is 8192 tokens, which restricts both the input size and the number of solutions the LLM can return. For larger projects, this constraint can render LLMs impractical. Developing and evaluating more sophisticated techniques to handle situations with large amounts of input and output data is, therefore, essential in future work.

In the experiments, we used the LLM GPT-4 as it is one of the biggest LLMs and is considered state-of-the-art. However, there are still other LLMs that can be applied to this problem and might improve results. For example, using LLMs that are specialized in a certain domain or for a certain use case might affect the outcomes. Additionally, we have not yet explored fine-tuning or other similar approaches and techniques.

5.5.7. Threats To Validity

While the evaluation demonstrated the approaches' advantages and capabilities, it is important to consider the context in which these results were obtained. For example, the approaches are based on the assumption that the SADs are written in English. The pre-processing and many heuristics have to be exchanged to adapt them to other languages. Additionally, the approaches assume that the artifacts use similar and unique naming. With dissimilar or ambiguous names, the approach fails to consistently and accurately recover trace links.

Besides these assumptions, there are threats to the validity of the evaluation. I address the potential threats to validity in the following, guided by the framework established by Runeson and Höst [235]. Additionally, for the experiments with LLMs, I also consider the discussion of Sallou et al. [237] that focuses particularly on threats of using LLMs in software engineering.

Construct Validity

There are several threats to construct validity. We applied a common experimental design to mitigate these and employed widely accepted metrics. Still, there is a potential bias in the selection of projects. We mostly selected previously studied projects. To reduce the bias, we selected projects from varying domains with different characteristics, including size, architectural styles, architectural patterns, and the ratio of documentation to lines of code. These steps were intended to ensure that our results are both robust and representative. However, certain dataset characteristics, such as the low proportion of trace links in the evaluation of the TLR approaches, likely influenced the observed high specificity. In environments with different data distributions, the approaches' performance might vary, potentially affecting their relative strengths. This variability underscores the need for caution when generalizing our findings beyond the specific contexts studied. The baseline approaches are either basic or not created for the specific use case, limiting the comparison as targeted approaches are likely more fine-tuned to the problem, leading to better performances.

One additional threat arises from the fact that two of the evaluation projects, MediaStore, and TEAMMATES, were known and used to create the approach for TLR between SAD and SAM. While I tried to generalize from these projects and tried to avoid overfitting the approach on these projects, there are threats to validity. To balance this, most projects (60%) in the evaluation were not involved in creating the approaches. Moreover, more than half (i.e., 55%) of the trace links in the gold standard are not from these two projects. Additionally, these two projects are among the worst-performing approaches, and the evaluation results are worse than average (see Table 5.5). This indicates that there was likely no or only limited (over-) fitting on these approaches.

In the evaluation of ArDoCo for TLR between SAD and SAM, some baseline approaches were only evaluated on three of the five projects in a pre-study. The reason is the required effort to adapt and optimize the approaches for every project. Due to their bad performance on these three projects, we decided to drop these approaches. However, it is still possible that the approaches perform differently and perform excellently on the other two projects (BigBlueButton and JabRef). There is no indication of this, but this still threatens validity.

For the LLMs, there is a myriad of parameters that can be tuned. We perform basic tuning in the evaluation but do not delve deep into it. Other or more elaborate tuning, e.g., of the prompts or the overall input handling, might affect the results.

Internal Validity
In terms of internal validity, there can be several forms of bias and threats for the several evaluations that arise from the possibility that factors other than those we intended to study could influence the evaluation. Additionally, there is a risk of misinterpreting the causes of certain results, leading to incorrect conclusions. To mitigate these threats, we adhered to established practices designed to minimize bias, including thorough cross-checking and validating our interpretations. Nonetheless, the evaluation used open-source projects, and it is important to recognize that open-source projects can vary significantly in terms of code quality, documentation, and consistency. This variability introduces noise and errors into the data, which could distort the evaluation process and lead to either inflated or deflated performance metrics.

External Validity
The research design also faces threats to external validity. There can be an issue because some evaluated projects are academic and designed to mimic real applications, but they may contain differences that influence generalizability. Further, while the dataset includes projects of varying domains and sizes, it does not fully represent the spectrum of potential application scenarios. Certain project types or characteristics may be overrepresented or underrepresented, potentially skewing the results and limiting the applicability of the findings to other projects or real-world scenarios. Additionally, while the Wilcoxon signed-rank test supports the statistical significance of the difference in F_1-score, it is a non-parametric test and may not account for all forms of variability in the data. To confidently generalize these findings, further tests and analyses on larger or more varied datasets are necessary.

Regarding SAMs and approaches that target SAMs, by focusing exclusively on (structural) component-based architecture models, we may limit the generalizability of the findings to other architectural paradigms or views. Although this view is commonly employed in architectural studies according to Tian et al. [268], results could differ if, for example, more logical descriptions or views of the architecture are used, or if there are greater variations in abstraction levels.

In the evaluation of TLR that focuses on code as one artifact, we only looked at Java projects. While the approaches use intermediate artifacts to abstract from the programming language, other programming languages might have other paradigms or common development patterns (e.g., how projects are typically structured) that can influence the project. Consequently, there is a certain threat to validity regarding the generalizability to projects that use other programming languages.

Lastly, while the self-developed baselines for TLR are simple and only allow limited comparison, they still allow us to show the advantages of our approaches against basic approaches. More importantly, they provide further insights and perspectives on the problem domain.

For the experiments with LLMs, further threats exist. One threat is the fact that we only used one or two LLMs. Other LLMs might return distinct results affecting the overall outcome of the evaluation.

One other particular threat is the use of OpenAI's closed-source models for evaluation. Since the training data of the models is unknown, we cannot assure the absence of data leakage.

Moreover, the non-determinism of the LLMs issues further threats. To mitigate this, we reduce the randomness of the LLMs responses by setting the temperature to 0.

Reliability

We followed rigorous approaches to construct the required gold standards to ensure reliability. These steps were intended to ensure the consistency and repeatability of our findings. For example, there were at least two researchers involved in independently generating these gold standards in order to reduce the risk of individual bias. Discrepancies were resolved through comparison and discussion. However, it is crucial to recognize that potential biases may still exist, such as in selecting benchmarks or varying interpretations of artifacts by different researchers. These factors can impact reliability to some extent.

For the prompts used during the evaluation of LLMs, we base them on common techniques and methods from literature to improve reliability.

Conclusion

Despite the significant steps we have taken to address potential threats to validity, certain limitations remain. Continuous efforts to refine and validate the methodology are essential to enhance and confirm our findings and further enhance the reliability of the study. Future research should focus on exploring a broader range of datasets and methodological approaches, which will help to further validate and refine the results of this study.

6. Detecting Inconsistencies in Software Architecture Documentation

The Traceability Link Recovery (TLR) approaches presented in previous chapters (cf. Chapter 5) establish connections between segments of the artifact that address the same entity. For instance, in the TLR approach ArDoCo (cf Section 5.2), components and similar software architecture entities from the Software Architecture Models (SAMs) are linked to sentences where they are mentioned in the Software Architecture Documentations (SADs). With these links, we can pinpoint intra-systemic structural design decisions (see the taxonomy in Chapter 4), e.g., about components or interfaces. It is important to ensure that the information across different artifacts remains consistent. Therefore, leveraging trace links between artifacts, which point to existence decisions, allows us to detect inconsistencies by examining matching or conflicting information regarding the existence of entities.

In this chapter, I present the extension of the approach ArDoCo (Architecture Documentation Consistency) to detect inconsistencies between SADs and SAMs. As such, this chapter aims to answer **RQ 4**. The chapter is based on my publications at ICSA 2023 [124].

I focus on two kinds of inconsistencies that I call Unmentioned Model Elements (UMEs) and Missing Model Elements (MMEs) to examine the consistency of such intra-systemic structural design decisions.

UMEs are inconsistencies where there is a software architecture entity in the model (i.e., the SAM) that cannot be found in the text (i.e., the SAD). One example is a *Cache* component that was later added to the model but not added to the documentation. Reasoning and details about the cache component, such as perceived benefits, expected workloads, planned configurations, or similar, might not persist, especially if the involved developers leave.

In the other direction, MMEs are inconsistencies where a software architecture entity is mentioned in the text but cannot be found in the model. Several situations can naturally lead to such inconsistencies. For instance, in prescriptive documentation, there may be entities that are planned but not yet implemented in the model. Another common situation occurs when the model is updated and evolves, resulting in removing a previously existing entity from the model without corresponding updates in the text.

Additionally, both kinds of inconsistencies may emerge when different teams or individuals maintain the model and the documentation, leading to misalignment. Ensuring synchronization between the model and the documentation can be particularly challenging in fast-paced development environments, where changes are frequent and rapid. Notifying developers about these inconsistencies allows them to resolve potential discrepancies and maintain the integrity and accuracy of both the model and the documentation.

To detect and report these inconsistencies, we use and extend ArDoCo for TLR between SAD and SAM as shown in the solution architecture overview in Figure 6.1. The left part displays the TLR, while the right part shows the subsequent ID that utilizes the results from ArDoCo. The approach leverages the trace links and intermediate results, particularly the *RecommendedInstances*.

The goal of the approach is to identify inconsistencies and report them to the user. As previously discussed, not every detected inconsistency is inherently problematic [197], but the user needs to be aware of them to assess their severity. After inconsistencies are reported, the user can then decide whether they need or want to address them. Consequently, the approach presents the detected inconsistencies to the user.

Importantly, the approach is designed to detect inconsistencies and report them to the user for further decisions and potential resolution. Currently, it does not support automatic repair or resolution of these inconsistencies.

In Section 6.1, I will present details on the approach for detecting UMEs. Subsequently, the details of the approach for detecting MMEs are provided in Section 6.2. The limitations of the ArDoCo approach for inconsistency detection are discussed in Section 6.3. Section 6.4 then focuses on the evaluation to answer the research questions concerning UMEs (**RQ 4.1**), MMEs (**RQ 4.2**), and the exploratory application of LLMs for them (**RQ 4.3**)

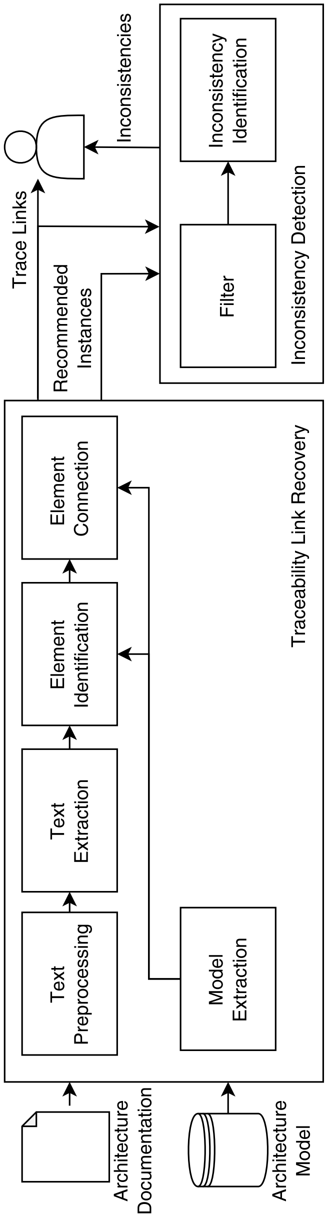


Figure 6.1.: Overview of the ArDoCo approach for inconsistency detection, by Keim et al. [124]

6.1. Detecting Unmentioned Model Elements

As introduced, Unmentioned Model Elements are inconsistencies where there is an entity in the SAM that is not documented in the SAD. The idea to detect this kind of inconsistency using TLR is straightforward. We can interpret the definition of UMEs as the necessity of each model entity to be mentioned at least once in the text. This means that there needs to be at least one trace link and, thus, the absence of trace links for a given model entity indicates a UME.

To detect UMEs between SAD and SAM, the approach counts the number of found trace links for each model entity. If the number of trace links for a certain model element is below a given threshold (default: one), the approach reports a UME.

To allow flexibility, there are several configuration options.

First, the users can configure the preferred types of the metamodel that should be considered. For example, some users might be interested in only considering components and/or interfaces. Per default, the approach looks at *Components* for UML and at *BasicComponents* and *CompositeComponents* for PCM (cf. Section 2.1).

Second, the users can configure the minimum number of mentions, i.e., trace links. This allows the users to adapt the required documentation for each entity as they see fit or require.

Third, the users can add a whitelist of entities that should not be detected as inconsistencies. This can be handy, for example, when certain dummy or fine-grained components are present in the model for demonstration purposes but do not need any documentation. Moreover, users can exclude components that are only present for test purposes. The whitelisting supports regular expressions to allow, for example, to name prefixes or suffixes.

6.2. Detecting Missing Model Elements

Missing Model Elements are inconsistencies where some part(s) of the text mention an architecturally relevant entity that cannot be found in the model. If an architecturally relevant entity of the text is found in the model, a trace link should exist between the corresponding parts. Consequently, the approach needs to look at architecturally relevant entities in the text and needs to figure out if there is no trace link that connects the entity to the model.

To detect MMEs, the approach can utilize *RecommendedInstances* as they represent elements in the text that should also be elements in the model. If a *RecommendedInstance* is not traced to the model M, i.e., no trace link exists to a model element $m \in \mathrm{M}$, there are possible inconsistencies regarding MMEs. Formally, this can be defined as follows:

$$\begin{aligned} \mathrm{TLs} &\subseteq \{(r, m) \mid r \in \mathrm{RIs}, m \in \mathrm{M}\} \\ \mathrm{MMEs} &= \{r_i \mid r_i \in \mathrm{RIs} \land \nexists (r_i, m) \in \mathrm{TLs}\} \end{aligned}$$

With this definition and idea, it is easy to detect MMEs: Simply return those *RecommendedInstances* that could not be linked to any model element. However, one difficulty of this approach lies within the paradigm of ArDoCo to increase recall by not discarding possible solutions early. This is a reasonable strategy for TLR because the *Element Connection* step (see Section 5.2.6) does not further use *RecommendedInstances* that cannot be linked. Yet, in the case of detecting MMEs, the approach needs to look at exactly those and the paradigm results in a high number of unconnected *RecommendedInstances.* At the same time, there is a high chance that some unconnected *RecommendedInstances* are not architecturally relevant entities. While the paradigm allows achieving high recall, it can result in low precision. Although recall is more important to not miss any inconsistencies and as the underlying assumption is that a human checks on the reported inconsistencies, a low precision can decrease acceptance for the tool. Balancing the two interests is a key challenge for this approach. As a consequence, the approach includes a filter step to remove *RecommendedInstances* that are unlikely model elements to increase precision. The approach consecutively uses the following three different heuristics: a *threshold filter*, an *occurrence filter*, and a *filter for unwanted words.*

The *threshold filter* evaluates each *RecommendedInstance* based on its confidence value. This filter removes *RecommendedInstances* with low confidence in being model elements, as determined by heuristics (see Section 5.2.5). While a fixed minimum confidence threshold can be set, determining the optimal threshold is challenging and often project-specific.

To address this, the filter offers a dynamic threshold mode. In this mode, the filter calculates dynamic threshold values by multiplying the highest confidence value by a factor $f \leq 1.0$, with a default value of $f = 0.7$ based on preliminary studies. These preliminary studies are made with the projects MediaStore and TEAMMATES, similar to my procedure for TLR between SAD and SAM (see Section 5.2). The other approaches from the benchmark dataset (see also Section 5.5.2) for the evaluation were not considered during the development of the approach.

In addition to the overall confidence of a *RecommendedInstance*, the threshold filter also examines its confidence concerning its name and type. It filters out *RecommendedInstances* with low confidence by comparing the confidence levels for name and type in two scenarios. In the most common scenario, the *RecommendedInstance* needs decent confidence values (default: >0.3) for both name and type. In the second scenario, the *RecommendedInstance* must have a high confidence value (default: > 0.8) for either name or type. In this latter case, it is assumed that the high confidence indicates the *RecommendedInstance* is likely a model element. The threshold values for these two scenarios are independent of the dynamic threshold value previously mentioned.

The *occurrence filter* eliminates *RecommendedInstances* that occur infrequently in the text. We assume that less frequent *RecommendedInstances* are less important and less likely to be architecturally relevant entities. Therefore, these entities are less likely to be necessary in the SAM. Consequently, the filter removes all *RecommendedInstances* that appear fewer than a specified number of times, with a default threshold set to 2 occurrences.

The third filter, the *filter for unwanted words*, evaluates *RecommendedInstances* based on their constituent words and removes those containing undesired terms. The list of undesired words includes terms that refer to common computer science entities or domain-specific terms. These terms may appear similar to actual model elements in the text and are difficult to distinguish. To address this, the filter uses two blacklists: a general blacklist and a domain-specific blacklist.

The general blacklist consists of commonly used software engineering terms. These terms frequently appear in SADs and resemble named model entities but often are not actual entities. Examples include programming languages (e.g., “Java”), technologies (e.g., “servlet”), and common abbreviations (e.g., “CPU”). The general blacklist that was used for the evaluation can be found in the appendix in Section A.1.

The project-specific blacklist includes words and terms specific to the project's domain that do not represent model entities. This might include technologies and tools used in the project, such as MongoDB, Kafka, or WebRTC. While the general blacklist is intended for universal application across projects, the project-specific blacklist must be customized for each project. Example project-specific blacklists that were used for the evaluation can be found in the appendix in Section A.1. Adding a few targeted terms can significantly improve the performance of MME detection.

Using blacklists is a straightforward method to address this complex problem and enhance results. However, a major drawback is the manual creation and maintenance required for these blacklists. We believe the effort to create and maintain these lists is reasonable. Users can opt not to create project-specific blacklists but can still reduce the number of false positives, thereby improving the effectiveness of our approach. In future work, we plan to enhance semantic analyses to minimize the reliance on blacklists.

After the application of these filters, the remaining *RecommendedInstances* are reported as inconsistencies. The reporting is done on a sentence level, where the inconsistencies of each sentence are collectively presented to the user. In my view, reporting on sentence level makes it easy for the user to grasp the context. At the same time, this eases comparisons with competing approaches and baselines.

6.3. Limitations

The ArDoCo approach for inconsistency detection is subject to certain limitations stemming from its underlying assumptions and design decisions. One key limitation is the assumption that the SAD is written in English. While the TLR component of the approach is language-dependent, the ID component is mostly language-independent. Nevertheless, changing the language can affect the accuracy and reliability of the results.

Another key assumption is that every sentence in the SAD should be traceable to at least one model element within the SAM. This reflects a widely accepted practice in TLR, where each trace artifact is typically linked to a corresponding element (cf. [93, 40, 97]). Automated traceability approaches often aim to identify the most appropriate targets for each source artifact type, such as mapping a requirement to its most relevant classes. Since our inconsistency detection approach relies heavily on trace links between sentences and model elements, we align our methodology with established TLR practices. While this assumption may slightly overestimate the significance of individual sentences, it is a reasonable approximation to achieve the desired goal.

6.4. Evaluation

In this chapter, I tackled the detection of inconsistencies between SADs and SAMs by presenting two approaches. Both approaches use TLR approaches and their final and intermediate results, i.e., identified architecturally relevant entities and trace links. The first approach identifies Unmentioned Model Elements by checking if a model element has trace links to the documentation. The second approach identifies Missing Model Elements by analyzing if an element that is mentioned in the SAD is linked to the SAM.

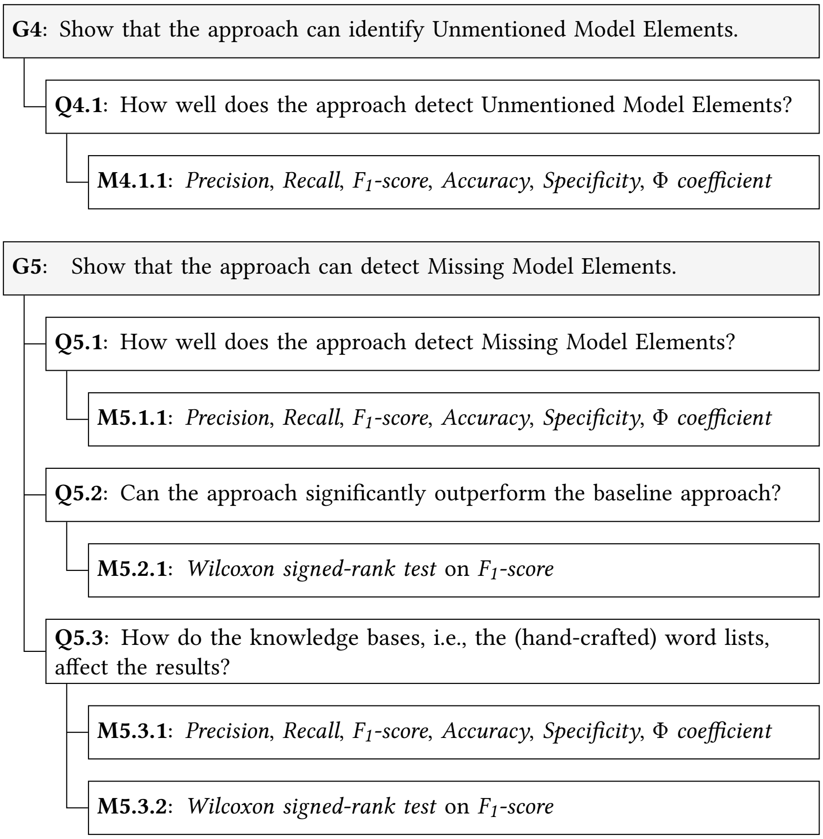
In the following, I first present my evaluation plan with the help of a GQM plan in Section 6.4.1. In Section 6.4.2, I evaluate the detection of UMEs and answer **RQ 4.1** before I evaluate the approach for detecting MMEs in Section 6.4.3 for **RQ 4.2**. I explore the application of LLMs for inconsistency detection in Section 6.4.4 and answer **RQ 4.3** before I end the evaluation of the approaches for ID with a discussion about threats to validity in Section 6.4.5.

6.4.1. Evaluation Plan

For this evaluation, I again base the evaluation on the GQM approach by Basili et al. [15]. For each contribution regarding ID, I define goals and derive corresponding questions for each goal to examine the fulfillment of the goals. I then use several metrics (see also Section 2.8) to answer each question.

Figure 6.2 shows the goal, questions, and metrics for the ID. The major goals are to show the applicability and performance of the ID when detecting MMEs and UMEs, respectively. The questions and metrics represent these goals.

For UMEs, I assess the performance (cf. Question **Q4.1**). Unfortunately, to the best of my knowledge, there is no existing work and no simple baseline to compare the results to. As before, I compare the results with gold standards



- **G4**: Show that the approach can identify Unmentioned Model Elements.
  - **Q4.1**: How well does the approach detect Unmentioned Model Elements?
    - **M4.1.1**: *Precision*, *Recall*, *F_1-score*, *Accuracy*, *Specificity*, *Φ coefficient*
- **G5**: Show that the approach can detect Missing Model Elements.
  - **Q5.1**: How well does the approach detect Missing Model Elements?
    - **M5.1.1**: *Precision*, *Recall*, *F_1-score*, *Accuracy*, *Specificity*, *Φ coefficient*
  - **Q5.2**: Can the approach significantly outperform the baseline approach?
    - **M5.2.1**: *Wilcoxon signed-rank test* on *F_1-score*
  - **Q5.3**: How do the knowledge bases, i.e., the (hand-crafted) word lists, affect the results?
    - **M5.3.1**: *Precision*, *Recall*, *F_1-score*, *Accuracy*, *Specificity*, *Φ coefficient*
    - **M5.3.2**: *Wilcoxon signed-rank test* on *F_1-score*

Figure 6.2.: Goals, Questions, and Metrics for Evaluating the Approaches for Inconsistency Detection

and calculate the performance using the metrics precision, recall, F_1-score, accuracy, specificity, and the Φ coefficient.

For MMEs, I first evaluate the performance (Question **Q5.1**) before comparing the performance to baseline approaches (Question **Q5.2**). I compare the results of the approach and the baseline with the expected results from a gold standard with the previously mentioned metrics (Metric **M5.1.1**). To assess if the results are significant, I use the Wilcoxon signed-rank test on the F_1-score (Metric **M5.2.1**).

The approach for detecting MMEs uses filters with filter lists, I evaluate the effect of these lists on the results (Question **Q5.3**). I use the same metrics as before (see Metric **M5.3.1**), and I check if the differences are significant using the Wilcoxon signed-rank test on the F_1-score (Metric **M5.3.2**).

6.4.2. Unmentioned Model Elements

UMEs refer to inconsistencies where a software architecture entity present in the SAM is not documented in the SAD. The presented approach uses TLR and the resulting trace links to detect these inconsistencies (see Section 6.1).

In the following evaluation, I realize Goal **G4** about showing that this approach can actually identify UMEs by answering Question **Q4.1**. First, I describe the methodology in Section 6.4.2.1 before presenting the results in Section 6.4.2.2 and discussing them in Section 6.4.2.3.

6.4.2.1. Methodology

For evaluating the ID approach to identify UMEs (see Section 6.1), I utilize the same projects and dataset as for the evaluation of the TLR approaches that I describe in Section 5.5.2.

To construct the required gold standards, we manually compare the documentation and the model to identify undocumented model elements. Similar to the process used for TLR, multiple researchers independently created the gold standards. They then merged their versions after discussing and resolving any discrepancies. The annotators detected four UMEs for MediaStore, five for TeaStore, and one each for BigBlueButton and JabRef. For TEAMMATES, the researchers could not identify UMEs.

With the gold standards, we can evaluate the approach for detecting UMEs by comparing the results of the approach with the expected results. We then can calculate the metrics precision, recall, F_1-score, accuracy, specificity, and the Φ coefficient (see also Metric **M4.1.1**) to answer Question **Q4.1**.

Table 6.1.: Evaluation Results For Detecting UMEs With The Metrics Precision (P), Recall (R), F_1-score (F_1), Accuracy (Acc), Specificity (Spec), And The Φ Coefficient.

Project	**P**	**R**	**F_1**	**Acc**	**Spec**	**Φ**
MediaStore	0.67	1.00	0.80	0.88	0.83	0.75
TeaStore	1.00	1.00	1.00	1.00	1.00	1.00
BigBlueButton	1.00	1.00	1.00	1.00	1.00	1.00
JabRef	1.00	1.00	1.00	1.00	1.00	1.00
Average	0.92	1.00	0.95	0.98	0.97	0.94
Weighted Avg.	0.88	1.00	0.93	0.95	0.94	0.91

6.4.2.2. Results and Interpretation

Table 6.1 provides the evaluation results for the four projects.

For the TeaStore, BigBlueButton, and JabRef projects, the results demonstrate a perfect score of 100% across all evaluation metrics, indicating flawless performance in detecting UMEs. These outcomes highlight the approach's exceptional effectiveness in these cases, consistently identifying all UMEs without any errors.

In the case of the MediaStore project, although all UMEs are successfully detected, resulting in perfect recall. Yet, only two-thirds of the identified inconsistencies are correct, resulting in a precision of 67%. This lower precision negatively impacts the F_1-score, accuracy, specificity, and Φ coefficient.

The average precision across all projects is 92%. While this represents a high precision, it is slightly diminished by the results from MediaStore. On average, the results are robust and the Φ coefficient indicates a strong positive correlation.

6.4.2.3. Discussion & Conclusion

In summary, the approach to detect UMEs across various software projects demonstrates a strong overall performance. It consistently achieves perfect recall and near-perfect scores for most metrics across all projects, with the exception of MediaStore. The consistently high average and weighted average scores across all metrics underscore the effectiveness of the method, particularly in its ability to reliably identify all UMEs. However, there remains some room for improvement in terms of precision.

The effectiveness of this approach is closely tied to the performance of the TLR approach. In cases where trace links are inaccurately identified, the detection process may overlook certain inconsistencies. Furthermore, if trace links are missing, the approach may falsely report UMEs. As a result, the recall of the TLR approach directly influences the precision of the UME detection method, while the precision of the TLR approach impacts the recall of this method.

Overall, these interdependencies highlight the critical role of accurate TLR in ensuring the reliability of inconsistency detection. Improvements in the precision and recall of the TLR approach could lead to further enhancements in the performance of the UME detection approach, particularly in more complex projects where the risk of inaccuracies is higher.

6.4.3. Missing Model Elements

MMEs are inconsistencies where a software architecture entity is mentioned in the text but cannot be found in the model. The approach to detect these inconsistencies uses trace links and *RecommendedInstances* from the TLR part (see Section 6.2). In this section, we evaluate this approach to show its ability to detect MMEs (Goal **G5**). For this, Section 6.4.3.1 presents the methodology to tackle the questions to achieve the goal. The evaluation results are presented in Section 6.4.3.2 before I discuss them in Section 6.4.3.3.

6.4.3.1. Methodology

For this evaluation, a dataset containing MMEs is essential. However, to the best of our knowledge, no such dataset currently exists. Additionally, the benchmark dataset previously used for evaluating our TLR approach and for the UMEs does not include MMEs. Consequently, we decided to artificially create a dataset based on the existing benchmark dataset.

To simulate MMEs, we systematically removed single model elements from the SAM while preserving the SAD. The trace links defined in the gold standard for TLR then indicate where inconsistencies related to the removed model elements are expected to occur. This setup allows us to leverage the existing gold standard and automate the evaluation process. However, this is not fully realistic and may create models where key components are missing. This creates threats to validity that I discuss in Section 6.4.5.

The evaluation is performed through multiple runs. In each run, a SAM is provided with all model elements except for one, which is deliberately removed for that run. This process is repeated until each model element has been removed once. For each run, the performance is evaluated by counting the true positives, false positives, false negatives, and true negatives.

Multiple runs are performed for each project, so the results are aggregated at the project level, and metrics are calculated using the micro-average approach. Specifically, all true positives, false positives, false negatives, and true negatives across all runs are accumulated, providing the basis for calculating the evaluation metrics (see Metric **M5.1.1**).

Filter configurations We evaluate three filter configurations. The first configuration uses both filter lists for unwanted words (see also Section 6.2): the general blacklist, containing common software engineering-related terms, and the domain-specific one. The latter contains an average of 10 words for our projects in the dataset, ranging from 5 to 22 words (median: 7). The second configuration only uses the general blacklist. The third configuration then does not use any filter list at all. This way, we can assess the influence of the filter lists on the performance (Question **Q5.3**). The concrete lists are shown in the appendix in Section A.1.

Baseline We need a baseline approach to compare our approach further and answer Question **Q5.2**. To the best of our knowledge, there is no existing approach to detect MMEs. Therefore, we provide a baseline for this task to establish a lower bound. The baseline approach operates under the assumption that each sentence in the textual documentation corresponds to a specific model element. This assumption is grounded in the principle that no unnecessary sentences exist in SADs. Consequently, each sentence should ideally be associated with at least one trace link. Using this assumption, the baseline approach identifies sentences that do not contain any trace link and reports them as inconsistencies related to MMEs.

The challenging part of this task lies in identifying meaningful sentences that are relevant to the model. Some sentences describe architectural properties

Table 6.2.: Evaluation Results For Detecting MMEs Using **All** Filters. The Metrics Precision (P), Recall (R), F_1-score (F_1), Accuracy (Acc), Specificity (Spec), And The Φ Coefficient Are Displayed.

Project	**P**	**R**	**F_1**	**Acc**	**Spec**	**Φ**
MediaStore	0.13	0.79	0.22	0.70	0.68	0.23
TeaStore	0.95	0.70	0.81	0.98	1.00	0.81
TEAMMATES	0.15	0.75	0.25	0.85	0.86	0.29
BigBlueButton	0.67	0.46	0.55	0.96	0.99	0.54
JabRef	1.00	0.44	0.62	0.87	1.00	0.62
Average	0.58	0.63	0.49	0.87	0.90	0.49
Weighted Avg.	0.51	0.63	0.45	0.88	0.90	0.46

that cannot be directly represented in the model, such as design principles or technological choices (cf. [146, 125]), or provide additional context, scope, or clarifications. Moreover, the assumption does not account for cases where a single sentence references multiple model elements. If one of these elements is indeed missing from the model, the baseline approach may fail to detect the inconsistency.

While these edge cases typically occur infrequently, we anticipate that the baseline approach will achieve high recall, thereby serving as a reliable lower bound for evaluation. The performance gap between the baseline and our proposed approach highlights our approach's enhanced capabilities in this context.

6.4.3.2. Results and Interpretation

Table 6.2 presents the evaluation results for assessing the performance of the approach to detect MMEs to answer Question **Q5.1**. The results vary significantly across the projects.

TeaStore and JabRef achieve very high precision (95% and 100%, respectively), indicating very few false positives. At the same time, MediaStore and TEAMMATES show low precision (13% and 15%), suggesting many false positives were detected. BigBlueButton shows moderate precision at 67%. The average precision across all projects is 58%, while the weighted average is slightly lower at 51%.

Regarding recall, MediaStore, TEAMMATES, and TeaStore show high recall (79%, 75%, and 70%, respectively), indicating a high sensitivity in detecting MMEs. BigBlueButton and JabRef have lower recall values (46% and 44%), showing that a substantial number of MMEs were missed. The average recall across projects is 63%, suggesting that the approach is fairly sensitive, though again, this is not uniform across projects.

For the F_1-score, the results for TeaStore and Jabref (81% and 62%, respectively) reflect a good balance between precision and recall. However, MediaStore and TEAMMATES (22% and 25%) highlight a poor balance and significant room for improvement in both precision and recall. BigBlueButton performs moderately with an F_1-score of 55%. The average F_1-score is 49%, and the weighted average is 45%, indicating an overall moderate performance with room for improvement.

Similarly to the TLR tasks, the imbalanced dataset and the high number of true negatives influence the results for accuracy and specificity. Accuracy is high across most projects, with TeaStore, BigBlueButton, and JabRef showing near-perfect accuracy (98%, 96%, and 87%, respectively). Even in projects with lower precision and F1-scores, such as MediaStore and TEAMMATES, accuracy is relatively high (70% and 85%, respectively). The average accuracy is 87%, and the weighted average is 88%, indicating that overall classification performance is strong, mainly due to correctly identifying non-MMEs. Similarly, specificity is exceptionally high across all projects, ranging from 68% (MediaStore) to 100% (TeaStore and JabRef). The average and weighted average specificity are both 90%, indicating that the approach is highly effective at avoiding false positives in identifying non-MMEs.

In this setting, the Φ coefficient that assesses the overall correlation between the predicted and actual classifications is important. The results range from 23% (MediaStore) to 81% (TeaStore), showing variability in the strength of the classification. The average and weighted average are both 49%, suggesting a moderate but not outstanding correlation.

To tackle Question **Q5.3**, Table 6.3 show the detailed results for each project when ArDoCo is only using the common words filter, Table 6.4 shows the performance for each project when no filters are used.

ArDoCo with all filters has the highest precision (58% for average, 51% for weighted average). The precision drops significantly when filters are reduced or removed, with the “No Filters” configuration scoring just 17% for both

Table 6.3.: Evaluation Results For Detecting MMEs Using The **Common Words** Filters. The Metrics Precision (P), Recall (R), F_1-score (F_1), Accuracy (Acc), Specificity (Spec), And The Φ Coefficient Are Displayed.

Project	**P**	**R**	**F_1**	**Acc**	**Spec**	**Φ**
MediaStore	0.10	0.86	0.17	0.54	0.52	0.18
TeaStore	0.19	0.74	0.31	0.81	0.81	0.31
TEAMMATES	0.09	0.75	0.16	0.75	0.75	0.20
BigBlueButton	0.19	0.56	0.28	0.85	0.87	0.26
JabRef	1.00	0.44	0.62	0.87	1.00	0.62
Average	0.31	0.67	0.31	0.76	0.79	0.31
Weighted Avg.	0.23	0.68	0.27	0.77	0.78	0.27

Table 6.4.: Evaluation Results For Detecting MMEs Using **No** Filters. The Metrics Precision (P), Recall (R), F_1-score (F_1), Accuracy (Acc), Specificity (Spec), And The Φ Coefficient Are Displayed.

Project	**P**	**R**	**F_1**	**Acc**	**Spec**	**Φ**
MediaStore	0.10	0.86	0.17	0.54	0.52	0.18
TeaStore	0.12	0.74	0.21	0.68	0.68	0.20
TEAMMATES	0.09	0.82	0.16	0.73	0.72	0.21
BigBlueButton	0.14	0.56	0.22	0.80	0.81	0.20
JabRef	0.42	0.44	0.43	0.73	0.82	0.26
Average	0.17	0.68	0.24	0.70	0.71	0.21
Weighted Avg.	0.17	0.83	0.25	0.84	0.85	0.24

average and weighted average. Still, the results with no filters are better than the baseline. As the filters only remove results, ArDoCo with all filters applied achieves a slightly lower recall (63% for both average and weighted average). As a result, ArDoCo with all filters achieves the highest F_1-scores with an average of 49% and a weighted average of 45%. Removing or reducing filters in ArDoCo results in lower F_1-scores, with the “No Filters” configuration showing a noticeable drop.

The results demonstrate that applying all filters in ArDoCo leads to the best performance. This configuration maximizes precision, accuracy, specificity, and the Φ coefficient while still maintaining a reasonable recall. Reducing or

Table 6.5.: Comparison Of The Approach ArDoCo For Detecting MMEs With The Baseline And Influence of Filters Using The Metrics Precision (P), Recall (R), F_1-score (F_1), Accuracy (Acc), Specificity (Spec), And The Φ coefficient.

Approach	**P**	**R**	**F_1**	**Acc**	**Spec**	**Φ**
Average						
- Baseline	0.10	0.61	0.16	0.42	0.43	0.00
- ArDoCo (All Filters)	**0.58**	0.63	**0.49**	**0.87**	**0.90**	**0.49**
- ArDoCo (Common)	0.31	0.67	0.31	0.76	0.79	0.31
- ArDoCo (No Filters)	0.17	**0.68**	0.24	0.70	0.71	0.21
Weighted Average						
- Baseline	0.09	0.64	0.15	0.41	0.40	0.01
- ArDoCo (All Filters)	**0.51**	0.63	**0.45**	**0.88**	**0.90**	**0.46**
- ArDoCo (Common)	0.23	0.68	0.27	0.77	0.78	0.27
- ArDoCo (No Filters)	0.17	**0.83**	0.25	0.84	0.85	0.24

removing filters results in higher recall but at the cost of lower precision and specificity, leading to worse F_1-scores and overall performance.

To tackle Question **Q5.2**, Table 6.5 presents the average evaluation results of different filter options of the ArDoCo approach as well as the results for the baseline. Detailed results for the baseline are displayed in the appendix in Table B.7.

The baseline approach has the lowest precision (10% for average and 09% for weighted average). Recall is highest for ArDoCo without any filters (68% for average and 83% for weighted average). The baseline approach still maintains a high recall (61% for average, 64% for weighted average) despite its poor precision. Regarding F_1-score, the baseline scores are very low in this metric (16% for average, 15% for weighted average), reflecting its imbalance between precision and recall.

The other metrics follow a similar trend. ArDoCo with all filters achieves the best results for accuracy, specificity, and the Φ coefficient. Configurations with fewer or no filters show worse results. The baseline has much lower scores. This is particularly visible in the Φ coefficient, where the baseline has almost no correlation.

While having high recall, the baseline approach performs poorly across all other metrics. Its low precision and specificity highlight its inability to effectively distinguish between true and false positives, leading to poor overall effectiveness. ArDoCo with no filter still outperforms the baseline with a 50% higher average F_1-score.

To further answer Question **Q5.2** and Question **Q5.3** about evaluating the effectiveness of our approach with all filters compared to the other filter configurations and the baseline, we apply Wilcoxon's signed-rank test to analyze the significance of the observed differences in F_1-scores. Our results indicate that the approach (with filters) achieves statistically significant improvements over the others at the 0.05 significance level. Additionally, the no-filter variant also significantly outperforms the baseline.

6.4.3.3. Discussion & Conclusion

The evaluation results demonstrate notable variability in performance across different projects, highlighting both strengths and limitations of the approach.

The approach consistently performs better for TeaStore than for other projects across all metrics, achieving high precision, recall, and F_1-scores, alongside near-perfect accuracy and specificity. This indicates a well-balanced detection capability with minimal errors, suggesting that the approach is highly effective in this context.

In contrast, the results for MediaStore and TEAMMATES exhibit significant weaknesses, particularly in precision and F_1-score, despite comparatively higher recall and accuracy. This suggests that while the approach is sensitive in detecting instances of MMEs, it is prone to generating false positives in these projects, leading to lower precision and reduced F_1-scores.

The approach shows a more balanced performance for BigBlueButton and JabRef. However, their relatively lower recall indicates a tendency to miss some MMEs, reflecting a trade-off between detecting all relevant instances and avoiding false positives.

Overall, to answer Question **Q5.1**, the approach demonstrates decent performance in detecting MMEs, with particular strengths in recall and specificity. However, precision, particularly in projects like MediaStore and TEAM-

MATES, remains a critical challenge due to the prevalence of false positives. One of the biggest challenges is terms and phrases that have close similarity to actual component names, like “reencoding” and *ReEncoder* or an “image processing” and an *ImageProcessor*. This, on the one hand, can reduce the precision for the TLR steps, but on the other hand, directly influence the ID. Such terms are hard to distinguish, and the employed filters cannot remove them. MediaStore and TEAMMATES contain more such terms than the other projects, resulting in the approach’s low precision for these projects. Addressing other limitations and solving this issue, for example, with specialized heuristics, could significantly enhance the approach’s robustness.

Regarding Question **Q5.3**, the evaluation highlights that ArDoCo is highly effective in detecting MMEs when comprehensive filtering is applied, as evidenced by the significantly better results than the baseline and less-filtered configurations. The trade-off between precision and recall becomes apparent when filters are reduced. While recall improves, it is accompanied by a substantial decline in precision, resulting in lower F_1-scores and overall accuracy.

These findings emphasize the crucial role of filtering in achieving balanced performance, particularly in scenarios where minimizing false positives and, thus, maximizing precision is essential. Importantly, implementing these filters is neither complex nor resource-intensive, yet yields substantial improvements in results. The filtering strategies are instrumental in optimizing the precision-recall trade-off, thereby ensuring reliable detection performance.

Future work could focus on reducing false positives, especially in projects where precision is low, to enhance performance. The observed variability in project performance suggests that adapting or fine-tuning the detection approach to account for specific project characteristics could further improve outcomes.

Future research should also focus on three primary enhancements. First, research could focus on reducing false positives, especially in projects where precision is low. The observed variability in project performance suggests that adapting or fine-tuning the detection approach to account for specific project characteristics could further improve outcomes. Second, efforts should be made to refine ArDoCo to minimize or even eliminate the need for filters by more precisely identifying architecturally relevant entities through advanced techniques, similar to *Named Entity Recognition*, as discussed in

Section 2.3.2.9. Third, the approach's usability could be improved by developing tools and methods that facilitate the creation of filters, making the system more accessible to end-users.

6.4.4. Exploratory Application of LLMs for Inconsistency Detection

Analogously to TLR, the advancements of LLMs offer possibilities to tackle ID, using the abilities of LLMs to handle text and text-like artifacts with their understanding of semantics and context. This makes LLMs a solid choice to base approaches for ID on and to compare our approach ArDoCo with such an LLM-based approach.

For the LLM-based approach, I propose two variants: a direct end-to-end variant and a variant that utilizes trace links. I present these variants in Section 6.4.4.1 before evaluating them in Section 6.4.4.2.

6.4.4.1. Approach Variants

The first variant directly addresses the problem end-to-end, similar to the TLR approach: The LLM solves the problem by analyzing the SADs and SAMs directly.

The second variant employs TLR to establish relationships between the input documents as trace links. This approach proceeds as in the first variant but additionally provides the trace links to the LLM. The rationale is that the pre-established trace links serve as a foundational context, helping the LLM to detect inconsistencies more effectively. This approach hypothesizes that the preliminary mapping enriches the LLM's contextual comprehension, facilitating more precise inconsistency detection.

There are dedicated approaches for identifying UMEs and for identifying MMEs. For querying the LLM, the approaches use the same techniques as for the TLR variant (see Section 5.5.6), i.e., Chain-of-Thought, zero-shot, and few-shot prompting.

Table 6.6.: Weighted Average Results for identifying UMEs using the LLM-based approaches on the benchmark dataset using different prompting techniques. Lines with TLs indicate the provision of the gold standard trace links to the approach, LLM-TLs stands for the provision of the trace links from the best-performing LLM-based approach for TLR.

Approach	P	R	F_1
ArDoCo	0.88	**1.00**	0.93
Zero-shot	0.77	0.97	0.80
Few-shot	0.71	0.78	0.73
Chain-of-Thought	0.65	0.74	0.69
Ground truth TLs + Zero-shot	0.91	0.98	0.93
Ground truth TLs + Few-shot	**1.00**	**1.00**	**1.00**
Ground truth TLs + Chain-of-Thought	0.89	0.96	0.91
LLM-TLs + Few-shot	0.52	0.47	0.49

6.4.4.2. Evaluation

For the evaluation, we again use the dataset described in Section 5.5.2 and calculate the metrics precision (P), recall (R), and F_1-score (F_1). This evaluation uses the LLM GPT-4.

For the second variant, which additionally uses trace links as input, we provide trace links from the gold standard for TLR to ensure correctness. This evaluation assesses the maximum potential of the approach, assuming a TLR approach can identify all trace links without error. This assumption likely will not hold for automated approaches but helps identify the ceiling of the approach. For the best-performing variant, we also evaluate using the trace links from the best-performing LLM-based TLR approach (see Section 5.5.6).

Table 6.6 shows the average result for the identification of UMEs, and Table 6.7 shows the detailed results for each project. Without trace links, the LLM performs well but does not match the performance of ArDoCo. When looking into the details, the LLM-based approaches are outperformed in each benchmark project. Results vary significantly, with BigBlueButton being the worst-performing project with an F_1-score of 40% in zero-shot prompting. While the recall is still perfect in the zero-shot setting of BigBlueButton, it significantly drops for the other prompting variants. Across all prompting

Table 6.7.: Evaluation Results For Each Project For Detecting UMEs With The Metrics Precision (P), Recall (R), F_1-score (F_1). In This Version, No Trace Links Are Provided.

	Project	P	R	F_1
Zero-Shot	MediaStore	0.79	1.00	0.88
	TeaStore	1.00	0.88	0.93
	BigBlueButton	0.28	1.00	0.40
	JabRef	1.00	1.00	1.00
	Average	**0.77**	**0.97**	**0.80**
Few-Shot	MediaStore	0.80	1.00	0.89
	TeaStore	1.00	1.00	1.00
	BigBlueButton	0.02	0.10	0.03
	JabRef	1.00	1.00	1.00
	Average	**0.71**	**0.78**	**0.73**
Chain-of-Th.	MediaStore	0.79	0.95	0.85
	TeaStore	0.80	1.00	0.89
	BigBlueButton	0.00	0.00	0.00
	JabRef	1.00	1.00	1.00
	Average	**0.65**	**0.74**	**0.69**

variants, BigBlueButton performs worst, even returning nothing when using Chain-of-Thought prompting.

In contrast to TLR results, zero-shot prompting outperforms other techniques. Few-shot prompting performs second best, and Chain-of-Thought prompting performs worst among the three variants.

When provided with trace links from the gold standard, results improve substantially. Few-shot prompting achieves an F_1-score of 100%, while zero-shot and Chain-of-Thought prompting score 93% and 91%, respectively. However, providing ground truth trace links to ArDoCo also results in perfect scores by design.

With correct trace links, the approaches slightly outperform ArDoCo, when ArDoCo is not provided with ground truth trace links. However, with imperfect TLR results that come from the LLM itself, the performance for the LLMs drops significantly. Evaluating the few-shot prompting variant shows the F_1-score drops to 49%. For various projects, F_1-score ranges from 0% for

Table 6.8.: Weighted Average Results for identifying MMEs using the LLM-based approaches on the benchmark dataset using different prompting techniques. Lines with *TLs* indicate the provision of the gold standard trace links to the approach.

Approach	**P**	**R**	**F_1**
ArDoCo	**0.58**	**0.63**	**0.49**
Zero-shot	0.21	0.43	0.24
Few-shot	0.39	0.51	0.42
Chain-of-Thought	0.37	0.54	0.40
Ground truth TLs + Zero-shot	0.14	0.38	0.17
Ground truth TLs + Few-shot	0.26	0.48	0.30
Ground truth TLs + Chain-of-Thought	0.14	0.38	0.17

BigBlueButton to 80% for JabRef. The average results are lower than all other variants, including those without trace links. Thus, LLM-based approaches excel with accurate information but suffer greatly from inaccuracies.

Table 6.8 and Table 6.9 show the evaluation results when using LLMs to detect the second kind of inconsistencies, MMEs. The average precision and recall of ArDoCo are much higher compared to the results of the LLM-based approaches. However, due to the variance of ArDoCo's results, the F_1-score is only seven percentage points higher than that of the few-shot approach, which is the best-performing approach based on the F_1-score.

Providing the trace links from the ground truth to the approaches has a counterintuitive effect, significantly reducing their performance. Further research is needed to understand why this occurs. Due to the poor results when using ideal trace links, we skip the evaluation of automatically generated trace links in this instance.

Overall, the application of LLMs for this use case in its current form is not yet convincing. Future research is necessary to determine how to effectively incorporate LLMs for this purpose. In particular, as before for TLR, it is important to assess how to deal with the maximum token limit an LLM can process, for example, by employing more sophisticated RAG-based approaches.

Table 6.9.: Evaluation Results For Each Project For Detecting MMEs With The Metrics Precision (P), Recall (R), F_1-score (F_1). In This Version, No Trace Links Are Provided.

	Project	P	R	F_1
Zero-Shot	MediaStore	0.03	0.30	0.06
	TeaStore	0.05	0.21	0.08
	TEAMMATES	0.81	0.88	0.82
	BigBlueButton	0.03	0.08	0.04
	JabRef	0.13	0.67	0.22
	Average	**0.21**	**0.43**	**0.24**
Few-Shot	MediaStore	0.12	0.30	0.16
	TeaStore	0.21	0.21	0.21
	TEAMMATES	0.81	0.81	0.81
	BigBlueButton	0.12	0.38	0.17
	JabRef	0.71	0.83	0.73
	Average	**0.39**	**0.51**	**0.42**
Chain-of-Thought	MediaStore	0.08	0.30	0.13
	TeaStore	0.21	0.21	0.21
	TEAMMATES	0.82	0.94	0.82
	BigBlueButton	0.09	0.42	0.14
	JabRef	0.63	0.83	0.68
	Average	**0.37**	**0.54**	**0.40**

6.4.5. Threats to Validity

In the following, I discuss the threats to validity. It is structured based on the framework proposed by Runeson and Höst [235], as previously applied in Section 4.3.5 and Section 5.5.7. Given that the same dataset is employed, several threats are analogous to those already addressed in the evaluation of the TLR approaches in Section 5.5.7. As such, I will only go briefly over the most important aspects and focus on the threats that are particularly relevant for our evaluation of ID.

For the exploratory studies concerning LLMs, the same threats apply that are already discussed for TLR in Section 5.5.7, based on the discussion of Sallou et al. [237]. Consequently, I will not repeat them here.

Construct Validity
To evaluate our proposed methods, we employ well-established metrics that are aligned with the GQM paradigm, ensuring that they effectively address our research questions while mitigating threats to construct validity. Furthermore, we adopt a widely accepted research design that involves comparing the outcomes of our approaches against expected results derived from a ground truth. These strategies help ensure that our findings are robust and relevant.

However, despite these efforts, as discussed in Section 5.5.7, potential biases remain in the selection of the projects for evaluation, which could influence the outcomes. This selection bias is a significant threat, as it might affect the generalizability and relevance of our results across other contexts and datasets. Moreover, two projects, MediaStore and TEAMMATES, were used in preliminary studies to develop and tune the approach, most notably for the dynamic threshold used in the detection of MMEs. Similar to the threats for the TLR, there are threats to validity due to overfitting the approach on these projects. However, the results for these approaches are the worst among the evaluated projects, indicating that either there was no fine-tuning or overfitting on these projects or it did not result in inflated results.

For detecting MMEs, we simulate the inconsistencies by removing model elements, thereby introducing artificial inconsistencies. While this approach allows controlled experimentation, it may not accurately reflect real-world scenarios. For example, removing a core component or critical connector might oversimplify the system model, leading to a scenario that does not capture realistic system behavior. Consequently, the results obtained from these artificial cases could misrepresent the effectiveness of our method under more complex and realistic conditions.

Moreover, the baseline approach for detecting MMEs is intentionally simple, relying on a strict assumption to serve as a lower bound for comparison. While this allows for a clear benchmark, the simplicity of the baseline could result in an unrepresentative comparison, limiting the depth of insight gained from the evaluation.

Internal Validity
Several factors may influence the extent to which our evidence supports the claims regarding cause-and-effect relationships. To mitigate threats to internal validity, we have adhered to established practices aimed at minimizing bias, such as the careful selection of evaluation metrics and dataset

preparation. However, certain design decisions in our evaluation process may still introduce unintended biases.

The benchmark dataset used for the identification of UMEs contains a limited number of inconsistencies, raising concerns about the generalizability of our findings. For example, the effectiveness of our approach might be influenced by the volume of inconsistencies present. A higher number of inconsistencies could lead to different results for TLR, which in turn could affect the overall performance in detecting inconsistencies.

External Validity
The primary threat to external validity stems from the composition and characteristics of the benchmark dataset, as discussed earlier in Section 5.5.7. The dataset is limited in scope and diversity, raising concerns about the generalizability of our findings. Further tests and analyses on larger and more diverse datasets are necessary to generalize the findings.

Moreover, the benchmark dataset projects contain only a small number of UMEs. Although our approach successfully detects all of them in the current dataset, it is unclear whether it would perform similarly in other projects with different structures or more complex inconsistencies. This uncertainty limits our confidence in the scalability and adaptability of our methods to real-world scenarios.

For MMEs, the artificial inconsistencies created during the simulation may introduce confounding factors or noise that reduce the validity of the evaluation. The potential introduction of non-representative inconsistencies complicates assessing whether our method would perform similarly in real-world settings. In real-world settings, detecting MMEs might differ from the evaluation setting, restricting the validity of statements about generalizability.

Reliability
As emphasized in Section 5.5.7, we have taken several steps to ensure the reliability of our findings. These include adhering to rigorous guidelines when constructing the gold standards and using established processes when conducting our experiments. By using the benchmark dataset, we mitigated some of the bias associated with dataset selection.

However, the process of constructing the gold standards for UMEs remains prone to subjective bias, especially because inconsistencies are identified manually. While reusing the gold standards created for TLR during the evaluation of MME detection reduces bias to some extent, the inherent subjectivity involved in establishing these gold standards could still affect the reliability of our results.

Conclusion
In summary, while our evaluation provides valuable insights into our approaches for inconsistency detection, several limitations must be considered. Construct validity is challenged by the artificial nature of simulated inconsistencies, while internal validity may be compromised by the limited scope of the dataset. The restricted external validity raises questions about generalizability, and reliability concerns persist due to the subjective aspects of constructing ground truth specifications.

To address these limitations, future research should focus on more comprehensive datasets, including those with naturally occurring inconsistencies, to better reflect real-world scenarios. Additionally, more advanced baseline methods could provide a more meaningful comparison, enhancing the robustness of our evaluation.

Part III.

Epilogue

7. Conclusion and Future Work

I looked into Traceability Link Recovery (TLR) for Software Architecture Documentation (SAD) in this dissertation. Making Architectural Design Decisions (ADDs) is fundamental for architectural design, directly contributing to the system's chances for success by improving the development, maintenance, and evolution of a system. Therefore, I first surveyed what kinds of fine-grained ADDs can be found in SAD and created a taxonomy of ADDs. Then, I looked into architecture documentation directly and how to recover links between the different forms of documentation, namely natural language-based documentation (SADs) and Software Architecture Models (SAMs). I explicitly looked into TLR between SADs and SAMs, between SAMs and code, and transitive TLR between SADs and code using SAMs as intermediate artifacts. The created trace links can support developers, e.g., in synchronizing artifacts and keeping them consistent. To show that, I explored the use of TLR for automated Inconsistency Detection (ID) within architectural documentation, focusing on two types of inconsistencies, Unmentioned Model Elements (UMEs) and Missing Model Elements (MMEs).

In this chapter, I will first conclude the presented and evaluated work in Section 7.1. Then, I will look into possible future work in Section 7.2.

7.1. Conclusion

In this section, I conclude my work by summarizing my contributions and briefly discussing them. For this, Table 7.1 shows an overview of the results of my approaches in the evaluation with selected competing baseline approaches for comparison. I selected the baselines based on my research questions and based on the approaches' performances, i.e., best average F_1-score.

Table 7.1.: Overview Of The Average Results Of The Proposed Approaches For TLR And ID And A Selected Corresponding Baseline Approach Using the Metrics Precision (P), Recall (R), and F_1-score (F_1), and, where available, Accuracy (Acc), Specificity (Spec), And The Φ Coefficient (Φ). The LLM-based approaches that use trace links (TLs) use the ground truth trace links.

Approach	P	R	F_1	Acc	Spec	Φ
ADD Classification with BERT (Section 4.4)						
Binary Classification	0.90	0.94	0.92	-	-	-
Multi-Class Classif.	0.58	0.56	0.56	-	-	-
Multi-Class & -Label	0.68	0.43	0.50	-	-	-
TLR SAD-SAM (Sections 5.2 and 5.5.6)						
Baseline	0.78	0.58	0.65	0.96	0.99	0.64
ArDoCo	0.87	0.82	0.82	0.98	0.99	0.82
LLM (RAG + CoT)	0.52	0.90	0.64	-	-	-
TLR SAM-Code (Section 5.3)						
Baseline	0.21	0.32	0.20	0.99	0.99	0.22
ArCoTL	0.98	0.99	0.98	1.00	1.00	0.98
TLR SAD-Code (Section 5.4)						
ArDoCode	0.27	0.78	0.37	0.82	0.81	0.37
TransArC	0.87	0.81	0.82	0.98	0.98	0.82
ID UME (Sections 6.1 and 6.4.4)						
ArDoCo	0.92	1.00	0.95	0.98	0.97	0.94
LLM (Zero-Shot)	0.77	0.97	0.80	-	-	-
LLM (Ground truth TLs + Few-Shot)	1.00	1.00	1.00	-	-	-
ID MME (Sections 6.2 and 6.4.4)						
Baseline	0.10	0.61	0.16	0.42	0.43	0.00
ArDoCo (All Filters)	0.58	0.63	0.49	0.87	0.90	0.49
ArDoCo (Common F.)	0.31	0.67	0.31	0.76	0.79	0.31
ArDoCo (No Filters)	0.17	0.68	0.24	0.70	0.71	0.21
LLM (Few-Shot)	0.39	0.51	0.42	-	-	-
LLM (Ground truth TLs + Few-Shot)	0.26	0.48	0.30	-	-	-

7.1.1. Taxonomy for Design Decisions in Architecture Documentation

To answer **RQ 1** about the kinds of Architectural Design Decisions in architecture documentation, I created a taxonomy to organize these ADDs into well-defined and fine-grained categories, to allow for a clear, systematic approach to document and communicate architectural intent. The taxonomy is built in an iterative process, starting from an initial taxonomy created based on literature. Refining the initial taxonomy used documentation from open-source projects in multiple iterations. The resulting taxonomy consists of 36 classes with 24 leaf classes to classify ADDs in a fine-grained manner. Following a GQM plan, I evaluated the taxonomy's purpose, structure, and application.

To answer **RQ 2**, I presented an automated classification approach based on the LLM BERT and classifies according to the fine-grained taxonomy for ADDs. BERT performed well in the pre-study, which consisted of a requirement classification task. For the classification of ADDs, three different settings were evaluated. Binary classification, i.e., deciding if an ADD is present, achieved the best performance, with a precision of 90%, recall of 94%, and an F_1-score of 92%, indicating robust performance in differentiating between two classes. However, performance declined significantly for multi-class and multi-label classification tasks, with F_1-scores of 56% and 50%, respectively. This suggests that while BERT is well-suited for binary classification, its effectiveness diminishes when handling complex multi-class and multi-label scenarios, likely due to increased classification complexity.

7.1.2. Recovering Trace Links in Architecture Documentation

I tackled **RQ 3** and answered how well trace links can be automatically recovered between the different software architecture artifacts by introducing three new approaches looking at the artifacts SAD, SAM, and code. I discuss the new approaches in the next paragraphs and answer the corresponding research questions. To make the approaches independent of concrete programming or modeling languages, I introduced intermediate representations of artifacts. This way, new languages only require a new transformation into the intermediate representation.

I presented my first approach for TLR, ArDoCo, that recovers trace links between SAD and SAM, tackling **RQ 3.1**. The approach has multiple processing steps that are executed in a pipeline. Each processing step uses various heuristics. In general, ArDoCo tries to identify architecturally relevant parts within the SAD and tries to map them to the SAM to create the trace links.

In the evaluation, ArDoCo outperformed the other approaches with an F_1-score of 82%, accuracy of 98%, and specificity of 99%, demonstrating high precision and recall. The detailed results also showed the challenge of creating heuristics that yield high precision and high recall for all evaluation projects. The approach lacked some precision for TEAMMATES but at the same time lacked recall for MediaStore. The baseline approach performed moderately with an F_1-score of 65%.

Concerning **RQ 3.4**, the LLM-based approach that uses RAG + CoT had a strong recall (90%) but low precision (52%), resulting in an F_1-score of 64%. Accordingly, LLMs cannot outperform the excellent performance of ArDoCo in a simple approach. More advanced LLM-based approaches might, at some point, exceed ArDoCo's performance, but more effort is required. Moreover, a combination of LLMs and the heuristics of ArDoCo might be worthwhile; I discuss this further in Section 7.2.

The second TLR approach, ArCoTL, tackles **RQ 3.2**, i.e., TLR between SAM and code. The approach uses different heuristics to assess if two artifacts are (semantically) similar. The heuristics are executed using a computational graph that orders the execution and defines how the individual results of the heuristics are combined.

In the evaluation, ArCoTL achieved near-perfect results across all metrics, including precision, recall, and F_1-score of 98%, as well as full accuracy and specificity scores of 100%. This indicates ArCoTL's excellent ability to capture SAM-Code relationships accurately in the evaluation dataset. The baseline performed poorly, with an F_1-score of only 20%, suggesting it struggled to identify SAM-Code patterns effectively. This shows that the task is non-trivial, highlighting ArCoTL's capacity further.

The third TLR approach combines the previous two for a transitive approach to address TLR between SAD and code and **RQ 3.3**. The underlying idea is that SAMs as intermediate artifacts can reduce the semantic gap between the source and target artifact to recover the trace links more effectively. As such, the approach combines the results of the two approaches transitively by glueing together trace links with the same element “in the middle” from the SAM.

In the evaluation, TransArC achieved significantly better results with an F_1-score of 82%, high accuracy (98%), and specificity (98%), indicating its strong performance in identifying SAD-Code relations. The detailed results also show that the performance of the involved TLR approaches influences the results for TransArC in some parts. In contrast to the strong performance of TransArC, ArDoCode, an adaptation of ArDoCo to work with code instead of SAM, had lower performance, with an F_1-score of only 37% and an accuracy of 82%, despite being the best baseline approach. These excellent results indicate that the transitive approach can bridge the semantic gap much better than other approaches in this evaluation.

7.1.3. Detecting Inconsistencies in Architecture Documentation

To identify inconsistencies, I presented approaches that use the trace links and other intermediate results from the TLR approach ArDoCo. This way, I address **RQ 4** and its corresponding sub-questions.

To address **RQ 4.1**, I use trace links to identify model elements not represented in the documentation (called Unmentioned Model Elements (UMEs)). In the evaluation, ArDoCo demonstrated strong performance with an F_1-score of 95%. To answer **RQ 4.3** for UMEs, I also evaluated the performance of LLMs in this task. In a zero-shot setting, the LLMs achieved an F_1-score of 80%, indicating a higher rate of misidentifications. However, when using ground truth trace links and few-shot prompting, the LLM-based approach achieved perfect precision, recall, and F_1-score of 100%, highlighting excellent identification capabilities when the necessary information is available. Similarly, providing ground truth trace links to ArDoCo results in perfect scores by design.

To address **RQ 4.2**, the identification of architecturally relevant elements that are mentioned in the documentation but missing in the model (called Missing Model Elements (MMEs)), I use the intermediate results of ArDoCo in combination with the recovered trace links. ArDoCo initially identifies relevant entities that should be linked. If no trace links are found, the approach identifies a potential inconsistency. These potential inconsistencies can then be further filtered (e.g., based on common names, blacklists, etc.). In the evaluation, ArDoCo using all filters achieves the highest F_1-score of 49%, supported by high specificity (90%) and accuracy (87%). The baseline method performed poorly, with an F_1-score of only 16%, indicating limited efficacy. I evaluated the performance of LLM-based approaches to answer **RQ 4.3** for MMEs, but these showed limited performance with F_1-scores around 30%-42%, suggesting that MMEs were just as challenging for LLMs as for the ArDoCo approach.

7.2. Future Work

In this dissertation, I presented several contributions to enhance the understanding and management of software architecture: a taxonomy for architectural design decisions, a classification approach leveraging this taxonomy, TLR between SADs, SAMs, and code, and a method for detecting inconsistencies between SADs and SAMs. While these contributions address significant gaps in architectural knowledge management and consistency, several promising directions for future research remain. In this section, I will propose several options for future work to extend my contributions.

Exploring the taxonomy of architectural design decisions reveals several potential directions for future work. For instance, this study did not fully investigate the use of the taxonomy for automated inconsistency detection due to shortcomings in the automated classification. Applying the taxonomy specifically for automated inconsistency detection could help evaluate its robustness and potentially expose limitations that future research should address.

Achieving high accuracy for automated classification of ADDs would enable targeted consistency checks. While current results in automated classification are promising, they remain insufficient for practical applications such as automated inconsistency detection. Consequently, further research is needed to enhance classification accuracy. Given recent advancements in machine learning and LLMs, improvements in automated classification for these tasks are likely.

Future work in TLR between SAD and SAM could focus on several key areas. First, expanding experiments across a wider range of projects would allow for assessing the method's generalizability and robustness in diverse contexts.

The evaluation looked at OSS that might be structurally different from closed-source software. Moreover, the software contained the descriptions and mainly the names of the components. The approach uses this information for its linking. However, if other documentation is different and, e.g., does not describe the components in this way, the approach will likely perform worse. For example, if components are not named but only vaguely described by their responsibilities, the approach's performance will drop. In such cases, another strategy is required that, for example, first creates a suitable name for these described components before proceeding with the other steps of ArDoCo. Further, the approach is also currently limited to structural SAMs; extending the evaluation to other types of SAMs could reveal additional insights and potential adaptations.

In the first steps of ArDoCo, the approach tries to identify architecturally relevant entities using heuristics, similar to Named Entity Recognition (NER). The heuristic-based NER used to identify architecturally relevant entities could benefit from advancements in natural language processing. Utilizing recent techniques, such as LLMs, may improve the precision and adaptability of entity detection, leading to more accurate TLR.

While the ArCoTL approach for TLR between SAM and code achieved near-perfect results, several tasks remain open for future work. Expanding this approach to additional programming languages is a key priority. Although the intermediate code representation aims to abstract away language-specific details, this study primarily focused on Java. Applying the approach to other languages, especially those that follow different programming paradigms, may introduce unique challenges and could impact performance. Thus, future work should focus on developing additional transformations for a broader range of languages and evaluating their performance in TLR. This expansion will help determine the robustness and adaptability of ArCoTL across diverse programming environments. Furthermore, future work should look into other ideas, such as the briefly discussed incorporation of architecture recovery approaches. This requires looking into, for example, TLR between the existing SAM and an extracted one.

The transitive TLR approach between SAD and code (TransArC) shows promising results, significantly outperforming other methods. Related work (cf. Section 3.2.3) supports this conclusion, suggesting that using intermediate artifacts to bridge the semantic gap is effective. This general principle, i.e., leveraging specialized artifacts to bridge semantic differences, could apply to other cases where artifacts exist between two endpoints of a gap. However, further research is needed to determine which artifacts are most suitable to serve as intermediates in different contexts. Future work should evaluate TransArC across a broader range of projects to assess its versatility across diverse settings, domains, and scenarios. Additionally, it will be valuable to examine the transitive approach's adaptability for other TLR tasks, such as linking requirements to code. This exploration will involve identifying appropriate and efficient intermediate artifacts for various linkage scenarios and applications. Moreover, it can be worthwhile to adapt other approaches like TRIAD [75] and explore the combination of TransArC with them.

The current approach prioritizes precision by disregarding direct mentions of code entities in SADs that lack links to SAMs. Future research should explore integrating this approach with complementary methods to address cases where documentation mentions code entities absent from the design artifacts. Such a combination could enhance the overall effectiveness and completeness of the transitive approach, broadening its utility.

One challenge with the transitive approach is its reliance on SAMs as intermediate artifacts. Many projects lack existing SAMs, and creating them from scratch often requires significant overhead. The usage of architecture recovery can be interesting for the transitive approach. Although architecture recovery techniques exist, they are frequently inaccurate, highly specialized, or limited to specific purposes, such as performance modeling. Notably, the transitive approach requires minimal architectural information from SAMs, primarily the names of key entities like components and interfaces. Therefore, future work should investigate whether a streamlined recovery method could capture enough architectural detail to supply TransArC with the necessary information. Given their strong performance in related tasks, LLMs present a promising starting point. Another future research direction could be recovering an architecture model from code and one from SAD. Then, a TLR approach can try to connect these architecture models, basically moving the original artifacts closer together.

The inconsistency detection in this dissertation addresses the identification of UMEs and MMEs. The method for detecting UMEs relies on the correctness of trace links but does not assume specific characteristics of the artifacts. Consequently, this approach is adaptable to other artifacts that can be automatically linked via TLR. For instance, trace links between SAD and code can be leveraged to identify UMEs across these artifacts. However, configuration adjustments are required to specify which elements in the code (such as packages, classes, etc.) should correspond to elements in the architecture documentation. This configuration approach can be similarly applied for the detection of MMEs.

The detection of MMEs depends significantly on identifying architecturally relevant entities within ArDoCo, much like NER. As previously discussed, this process could be enhanced using modern machine learning techniques, such as LLMs. By improving the detection of architecturally relevant entities, the precision of MME detection can be directly increased. Enhanced entity detection may also reduce the reliance on additional filters. Future work should explore these enhancements further.

To further enhance inconsistency detection in SAD, future work could focus on identifying a wider range of inconsistency types and refining and automating the classification and detection processes. In addition to detecting UMEs and MMEs, inconsistency detection approaches could address other forms of inconsistencies.

Examples of such inconsistencies include behavioral and interaction mismatches. These occur when the expected interactions between components, as described in the SAD, are not reflected in the SAM or the codebase. Detection of these inconsistencies would involve verifying that interactions specified in SAD align with interaction patterns in SAM, possibly using sequence diagrams or behavioral contracts.

Other examples of inconsistencies are structural discrepancies that arise when the architectural layers or component hierarchies in the SAD differ from those in the SAM. Detecting these inconsistencies may involve automated structural analysis, comparing documented module layers, dependencies, and component distributions with those in the SAM or source code.

Further, after detecting the mention of an architectural pattern in the SAD, an approach can check the SAM (or code) if the pattern is implemented. For this, there is existing work to detect architectural patterns in code, for example, by Mirakhorli et al. [188]. In an exploratory study, I collaborated with them and looked into detecting architectural patterns in code using BERT (see [127]). Beyond simple presence verification of patterns, more complex inconsistencies might involve variations or deviations in how architectural patterns are applied. For instance, if the SAD describes a layered architecture, the detection approach could verify that component interactions respect the specified layer boundaries, ensuring that dependencies do not bypass layers in ways that violate the documented pattern.

Mentions of non-functional aspects such as performance, security, and scalability requirements in the SAD may also not always be consistently implemented in the SAM. For instance, if the SAD specifies performance constraints, these could be checked against annotations in the SAM, simulation results, or profiling data from the implementation to ensure alignment with the documented expectations.

A practical future direction involves creating integrated tools that facilitate ongoing consistency checks during development. By embedding inconsistency detection mechanisms into continuous integration (CI) pipelines, inconsistencies between SAD and SAM could be automatically detected as the architecture evolves, enabling architects and developers to address issues early. Such a tool could provide real-time feedback, suggest potential resolutions, and facilitate updates to the SAD or SAM to reflect architectural changes.

In summary, extending inconsistency detection for SAD and SAM requires covering a broader array of inconsistency types, advancing automation techniques, enhancing integration with development workflows, and refining classification and detection methodologies. This would lead to a more robust and maintainable architecture documentation process, reducing the likelihood of discrepancies between design documentation and implementation over time.

Bibliography

The titles of most entries are hyperlinks resolving the DOIs or pointing to other online sources for the entries.

[1] Z. S. H. Abad, O. Karras, P. Ghazi, M. Glinz, G. Ruhe, and K. Schneider. "What Works Better? A Study of Classifying Requirements". In: *2017 IEEE 25th International Requirements Engineering Conference (RE)*. 2017, pp. 496–501.

[2] K. Aggarwal. *Software engineering*. New Age International, 2005.

[3] S. A. Ajila and D. Wu. "Empirical study of the effects of open source adoption on software development economics". In: *Journal of Systems and Software* 80.9 (2007), pp. 1517–1529.

[4] Z. Alexeeva, D. Perez-Palacin, and R. Mirandola. "Design Decision Documentation: A Literature Overview". In: *Software Architecture*. Springer International Publishing, 2016, pp. 84–101.

[5] N. Ali, S. Baker, R. O'Crowley, S. Herold, and J. Buckley. "Architecture consistency: State of the practice, challenges and requirements". In: *Empirical Software Engineering* 23.1 (2017), pp. 224–258.

[6] R. Ali, F. Dalpiaz, and P. Giorgini. "Reasoning with contextual requirements: Detecting inconsistency and conflicts". In: *Information and Software Technology* 55.1 (2013), pp. 35–57.

[7] ANSI/NISO. *Guidelines for the Construction, Format, and Management of Monolingual Controlled Vocabularies*. Standard. NISO, 2005.

[8] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. "Recovering Traceability Links between Code and Documentation". In: *IEEE Transactions on Software Engineering* 28.10 (2002), pp. 970–983.

[9] G. Antoniol, A. Potrich, P. Tonella, and R. Fiutem. "Evolving object oriented design to improve code traceability". In: *Proceedings Seventh International Workshop on Program Comprehension*. 1999, pp. 151–160.

[10] C. Arora, M. Sabetzadeh, L. Briand, and F. Zimmer. “Automated Checking of Conformance to Requirements Templates Using Natural Language Processing”. In: *IEEE Transactions on Software Engineering* 41.10 (2015), pp. 944–968.

[11] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. “DBpedia: A Nucleus for a Web of Open Data”. In: *The Semantic Web.* Springer Berlin Heidelberg, 2007, pp. 722–735.

[12] T. W. W. Aung, H. Huo, and Y. Sui. “A Literature Review of Automatic Traceability Links Recovery for Software Change Impact Analysis”. In: *Proceedings of the 28th International Conference on Program Comprehension.* ICPC ’20. Association for Computing Machinery, 2020, pp. 14–24.

[13] M. Awad and R. Khanna. “Support Vector Machines for Classification”. In: *Efficient Learning Machines: Theories, Concepts, and Applications for Engineers and System Designers.* Apress, 2015, pp. 39–66.

[14] E. Barba, L. Procopio, and R. Navigli. “ConSeC: Word Sense Disambiguation as Continuous Sense Comprehension”. In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing.* Association for Computational Linguistics, 2021, pp. 1492–1503.

[15] V. R. Basili, G. Caldiera, and H. D. Rombach. “The Goal Question Metric Approach”. In: *Encyclopedia of software engineering* (1994), pp. 528–532.

[16] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice.* Addison-Wesley Professional, 2003.

[17] D. Bedford. “Evaluating Classification Schema and Classification Decisions”. In: *Bulletin of the American Society for Information Science and Technology* 39.2 (2013), pp. 13–21.

[18] P. Berta, M. Bystrický, M. Krempaský, and V. Vranić. “Employing issues and commits for in-code sentence based use case identification and remodularization”. In: *Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems.* ECBS ’17. Association for Computing Machinery, 2017.

[19] M. Bevilacqua and R. Navigli. “Breaking Through the 80% Glass Ceiling: Raising the State of the Art in Word Sense Disambiguation by Incorporating Knowledge Graph Information”. In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics.* Association for Computational Linguistics, 2020, pp. 2854–2864.

[20] M. Bhat, K. Shumaiev, A. Biesdorf, U. Hohenstein, and F. Matthes. “Automatic Extraction of Design Decisions from Issue Management Systems: A Machine Learning Based Approach”. In: *Software Architecture.* LNCS. Springer International Publishing, 2017, pp. 138–154.

[21] M. Bhat, C. Tinnes, K. Shumaiev, A. Biesdorf, U. Hohenstein, and F. Matthes. “ADeX: A Tool for Automatic Curation of Design Decision Knowledge for Architectural Decision Recommendations”. In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C).* 2019, pp. 158–161.

[22] T. Biggerstaff, B. Mitbander, and D. Webster. “The concept assignment problem in program understanding”. In: *Proceedings of the Working Conference on Reverse Engineering.* 1993, pp. 27–43.

[23] C. M. Bishop and N. M. Nasrabadi. *Pattern recognition and machine learning.* Vol. 4. 4. Springer, 2006.

[24] D. M. Blei, A. Y. Ng, and M. I. Jordan. “Latent Dirichlet Allocation”. In: *Advances in Neural Information Processing Systems 14 [Neural Information Processing Systems: Natural and Synthetic, NIPS 2001].* MIT Press, 2001, pp. 601–608.

[25] M. Borg, P. Runeson, and A. Ardö. “Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability”. In: *Empirical Software Engineering* 19.6 (2014), pp. 1565–1616.

[26] N. Borovits, I. Kumara, D. Di Nucci, P. Krishnan, S. D. Palma, F. Palomba, D. A. Tamburri, and W.-J. v. d. Heuvel. “FindICI: Using machine learning to detect linguistic inconsistencies between code and natural language descriptions in infrastructure-as-code”. In: *Empirical Software Engineering* 27.7 (2022), p. 178.

[27] S. Brown. *The C4 model for visualising software architecture.* 2018.

[28] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020*. 2020.

[29] B. Bruegge, O. Creighton, J. Helming, and M. Kögel. "Unicase – an Ecosystem for Unified Software Engineering". In: *ICGSE'08: Distributed Software Development: Methods and Tools for Risk Management* (2012).

[30] A. Bucaioni, A. Di Salle, L. Iovino, L. Mariani, and P. Pelliccione. "Continuous Conformance of Software Architectures". In: *2024 IEEE 21st International Conference on Software Architecture (ICSA)*. 2024, pp. 112–122.

[31] Y. Cai, H. Wang, S. Wong, and L. Wang. "Leveraging Design Rules to Improve Software Architecture Recovery". In: *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*. QoSA '13. Association for Computing Machinery, 2013, pp. 133–142.

[32] C. Carrillo, R. Capilla, O. Zimmermann, and U. Zdun. "Guidelines and Metrics for Configurable and Sustainable Architectural Knowledge Modelling". In: *Proceedings of the 2015 European Conference on Software Architecture Workshops*. ECSAW '15. Association for Computing Machinery, 2015.

[33] D. Charlet and G. Damnati. "SimBow at SemEval-2017 Task 3: Soft-Cosine Semantic Similarity between Questions for Community Question Answering". In: *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*. Association for Computational Linguistics, 2017, pp. 315–319.

[34] J. Chen, B. Goudey, N. Geard, and K. Verspoor. "Integration of background knowledge for automatic detection of inconsistencies in gene ontology annotation". In: *Bioinformatics* 40 (2024).

[35] L. Chen, D. Wang, J. Wang, and Q. Wang. "Enhancing Unsupervised Requirements Traceability with Sequential Semantics". In: *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. 2019, pp. 23–30.

[36] N. Chomsky. *Syntactic Structures.* Mouton de Gruyter, 1957.

[37] N. Chomsky. *The Minimalist Program.* The MIT Press, 1995.

[38] S. Çiraci, H. Sözer, and B. Tekinerdogan. "An Approach for Detecting Inconsistencies between Behavioral Models of the Software Architecture and the Code". In: *IEEE 36th Annual Computer Software and Applications Conference.* 2012, pp. 257–266.

[39] J. Cleland-Huang. "Traceability in Agile Projects". In: *Software and Systems Traceability.* Springer London, 2012, pp. 265–275.

[40] J. Cleland-Huang, O. Gotel, A. Zisman, et al. *Software and systems traceability.* Vol. 2. 3. Springer, 2012.

[41] J. Cleland-Huang, S. Mazrouee, H. Liguo, and D. Port. *PROMISE: NFR.* Zenodo, 2007. DOI: `10.5281/zenodo.268542`.

[42] CoEST. *Center of Excellence for Software And Systems Traceability.* `https://web.archive.org/web/20230518011309/http://www.coest.org/`. Accessed: 2024-11-27.

[43] J. Cohen. "A coefficient of agreement for nominal scales". In: *Educational and psychological measurement* 20.1 (1960), pp. 37–46.

[44] S. Cook, C. Bock, P. Rivett, T. Rutt, E. Seidewitz, B. Selic, and D. Tolbert. *Unified Modeling Language (UML) Version 2.5.1.* Standard. Object Management Group (OMG), 2017.

[45] H. Cramér. *Mathematical methods of statistics.* Vol. 26. Princeton university press, 1946.

[46] F. Dalpiaz, D. Dell'Anna, F. B. Aydemir, and S. Çevikol. "Requirements Classification with Interpretable Machine Learning and Dependency Parsing". In: *2019 IEEE 27th International Requirements Engineering Conference (RE).* 2019, pp. 142–152.

[47] M.-C. De Marneffe and C. D. Manning. *Stanford typed dependencies manual.* Tech. rep. Technical report, Stanford University, 2008.

[48] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. "A fast and elitist multiobjective genetic algorithm: NSGA-II". In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 182–197.

[49] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. "Indexing by latent semantic analysis". In: *Journal of the American Society for Information Science* 41.6 (1990), pp. 391–407.

[50] A. Dekhtyar and V. Fong. “RE Data Challenge: Requirements Identification with Word2Vec and TensorFlow”. In: *2017 IEEE 25th International Requirements Engineering Conference (RE)*. 2017, pp. 484–489.

[51] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, 2019, pp. 4171–4186.

[52] W. Ding, P. Liang, A. Tang, H. v. Vliet, and M. Shahin. “How Do Open Source Communities Document Software Architecture: An Exploratory Survey”. In: *19th International Conference on Engineering of Complex Computer Systems*. 2014, pp. 136–145.

[53] V. Dobrovolskii. “Word-Level Coreference Resolution”. In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2021, pp. 7670–7675.

[54] Z. Dong, A. Andrzejak, D. Lo, and D. Costa. “ORPLocator: Identifying Read Points of Configuration Options via Static Analysis”. In: *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. 2016, pp. 185–195.

[55] A. Egyed. “Scalable consistency checking between diagrams - the VIEWINTEGRA approach”. In: *16th Annual International Conference on Automated Software Engineering (ASE 2001)*. 2001, pp. 387–390.

[56] A. Egyed, F. Graf, and P. Grünbacher. “Effort and Quality of Recovering Requirements-to-Code Traces: Two Exploratory Experiments”. In: *2010 18th IEEE International Requirements Engineering Conference*. 2010, pp. 221–230.

[57] L. Ehrlinger and W. Wöß. “Towards a definition of knowledge graphs.” In: *SEMANTiCS (Posters, Demos, SuCCESS)* 48.1-4 (2016), p. 2.

[58] “Enabling consistency in view-based system development – The Vitruvius approach”. In: *Journal of Systems and Software* 171 (2021), p. 110815.

[59] D. Falessi, M. Becker, and G. Cantone. “Design Decision Rationale: Experiences and Steps Ahead towards Systematic Use”. In: *SIGSOFT Softw. Eng. Notes* 31.5 (2006).

[60] D. Falessi, J. Roll, J. L. Guo, and J. Cleland-Huang. “Leveraging Historical Associations between Requirements and Source Code to Identify Impacted Classes”. In: *IEEE Transactions on Software Engineering* 46.4 (2020), pp. 420–441.

[61] A. Fantechi and E. Spinicci. “A Content Analysis Technique for Inconsistency Detection in Software Requirements Documents.” In: (2005), pp. 245–256.

[62] J.-M. Favre. “Foundations of model (Driven)(Reverse) engineering”. In: *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2005.

[63] A. R. Feinstein and D. V. Cicchetti. “High agreement but low kappa: I. The problems of two paradoxes”. In: *Journal of clinical epidemiology* 43.6 (1990), pp. 543–549.

[64] S. Feldmann, S. J. I. Herzig, K. Kernschmidt, T. Wolfenstetter, D. Kammerl, A. Qamar, U. Lindemann, H. Krcmar, C. J. J. Paredis, and B. Vogel-Heuser. “A comparison of inconsistency management approaches using a mechatronic manufacturing system design case study”. In: *2015 IEEE International Conference on Automation Science and Engineering (CASE)*. 2015, pp. 158–165.

[65] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. “CodeBERT: A Pre-Trained Model for Programming and Natural Languages”. In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, 2020, pp. 1536–1547.

[66] M. Feurer and F. Hutter. “Hyperparameter optimization”. In: *Automated machine learning: Methods, systems, challenges* (2019), pp. 3–33.

[67] J. Firth. “A Synopsis of Linguistic Theory, 1930-1955”. In: *Studies in Linguistic Analysis* (1957), pp. 10–32.

[68] J. L. Fleiss. “Measuring nominal scale agreement among many raters.” In: *Psychological bulletin* 76.5 (1971), p. 378.

[69] L. Fu, P. Liang, X. Li, and C. Yang. “A Machine Learning Based Ensemble Method for Automatic Multiclass Classification of Decisions”. In: *Evaluation and Assessment in Software Engineering*. EASE 2021. Association for Computing Machinery, 2021, pp. 40–49.

[70] D. Fucci, E. Alégroth, and T. Axelsson. “When traceability goes awry: An industrial experience report”. In: *Journal of Systems and Software* 192 (2022), p. 111389.

[71] D. Fuchß, S. Corallo, J. Keim, J. Speit, and A. Koziolek. “Establishing a Benchmark Dataset for Traceability Link Recovery between Software Architecture Documentation and Models”. In: *2nd International Workshop on Mining Software Repositories for Software Architecture - Co-located with 16th European Conference on Software Architecture.* 2022.

[72] D. Fuchß, J. Keim, S. Corallo, and A. Koziolek. *The ArDoCo Benchmark*. Dataset. 2022.

[73] B. Fuglede and F. Topsoe. “Jensen-Shannon divergence and Hilbert space embedding”. In: *International Symposium on Information Theory, 2004. ISIT 2004. Proceedings.* 2004.

[74] P. Gage. “A new algorithm for data compression”. In: *C Users Journal* 12.2 (1994), pp. 23–38.

[75] H. Gao, H. Kuang, W. K. G. Assunção, C. Mayr-Dorn, G. Rong, H. Zhang, X. Ma, and A. Egyed. “TRIAD: Automated Traceability Recovery based on Biterm-enhanced Deduction of Transitive Links among Artifacts”. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE ’24. Association for Computing Machinery, 2024.

[76] H. Gao, H. Kuang, X. Ma, H. Hu, J. Lü, P. Mäder, and A. Egyed. “Propagating Frugal User Feedback through Closeness of Code Dependencies to Improve IR-based Traceability Recovery”. In: *Empir Software Eng* 27.2 (2022), p. 41.

[77] H. Gao, H. Kuang, K. Sun, X. Ma, A. Egyed, P. Mäder, G. Rong, D. Shao, and H. Zhang. “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. Association for Computing Machinery, 2022, p. 1.

[78] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, M. Wang, and H. Wang. *Retrieval-Augmented Generation for Large Language Models: A Survey*. 2023.

[79] V. Gervasi and D. Zowghi. "Reasoning about Inconsistencies in Natural Language Requirements". In: *ACM Trans. Softw. Eng. Methodol.* 14.3 (2005), pp. 277–330.

[80] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. D. Lucia. "On Integrating Orthogonal Information Retrieval Methods to Improve Traceability Recovery". In: *2011 27th IEEE International Conference on Software Maintenance (ICSM).* 2011, pp. 133–142.

[81] Z. Ghahramani. "Unsupervised Learning". In: *Advanced Lectures on Machine Learning: ML Summer Schools 2003, Canberra, Australia, February 2 - 14, 2003, Tübingen, Germany, August 4 - 16, 2003, Revised Lectures.* Springer Berlin Heidelberg, 2004, pp. 72–112.

[82] A. Ghannem, M. S. Hamdi, M. Kessentini, and H. H. Ammar. "Search-based requirements traceability recovery: A multi-objective approach". In: *2017 IEEE Congress on Evolutionary Computation (CEC).* 2017, pp. 1183–1190.

[83] B. Golden. "A Unified Formalism for Complex Systems Architecture". PhD thesis. Ecole Polytechnique X, 2013.

[84] O. Gotel, J. Cleland-Huang, J. H. Hayes, A. Zisman, A. Egyed, P. Grünbacher, A. Dekhtyar, G. Antoniol, and J. Maletic. "The Grand Challenge of Traceability (v1.0)". In: *Software and Systems Traceability.* Springer London, 2012, pp. 343–409.

[85] O. Gotel, J. Cleland-Huang, J. H. Hayes, A. Zisman, A. Egyed, P. Grünbacher, A. Dekhtyar, G. Antoniol, J. Maletic, and P. Mäder. "Traceability Fundamentals". In: *Software and Systems Traceability.* Springer London, 2012, pp. 3–22.

[86] M. Grootendorst. *BERTopic: Neural topic modeling with a class-based TF-IDF procedure*. 2022.

[87] T. Gruber. "A translation approach to portable ontology specifications". In: *Knowledge Acquisition* 5.2 (1993), pp. 199–220.

[88] N. Guarino, D. Oberle, and S. Staab. "What Is an Ontology?" In: *Handbook on Ontologies.* Springer Berlin Heidelberg, 2009, pp. 1–17.

[89] J. Guo, J. Cheng, and J. Cleland-Huang. “Semantically Enhanced Software Traceability Using Deep Learning Techniques”. In: *Proceedings of the 39th International Conference on Software Engineering*. ICSE ’17. IEEE Press, 2017, pp. 3–14.

[90] K. L. Gwet. *Handbook of inter-rater reliability: The definitive guide to measuring the extent of agreement among raters*. Advanced Analytics, LLC, 2014.

[91] Z. S. Harris. *Distributional Structure*. 1954.

[92] T. Hastie, R. Tibshirani, and J. Friedman. “Unsupervised Learning”. In: *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer New York, 2009, pp. 485–585.

[93] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram. “Advancing candidate link generation for requirements tracing: the study of methods”. In: *IEEE Transactions on Software Engineering* 32.1 (2006), pp. 4–19.

[94] R. Hebig, T. H. Quang, M. R. V. Chaudron, G. Robles, and M. A. Fernandez. “The Quest for Open Source Projects That Use UML: Mining GitHub”. In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. MODELS ’16. Association for Computing Machinery, 2016, pp. 173–183.

[95] T.-M. Hesse, A. Kuehlwein, and T. Roehm. “DecDoc: A Tool for Documenting Design Decisions Collaboratively and Incrementally”. In: *2016 1st International Workshop on Decision Making in Software ARCHitecture (MARCH)*. 2016, pp. 30–37.

[96] T. Hey. “Automatische Wiederherstellung von Nachverfolgbarkeit zwischen Anforderungen und Quelltext”. PhD thesis. Karlsruher Institut für Technologie (KIT), 2023. 314 pp.

[97] T. Hey, F. Chen, S. Weigelt, and W. F. Tichy. “Improving Traceability Link Recovery Using Fine-grained Requirements-to-Code Relations”. In: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME). 2021, pp. 12–22.

[98] T. Hey, J. Keim, A. Koziolek, and W. F. Tichy. “NoRBERT: Transfer Learning for Requirements Classification”. In: *2020 IEEE 28th International Requirements Engineering Conference (RE)*. 2020, pp. 169–179.

[99] G. E. Hinton and S. T. Roweis. “Stochastic Neighbor Embedding”. In: *Advances in Neural Information Processing Systems 15 [Neural Information Processing Systems, NIPS 2002]*. MIT Press, 2002, pp. 833–840.

[100] C. A. R. Hoare. “Communicating Sequential Processes”. In: *Commun. ACM* 21.8 (1978), pp. 666–677.

[101] S. Hochreiter and J. Schmidhuber. “Long Short-Term Memory”. In: *Neural Computation* 9.8 (1997), pp. 1735–1780.

[102] J. Howard and S. Ruder. “Universal Language Model Fine-tuning for Text Classification”. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, 2018, pp. 328–339.

[103] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt. “CodeSearchNet Challenge: Evaluating the State of Semantic Code Search”. In: *ArXiv preprint* abs/1909.09436 (2019).

[104] IEEE. “IEEE Standard for Technical Reviews and Audits on Defense Programs”. In: *IEEE Std 15288.2-2014* (2015), pp. 1–168.

[105] ISO/IEC. “Information technology – Open distributed processing – Reference model: Foundations”. In: *ISO/IEC 10746-2:2009* (2009).

[106] ISO/IEC. “ISO/IEC TR 19759:2015 Software Engineering – Guide to the Software Engineering Body of Knowledge (SWEBOK)”. In: *ISO/IEC TR 19759:2015* (2015).

[107] ISO/IEC. “Road vehicles – Functional safety”. In: *ISO/IEC 26262:2018* (2018).

[108] ISO/IEC. “Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Quality model overview and usage”. In: *ISO/IEC 25002:2024* (2024).

[109] ISO/IEC/IEEE. “ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary”. In: *ISO/IEC/IEEE 24765:2017(E)* (2017), pp. 1–541.

[110] ISO/IEC/IEEE. “ISO/IEC/IEEE Systems and Software Engineering – Architecture Description”. In: *ISO/IEC/IEEE 42010:2011(E)* (2011).

[111] P. Jaccard. “Lois de distribution florale dans la zone alpine”. In: *Bull Soc Vaudoise Sci Nat* 38 (1902), pp. 69–130.

[112] A. Jansen and J. Bosch. “Software Architecture as a Set of Architectural Design Decisions”. In: *5th Working IEEE/IFIP Conference on Software Architecture (WICSA’05).* 2005, pp. 109–120.

[113] A. Jansen, J. Bosch, and P. Avgeriou. “Documenting after the fact: Recovering architectural design decisions”. In: *Journal of Systems and Software* 81.4 (2008), pp. 536–557.

[114] A. Jansen, J. van der Ven, P. Avgeriou, and D. K. Hammer. “Tool Support for Architectural Decisions”. In: *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA’07).* 2007, pp. 4–4.

[115] T. de Jong and J. M. E. M. van der Werf. “Process-Mining Based Dynamic Software Architecture Reconstruction”. In: *13th European Conference on Software Architecture.* Association for Computing Machinery, 2019, pp. 217–224.

[116] M. Joshi, D. Chen, Y. Liu, D. S. Weld, L. Zettlemoyer, and O. Levy. “SpanBERT: Improving Pre-training by Representing and Predicting Spans”. In: *Transactions of the Association for Computational Linguistics* 8 (2020), pp. 64–77.

[117] D. Jurafsky and J. H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.* 2nd. Prentice Hall, 2008.

[118] M. Kamalrudin, J. Grundy, and J. Hosking. “Managing Consistency between Textual Requirements, Abstract Interactions and Essential Use Cases”. In: *IEEE Annual Computer Software and Applications Conference.* 2010, pp. 327–336.

[119] A. Kaplan, T. Kühn, S. Hahner, N. Benkler, J. Keim, D. Fuchß, S. Corallo, and R. Heinrich. “Introducing an Evaluation Method for Taxonomies”. In: *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering 2022.* EASE ’22. Association for Computing Machinery, 2022, pp. 311–316.

[120] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. “Scaling Laws for Neural Language Models”. In: *ArXiv preprint* abs/2001.08361 (2020).

[121] T. Kasami. “An efficient recognition and syntax-analysis algorithm for context-free languages”. In: *Coordinated Science Laboratory Report no. R-257* (1966).

[122] M. Keeling. “Love Unrequited: The Story of Architecture, Agile, and How Architecture Decision Records Brought Them Together”. In: *IEEE Software* 39.4 (2022), pp. 90–93.

[123] J. Keim, S. Corallo, D. Fuchß, T. Hey, T. Telge, and A. Koziolek. “Recovering Trace Links Between Software Documentation And Code”. In: *Proceedings of 46th IEEE International Conference on Software Engineering (ICSE 2024).* 2024.

[124] J. Keim, S. Corallo, D. Fuchß, and A. Koziolek. “Detecting Inconsistencies in Software Architecture Documentation Using Traceability Link Recovery”. In: *2023 IEEE 20th International Conference on Software Architecture (ICSA).* 2023, pp. 141–152.

[125] J. Keim, T. Hey, B. Sauer, and A. Koziolek. *A Taxonomy for Design Decisions in Software Architecture Documentation.* Tech. rep. 2022.

[126] J. Keim, T. Hey, B. Sauer, and A. Koziolek. *Supplementary Material of "A Taxonomy for Design Decisions in Software Architecture Documentation".* 2022.

[127] J. Keim, A. Kaplan, A. Koziolek, and M. Mirakhorli. “Does BERT Understand Code?– An Exploratory Study On The Detection Of Architectural Tactics In Code”. In: *Software Architecture. ECSA 2020.* Springer International Publishing. 2020, pp. 220–228.

[128] J. Keim and A. Koziolek. “Towards Consistency Checking Between Software Architecture and Informal Documentation”. In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C).* 2019, pp. 250–253.

[129] J. Keim, S. Schulz, D. Fuchß, C. Kocher, J. Speit, and A. Koziolek. “Tracelink Recovery for Software Architecture Documentation”. In: *Software Architecture. ECSA 2021.* Springer International Publishing, 2021, pp. 101–116.

[130] S. Kim and D. Kim. “Automatic Identifier Inconsistency Detection Using Code Dictionary”. In: *Emp. Softw. Engg.* 21.2 (2016), pp. 565–604.

[131] Y. R. Kirschner, M. Gstür, T. Sağlam, S. Weber, and A. Koziolek. “Retriever: A view-based approach to reverse engineering software architecture models”. In: *Journal of Systems and Software* 220 (2025), p. 112277.

[132] Y. R. Kirschner, J. Keim, N. Peter, and A. Koziolek. “Automated Reverse Engineering of the Technology-Induced Software System Structure”. In: *Software Architecture.* Springer Nature Switzerland, 2023, pp. 283–291.

[133] Y. Kirstain, O. Ram, and O. Levy. “Coreference Resolution without Span Representations”. In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 2: Short Papers).* Association for Computational Linguistics, 2021, pp. 14–19.

[134] J. von Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, and P. Cao. “How to Build a Benchmark”. In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering.* ICPE ’15. Association for Computing Machinery, 2015, pp. 333–336.

[135] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev. “TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research”. In: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS).* 2018, pp. 223–236.

[136] H. Klare. “Multi-model consistency preservation”. In: *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings.* MODELS ’18. Association for Computing Machinery, 2018, pp. 156–161.

[137] A. Kleebaum, B. Paech, J. O. Johanssen, and B. Bruegge. “Continuous Rationale Identification in Issue Tracking and Version Control Systems”. In: *Joint Proceedings of REFSQ-2021 Workshops, OpenRE, Posters and Tools Track, and Doctoral Symposium (to appear).* 2021.

[138] J. Kober, J. A. Bagnell, and J. Peters. “Reinforcement learning in robotics: A survey”. In: *The International Journal of Robotics Research* 32.11 (2013), pp. 1238–1274.

[139] V. Konda and J. Tsitsiklis. “Actor-Critic Algorithms”. In: *Advances in Neural Information Processing Systems.* Vol. 12. MIT Press, 1999.

[140] M. Konersmann, A. Kaplan, T. Kühn, R. Heinrich, A. Koziolek, R. Reussner, J. Jürjens, M. al-Doori, N. Boltz, M. Ehl, D. Fuchs, K. Groser, S. Hahner, J. Keim, M. Lohr, T. Sağlam, S. Schulz, and J.-P. Töberg.

"Evaluation Methods and Replicability of Software Architecture Research Objects". In: *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. 2022, pp. 157–168.

[141] H. Koziolek, J. Happe, S. Becker, and R. Reussner. "Evaluating Performance of Software Architecture Models with the Palladio Component Model". In: (2008).

[142] A. Kozlenkov and A. Zisman. "Are their design specifications consistent with our requirements?" In: *Proceedings IEEE Joint International Conference on Requirements Engineering*. 2002, pp. 145–154.

[143] K. Krippendorff. *Content Analysis: An Introduction to Its Methodology*. SAGE Publications, 2018.

[144] K. Krippendorff. "Reliability in content analysis: Some common misconceptions and recommendations". In: *Human communication research* 30.3 (2004), pp. 411–433.

[145] P. Kruchten. "The 4+1 View Model of architecture". In: *IEEE Software* 12.6 (1995), pp. 42–50.

[146] P. Kruchten. "An Ontology of Architectural Design Decisions in Software-Intensive Systems". In: *2nd Groningen Workshop on Software Variability* (2004), pp. 54–61.

[147] H. Kuang, P. Mäder, H. Hu, A. Ghabi, L. Huang, J. Lü, and A. Egyed. "Can Method Data Dependencies Support the Assessment of Traceability Between Requirements and Source Code?" In: *J. Softw. Evol. Process* 27.11 (2015), pp. 838–866.

[148] T. Kühn, D. Fuchß, S. Corallo, L. König, E. Burger, J. Keim, M. Mazkatli, T. Saglam, F. Reiche, A. Koziolek, and R. Reussner. *A formalized classification schema for model consistency : technical report*. 2023.

[149] Z. Kurtanović and W. Maalej. "Automatically Classifying Functional and Non-Functional Requirements Using Supervised Machine Learning". In: *2017 IEEE 25th International Requirements Engineering Conference (RE)*. 2017, pp. 490–495.

[150] M. J. Kusner, Y. Sun, N. I. Kolkin, and K. Q. Weinberger. "From Word Embeddings To Document Distances". In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015*. Vol. 37. JMLR Workshop and Conference Proceedings. JMLR.org, 2015, pp. 957–966.

[151] P. Laban, T. Schnabel, P. N. Bennett, and M. A. Hearst. “SummaC: Re-Visiting NLI-based Models for Inconsistency Detection in Summarization”. In: *Transactions of the Association for Computational Linguistics* 10 (2022), pp. 163–177.

[152] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. “Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports (N)”. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 2015, pp. 476–481.

[153] J. R. Landis and G. G. Koch. “The measurement of observer agreement for categorical data”. In: *biometrics* (1977), pp. 159–174.

[154] M. Langhammer, A. Shahbazian, N. Medvidovic, and R. H. Reussner. “Automated extraction of rich software models from limited system information”. In: *13th Working IEEE/IFIP Conference on Software Architecture (WICSA).* IEEE. 2016, pp. 99–108.

[155] T.-D. B. Le, R. J. Oentaryo, and D. Lo. “Information retrieval and spectrum based bug localization: better together”. In: *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering.* ESEC/FSE 2015. Association for Computing Machinery, 2015, pp. 579–590.

[156] V. I. Levenshtein et al. “Binary codes capable of correcting deletions, insertions, and reversals”. In: *Soviet physics doklady.* Vol. 10. 8. Soviet Union. 1966, pp. 707–710.

[157] J. J. Li and J. R. Horgan. “To maintain a reliable software specification”. In: *International Symposium on Software Reliability Engineering.* IEEE. 1998, pp. 59–68.

[158] X. Li, P. Liang, and Z. Li. “Automatic Identification of Decisions from the Hibernate Developer Mailing List”. In: *Proceedings of EASE ’20.* ACM, 2020, pp. 51–60.

[159] J. Lin, Y. Liu, Q. Zeng, M. Jiang, and J. Cleland-Huang. “Traceability Transformed: Generating More Accurate Links with Pre-Trained BERT Models”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE).* 2021, pp. 324–335.

[160] J. Lin, Y. Liu, Q. Zeng, M. Jiang, and J. Cleland-Huang. “Traceability transformed: Generating more accurate links with pre-trained BERT models”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE).* IEEE. 2021, pp. 324–335.

[161] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2019.

[162] S. Lohar, S. Amornborvornwong, A. Zisman, and J. Cleland-Huang. "Improving Trace Accuracy Through Data-driven Configuration and Composition of Tracing Features". In: *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. ACM, 2013, pp. 378–388.

[163] C. M. Lüders, A. Bouraffa, and W. Maalej. "Beyond duplicates: towards understanding and predicting link types in issue tracking systems". In: *Proceedings of the 19th International Conference on Mining Software Repositories*. MSR '22. Association for Computing Machinery, 2022, pp. 48–60.

[164] C. M. Lüders, T. Pietz, and W. Maalej. "Automated Detection of Typed Links in Issue Trackers". In: *2022 IEEE 30th International Requirements Engineering Conference (RE)*. 2022, pp. 26–38.

[165] C. M. Lüders, T. Pietz, and W. Maalej. "On understanding and predicting issue links". In: *Requirements Engineering* 28.4 (2023), pp. 541–565.

[166] I. Lytra and U. Zdun. "Inconsistency Management between Architectural Decisions and Designs Using Constraints and Model Fixes". In: *2014 23rd Australian Software Engineering Conference*. 2014, pp. 230–239.

[167] P. Mäder and A. Egyed. "Assessing the effect of requirements traceability for software maintenance". In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. 2012, pp. 171–180.

[168] P. Mäder and A. Egyed. "Do developers benefit from requirements traceability when evolving and maintaining a software system?" In: *Empirical Software Engineering* 20 (2015), pp. 413–441.

[169] P. Mäder, P. L. Jones, Y. Zhang, and J. Cleland-Huang. "Strategic Traceability for Safety-Critical Projects". In: *IEEE Software* 30.3 (2013), pp. 58–66.

[170] D. Mahato, D. Dudhal, D. Revagade, and Y. Bhargava. “A Method to Detect Inconsistent Annotations in a Medical Document using UMLS”. In: *Proceedings of the 11th Annual Meeting of the Forum for Information Retrieval Evaluation.* FIRE ’19. Association for Computing Machinery, 2019, pp. 47–51.

[171] A. Mahmoud and N. Niu. “On the Role of Semantics in Automated Requirements Tracing”. In: *Requirements Eng* 20.3 (2015), pp. 281–300.

[172] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang. “What industry needs from architectural languages: A survey”. In: *IEEE Transactions on Software Engineering* 39.6 (2012), pp. 869–891.

[173] C. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. Bethard, and D. McClosky. “The Stanford CoreNLP Natural Language Processing Toolkit”. In: *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations.* Association for Computational Linguistics, 2014, pp. 55–60.

[174] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[175] A. Marcus and J. I. Maletic. “Recovering Documentation-to-source-code Traceability Links Using Latent Semantic Indexing”. In: *Proceedings of the 25th International Conference on Software Engineering.* ICSE ’03. IEEE Computer Society, 2003, pp. 125–135.

[176] L. Màrquez, X. Carreras, K. C. Litkowski, and S. Stevenson. “Semantic Role Labeling: An Introduction to the Special Issue”. In: *Computational Linguistics* 34.2 (2008), pp. 145–159.

[177] B. W. Matthews. “Comparison of the predicted and observed secondary structure of T4 phage lysozyme”. In: *Biochimica et Biophysica Acta (BBA)-Protein Structure* 405.2 (1975), pp. 442–451.

[178] C. Mayr-Dorn, M. Vierhauser, S. Bichler, F. Keplinger, J. Cleland-Huang, A. Egyed, and T. Mehofer. “Supporting Quality Assurance with Automated Process-Centric Quality Constraints Checking”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE).* 2021, pp. 1298–1310.

[179] M. Mazkatli, D. Monschein, M. Armbruster, R. Heinrich, and A. Koziolek. “Continuous Integration of Architectural Performance Models with Parametric Dependencies–The CIPM Approach”. In: (2022).

[180] M. L. McHugh. “Interrater reliability: the kappa statistic”. In: *Biochemia medica* 22.3 (2012), pp. 276–282.

[181] N. Medvidovic and R. N. Taylor. “Software architecture: foundations, theory, and practice”. In: *2010 ACM/IEEE 32nd International Conference on Software Engineering*. Vol. 2. 2010, pp. 471–472.

[182] C. Miesbauer and R. Weinreich. “Classification of Design Decisions – An Expert Survey in Practice”. In: *Software Architecture*. LNCS. Springer, 2013, pp. 130–145.

[183] T. Mikolov, E. Grave, P. Bojanowski, C. Puhrsch, and A. Joulin. “Advances in Pre-Training Distributed Word Representations”. In: *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*. European Language Resources Association (ELRA), 2018.

[184] T. Mikolov, K. Chen, G. Corrado, and J. Dean. “Efficient Estimation of Word Representations in Vector Space”. In: *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*. 2013.

[185] G. A. Miller. “WordNet: A Lexical Database for English”. In: *Speech and Natural Language: Proceedings of a Workshop Held at Harriman, New York*. 1992.

[186] C. Mills, J. Escobar-Avila, and S. Haiduc. “Automatic Traceability Maintenance via Machine Learning Classification”. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018, pp. 369–380.

[187] C. Mills, J. Escobar-Avila, A. Bhattacharya, G. Kondyukov, S. Chakraborty, and S. Haiduc. “Tracing with Less Data: Active Learning for Classification-Based Traceability Link Recovery”. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019, pp. 103–113.

[188] M. Mirakhorli and J. Cleland-Huang. “Detecting, Tracing, and Monitoring Architectural Tactics in Code”. In: *IEEE Transactions on Software Engineering* 42.3 (2016), pp. 205–220.

[189] T. M. Mitchell. *Machine learning*. McGraw-Hill Education, 1997.

[190] S. Molenaar, T. Spijkman, F. Dalpiaz, and S. Brinkkemper. "Explicit Alignment of Requirements and Architecture in Agile Development". In: *REFSQ 2020*. LNCS. Springer International Publishing, 2020, pp. 169–185.

[191] K. Moran, D. N. Palacio, C. Bernal-Cárdenas, D. McCrystal, D. Poshyvanyk, C. Shenefiel, and J. Johnson. "Improving the Effectiveness of Traceability Link Recovery Using Hierarchical Bayesian Networks". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE '20. Association for Computing Machinery, 2020, pp. 873–885.

[192] S. Nejati, M. Sabetzadeh, D. Falessi, L. Briand, and T. Coq. "A SysML-based approach to traceability management and design slicing in support of safety certification: Framework, tool support, and case studies". In: *Information and Software Technology* 54.6 (2012), pp. 569–590.

[193] A. Nhlabatsi, Y. Yu, A. Zisman, T. Tun, N. Khan, A. Bandara, K. M. Khan, and B. Nuseibeh. "Managing Security Control Assumptions Using Causal Traceability". In: *2015 IEEE/ACM 8th International Symposium on Software and Systems Traceability*. 2015, pp. 43–49.

[194] K. Nishikawa, H. Washizaki, Y. Fukazawa, K. Oshima, and R. Mibe. "Recovering transitive traceability links among software artifacts". In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2015, pp. 576–580.

[195] J. Nivre, D. Zeman, F. Ginter, and F. Tyers. "Universal Dependencies". In: *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Tutorial Abstracts*. Association for Computational Linguistics, 2017.

[196] B. Nuseibeh, S. Easterbrook, and A. Russo. "Leveraging inconsistency in software development". In: *Computer* 33.4 (2000), pp. 24–29.

[197] B. Nuseibeh, S. Easterbrook, and A. Russo. "Making inconsistency respectable in software development". In: *Journal of Systems and Software* 58.2 (2001), pp. 171–180.

[198] Object Management Group (OMG). "Architecture-Driven Modernization: Knowledge Discovery Meta-Model (KDM)". In: *OMG.org* (2016).

[199] J. W. Oller. "On the relation between syntax, semantics, and pragmatics". In: *Linguistics* 10.83 (1972), pp. 43–55.

[200] T. Olsson and J. Grundy. “Supporting traceability and inconsistency management between software artifacts”. In: *6th International Conference on Software Engineering and Applications* (2002), pp. 484–489.

[201] T. Olsson, M. Ericsson, and A. Wingkvist. “Semi-Automatic Mapping of Source Code Using Naive Bayes”. In: *13th European Conference on Software Architecture.* Association for Computing Machinery, 2019, pp. 209–216.

[202] C. Pacheco, S. K. Lahiri, and T. Ball. “Finding errors in .net with feedback-directed random testing”. In: *Proceedings of the 2008 International Symposium on Software Testing and Analysis.* ISSTA ’08. Association for Computing Machinery, 2008, pp. 87–96.

[203] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. “Feedback-Directed Random Test Generation”. In: *29th International Conference on Software Engineering (ICSE’07).* 2007, pp. 75–84.

[204] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. D. Lucia. “When and How Using Structural Information to Improve IR-Based Traceability Recovery”. In: *2013 17th European Conference on Software Maintenance and Reengineering.* 2013, pp. 199–208.

[205] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshynanyk, and A. De Lucia. “How to effectively use topic models for software engineering tasks? An approach based on Genetic Algorithms”. In: *2013 35th International Conference on Software Engineering (ICSE).* 2013, pp. 522–531.

[206] S. Panthaplackel, J. J. Li, M. Gligoric, and R. J. Mooney. “Deep Just-In-Time Inconsistency Detection Between Comments and Source Code”. In: *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021.* AAAI Press, 2021, pp. 427–435.

[207] C. H. Papadimitriou, H. Tamaki, P. Raghavan, and S. Vempala. “Latent semantic indexing: A probabilistic analysis”. In: *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems.* 1998, pp. 159–168.

[208] D. Parnas. “Software aging”. In: *Proceedings of 16th International Conference on Software Engineering.* 1994, pp. 279–287.

[209] D. L. Parnas. “Precise Documentation: The Key to Better Software”. In: *The Future of Software Engineering*. Springer, 2011, pp. 125–148.

[210] H. Paulheim. “Knowledge graph refinement: A survey of approaches and evaluation methods”. In: *Semantic web* 8.3 (2016), pp. 489–508.

[211] Z. Pauzi and A. Capiluppi. “Applications of natural language processing in software traceability: A systematic mapping study”. In: *Journal of Systems and Software* 198 (2023), p. 111616.

[212] K. Pearson. “LIII. On lines and planes of closest fit to systems of points in space”. In: *The London, Edinburgh, and Dublin philosophical magazine and journal of science* 2.11 (1901), pp. 559–572.

[213] K. Pearson. “VII. Note on regression and inheritance in the case of two parents”. In: *proceedings of the royal society of London* 58.347-352 (1895), pp. 240–242.

[214] K. Pearson. “X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling”. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 50.302 (1900), pp. 157–175.

[215] J. Pennington, R. Socher, and C. Manning. “GloVe: Global Vectors for Word Representation”. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2014, pp. 1532–1543.

[216] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. “Deep Contextualized Word Representations”. In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. Association for Computational Linguistics, 2018, pp. 2227–2237.

[217] M. F. Porter. “An algorithm for suffix stripping”. In: *Program* 14.3 (1980), pp. 130–137.

[218] D. Poshyvanyk, M. Gethers, and A. Marcus. “Concept location using formal concept analysis and information retrieval”. In: *ACM Trans. Softw. Eng. Methodol.* 21.4 (2013).

[219] S. Pradhan, A. Moschitti, N. Xue, H. T. Ng, A. Björkelund, O. Uryupina, Y. Zhang, and Z. Zhong. "Towards Robust Linguistic Analysis using OntoNotes". In: *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*. Association for Computational Linguistics, 2013, pp. 143–152.

[220] S. S. Pradhan, E. Hovy, M. Marcus, M. Palmer, L. Ramshaw, and R. Weischedel. "OntoNotes: A Unified Relational Semantic Representation". In: *International Conference on Semantic Computing (ICSC 2007)*. 2007, pp. 517–526.

[221] L. J. Pruijt, C. Köppe, J. M. van der Werf, and S. Brinkkemper. "HUSACCT: architecture compliance checking with rich sets of module and rule types". In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE '14. Association for Computing Machinery, 2014, pp. 851–854.

[222] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. "Improving language understanding with unsupervised learning". In: (2018).

[223] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. "Language models are unsupervised multitask learners". In: *OpenAI blog* 1.8 (2019), p. 9.

[224] P. Ralph. "Toward Methodological Guidelines for Process Theories and Taxonomies in Software Engineering". In: *IEEE Transactions on Software Engineering* 45.7 (2019), pp. 712–735.

[225] B. Ramesh and M. Jarke. "Toward reference models for requirements traceability". In: *IEEE Transactions on Software Engineering* 27.1 (2001), pp. 58–93.

[226] M. Rath, D. Lo, and P. Mäder. "Analyzing Requirements and Traceability Information to Improve Bug Localization". In: *Proceedings of the 15th International Conference on Mining Software Repositories*. MSR '18. Association for Computing Machinery, 2018, pp. 442–453.

[227] M. Rath, J. Rendall, J. L. C. Guo, J. Cleland-Huang, and P. Mäder. "Traceability in the Wild: Automatically Augmenting Incomplete Trace Links". In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. ACM, 2018, pp. 834–845.

[228] C. C. Raţiu, W. K. G. Assunção, R. Haas, and A. Egyed. "Reactive Links across Multi-Domain Engineering Models". In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems.* MODELS '22. Association for Computing Machinery, 2022, pp. 76–86.

[229] P. Rempel and P. Mäder. "Preventing Defects: The Impact of Requirements Traceability Completeness on Software Quality". In: *IEEE Transactions on Software Engineering* 43.8 (2017), pp. 777–797.

[230] P. Rempel and P. Mäder. "Estimating the Implementation Risk of Requirements in Agile Software Development Projects with Traceability Metrics". In: *REFSQ 2015.* Springer International Publishing, 2015, pp. 81–97.

[231] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, and K. Krogmann. *Modeling and simulating software architectures: The Palladio approach.* MIT Press, 2016.

[232] A. D. Rodriguez, J. Cleland-Huang, and D. Falessi. "Leveraging Intermediate Artifacts to Improve Automated Trace Link Retrieval". In: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME).* 2021, pp. 81–92.

[233] D. V. Rodriguez and D. L. Carver. "Multi-Objective Information Retrieval-Based NSGA-II Optimization for Requirements Traceability Recovery". In: *2020 IEEE EIT.* 2020 IEEE EIT. ISSN: 2154-0373. 2020, pp. 271–280.

[234] M. Ruiz, J. Y. Hu, and F. Dalpiaz. "Why don't we trace? A study on the barriers to software traceability in practice". In: *Requirements Engineering* (2023), pp. 1–19.

[235] P. Runeson and M. Höst. "Guidelines for Conducting and Reporting Case Study Research in Software Engineering". In: *Empir Software Eng* 14.2 (2008), p. 131.

[236] R. Salakhutdinov. "Deep learning". In: *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14.* ACM, 2014, p. 1973.

[237] J. Sallou, T. Durieux, and A. Panichella. “Breaking the Silence: the Threats of Using LLMs in Software Engineering”. In: *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results.* ICSE-NIER’24. Association for Computing Machinery, 2024, pp. 102–106.

[238] A. Schlutter and A. Vogelsang. “Improving Trace Link Recovery Using Semantic Relation Graphs and Spreading Activation”. In: *Requirements Engineering: Foundation for Software Quality.* Springer International Publishing, 2021, pp. 37–53.

[239] A. Schlutter and A. Vogelsang. “Trace Link Recovery using Semantic Relation Graphs and Spreading Activation”. In: *2020 IEEE 28th International Requirements Engineering Conference (RE).* 2020, pp. 20–31.

[240] M. Schmitt Laser, N. Medvidovic, D. M. Le, and J. Garcia. “ARCADE: an extensible workbench for architecture recovery, change, and decay evaluation”. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 2020, pp. 1546–1550.

[241] S. Schröder and M. Riebisch. “An Ontology-Based Approach for Documenting and Validating Architecture Rules”. In: *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings.* ECSA ’18. Association for Computing Machinery, 2018.

[242] S. Schröder and M. Riebisch. “Architecture conformance checking with description logics”. In: *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings.* ECSA ’17. Association for Computing Machinery, 2017, pp. 166–172.

[243] R. Sennrich, B. Haddow, and A. Birch. “Neural Machine Translation of Rare Words with Subword Units”. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* Association for Computational Linguistics, 2016, pp. 1715–1725.

[244] A. Shahbazian, Y. Kyu Lee, D. Le, Y. Brun, and N. Medvidovic. “Recovering Architectural Design Decisions”. In: *2018 IEEE International Conference on Software Architecture (ICSA).* 2018, pp. 95–104.

[245] M. Shahin, P. Liang, and M. R. Khayyambashi. “Architectural Design Decision: Existing Models and Tools”. In: *2009 Joint WICSA & ECSA.* 2009, pp. 293–296.

[246] Y. Shin, J. H. Hayes, and J. Cleland-Huang. "Guidelines for Benchmarking Automated Software Traceability Techniques". In: *2015 IEEE/ACM 8th International Symposium on Software and Systems Traceability*. 2015, pp. 61–67.

[247] S. E. Sim, S. Easterbrook, and R. C. Holt. "Using Benchmarking to Advance Research: A Challenge to Software Engineering". In: *Proceedings of the 25th International Conference on Software Engineering*. ICSE '03. IEEE Computer Society, 2003, pp. 74–83.

[248] K. Souali, O. Rahmaoui, and M. Ouzzif. "An overview of traceability: Definitions and techniques". In: *2016 4th IEEE International Colloquium on Information Science and Technology (CiSt)*. 2016, pp. 789–793.

[249] C. Spearman. "The Proof and Measurement of Association between Two Things". In: *The American Journal of Psychology* 15.1 (1904), pp. 72–101.

[250] J. Speit. "Hierarchical Classification of Design Decisions using pretrained Language Models". MA thesis. Karlsruher Institut für Technologie (KIT), 2023. 79 pp.

[251] G. Starke, M. Simons, S. Zörner, R. D. Müller, and H. Lösch. *arc42 by Example: Software architecture documentation in practice*. Leanpub, 2023.

[252] P. Stevens. "Bidirectional Transformations in the Large". In: *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2017, pp. 1–11.

[253] P. Stevens. "Maintaining consistency in networks of models: bidirectional transformations in the large". In: *Softw. Syst. Model.* 19.1 (2020), pp. 39–65.

[254] M. Strittmatter and A. Kechaou. *The Media Store 3 Case Study System*. Tech. rep. 1. Karlsruher Institut für Technologie (KIT), 2016. 35 pp.

[255] Student. "The probable error of a mean". In: *Biometrika* 6.1 (1908), pp. 1–25.

[256] N. Stulova, A. Blasi, A. Gorla, and O. Nierstrasz. "Towards Detecting Inconsistent Comments in Java Source Code Automatically". In: *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2020, pp. 65–69.

[257] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. “The Structure and Value of Modularity in Software Design”. In: *SIGSOFT SE Notes* 26.5 (2001), pp. 99–108.

[258] R. S. Sutton. “Reinforcement learning: An introduction”. In: *A Bradford Book* (2018).

[259] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Pearson Education, 2002.

[260] J. Tabassum, M. Maddela, W. Xu, and A. Ritter. “Code and Named Entity Recognition in StackOverflow”. In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics.* Association for Computational Linguistics, 2020, pp. 4913–4926.

[261] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. “tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies”. In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation.* IEEE, 2012.

[262] A. Tang, P. Liang, V. Clerc, and H. van Vliet. “Traceability in the Co-evolution of Architectural Requirements and Design”. In: *Relating Software Requirements and Architectures.* Springer, 2011, pp. 35–60.

[263] “Taxonomies in software engineering: A Systematic mapping study and a revised taxonomy development method”. In: *Information and Software Technology* 85 (2017), pp. 43–59.

[264] A. Taylor, M. Marcus, and B. Santorini. “The Penn treebank: an overview”. In: *Treebanks: Building and using parsed corpora* (2003), pp. 5–22.

[265] T. Telge. “Automatisierte Gewinnung von Nachverfolgbarkeitsverbindungen zwischen Softwarearchitektur und Quelltext”. MA thesis. Karlsruher Institut für Technologie (KIT), 2023. 87 pp.

[266] B. Tesar. “Using Inconsistency Detection to Overcome Structural Ambiguity”. In: *Linguistic Inquiry* 35.2 (2004), pp. 219–253.

[267] J. Thomas, A. Nicolaescu, and H. Lichter. “Static and Dynamic Architecture Conformance Checking: A Systematic, Case Study-Based Analysis on Tradeoffs and Synergies.” In: *QuASoQ APSEC.* 2017, pp. 6–13.

[268] F. Tian, P. Liang, and M. A. Babar. “Relationships between software architecture and source code in practice: An exploratory survey and interview”. In: *Information and Software Technology* 141 (2022), p. 106705.

[269] Y. Tian, D. Lo, and J. Lawall. “SEWordSim: Software-Specific Word Similarity Database”. In: *Companion Proceedings of the 36th International Conference on Software Engineering.* ICSE Companion 2014. Association for Computing Machinery, 2014, pp. 568–571.

[270] E. F. Tjong Kim Sang. “Introduction to the CoNLL-2002 Shared Task: Language-Independent Named Entity Recognition”. In: *COLING-02: The 6th Conference on Natural Language Learning 2002 (CoNLL-2002).* 2002.

[271] F. G. Toosi, J. Buckley, and A. R. Sai. “Source-Code Divergence Diagnosis Using Constraints and Cryptography”. In: *Proceedings of the 13th European Conference on Software Architecture - Volume 2.* Association for Computing Machinery, 2019, pp. 205–208.

[272] G. Töpper, M. Knuth, and H. Sack. “DBpedia ontology enrichment for inconsistency detection”. In: *Proceedings of the 8th International Conference on Semantic Systems.* I-SEMANTICS ’12. Association for Computing Machinery, 2012, pp. 33–40.

[273] H. Tran, U. Zdun, and S. Dustdar. “View-based and Model-driven Approach for Reducing the Development Complexity in Process-Driven SOA”. In: *Proceedings on the BPSC 2007.* Gesellschaft für Informatik. 2007, pp. 105–124.

[274] R. Tsuchiya, K. Nishikawa, H. Washizaki, Y. Fukazawa, Y. Shinohara, K. Oshima, and R. Mibe. “Recovering transitive traceability links among various software artifacts for developers”. In: *IEICE TRANSACTIONS on Information and Systems* 102.9 (2019), pp. 1750–1760.

[275] J. Tyree and A. Akerman. “Architecture decisions: demystifying architecture”. In: *IEEE Software* 22.2 (2005), pp. 19–27.

[276] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017.* 2017, pp. 5998–6008.

[277] W. Von Humboldt. *On language: On the diversity of human language construction and its influence on the mental development of the human species.* 1836.

[278] Z. Wan, Y. Zhang, X. Xia, Y. Jiang, and D. Lo. "Software Architecture in Practice: Challenges and Opportunities". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* ESEC/FSE 2023. Association for Computing Machinery, 2023, pp. 1457–1469.

[279] S. Wang, D. Lo, and J. Lawall. "Compositional Vector Space Models for Improved Bug Localization". In: *2014 IEEE International Conference on Software Maintenance and Evolution.* 2014, pp. 171–180.

[280] S. Wang, Y. Zhou, Z. Han, C. Tao, Y. Xiao, Y. Ding, J. Ghosh, and Y. Peng. *Uncovering Misattributed Suicide Causes through Annotation Inconsistency Detection in Death Investigation Notes.* 2024.

[281] W. Wang, N. Niu, H. Liu, and Z. Niu. "Enhancing Automated Requirements Traceability by Resolving Polysemy". In: *IEEE 26th RE.* 2018, pp. 40–51.

[282] X. Wang, Y. Jiang, N. Bach, T. Wang, Z. Huang, F. Huang, and K. Tu. "Automated Concatenation of Embeddings for Structured Prediction". In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers).* Association for Computational Linguistics, 2021, pp. 2643–2660.

[283] C. J. C. H. Watkins and P. Dayan. "Q-learning". In: *Machine Learning* 8.3–4 (1992), pp. 279–292.

[284] S. Weigelt, J. Keim, T. Hey, and W. F. Tichy. "Unsupervised Multi-Topic Labeling for Spoken Utterances". In: *2019 IEEE International Conference on Humanized Computing and Communication (HCC).* 2019, pp. 38–45.

[285] R. Weinreich, I. Groher, and C. Miesbauer. "An expert survey on kinds, influence factors and documentation of design decisions in practice". In: *Future Generation Computer Systems* 47 (2015). Special Section: Advanced Architectures for the Future Generation of Software-Intensive Systems, pp. 145–160.

[286] F. Wilcoxon. “Individual Comparisons by Ranking Methods”. In: *Breakthroughs in Statistics: Methodology and Distribution.* Springer New York, 1992, pp. 196–202.

[287] R. J. Williams. “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine Learning* 8.3–4 (1992), pp. 229–256.

[288] J. P. Winkler, J. Grönberg, and A. Vogelsang. “Optimizing for Recall in Automatic Requirements Classification: An Empirical Study”. In: *2019 IEEE 27th International Requirements Engineering Conference (RE).* 2019, pp. 40–50.

[289] W. E. Winkler. “String comparator metrics and enhanced decision rules in the Fellegi-Sunter model of record linkage.” In: (1990).

[290] R. Wohlrab, U. Eliasson, P. Pelliccione, and R. Heldal. “Improving the Consistency and Usefulness of Architecture Descriptions: Guidelines for Architects”. In: *2019 IEEE ICSA.* 2019, pp. 151–160.

[291] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li. “Measuring Program Comprehension: A Large-Scale Field Study with Professionals”. In: *IEEE Transactions on Software Engineering* 44.10 (2018), pp. 951–976.

[292] L. Xu and J. D. Choi. “Revealing the Myth of Higher-Order Inference in Coreference Resolution”. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP).* Association for Computational Linguistics, 2020, pp. 8527–8533.

[293] I. Yamada, A. Asai, H. Shindo, H. Takeda, and Y. Matsumoto. “LUKE: Deep Contextualized Entity Representations with Entity-aware Self-attention”. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP).* Association for Computational Linguistics, 2020, pp. 6442–6454.

[294] L. Yang and A. Shami. “On hyperparameter optimization of machine learning algorithms: Theory and practice”. In: *Neurocomputing* 415 (2020), pp. 295–316.

[295] Z. Yang, Z. Dai, Y. Yang, J. G. Carbonell, R. Salakhutdinov, and Q. V. Le. “XLNet: Generalized Autoregressive Pretraining for Language Understanding”. In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019.* 2019, pp. 5754–5764.

[296] W. K. Youn, S. B. Hong, K. R. Oh, and O. S. Ahn. “Software certification of safety-critical avionic systems: DO-178C and its impacts”. In: *IEEE Aerospace and Electronic Systems Magazine* 30.4 (2015), pp. 4–13.

[297] D. H. Younger. “Recognition and parsing of context-free languages in time n3”. In: *Information and control* 10.2 (1967), pp. 189–208.

[298] C. Yu, J. Liu, S. Nemati, and G. Yin. “Reinforcement Learning in Healthcare: A Survey”. In: *ACM Computing Surveys* 55.1 (2021), pp. 1–36.

[299] G. Yu, Y. Yang, X. Wang, H. Zhen, G. He, Z. Li, Y. Zhao, Q. Shu, and L. Shu. “Adversarial active learning for the identification of medical concepts and annotation inconsistency”. In: *Journal of Biomedical Informatics* 108 (2020), p. 103481.

[300] M. Zhang, C. Tao, H. Guo, and Z. Huang. “Recovering Semantic Traceability between Requirements and Source Code Using Feature Representation Techniques”. In: *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS).* 2021, pp. 873–882.

[301] Y. Zhang, C. Wan, and B. Jin. “An empirical study on recovering requirement-to-code links”. In: *17th IEEE/ACIS SNPD.* 2016, pp. 121–126.

[302] T. Zhao, Q. Cao, and Q. Sun. “An Improved Approach to Traceability Recovery Based on Word Embeddings”. In: *2017 24th Asia-Pacific Software Engineering Conference (APSEC).* 2017, pp. 81–89.

[303] C. Zhong, H. Zhang, H. Huang, Z. Chen, C. Li, X. Liu, and S. Li. “DOMICO: Checking conformance between domain models and implementations”. In: *Software: Practice and Experience* 54.4 (2024), pp. 595–616.

[304] Y. Zhou, C. Wang, X. Yan, T. Chen, S. Panichella, and H. Gall. “Automatic Detection and Repair Recommendation of Directive Defects in Java API Documentation”. In: *IEEE Transactions on Software Engineering* 46.9 (2020), pp. 1004–1023.

[305] O. Zimmermann, T. Gschwind, J. Küster, F. Leymann, and N. Schuster. “Reusable Architectural Decision Models for Enterprise Application Development”. In: *Software Architectures, Components, and Applications.* Lecture Notes in Computer Science. Springer, 2007.

[306] X. Zou, R. Settimi, and J. Cleland-Huang. “Phrasing in Dynamic Requirements Trace Retrieva”. In: *30th Annual International Computer Software and Applications Conference (COMPSAC’06).* Vol. 1. 2006, pp. 265–272.

Appendix

A. Approaches

A.1. Lists For The Filters In ArDoCo's MME Detection

The general filter list for the *common words* contains the following words: cpu, gpu, file, directory, event, bus, browser, chrome, firefox, safari, edge, instance, object, module, code, java, javascript, nodejs, npm, kotlin, request, response, servlet, unit, test.

The following lists show the words that are on the corresponding project's domain-specific filter list.

- **BigBlueButton**: conversion, core, cpu, file, front, integration, nodejs, party, process, side, svg
- **JabRef**: aspect, bibdatases, bibentries, bus, event
- **MediaStore**: download, file, log, meta, server
- **TEAMMATES**: assertion, backdoor, check, classes, code, cron, end, failure, javascript, key, limit, minute, origin, processing, queue, request, servlet, task, testing, unit, utility, origin
- **TeaStore**: instance, item, name, product, rankings, rating, size

B. Evaluation

B.1. Results Of Baseline Approach For TLR Between SAD And SAM

Table B.1.: Detailed Results of STD for TLR between SAD and SAM Using the Metrics Precision (P), Recall (R), F_1-score (F_1), Accuracy (Acc), Specificity (Spec), And The Φ Coefficient (Φ).

Project	**P**	**R**	**F_1**	**Acc**	**Spec**	**Φ**
MediaStore	1.00	0.62	0.77	0.98	1.00	0.78
TeaStore	0.88	0.56	0.68	0.97	1.00	0.69
TEAMMATES	0.76	0.80	0.78	0.99	0.99	0.77
BigBlueButton	0.38	0.17	0.24	0.94	0.98	0.23
JabRef	0.87	0.72	0.79	0.91	0.97	0.74
Average	0.78	0.58	0.65	0.96	0.99	0.64
Weighted Avg.	0.75	0.53	0.61	0.96	0.99	0.60

B.2. Results Of Baseline Approach For TLR Between SAM And Code

B.3. Results Of Baseline Approaches For TLR Between SAD And Code

B.4. Results Of Baseline Approach For ID Of MMEs

Table B.2.: Results of the Baseline STD for TLR between SAMs And Code Using the Metrics Precision (P), Recall (R), F_1-score (F_1), Accuracy (Acc), Specificity (Spec), And The Φ Coefficient (Φ)

Project	**P**	**R**	**F_1**	**Acc**	**Spec**	**Φ**
MediaStore	0.32	0.50	0.39	0.99	0.99	0.40
TeaStore	0.55	0.30	0.39	1.00	1.00	0.40
TEAMMATES	0.01	0.27	0.02	0.97	0.97	0.05
BigBlueButton	0.07	0.09	0.08	1.00	1.00	0.07
JabRef	0.09	0.43	0.15	1.00	1.00	0.19
Average	0.21	0.32	0.20	0.99	0.99	0.22
Weighted Avg.	0.08	0.31	0.10	0.99	0.99	0.13

Table B.3.: Results Of TAROT For TLR Between SADs And Code Using the Metrics Precision (P), Recall (R), F_1-score (F_1), Accuracy (Acc), Specificity (Spec), And The Φ Coefficient (Φ).

Project	**P**	**R**	**F_1**	**Acc**	**Spec**	**Φ**
MediaStore	0.09	0.18	0.10	0.91	0.93	0.07
TeaStore	0.19	0.44	0.27	0.81	0.84	0.20
TEAMMATES	0.06	0.32	0.11	0.76	0.78	0.05
BigBlueButton	0.07	0.18	0.10	0.91	0.93	0.07
JabRef	0.32	1.00	0.48	0.33	0.02	0.05
Average	0.15	0.44	0.22	0.75	0.71	0.10
Weighted Avg.	0.19	0.63	0.29	0.58	0.44	0.06

Table B.4.: Results Of FTLR For TLR Between SADs And Code Using the Metrics Precision (P), Recall (R), F_1-score (F_1), Accuracy (Acc), Specificity (Spec), And The Φ Coefficient (Φ)

Project	**P**	**R**	**F_1**	**Acc**	**Spec**	**Φ**
MediaStore	0.15	0.26	0.19	0.97	0.98	0.18
TeaStore	0.19	0.25	0.21	0.85	0.90	0.13
TEAMMATES	0.06	0.30	0.10	0.76	0.78	0.04
BigBlueButton	0.04	0.42	0.07	0.68	0.69	0.04
JabRef	0.32	0.93	0.48	0.36	0.09	0.03
Average	0.15	0.43	0.21	0.72	0.69	0.08
Weighted Avg.	0.19	0.59	0.28	0.57	0.46	0.04

Table B.5.: Results Of CodeBERT For TLR Between SADs And Code Using the Metrics Precision (P), Recall (R), F_1-score (F_1), Accuracy (Acc), Specificity (Spec), And The Φ Coefficient (Φ)

Project	P	R	F_1	Acc	Spec	Φ
MediaStore	0.25	0.10	0.14	0.98	1.00	0.15
TeaStore	0.26	0.57	0.35	0.83	0.86	0.30
TEAMMATES	0.08	0.22	0.12	0.86	0.89	0.07
BigBlueButton	0.07	0.49	0.12	0.80	0.81	0.12
JabRef	0.49	0.83	0.61	0.67	0.59	0.39
Average	0.23	0.44	0.27	0.83	0.83	0.21
Weighted Avg.	0.28	0.53	0.36	0.76	0.74	0.23

Table B.6.: Results Of ArDoCode For TLR Between SADs And Code Using the Metrics Precision (P), Recall (R), F_1-score (F_1), Accuracy (Acc), Specificity (Spec), And The Φ Coefficient (Φ)

Project	P	R	F_1	Acc	Spec	Φ
MediaStore	0.05	0.66	0.09	0.82	0.82	0.15
TeaStore	0.20	0.74	0.31	0.74	0.74	0.28
TEAMMATES	0.37	0.92	0.53	0.93	0.93	0.56
BigBlueButton	0.07	0.57	0.13	0.78	0.79	0.14
JabRef	0.66	1.00	0.80	0.84	0.76	0.71
Average	0.27	0.78	0.37	0.82	0.81	0.37
Weighted Avg.	0.47	0.92	0.62	0.87	0.83	0.59

Table B.7.: Evaluation Results Of The Baseline For Detecting MMEs. The Metrics Precision (P), Recall (R), F_1-score (F_1), Accuracy (Acc), Specificity (Spec), And The Φ Coefficient Are Displayed.

Project	P	R	F_1	Acc	Spec	Φ
MediaStore	0.15	0.93	0.25	0.49	0.45	0.22
TeaStore	0.12	0.70	0.20	0.37	0.33	0.02
TEAMMATES	0.03	0.61	0.07	0.30	0.29	-0.05
BigBlueButton	0.08	0.65	0.14	0.45	0.44	0.04
JabRef	0.14	0.17	0.14	0.49	0.65	-0.22
Average	0.10	0.61	0.16	0.42	0.43	0.00
Weighted Avg.	0.09	0.64	0.15	0.41	0.40	0.01