Analysis of Project-Intrinsic Context for Automated Traceability Between Documentation and Code

Bachelor's Thesis of

Tobias Thirolf

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

First examiner:	Prof. Dr.-Ing. Eric Sax (ITIV)
Second examiner:	Prof. Dr.-Ing. Anne Koziolek (KASTEL)
First advisor:	Dominik Fuchß, M.Sc. (KASTEL)
Second advisor:	Dr.-Ing. Tobias Hey (KASTEL)

May 26, 2025 – December 24, 2025

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Analysis of Project-Intrinsic Context for Automated Traceability Between Documentation and Code (Bachelor's Thesis)

I declare that I have developed and written the enclosed thesis completely by myself. I have used generative AI as a tool for grammatical and stylistic revision of text passages. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. I have followed the by-laws to implement scientific integrity at KIT.

Ludwigshafen am Rhein, December 24, 2025

. . . . . .

(Tobias Thirolf)

Abstract

Traceability Link Recovery (TLR) between Software Architecture Documentation (SAD) and source code is essential for preserving architectural consistency and facilitating system comprehension. However, established automated approaches—ranging from Information Retrieval (IR) to modern Large Language Models (LLMs)—often fail to recover non-trivial links that depend on project-specific context, runtime configurations, or polyglot implementations. This thesis addresses this limitation by challenging the static, code-centric assumptions of existing benchmarks and retrieval strategies.

First, we conducted a rigorous manual analysis of four open-source projects (*JabRef*, *MediaStore*, *TEAMMATES*, and *TeaStore*). This qualitative assessment suggested that a significant portion of architectural links relies on implicit knowledge and non-Java artifacts often ignored by standard analysis. Based on these findings, we systematically refined the existing gold standards, transitioning from a keyword-based to an intent-based baseline. A comparison between this refined standard and the original benchmarks implies a substantial divergence in traceability definitions.

Second, to address these structural mismatches, we propose a novel Component-Centric Preprocessing approach integrated into the LiSSA framework. Unlike naive retrieval methods that blindly fetch code chunks, our approach utilizes LLM-based agents to first map documentation sentences to architectural components and then resolve those components to their physical directory locations using semantic reasoning and build configuration heuristics.

We evaluated this approach against state-of-the-art baselines (ArDoCo) and standard RAG configurations. The results demonstrate that our agentic, component-aware strategy achieves a weighted average F1-score of **.554** on the refined, diverse-artifact benchmark, significantly outperforming naive retrieval (.125) and model-based baselines (.382). These findings confirm that integrating architectural reasoning and handling project-intrinsic context are prerequisites for recovering the complex, configuration-driven links found in modern software systems.

Zusammenfassung

Die Wiederherstellung von Traceability-Links (Traceability Link Recovery, TLR) zwischen Softwarearchitekturdokumentation (SAD) und Quellcode ist essenziell für die Wahrung der architektonischen Konsistenz und das Systemverständnis. Etablierte automatisierte Ansätze – von Information Retrieval (IR) bis hin zu modernen Large Language Models (LLMs) – scheitern jedoch häufig daran, nicht-triviale Links wiederherzustellen, die von projektspezifischem Kontext, Laufzeitkonfigurationen oder polyglotten Implementierungen abhängen. Diese Arbeit adressiert diese Einschränkung, indem sie die statischen, code-zentrierten Annahmen bestehender Benchmarks und Retrieval-Strategien hinterfragt.

Zunächst führten wir eine rigorose manuelle Analyse von vier Open-Source-Projekten durch (*JabRef*, *MediaStore*, *TEAMMATES* und *TeaStore*). Diese qualitative Bewertung legte nahe, dass ein signifikanter Teil der architektonischen Links auf implizitem Wissen und Nicht-Java-Artefakten beruht, die von Standardanalysen oft ignoriert werden. Basierend auf diesen Erkenntnissen verfeinerten wir systematisch die existierenden Goldstandards, wobei wir von einer schlüsselwortbasierten zu einer absichtsbasierten (intent-based) Baseline übergingen. Ein Vergleich zwischen diesem verfeinerten Standard und den ursprünglichen Benchmarks impliziert eine erhebliche Divergenz in den Traceability-Definitionen.

Zweitens schlagen wir zur Adressierung dieser strukturellen Diskrepanzen einen neuartigen Ansatz der komponenten-zentrierten Vorverarbeitung (Component-Centric Preprocessing) vor, der in das LiSSA-Framework integriert ist. Im Gegensatz zu naiven Retrieval-Methoden, die blind Code-Fragmente abrufen, nutzt unser Ansatz LLM-basierte Agenten, um zunächst Sätze der Dokumentation auf Architekturkomponenten abzubilden und diese Komponenten anschließend mithilfe von semantischem Reasoning und Heuristiken der Build-Konfiguration auf ihre physischen Verzeichnisorte aufzulösen.

Wir evaluierten diesen Ansatz gegen State-of-the-Art-Baselines (ArDoCo) und Standard-RAG-Konfigurationen. Die Ergebnisse zeigen, dass unsere agentenbasierte, komponentenbewusste Strategie einen gewichteten durchschnittlichen F1-Score von **.554** auf dem verfeinerten Benchmark mit diversen Artefakten erzielt und damit das naive Retrieval (.125) sowie modellbasierte Baselines (.382) signifikant übertrifft. Diese Erkenntnisse bestätigen, dass die Integration von architektonischem Reasoning und die Berücksichtigung von projektinternem Kontext Voraussetzungen sind, um die komplexen, konfigurationsgetriebenen Links in modernen Softwaresystemen wiederherzustellen.

Contents

List of Figures

List of Tables

1. Introduction

Software architecture serves as the blueprint of a software system, defining its high-level structure, components, and the interactions between them. It is the primary vehicle for stakeholder communication, guiding development, maintenance, and evolution. However, a fundamental challenge in software engineering is the phenomenon of *architectural drift*: as the source code evolves to meet new requirements, the corresponding documentation frequently lags behind, eventually becoming obsolete or misleading.

To mitigate this disconnect, *Traceability Link Recovery* (TLR) aims to establish and maintain explicit links between architectural documentation (SAD) and the implementation artifacts (source code). Valid traceability links allow developers to verify compliance, perform impact analysis, and navigate complex systems with confidence.

1.1. Problem Statement

While manual traceability is accurate, it is prohibitively expensive and labor-intensive to maintain. Consequently, automated approaches have been a subject of extensive research. Historically, these approaches have relied on Information Retrieval (IR) techniques (matching keywords between text and code) or static analysis methods that reconstruct architecture from explicit code structures like Java packages.

However, modern software systems have evolved beyond simple package hierarchies. In contemporary architectures, a logical "component" is rarely just a single directory of Java files. Instead, it is often a polyglot assembly of source code, configuration files (XML, YAML, Gradle), and frontend artifacts (TypeScript, HTML). Our analysis reveals that state-of-the-art approaches struggle in this landscape because they lack *context*. They treat the codebase as a flat list of files, ignoring the hierarchical project structure and the implicit naming conventions that developers use to organize these files into components. Furthermore, existing benchmarks for this task often suffer from a "static bias," rewarding approaches that find Java classes while penalizing those that correctly identify the configuration files that actually define the system's runtime behavior.

1.2. Objective and Contribution

The primary objective of this thesis is to improve the accuracy and semantic validity of automated traceability between SADs and code by moving beyond simple similarity

matching. We hypothesize that "finding the code" is a two-step reasoning process: first, identifying *what* architectural component is described, and second, determining *where* that component physically resides in the project structure.

To achieve this, we introduce a **Component-Centric Preprocessing** approach integrated into the *LiSSA* (Linking Software System Artifacts) framework. This approach leverages Large Language Models (LLMs) not just as text matchers, but as autonomous agents capable of exploring the project's file system and interpreting build configurations.

The specific contributions of this work are:

1. **Empirical Analysis of Traceability Benchmarks:** We provide a detailed manual analysis of four open-source projects (*JabRef*, *MediaStore*, *TEAMMATES*, *TeaStore*), exposing significant limitations in existing gold standards. We identify that up to 31% of links in established benchmarks may be semantically "unreasonable" given the documentation's intent.
2. **Refinement of Gold Standards:** We contribute a refined set of gold standards that transition from a keyword-based to an intent-based definition of traceability, systematically including non-code artifacts and configuration files that were previously ignored.
3. **A Component-Centric Retrieval Approach:** We propose a novel preprocessing pipeline that resolves architectural components to physical directories before retrieval. This allows the system to dynamically adjust its search scope, filtering out irrelevant artifacts and focusing on the specific sub-tree of the project where the component resides.
4. **Agentic Project Navigation:** We demonstrate that treating the retrieval task as an agentic exploration problem (where an LLM can "browse" the file system) significantly outperforms static heuristics in polyglot environments.

1.3. Thesis Structure

The remainder of this thesis is structured as follows:

- **chapter 2 (Foundations):** Introduces the core concepts of software architecture, traceability link recovery, and the underlying technologies (LLMs, RAG) used in this work, as well as the framework that was extended.
- **chapter 3 (Related Work):** Introduces other works that are related to Traceability Link Recovery, including those acting as baselines.
- **chapter 4 (Analysis of Datasets):** Presents the manual audit of existing benchmarks, detailing the "structural mismatches" found and defining the guidelines used for refining the gold standards.

- **chapter 5 (Approach):** Details the proposed Component-Centric Preprocessing pipeline, explaining the prompt engineering, agentic workflow, and ambiguity resolution mechanisms.
- **chapter 6 (Evaluation):** Empirically evaluates the approach against state-of-the-art baselines (ArDoCo) and standard retrieval methods, analyzing the impact of reasoning capabilities and diverse artifact types on performance.
- **chapter 7 (Conclusion):** Summarizes the findings, discusses threats to validity, and outlines potential avenues for future research.

2. Foundations

2.1. Software Artifacts

In the context of software engineering, a software artifact is formally defined by the ISO/IEC 19506:2012 standard as "*a tangible, machine-readable document created during software development*" [22]. While the term can broadly encompass any intentional product of human activity [21], within this domain, it refers specifically to the diverse set of outputs created to support the software lifecycle, from implementation and testing to maintenance and comprehension [33].

These artifacts are generally categorized into two primary groups: *code artifacts* (e.g., source code, build scripts, configuration files) and *documentation artifacts* (e.g., requirements specifications, user manuals, architecture descriptions) [33, 34, 29]. Understanding the relationship between these two categories is central to the problem of traceability.

2.1.1. Software Architecture Documentation (SAD)

A critical subset of documentation is the **Software Architecture Documentation (SAD)**. The ISO/IEC/IEEE 42010 standard defines an architecture description as an artifact "*used to express an architecture*" [23].

Software architecture itself represents a high-level abstraction of a system, capturing its fundamental structural elements, such as components, connectors, and their relationships. This abstraction is pivotal for managing complexity, as it provides a condensed view that transcends individual classes or packages. By emphasizing the grouping of elements into cohesive units, architecture enables stakeholders to reason about the system's functionality, scalability, and evolution without getting lost in implementation details [12, 28].

To make this abstract information accessible, various documentation approaches exist, ranging from ontology-based formalizations to interactive visual tools [18, 37]. However, in industrial practice, unstructured text-based documentation remains the predominant format. Unlike formal requirements, which often follow strict templates (e.g., "The system shall..."), SADs typically consist of natural language prose. They frequently contain loosely organized sections, implicit references to components, and meta-level commentary. This lack of structural constraint presents significant challenges for automated analysis and information extraction [10].

2.1.2. Architecture Recovery

The reliance on manual documentation leads to a common problem: SADs often become missing, outdated, or incomplete as the code evolves. In such cases, the system's actual architecture must be reclaimed through **Software Architecture Recovery (SAR)**. This process involves analyzing the source code, specifically its structural characteristics, package hierarchies, and class interactions, to reconstruct the implemented architecture. Techniques in this domain aim to reverse-engineer models that mirror component boundaries and dependencies, thereby restoring the lost link between the code and its high-level design [28].

2.2. Traceability Link Recovery (TLR)

In the domain of modern software engineering, the lifecycle of a system involves the creation and evolution of a diverse set of artifacts, including functional requirements, architectural documentation, source code, and test specifications. A comprehensive understanding of the interdependencies between these artifacts is indispensable for critical activities such as impact analysis, consistency checking, and compliance verification.

Traceability links serve as the connective tissue in this ecosystem, establishing explicit relationships between heterogeneous artifacts. For instance, a trace link might map a high-level architectural constraint described in natural language to the specific software component that implements it. However, the manual creation and maintenance of these links are notoriously labor-intensive and error-prone. As systems evolve, manually maintained links frequently suffer from *traceability decay*, rendering them unreliable. To address this, **Traceability Link Recovery** provides automated mechanisms to identify and sustain these links, thereby significantly reducing the cognitive load on developers and ensuring the integrity of the development process.

Historically, the dominant paradigm for automated TLR has been **Information Retrieval**. These approaches rely on the fundamental premise of *textual coherence*: they assume that documentation is written in descriptive natural language and that developers utilize meaningful identifiers (e.g., class and variable names) in the source code. Consequently, artifacts that are semantically related are expected to share significant textual similarity. IR-based techniques typically vectorize the textual content of artifacts and calculate similarity scores (using metrics such as TF-IDF or Cosine Similarity) to rank candidate links, operating on the hypothesis that high textual overlap correlates with a valid trace link [35].

2.3. Metrics

Evaluation of the traceability performance happens through standard binary classification metrics. These are calculated based on the confusion matrix, which defines all four possible outcomes for a classification with respect to the comparing ground truth.

- **True Positive** (**TP**): A *positive* classification that *truly* is positive.
- **False Positive** (**FP**): A *positive* classification that *falsely* is positive (ground truth expects negative classification).
- **True Negative** (**TN**): A *negative* classification that *truly* is negative.
- **False Negative** (**FN**): A *negative* classification that *falsely* is negative (ground truth expects positive classification).

These results are accumulated over all classifications representing the fundamental set to evaluate the result. In the task that my work focuses on, the results of all linked code artifacts for each sentence is accumulated.

Depending on the classification task, only specific results might be relevant. As for different datasets the number of the classifications results naturally is not the same, a normalization is necessary. Furthermore, combinations of these outcomes provide information about the total number of *positive*/*negative* classifications as well as what the ground truth expected. The following metrics make use of this while additionally providing normalization.

2.3.1. Precision

The **precision** is a ratio defining how many of the *positive* classifications were actually *truly* positive. Therefore, when more *FPs* are present then the value goes closer towards 0. As example, a naive classification which always yields a *positive* result will naturally have a relatively small precision. On the other hand, a value close to 1 is achieved by having less *FPs*. The sum of *TPs* and *FPs* gives the total number of *positive* classifications that the approach produced, providing the wanted normalization.

$$Precision = \frac{TP}{TP + FP} \in [0, 1]$$

For traceability evaluation, this metric provides insight into the correctness of the linked artifacts. Linking too many artifacts increases the required effort to manually validate the results that the approach produced, which makes its usage impractical.

2.3.2. Recall

The **recall** is a ratio defining how many of the *expected positive* results were actually classified as *positive*. Therefore, having less *FNs* results in a value closer towards 1. While the precision of the naive classification before is small, its recall will be relatively high. The sum of *TPs* and *FNs* hereby gives the total number of results that the ground truth expected to be *positive*, again, providing the normalization.

$$Recall = \frac{TP}{TP + FN} \in [0, 1]$$

For traceability evaluation, this metric provides insight into the completeness of relevant artifacts to be linked. Intuitively, it answers the question how well the approach links at least those artifacts that are expected/relevant. Smaller values indicate that increasingly manual effort is necessary to find those relevant artifacts that the approach did not link.

2.3.3. F1-Score

The **F1-Score** combines both *precision* and *recall*. It is defined as the harmonic mean of these two metrics and hence normalized as well.

$$F1\ score = \frac{2 * Precision * Recall}{Precision + Recall} = \frac{2 * TP}{2 * TP + FP + FN} \in [0, 1]$$

It is notable here, that the *F1-score* alone does not represent a sufficient comparison because there are always multiple combinations of *precision* and *recall* resulting in the same value. Therefore, it is more like a single-number summary of the overall performance.

2.3.4. Sentence Normalized F1-Score

The normalizations before are based on the *total* number of (expected) classifications which do not take information about their subsets. The results of the traceability task here however, is accumulated over all sentences in the documentation. This makes it hardly indistinguishable whether the approach actually performed well on each sentence, or just on those, possibly few, that contribute a lot to the total number. To take the performance by sentence into account, the F1-score is additionally applied to each of them individually.

Unlike before, when only considering a single sentence, it is possible that there are no *positive* classifications produced by the approach and expected by the ground truth. This case describes an optimal classification because there are only *TNs* in the result. More intuitively: When the ground truth expects no links for this sentence ($TP = FN = 0$) then the classification is optimal if and only if the approach linked no artifacts ($TP = FP = 0$).

$$\text{F1-Score} = \begin{cases} 1, & \text{if } TP = FP = FN = 0 \\ \frac{2*TP}{2*TP+FP+FN}, & \text{otherwise} \end{cases} \in [0, 1]$$

These scores are then averaged, which corresponds to the Macro-averaged F1 Score, yielding the final sentence normalized (SN) score for the whole dataset.

$$\text{SN-F1-Score} = \frac{\sum_{i=1}^{n} \text{F1-Score}_i}{n} \qquad \text{where } n \text{ is the number of sentences}$$

$$\text{SN-Recall} = \frac{\sum_{i \in R} \text{Recall}_i}{|R|} \in [0, 1] \qquad \text{with } R = \{i \mid TP_i + FP_i \neq 0\}$$

2.4. The LiSSA Framework

The **Li**nking **S**oftware **S**ystem **A**rtifacts (**LiSSA**) framework, originally proposed by Fuchß et al. [17], serves as the technical foundation for the traceability approaches discussed in this work. Implemented in Java, LiSSA is designed as a versatile, task-agnostic framework for Traceability Link Recovery (TLR). Unlike specialized tools tailored solely for requirements-to-code or documentation-to-code tasks, LiSSA provides a generalized pipeline that leverages Large Language Models (LLMs) in conjunction with Retrieval-Augmented Generation (RAG) to recover links between arbitrary software artifacts.

2.4.1. Pipeline Architecture

The core of the framework is a multi-stage pipeline that transforms raw artifacts into verified trace links. An overview of this workflow is illustrated in Figure 2.1. The process consists of four distinct stages:

1. **Preprocessing:** The original software artifacts (e.g., requirement documents, source code files) are ingested and segmented into smaller granular units referred to as elements. The objective of this stage is to produce a discrete textual representation for each element suitable for embedding and retrieval. The framework supports various granularity strategies, such as splitting documentation by sentences or lines, and chunking source code at the method or arbitrary block level.
2. **Retrieval:** Once preprocessed, the system identifies relevant target candidates for each source element. This is achieved by computing vector embeddings for all elements and selecting the top-k source-target pairs based on cosine similarity. This stage acts as a filter to reduce the search space for the computationally expensive classification step.
3. **Classification:** The candidate pairs identified in the retrieval stage are analyzed by an LLM to determine the existence of a semantic link. The framework supports multiple classification modes to tailor the model's behavior:
    - *Simple Mode:* The model is queried for a direct binary decision (link/no-link).

- *Reasoning Mode:* The model is instructed to generate a textual justification for its decision before providing the final classification, encouraging chain-of-thought reasoning.
- *Mock Mode:* A pass-through mode that treats all retrieved candidates as valid links, primarily used to evaluate the isolated performance of the retrieval stage.

4. **Aggregation:** Finally, the verified links between individual elements are aggregated back to their parent artifacts to generate the final traceability matrix.

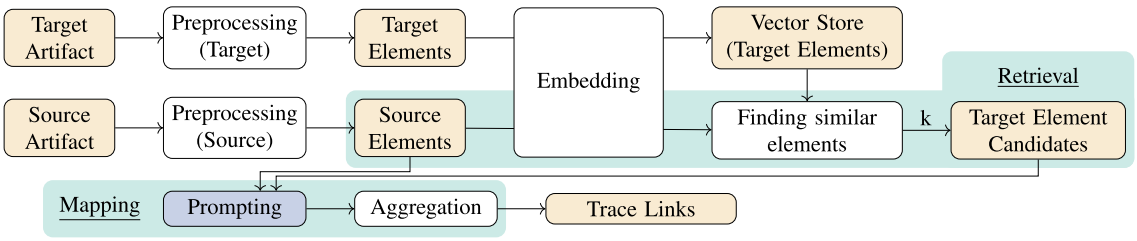

Figure 2.1.: Overview of the LiSSA approach pipeline [17]

2.4.2. Capabilities and Configuration

LiSSA is built to support a wide range of software artifact types, including natural language requirements, architecture documentation, source code, and XML-based UML models. These artifacts can be combined in arbitrary pairs to define specific traceability tasks.

The execution of the pipeline is declarative, controlled via configuration files that specify the input artifacts, the chosen preprocessing strategies, the embedding models for retrieval, and the specific LLM to be used for classification. This architecture allows for extensive experimentation with different model configurations and processing strategies without altering the underlying codebase.

3. Related Work

Existing research on automated traceability has predominantly targeted requirements–code and design–code traceability. However, software architecture documentation differs substantially from these artifact types in structure, abstraction level, and purpose. This thesis addresses this underexplored gap by analyzing how project-intrinsic context can support traceability between architecture documentation and source code.

3.1. Artifact Types in TLR

As Antoniol et al. [3] survey, IR-based approaches have been applied to a wide variety of artifacts. However, this extensive list highlights a significant gap: the near-total absence of Software Architecture Documentation (SAD) as a source artifact. Most foundational work, including [2], uses the term 'documentation' to refer to functional requirements. This is the primary gap my thesis addresses. SAD presents different challenges than requirements; it contains structural, design-pattern, and cross-cutting concern information that is not present in typical requirement lists.

While the majority of traceability research focuses on requirements, a few recent and highly relevant approaches have begun to tackle the specific challenges of Software Architecture Documentation (SAD). The TransArC approach Keim et al. [27] is a primary example, addressing the large semantic gap by using component-based architecture models as an intermediate step. It chains two separate tools, ArDoCo [26] and ArCoTL, to create transitive links. This method effectively bridges the gap, but its reliance on a multi-tool chain and explicit intermediate Software Architecture Models (SAM), which are not always present, makes it inapplicable. The work of Fuchß et al. [15] builds directly on this specific problem extending the approach. They investigated whether the use of modern LLMs can bridge this gap, reducing the need for architectural models to be present.

3.2. Techniques used in TLR

The foundation of automated traceability is dominated by classical Information Retrieval (IR) techniques, which are semi-automatic methods that have served as the baseline for the field for many years. This approach, which focuses primarily on Requirements Traceability Recovery (RTR) [35], treats artifacts as text and follows a standard pipeline: (1) parsing and pre-processing artifacts, (2) indexing the corpus into a term-by-document matrix, and (3)

computing similarity (e.g., Cosine Similarity) to rank candidate links [3]. This family of techniques includes the Vector Space Model (VSM), Latent Semantic Indexing (LSI), and Probabilistic IR models. While these methods were widely studied, key comparative research was essential in understanding their true impact. Studies have demonstrated that many methods, such as VSM and LSI, are "almost equivalent" in their overall performance [32]. The paper of Rodriguez and Carver [35] further detailed their comparative effectiveness, finding that VSM and IR Probabilistic models consistently achieve high precision but low recall. LSI, despite attempting to solve the synonym problem, often performed worse than VSM, highlighting the persistent challenge of relying on term-based similarity. The resulting performance profile (high precision and low recall) defines the critical weakness of classical IR: the inability to find a necessary balance between the two metrics [35, 3]. This trade-off is directly caused by the "vocabulary mismatch" problem [3], the semantic gap between high-level documentation and source code terminology. Consequently, these techniques remain semi-automatic, as developers must still manually review the resulting ranked lists to remove false positives and recover missing links [35].

To combat the vocabulary mismatch problem, numerous enhancement strategies have been proposed, as cataloged by Antoniol et al. [3]. These include lexical improvements like abbreviation expansion and splitting identifiers [20, 25, 7], using POS tagging [9], and more advanced methods like machine learning [19, 31, 11]. Recent reviews of the field, such as by Mohamed et al. [30], identify LLM-assisted traceability as a primary vector for current (2024-2025) research. These models are being explored for suggesting links and checking semantic consistency [17, 1]. While earlier ML approaches required specific feature engineering [19, 31], LLMs offer a powerful, end-to-end method for understanding deep semantic context, representing a significant evolution from the classical IR methods discussed [1].

3.3. Analysis of Datasets and Gold Standards

A persistent challenge in traceability, even with modern methods, is the presence of "noise links", i. e., spurious or irrelevant connections that obscure correct traces [8]. Past research has attempted to filter this noise using methods like decision trees or classifying code elements [8]. This highlights the critical importance of evaluating the quality of the underlying datasets and gold standards. A core contribution of my thesis is an in-depth analysis of the datasets used in [27, 15]. I analyze the prevalence and nature of this "noise" to establish a more reliable baseline before applying advanced techniques, a step often overlooked in prior work.

4. Analysis of Datasets and Refinement of Gold Standards

One of the primary contributions of this thesis is the systematic analysis and refinement of existing traceability benchmarks. The initial objective of this analysis was to investigate why state-of-the-art Trace Link Recovery (TLR) approaches fail to classify certain links correctly. Specifically, I sought to identify the project-intrinsic context, such as configuration files, naming conventions, and implicit architectural patterns, that is required to recover these links accurately.

This chapter presents the results of this analysis and the subsequent refinement of the gold standards for four open-source projects: *JabRef*, *MediaStore*, *TEAMMATES*, and *TeaStore*. Note that while the *BigBlueButton* dataset was initially considered, it was excluded from the final refined benchmark due to time constraints preventing a complete re-evaluation of its extensive documentation.

4.1. Dataset Overview and Provenance

This research builds upon established benchmarks that have been widely used in the scientific community for traceability tasks.

4.1.1. Origin of Resources

The datasets originate from the benchmark suite established by Fuchß et al. [16], which is publicly available on Zenodo [13]. These datasets were further extended by Keim et al. [27] specifically for the SAD-to-Code traceability task. For this thesis, the artifacts were sourced directly from the associated GitHub repository [14] (pre-release v1.1).

Each dataset consists of three primary components relevant to this work:

1. **Text Adaptations (SADs):** Plain text versions of the original Software Architecture Documentation. These were originally adapted from project repositories or websites to facilitate processing by Natural Language Processing (NLP) tools [16]. The quality and fidelity of these adaptations are critically analyzed in subsection 4.2.2.

2. **Code Models:** Specifications of the exact source code revisions used for the benchmark. Based on these models, the source code was cloned from the respective public repositories to serve as the target for trace link recovery.
3. **Gold Standards:** Ground truth files (provided as `.csv`) defining the correct trace links. Specifically, I utilize the `goldstandard_sad_to_code` files, which map sentences from the text adaptation to artifacts in the code model.

4.1.2. Subject Projects

The projects selected for this benchmark were originally chosen to ensure heterogeneity in domain, size, and architecture [16].

- **JabRef:** An open-source, cross-platform citation and reference management tool. It is a mature Java desktop application often used as a subject in software engineering research [24].
- **MediaStore:** A reference implementation of a web-based e-commerce store. Unlike the other subjects, it is not a community-driven open-source project but a demonstration artifact designed for educational and research purposes.
- **TEAMMATES:** A web-based peer evaluation and feedback system for students and teachers. It features a complex, distributed architecture involving Google App Engine, making it a challenging subject for traceability due to its heavy reliance on configuration and polyglot components (Java, TypeScript, HTML).
- **TeaStore:** A microservices-based reference application for benchmarking performance and resource management. It is designed to demonstrate cloud-native architectural patterns, where runtime behavior is often determined by deployment configurations rather than static code structure.
- **BigBlueButton:** An open-source web conferencing system supporting real-time sharing of audio, video, and screens [6]. (Excluded from refinement as noted above).

4.2. Analysis of SAD Characteristics and Existing Benchmarks

To justify the need for a refined gold standard, one must first analyze the linguistic nature of the source artifacts (the SADs) and compare this with the linking logic employed by existing benchmarks. This comparison reveals a structural mismatch between what the documents actually contain and how the existing gold standards attempt to link them.

4.2.1. Linguistic Characteristics of SADs

SADs are the source artifacts from which trace links are to be recovered. By definition, they represent the architecture of the software, implying a high-level view of the system. Ideally, architectural documentation focuses on high-level relationships, hierarchies, and interfaces.

However, an analysis of the original SADs (e.g., *TEAMMATES*, *TeaStore*) reveals a distinct *General-to-Specific* structure that contradicts the assumption of a purely high-level document. These documents typically define sections that first provide a brief overview of the components, followed by detailed descriptions.

- **General:** Usually, the first paragraph. Act as an introduction. They explicitly name the components and state their general purpose, often at a high level.
- **Specific:** Structurally corresponding to introduced components. These sentences elaborate on specific code artifacts, or logic on diverse levels, often down to method or statement level.

To quantify this observation, sentences were classified as *General* (general purpose) or *Specific* (describing specific parts/logic). Table 4.1 presents the results.

Table 4.1.: Number of Sentences in each SAD adaptation with their Section Distribution

	JabRef	**MediaStore**	**TEAMMATES 2021**	**TEAMMATES 2023**	**TeaStore**	**Average**
Total Sentences	13	37	198	151	43	
General	4 (31%)	10 (27%)	24 (12%)	30 (20%)	4 (9%)	**(20%)**
Specific	9 (69%)	27 (73%)	174 (88%)	121 (80%)	39 (91%)	**(80%)**

The data reveals that, on average, **80%** of the sentences in a SAD are *Specific*. This is unexpectedly high for documentation meant to represent a high-level view. Linguistically, this shift is crucial:

- *General* sentences typically reference the explicit **Named Entity**.
- *Specific* sentences, having often already established the context, might rely on it implicitly. For example, their context might be clear through being part of a section, or being a follow up sentence.

4.2.2. Analysis of Text Adaptations

While most SADs are maintained as structured Markdown files, existing baseline approaches rely on plain text adaptations. The primary motivation for this transformation is technical applicability: plain text is easier to split into sentences for Natural Language Processing (NLP) pipelines [16]. Consequently, the underlying assumption is that the text adaptation acts as a faithful proxy, providing the same information as the original artifact.

However, an analysis of the datasets reveals significant inconsistencies that contradict this assumption. These adaptations introduce noise, omit critical context, and in some cases, actively distort the semantic meaning of the architectural documentation.

4.2.2.1. Structural Inconsistency

The process of creating text adaptations was not applied uniformly across projects.

- **Headline Retention:** Headlines were preserved in *BigBlueButton* but removed in *TeaStore* and *TEAMMATES*.
- **Layout Loss:** All adaptations removed empty lines. In projects where headlines were also removed, the original sectioned structure becomes virtually unidentifiable. This flattening removes the scoping context that headers provide for the sentences beneath them.

4.2.2.2. Inconsistent Content Omission

Significant portions of the original documentation were omitted without clear reasoning. *JabRef*'s adaptation contains only an arbitrary subset of the original text. While some omissions were justified by claims that sections referenced external documentation, this rule was inconsistent; *BigBlueButton*'s adaptation, for example, retains sentences that serve solely as pointers (e.g., "*See 'Automatically apply configuration changes on restart'*"). Furthermore, visual information was handled erratically. The architectural overview diagram in *BigBlueButton* was ignored entirely, whereas in *TEAMMATES*, the diagram was replaced by a synthetically inserted sentence listing every contained component (information that did not exist textually in the original).

4.2.2.3. Semantic Distortion in TEAMMATES

The *TEAMMATES* adaptation exhibits the most severe deviations, where the adaptation process altered the actual meaning of the documentation.

First, distinct components such as *UI (Browser)* and *UI (Server)* were arbitrarily grouped into a single generic *UI* component, obscuring the client-server distinction. Second, the hierarchy of lists was flattened. Nested sub-steps (originally unordered lists inside ordered lists) were converted into top-level items, creating the false impression that they represent independent procedural steps rather than detailed elaborations.

Technical notation was simplified to the point of inaccuracy. A prime example is the regular expression `*Db`.

- **Original:** "*Represented by the '*Db' classes.*"
- **Adaptation:** "*Represented by the Db classes.*"

In the original context (the *Storage API* section), the asterisk implied a wildcard pattern matching classes ending in *Db*. The adaptation removes the wildcard, leading to ambiguity. This precise error has led to documented misinterpretations, where researchers incorrectly linked this sentence to unrelated *DataBundle* classes rather than the intended storage logic [36]. Since the section header "*Storage API*" was also removed, the context required to resolve this ambiguity is non-restorable.

Finally, the adaptation destroys hierarchical package information through arbitrary abstraction. In the original SAD, sub-packages are displayed as nested listings [5]. The text adaptation replaces the parent package structure with a generic placeholder, resulting in `x.util`, `x.e2e`, and `x.lnp` [4]. Combined with the previously mentioned flattening of the document structure, the adaptation fails to relate `x.lnp` as a sub-package of `e2e.cases`. This loss of hierarchy is critical, as it obscures the parent-child relationship between the architectural component and its sub-modules, a point that is central to the gold standard inconsistencies discussed in subsection 4.2.3.

4.2.2.4. Summary of Adaptation Analysis

The text adaptations do not serve as a faithful representation of the original SADs. By removing hierarchy, simplifying technical patterns, and inconsistently handling visual data, they present a distorted view of the architecture. This distortion actively facilitates misleading interpretations of the code, a problem that directly impacts the validity of the gold standards discussed in the following section.

4.2.3. The Structural Mismatch of Existing Gold Standards

The existing gold standards for SAD-to-code TLR were derived by merging SAD-to-SAM and SAM-to-code tasks [27]. Consequently, the resulting links are heavily influenced by the named components defined in the intermediate SAM. The implicit linking strategy governing these standards is **Named Entity-Based Linking**: A sentence is linked to all of a component's code if and only if the sentence *explicitly* mentions that component. When we apply this rule to the observed reality of SADs, two significant theoretical problems emerge.

[5] The WebUI provides the TeaStore front-end [...].
[6] It contains logic to save and retrieve values from cookies.

Figure 4.1.: SAD Sentences of the TeaStore Project; Named Entities in green, specific but unnamed Code in blue

First, consider a sentence that falls into the *Specific* category but still references the Named Entity. An example of such is shown in Figure 4.1. The Named Entity-based approach links sentence 6 to the *entire* WebUI component. In the case of *TeaStore*, the *WebUI* component consists of 77 distinct code artifacts. However, the sentence actually describes a logic located in exactly one of those files. The stakeholder is presented with 77 files for a sentence that

describes one. Since 80% of the document is specific, this approach forces the gold standard to systematically overestimate the scope of the majority of sentences, increasing the manual effort required to make use of the suggested trace links.

Second, consider a sentence that falls into the *Specific* category but solely relies on implicit context. Figure 4.2 illustrates this scenario. Sentences 12, 13, and 15 each describe detailed cases of a specific logic. Whereas sentence 12 repeats the component name, 13 and 15 do not; they are allowed to drop the name due to the context being set beforehand. A Named Entity-based gold standard, looking for explicit references, will fail to link sentences 13 and 15 entirely.

[11] The Image Provider matches the provided product ID or UI name and the image size to a unique image identifier.
[12] - If the product ID or UI name is not available to the Image Provider, a standard "not found" image will be delivered in the correct size.
[13] - If the product ID or UI name is found but not in the requested size, the largest image will be loaded and scaled.
[15] - If the product ID or UI name and size is found, the image will be loaded and delivered.

```
// Try to retrieve image from disk or from cache
long imgID = db.getImageID(key, size);
if (imgID != 0) { storedImg = storage.loadData(imgID); }
// If we dont have the image in the right size, get the biggest one and scale it
if (storedImg == null) {
 storedImg = storage.loadData(db.getImageID(key, stdSize));
 if (storedImg != null) {
  storedImg = scaleAndRegisterImg(storedImg.getImage(), key, size);
 } else {
  storedImg = storage.loadData(db.getImageID(IMAGE_NOT_FOUND, size));
  if (storedImg == null) { storedImg = scaleAndRegisterImg(storage.loadData(
                db.getImageID(IMAGE_NOT_FOUND, stdSize)).getImage(),
                new ImageDBKey(IMAGE_NOT_FOUND), size); }
 }
}
return storedImg.toString();
```

Figure 4.2.: Traceability Example for the TeaStore Project; SAD on the left with their [sentence ID], Code on the right, colored correspondingly

This leads to a problem: The more a SAD provides specific information about the code, the more likely a Named-Entity approach is to either overlook it (missing link) or drown it in irrelevant files (coarse-grained link).

4.2.4. Conclusion: Justification for Refinement

The classification results demonstrate that SADs are not merely lists of components, but hierarchies of description where the vast majority of the content (80%) is dedicated to specificities. However, the existing gold standards treat the SAD as a flat list of Named Entities. The more sentences a SAD contains that describe specific parts, which, as demonstrated, represents the dominant case, the more problematic Named-Entity-based linking becomes. It fails to capture the *intention* of the specific sentences. Therefore, it is reasonable to conclude that the existing gold standards do not accurately represent the task of tracing the *content* of the SAD to the code. They merely track the *mentions* of Named-Entities. This structural expectation of failure justifies the need for a refined gold standard: one that abandons the reliance on explicit names and instead links based on the specific semantic objective of the sentence.

4.3. Refinement of the Gold Standard

Based on the structural mismatch identified in the previous section, I performed a systematic refinement of the existing gold standards. The objective was to transition from a *Named Entity* baseline, which links based on explicit keyword matches, to an *Intent-Based* baseline that links based on the semantic reality of what the author intended to describe.

4.3.1. Methodology and Guidelines

The refinement was conducted through a manual re-evaluation of every trace link in the dataset. Since the original documentation did not establish strict traceability rules beforehand, I adopted a set of guidelines to govern the decision-making process. The overarching principle was to identify the code artifacts that the author of the SAD *intended* to point to, even when the terminology was ambiguous or subjective.

4.3.1.1. Guideline 1: Semantic Intent vs. Explicit Naming

The primary guideline was to assess the information presented in the sentence and resolve it against the code that corresponds to that description. I assumed that if a sentence describes a specific behavior or feature, code artifacts corresponding to that description must exist.

- **Granularity:** If a sentence describes a specific logic (e.g., "*It contains logic to save and retrieve values from cookies.*"), I linked only the specific classes implementing that logic, removing links to the generic component root or unrelated utility files.
- **Implicit Context:** I resolved implicit references where the subject of a sentence was implied by the preceding context (e.g., a header or previous sentence).
- **Scope:** The analysis was not restricted to Java code. I systematically included configuration files, build scripts, and frontend artifacts (HTML, TypeScript) if they were the true location of the described intent.

4.3.1.2. Guideline 2: Handling Inconsistencies and Drift

A major challenge was dealing with the inconsistencies between the documentation and the evolving codebase, particularly in *TEAMMATES*. To resolve this, I adopted the guideline of considering the SAD as if it corresponded to the current state of the code:

- **Semantic Validity:** If a sentence described a valid feature that still exists (even if implemented differently), I linked it to the current implementation.

- **Dead References:** Conversely, if the documentation explicitly named a package or artifact that no longer exists (e.g., the moved `client.remoteapi` package in *TEAMMATES*), I did *not* attempt to link it to its new location. Linking to an artifact that contradicts the explicit text was deemed an invalid interpretation of the document's pointers.
- **Text Adaptation Correction:** For *TEAMMATES*, I utilized a revised text adaptation that aligns more closely with the code's reality, allowing me to maintain the contextual information of sentences that were previously distorted or lost in the baseline adaptation.

4.3.2. Correction of Structural Benchmarking Inconsistencies

Beyond refining individual sentences, the analysis revealed systematic differences stemming from the underlying *SAD-to-SAM* (Software Architecture Model) approach used in the original benchmarks. The original gold standards typically linked code to SAM components and then mapped those components to SAD sections. Our analysis identified inconsistencies in this mapping process:

- **Component Splitting (MediaStore):** The *MediaStore* SAD describes a *Web* presentation layer. The code implements this as two projects: a presentation layer (`mediastore.web`) and a delegation layer (`mediastore.ejb.facade`). The original gold standard linked only to the Facade, effectively excluding the actual web artifacts (XHTML, Beans) described in the text. Our refinement corrected this to include both.
- **Component Merging (TeaStore):** Conversely, the *TeaStore* SAD describes a single *Recommender* component. The original benchmark treated the implementation as two distinct components, often failing to link sentences that described the recommender as a whole to all its constituent parts.
- **Inconsistent Artifact Inclusion:** I observed instances where a specific sentence was linked to code artifacts (likely due to a Named Entity match) that were paradoxically *excluded* when the component as a whole was linked elsewhere. This suggests an inconsistency in how component boundaries were defined and enforced in the original standard versus how other named entities were treated.

4.3.3. Quantitative Results

The refinement produced a fundamental shift in the composition of the gold standards, as detailed in Table 4.2.

The data reveals a distinct difference in the density of Java links. In *JabRef*, the number of Java links decreased from 8268 to 3851, and in *TeaStore*, from 701 to 314. This indicates a difference in granularity: the refined standard tends to link specific classes rather than entire packages or components. Conversely, *TEAMMATES* shows an increase in total links

Table 4.2.: Refined Gold Standard Difference showing Number of Linked Artifacts by their Type

		JabRef	MediaStore	TEAMMATES	TeaStore
Old	**Java**	8268	50	8165	701
	Shell	—	—	—	6
	Sum	8268	50	8165	707
		JabRef	**MediaStore**	**TEAMMATES**	**TeaStore**
New	**Java**	3851	64	5968	314
	Other	—	**.xhtml** 9	**.yaml** 7 **.spec.ts** 3300 **.ts** 4990	**.sh** 2 **.jsp** 52 **.png** 22 **.xml** 30
	Sum	3851	73	14265	420

due to the systematic inclusion of non-Java artifacts (TypeScript, HTML, XML), which were previously outside the scope of the benchmark.

To characterize the divergence between the two standards, Table 4.3 classifies the links based on the refined guidelines. On average, **31%** of sentences in the original benchmarks contained links that were excluded in the refined dataset (classified as *Noise* under the new guidelines), while **48%** of sentences lacked links that were identified as necessary context during refinement (*Missing*).

Table 4.3.: Number of Sentences in each Dataset containing at least one Noise or Missing Link

# Sentences	**JabRef**	**MediaStore**	**TEAMMATES**	**TeaStore**	**Average**
Total **Containing Links**	13 10 (77%)	37 25 (68%)	198 71 (36%)	43 23 (53%)	 (58%)
Noise (of Total) **(of Containing Links)**	7 (54%) (70%)	2 (5%) (8%)	48 (24%) (68%)	18 (42%) (78%)	**(31%)** **(56%)**
Missing (of Total)	6 (46%)	17 (46%)	69 (35%)	28 (65%)	**(48%)**

This divergence is visualized in Figure 4.3. The plot contrasts the links present only in the old standard (*Noise* / More) against the links present only in the new standard (*Missing*). The comparison highlights that the two standards prioritize different aspects of traceability: the original standard favors broad component-level linking, while the refined standard prioritizes specific, semantic implementation artifacts. Plots for the other projects, as well as the comparisons of the total number of linked code artifacts by sentences, can be found in Appendix A.

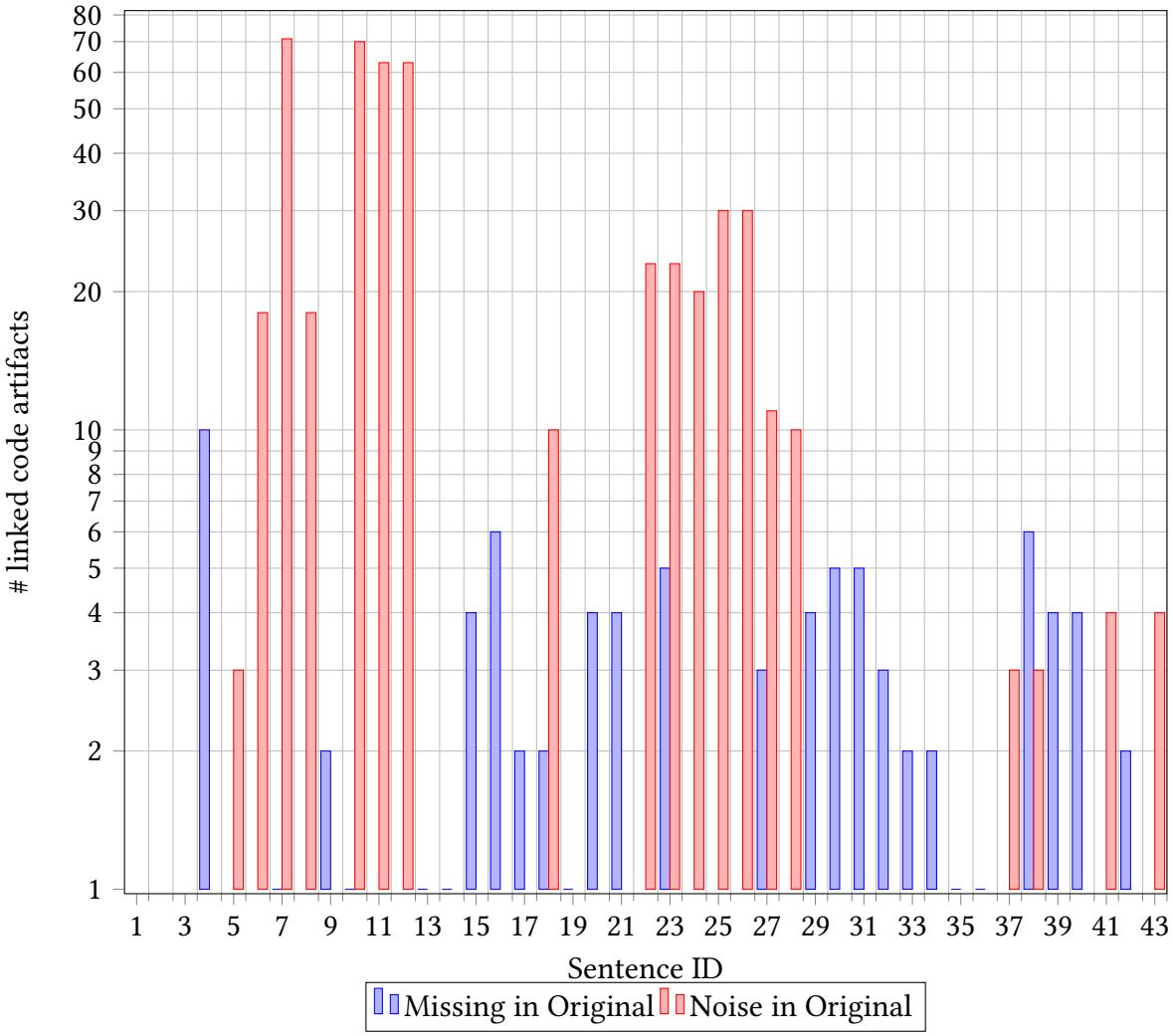

Figure 4.3.: TeaStore: Difference between the Original and Refined Gold Standard

4.4. Qualitative Findings: The Role of Context in Traceability

The sentence-by-sentence analysis conducted to refine the gold standard yielded more than just a corrected dataset. It revealed that accurate SAD-to-code traceability relies heavily on context that lies outside the static Java source code. While the previous section quantified the *noise* in existing benchmarks, this section categorizes the *project-intrinsic knowledge* required to recover valid links.

The following findings highlight specific patterns where trace links are non-trivial, meaning they cannot be recovered by matching keywords or traversing the Java AST alone. It is important to note that the cases presented here represent only a fraction of the identified patterns. The complete set of analysis notes for every sentence in the corpus is available in the replication package attached to this thesis.

4.4.1. Configuration as Architectural Truth

A critical finding of the analysis is that the *truth* of an architectural description is often established in configuration files rather than Java classes. In three of the four analyzed projects, the SAD describes behaviors or constraints that are technically fulfilled only when specific XML, JSON, or Gradle files are correctly set up. A code-only trace fails to capture these links because the source code often provides only the *potential* for a behavior, while the configuration provides the *fact*.

4.4.1.1. Runtime Implementation Selection

Modern architectures frequently use Dependency Injection (DI) or deployment parameters to select specific algorithms at runtime. The SAD often describes the result of this selection as a hard fact.

- **TeaStore (Recommender Strategy):** The documentation states: "*If the user is known, Slope One [...] is applied.*" Static analysis of the Java code shows multiple available algorithms (e.g., `SlopeOneRecommender`, `PopularityBasedRecommender`). The assertion that Slope One *is applied* is valid only because of the deployment chain: `baseContext.xml` wires the beans, and `start.sh` (invoked by `Dockerfile`) sets the environment variables that force this specific selection.
- **MediaStore (Feature Toggling):** Features like *watermarking* and *re-encoding* are described as mandatory steps in the processing pipeline. In the code, these are implemented as EJB interceptors (`TagWatermarkingImpl`, `ReEncoderImpl`). Their execution is not hard-coded in the business logic but orchestrated entirely by the `ejbconfig.xml` descriptor. Without linking this configuration, the architectural promise that "*all downloaded files are watermarked*" cannot be verified.

4.4.1.2. Implicit Control Flow and Security Boundaries

The documentation often describes a flow of control or a security boundary that is invisible in the Java call graph because it is managed by the container or build tools.

- **TEAMMATES (Request Filtering):** The SAD describes a specific security chain: "*custom filters are applied according to the order specified in web.xml.*" The application of filters like `OriginCheckFilter` is not triggered by Java method calls but by the servlet container reading `web.xml`.
- **TEAMMATES (Static Assets):** The statement "*Requests for static asset files [...] are served directly without going through web.xml*" describes a negative architectural constraint (bypassing the standard servlet flow). This behavior is implemented by the Angular service worker configuration in `ngsw-config.json`. A code-only trace misses the artifact that actually enables this *direct serving* capability.
- **TEAMMATES (API Boundaries):** The SAD mentions a "*Web API*" protected by access control. In the source code, the boundary between the UI Browser (Angular) and UI Server (Java) is bridged by build tasks. Specifically, `build.gradle` contains the task execution logic that generates the TypeScript clients from the Java definitions. Understanding the *Web API* as a distinct architectural entity requires tracing this build configuration, which acts as the glue between the two stacks.

4.4.1.3. Definition of Architectural Scopes

Finally, high-level terms used in the SAD often map to sets of artifacts defined by testing or build configurations rather than package boundaries.

- **TEAMMATES (Test Suites):** The term "*Component tests*" is semantically ambiguous in the folder structure, where unit and integration tests coexist. Its precise definition, which tests are pure unit tests versus integration tests, is explicitly defined in the test suite configuration `testng-component.xml` and the custom runner `TestNgXmlTest.java`. Recovering the link to *Component tests* requires parsing this configuration to identify the included classes.

4.4.2. Project-Specific Intrinsic Patterns

Beyond configuration, accurate traceability requires navigating the unique structural and terminological realities that characterize each codebase. Our analysis reveals that standard heuristics, such as mapping components to directories or assuming consistent naming, often fail due to project-specific patterns.

4.4.2.1. Terminological Ambiguity and Naming Collisions

A major source of confusion is the collision between general technical terms and specific component names, as well as ambiguous naming conventions.

- **TEAMMATES (Suffix Overloading):** The project enforces a naming convention where business logic classes end with the suffix `*Logic` (e.g., `AccountsLogic`). However, the project also contains a distinct component explicitly named *Logic*. This creates a *linguistic trap* for automated tools: references to *Logic classes* in the SAD are ambiguous, they could refer to the component boundary or the class suffix.
- **TeaStore (Overloaded Terms):** The term *PersistenceProvider* is used in the SAD to refer to the *Persistence* component (*tools.descartes.teastore.persistence*). However, in the code context, *Persistence Provider* is also the technical term for the JPA implementation (EclipseLink). A keyword-based approach fails to distinguish between the architectural role and the library dependency.

4.4.2.2. Structural Divergence and Scattering

The assumption that architectural components map one-to-one to code directories is frequently violated by project-specific organizational patterns.

- **JabRef (Scattered Components):** Unlike most components in the project, which are directory-disjunct, the *Preferences* component is structurally scattered. Parts of it reside in the Logic layer, while others reside in the GUI. They are identified solely by the `*Preferences` naming suffix rather than a unified package, requiring a cross-cutting search strategy.
- **MediaStore (Component Splitting):** The SAD describes a single architectural entity (the *Web* component). The implementation, however, splits this into two physically distinct projects to separate concerns: `mediastore.web` (Presentation) and `mediastore.ejb.facade` (Delegation). Recovering links for *Web* requires mapping a single documentation concept to multiple, separated code roots.
- **TEAMMATES (Non-Disjunct Test Scopes):** The strict separation of components breaks down in the testing layer. *Test components* explicitly contain and compile code from other production components to facilitate integration testing. This implies that components in this project are not mathematically disjunct sets; while production code is separated, test code intentionally overlaps.

4.4.2.3. Domain Concept Fragmentation

Ideally, a domain concept described in the SAD maps to a single entity in the code. In reality, projects often fragment these concepts into multiple, context-specific classes.

- **MediaStore (The *Audio* Concept):** The simple concept of an *audio file* is implemented via three distinct classes, each serving a specific architectural scope:
  - `Audio`: The persistable JPA entity, visible only to the *DbManager*.
  - `AudioFileInfo`: A lightweight metadata wrapper used by unrelated components to avoid coupling to the persistence layer (explicitly documented in Javadoc).
  - `AudioFile`: A specific junction class combining metadata with physical storage paths.

  Correctly linking a general sentence about *audio files* requires deducing which of these three specific implementations is meant based on the surrounding architectural context (e.g., *storage* vs. *metadata*).

4.4.2.4. Intent vs. Implementation Reality

Finally, the *architectural truth* described in the SAD can be contradicted by the *implementation reality* dictated by the project's specific purpose.

- **TeaStore (Test-Oriented Design):** The SAD describes the *Auth* component as responsible for secure user management (e.g., *Passwords are hashed*). However, because TeaStore is designed as a *performance test application*, the actual user generation is hardcoded and executed at startup within the *Persistence* component to ensure reproducible test states. A naive trace would look for user creation logic in *Auth*, whereas the *true* link, driven by the project's nature as a testbed, lies in the *Persistence* data generators.

4.4.3. Implicit Communication and Constraints

Finally, the analysis reveals that high-level interactions and architectural constraints described in the SAD are frequently invisible to standard static analysis. In these cases, the *link* is not a direct dependency but a pattern or a validation rule.

4.4.3.1. Decoupled Communication Patterns

When components interact via decoupled mechanisms rather than direct method calls, standard Abstract Syntax Tree (AST) analysis fails to capture the connection.

- **TeaStore (REST vs. AST):** The high-level communication between TeaStore components occurs almost exclusively via REST endpoints. A static analysis of the Java code sees these components as isolated islands, as there are no direct method invocations between them. Recovering these links requires recognizing a specific pattern: a URL string in a client (e.g., in *registryclient*) matching a `@Path` annotation in a server component. Approaches relying on standard call graphs will systematically miss these

interactions, even though they constitute the primary architectural glue described in the SAD.

4.4.3.2. Delegation and Encapsulation

SADs often describe components as monolithic entities that *perform* actions, whereas the code implements them as thin wrappers delegating to internal helpers.

- **MediaStore (The *Manager* Rule):** The SAD claims the `UserDBAdapter` component *encapsulates database access* and *creates queries.* In the code, the main entry point (*UserDBAdapterImpl*) is effectively an empty shell that delegates all logic to an injected `DbManager` bean. The architectural behavior (query creation) is located in the delegate, not the primary implementation class. A keyword-based trace to the component root often misses the `DbManager`, thereby missing the artifact that actually fulfills the architectural description.

4.4.3.3. Constraint Enforcement via Tests

Architectural rules described in the SAD (e.g., *Layer A cannot access Layer B* or *Package X is hidden*) are often enforced by the build system rather than the Java compiler.

- **JabRef & TEAMMATES (Test-Based Constraints):** Statements about visibility (e.g., "*Classes in `storage.entity` are not visible outside*") are often effectively *false* in the static source code, where the classes may be declared `public` for technical reasons. However, they are *true* in the project context because specific architecture-related test cases (e.g., `MainArchitectureTest.java`) fail the build if these rules are violated. In these instances, the trace link for the *enforcement* of the constraint points to the test case, which serves as the executable specification of the architecture.

4.4.4. Intent vs. Implementation Reality

A unique category of context involves the discrepancy between the *Architectural Intent* described in the SAD and the *Implementation Reality* dictated by the project's specific purpose or history. In these cases, the code structure contradicts the documentation, not because of drift, but because of specific non-functional requirements like testability.

4.4.4.1. Test-Oriented Design vs. Production Architecture

Projects designed as research testbeds often implement architectural shortcuts that contradict their high-level design description.

- **TeaStore (Hardcoded Logic):** The SAD describes the *Auth* component as responsible for secure user management (e.g., *Passwords are hashed*). However, TeaStore is designed specifically as a *performance test application* for benchmarking. To ensure reproducible test states, the actual user generation is hardcoded and executed at startup within the *Persistence* component. A naive trace would look for user creation logic in the *Auth* component, whereas the *true* link, driven by the project's nature as a deterministic testbed, lies in the *Persistence* data generators (*DataGenerator.java*).

4.4.4.2. Evolutionary Vestiges

Codebases often contain *vestigial* structures, artifacts that remain from previous architectural versions but are now used differently than their names suggest.

- **MediaStore (The Facade Drift):** The *Book* version of the architecture describes a rich *WebGUI* component that handles presentation logic. The actual code, however, splits this into a *Web* project and a *Facade* EJB. The *Façade* component in the code has evolved into a thin wrapper that strictly delegates calls, stripping it of the presentation logic described in the original architectural intent. Recovering the correct link requires understanding this evolutionary split: the *WebGUI* concept in the SAD now maps primarily to the *mediastore.web* project, not the *facade* package.

5. Approach

Traditional Information Retrieval (IR) approaches for traceability link recovery typically retrieve a fixed number of code artifacts (k) for each documentation element. However, our analysis of the datasets reveals a fundamental structural mismatch: architectural components vary significantly in size. If a component consists of hundreds of files (e.g., the *Logic* component in TEAMMATES), retrieving a small k misses relevant artifacts (low recall). Conversely, if a component is small, a large k introduces noise (low precision).

To address this, this thesis proposes a *Component-Centric Preprocessing* stage. Instead of linking text directly to code, we use architectural components as intermediate entities. The core idea is to first identify which component a sentence describes, and then retrieve the specific code artifacts that constitute that component.

The preprocessing stage consists of two parallel pipelines: *Documentation Preprocessing* and *Code Preprocessing*. As illustrated in Figure 5.1, both pipelines aim to map their respective artifacts to a unified set of Component Names. The Documentation pipeline extracts component references from the text, while the Code pipeline identifies the physical location (directories) of components in the project structure. The intersection of these two processes allows us to link a sentence to a specific directory scope, thereby filtering the search space for the subsequent retrieval stage.

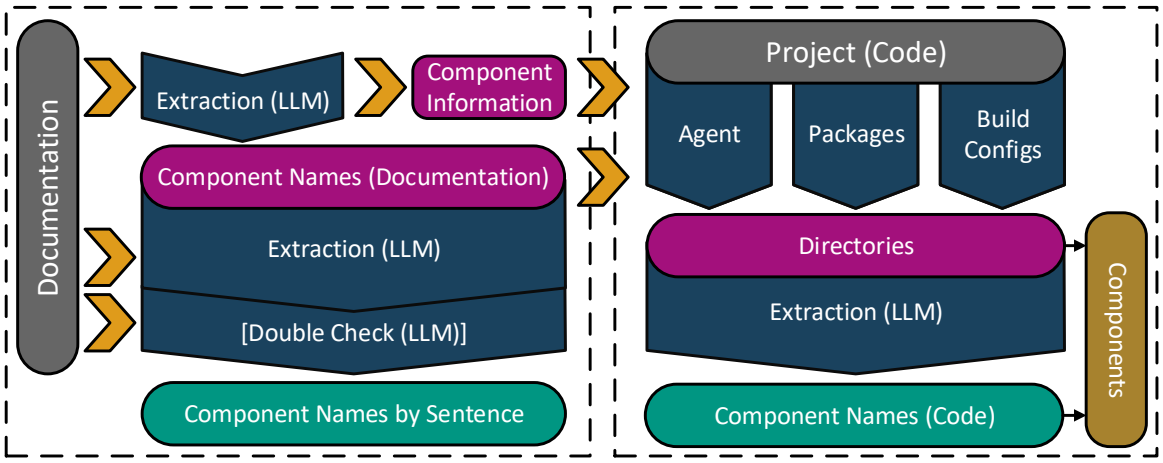

Figure 5.1.: Overview of the Preprocessing Stage; *Documentation Preprocessing* on the left, *Code Preprocessing* on the right

5.1. Documentation Preprocessing

The goal of documentation preprocessing is to transform unstructured natural language sentences into structured metadata identifying the referenced architectural components.

5.1.1. Component Names Extraction

The component name serves as the primary key for alignment. Besides distinguishing one component from another, the name itself often embodies technical information (e.g., *Persistence Provider*) that assists in locating the corresponding code.

To ensure the extracted names are usable for code mapping, the extraction process is governed by strict constraints enforced via the prompt (see Prompt 2):

- **Extraction of Simple Names:** The Large Language Model (LLM) is instructed to extract names exactly as they appear, without interpretation. It must prefer simple, explicit names (e.g., *WebUI*) over descriptive phrases. This mimics the naming conventions found in codebases (e.g., package names) and ensures consistency when a component is referenced by variable suffixes.
- **Project Name Exclusion:** Documentation often references the project itself (e.g., “TeaStore consists of...”). If the project name is not explicitly distinguished from its components, it may be falsely identified as a component. To prevent this, the system first retrieves the project name via an independent query (Prompt 1) and explicitly instructs the extraction model to exclude this name from the results.
- **Affix Removal:** Documentation frequently appends generic descriptors to component names (e.g., "Web *Service*", "Logic *Component*"). While semantically useful, these suffixes often do not exist in the file system. The extraction prompt instructs the LLM to strip these non-essential prefixes or suffixes if they obscure the core component name.
- **Filtering Main Components:** Due to the ambiguity of the term *component,* the LLM is restricted to identifying only the *main architectural components* of the system, filtering out transient objects or minor utility classes that do not represent architectural boundaries.

The output is forced into a structured JSON schema to ensuring parsability. These extracted results represent the ground truth for component identification and serve as the input for the subsequent component information extraction.

5.1.2. Sentence-Level Component Mapping

Following the extraction of high-level component metadata, the next step is the granular mapping of individual sentences to these components. This step bridges the gap between the document-level understanding and the specific code retrieval required later.

The objective is to produce a definitive assignment of component names for each sentence in the SAD. This output serves as the primary input for the final *Double Check* phase, where ambiguities are resolved before code retrieval begins.

5.1.2.1. Extraction Strategy

We employ an LLM to analyze the entire SAD sentence by sentence. To ensure traceability, the documentation is provided with explicit sentence identifiers. The model is instructed to function as an architectural analyst using the following system message:

> "Your task is to analyse software architecture documentation. Each sentence of the documentation is prefixed with its identifier. Extract for each sentence in which components of the project one needs to look to find the code that it describes."

By framing the task as *finding the code,* the prompt encourages the model to look beyond generic mentions and focus on the architectural intent of the specific sentence.

5.1.2.2. Result of Extraction

The critical outcome of this process is, for each sentence, a list of names of the components described in the sentence.

Crucially, the model is constrained to select these names from the *standardized list* of components generated in the initial extraction stage. This ensures strictly typed results and prevents the generation of hallucinations or synonyms that do not match the identified architecture. In cases where a sentence does not describe any specific architectural component (e.g., general project goals or meta-text), the model is instructed to map it to a reserved **dummy identifier**. This distinction allows the subsequent retrieval stage to filter out irrelevant sentences entirely, maintaining high precision.

5.1.3. Ambiguity Resolution (Double Check)

A significant challenge in text analysis is terminological ambiguity. Two components may share similar names (e.g., *Web* vs. *WebUI* in *TeaStore*), functionally overlapping responsibilities, or inconsistent usage in the documentation. A single-pass extraction often fails to distinguish these subtle differences, leading to lower precision.

To mitigate this, we employ a targeted *Double Check* mechanism. Unlike the previous steps which operate linearly, this stage is designed as a *Reflective Verification Loop*. It does not merely extract; it critiques the previous extraction using a generated knowledge base of ambiguities.

5.1.3.1. Phase 1: Global Ambiguity Detection

Before verifying individual sentences, the system first analyzes the project's component list holistically to identify potential sources of confusion.

We provide an LLM with the complete list of extracted component names and the full documentation. The model is instructed to identify pairs or groups of components that share ambiguities arising from:

- **Nominal Similarity:** Names that are linguistically close (e.g., *Auth* vs. *Authorization*).
- **Structural Overlap:** Components that are part of a named structure that does not distinguish between them (e.g., distinct backend and frontend components both referred to as *the UI*).
- **Functional Overlap:** Components that are responsible for similar tasks or contain similar software artifacts.

Result: The output is a knowledge base of *Ambiguity Cases*. Each case contains the set of confusing components and, crucially, explicit instructions on how to resolve the ambiguity based on the text (e.g., "When the text mentions 'UI', it refers to the 'WebUI' component only if specific frontend technologies like HTML are mentioned; otherwise, check for 'UI Server'.").

5.1.3.2. Phase 2: Contextual Verification

The second phase applies this knowledge to the results of the Sentence-Level Mapping. For every component assigned to a sentence in the previous step, the system determines if it belongs to a known Ambiguity Case.

If a component is flagged as ambiguous, it undergoes a rigorous verification process. An LLM is presented with:

1. The specific sentence in question.
2. The candidate component extracted in the previous step.
3. The list of *other* components extracted for the same sentence (potential competitors).
4. The Additional Information from the Ambiguity Case (the resolution rules generated in Phase 1).

The model acts as a judge, asked to decide whether the extracted component is truly expected to contain the code described in the sentence. It must justify its reasoning using the provided ambiguity resolution rules.

Result: The output is a boolean `finalDecision`. If the model returns `false`, the component is removed from the sentence's mapping. This filtering step significantly reduces false positives by enforcing strict semantic boundaries between architecturally similar entities.

5.1.4. Component Information Extraction

Mere names are often insufficient to locate a component in a complex codebase. A component named *Logic* could be located in 'src/main/java/logic', 'src/core/business', or scattered across multiple packages. To aid the Code Preprocessing agent in locating these components, we extract detailed architectural metadata for each identified component.

5.1.4.1. Extraction Strategy

For every component name extracted in the previous step, we trigger a specific extraction task. The LLM is provided with the full documentation and the target component name. It is instructed to scan the text for specific hints that might indicate where this component lives in the file system.

5.1.4.2. Extracted Metadata

The model is constrained to return a structured set of descriptive fields. Unlike the previous steps which focused on normalization, this step focuses on gathering search heuristics :

- **Packages:** Explicit mentions of package names (e.g., "`org.jabref.logic`") associated with the component.
- **Directories:** Explicit file paths or root folders mentioned in the text (e.g., "The web assets are located in `src/web/`").
- **File Types:** The specific file extensions or technologies associated with the component (e.g., ".jsp", ".ts", ".xml"). This allows the agent to verify if a candidate directory contains the expected artifacts.
- **Named Entities:** Specific classes, configuration files, or unique identifiers mentioned as part of the component (e.g., "Contains the `UserEntity` class" or "Configured via `web.xml`"). These serve as *anchors* for verification.
- **Purpose:** A high-level description of the component's responsibility (e.g., "Handles database persistence"). This supports semantic similarity searches if exact name matching fails.

- **Production vs. Test:** A classification of whether the component represents production code or testing infrastructure. This helps the agent distinguish between ‘src/main‘ and ‘src/test‘ hierarchies, a common source of ambiguity.

This rich metadata profile provides the Agentic Code Processor with a set of specific criteria to validate its search results, significantly increasing the probability of locating the correct physical directory.

5.2. Code Processing

While Documentation Preprocessing identifies *what* needs to be found, Code Processing identifies *where* it exists. The objective is to resolve the extracted component names into concrete source code directories.

In well-engineered software, architectural boundaries often align with technical boundaries. Developers utilize grouping mechanisms, such as placing all *Payment* logic into a `com.project.payment` package or a distinct microservice directory, to enforce modularity. This explicit structural intent serves as a high-fidelity signal of component location. By clustering these directories, we capture the *physical* architecture of the system.

As shown in Figure 5.1, the result of this stage is a mapping of **Component Names** to **Directories**. Any source code artifact contained within these directories is considered part of the component. We employ two complementary strategies to achieve this mapping: an Agentic approach and a Project-Based approach.

5.2.1. Agentic Based

The fundamental challenge in this stage is the semantic gap between documentation and code. A component described as *The Payment Gateway* in the SAD might be implemented in a directory named `acme-pay-lib`. Static analysis cannot easily bridge this gap; it requires *human-like* judgment to infer that the two entities are equivalent based on context.

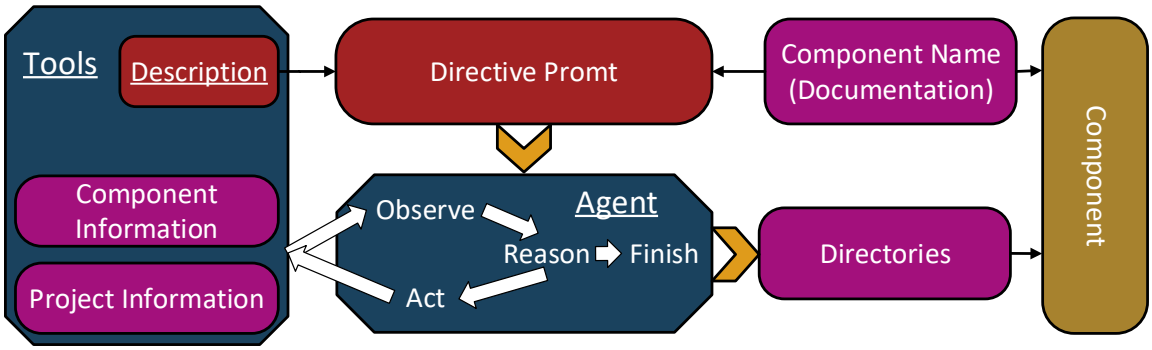

Figure 5.2.: Overview of the Agentic Based Approach

To automate this reasoning, we employ an LLM-based Agent designed to mimic a human developer exploring a repository. The workflow is illustrated in Figure 5.2. The Agent operates on the ReAct (Reasoning → Action → Observation) principle and is provided with the **Component Name** and the **Component Information** extracted in the previous stage.

The Agent is equipped with a set of file-system tools:

- **List Directories:** Allows the agent to traverse the project structure.
- **Fuzzy Search:** To overcome exact-match failures, the agent can search for directories similar to a given string. This uses an embedding-based ranking to return the top-k most similar directory names in the project.
- **File Type Analysis:** The agent can request a summary of file extensions in a directory. This allows it to verify, for example, that a *WebUI* component actually contains `.html` or `.ts` files.

The Agent iteratively explores the codebase, hypothesizing locations based on the component name (e.g., "I should look for a folder named 'ui' or 'web'"), executing tools, and refining its search based on observations. This allows it to locate components even when naming conventions are inconsistent or when components are nested deeply within the hierarchy.

5.2.2. Project Based

While the Agentic approach infers structure through exploration, the *Project Based* approach extracts structure explicitly defined by the project's build configuration or language semantics.

5.2.2.1. Build Configuration Extraction (Maven)

This strategy relies on the standard directory layout enforced by the Maven build system. It scans the repository for the existence of *pom.xml* files. In multi-module Maven projects, the location of these files often demarcates the root of a distinct architectural module.

Mechanism: The system traverses the file tree to locate all *pom.xml* files. The directories containing these files are extracted as candidate component roots. Limitations: This approach is strictly limited to Maven-based project structures. Consequently, for projects utilizing other build systems, such as *TEAMMATES* and *JabRef*, which rely on Gradle, this heuristic yields no results. It is therefore most effective when applied to projects like *TeaStore* or *BigBlueButton* (Java modules).

5.2.2.2. Package Structure Analysis (Java AST)

To address the limitations of build-file dependence, the second strategy leverages the semantic structure of the Java source code itself. Instead of looking at file system metadata, this approach analyzes the package hierarchy defined in the Abstract Syntax Tree (AST).

Mechanism: The system parses the Java files to construct the full package hierarchy of the project. It then traverses this hierarchy from the root package (e.g., *com*) downwards. The *First Split* Heuristic: The traversal continues as long as a package has exactly one subpackage (linear hierarchy). The traversal stops at the *first split*, the point in the hierarchy

where a package branches into two or more sub-packages (e.g., *com.project* splitting into *com.project.ui* and *com.project.logic*). Resolution: The sub-packages resulting from this split are identified as the *Component Root Packages*. These semantic packages are then resolved back to their corresponding physical directories on the file system.

5.2.2.3. Mapping to SAD Components

The final step is to reconcile the *Technical Names* extracted from these heuristics (e.g., the directory name of a *pom.xml* or the suffix of a split package) with the *Documentation Names* extracted from the SAD. We employ an LLM to perform this matching based on string similarity and semantic equivalence.

6. Evaluation

6.1. Experiment Setup

To ensure reproducibility and comparability, the experiments were conducted using a strictly versioned environment for both the data and the underlying models.

6.1.1. Dataset Acquisition and Versioning

The source code for all subject projects was acquired by cloning the repositories at the specific Git revision hashes defined in the benchmark suite [14]. This ensures that the code structure analyzed in our experiments matches exactly with the artifacts referenced in the gold standards, preventing drift due to project evolution.

6.1.2. Large Language Models

The experiments utilized two primary model configurations to evaluate the impact of reasoning capabilities on traceability performance:

- **GPT-4o (referred to as gpt-4.1):** Used as the standard baseline for the Agentic approach and double-check mechanisms.
- **GPT-o1 (referred to as gpt-5):** Deployed in specific ablation studies to assess the performance gains from advanced reasoning capabilities in component recovery tasks.

All models were accessed via the OpenAI API with temperature settings fixed at 0.0 to minimize non-deterministic variance in the outputs.

6.1.3. Evaluation Methodology and Metrics

While the **LiSSA** and **ArDoCo** frameworks provide built-in calculators for standard Information Retrieval metrics (Precision, Recall, F1), this study primarily utilizes *Sentence-Normalized Scores* to prevent large components from skewing the results.

6.1.3.1. Calculation of Sentence-Normalized Scores

To compute these scores, we implemented a post-processing mapping strategy:

1. **Aggregation:** Both the retrieved trace links (from the framework output) and the expected links (from the Gold Standard) are grouped by their corresponding Sentence ID.
2. **Scoring:** For each sentence, we calculate the local Precision, Recall, and F1-Score based on the set of links associated with that specific sentence ID (as defined in subsection 2.3.4).
3. **Averaging:** The final system score is the macro-average of these local sentence scores.

6.1.3.2. Implementation and Validation

For the **LiSSA** framework, this logic was integrated directly into the analysis pipeline, allowing it to run automatically at the conclusion of an evaluation cycle. For the **ArDoCo** baselines, the scores were computed externally using a custom evaluation script processing the framework's raw output files. This script is available in the replication package attached to this thesis.

To validate the correctness of this external calculation, we also computed the standard global metrics (accumulated confusion matrix) using the script and compared them against the native reports generated by ArDoCo. The results matched exactly, confirming that the calculation logic for the sentence-normalized scores is consistent and representative of the baselines' performance.

6.2. Component Recovery Evaluation

To assess the efficacy of the Code Preprocessing stage, we conducted an isolated evaluation of the component recovery mechanism. The objective was to determine how accurately the system could map a *Component Name* (as defined in the SAD) to the correct set of source code artifacts.

6.2.1. Methodology

For this evaluation, we manually identified the target components in the Gold Standards and compared them against the components constructed by our Code Preprocessing approaches. We calculated the *F1-Score* based on the retrieval of artifacts:

- **True Positives:** Artifacts correctly assigned to the component (present in both the approach's output and the Gold Standard).

Table 6.1.: F1-Score Averages across all Datasets considering only the Results of the Code Preprocessing of the *expected* Components

Approach			GS Reworked All Mentioned	GS Reworked Java/Shell	GS Old
Agent gpt-4.1			.830	.765	.870
Agent gpt-5			**.968**	**.968**	.890
Agent gpt-4.1	Packages	Build Configs	.939	.927	.910
	Packages		.795	.924	**.915**
	Packages	Build Configs	.865	.924	**.915**

- **False Positives:** Artifacts incorrectly assigned to the component.
- **False Negatives:** Valid artifacts missed by the approach.

We evaluated multiple configurations, including the pure *Agentic* approach (using different underlying models) and the *Project-Based* heuristics (Packages and Build Configurations). These were tested against three baseline datasets: the original Gold Standard ("GS Old"), and our refined standards (*GS Reworked*) filtered for all mentioned artifacts or restricted to Java/Shell files.

6.2.2. Results

The results, summarized in Table 6.1, demonstrate the high fidelity of the proposed preprocessing strategies.

6.2.2.1. Agentic vs. Heuristic Performance

The data reveals that the Agentic approach utilizing GPT-5 achieves the highest overall performance, with a near-perfect F1-score of **.968** on the refined gold standards. This suggests that advanced reasoning capabilities are crucial for bridging the semantic gap between documentation names and code directories, particularly when naming conventions are inconsistent.

However, the Project-Based approach combining both *Packages* and *Build Configs* heuristics also performs exceptionally well (.939), outperforming the GPT-4 based agent. This validates the hybrid strategy: explicit structural signals (like ‘pom.xml‘ or package splits) provide a robust baseline, while the Agent is necessary to handle edge cases or non-standard structures that heuristics miss.

Notably, the heuristic approaches perform significantly better on the "Java/Shell Only" subset (.924) than on the full artifact set (.795). This is expected, as the "Packages" heuristic is inherently designed to traverse Java ASTs and may miss non-code resources (like HTML or config files) that are part of the component but reside outside the package hierarchy. The Agent, being file-system aware, does not suffer from this limitation.

6.3. Feature Comparison and Ablation Study

To understand the individual contributions of the proposed components, we performed a comparative analysis of different pipeline configurations. Table 6.2 presents the average F1-scores across the datasets, distinguishing between the impact of the Ambiguity Resolution (Double Check) mechanism and the different Code Preprocessing strategies.

6.3.1. Impact of Ambiguity Resolution (Double Check)

The comparison reveals a distinct divergence in performance depending on the target Gold Standard.

- **Performance on Refined Standards:** On the *GS Reworked All Mentioned* dataset, the **Double Check** mechanism yields the highest overall performance. The Agentic approach with Double Check achieves a weighted average F1-score of **.554**, significantly outperforming the configuration without Double Check (.412). This confirms that the filtering step is crucial for high-precision scenarios where distinguishing between semantically similar components (e.g., "Web" vs. "WebUI") is required.
- **Inverse Trend on Legacy Standards:** Conversely, on the *GS Old* dataset, the configuration *without* Double Check performs better (.535 vs. .513). This aligns with our findings regarding "Component Flooding" in the original benchmarks. Since the old standards often linked artifacts broadly without semantic precision, the aggressive filtering of the Double Check mechanism is penalized as "False Negatives," while the noisier, unfiltered approach is rewarded.

6.3.2. Agentic vs. Heuristic Code Preprocessing

The table also highlights the limitations of pure project-based heuristics when dealing with diverse artifact types.

- **Artifact Diversity:** When considering all artifact types (*GS Reworked All Mentioned*), the **Agentic (GPT-4.1)** approach dominates, achieving a weighted average of **.554** compared to **.285** for the combined Heuristic approach (Pack + Poms). This disparity is driven by the heuristics' inability to locate non-code artifacts (e.g., documentation, HTML, configuration) that exist outside the package hierarchy.
- **Code-Centric Parity:** When the task is restricted to Java and Shell files (*GS Reworked Java/Shell*), the heuristic approaches become competitive (.362 vs. .372). This suggests that for purely code-focused traceability, standard project structure analysis (AST and Build Configs) is a viable, lower-cost alternative to agentic exploration.

Table 6.2.: Comparison of Average F1-Scores across all Gold Standards

Approach		GS Reworked All Mentioned		GS Reworked Java/Shell		GS Old	
		Avg. (wo/ TM)	w. Avg. (wo/ TM)	Avg.	w. Avg.	Avg.	w. Avg.
Double Check	Agent gpt-4.1	**.436** (.389)	**.554** (**.475**)	.399	**.469**	.513	.494
	Agent gpt-5	.377 (.363)	.404 (.353)	.371	.372	––	
	Agent gpt-4.1 + Pack + Poms	.371 (**.419**)	.285 (.473)	.382	.363	.497	.487
	Pack	.370 (.416)	.286 (.454)	.396	.362	**.561**	.504
	Pack + Poms	.349 (.387)	.286 (.454)	.354	.361	.500	.490
wo/ Double Check	Agent gpt-4.1	.409 (.408)	.412 (.416)	**.404**	.396	.535	**.521**

6.4. Comparative Evaluation against Baselines

To contextually validate the performance of the proposed *Component-Centric* approach, we compared selected configurations (Agentic and Heuristic/Pack) against two distinct classes of baselines:

1. **Naive LiSSA Baselines:** Standard RAG configurations (‘Sentence/None/k‘) that retrieve a fixed number of code chunks ($k = 20, 40$) directly for each sentence without intermediate component resolution.
2. **State-of-the-Art Approaches (ArDoCo):** The *Trace Link Recovery* (TLR) strategies provided by the ArDoCo framework, specifically *ExArch* (Extended Architecture) and *TransArC* (Transformer-based Architecture Recovery). These represent the current gold standard for model-based traceability.

The comparative results are summarized in Table 6.3.

6.4.1. Comparison with Naive Retrieval

The most immediate finding is the stark performance gap between the proposed approach and the naive baselines. On the *GS Reworked All Mentioned* dataset, the naive LiSSA configuration ($k = 40$) achieves a weighted average F1-score of only **.125**. in contrast, the proposed Agentic approach achieves **.554**.

This discrepancy empirically validates the core hypothesis of this thesis: architectural components vary too widely in size for fixed-k retrieval to be effective. A naive retrieval either misses the majority of a large component (low recall) or floods the results for a small component (low precision). The Component-Centric approach, by resolving the *scope* of the component first, dynamically adjusts the retrieval window, resulting in a 4x improvement in F1-score.

6.4.2. Comparison with State-of-the-Art (ArDoCo)

The comparison with ArDoCo reveals a nuanced trade-off between *Model-Based* and *Agentic* strategies, particularly regarding artifact diversity.

Table 6.3.: Comparison of Average F1-Scores across all Gold Standards

Approach		GS Reworked All Mentioned		GS Reworked Java/Shell		GS Old	
		Avg. (wo/ TM)	**w. Avg. (wo/ TM)**	**Avg.**	**w. Avg.**	**Avg.**	**w. Avg.**
LiSSA	Agent gpt-4.1	**.436** (.389)	**.554** (.475)	.399	**.469**	.513	.494
	Pack	.370 (.416)	.286 (.454)	.396	.362	.561	.504
Baselines		**Avg. (wo/ TM)**	**w. Avg. (wo/ TM)**	**Avg.**	**w. Avg.**	**Avg.**	**w. Avg.**
LiSSA	Sentence/None/20	.137 (.152)	.086 (.070)	.160	.128	.130	.054
	Sentence/None/40	.135 (.136)	.125 (.099)	.154	.167	.133	.077
ArDoCo	ExArch	.389 (.434)	.320 (.536)	.432	.441	**.733**	**.866**
	ExArch 5	.406 (**.457**)	.320 (**.537**)	**.450**	.441	——	
	TransArC	.431 (**.463**)	.382 (**.545**)	**.498**	**.526**	**.797**	**.882**

6.4.2.1. Polyglot vs. Monolingual Performance

On the *GS Reworked All Mentioned* dataset, the Agentic LiSSA approach outperforms the best ArDoCo configuration (TransArC) in weighted average F1-score (**.554** vs. **.382**). This advantage is primarily driven by the *TEAMMATES* project.

- **With TEAMMATES:** The Agentic approach excels because it can navigate the file system to locate TypeScript, HTML, and configuration files, which ArDoCo's Java-centric model extraction typically overlooks.
- **Without TEAMMATES:** When looking at the parenthesized values (excluding TEAMMATES), TransArC regains the lead (**.545** vs. **.475**).

This indicates that while ArDoCo remains superior for pure Java architectures (JabRef, TeaStore), the Agentic approach provides critical robustness for modern, polyglot web architectures.

6.4.2.2. Performance on Legacy Standards

On the *GS Old* dataset, ArDoCo demonstrates dominant performance (**.882** vs. **.494**). This aligns with our qualitative analysis of the benchmarks: the original gold standard heavily favors static Java structures (packages/classes) and often excludes the runtime configurations and non-code artifacts that the Agentic approach is designed to find. Essentially, ArDoCo is optimized for the *Static Structure* view of architecture, while the Agentic approach is optimized for the *Semantic Intent* view captured in the refined gold standard.

6.5. Threats to Validity

To ensure a balanced interpretation of the findings, we discuss the potential threats to the validity of this study, categorized into internal, external, construct, and reliability threats.

6.5.1. Internal Validity

Internal validity concerns factors that might have influenced the results within the experimental setting.

6.5.1.1. Subjectivity in Gold Standard Refinement

The most significant threat to the internal validity of this thesis is the potential bias in the refinement of the gold standards. The re-evaluation of the datasets—shifting from named-entity matching to intent-based matching—was conducted by a single author. While strict guidelines were established to standardize the decision-making process (e.g., regarding dead references and semantic intent), the lack of inter-rater agreement checking means the refined standards inevitably reflect the subjective architectural interpretation of the author. Consequently, the reported performance improvements on the refined datasets must be interpreted as improvements relative to this specific interpretation of traceability, rather than an objective, proven truth.

6.5.1.2. Implementation Correctness

The calculation of *Sentence-Normalized Scores* required a custom implementation to post-process the output of both the LiSSA and ArDoCo frameworks. Errors in this aggregation logic could skew the comparison. To mitigate this, we validated the implementation by computing standard global metrics (Precision/Recall) using the same script and confirming they matched the native reports generated by the ArDoCo framework.

6.5.2. External Validity

External validity concerns the generalizability of the results to other software projects.

6.5.2.1. Sample Size and Selection Bias

The evaluation was limited to four open-source projects (*JabRef*, *MediaStore*, *TEAMMATES*, and *TeaStore*). While these projects were selected to represent a degree of heterogeneity (monolithic desktop, distributed web, microservices), they share a common ecosystem: they are all primarily Java-based or heavily rooted in Java architecture. The heuristics developed for the "Project-Based" preprocessing (e.g., Maven/Gradle analysis, Java AST traversal) are language-specific. Therefore, the results may not generalize to ecosystems with fundamentally different structural paradigms (e.g., Python, C++, or Go projects).

6.5.2.2. Quality of Documentation

The SADs used in this study, while varying in style, are relatively structured and complete. The proposed Component-Centric approach relies on the assumption that the documentation contains extractable "Component Names" and descriptive "Component Information." The approach may degrade significantly when applied to sparse, outdated, or low-quality documentation where architectural intent is not explicitly stated.

6.5.3. Construct Validity

Construct validity questions whether the experiment actually measures what it claims to measure.

6.5.3.1. The Directory-Based Component Assumption

A core construct of our approach is the assumption that architectural components map to physical directories in the file system. While this holds true for many modular architectures (e.g., Microservices or clean Package-by-Feature layouts), it fails to capture cross-cutting concerns (e.g., aspect-oriented implementations) or "scattered" components where a logical entity is spread across multiple disjoint packages. In such cases, the Component-Centric preprocessing would likely fail to locate the correct scope, leading to lower recall compared to naive retrieval methods.

6.5.3.2. Metric Selection

We utilized *Sentence-Normalized F1-Scores* as the primary metric. While we argue this is more representative for architectural traceability (preventing large components from dominating the score), it makes direct comparison with other literature—which often reports global metrics—more difficult. A system could theoretically perform well on sentence-normalization but poorly on global recall if it consistently misses the largest, most complex components.

6.5.4. Reliability

Reliability refers to the reproducibility of the results.

6.5.4.1. LLM Stochasticity

The experiments relied on proprietary Large Language Models (GPT-4o, GPT-o1). Despite setting the temperature parameter to 0.0 to maximize determinism, these models are inherently non-deterministic and subject to backend updates by the provider. Therefore, exact replication of the generated component names and trace links cannot be guaranteed over time, even with identical prompts.

7. Conclusion

Traceability Link Recovery (TLR) between Software Architecture Documentation (SAD) and source code is a critical enabler for software comprehension, compliance verification, and architectural consistency checking. However, established approaches have historically relied on a "static" view of traceability—assuming that architectural concepts map cleanly to explicit code structures like packages or classes. This thesis challenged that assumption by demonstrating that valid traceability links are often context-dependent, relying on runtime configurations, project-specific conventions, and implicit architectural knowledge that static analysis alone cannot capture.

7.1. Summary of Contributions

This work makes three primary contributions to the field of automated traceability:

7.1.1. 1. Identification of Contextual Dependency

Through a rigorous manual analysis of four open-source projects (*JabRef*, *MediaStore*, *TEAMMATES*, and *TeaStore*), I revealed that the "truth" of an architectural link often lies outside the Java source code. I identified that valid links are frequently established by:

- **Configuration files** (e.g., XML, JSON, Gradle) that define runtime injection and component boundaries.
- **Implicit conventions** (e.g., suffix-based naming) that bridge the gap between documentation terms and code artifacts.
- **Polyglot artifacts** (e.g., TypeScript, HTML) that constitute the actual implementation of "UI" or "Web" components in modern architectures.

This analysis exposed a fundamental "structural mismatch" in existing benchmarks, where gold standards either flooded results with entire components (low precision) or ignored non-Java artifacts entirely (low recall).

7.1.2. 2. Refinement of Traceability Benchmarks

Based on these findings, I systematically refined the gold standards for the analyzed projects. I transitioned from a "Named Entity" baseline to an "Intent-Based" baseline, filtering out 31% of unreasonable links and adding 48% of missing context. The refined benchmarks provide a more realistic target for modern TLR approaches, rewarding precision and penalizing the "component flooding" common in previous standards.

7.1.3. 3. Component-Centric Preprocessing Approach

To address the identified challenges, I proposed a novel Component-Centric Preprocessing stage for the LiSSA framework. Instead of retrieving a fixed number of code chunks (k) for every sentence, our approach introduces an intermediate reasoning step:

1. **Documentation Preprocessing:** Extracts architectural intent to identify *which* components a sentence describes.
2. **Code Preprocessing:** Uses an Agentic or Project-Based strategy to resolve *where* those components physically reside (directories).
3. **Ambiguity Resolution:** A *Double Check* mechanism that actively filters out false positives caused by similar naming conventions.

This allows the subsequent retrieval stage to dynamically adjust its scope, targeting the specific directory sub-tree relevant to the architectural concept.

7.2. Key Findings

The evaluation of the proposed approach against state-of-the-art baselines (ArDoCo) and naive retrieval methods yielded several key insights:

- **Superiority in Diverse Architectures:** On the refined, diverse benchmark (*GS Reworked All Mentioned*), the Agentic approach achieved a weighted average F1-score of **.554**, significantly outperforming the best naive baseline (.125) and the state-of-the-art ArDoCo TransArC (.382). This performance gap is largely driven by the agent's ability to navigate polyglot file systems (e.g., in *TEAMMATES*), whereas traditional model-based approaches are often confined to Java ASTs.
- **The Necessity of Reasoning:** The ablation study confirmed that "reasoning matters." The Agentic approach utilizing GPT-5 for component recovery achieved a near-perfect F1-score of **.968**, compared to .830 for GPT-4. Furthermore, the "Double Check" mechanism proved essential for high-precision tasks, improving the F1-score on the refined benchmark from .412 to .554 by effectively filtering ambiguity.

- **Bias in Legacy Benchmarks:** Interestingly, the heuristic and model-based baselines (ArDoCo) continued to dominate on the old gold standards (*GS Old*). This inverse correlation confirms our hypothesis that legacy benchmarks are biased toward static code structures, punishing approaches that attempt to recover the more semantic, configuration-driven links identified in our refinement.

7.3. Future Work

While the Component-Centric approach demonstrates significant improvements, several avenues for future research remain:

- **Expansion of Artifact Types:** The current code preprocessing focuses heavily on directory-based components. Future work could extend this to support "scattered" components (like the cross-cutting *Preferences* in JabRef) that are defined by tagging interfaces or annotations rather than folder structure.
- **Dynamic Configuration Analysis:** Currently, configuration files are treated as text or simple existence signals. Integrating a deeper, semantic understanding of build definitions (e.g., parsing the actual dependency graph in Gradle) could further improve the accuracy of the Project-Based heuristics.
- **Broader Project Scope:** Validating the approach on a larger set of industrial projects would help generalize the findings beyond the four open-source subjects analyzed here.

7.4. Closing Remarks

This thesis illustrates that "finding the code" is no longer just a search problem—it is a translation problem. By treating architectural documentation not as a bag of keywords but as a set of semantic intents that must be mapped to physical project structures, I can recover traceability links that reflect the true complexity of modern software systems. The proposed Component-Centric approach provides a robust, extensible foundation for this new generation of context-aware traceability.

Bibliography

[1] Ebube Alor, SayedHassan Khatoonabadi, and Emad Shihab. *Evaluating the Use of LLMs for Documentation to Code Traceability*. June 2025. DOI: 10.48550/arXiv.2506.16440. arXiv: 2506.16440 [cs]. (Visited on 06/23/2025).

[2] G. Antoniol et al. "Recovering Traceability Links between Code and Documentation". In: *IEEE Transactions on Software Engineering* 28.10 (Oct. 2002), pp. 970–983. ISSN: 1939-3520. DOI: 10.1109/TSE.2002.1041053.

[3] Giulio Antoniol et al. "Recovering Traceability Links Between Code and Documentation: A Retrospective". In: *IEEE Transactions on Software Engineering* 51.3 (Mar. 2025), pp. 825–832. ISSN: 1939-3520. DOI: 10.1109/TSE.2025.3534027.

[4] *ArDoCo Benchmark - TEAMMATES - Text Adaptation*. URL: https://github.com/ardoco/benchmark/blob/308dd2a96dac68e3399d948193a4c1386289809d/teammates/text_2021/teammates.txt (visited on 12/21/2025).

[5] *ArDoCo Benchmark - TEAMMATES Documentation - E2E*. URL: https://github.com/ardoco/teammates/blob/658280d58f799e9330d427e4ed81ac0e8145408c/docs/design.md#e2e-component (visited on 12/21/2025).

[6] *BigBlueButton*. URL: https://github.com/bigbluebutton/bigbluebutton (visited on 10/18/2025).

[7] Dave Binkley and Dawn Lawrie. "The Impact of Vocabulary Normalization". In: *Journal of Software: Evolution and Process* 27.4 (2015), pp. 255–273. ISSN: 2047-7481. DOI: 10.1002/smr.1710.

[8] Yingkui Cao et al. "Toward Accurate Link between Code and Software Documentation". In: *Science China Information Sciences* 61.5 (Apr. 20, 2018), p. 050105. ISSN: 1869-1919. DOI: 10.1007/s11432-017-9402-3.

[9] Giovanni Capobianco et al. "Improving IR-based Traceability Recovery via Noun-Based Indexing of Software Artifacts". In: *Journal of Software: Evolution and Process* 25.7 (2013), pp. 743–762. ISSN: 2047-7481. DOI: 10.1002/smr.1564.

[10] Xiaofan Chen and John Grundy. "Improving automated documentation to code traceability by combining retrieval techniques". In: *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*. ASE '11. USA: IEEE Computer Society, 2011, pp. 223–232. ISBN: 9781457716386. DOI: 10.1109/ASE.2011.6100057. URL: https://doi.org/10.1109/ASE.2011.6100057.

[11] Xiaofan Chen et al. "DCTracVis: A System Retrieving and Visualizing Traceability Links between Source Code and Documentation". In: *Automated Software Engineering* 25.4 (Dec. 1, 2018), pp. 703–741. ISSN: 1573-7535. DOI: 10.1007/s10515-018-0243-8.

[12] Stephane Ducasse and Damien Pollet. “Software Architecture Reconstruction: A Process-Oriented Taxonomy”. In: *IEEE Transactions on Software Engineering* 35.4 (July 2009), pp. 573–591. ISSN: 1939-3520. DOI: 10.1109/TSE.2009.19. (Visited on 06/21/2025).

[13] Dominik Fuchß et al. *Ardoco/Benchmark*. Aug. 2022. DOI: 10.5281/ZENODO.6966832.

[14] Dominik Fuchß et al. *Ardoco/Benchmark*. URL: https://github.com/ardoco/benchmark/tree/9444c8100421d7a6f49435b9af44e7b7c6940620 (visited on 10/18/2025).

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

[16] Dominik Fuchß et al. “Establishing a Benchmark Dataset for Traceability Link Recovery Between Software Architecture Documentation and Models”. In: *Software Architecture. ECSA 2022 Tracks and Workshops*. Ed. by Thais Batista et al. Cham: Springer International Publishing, 2023, pp. 455–464. ISBN: 978-3-031-36889-9. DOI: 10.1007/978-3-031-36889-9_30.

[17] Dominik Fuchß et al. “LiSSA: Toward Generic Traceability Link Recovery Through Retrieval- Augmented Generation”. In: *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 2025, pp. 1396–1408. DOI: 10.1109/ICSE55347.2025.00186.

[18] Klaas Andries de Graaf et al. “Ontology-based Software Architecture Documentation”. In: *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*. 2012, pp. 121–130. DOI: 10.1109/WICSA-ECSA.212.20.

[19] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. “Semantically Enhanced Software Traceability Using Deep Learning Techniques”. In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). May 2017, pp. 3–14. DOI: 10.1109/ICSE.2017.9.

[20] Emily Hill et al. “AMAP: Automatically Mining Abbreviation Expansions in Programs to Enhance Software Maintenance Tools”. In: *Proceedings of the 2008 International Working Conference on Mining Software Repositories*. MSR ’08. New York, NY, USA: Association for Computing Machinery, May 10, 2008, pp. 79–88. ISBN: 978-1-60558-024-1. DOI: 10.1145/1370750.1370771.

[21] Nurbay Irmak. “Software Is an Abstract Artifact”. In: *Grazer Philosophische Studien* 86.1 (2012), pp. 55–72. DOI: 10.1163/9789401209182_005.

[22] *ISO/IEC 19506:2012(En), Information Technology — Object Management Group Architecture-Driven Modernization (ADM) — Knowledge Discovery Meta-Model (KDM)*. URL: https://www.iso.org/obp/ui/en/#iso:std:iso-iec:19506:ed-1:v1:en (visited on 11/05/2025).

[23] *ISO/IEC/IEEE 42010:2022(En), Software, Systems and Enterprise — Architecture Description*. URL: https://www.iso.org/obp/ui/en/#iso:std:iso-iec-ieee:42010:ed-2:v1:en (visited on 11/05/2025).

[24] *JabRef in the Media*. URL: https://github.com/JabRef/jabref/wiki/JabRef-in-the-Media (visited on 10/18/2025).

[25] Yanjie Jiang et al. "Automated Expansion of Abbreviations Based on Semantic Relation and Transfer Expansion". In: *IEEE Transactions on Software Engineering* 48.2 (Feb. 2022), pp. 519–537. ISSN: 1939-3520. DOI: 10.1109/TSE.2020.2995736.

[26] Jan Keim et al. "Detecting Inconsistencies in Software Architecture Documentation Using Traceability Link Recovery". In: *20th IEEE International Conference on Software Architecture (ICSA)*. 2023, p. 141. ISBN: 979-8-3503-9749-9. DOI: 10.1109/ICSA56044.2023.00021. (Visited on 05/16/2025).

[27] Jan Keim et al. "Recovering Trace Links Between Software Documentation And Code". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. New York, NY, USA: Association for Computing Machinery, Apr. 12, 2024, pp. 1–13. ISBN: 979-8-4007-0217-4. DOI: 10.1145/3597503.3639130.

[28] Yves R. Kirschner et al. "Retriever: A view-based approach to reverse engineering software architecture models". In: *Journal of Systems and Software* 220 (2025), p. 112277. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2024.112277. URL: https://www.sciencedirect.com/science/article/pii/S0164121224003212.

[29] Yuzhan Ma et al. "Automatic Classification of Software Artifacts in Open-Source Applications". In: *Proceedings of the 15th International Conference on Mining Software Repositories*. MSR '18. New York, NY, USA: Association for Computing Machinery, May 28, 2018, pp. 414–425. ISBN: 978-1-4503-5716-6. DOI: 10.1145/3196398.3196446.

[30] Abdelrahman Mohamed et al. "A Review on Detecting and Managing Documentation Drift in Software Development". In: *2025 International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC)*. 2025 International Mobile, Intelligent, and Ubiquitous Computing Conference (MIUCC). Sept. 2025, pp. 546–552. DOI: 10.1109/MIUCC66482.2025.11196773.

[31] Kevin Moran et al. "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. New York, NY, USA: Association for Computing Machinery, Oct. 1, 2020, pp. 873–885. ISBN: 978-1-4503-7121-6. DOI: 10.1145/3377811.3380418.

[32] Rocco Oliveto et al. "On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery: A Ten-Year Retrospective". In: *Proceedings of the 28th International Conference on Program Comprehension*. ICPC '20. New York, NY, USA: Association for Computing Machinery, Sept. 12, 2020, p. 1. ISBN: 978-1-4503-7958-8. DOI: 10.1145/3387904.3394491.

[33] Rolf-Helge Pfeiffer. “What Constitutes Software? An Empirical, Descriptive Study of Artifacts”. In: *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR ’20. New York, NY, USA: Association for Computing Machinery, Sept. 18, 2020, pp. 481–491. ISBN: 978-1-4503-7517-7. DOI: 10.1145/3379597.3387442.

[34] Gregorio Robles, Jesus M. Gonzalez-Barahona, and Juan Julian Merelo. “Beyond Source Code: The Importance of Other Artifacts in Software Development (a Case Study)”. In: *Journal of Systems and Software*. Selected Papers from the Fourth Source Code Analysis and Manipulation (SCAM 2004) Workshop 79.9 (Sept. 1, 2006), pp. 1233–1248. ISSN: 0164-1212. DOI: 10.1016/j.jss.2006.02.048.

[35] Danissa V. Rodriguez and Doris L. Carver. “Comparison of Information Retrieval Techniques for Traceability Link Recovery”. In: *2019 IEEE 2nd International Conference on Information and Computer Technologies (ICICT)*. 2019 IEEE 2nd International Conference on Information and Computer Technologies (ICICT). Mar. 2019, pp. 186–193. DOI: 10.1109/INFOCT.2019.8710919.

[36] Dennis Steinbuch. “Ein Ansatz zur Traceability Link Recovery für natürlichsprachliche Software-Dokumentation und Quelltext”. In: (Jan. 25, 2024), p. 35. DOI: 10.5445/IR/1000167654. (Visited on 10/29/2025).

[37] Moon Ting Su, Christian Hirsch, and John Hosking. “KaitoroBase: Visual Exploration of Software Architecture Documents”. In: *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’09. USA: IEEE Computer Society, Nov. 2009, pp. 657–659. ISBN: 978-0-7695-3891-4. DOI: 10.1109/ASE.2009.26. (Visited on 06/21/2025).

A. Appendix

A.1. MediaStore Gold Standard Comparison

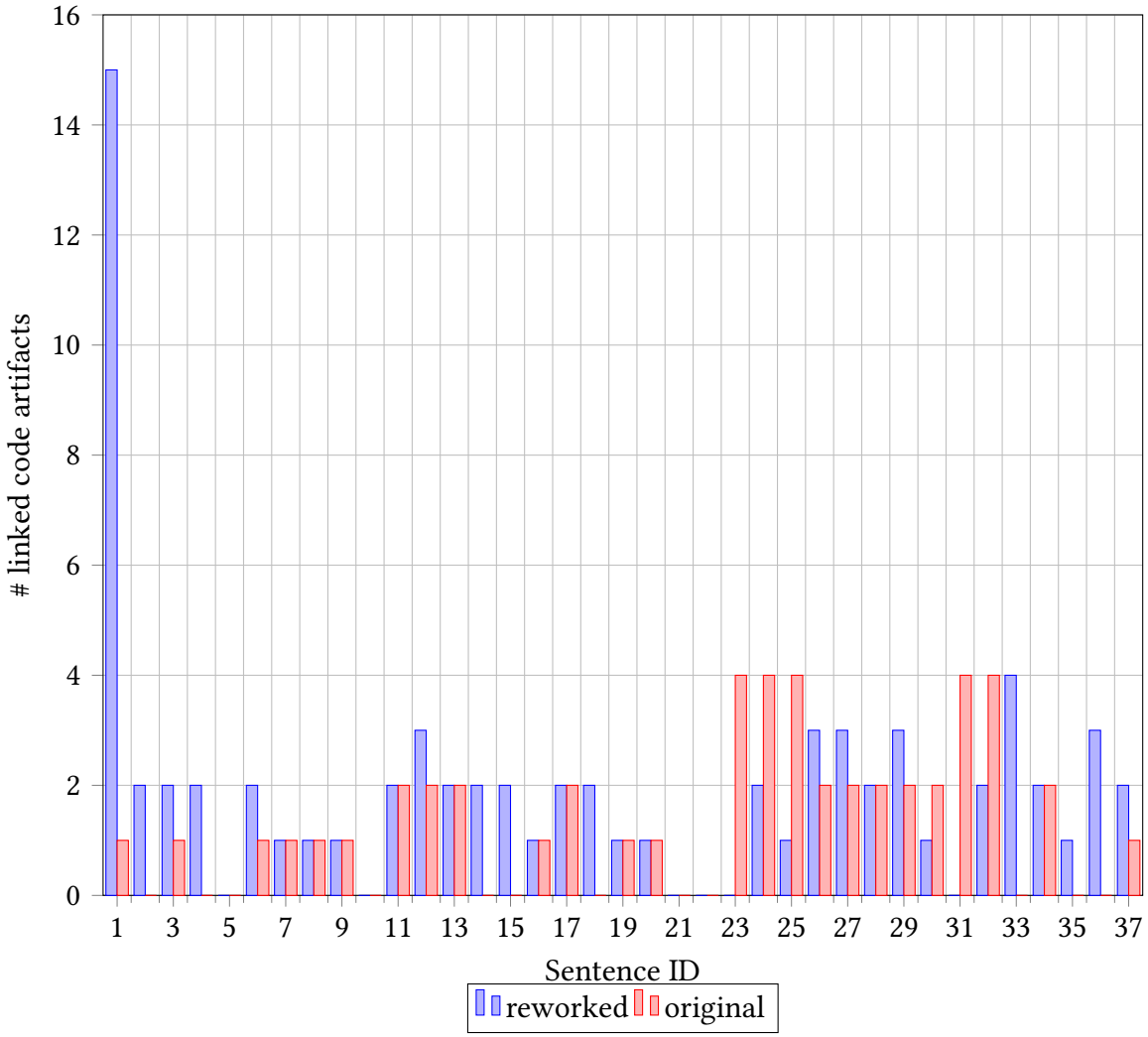

Figure A.1.: MediaStore reworked and original gold standard linked code artifacts comparison

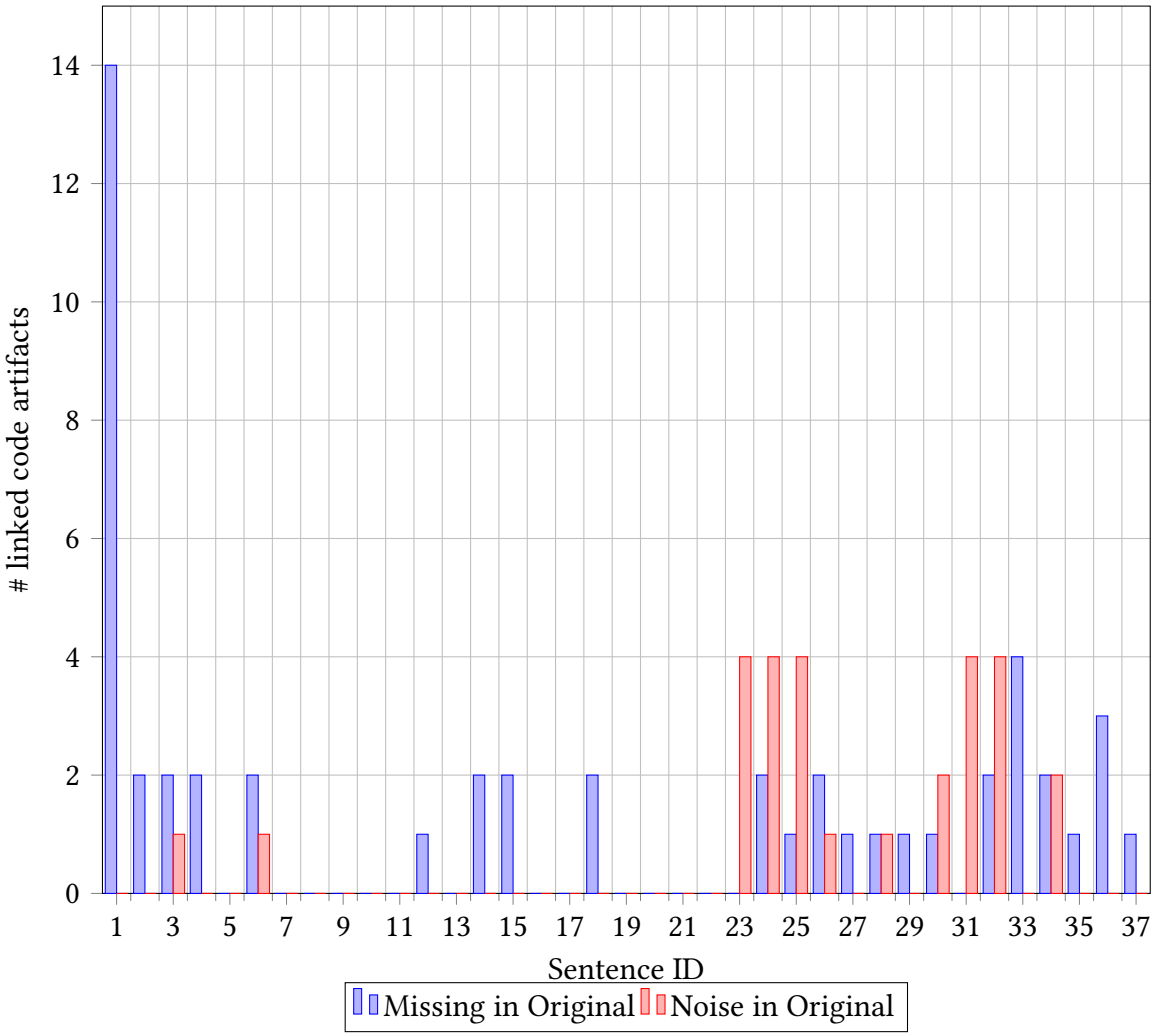

Figure A.2.: MediaStore original gold standard linked code artifacts difference to reworked

A.2. JabRef Gold Standard Comparison

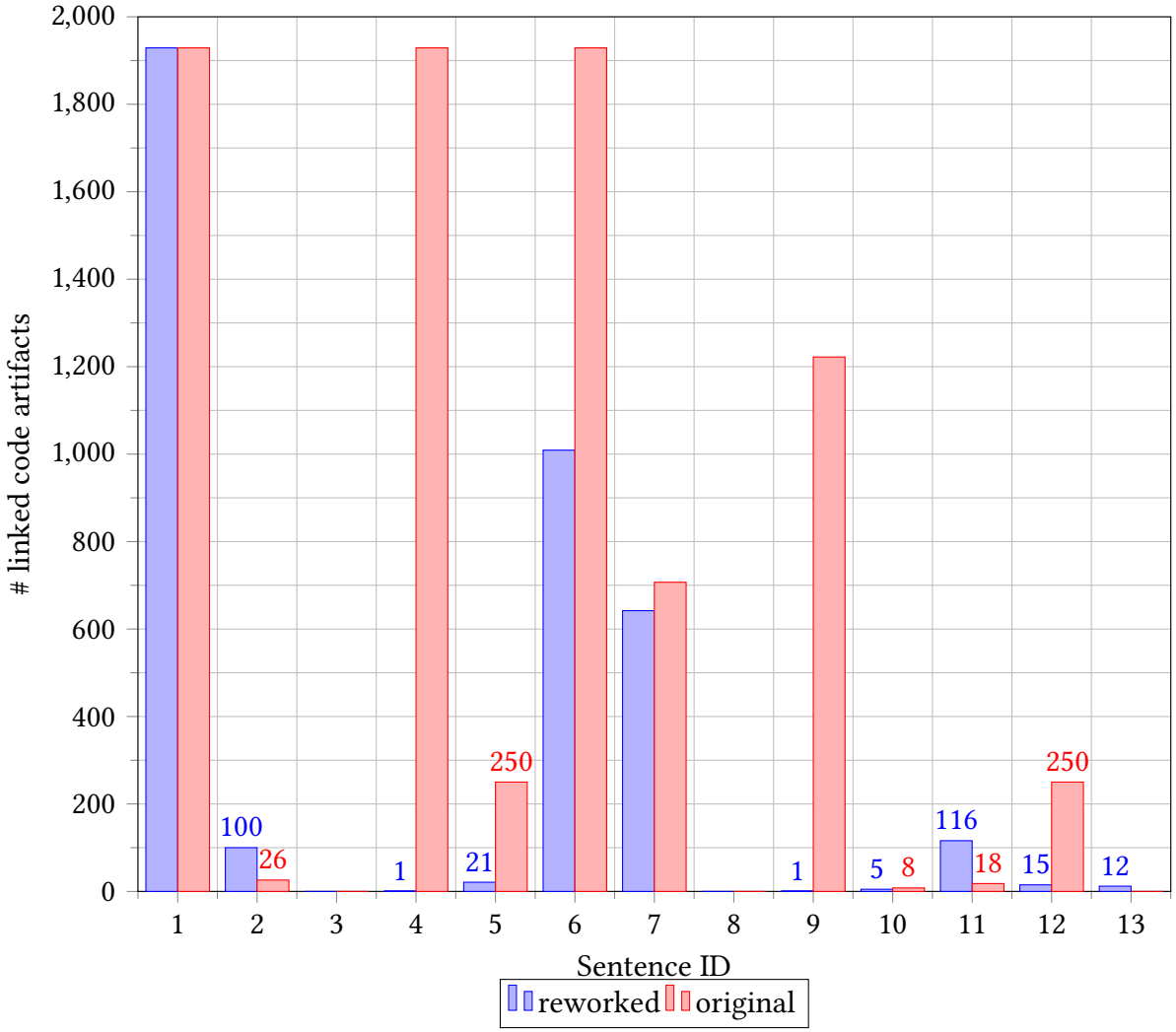

Figure A.3.: JabRef reworked and original gold standard linked code artifacts comparison

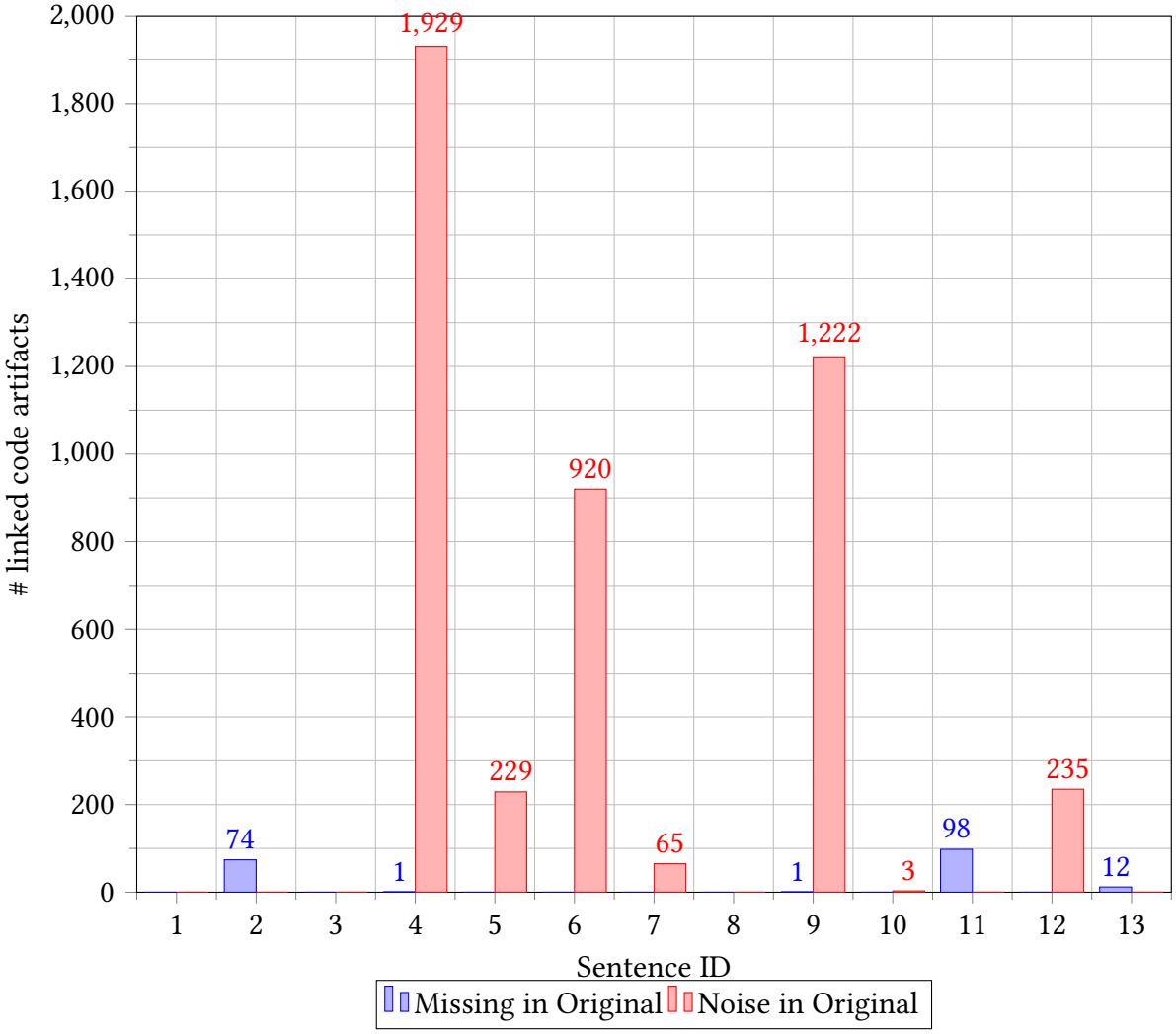

Figure A.4.: JabRef original gold standard linked code artifacts difference to reworked

A.3. TeaStore Gold Standard Comparison

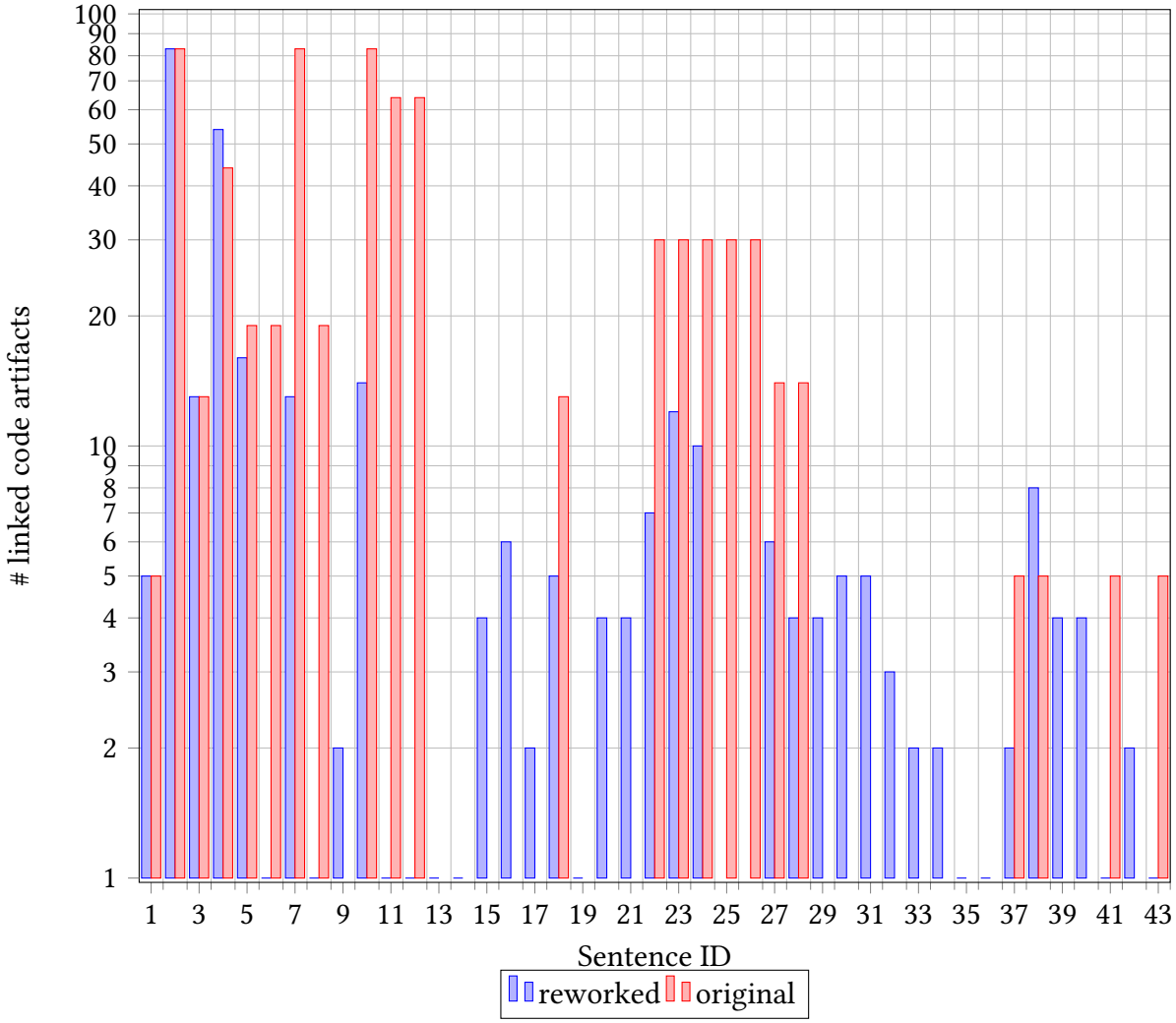

Figure A.5.: TeaStore reworked and original gold standard linked code artifacts comparison

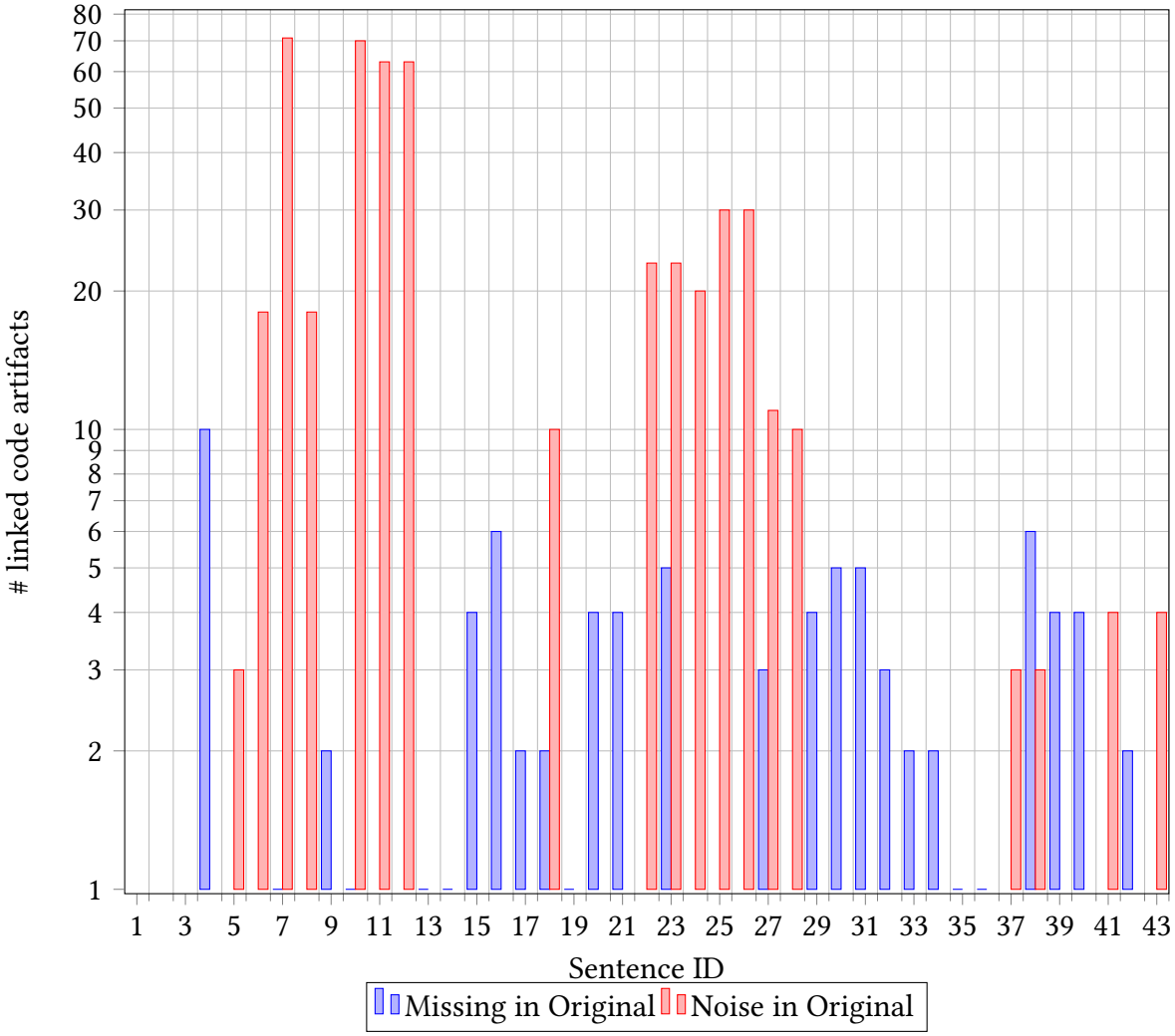

Figure A.6.: TeaStore original gold standard linked code artifacts difference to reworked

A.4. TEAMMATES Gold Standard Comparison

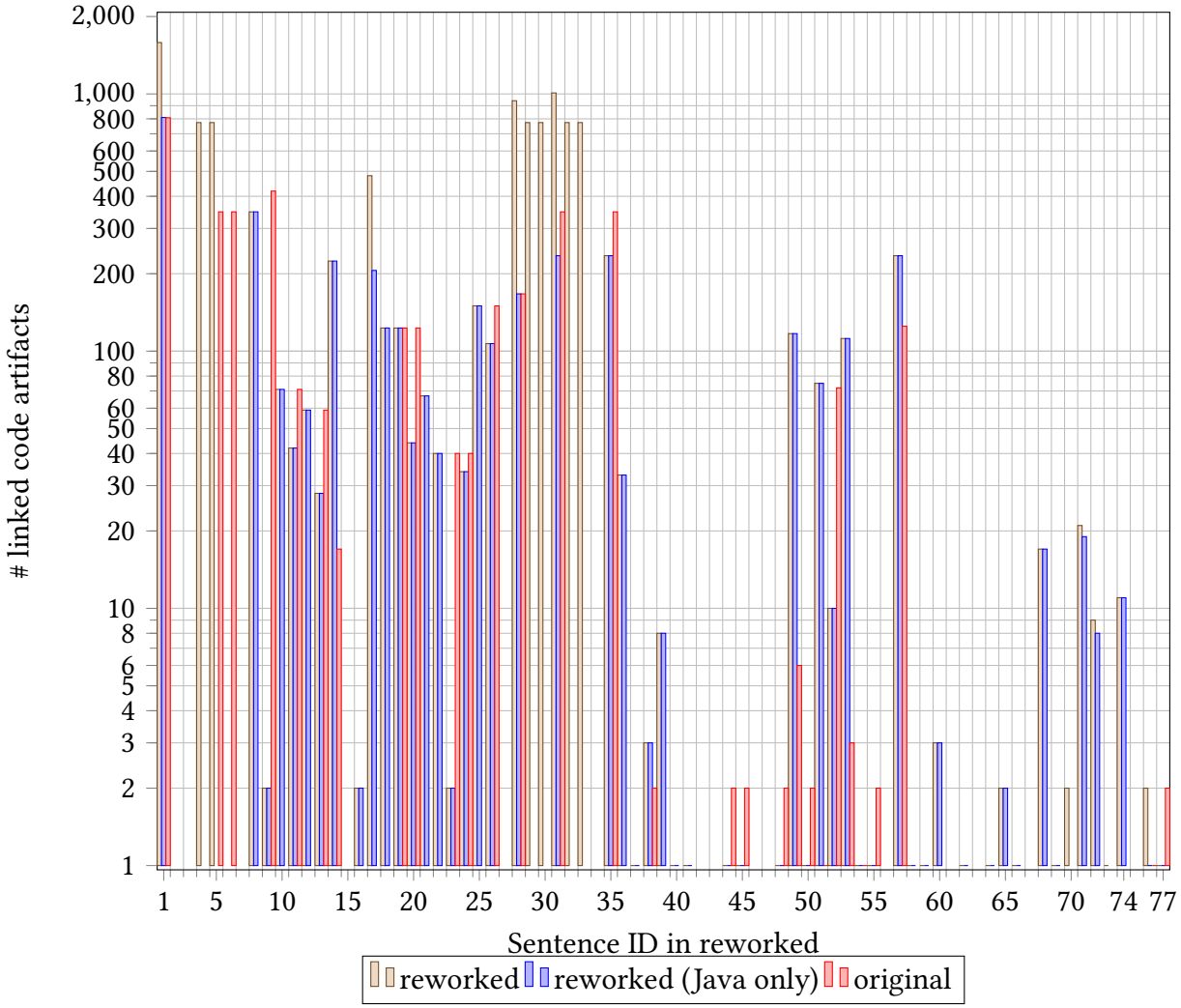

Figure A.7.: TEAMMATES reworked and original gold standard linked code artifacts comparison

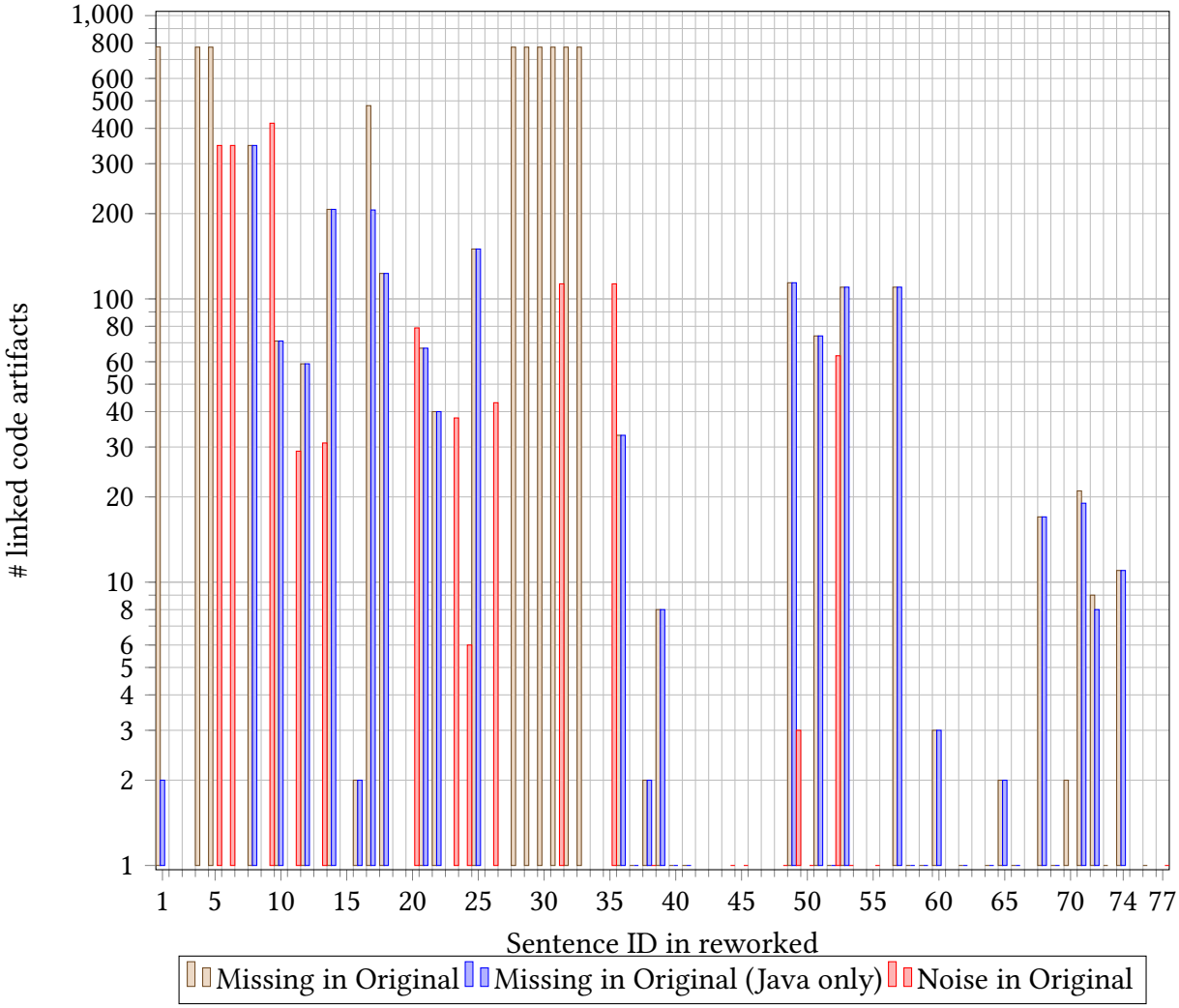

Figure A.8.: TEAMMATES original gold standard linked code artifacts difference to reworked

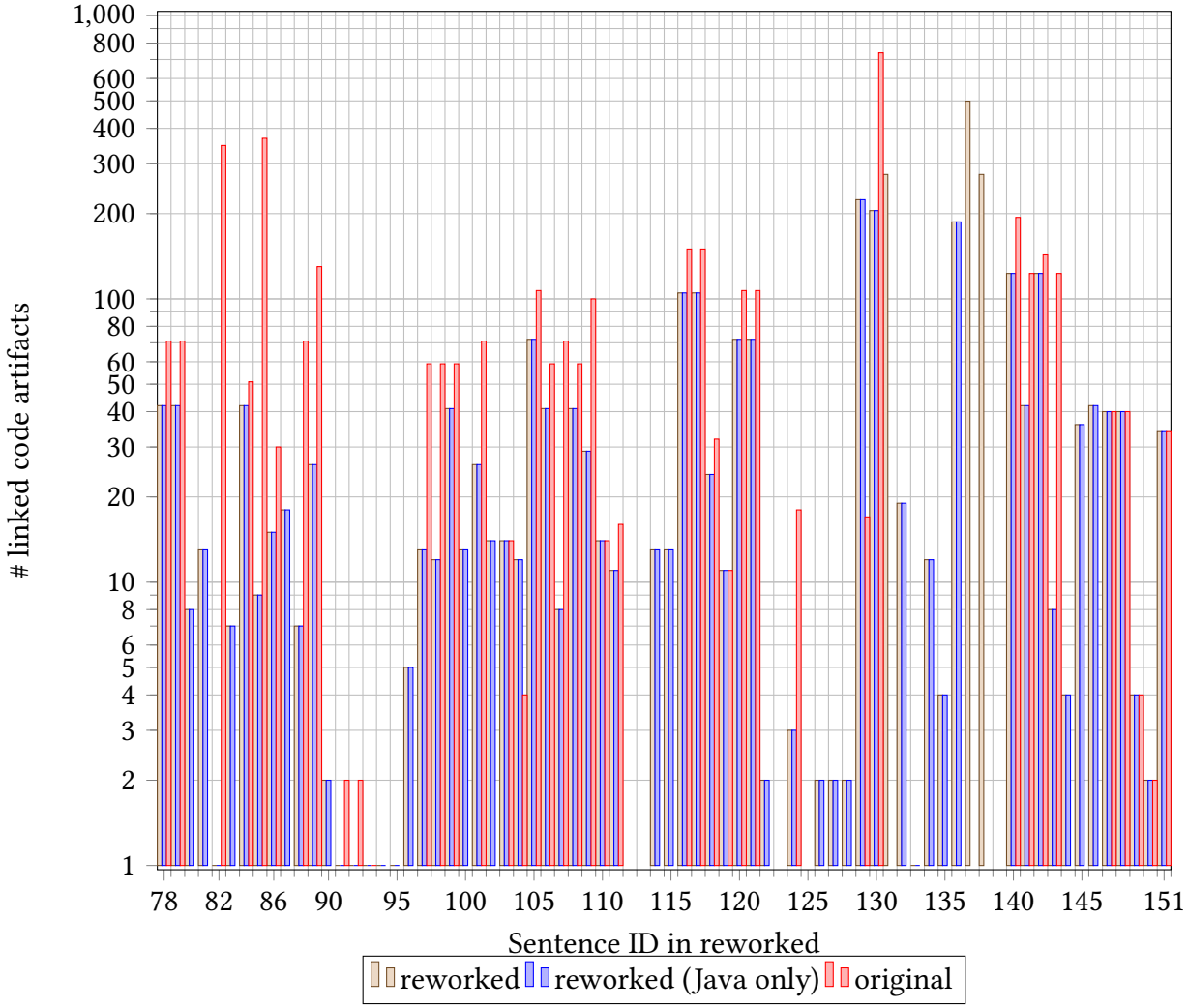

Figure A.9.: TEAMMATES reworked and original gold standard linked code artifacts comparison - part 2

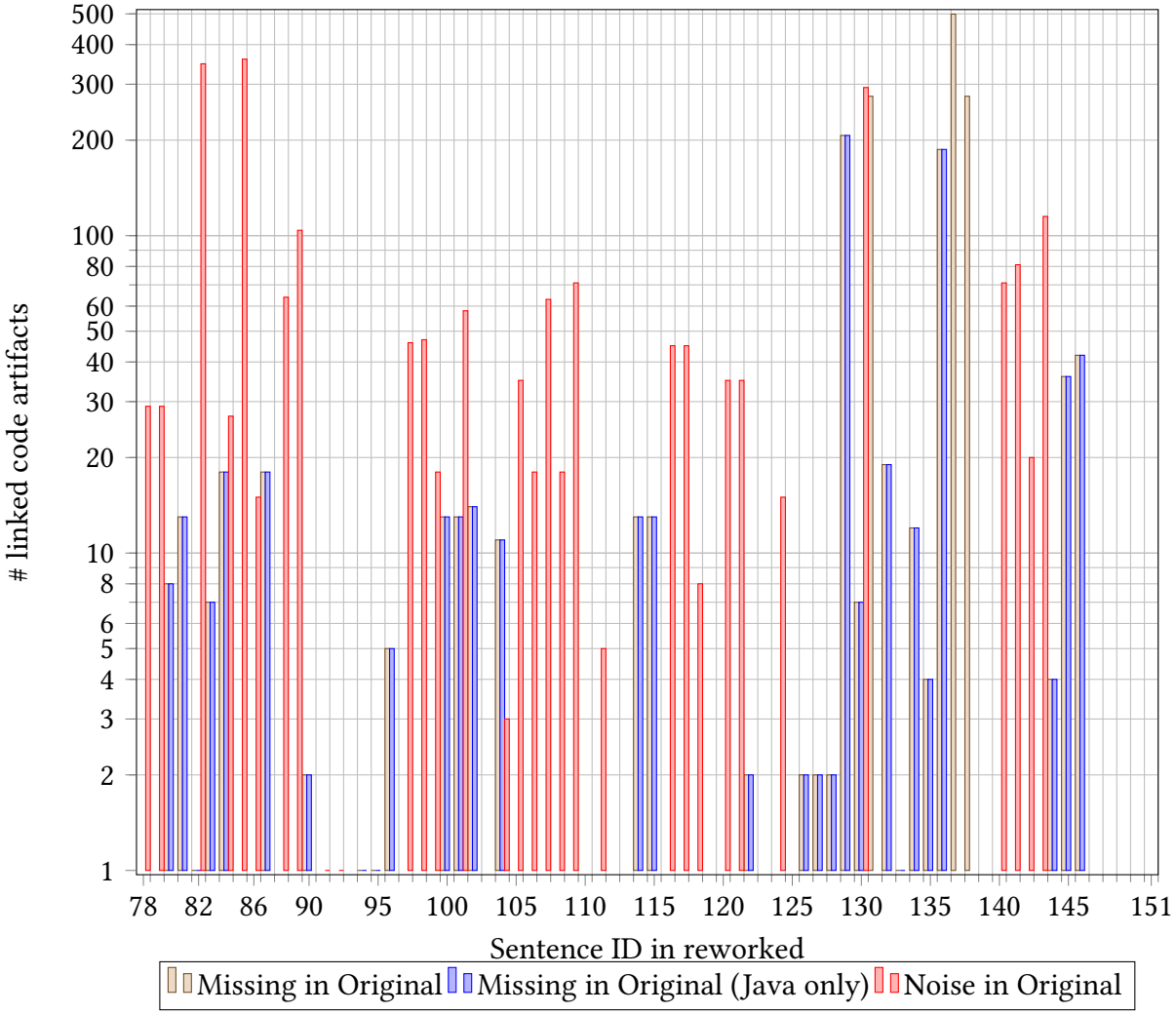

Figure A.10.: TEAMMATES original gold standard linked code artifacts difference to reworked - part 2

A.5. Prompts

prompt 1 Prompt that extracts the name of the project

Prompt

```
systemMessage = "You are to interpret a software architecture
    documentation. Identify the name of the project.
    Answer by returning this name and nothing else.";
userMessage = """
    Here is the full documentation containing all sentences:
    ```
    <<<documentation>>>
    ```
    """;
```

prompt 2 Prompt that extracts the names of relevant components

Prompt

```
systemMessage = """
    You are to interpret a software architecture documentation.
        Identify main components that are explicitly mentioned in
        the documentation.
    Ensure hereby the following:
    - They are simple names excluding the name of the project
        (`<<<project_name>>>`)
    - They exclude prefixes and suffixes that are not essential
    - They represent main components of the project
    """;
userMessage = """
    Here is the full documentation containing all sentences:
    ```
    <<<documentation>>>
    ```
    """;
```

prompt 3 Prompt that extracts ambiguities between component names

Prompt

```
systemMessage = """
  You'll be given a list of components in a software project
     that are described in the documentation.
  Your task is to identify pairs of components that share
     ambiguities.
  These might arise through:
  - similar names
  - inconsistent use in the documentation
  - being part of a named structure that does not distinguish
     between them
  - being responsible for similar things or containing similar
     software artifacts
  Other reasons for ambiguities are also possible.

  ## Output
  Return a list of ambiguity cases and provide information about
     how these ambiguities can be resolved when reading a
     sentence of the documentation.
  """;
userMessage = """
  List of components:
  <<<component_names>>>

  Full documentation:
  ```
  <<<documentation>>>
  ```
  """;
```

prompt 4 Prompt that double checks an ambiguous component name

Prompt

```
systemMessage = """
  You'll be given a sentence from a documentation that describes the components of a software project.
  Your task is to make sure whether the extracted component **truly** is expected to contain the code that is described by the sentence.
  Use information about ambiguities, that is provided as well, to justify your reasoning.

  1. Explain whether the extracted component actually is expected to contain what is described by the sentence.
  2. Then give your final decision.
  """;
userMessage = """
  The full documentation containing all sentences:
  ```
  <<<documentation>>>
  ```

  The sentence: `<<<sentence>>>`

  The extracted component: `<<<component_name>>>`

  Other extracted components for this sentence: <<<other_component_names>>>

  Additional information about ambiguities regarding this and other components:
  ```
  <<<ambiguity_information>>>
  ```
  """;
```

prompt 5 Prompt that extracts information about a component

Prompt

```
systemMessage = """
    Your task is to analyse software architecture documentation
        and collect information about components in the project.
    Each sentence of the documentation is prefixed with its
        identifier.
    """;
userMessage = """
    Full documentation:
    ```
    %s
    ```

    The component to analyze: `%s`
    """;
```