Karlsruhe Institute of Technology

Software Design and Quality

On the Impact of Context on Automated Requirements-to-Code Traceability Link Recovery

Master's Thesis of

Philip Klemens

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

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

12. May 2025 – 12. November 2025

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

On the Impact of Context on Automated Requirements-to-Code Traceability Link Recovery (Master's Thesis)

I declare that I have developed and written the enclosed thesis completely by myself. 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.

Karlsruhe, 12. November 2025

. . . . . .

(Philip Klemens)

Abstract

Requirement traceability can meaningfully support practitioners in a wide range of software engineering and maintenance tasks. However, manually maintaining traceability information is laborious and error-prone. Automated traceability link recovery techniques can address this issue. Existing approaches for requirement-to-code traceability link recovery (TLR), however, either require project-specific training or fail to produce results of sufficient quality for real-world use.

In this master's thesis, we explore the use of context in automated requirement traceability link recovery between requirements and source code artifacts. Building directly on the existing Linking-Software-System-Artifacts (LiSSA) framework and recent work in related tasks, we leverage both embeddings and large language modellarge language models (LLMs) to investigate how contextual information can enhance traceability link retrieval. Our evaluation shows that, while context can be beneficial and improve both the precision and recall of LiSSA's retrieval when applied appropriately, these improvements are modest and depend on the careful selection of strategies and preprocessing techniques.

We also implement and evaluate an agentic approach to requirement-to-code TLR. Our findings indicate that, while LLM-based agents can recover some of the trace linktrace links (TLs) present in a project, the agent's initial prompting and the tools available to it have a substantial impact on the results. Nevertheless, both the precision and recall of our agentic approach currently fall short of traditional Information Retrieval (IR)-based techniques.

Zusammenfassung

Requirement Traceability kann Anwender in vielen Software-Engineering- und Wartungsaufgaben wirksam unterstützen. Das manuelle Erstellen und Pflegen von Traceability-Informationen ist jedoch aufwendig und fehleranfällig. Dieses Problem kann durch den Einsatz automatisierter Traceability Link Recovery-Methoden adressiert werden. Bestehende Methoden erfordern jedoch entweder projektspezifische Trainingsdaten oder liefern Ergebnisse, deren Qualität nicht ausreicht, um Anwender ausreichend zu unterstützen.
In dieser Masterarbeit untersuchen wir die automatisierte Wiederherstellung von Traceability Links zwischen Anforderungen und dem Source Code eines Projekts. Dabei bauen wir direkt auf dem LiSSA-Framework und auf Forschung in verwandten Gebieten auf, indem wir Embedding-Modelle und LLMs nutzen, um zu untersuchen, wie diese es ermögichen Kontextinformationen einzusetzen, um die Qualität der wiederhergestellten Links zu verbessern. Unsere Evaluation zeigt, dass der Einsatz von Kontext zwar zu besseren Resultaten führen kann, diese Verbesserungen jedoch nur einen geringen Umfang haben und eine sorgfältige Auswahl von Vorverarbeitungsschritten und Retrieval-Strategien erfordern.
Darüber hinaus implementieren wir eine agentische Herangehensweise für TLR. Unsere Evaluation zeigt, dass ein LLM-basierter Agent zwar TLs wiederherstellen kann, dass jedoch das initiale Prompting sowie Art und Umfang der verfügbaren Werkzeuge einen großen Einfluss auf die resultierenden TLs haben. Zudem zeigt sich, dass der von uns vorgestellte agentische Ansatz derzeit noch nicht die Qualität traditioneller IR-basierter Methoden erreicht.

Contents

List of Figures

List of Tables

1. Introduction

During the creation and evolution of software, a wide range of artifacts are produced. These artifacts do not exist in isolation. Although all artifacts share the commonality of being part of the same project, for each artifact there are other artifacts that are closely related to it.
Examples of such close relationships include: the "implements" relationship between a requirement and a piece of source code; the "tests" relationship between a function and a test case; or a "refines" relationship between two requirements. Making these relationships between each pair of related artifacts explicit yields a set of edges, referred to as trace linktrace links (TLs) [12].

As software evolves and requirements change frequently over its lifetime, conceptually requirements form the origin of a software project. Consequently, in a completed software project, paths consisting of TLs can be followed from requirements, across the different artifacts making up the software, to the behavior of the realized software as verified in tests.
The ability to trace these paths and easily verify that all requirements have been implemented and verified is referred to as requirement traceability. In safety-critical domains such as healthcare, automotive, or aviation, requirement traceability is mandated by law and standards. Furthermore, the benefits of TLs are not limited to these domains. Beyond verifying whether all requirements have been implemented in the source code, TLs can help developers in a variety of software engineering tasks such as maintenance and impact analysis. Having access to TLs can be highly beneficial for software maintenance tasks [7, 33, 16]
The type of TLs we focus on in this work—requirement-to-code trace links—have an additional benefit that has only recently become relevant.
Large Language Models (LLMs) capable of generating code from natural language have enabled users with little or no programming or software engineering expertise to create non-trivial software. Users generally state the features they want implemented at a high level of abstraction, independent of language features or the elements making up the software. While they may not be aware of the concept or term, they are effectively stating requirements.
Since they lack an understanding of the code the artificial intelligence (AI) is producing for them, they cannot accurately assess whether these requirements have been implemented as specified.
This is where links generated by an integrated automated TLR technique can be helpful. Running an automated TLR approach with the user's requests and the AI's generated source code as its inputs will—assuming correct recovery—produce no TLs for unimplemented requirements and may also indicate superfluous source code.
In essence, the TLR technique can act as a verifier of correct AI behavior.

Depending on the technologies used, many TLs are either explicit or can be extracted from the project with minimal effort when needed. This is most often the case for artifacts of the same type; for instance, to be a valid Java class file, it must contain references to all types used by that class. Some links between artifacts that do not share a common type are also made explicit without any additional effort from developers. A basic example of such links is found between

UML class diagrams and the source code of the classes they visualize, where connections can be extracted by matching class identifiers in the diagram to those in the source code.

For many other pairs of artifact types, particularly those that do not share a common level of abstraction, no such straightforward TL recovery is possible. Instead, TLs need to be either manually maintained by practitioners or recovered using automated TLRs. Maintaining accurate TLs throughout the creation and evolution of software becomes increasingly laborious and error-prone as the project grows in size and complexity. This makes automated TLR a worthwhile and important research goal.

Gap One reason for insufficient performance is the inherent semantic gap between requirements and code, which has long been acknowledged as one of the key challenges in automated requirement-to-code TLR.
A second challenge is acknowledged and addressed less frequently. While requirements—and even their constituent sentences— can stand and be understood on their own to some degree, this is not the case for source code elements like class source files.
Instead, what a class "does" can be distributed across it and its superclasses, and each method may only capture a single step in a greater sequence of instructions implementing a behavior. Clearly, there are instances of artifact pairs where viewing only their content, and not their context and relationships to other artifacts, is not enough to understand their role. As a consequence, for these artifacts, determining whether they are linked by a TL cannot be made solely on this basis. Yet, this is exactly what many existing TLRs attempt to do.

Approach This begs the question of what the optimal subset is, i.e., which parts of and information about the project improve the quality of the recovered TLs and in which ways parts of the project can be aggregated into a more concise representation without losing their benefit. Our overall aim with this work is to improve the accuracy of the recovered TLs. Beyond exploring ways to narrow the semantic gap between requirements and code, we focus on whether context—defined by the existing TLs between source code elements—is one such beneficial form of additional contextual information.
We will first explore different techniques with the goal of increase the similarity of related elements, without a corresponding increase in the pairwise similarity of unrelated elements.

2. Foundations

This chapter outlines the foundations underlying this work. It begins with a brief overview of the historical roots of automated TLR techniques, followed by a discussion of how LLMs can be employed for automated TLR—an overarching question of which several aspects are explored in this thesis. In addition, we provide a general overview of how LLMs can and have been used for software engineering tasks. Finally, since our techniques were developed and implemented as part of the Linking-Software-Artifacts (LiSSA) framework, we conclude the chapter with an introduction to how it approaches recovering TLs between different types of artifacts.

2.1. Automated Traceability Link Recovery

Historically, TLR techniques have been primarily IR-based [2]. Techniques following this paradigm rely on a similarity metric defined between artifacts. Their underlying rationale is that artifacts linked by a TL exhibit a high degree of similarity, whereas unrelated artifacts exhibit low similarity. Thus, determining the set of TLs between two artifact sets (e.g., requirements and classes) can be formulated as computing the similarity between each pair of elements and then selecting those pairs whose similarity scores exceed a given threshold to constitute the recovered TLs.
In typical implementations, this involves computing, for each source element, a ranked list of target elements sorted in descending order of similarity. The final set of recovered TLs is then defined as the links obtained by pairing each source element with its top k most similar target elements.
Since the underlying metric used to compute similarity has a substantial impact on the accuracy of the recovered TLs, a wide range of similarity metrics has been explored for this purpose. Among these, one of the most widely used approaches is based on Vector Space ModelVector Space Models (VSMs). In such models, artifacts are embedded—represented as vectors in a high-dimensional vector space—where their similarity can be efficiently computed using vector distance measures.
Historically, these embeddings were sparse, with one dimension per term. Since both natural language and source code contain many terms with low information content regarding a text's content, so called stopwords, they terms are removed. To furter reduce variation, the remaining words are lemmatized before the artifacts are embedded. Which words are considered to be stopwords To address vocabulary mismatches between different artifact types, Latent Semantic Indexing (LSI) was introduced [22], building upon earlier VSM-based methods. LSI reduces the dimensionality of embedding vectors through singular value decomposition (SVD).
This process maps artifacts containing semantically related terms (e.g., "automobile" and "car") to similar dimensions. As a result, the resulting vectors are denser, and artifacts containing related terms exhibit higher similarity.

A different way to model artifact similarity is by representing their content as a combination of topics, which can then be compared using vector metrics. Latent Dirichlet Allocation (LDA) provides such a topic-modeling-based approach [30, 4].

An alternative means of obtaining dense embeddings that more accurately capture semantic similarity across artifacts—beyond matching individual terms—is the use of neural embedding models [10]. TLR has also been framed as an Machine Learning (ML) problem [36]. In this context, recovering TLs is most commonly viewed as a binary classification problem, where the model is trained to classify pairs of artifacts as either connected by a TL or unconnected. Different ML models, including random forests, support vector machines, and deep neural networks, have been applied to TLR [26].

ML-based techniques can outperform IR-based ones and unlike them are capable to modeling and interpreting the semantics of both requirements and source code artifacts [36], they are limited by their need for training data. The size of the necessary training data set can be reduced by fine-tuning pre-trained foundational models - e.g., LLMs — for the specific software project. Lin et al. show that its possible to transfer model knowledge acquired by self-supervised training on the code search task to TLR, avoiding the need for manually created TLs.
However, since ML models trained or fine-tuned on a specific project learn the project structure and do not seem to generalize well to unseen projects, evidenced by a lack of published ML TLR techniques that do not require additional training prior to being used for TLR on a new project. Much of the research on ML-based TLR focuses on network architectures and training schemes aimed at reducing the amount of training data required to effectively utilize these models. As a result, ML-based TLR is significantly less relevant to this thesis than techniques based on IR.

2.2. Large Language Models

In recent years, large language modellarge language models (LLMs) have had a tremendous impact across a wide range of research domains and practical applications. LLMs represent the state-of-the-art approach to general-purpose language understanding and generation [27]. With the proliferation of transformer-based architectures, state of the art (SOTA) LLMs now almost exclusively operate as next-token predictors. During autoregressive inference—i.e., by inputting the already generated prefix of the output back into the LLM—a complete textual response is produced. The size of LLMs, measured by the number of trainable parameters, has rapidly and substantially increased. While an increase in model size generally correlates with improved performance under established scaling laws, it also necessitates access to large amounts of high-quality training data. Consequently, training SOTA LLMs has become prohibitively expensive for most individual practitioners and smaller organizations. This challenge may be mitigated by fine-tuning a pretrained foundation model, i.e., continuing a model's training on a smaller, more relevant dataset. In many cases, however, extensive fine-tuning is not required for reasonable performance, as LLMs have been shown to generalize effectively across a broad range of natural language tasks.

Particularly relevant from a software engineering perspective is LLMs' capability to write and understand source code [29].

2.2.0.1. Retrieval-Augmented Generation

The capability of an LLM to output and reason about information is defined by its training process and training data. Thus, an LLM cannot be expected to possess knowledge that was not part of its training. However, in many applications of LLMs, access to specialized or up-to-date information is necessary. There is a clear correlation between model size—i.e., the number of trainable parameters—and performance. Even training comparatively smaller models is prohibitively expensive and time-consuming. The most lightweight way to provide new information to an LLM is to make a representation of that information part of the input. This approach is feasible when queries are specialized and the required amount of information is small.

Retrieval-Augmented Generation (RAG) [21] aims to address this problem and has found broad adoption. Instead of prepending fixed information to the input, relevant pieces of information are retrieved based on the query. This necessitates the maintenance of a database of knowledge items, most commonly consisting of snippets of preprocessed documents. In addition to the information repository, RAG also requires a retrieval algorithm to determine which elements in the database are most relevant to the query. One such approach uses the textual similarity between the query and retrieval candidates, calculated as the distance between their embeddings.

2.2.0.2. LLM Agents and Agentic AI

Creating software that emulates human-like reasoning and can handle a variety of situations and tasks has long been a focus of research [37]. Agentic AI has emerged as a promising approach to achieving this goal. Unlike in a Question-Answering (QA) setting—where the user or application formulates a question or task and then receives a direct answer from the LLM—in an agentic context, the outputs are largely hidden from the user. Instead of a simple exchange of messages, in the most straightforward implementation of agentic AI, the LLM participates in a message exchange with the application. This typically begins with a description of its task, an initial state including a user-specified goal, and a set of functions available to it. The agent is then expected to plan a solution to the task and begin executing it by calling functions via specific output tokens. The application provides tool results and updates on state changes resulting from their use or from the passage of time. The functions available to the agent depend on the task and may also change as the agent progresses toward a solution.
A key advantage of this approach over traditional QA-based interaction is that it does not require the user to explicitly provide contextual information for each query, nor do they need to know how to solve the task themselves. There are various definitions of what constitutes an AI agent or agentic AI [5]. In the context of this work, we consider any use of an LLM that is able to call functions and determine when to output its final message or answer as an instance of agentic AI.

2.3. LiSSA

Fuchß et al. propose the Linking-Software-System-Artifact (LiSSA) [10] as a generic framework solving the TLR task by leveraging a LLM as a zero-shot classifier for TLs. The framework (see

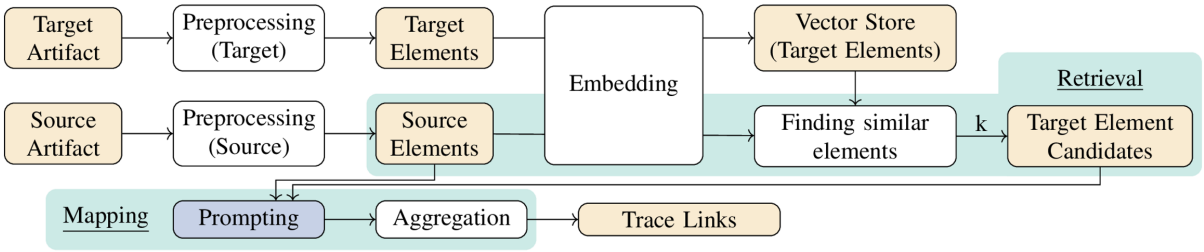


Figure 2.1.: Overview of LiSSA by Fuchß et al. [10] from their paper, showing the different modular components of the framework (white and blue), inputs, final as well as intermediate results

Figure 2.1) can be divided into two stages: The first is retrieval, designed to retrieve relevant target artifacts and thereby decrease the number of source-target pairs passed on to the second major component. This component is implemented using RAG. After initial preprocessing, an embedding is generated for each of the project's source and target elements and stored in a vector store. For a given source element whose outgoing TLs are to be recovered, the cosine similarity between its embedding and those of the target elements is evaluated. The target elements with the top-k most similar embeddings are passed on to the next core component: mapping. The mapping consists of prompting the LLM with each pair—consisting of the source element and one of the candidate target elements—and tasking it with determining whether a TL exists between them. Fuchß et al. present different interchangeable preprocessing modules, as well as two different LLM prompts: KISS and chain-of-thought. In the course of evaluating the different variations of the framework on the requirement-to-code TLR task, the authors determine that, when evaluated across the entire test dataset, LiSSA outperforms VSM-based TLR techniques. Among the different preprocessing modules, providing the entire artifact to the LLM yields the highest F_2 score for the combined classifier, indicating that unlike IR-based methods for TL recovery, LLMs are capable of utilizing context of both requirements and methods for a more accurate TL recovery.

2.4. Metrics

Fundamentally any fully automated TLR technique is equivalent to a binary classifier, classifying potential TLs, i.e. tuples of artifacts either as positive if they are linked by a TL or as negative if this is not the case. As any binary classifier, we can evaluate the performance on a given dataset using the two main metrics of precision (P) and recall (R). As shown in Equation 2.1, precision is defined as the number of true positives divided by the total number of inputs classified as positives, while recall is the number of true positives divided by the the sum of true positives and false negatives. In other words, precision is the fraction of correctly classified positives and recall is the fraction of true positives in the gold standard or ground truth the classifer has correctly classified.

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

Generally, precision and recall represent a trade off. To be able to reduce classifier performance to a single scalar and enable comparison with other classifers, the harmonic mean of recall and precision is used. This metric, defined by Equation 2.2 is referred to as the F_1-Score.

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R} \tag{2.2}$$

In many applications, its more important not to miss true positives, then it is to reduce the number of false positives, i.e. recall is valued more highly than precision. In these cases the F_2-Score, a mean weighted to put a greater emphasis towards the recall value is used. This is highly applicable to automated TLR, since a user is much more likely to notice a erroneous TL in the output, than they are to manually recover TLs from the potentially very large set of combinations of requirements and classes.

$$F_2 = (1 + 2^2) \cdot \frac{P \cdot R}{(2^2 \cdot P) + R} = 5 \cdot \frac{P \cdot R}{4P + R} \tag{2.3}$$

3. Related Work

Closely related to the subject of this thesis is prior work in three categories. The first consists of studies that share our goal of improving the recovery of TLs between requirements and code, with particular focus on those that share some aspects of our approach. The second category includes works dealing with similar tasks, most prominently those presenting automated issue-to-commit TLR techniques.

Increasing Similarity A core challenge of any IR-based TLR aiming to recover TLs between natural language and source code artifacts is the *semantic gap*. This gap becomes evident in the low similarity between requirements and related classes [13]. How a core IR method and the measures of textual similarity or relatedness it provides can best be employed in an overall algorithm is a central question in much of the recent work on TLR [3].

Gao et al. [11] determine co-occurring terms in both requirements and source code, which they call *consensual biterms*, and insert them into the artifact representations before passing them into the core IR method, of which they explore several variants. The need to go beyond pairwise textual similarity of the unprocessed artifacts has long been acknowledged and has driven the development of different preprocessing strategies. Additionally, multiple metrics and probabilities can be combined. Moran et al. [28] combine several textual similarity metrics, known transitive TLs, and the similarity assessments of developers into a single probabilistic model.

Structural Information Structural information can be useful in recovering TLs between requirements and code, but it may also risk the propagation of false links, leading to a greater number of false positives TLs being recovered [31].
In addition to enriching artifacts with consensual biterms, Gao et al. extend **T**raceability **R**ecovery by b**i**Iterm-enh**A**nced **D**eduction of transitive links (TRIAD) and **T**race**A**bility **R**ecovery by c**O**sensual bi**T**erms (TAROT) by leveraging intermediate artifacts, i.e., artifacts related to both potential source and target elements. Unlike TAROT, biterms are considered consensual if they occur in intermediate artifacts and either the source or the target element. They define their final similarity metric as a linear combination of the similarity between the source and target artifacts and the similarity of the paths linking them across intermediate artifacts.
Kuang et al. [18] use structural information by employing a precomputed graph modeling both method-level data dependencies and call dependencies to expand an initial set of true TLs. A different perspective on using structural information involves exploiting the structure of trace artifacts to subdivide them into sets of elements over whose union TLs are then recovered. While this further increases the search space of potential TLs, it can allow for more precise TLR.
Hey et al. present this approach in Fine-grained Traceability Link Recovery (FTLR) [14]. After recovering links between requirement sentences and class methods, their technique determines

whether a requirement and class are related based on whether their sentences are among those most "voted" for by the methods. Since not every method may be linked to a requirement, two thresholds are used to filter the initially determined sentence–method links, avoiding erroneous recovery of links between dissimilar artifacts.
While TLR is not the primary focus of Jin et al.'s approach to generating user-level requirements for software project repositories, requirement-to-code trace links are recovered as a byproduct of the hierarchical structure established during the execution of USERTRACE [17]. To achieve this, the authors employ a set of specialized AI agents with clearly defined roles that collaboratively process and refine artifacts across multiple abstraction levels in a bottom up order of execution.

LLMs Traceability Link Recovery (TLR) involving natural language artifacts inherently relies on techniques capable of analyzing natural language. With large LLMs defining the current state of the art, developments and trends surrounding LLMs are highly relevant to this field [34]. In fact, LLMs may help bridge the gap between traditional, lightweight TLR approaches and the accuracy achieved by less accessible, resource-intensive ML-based techniques. This is one of the core concepts of LiSSA, as outlined in Section 2.3.
When recovering links between requirements, Fuchß et al. [9] also explore foregoing traditional similarity-based retrieval entirely. Instead, they employ an ensemble of different LLMs to filter the large search space of potential TLs, using increasingly larger and therefore more capable—but also more computationally expensive—models.
Requirement-to-code TLRs are closely related to other TLR tasks that process both natural language and source code artifacts. One of these closely related tasks is issue–commit linking, i.e., recovering TLs between commits and issues in a software project. Of the two, we consider requirement-to-code TLR to be the generally more difficult task. First, the natural language in commits and issues is usually expressed at a similar level of abstraction and is aimed at developers. Second, issues and commits have associated process information that automated TLR techniques can exploit [32], such as creation dates, authors, and participants in associated discussions [1].

Due to the advantages provided by issue–commit traceability and the greater availability of training and evaluation data, this field has recently garnered substantial research interest, including both ML- and non-ML-based techniques. Lin et al. present a family of fine-tuned BERT models to classify pairs of issues and commits based on whether they are linked by a TL. They address data scarcity by outlining a training regime in which their models are first pre-trained on code search—i.e., recovering links between functions and their associated comments—on unrelated datasets, then trained on smaller amounts of project-specific data, and finally fine-tuned on both the target task and project using a small number of manually created issue–commit TLs.

Huang et al.'s approach to issue–commit TLR closely resembles that of LiSSA. The large search space of potentially linked issues and commits is first filtered using a top-k selection based on pairwise embedding similarity. Unlike LiSSA, in which each candidate pair is classified, they task an LLM with reordering the list of candidates based on commit messages for each issue represented by its title and description. Using a smaller k, the top elements of this re-ranked list are then returned as the final output of the technique.
Akhavan et al. present a different approach to issue–commit TLR, also using an LLM, but in an autonomous agent implementation [1]. Instead of presenting an LLM or classifier with issue–commit pairs to classify, their agent receives an issue, a link to the project repository,

and a set of 20 functions that it can use to explore the repository, including functions to access source code and process information. It then autonomously performs the linking task and, according to their evaluation, exceeds previous issue–commit TLR techniques in Hit@1 accuracy, i.e., the rate at which the correct commit is identified as the top candidate, by between 60% and 262%.

4. Analysis

In this chapter we will go into more detail on the core challenges of automated requirement-to-code TLR using some of the most commonly used evaluation datasets for TLR as an example. Based on these observations we will present or approach to improvide the results produced by the LiSSA framework, particularly with respect to its retrieval stage.

As mentioned in chapter 1, manually creating and maintaining requirement-to-code traceability information, i.e., TLs, is a worthwhile but laborious task. In domains where this process is mandated by law, projects are typically proprietary. Consequently, there is a shortage of publicly available, large, and recently created datasets consisting of both requirements, source code, and TLs.

For our analysis as well as the evaluation of our techniques, we use a subset of the Center of Excellence for Software Traceability (CoEST) dataset, consisting of the ETour, iTrust, and SMOS projects. We also use the Dronology dataset, which includes the project's source code as well as two sets of requirements at different levels of abstraction and their corresponding TLs. These datasets have been widely adopted and used for the evaluation of various TLR techniques, thus allowing us to compare our results with those achieved by prior approaches. Table 4.1 shows the number of requirements, classes, and TLs between them for each of the datasets.

Notably, while all of the listed datasets are written in Java and follow an Object-oriented programming (OOP) paradigm, they contain substantially different numbers of requirements, classes, and links.
It is, however, important to keep in mind that these datasets represent only a limited sample and should not be considered representative of the breadth of real-world software projects. Firstly, they are all relatively small, with between 99 (SMOS) and 432 (Dronology) classes. But they may still make up a meaningful range of inputs. Despite their limited sizes, they cover a

Table 4.1.: Number and information regarding the of requirements, classes and gold-standard TLs between them for a subsets of the CoEST dataset and Dronology

Dataset	Links	Requirements	Classes	with JavaDoc	Methods	with JavaDoc
Dronology-DD	722	211	432	41.4%	2428	12.1%
Dronology-RE	587	99				
ETour_en	308	58	118	77.1%	1025	69.8%
iTrust	286	131	219	90.8%	1510	44.1%
SMOS	1044	67	99	63.6%	456	91.2%

Table 4.2.: Overview of the some of the datasets commonly used to evaluate requirement-to-code TLR approaches.

Table 4.3.: Evaluation results of previous IR-based TLR techniques presented by Fuchß et al. [10] and the ML-based TRAIL [26]

Approach	SMOS		ETour		iTrust		Dronology (RE)		Dronology (DD)	
	P	R	P	R	P	R	P	R	P	R
VSM_{OPT}	.430	.414	.557	.427	.208	.227	**.844**	.087	**.846**	.071
LSI_{OPT}	.415	.430	.452	.453	.251	.255	.333	.107	.757	.074
$COMET_{OPT}$	.195	**.572**	.410	.468	.361	.231	—	—	—	—
FTLR	.444	.331	.379	.633	.165	.339	.183	.161	.129	.154
$FTLR_{OPT}$	.314	.588	.505	.597	.234	.241	.184	.170	.140	.147
$LiSSA_{retrieval}$	.325	.418	.216	**.815**	.058	.531	.128	**.420**	.085	**.482**
LiSSA	.590	.105	.409	.734	199	.451	.226	.344	.177	.380
TRAIL	**.871**	**.735**	**.572**	.650	**.568**	**.658**	—	—	—	—

variety of domains, including tourism (eTour), healthcare (iTrust), and cyber-physical systems (Dronology). Their differences in how requirements are expressed and in how densely their source code is annotated with JavaDoc comments may be more impactful for TLR.
In the two sets of requirements for Dronology, each requirement consists of a short title and a single-sentence description of the desired software behavior.
In ETour, each requirement artifact is a full use case, containing a description, optional participating actors and entry conditions, the flow of events, exit conditions, and quality requirements.
In the iTrust dataset, the use cases have been decomposed into their individual steps and exit conditions; however, these individual requirements are inconsistent in their length and level of detail with which they describe aspects of the software.
Requirements for SMOS follow a similar format to ETour, containing full use cases with a title, description, participating actors, individual steps, and a post condition. However, unlike the other projects, SMOS requirements and comments are entirely written in Italian.
As shown in Table 4.1, the projects also differ in the amount and type of JavaDoc comments they contain. Only 41.4% of Dronology's methods have attached JavaDoc comments, while this is the case for 90.8% of iTrust's methods. The percentage of classes with attached JavaDoc comments varies even more, with only 12.1% in Dronology compared to 91.2% in SMOS.

SOTA Results Due to their relatively small size, high number of comments, and creation in an academic context, it seems unlikely that the chosen datasets represent a particularly difficult input for TLR techniques. At the same time, based on the results achieved by previous TLR techniques, they cannot be considered a particularly easy input either.
When comparing the precision and recall reported by different IR-based TLR techniques, as curated by Fuchß et al., with those achieved by the ML-based **TRA**ceability l**I**nk c**L**assifier (TRAIL) shown in Table 4.3, a substantial performance gap between SOTA IR- and ML-based TLR techniques becomes apparent. This observation also holds true for other techniques [36].

For SMOS, eTour, and iTrust, TRAIL substantially outperforms the respective optimal IR-based technique. While Moran et al. do not evaluate Hierarchi**C**al Pr**O**babilistic **M**odel for Softwar**E** **T**raceability (COMET)'s performance on the Dronology dataset, the reported precision and recall for the remaining three projects make a substantially better performance of COMET on

Dronology seem unlikely. Irrespective of TRAIL's performance on Dronology, which has also not been evaluated, the performance of all listed SOTA IR-based requirement-to-code TLR techniques remains insufficient for beneficial use by real-world practitioners.
Nevertheless, despite TRAIL's superior performance, even if we extrapolate from these results on a limited set of Java datasets to the full breadth of software projects this does not imply that IR-based methods have become obsolete.
A key barrier to the adoption of ML-based techniques is their reliance on training data—specifically, existing TLs within the project to which the technique is to be applied. Creating these initial TLs manually still requires the same labor-intensive TLR process described earlier and demands that practitioners produce a high-quality set of training links. Such a set should capture both "easy" and "hard" links and maintain a balanced distribution across different types of traceability relations.
Existing ML-based TLR techniques have also been shown to generalize poorly to unseen projects, requiring retraining on project-specific data as well as retraining to reflect project evolution.
In contrast, IR-based TLR techniques present a substantially lower barrier to adoption, as they require neither training nor preexisting data and typically involve only the selection of a few hyperparameters. While many IR-based techniques use representation learning models that have been previously trained on large text corpora [25, 36], this does not impact the use of these TLR techniques by practitioners.
For these reasons, this thesis focuses on improving the performance of IR-based techniques that do not require prior training. Specifically, we aim to improve the results produced by LiSSA, targeting both the retrieval and classification stage of the framework.

Analysis At the core of IR-based TLR techniques lies a similarity metric. In LiSSA, this role is fulfilled by the cosine similarity between the neural embeddings of the source and target elements' contents. As previously stated and shown in Table 4.3, the quality of LiSSA's results is not sufficient for real-world use. Before presenting our general approach to improving its performance, it may be beneficial to analyze why this is the case.
As outlined in Section 2.3, LiSSA's retrieval produces a set of k candidate target elements for each source element. Figure 4.1 shows the distribution of pairwise similarities for both related (solid) and unrelated (dotted) pairs of artifacts in each of the evaluation datasets.
While the distributions have been normalized in the figure, it is important to keep in mind that the number of unrelated pairs of elements is substantially higher than that of related pairs (see Table 4.1). The first obvious observation is that none of the pairs in any of the datasets exhibit a similarity greater than 0.8, while the overall similarities for both related and unrelated artifacts tend toward the lower end of the remaining range.
Secondly, while the core assumption of IR-based TLR holds true—namely, that artifacts linked by a TL exhibit higher similarity on average than those not linked—there is substantial overlap between the two distributions for each dataset. This directly implies both the strong trade-off between precision and recall and the resulting low aggregated F_1 and F_2 scores.
There is neither an appropriate threshold for filtering potential TLs based on their similarity, nor a value of k that retrieves nearly all true positive links without also introducing a much larger fraction of false positives. For low values of k, the retrieval stage will generally recover those TLs whose source and target elements exhibit high similarity. This set will typically consist largely of true positive TLs, yielding high precision. However, since many true positive links connect source and target elements that are not highly similar, they will not be retrieved, resulting in low recall.

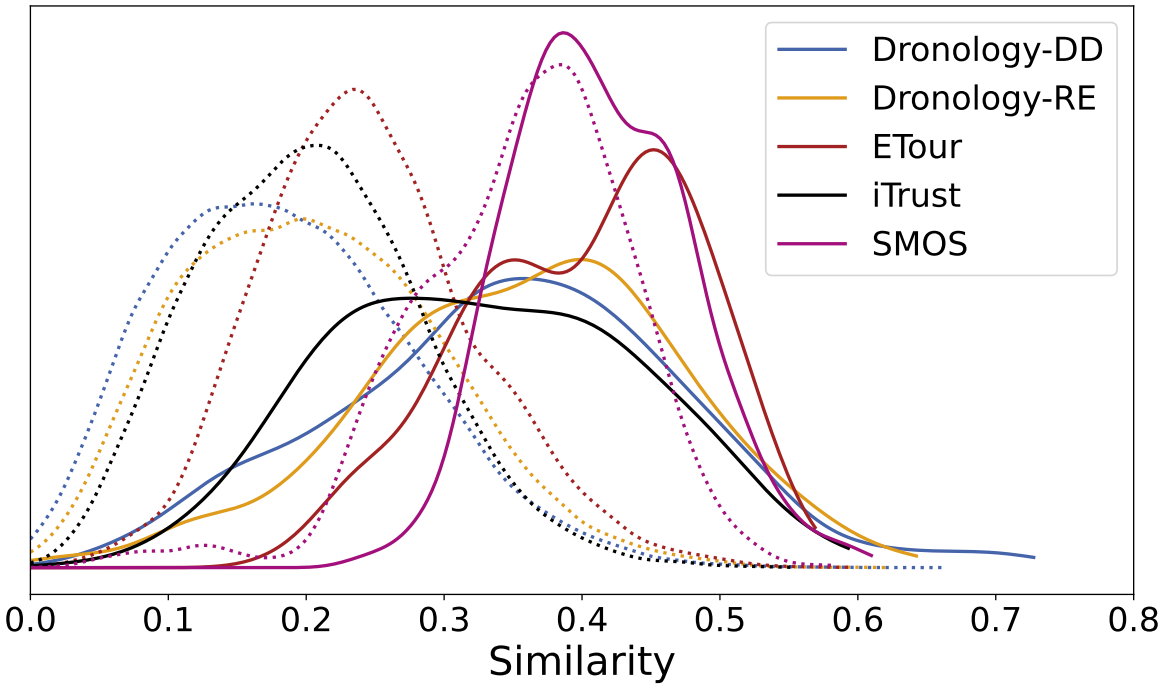


Figure 4.1.: Smoothed distribution of the cosine similarities of requirements and classes linked by a TL (solid) and those not linked by a TL (dotted)

Conversely, for high k values, even true positive links with low similarity will be retrieved. But because the number of unrelated target elements is much greater than that of related classes, comparable similarity values lead to a substantially greater percentage of false positives than true positives in the result.

Approach Conceptually, our approach is based on the assumption that textual similarity between potential source and target elements provides a meaningful but, on its own, insufficient heuristic to accurately capture traceability relationships between requirements and code [38]. Our hypothesis is that at least some of the reasons these techniques fail to produce sufficiently accurate results is that textural similarity of artifact pairs viewed in isolation can not capture a substantial amount of the existing TLs.
The core idea of our approach to improving these results is to leverage the context of source code elements as enabled by the use of LLMs and SOTA embedding models.
In particular, we focus on two distinct forms of source code context. The first is defined by the relationships between classes. In the OOP paradigm, both logic and state are distributed among objects, which are defined by classes. Thus, there is rarely a one-to-one relationship between requirements and code elements. Especially for requirements stated at a high level of abstraction, a single requirement may be linked to a large set of classes, not all of which exhibit high textual similarity to the requirement.
However, while some of these classes may be textually dissimilar, they are likely to use—or be used by—other classes related to the same requirement. Consequently, by taking into account closely related classes, the similarity score they receive during retrieval can be increased, potentially leading to a higher number of true positives in the final set of TLs.
At the same time, this approach can also reduce the number of false positives. Figure 4.2 shows a simple constructed example containing one requirement and four classes. In addition to two related classes, both exhibiting high similarity to the requirement, there are two unrelated

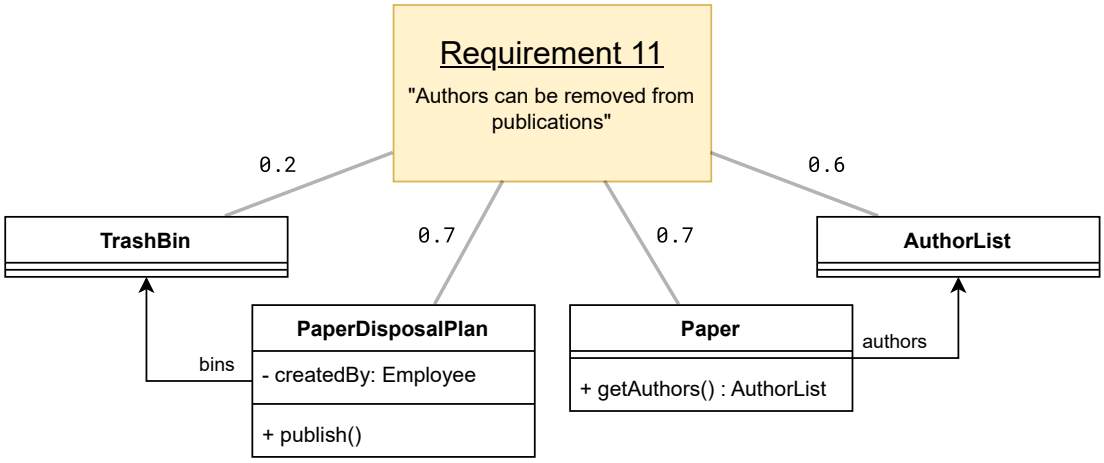


Figure 4.2.: A minimal example illustrating how the context of class artifacts can influence the determination of their relationship to a requirement. The connecting lines between the requirement and the classes are annotated with their respective textual similarity scores.

classes, one of which may result in a false positive TL. While some unrelated classes may be outliers and show high textual similarity with a given requirement due to matching terms in their identifiers (e.g., `PaperDisposalPlan`), these classes are unlikely to be closely related to other classes with comparably high similarity. Thus, incorporating contextual information can help avoid such false positives. The specific mechanisms by which this is achieved will be presented in the next chapter.

The second aspect we are going to explore is shrinking the amount of context included. Hey et al. have shown that performing TLR on more fine-grained elements can be worthwhile and that taking into account the removed context can further benefit the accuracy of the recovered TLs.
However, they were substantially limited in the form and amount of context that can be included using the Word Mover's Distance (WMD) [19] as their underlying similarity metric. Both neural embedding models and LLMs enable new and more extensive ways of leveraging context. Specifically, embedding models like those used by LiSSA are more tolerant of stopwords, non-lemmatized terms, and unlike the historical bag-of-words represenation can take into account of the semantics of both natural language and source code.

Beyond these ways of leverage context to modify similarities, we also explore different avenues to increase the similarity of related elements overall without also increasing the similarity of unrelated pairs to the same degree. All three of these aspects of the thesis will be presented and evaluated in the the following chapter.

To leverage context as part of TLR approach, context needs to be made accessible to the technique. As described in 2.3, LiSSA is designed to be as generic as possible to be able to recover TLs between elements of any type. The downside of this approach is that the context of and relationships between elements are not part of LiSSA's element objects. This means we need to create a context model separate from the elements being processed.
Since different modules may rely on different forms of context, this approach has the additional benefit of allowing different forms of context models to be used depending on the modules included in a given configuration. E.g. if the only context required by the pipeline is the inheritance hierarchy of the class elements, collect a potentially very large set of method relationships from the input project's source code can be avoided.

4.0.0.1. The Code Context Model

Relationships i.e. trace links between source code elements can be derived directly from the project's source code. No additional documentation of these links is required.

We parse the entirety of the project's source code into a graph containing three different types of nodes: Packages, classes and methods. The model does not differentiate between concrete classes, abstract classes, interfaces or enums. Each node is uniquely identified by the fully qualified name of the code element it represents. Being able to unambiguously map elements to nodes is necessary to be able to access their context inside LiSSA's modules.

Each of the nodes is constructed on the basis of the code's AST, but we limit their content to aspects relevant for TLR. Both class and method nodes contain their raw source code, i.e. the content of the class file and the code snippet consisting of the method signature and body, respectively.

Our model also contains what we consider to be the two most important types of trace link between source code elements: The edges of the inheritance hierarchy graph and the method's caller-callee relationships. The former connects each class node with the class nodes it extends or implements, or is extended or implemented by. The latter connect each method node with methods called in the method it represents and vice versa.

5. Retrieval

At this stage, two types of objects are relevant. Artifacts are the input to the framework and the entities between which trace links (TLs) are to be recovered. LiSSA supports a very wide range of such artifacts. For automated requirement-to-code traceability link recovery (TLR), in most cases (including in our evaluations), artifacts are the project's individual requirements on one side and its classes on the other. After they have been created, LiSSA's first step is to process the input artifacts into elements that will be used by the remaining modules.

5.1. Preprocessing

Automated TLR techniques usually aim to recover TLs between requirements and classes.

In LiSSA preprocessors are unrestricted in how many elements they can produce for each artifact they receive as an input. While they can take advantage of this to produce more than one representation per class, this ability is intended to subdivide the artifact into its components. For classes, the most straightforward subcomponents a class can be subdivided into are its methods. Since TLR on the method level has already been previously explored and been shown to yield good results, we implement preprocessors targetting each of these levels and dedicate the following two subsections to their introductions.

5.1.1. Classes

Viewing the retrieval stage as a black-box classifier whose results will be further processed can be helpful for understanding its importance. Conceptually, requirement–code TLs can exist between any pair of requirements and classes, leading to a large search space. For efficient IR methods, this is not an unfeasibly large input size. LLM inference is much more computationally expensive, thus if an LLM is to classify each candidate TL, a preselection step is necessary. In LiSSA, determining this preselection is the role of the retrieval. Its approach is equivalent to iterating over each potential TL and classifying it as positive if the target element is among the k most similar elements to the source.

Forgoing the possibility of using a different embedding model, there are two ways to alter the similarity score of two elements. The first is by modifying the preprocessing step to transform the string before it is embedded. LiSSA already implements this to a limited extent through its preprocessors. The second is by modifying the retrieval itself—specifically, how the similarity between two elements is calculated. By default, LiSSA uses the cosine distance between the query and candidate embeddings, but conceptually, any function that produces a score (and thus a ranking of candidates) can be used. In both of these approaches, code context can be leveraged.

Focusing on the preprocessing step first, before we discuss concrete preprocessor implementations, it is important to keep in mind the module's purpose. Ideally, given a requirement and a source code artifact (e.g., a Java class) linked by a TL, the code artifact should be transformed into a string whose embedding is virtually identical to that of the requirement, maximizing their similarity. Besides restating the content in a different way, this may also include selecting the most relevant parts of the artifact and eliminating noisy or misleading parts. In practice, achieving a perfect similarity score is not possible. Instead, we aim for the embedding of the source code artifact to be as similar to the requirement's embedding as possible. At the same time, the transformation should not increase the similarity between the class and unrelated requirements—or at least to a lesser degree. If this is not the case, unrelated requirements may "overtake" related ones in the similarity ranking established during retrieval, and if the change is sufficiently large, thereby decrease the number of true positives while increasing the number of false positives the retrieval produces.

One approach to finding such a transformation is to examine what separates the two types of artifacts. The most obvious difference is the semantic gap between them. Even if each requirement of a software project were linked to exactly one class in its source code, the two artifacts would still express the same information in very different ways, and thus there can be no guarantee that their embeddings will be similar.

LLMClassSummaryPreprocessor Transforming either the requirement or the class to bridge this gap could address the issue. Of the two options—transforming the requirement into a valid Java class or transforming the class into a natural-language expression—we chose the latter. We cannot assume that a given requirement is implemented in a single class, so deriving a class from it when it spans multiple classes would likely yield a result dissimilar to the real target class, leading to low similarity and poor retrieval accuracy. While we can make the same argument for the common case of a class implementing multiple requirements, we assume that the embedding model is more tolerant of additional information in a text than it would be of a potentially unrelated Java class. Since LiSSA already provides a framework for utilizing LLMs, and because LLMs have demonstrated strong capabilities in explaining and summarizing code in various programming languages, we use an LLM for our first preprocessor, which implements the transformation from source code to natural language. Given a source code artifact, the `LLMClassSummaryPreprocessor` prompts an LLM to generate a natural-language summary consisting of a configurable number of sentences. As described, this summary is then embedded.

LLMClassSummaryPreprocessor Prompt

Summarize the following software artifact using up to **number_of_sentences** sentences:
artifact_content

JavaDocEnhancerPreprocessor For very fine-grained requirements—such as those prescribing the use of specific hard-coded values—there is a risk that the LLM may omit references to the corresponding parts of the source code. We could address this by increasing the level of detail and length of the summary via the prompt. However, greatly increasing the summary length may also introduce irrelevant information (i.e., noise) and result in a "watered-down" embedding. Presumably, the developer writing the classes, methods, and associated Javadoc is aware of the requirements the class is intended to implement and has written them accordingly.

Thus using the set Javadoc comments in a class as its representation may be more accurate than including the source code instructions. A clear downside of this preprocessor is its reliance on developers to write sufficiently detailed Javadoc and maintain it as the software evolves. If there is no Javadoc associated with a class, or if the documentation is incongruous with the code, this preprocessor will output a representation that might be semantically closer to the related requirements but contain completely different information. This could lead to performance falling below that of naïve retrieval using the full source code.

Still, leveraging documentation may be a way to achieve accurate retrieval. We can task an LLM with assessing existing Javadoc, improving it where it does not accurately describe the class and its methods, or generating Javadoc where it is missing or incomplete. Besides cases where Javadoc is missing, incomplete, or inaccurate, we can also expect this preprocessor to handle input where source code is documented in languages the embedding model has not been sufficiently trained on by enforcing English-language documentation.

JavaDocEnhancerPreprocessor Prompt

Add proper English-language Javadoc comments to the following Java class or interface.
Make sure to include:
Class-level Javadoc describing its purpose.
Method-level Javadoc describing parameters, return values, and exceptions (if any).
Field-level Javadoc where useful.
Do not restate your task. Only return the complete class with Javadoc comments added, preserving formatting. If the class already has English-language Javadoc comments, improve them where you see fit.
Class source: **artifact_content**

ClassSkeletonPreprocessor A second role preprocessors can fulfill in LiSSA is to remove noise from the artifact contents. Unlike natural language, source code always contains words and phrases unrelated to the purpose of the class. The most obvious example are programming language–specific keywords. Variable names are only as descriptive as the developer has made them to be. Particularly for local variables in methods, names are often chosen with little effort. The use of unrelated terms in the type names of variables that are frequently repeated in one class or method may also water down the specificity of the embedding.

For a given class, foregoing documentation, the most informative and usually most carefully named elements are its methods—or, to be more precise, its method signatures. Consequently, reducing methods to their signatures has been an established preprocessing step in previous IR-based TLRs techniques, particularly in FTLR, where method signatures are the code element for which TLs are recovered. Combining the set of signatures, excluding the parameter variables' names, with the class name gives us a string summarizing key properties of the class:
its name, the operations that can be invoked on it and its instances, and what we consider the most important types it interacts with in return and parameter types. We implement this transformation in the `ClassSkeletonPreprocessor`.

5.1.2. Methods

So far, we have only discussed preprocessors that generate exactly one element for each class in the project's source code. But this may not be the optimal level of granularity for TLR. In Object-Oriented Programming (OOP), a class defines a template from which objects are created. Each object represents and manages a part of the software's state and the operations performed on that state. A software system's behavior—how it reacts to user inputs—is generally not defined in a single class but emerges from the interaction of many objects. This interaction consists of objects invoking methods on other objects, forming a tree of method calls. Each node in this tree should do exactly one thing [23], i.e., each method can be described in natural language. Since these methods collectively define the system's behavior as specified in the project's requirements, this may be a better approach to recovering TLR between software artifacts.

We can reuse some of the same transformations we have already defined for class artifacts:

`MethodJavaDocPreprocessor` The `MethodJavaDocPreprocessor` returns a set of Elements each containing the method JavaDoc for one of the class artifact's methds.

`ToMethodContextPreprocessor` The source code making up a method is by definition shorter than a full class, generally significantly so. At the same time, by splitting a class into a set of sub elements, each generated in isolation from the others, we strip away some of its context. While this can be benefical in cases where a class declares multiple unrelated methods, e.g. methods that are part of two entirely distinct use cases, this context can also be important, most obviosuly in cases where one of the class's methods calls the other.
To account for both of these cases, we implement a preprocessor that will append the signatures - which we assume to be meaningful [15] - of all caller and callees methods to that of the target method.

`ToMethodsPreprocessor` Finally, since we assume that a `CodeContextModel` instance is available during preprocessing, we also reimplement a more concise and efficient version of LiSSA's `CodeMethodPreprocessor`. This implementation limits its preprocessing to resolving the class to a class node in the model and mapping the source code making up each of its methods to an element.

5.1.3. Implementation

Each preprocessor is implemented as a class extending LiSSA's abstract `Preprocessor` class, allowing each to be instantiated and used in place of an existing preprocessor implementation. As described above, all preprocessors expect artifacts corresponding to classes as their inputs. They output either one element per input artifact (for class-level preprocessors) or one element per method (for method-level preprocessors).
Preprocessors that take advantage of the source code structure and relationships between elements rely on the `CodeContextModel` obtained from the context store.
Preprocessors using an LLM all share a common superclass, `LLMBasedPreprocessor`, which encapsulates the logic required to instantiate the wrapper of the actual LLM API and cache its responses. The subclasses—i.e., the concrete preprocessors—interact with the model only

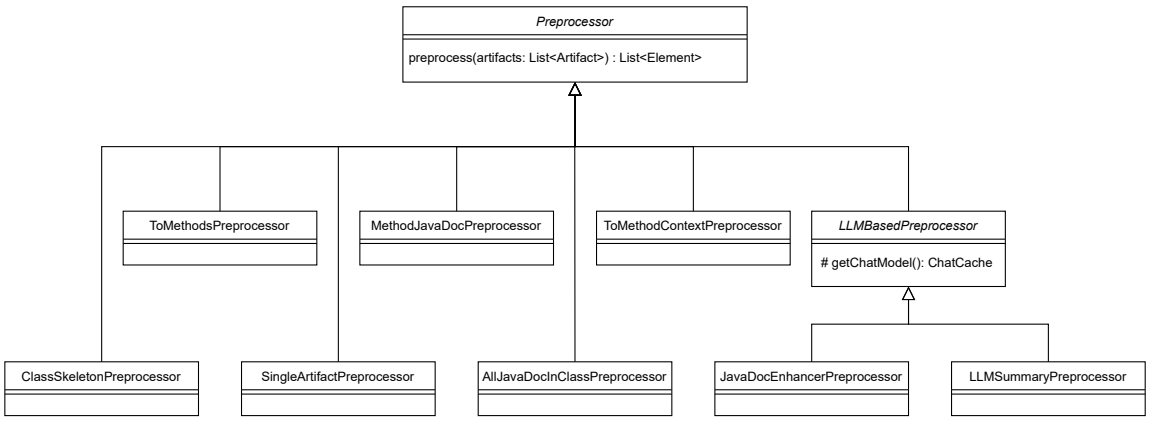


Figure 5.1.: An overview of the preprocessors we added to LiSSA as part of this work, also including the reimplemented `ToMethodPreprocessor` and the baseline `SingleArtifactPreprocessor`

through the interface provided by their parent's protected `getChatModel()` method.
Any preprocessors that rely on the source code element they are expected to produce a representation for having attached JavaDoc comments will default to returning the full underlying source code to avoid them producing empty strings, whose embedding will be a strictly less accurate representation than that of the source code. Preprocessors relying on classes having methods they can be subdivided into follow the same pattern of behavior, returning a single Element containing the full source for classes that do not declare methods.

5.2. Retrieval Strategies

So far, we have defined a set of preprocessors and how they transform the project's classes into representations that can be embedded, but we have not yet presented an approach for improving the retrieval itself using context information. This section introduces that.

In the LiSSA framework, a retrieval strategy defines how and which elements are retrieved. Its default retrieval strategy computes the cosine similarity between each element in the element store and the current query element, then retrieves those with the k highest similarities. Using this strategy, the context of the source code elements influences retrieval only insofar as that context has influenced the element's string representation before embedding.

However, we can also leverage the context in which code elements exist in the source code as part of the similarity calculation during retrieval.
Instead of using the pairwise similarity of a query and just a single given candidate element, we could take into account the similarity of the candidate's context and the query.
In our implementation, this means calculating the pairwise similarity between embeddings of both the candidate element and the elements in its neighborhood.
In addition to the similarity of the query and target element (as with the preexisting retrieval strategy), this yields two sets of similarity scores.
Since we need a single value to use as the overall similarity score of source and target, we first aggregate the two lists, then take a convex combination of the three remaining values as the total score. Conceptually, any function mapping a list of scalars to a single scalar and any set of coefficients is possible. In practice, we take the maximum and average of the sets as

the aggregation function to arrive at a single scalar. We normalize the three coefficients used to combine the aggregated weight of callers and callees with the main similarity to ensure a convex combination of individual scores.
Because some methods may not have any callers (e.g., main methods, methods intended and annotated as framework entry points) or callees (e.g., helper or very basic methods like getters), if one of these sets is empty the main similarity is used as a default value. This avoids tying the final similarity strongly to whether the method has both callers and callees.

$$s_{\text{main}} = \cos(q, e_{\text{main}}) \tag{5.1}$$

$$s_{\text{callers}} = \begin{cases} \text{agg}(\{\cos(q, e) \mid e \in \text{callers}(e_{\text{main}})\}), & \text{if callers}(e_{\text{main}}) \neq \emptyset, \\ s_{\text{main}}, & \text{if callers}(e_{\text{main}}) = \emptyset \end{cases} \tag{5.2}$$

$$s_{\text{callees}} = \begin{cases} \text{agg}(\{\cos(q, e) \mid e \in \text{callees}(e_{\text{main}})\}), & \text{if callees}(e_{\text{main}}) \neq \emptyset, \\ s_{\text{main}}, & \text{if callees}(e_{\text{main}}) = \emptyset \end{cases} \tag{5.3}$$

$$\begin{aligned} s_{\text{final}} &= w_{\text{main}} s_{\text{main}} + w_{\text{callers}} s_{\text{callers}} + w_{\text{callees}} s_{\text{callees}}, \\ & w_{\text{main}} + w_{\text{callers}} + w_{\text{callees}} = 1, \quad w_{\text{main}}, w_{\text{callers}}, w_{\text{callees}} \geq 0 \end{aligned} \tag{5.4}$$

This leaves us with a strategy we can slot into a LiSSA configuration. The most obvious usage is following the example of FTLR and choosing the weights heavily weighted towards the main similarity. But it leaves us with many more options. In particular, using this outline, we can replicate the default cosine-distance-based retrieval ($w_{\text{main}} = 1, w_{\text{calllers}} = w_{\text{callees}} = 0$) or characterize an element's similarity to a query with the maximum similarity of elements it is related to by either a caller or callee relationship. We will explore these in our evaluation of different retrieval strategies.

5.3. Evaluation

In this section, the previously introduced classifiers and retrieval strategies are evaluated. We subdivide our evaluation into three steps, first evaluating the preprocessors with the default retrieval strategy employed by LiSSA. Next we combine the preprocessors with context aware retrieval strategies and evaluate a set of full rertrieval configurations. Finally we compare the optimal configurations for each dataset as well as the one that produces the best results averaged across all datasets to those achieved by LiSSA.

With the exception of `MethodJavaDocEnhancerPreprocessor` we first evaluate all preprocessor that have been introduced in this chapter with the default cosine similarity retrieval strategy and compare their performance to that of `SingleArtifactPreprocessor` using the same default strategy. Evaluating the `MethodJavaDocEnhancerPreprocessor` has not been possible due to time constraints and the large number of LLM inferences required to evaluate its performance on the full test datasets.

In this first part of the evaluation, our goal is to show that, just like the preexisting preprocessors, our new preprocessors can enable the recovery of TLs between requirements and code based

on their outputs cosine similarity.
Condensing this using the GQM-Framework:

Goal 1: Increase the number of true positives TLs candidates recovered by LiSSA's retrieval stage.

Question How many of the TLs in each of the datasets are recovered using our preprocessors?

Metric Recall achieved by configurations using the different preprocessors

Thus, for the first part of our evaluation we do not report the precision retrieval with our preprocessors achieves. As part of the LiSSA framework, retrieval only constitutes the first step of the TLR process, and the retrieved TLs are further filtered. Therefore, precision and recall are not of equal importance. In particular, the recall of the retrieval acts as an upper bound for the recall of the entire framework. For this reason we focus on the achieved recall and compare different configurations based on the F_2-Score they achieve.

Goal 2: Investigate if and how retrieval using our preprocessors and retrieval strategies alter element similarities and the ordered retrieval lists produce during retrieval.

Question 2.1 How do our preprocessors effect the similarity of element linked by true positive TLs?

Metric Difference in similarity scores of element's linked by true positive TLs

Question 2.2 How do our preprocessors effect the retrieval ranking of element linked by true positive TLs?

Metric Indices of true related element's in the retrieval ranking l TLs

5.3.1. Parameters

The most important parameter of the retrieval is k, which defines how many target elements are retrieved for each source element. Independent of the preprocessor used, k represents a tradeoff between recall and precision: increasing k generally increases recall but decreases precision, as more candidate TLs are retrieved.
A second parameter that is independent of a specific preprocessor is the model used to embed the preprocessed elements.
For all evaluations, we use OpenAI's `text-embedding-3-large` model.
Preprocessors that employ an LLM also take the model as a parameter. In accordance with LiSSA's LLM interaction interface, each model is defined by its name and a random seed.
We use the same model for all evaluations, specifically OpenAI's `GPT-4o mini` with a random seed value of 42.

5.3.2. Results

In this section we report the results achieved by using our each of our preprocessors in LiSSA'S retrieval stage. The evaluated preprocessors are:

- `LLMClassSumaryPreprocessor`

- `JavaDocEnhancerPreprocessor`
- `AllJavaDocInClassPreprocessor`
- `ClassSkeletonPreprocessor`
- `MethodJavaDocPreprocesor`
- `ToMethodsPreprocessor`
- `ToMethodContextPreprocessor`

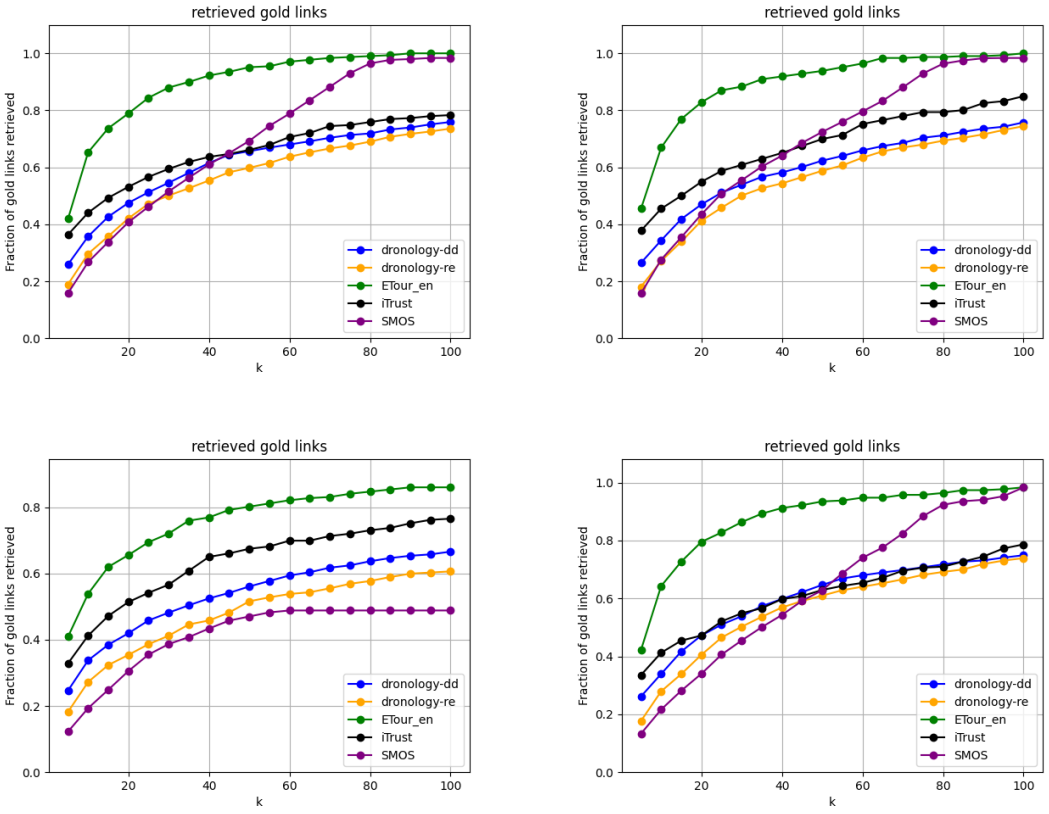


Figure 5.2.: Recall of retrieval using different preprocessors on the evaluation datasets. From left to right and top to bottom: `SingleArtifactPreprocessor`, `LLMClassSummaryPreprocessor`, `JavaDocInClassPreprocessor`, `JavaDocEnhancerPreprocessor`.

5.3.2.1. Preprocessors

We use the results obtained with the `SingleArtifactPreprocessor` and the default cosine similarity–based retrieval as a baseline for comparison.

Each plot shows one specific configuration of preprocessor and retrieval strategy. Each set of colored dots corresponds to one value of k and the resulting recall. However, they come with a some caveats: The connecting curve is added solely for readability. As the behavior of these curves shows, the fraction of gold-standard TLs does not necessarily grow linearly between sampled k values.

When interpreting these plots, it is important to keep in mind that a higher recall value for one dataset at a given k does not necessarily imply that the classifier performs better on that dataset. Each project has a different number of requirements, TLs, and classes (see Table 4.1), while the number of candidate TLs that can be retrieved is always $|\text{Requirements}| \cdot k$. In practice, this means that, for `iTrust`, a value of $k = 2$ could already be sufficient, whereas for `SMOS`, k must be greater than 16 for LiSSA to be able retrieve all gold-standard TLs, even assuming perfect similarity between gold-standard TL source and target elements using the given preprocessor. Nonetheless, these plots allow for an easy comparison between different configurations.
Since the space of potential configurations to evaluate is very large without accounting for the different datasets, but at the same many of the results follow the same general pattern, we mostly focus this section on the Dronology-DD dataset.

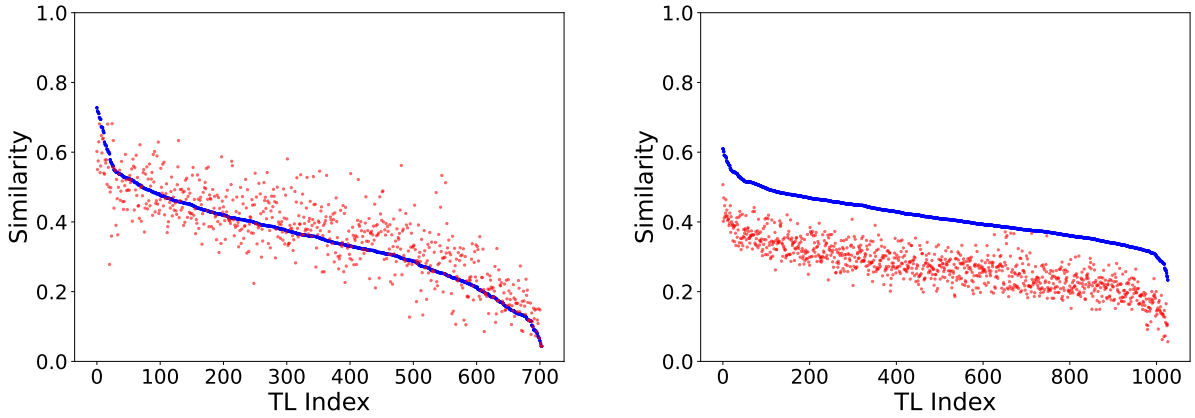
Starting with a baseline consisting of each preprocessor combined with the default retrieval method (i.e., taking the cosine similarity of embedding to rank the stored elements in relation to the query), the results show no major differences in performance among most of the new preprocessors. For both `LLMClassSummaryPreprocessor` and `JavaDocEnhancerPreprocessor`, the differences in recall values compared to the `SingleArtifactPreprocessor` are minor, and the curves exhibit largely similar behavior and bounds. While there are small variations in how many gold-standard TLs are retrieved for a given k across different preprocessors, these differences are not substantial.

The one clear exception are the result of `AllJavaDocInClassPreprocessor` on SMOS. Its recall is consistently lower than that of the other preprocessors by roughly 0.1, and it plateaus at 0.45 for k values greater than 80, falling well short of the near-1.0 recall achieved by the other preprocessors on this dataset. The most likely explanation for this is that both the requirements and documentation of SMOS are written in Italian. As shown in Table Table 4.1, SMOS does not have substantially fewer JavaDoc comments associated with its source code than the other projects, so a lack of comments is unlikely to be the reason for this disparity. Since the `JavaDocEnhancerPreprocessor`—which translates all non-English comments into English while refining them—achieves results more consistent with the other datasets, we can assume that the embedding model is less capable of accurately modeling the similarity between Italian comments and requirements with its embeddings.
For `Dronology-DD`, Dronology-RE iTrust and SMOS using the default retrieval strategy, each preprocessor falls short of achieving recall values that would make this configuration viable for real-world use.

Similarity Changes Each of the preprocessors should substantially increase the pairwise similarity between elements linked by gold-standard trace links, ideally up to a value of 1. Table 5.1 and the per-dataset and preprocessor mean changes in similarity show that this is not the case. With the exception of `JavaDocEnhancerPreprocessor` and `LLMClassSummaryPreprocessor` on SMOS, changes in mean similarity across each of the datasets are limited to less than 0.1. The majority of mean changes is also negative, decreasing similarity between linked source and target elements. Except for the `LLMClassSummaryPreprocessor` all class level preprocessors reduce the similarity. On SMOS, all preprocessors are detrimental to the similarity of gold-standard TLs. Using the most and least benefical preprocessor and dataset combinations as examples Figure 5.3 shows the individual similarty values for gold-standard TLs wiithout any preprocessing, i.e. their baseline similarity compared with their similarity after preprocessing. The plot shows that similarities prior and after preprocessing correlate and that even in our best

Table 5.1.: Baseline similarities and mean changes in the similarity of elements linked by a gold-standard TL for each of the preprocessors and evaluation datasets. For method preprocessors that produce more than one element per artifact, the maximum element similarity is used.

	Dronology-DD	Dronology-RE	ETour	iTrust	SMOS
Baseline	0.349	0.358	0.400	0.328	0.412
`AllJavaDocInClass`	−0.017	−0.017	−0.025	−0.014	−0.033
`ClassSkeleton`	−0.023	−0.027	−0.022	−0.012	−0.086
`JavaDocEnhancer`	−0.020	−0.018	−0.032	−0.007	−0.108
`LLMClassSummary`	−0.004	0.005	−0.018	0.005	−0.140
`MethodJavadoc`	0.005	−0.014	−0.030	0.028	−0.059
`ToMethodContext`	0.020	0.008	0.002	0.025	−0.015
`ToMethods`	0.017	0.000	−0.010	0.021	−0.017

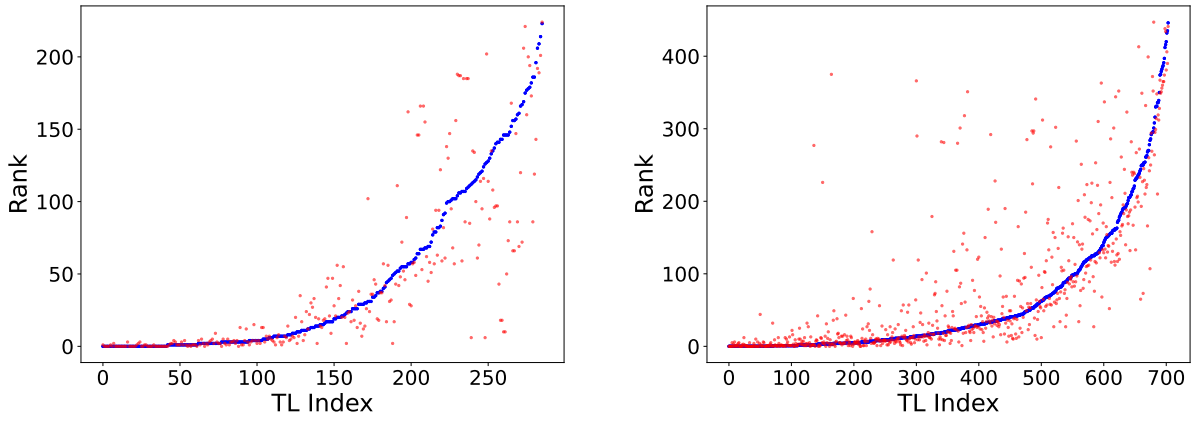


(a) `ToMethodContextPreprocessor` Dronology-DD **(b)** `LLMClassSummaryPreprocessor` on SMOS

Figure 5.3.: Similarity of elements linked a gold-standard TL without preprocessing (blue) and after being transformed by the preprocessor referenced in the respective subcaption (red).

evluated configuration, the preprocessor decreased the similarity of some TLs, even those with relatively high unpreprocessed similarity. In the worst case, the decrease is close to uniform across the entire set of TLs. These observations, also hold true for the other preprocessor and dataset combinations. Notably, as Table 5.1 shows, the method preprocessors generally produce a smaller decrease or even an overall increase in similarity in contrast to the class level preprocessors. This is largely due to the use of using the most similar method to represent the gold-standard TL. Using the mean or average method similarity yields comparable results to the class level preprocessors.

Rank Changes The change in similarity between requirements and linked class due to applying a preprocessor to the latter is not the only way to quantify the effect this preprocessor has on the set of retrieved TLs. For each source element, whether the correct target element will be retrieved depends on both its similarity to the source element as well as that of all other target elements. Specifically, a decrease in the similarity of the elements forming a gold-standard TL maybe be acceptable or even beneficial if there is a similar or stronger decrease in the similarity of unrelated elements.
One way to assess this is to examine how the use of different preprocessors effects the rank i.e.



(a) `LLMClassSummaryPreprocessor` on iTrust

(b) `AllJavaDocInClassPreprocessor` Dronology-DD

Figure 5.4.: Rank of elements linked a gold-standard in the retrieval list using the default cosine retrieval strategy TL without preprocessing (blue) and after being transformed by the preprocessor referenced in the respective subcaption (red).

the index of correct target elements in the ordered list of candidate target elements created during retrieval. For each preprocessor and each dataset, the mean difference in this index along with its value when using no preprocessor are show in Table 5.2. For most combinations of preprocessor and dataset, in the mean, the preprocessor is detrimental to the correct target element's rank. For method preprocessors this can be traced back to the substantially larger number of elements making up the set of potential target elements, but class level method preprocessors are also largely detrimental. While there are benefical preprocessors for all datasets except Dronology-DD, the negative effect of th detrimental preprocessors is generally greater than that of beneficial preprocessors on the same dataset.

Examining two examples in Figure 5.4, the effect of preprocessing in both the worse and best case configuration is not strictly positive or negative across all TLs in the dataset. It again, also seems to be correlated with its rank without preprocesing, with TLs with low ranks seeing their rank altered very little.

Table 5.2.: Baseline ranks and mean changes in rank of gold-standard target elements when using their source element during retrieval for each of the preprocessors and evaluation datasets. For method preprocessors that produce more than one element per artifact, the maximum element similarity is used.

	Dronology-DD	**Dronology-RE**	**ETour**	**iTrust**	**SMOS**
Baseline	60.8	65.7	12.3	45.1	32.1
`AllJavaDocInClass`	16.7	16.7	5.1	1.2	2.6
`ClassSkeleton`	10.9	12.6	1.5	0.3	2.8
`JavaDocEnhancer`	0.8	−0.4	2.3	−1.5	4.6
`LLMClassSummary`	3.8	1.2	−0.6	−3.2	−1.5
`MethodJavadoc`	434.6	471.1	170.5	375.7	114.2
`ToMethodContext`	505.0	553.8	185.5	369.6	102.8
`ToMethods`	546.2	598.0	225.4	378.1	97.7

Result Intersections While different preprocessors may retrieve a comparable number of gold-standard TLs, the specific TLs they recover may differ. For instance, a full class source file containing a hard-coded message string may exhibit greater similarity to a requirement referencing that string than a summary omitting it. Conversely, removing method bodies and thus most of a source file's content can drastically alter its similarity to a given requirement. Alternatively, an LLM-generated summary capable of inferring the class's intent despite inaccurately named methods may yield higher similarity to the corresponding requirement than the raw source code itself.
To examine these hypotheses, we analyze the sets of TLs retrieved by each preprocessor under the default cosine similarity retrieval with $k = 20$.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{5.5}$$

The overlap between two retrieved sets can be quantified using the Jaccard index, which measures the ratio of the intersection to the union of two sets A and B, as shown in Equation 5.5. A Jaccard index of 1 indicates identical sets, meaning that both preprocessors recover exactly the same TLs. This observation motivates the following GQM-Plan for the next stage of analysis:

Goal 3: Demonstrate that different preprocessors lead to distinct sets of TLs being recovered during retrieval

Question: How distinct are the sets of TLs recovered by each pair of preprocessors?

Metric: Jaccard index of different preprocessor's result sets

Table 5.3.: Pairwise Jaccard index of the sets of TLs retrieved using different preprocessors, all using the default retrieval strategy and $k = 20$ on the iTrust dataset

Preprocessor	SingleArtifact	AllJavaDocInClass	ClassSkeleton	JavaDocEnhancer	LLMClassSummary	MethodJavadoc	ToMethodContext
AllJavaDocInClass	0.47						
ClassSkeleton	0.48	0.45					
JavaDocEnhancer	0.69	0.42	0.48				
LLMClassSummary	0.60	0.41	0.47	0.61			
MethodJavadoc	0.31	0.28	0.31	0.32	0.30		
ToMethodContext	0.30	0.28	0.31	0.30	0.29	0.53	
ToMethods	0.30	0.29	0.32	0.30	0.30	0.55	0.82

Using the iTrust dataset as an example, Table 5.3 presents the pairwise similarity of TL sets retrieved by each preprocessor, represented by the Jaccard index.
The corresponding results for the other datasets exhibit comparable values and trends and are provided in Appendix A. At first glance, the sets of potential TLs produced by the different retrieval configurations appear relatively distinct. However, given that only a small proportion of the retrieved TLs are true positives, this impression may be misleading. For example, two preprocessors might recover nearly identical true TLs but differ substantially in their false positives, resulting in low overall similarity.
To account for this, we also examine the Jaccard indices computed over the subsets of true positive TLs, as shown in Table 5.4. This analysis confirms the concern: removing false positives

prior to computing the metric markedly increases the index values. With a Jaccard index of approximately 0.7, the sets of true TLs differ most between the `ToMethodContextPreprocessor` and the `ClassSkeletonPreprocessor`. For most other pairs, similarity is even higher, reaching up to 0.85 for the `SingleArtifactPreprocessor` and `JavaDocEnhancerPreprocessor`.
These high similarity scores indicate that the sets of true TLs recovered across preprocessors overlap substantially. In other words, no individual preprocessor appears to retrieve a fundamentally different subset of true TLs.
Taken together, these findings suggest that combining multiple preprocessors—by passing the union of their retrieved TLs to the classifier—would provide limited benefit.
Although the merged set would increase in size, its proportion of true positives would likely decrease, leading to additional classifier invocations for filtering false positives. Hence, this approach would incur additional computational cost without yielding a meaningful improvement in recall.

Table 5.4.: Pairwise Jaccard index of the sets of **true** TLs retrieved using different preprocessors, all using the default retrieval strategy and $k = 20$ on the iTrust dataset

Preprocessor	SingleArtifact	AllJavaDocInClass	ClassSkeleton	JavaDocEnhancer	LLMClassSummary	MethodJavadoc	ToMethodContext
AllJavaDocInClass	0.83						
ClassSkeleton	0.83	0.85					
JavaDocEnhancer	0.88	0.77	0.82				
LLMClassSummary	0.81	0.76	0.75	0.78			
MethodJavadoc	0.74	0.70	0.71	0.73	0.70		
ToMethodContext	0.78	0.76	0.78	0.75	0.73	0.85	
ToMethods	0.79	0.76	0.78	0.76	0.74	0.87	0.98

5.3.2.2. Retrieval Configurations

The ability to freely chose the preprocessor, number of elements to retrieve per source element as well as the retrieval strategy and its parameters for each dataset induces a large space of potential configuration that can be evaluated. To avoid overfitting or results to a specific dataset, we evaluate the results of each of the preprocessors we have implemented on every dataset described in chapter 4. We chose $k = 20$ both because it is a value that is both large enough to capture a significant fraction of the gold-standard links of each project, but is not so large to make the configuration unviable and because it enables a straightforward comparison of our results with those of Fuchß et al. For this section we also limit the discussion to the configurations that yield the highest F_2 score for each dataset.

Examining these for the different datasets shows a clear trend: Dronology-RE and SMOS benefit most from class level preprocessors, i.e. preprocessor that produce exactly one element to represent each artifact.

Dronology-RE For Dronology-RE, the top ten configurations all utilize either SingleArtifact-Peprocessor, JavaDocEnhancerPreprocessor or LLMClassSummary. These configurations also all use the context aware retrieval strategy, though they differ in the weights used. Barring

Table 5.5.: Dronology-RE – Top 10 Configurations

	Preprocessor	w_{main}	w_{calllers}	w_{callees}	P	R	F1	F2
1	SingleArtifact	0.4	0.3	0.3	0.131	0.443	0.203	0.3
2	SingleArtifact	0.2	0.4	0.4	0.129	0.436	0.2	0.296
3	JavaDocEnhancer	0.2	0.4	0.4	0.129	0.434	0.199	0.295
4	LLMClassSummary	0.6	0.2	0.2	0.128	0.433	0.198	0.293
5	SingleArtifact	0.6	0.2	0.2	0.128	0.433	0.198	0.293
6	LLMClassSummary	0.2	0.4	0.4	0.127	0.429	0.196	0.291
7	LLMClassSummary	0.4	0.3	0.3	0.127	0.429	0.196	0.291
8	SingleArtifact	0.8	0.1	0.1	0.127	0.429	0.196	0.291
9	JavaDocEnhancer	0.4	0.3	0.3	0.126	0.424	0.194	0.288
10	SingleArtifact	0.8	0	0.2	0.125	0.422	0.193	0.286

Table 5.6.: eTour – Top 10 Configurations

	Preprocessor	w_{main}	w_{calllers}	w_{callees}	P	R	F1	F2
1	MethodJavadoc	0.2	0.4	0.4	0.329	0.675	0.443	0.558
2	MethodJavadoc	0.4	0.3	0.3	0.325	0.675	0.438	0.555
3	ToMethodContext	0.6	0.2	0.2	0.329	0.662	0.439	0.55
4	ToMethodContext	0.2	0.4	0.4	0.331	0.659	0.441	0.55
5	ToMethodContext	0.4	0.3	0.3	0.33	0.659	0.439	0.549
6	ToMethodContext	0.8	0.1	0.1	0.323	0.662	0.434	0.547
7	ToMethodContext	1	0	0	0.32	0.659	0.43	0.544
8	MethodJavadoc	0.6	0.2	0.2	0.309	0.659	0.421	0.538
9	LLMClassSummary	0.6	0.2	0.2	0.222	0.834	0.35	0.537
10	SingleArtifact	0.4	0.3	0.3	0.222	0.834	0.35	0.537

the pure cosine similarity of the query and main candidate element, among the top ten configurations, the weights do not seem to play a major role in the accuracy of the recovered TLs.

SMOS For SMOS, the top ten are only made up by LLMSummaryPreprocesor and ClassSkeletonPreprocessor. Unlike Dronlogy-DD, there is clear trend of the former being superior to the latter. Both precision and recall are unaffected by the the strategy configuration.

Dronology-DD For the remaining three datasets, method level preprocessor are part of the majority of top performing configurations: For Dronology-DD, the MethodJavaDocPreprocessor combined with each of of the evaluated retrieval strategies are part of the top ten preprocessor, each producing very similar but distinct results. Five of these six take up the five topmost spots, with configurations using SingleArtifactPreprocesor, ToMethodContextPreprocessor and JavaDocEnhancerPreprocessor slightly underperforming them. Taking a coser look at the MethoddJvaDocPrepreocessor configurations, the more equal weight retrieval strategies seem to outperform those favoring either caller and calless or the main element, but with a different of less than 4% in F_2 scores between the best and worst performaing configurations, it's not possible to draw any larger conclusion about the optimal weight choices.

Table 5.7.: SMOS – Top 10 Retrieval Configurations

	Preprocessor	w_{main}	w_{calllers}	w_{callees}	P	R	F1	F2
1	LLMClassSummary	0.2	0.4	0.4	0.339	0.435	0.381	0.411
2	LLMClassSummary	0.4	0.3	0.3	0.339	0.435	0.381	0.411
3	LLMClassSummary	0.6	0	0.4	0.339	0.435	0.381	0.411
4	LLMClassSummary	0.6	0.2	0.2	0.339	0.435	0.381	0.411
5	LLMClassSummary	0.8	0	0.2	0.339	0.435	0.381	0.411
6	LLMClassSummary	0.8	0.1	0.1	0.339	0.435	0.381	0.411
7	LLMClassSummary	1	0	0	0.339	0.435	0.381	0.411
8	ClassSkeleton	0.2	0.4	0.4	0.323	0.415	0.363	0.393
9	ClassSkeleton	0.4	0.3	0.3	0.323	0.415	0.363	0.393
10	ClassSkeleton	0.6	0	0.4	0.323	0.415	0.363	0.393

Table 5.8.: Dronology-DD – Top 10 Retrieval Configurations

	Preprocessor	w_{main}	w_{calllers}	w_{callees}	P	R	F1	F2
1	MethodJavadoc	0.4	0.3	0.3	0.108	0.415	0.171	0.265
2	MethodJavadoc	0.6	0.2	0.2	0.107	0.417	0.171	0.264
3	MethodJavadoc	0.8	0.1	0.1	0.105	0.417	0.168	0.262
4	MethodJavadoc	0.2	0.4	0.4	0.107	0.411	0.17	0.262
5	MethodJavadoc	1	0	0	0.104	0.413	0.167	0.259
6	SingleArtifact	0.2	0.4	0.4	0.087	0.508	0.148	0.258
7	ToMethodContext	0.2	0.4	0.4	0.104	0.402	0.165	0.256
8	MethodJavadoc	0.8	0	0.2	0.103	0.406	0.164	0.256
9	JavaDocEnhancer	0.2	0.4	0.4	0.086	0.503	0.147	0.255
10	SingleArtifact	0.4	0.3	0.3	0.086	0.503	0.147	0.255

eTour For eTour, the list of top preprocessor is slightly more mixed, though still heavily weighted towards method level preprocessors. MethodJavaDocPreprocessor configurations take up ranks one,two and eight with those containing ToMethodContextPeprocessor make up ranks three to seven. Ranks nine and ten are LLMClassSummaryPreprocessor and SingleArtifactPreprocessor configuratinons, both producing results with the exact same precision and recall values.

5.3.3. Comparison with LiSSA

We compare our optimal configurations per dataset with the those Fuchß et. al present using the three original preprocessors and a mock classifier, i.e. one that will classify every TL produces by the retrieval as a true TL.
As shown in 5.10 none of our configurations can strictly outperform the preexisting preprocessors with purely cosine similarity based retrieval. On Dronology-DD, our configurations are even strictly worse than the `ChunkingPreprocessor`. For the other datasets, the comparison is more mixed: For eTour, iTrust and Dronology-RE our respective best configurations yields a higher F_2 score than any of the reference configurations. Unfortunately, for both eTour and iTrust this increase is the result of significantly increased precision outweighing a decrease in recall. For Dronlogy-DD the metrics precision and recall in between those produced by

Table 5.9.: iTrust – Top 10 Retrieval Configurations

	Preprocessor	w_{main}	w_{calllers}	w_{callees}	P	R	F1	F2
1	ToMethods	0.4	0.3	0.3	0.096	0.535	0.163	0.279
2	ToMethodContext	0.6	0.2	0.2	0.097	0.528	0.164	0.279
3	ToMethods	0.2	0.4	0.4	0.096	0.531	0.163	0.279
4	ToMethodContext	0.8	0.1	0.1	0.097	0.528	0.163	0.279
5	ToMethodContext	0.8	0	0.2	0.096	0.528	0.163	0.279
6	ToMethodContext	1	0	0	0.096	0.528	0.163	0.278
7	ToMethodContext	0.4	0.3	0.3	0.097	0.524	0.163	0.278
8	ToMethods	0.6	0.2	0.2	0.096	0.531	0.162	0.278
9	ToMethodContext	0.6	0	0.4	0.096	0.524	0.162	0.277
10	ToMethods	1	0	0	0.095	0.531	0.161	0.277

Table 5.10.: Comparison of our best configuration retrieval setup against LiSSA's default cosine similarity retrieval and original preprocessors.

Dataset	**Configuration**	**Precision**	**Recall**	F_1**-Score**	F_2**-Score**
SMOS	NONE	0.325	0.418	0.366	0.395
	CHUNK(200)	0.247	**0.546**	0.340	0.439
	METHOD	0.327	0.541	**0.408**	**0.479**
	Ours$_{OPT}$	**0.339**	0.435	0.381	0.411
eTour	NONE	0.216	**0.815**	0.342	0.525
	CHUNK(200)	0.091	**0.815**	0.164	0.315
	METHOD	0.073	0.597	0.130	0.245
	Ours$_{OPT}$	**0.329**	0.675	**0.443**	**0.558**
iTrust	NONE	0.058	0.531	0.105	0.202
	CHUNK(200)	0.066	0.563	0.119	0.225
	METHOD	0.063	**0.598**	0.114	0.221
	Ours$_{OPT}$	**0.096**	0.535	**0.163**	**0.279**
Dronology-RE	NONE	0.128	0.420	0.196	0.288
	CHUNK(200)	**0.150**	0.331	**0.206**	0.266
	METHOD	0.132	0.282	0.180	0.230
	Ours$_{OPT}$	0.131	**0.443**	0.203	**0.300**
Dronology-DD-	NONE	0.085	**0.482**	0.144	0.249
	CHUNK(200)	**0.119**	0.424	**0.186**	**0.281**
	METHOD	0.101	0.369	0.159	0.241
	Ours$_{OPT}$	0.108	0.415	0.171	0.265

no preprocessing and chunking, leading to a slightly greater F_2 score. For SMOS, our best configuration has the highest precision value, but with substantially lower recall value, has a lower than F_2 score than the MethodPreprocessor.

5.3.4. Conclusion

In this chapter, we have explored three different approaches to improving LiSSA's retrieval performance:

- Transforming artifacts into alternative representations
- Subdividing artifacts into components prior to transforming these components
- Combining the similarity between the query and both the candidate target and its neighborhood into a single metric

As part of each of these approaches, we have either incorporated or deliberately excluded aspects of the original artifacts' context. Our evaluation shows that all of these approaches can be beneficial to accurate traceability link recovery when used and combined appropriately. There is no "silver bullet" preprocessor among those we have implemented—none can reliably improve retrieval performance across all projects. Instead, the preprocessing strategy should be tailored to the specific project and the level of abstraction at which its requirements describe the software Similarly, none of the parameterizations of our context-aware retrieval strategy evaluated strictly outperforms all others. The only consistent trend regarding synergies between retrieval weights and the preprocessors used is that—unsurprisingly—strategies placing greater emphasis on the main element's similarity appear to be slightly better suited for method-level preprocessors than for class-level ones. Considering our results in absolute terms rather than relative to LiSSA, it is clear that our context-aware preprocessing and retrieval approaches still fall short of the quality required for practical, real-world use. On average across the datasets, our best configurations achieve, for $k = 20$, a precision of 0.223 and a recall of 0.522. In other words, they capture just over half of the traceability links present in the gold standard for these projects, and only about one in five recovered links is correct. Depending on the preprocessors used—and particularly for method-level preprocessors that rely on LLMs to produce their outputs—the modest gains achieved by some of our evaluation configurations may not justify the large number of LLM inferences required.

6. Agentic Classification

In this chapter, we present agentic approaches to TL recovery, their implementation, and their evaluation on the datasets described in chapter 4.
In the previous chapter, we have explored different ways to transform source code artifacts to arrive at an embedding more similar to those of the respective requirement.
None of the combinations of preprocessors and retrieval strategies produced results suitable for real-world use. Since the classifier is only presented with the elements that the retrieval has identified as candidates, poor retrieval performance acts as a limit for the classifier's performance.
As described in Section 2.3, skipping the retrieval and simply prompting one of LiSSA's LLM-based classifiers with every pair of requirement and class in the Cartesian product of the two sets would lead to a an unfeasibly large number of LLM classifications being necessary, making the technique both slow and expensive.
A different approach is needed. As outlined in chapter 1 one of the central challenges of leveraging source code context is the large, but still limited context window of even SOTA LLMs.

The evaluation of the different retrieval configurations also affirms that, at least when using textual similarity as a heuristic, it is not possible to find a fixed "radius" of context that is relevant to assess whether e.g. a class is linked to a requirement by a TL. Instead, which neighbor elements should be taken into account depends on the given input project, project as well as on the specific requirement and candidates classes examined.

6.1. Agents

One of the central advantages of modern machine-learning approaches—particularly deep learning—is their capacity to automatically learn task-relevant representations and strategies for solving problems directly from data, rather than relying on developers to predefine these features or decision rules [6]. Training a dedicated machine-learning model for TLR is beyond the scope of this thesis and, as outlined in chapter 1, is further constrained in practice by the need for retraining on project-specific TLs.
However, we can leverage the capabilities of large language models (LLMs)—and the representations and general knowledge they have acquired during pretraining—to approach the TLR task in a zero-shot manner.
This forms the foundation of the classification module in LiSSA, but we aim to expand the responsibilities of the LLM to also include the retrieval of candidate classes. Since prompting an LLM with the full project source is infeasible, the most straightforward way to involve an LLM in retrieval would be to prompt it with a requirement and the full list of class names, asking it to return those it considers linked to the requirement through a TL. Without any additional information, we can assume that the LLM will primarily rely on term matching

between requirement and class names, making broad assumptions about their content and potentially hallucinating aspects of the software to satisfy the request.
One way to improve the accuracy of TLs recovered by the LLM, is to provide it with richer information about the project and the classes it operates on. In this, we can resolve the conflict between the need to supply more context and the practical limitations imposed by the number of classes and the LLM's restricted input size by adopt lazy information access. Instead of making all available information part of the initial prompt, we limit the prompt to only the most basic entry point consisting of task, requirement and class names.
Any additional details are only provided only when the LLM explicitly requests them—that is, when they become relevant during its reasoning. The decision whether a detail has becomes relevant or not is left to the LLM, as is the exact approach it takes to recovering the TLs.
The framework only controls the LLM insofar as it defines the initial prompt, including the set of available tools and provides the requested information on demand to the LLM. To avoid the LLM exploring substantial parts of the project's source code or getting stuck in a loop, the framework will stop the process after a fixed number of iterations.
This use of an LLM, granting it access to functions it can call and letting it plan and execute a plan on its own across multiple iterations, but limiting interaction with a user to a single exchange of messages is usually described as agentic AI.

6.1.1. Agentic AI

In this subsection, we outline the details of how our agentic approach can recover TLs. The full process is shown in Figure 6.1 in the form of a UML activity diagram. The different required actions and decisions are divided between the application and the LLM. We begin by sending an initial message to the LLM to establish the context in which the agent is going to act autonomously. Most importantly, this includes its task, the project's package structure, and the requirement for which it is expected to find related classes. It is also provided with a list of tools, their descriptions, and examples of how to invoke them. We then expect the LLM to determine whether it can already identify the related classes. If this is the case, it is expected to output this list and trigger the end of the execution. Generally, the LLM will not determine related classes solely based on the requirement and the class names. Instead, it is expected to invoke one or more of the tools it has been provided with. In both cases, the decision whether to end the execution or call tools is communicated to the application via the LLM's response. The application will then either extract the related classes from the LLM's messages and return the induced TLs if a maximum number of iterations has been reached or the agent has triggered its exit, or it will execute the desired tools. The results of each tool invocation are then concatenated and, together with the full set of previous messages exchanged between the application and the LLM, sent back to the LLM.

As shown in Prompt 1, in the initial message sent to the LLM, the agent is presented with a short explanation of what constitutes a TL is and is informed that its task is to recover TLs for a specific requirement. It is also provided with a list of the available tools. As an entry point, the class and package hierarchy of the project is printed as an indented list of class and package names. Figure 6.2 shows an abridged version of the agent's execution from its "view" based on this prompt, including tool use and the output of the final classes the agent has identified as related to the requirement.

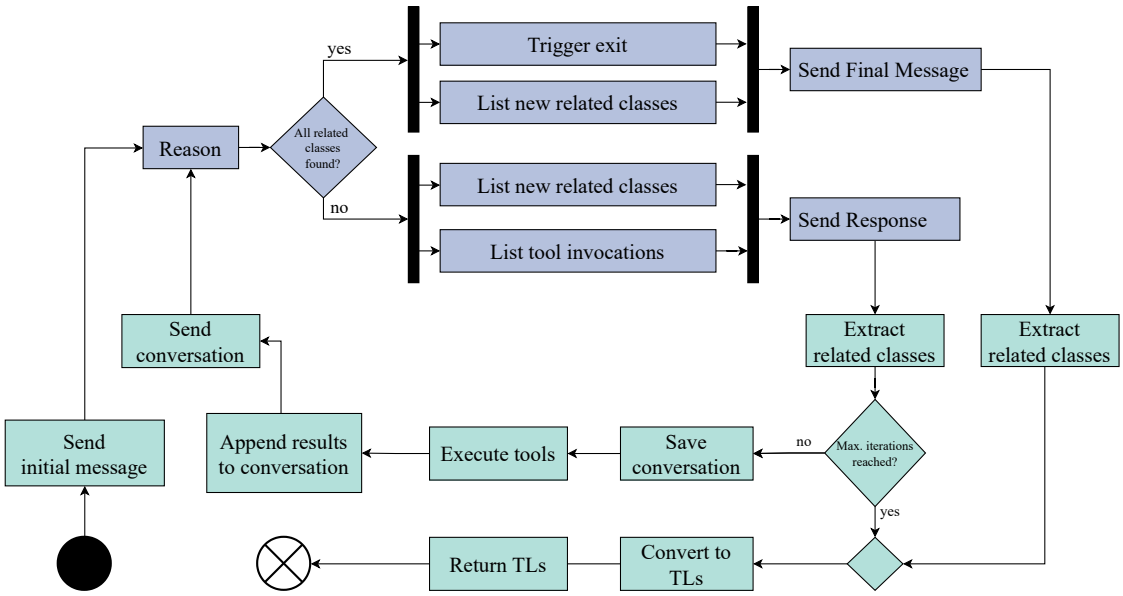


Figure 6.1.: Activity diagram showing the actions taken by the application (green) and the conceptual steps the LLM is prompted to perform

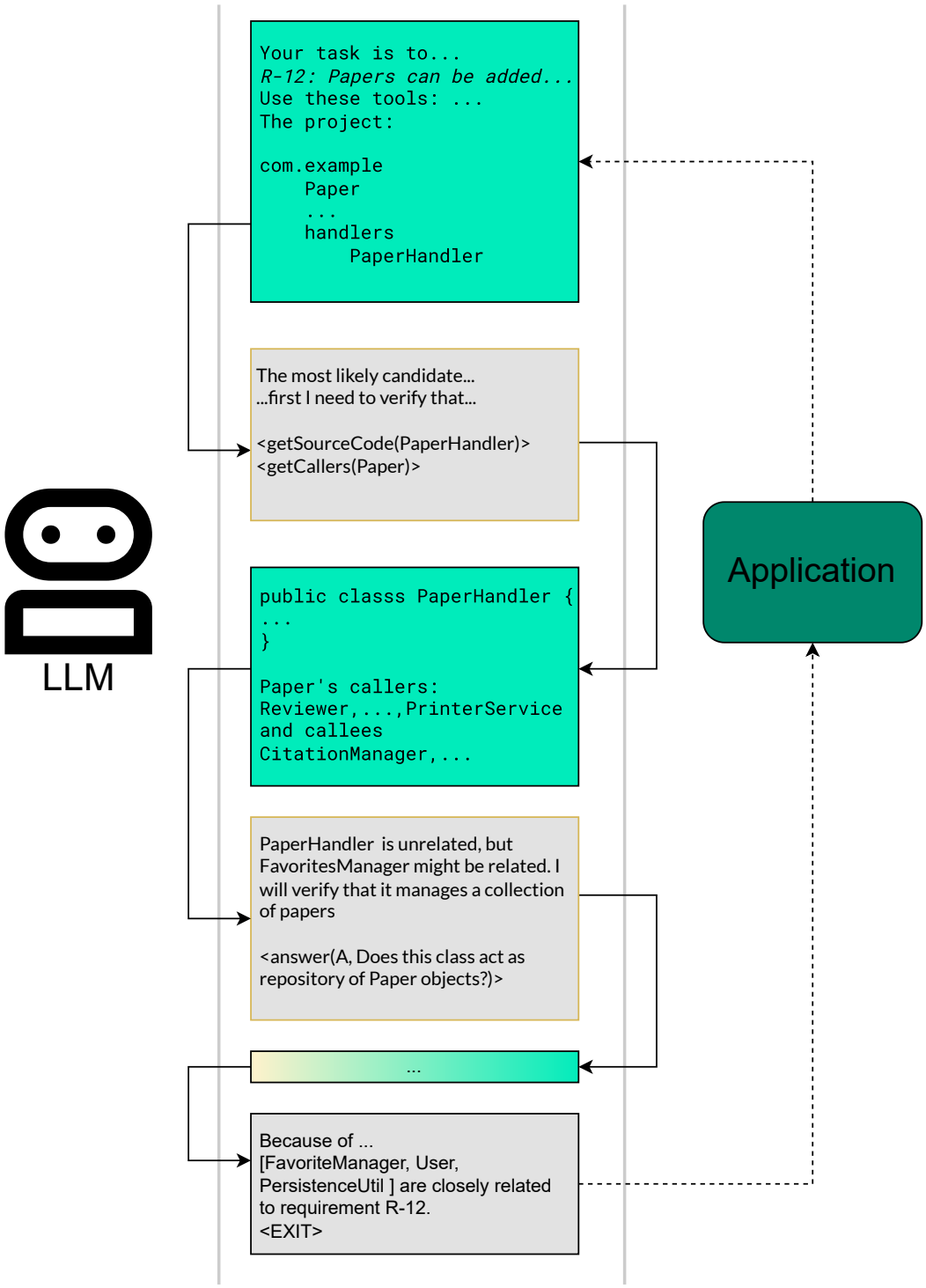


Figure 6.2.: Overview diagram of the iterative exchange of messages between the application and the LLM forming the execution of the agent.
Messages sent by the LLM are shown with a light gray background, and messages sent by the application with a light green background.

Prompt 1: Initial Agent Prompt
A traceability link (TL) is an implicit connection between a requirement and - in an object oriented programming paradigm - the classes in the project source code which implement the behavior specified in the requirement.
While there are exceptions, a single requirement is typically implemented by multiple classes and thus multiple TLs exist for each requirement.
However, this depends on the granularity of the requirement: Overarching descriptions of the system may be linked to many classes, while requirements proscribing specific values or algorithms are commonly linked to only a single or very few classes.

Your task is to recover the traceability links for this software project written in java.

The requirement you are supposed to find TLs for is:
...
getSourceCode: Retrieves the source code of a class. e.g. "getSourceCode(M̈yClass)̈"
...

To help you accomplish the task you have access to the following tools you can use to examine the project source code:
{...}
To use a tool justify its use and end your answer with its invocation like this:
<someToolsName(Example.java)>
You can also use multiple tools in a single answer like this: <someToolsName(Example.java)>, <someToolsName(SomeOtherClass.java)>

To start with, here is the projects package structure
{...}

A good first step is usually to determine classes that are likely related to the requirement, use tools to take a closer look then verify the assumptions you made. Don't be afraid to use additional tools if you find you have made the wrong initial guesses or want to investigate something about the project. If you are convinced you have found the correct TLs, justify the choices you made and output the classes connected to the requirement like this: [<ClassA>, <ClassF>, <ClassC>] and - if there is no more tool you want to use - finish your answer with <EXIT>
Let's think step by step.

6.1.1.1. Tools

Prompt 2: `Source Code Tool`
getSourceCode: Retrieves the source code of a class. e.g. "getSourceCode(M̈yClass)̈"

Internally each tool is implemented as a stateless class implementing the `AgentTool` interface. Facing the LLM, each provides a name, which is used to invoke it, a short description including its expected paramters and output and an example invocation. For an example of how this is commmunicated to the agent see Prompt 1 The agent can trigger tool executions by including

the tool's name and the desired parameters in its response. The framework then determines which `AgentTool` implementation to invoke and executes it accordingly. In accordance with its interface, each tool produces a string as output, which is concatenated with the invocation string that generated it and subsequently returned to the LLM. The agent is explicitly informed that multiple tool invocations can be issued within a single response. If a response contains several invocations, their individual outputs are concatenated and sent back to the agent as a single message.

Because an LLM may hallucinate and attempt to call imagined tools or invoke tools on elements that do not exist or are outside their designed scope, the framework must also handle invalid tool invocations. Each tool implementation is responsible for generating a descriptive error message, which is communicated to the LLM in the same manner as a valid output.

Tools generally access the code context model. Since TLR is a static analysis task, and the classification stage does not require modification of the model, tools only read values from the code context model rather than altering them. To enable the agent to effectively explore the code base, it can be equipped with the following tools:

Source Code The most direct way to determine whether a given class implements a requirement is to inspect its source code. Accordingly, the first tool provided to the agent returns the source code of a class corresponding to the supplied identifier.

Callers and Callees Beyond direct access to source code, it is beneficial to provide tools that aggregate information derivable from multiple source code accesses. This allows the agent to focus on solving the TLR task rather than manually analyzing potentially large quantities of source code.
Following the assumptions established in chapter 4 and chapter 5, the most relevant relationships between classes are those defined by inheritance and method-level caller–callee connections. Both relationships are therefore made accessible to the agent through dedicated tools.

Finder Because the agent cannot rely on a prior retrieval stage to supply candidate elements, it must independently identify relevant entry points for investigating relationships between source code artifacts. One possible starting point is the project's package structure, which the agent is initially provided with. However, this approach may bias the agent toward selecting classes whose names share lexical similarities with requirement phrases.
While the agent could, in principle, examine classes manually to locate relevant identifiers, this is impractical given the size of typical code bases and the limited number of iterations available. To address this, the agent is equipped with a `Finder` tool, which allows it to search for a string across the entire code base in a single operation, replacing numerous exploratory iterations with one efficient query. Given a search string, the tool returns the fully qualified names of all classes whose source code contains the input string. To improve robustness and mitigate inconsistent capitalization, both the search string and source code are lowercased before matching.

`Retriever` While the Finder enables efficient keyword-based searches, it still requires the agent to provide an exact search term—apart from case normalization—and to anticipate which terms may be relevant. If the exact string is absent, no results will be produced and the invocation becomes uninformative.
To avoid forcing the agent to "guess" relevant terms, the framework reuses the textual similarity–based retrieval mechanism from LiSSA's retrieval stage. The *`Retriever`* tool implements a top-k similarity search interface: given a query string and a value of k, it returns the k most similar project elements. Unlike the `Finder`, the `Retriever` also searches across both source code and requirements. Presenting the agent with similar requirements can aid in identifying or rejecting false positives. For requirements, the tool returns their full text; for classes, only fully qualified names are provided to prevent adding lengthy or potentially misleading code fragments to the dialogue.

Answerer If the LLM is regarded as an analogue to a human analyst performing traceability link recovery, each tool invocation can be interpreted as an attempt to answer a specific question about the code base. For example, the agent may access a class's source code to verify whether a particular algorithm is implemented within it. While a single line of code might suffice to answer such a question, retrieving the entire class introduces a substantial amount of irrelevant content, including unrelated instructions and potentially misleading comments. More targeted tools, such as the *Callers* and *Callees*, already help mitigate this issue by providing concise, relationship-focused information. Nevertheless, the diversity of possible questions makes it infeasible to predefine a specialized tool for each. To address this limitation, the *Answerer* tool enables the agent to query any class in the project using natural language. Given a class identifier and a natural language question, the tool resolves the identifier to its corresponding source code and then prompts an LLM to generate an answer based on that code. This design allows the agent to perform fine-grained reasoning about specific aspects of the code without directly processing large code fragments in context.

6.1.2. Alternative Classifiers

The agentic TLR approach, as described so far, has two main drawbacks compared to the existing LLM-based LiSSA classifiers. The first is the rapidly growing size of the input provided to the LLM at each step.
In LiSSA, the LLM used as a zero-shot classifier is prompted only with a short description of its task and, at most, a full requirement and the complete source code of a single class.
In our implementation of a TLR agent, the input is generally much larger. Theoretically, it might contain the full source code of all classes, interspersed with the text expressing the LLM's reasoning and tool invocations.
In practice, the agent does not access—i.e., add to its context window—a substantial fraction of the project's classes. However, since it is prompted to find all TLs for a given input requirement, it will typically access multiple classes, some of which it will reject as candidates but must still retain in the input.
Large inputs—especially those consisting largely of text irrelevant to the task—degrade the quality of the LLM's reasoning [8, 20]. Large amounts of irrelevant text are, however, unavoidable unless the agent consistently selects both the correct and minimal combination of tools and arguments necessary to verify the existence of TLs.
We implemented a classifier that prompts an LLM with an explanation of its task, a requirement,

and the project's package structure. But instead of providing it with a set of tools and asking it to autonomously determine the related classes, the model is instructed to generate a set of questions about the project's classes. It is told that these questions will be answered in the subsequent response and that it must make its final decision based solely on those answers, without performing any additional function calls.
Essentially, this implementation is almost equivalent to providing only the `Answerer` component and enforcing a two-message maximum exchange length.

In our main agentic implementation outlined in this chapter, all agents were free to make the final determination of which classes were linked to a requirement once all invoked tools had been executed and their results received.
Unfortunately, since LLMs are trained to be helpful—which in practice often means agreeable—the LiSSA classifiers tend to produce a large number of false positives. One way to mitigate this is by enforcing that a classification logically follows from statements or assumptions the model has explicitly expressed.
A key challenge with this approach, however, is that LLMs struggle with evaluating logical expressions or even simple formal proofs, whereas evaluating a boolean formula is trivial in most programming languages. For agent-based TL classification, this means that an agent may generate correct reasoning to justify a classification but still deduce the wrong verdict, thereby misclassifying the TL.
To address both of these issues, we implemented a final, more constrained agent. Instead of being tasked with solving the TLR task directly, this agent is prompted only to produce a boolean formula describing different potential sets of TLs. Each variable in the formula corresponds to a statement about a class in the project's source code that can be evaluated as either true or false. These individual assumptions are connected by logical AND operators, so each clause of the formula represents a potential constellation of TLs, each linking the input requirement to one or more classes, given that all component statements hold true.
After the LLM produces the formula, it is no longer involved in the TLR process. The framework parses the logical expression and uses the aforementioned `Answerer` to evaluate the truth value of each clause. If the formula evaluates to true, the corresponding TLs are returned; if it evaluates to false, no links are produced.
As with the other agents, we cannot assume that the LLM will correctly infer the complete set of candidate classes from the project structure—and because it cannot revise the candidate set after its initial response—the LLM is prompted not to produce a single conjunction of statements but rather a list of such terms, each representing a distinct potential set of TLs.

However, initial evaluation showed that in both implementations the LLM generally failed to accurately predict related classes and frequently asked questions that were too broad to be conclusively answered, resulting in very few recovered TLs. For this reason, we do not evaluate them in the following section and focus on our less constrained, autonomous implementation of agentic TLR.

6.2. Evaluation

Goal 4 Improve the accuracy of TLR by using an agentic framework.

Question 4.1 How many of the TLs in each of the datasets are recovered by our agents?

Metric Recall

Question 4.2 How many of the TLs recovered by our agents are true TLs?
Metric Precision

The central question guiding our evaluation is how effectively the agents fulfill their goal—namely, how well the different agents recover TLs.
As with the evaluation of preprocessors and retrieval strategies, the most important metrics for assessing the agentic classifier's performance are precision and recall. Since the agentic classifier directly outputs its results without any subsequent filtering stage, recall is not inherently more important than precision. However, recall should still be prioritized over precision, as it is generally easier for a human user to filter out false positives than to manually recover TLs that our technique has missed.

Goal 5 Investigate how iterative agents arrive at their results.

Question 5.1 Which tools are used the agents?
Metric Number of invocations per tool.

Question 5.2 How many iterations do agents take to arrive at their results
Metric Number of iterations

Question 5.3 What is the relationship between the set of output related classes and the set of classes having been used as tool arguments?
Metric Number of output classes in the set of tool argument classes, number of output classes not included

While LLMs can generalize well to unseen tasks, most are fine-tuned for specific use cases, differ in their capabilities and carry biases towards specific outputs. Consequently, not all LLMs will produce the same reasoning processes or recover the same TLs. We do not investigate how an individual LLM's architecture or training data influence whether the correct TLs are recovered by our agents. However, we can move beyond viewing the agents as black-box classifiers by analyzing the dialogue between the LLM and the surrounding framework.
For false negatives, we investigate whether the correct classes are never considered as candidates, or whether the LLM erroneously rejects them after evaluation.
For the iterative agents, we examine the set of classes the agent engages with—i.e., those on which it uses tools—and compare this set to both the gold standard classes and the final classes returned at the end of execution.
Finally, we compare the sets of recovered TLs between the different agent configurations, as well as with those recovered by our retrieval configurations. This latter comparison is particularly interesting, since agents represent a superficially distinct approach from traditional IR-based methods, yet LLM-based assessments of relatedness and embedding similarity may still be correlated [24].

Goal Compare the output of agentic TLR with that of context aware IR-based retrieval.

Question How does the quality of the recovered TLs by agents differ from those recovered using LiSSA's retrieval stage.
Metric Precision, Recall,F_1- and F_2-Scores

Question How similar are the sets of TLs recovered by agents to those produced by the retrieval stage?

Metric Jaccard index

6.2.1. Parameters

At the core of each agent is an LLM responsible for guiding both the retrieval and the assessment of classes that may form a TL with the input requirement. For our evaluation, the LLM used in this role is OpenAI's `GPT-4o mini`.
As in Section 5.3, we also use `GPT-4o mini` to answer questions about the source code of classes the Answerer tool is used on and calculate embeddings for the `Retriever` tool using OpenAI's `text-embedding-3-large` embedding model.
The agent is stopped after a maximum of 31 iterations, allowing for 30 sets of tool invocations and responses.

Since locating classes related to a requirement based on the project's structure requires a more project-wide perspective—extrapolating from retrieved information and reasoning about which classes are most relevant—we consider it more cognitively demanding than the bottom-up summarization and documentation enhancement tasks described in the previous chapter.
To account for this increased complexity and investigate whether a more capable LLM will produce substantially better results, we also evaluate our agents using `GPT-5 mini` as the the agent's main LLM and compare the performance of the two configurations directly.

Configurations Besides the number of iterations it is allowed to take which is equivalent to the number of times it can invoke tools and receive the results, the main way the agent can be configured is via the set of tools available to it.
On each of the datasets we executed the agent with different tool sets, specifically:

- The full set of tools
- All tools except the `SourceCode` tool
- The `SourceCode` tool, `Finder` and `Retriever`
- Only the `SourceCode` tool
- No tools

6.2.2. Results

As prescribed by our first GQM plan, we begin by reporting the results of executing our agent on each of the evaluation datasets, using the standard metrics of precision, recall and the combined F_1- and F_2-Scores to assess the quality of our agent's results.
Across all datasets, the agent fails to determine the classes related to the input requirement and thus does not recover a majority of the TLs existing in these projects, resulting in low recall values between 0.287 on iTrust and 0.099 on Dronology-RE. The difference between the maximum precision of 0.726 for SMOS and the minium precision of 0.152 for Dronology-DD is greater than that between the maximum and minimum recall values.
This can be traced back to the much greater number of TLs in relation to its number of

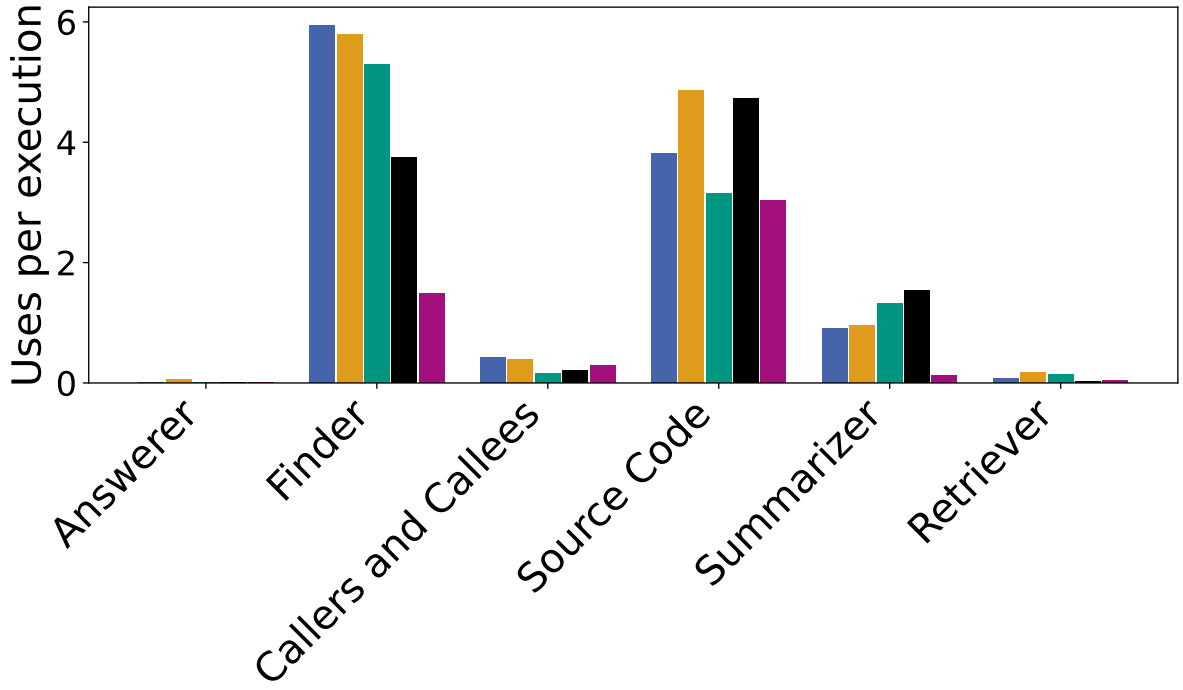


Figure 6.3.: Number of tool uses per dataset and tool, divided by the number of requirements (equivalent to the number of agents executions) of each project.
Datasets: Dronology-DD (blue), Dronology-RE (orange), eTour (green), iTrust (black), SMOS (purple)

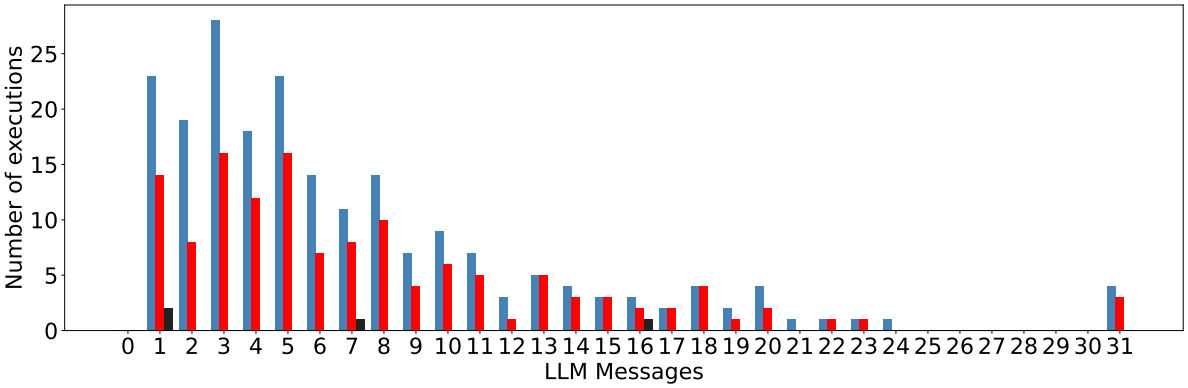


Figure 6.4.: Histogram of the length of the message exchanges between LLM and application for agent execution of all of Dronology-DD's requirements. For each length three values are shown: Total number of exchanges (blue), number of exchanges returning no correct TLs and number of exchanges returning no related classes.

requirements and classes in SMOS compared to the other projects (see Table 4.2. Taking the next highest precision value of 0.387 on eTour in SMOS'S stead leads to an interval of precision values of similar size as that of recall values.

Tool Use Moving on to investigating how the agent arrives these results we first examine the number and type of tools used. Figure 6.3 shows this as the average number of uses per agent execution of each tool for each of the datasets.
Among the six tools available to it, the agent heavily favors the `Finder` and `SourceCode` tool, i.e. the tools allowing it to find classes containing a substring and the one returning the full

Table 6.1.: The number of uses of the `Finder` tool per dataset including the number of successful uses (uses where at least one class containing the query string was found), failures and number of source code accesses targeting classes that have been part of a `Finder` results list.

Dataset	Total Finds	Failures	Successes	Subsequent Accesses
Dronology-DD	1255	921	334	203
Dronology-RE	574	449	125	84
ETour	307	248	59	65
iTrust	491	383	108	49
SMOS	100	68	32	42

source code of the referenced class. On all datasets, with the exception of SMOS, on average the Summarizer tool is only used once per agent execution. Answerer, Caller and Callees and `Retriever` are only used an even lower, negligable number of times.
Focusing on the two most frequently tools, we can see only two minor differences in the number of per requirement tool uses between datasets. On Dronology-DD, Dronlogy-RE and eTour, the `Finder` tool is used more frequently than the `SourceCode` tool, while the inverse is true for SMOS and iTrust.
Adding up uses of the different tools, the agent largely uses a similar number of tools per requirement when executed on each of the projects. The one exception from this is SMOS executed on which the agent uses substantially fewer tools.
It seems likely that the average number of tool uses is correlated with the average number of messages sent by the LLMs. Figure 6.5 shows the distribution of the exchange length for the executions on each dataset. While the average exchange lengths across datasets are similar, they differ in distribution. The box plot also shows a likely cause for SMOS low number of tool uses, not only is it's distribution the one with the lowest spread overall and lowest third quartile, it also lacks an strong outlier reaching the maximum message count.

Different tools differ not only in their results and implementation, but also in the arguments they expect and how they handle them. Tools that access a class's source code, its callers and callees, summaries, or that answer a question, all require a valid class identifier as their argument. The `Retriever` tool, in contrast, will return a valid, although potentially irrelevant, result for any input string. Barring substantial errors such as an LLM hallucinating classes that do not exist in the project, we can expect the LLM to invoke these tools only with valid arguments.
This is not the case for the `Finder` tool. We expect this tool to be used with query strings that frequently return no classes as the agent explores the project's source. When combined with the `SourceCode` tool, such uses likely constitute the majority of its invocations. Table 6.1 summarizes the number of `Finder` calls aggregated from a full execution of the agent on each dataset. It confirms this assumption, showing that between 68 % and 80 % of `Finder` uses yield no results. However, this does not mean that the tool is not valuable to the agent or central to the way it approaches the task. As indicated by the large number of subsequent source code accesses targeting classes that previously appeared in a `Finder` result, the agent seems to use this tool to first locate potentially relevant classes before retrieving their full source code.
Analogous to the similarity between the average number of tool uses and the number of messages per exchange, the number of classes returned by the agent is also highly consistent across datasets.
Despite differences in the total number of classes per project and the varying levels of abstrac-

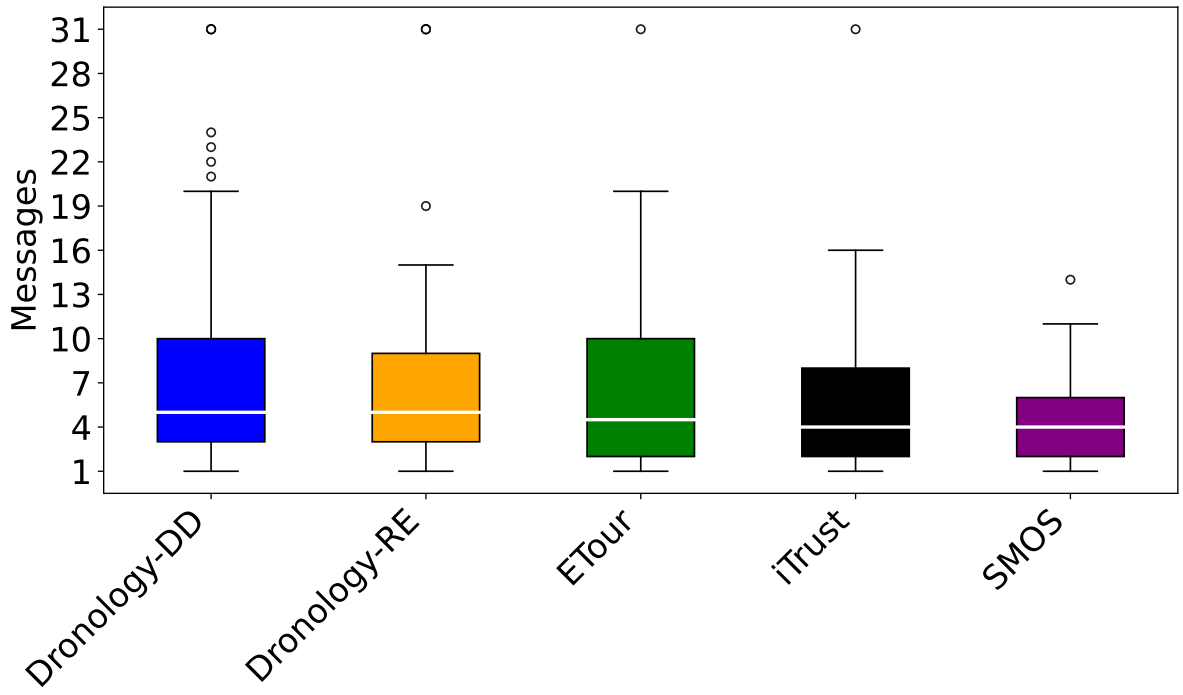


Figure 6.5.: Box plot of the number of messages the LLM sends to the application when execution the agent with all implemented tools on each of the datasets

tion in their requirements, the agent tends to return on average about three related classes per requirement. This may be a consequence of the example provided in the prompt, which illustrates an expected output consisting of three example classes.

To discuss the number of messages and their relation to accurate TLR, we again use our results on Dronology-DD as an example. Figure 6.4 shows a histogram of the number of messages exchanged during the executions of our agent on Dronology-DD, distinguishing between executions that produced at least one correct TL, no correct TLs, and those that did not return any TLs. Notably, 23 out of 211 executions end after only two messages have been exchanged, meaning the agent receives its task and directly outputs a list of related classes, triggering the exit condition in its response. In many of these cases, the agent lists only classes that are not linked to the requirement by a TL, and in some instances, the LLM appears to misunderstand its task and immediately terminates. However, a substantially larger portion of such short exchanges still produce at least one accurate TL. The majority of related classes, however, are returned only after a longer exchange of messages. There is no clear connection between the number of exchanged messages and whether the execution returns at least one accurate TL.

Finally, we compare the agent's performance on each dataset when provided with the different sets of tools described in Paragraph 6.2.1. The resulting precision, recall, and combined F_1- and F_2-scores shown in Table 6.3 suggest that the ability to use tools to search or query the project's source code, beyond directly accessing it through the `SourceCode` tool, is not beneficial. Providing the agent with no tools, or only the `SourceCode` tool, yields the highest precision and recall values. The one exception is again SMOS, likely due to the large number of TLs present in that dataset. In general, the availability of additional tools appears to decrease recall while not reliably increasing precision. Across all datasets, the differences between the maximum and minimum precision and recall values are small, not exceeding 0.075 (SMOS) and 0.064 (Dronology-RE).

Table 6.2.: Results across datasets for the agent with the full tool set

Dataset	Exec.	Rel. Classes	Precision	Recall	F_1-Score	F_2-Score
Dronology-DD	211	665	0.152	0.140	0.146	0.142
Dronology-RE	99	344	0.169	0.099	0.125	0.108
ETour	58	155	0.387	0.195	0.259	0.216
iTrust	131	381	0.215	0.287	0.246	0.269
SMOS	67	197	0.726	0.139	0.234	0.166

The detrimental influence of tools on the quality of recovered TLs does not stem from the agent failing to use them.
As shown in Figure 6.3, this is not the case for the full tool set, and we provide analogous plots and the underlying data in Section C.1. Rather, the issue may lie in how the agent uses the available tools. As discussed earlier, when tools are available, the agent tends to use them to efficiently scan the project, then employs the `SourceCode` tool to examine the identified classes. For configurations where the `SourceCode` tool is not available, the Summarizer tool fulfills this role. This usage pattern by itself does not seem problematic.
However, as shown in Table 6.4, for each configuration aggregated across all datasets, the majority of the classes whose source code is accessed—whether to be summarized, queried, or directly returned to the LLM—are not related to the input requirement, yet they are often returned as related by the agent. Conversely, the classes that are genuinely related to the requirement form only a minority of those accessed by these tools.This is partly due to the smaller overall number of related classes compared to unrelated ones, but also because a substantial number of accessed related classes are ultimately rejected, i.e., not returned as related by the agent. It is notable that, for all configurations except the one providing only the `SourceCode` tool, the percentage of classes that are accessed but neither related nor returned is greater than that of accurately classified related classes.
This indicates that the agent is very likely to consider accessed classes as related, regardless of whether this is true. In essence, the agent follows a similar two-stage approach to LiSSA: first retrieving candidate classes, then classifying them.
However, it performs this process with far less accurate and sophisticated retrieval techniques—searching only for classes that exactly match a query string—and using zero-shot classification without an explicit prompt, often over extensive unrelated text that diminishes its reasoning ability.

In conclusion, the agent appears overly willing to accept classes it has examined as related. Combined with the `Finder` tool's tendency to present potentially unrelated classes and the agent's bias toward returning only a small number of classes per execution, this leads to both low precision and low recall. Thus, the availability of `Finder` tools is not beneficial but instead actively detrimental to the agent's performance.
When the `Finder` tool is unavailable, this problematic usage pattern cannot occur; however, the agent then lacks an effective means to locate classes whose names or packages do not indicate relatedness. The high number of previously accessed false positives further indicates that the agent cannot reliably assess whether a class is related, and tends to consider it as such by default.

Table 6.3.: Evaluation results across datasets and tool sets.

Dataset	Toolset	Precision	Recall	F_1-Score	F_2-Score
Dronology-DD	All Tools	0.152	0.140	0.146	0.142
	- except `SourceCode`	0.120	0.112	0.116	0.114
	`SourceCode`, `Finder`, `Retriever`	0.151	0.149	0.150	0.149
	`SourceCodee`	**0.174**	0.188	**0.180**	0.185
	None	0.163	**0.196**	0.178	**0.188**
Dronology-RE	All Tools	0.169	0.099	0.125	0.108
	- except `SourceCode`	0.141	0.163	0.151	0.158
	`SourceCode`, `Finder`, `Retriever`	0.165	0.107	0.130	0.115
	`SourceCode`	0.218	0.154	0.181	0.164
	None	**0.220**	**0.163**	**0.188**	**0.172**
ETour	All Tools	0.387	0.195	0.259	0.216
	- except `SourceCode`	0.379	0.224	0.282	0.244
	`SourceCode`, `Finder`, `Retriever`	**0.406**	0.231	0.294	0.252
	`SourceCode`	0.382	**0.256**	**0.307**	**0.274**
	None	0.366	0.240	0.290	0.258
iTrust	All Tools	0.215	0.287	0.246	0.269
	- except `SourceCode`	0.174	0.257	0.207	0.234
	`SourceCode`, `Finder`, `Retriever`	0.196	0.287	0.233	0.262
	`SourceCode`	**0.225**	0.357	**0.276**	**0.319**
	None	0.186	**0.364**	0.246	0.305
SMOS	All Tools	**0.726**	0.139	0.234	0.166
	- except `SourceCode`	0.683	0.126	0.212	0.150
	`SourceCode`, `Finder`, `Retriever`	0.688	0.125	0.211	0.149
	`SourceCode`	0.645	0.132	0.220	0.157
	None	0.648	**0.145**	**0.237**	**0.172**

Table 6.4.: The number of classes used as tool arguments across the evaluation datasets and the number of these classes related and returned but unrelated to the execution's input requirement, i.e. linked by TL and not linked by a TL

Toolset	Total	True Positives(%)	Rejected (%)	False Positives (%)
All	2866	17.7	5.3	51.4
-Except `SourceCode`	2599	16.3	4.8	51.8
`SourceCode`, `Finder`, `Retriever`	2653	15.7	5.5	53.4
`SourceCode`	2755	21.6	7.1	63.1

6.3. Conclusion

In this chapter, we have presented an approach to TLR that employs an LLM-based agent. We have introduced a set of tools that the agent can use to lazily access a project's code base and requirements.
Our evaluation shows that, while this agentic approach is capable of locating some of the

classes related to an input requirement and thus recovering TLs, its performance falls short of that achieved by traditional IR-based TLR techniques, as well as the retrieval strategies presented in chapter 5.
We have identified two likely reasons for this performance gap:
First, the LLM controlling the agent's exploration of the code base tends to accept classes it is presented with as being related to the requirement without conducting deeper analysis.
Second, the agent lacks a clear "entry point" into the project. Instead, whenever available, it begins its investigation of the code base by using the finder tool, which essentially acts as an unsophisticated retrieval mechanism that often returns unrelated classes.
Beyond these two patterns leading to poor performance, they also prevent the agent from fully and evenly utilizing all of the available tools. Restricting the agent to tool sets that exclude those enabling this usage pattern does not substantially improve its performance either. This suggests that the sparse, high-level view of the projects—i.e., their class and package structure—is not sufficient to recover more than a small minority of the existing TLs.

7. Conclusion

In this chapter, we address the potential threats to the validity of our results and describe how we have mitigated them. We then provide an overview of the techniques and results presented in this thesis before concluding with a discussion of possible directions for future research on the use of context in automated requirement-to-code TLR.

7.1. Threats to validity

Following the guidelines of Runeson et al. [35], we discuss potential threats to the validity of our results, starting with construct validity.
The central goal of this thesis was to use context to improve the results achieved by LiSSA. For this reason we implemented our techniques as modules that can be used as part of the LiSSA framework. The quality of the produced results can be objectively quantified using the standard metrics of precision, recall, and their harmonic means in the form of F_1 and F_2 scores. Fuchß et al. report these metrics, and we use the same measures to compare our results with those achieved by LiSSA's existing modules. Thus, threats to construct validity are mitigated.
Regarding internal validity, there is the threat that the non-deterministic manner in which LLMs arrive at their outputs may influence our results. To mitigate this threat, outputs from LLMs are cached and reused across different executions of the framework.
We also configured the LLMs with a fixed random seed and low temperature.
As acknowledged in chapter 4, the datasets used in our evaluation may not be representative of all software projects.
We mitigate this risk by employing established and widely used datasets for requirement-to-code TLR and evaluating our techniques on each one.
Finally, to ensure reliability and enable replication, we document and publish the implementations of our techniques.
We also provide a replication package containing both the evaluation datasets used throughout this thesis and the intermediate output data used to produce the reported results.

7.2. Conclusion

In this thesis, we have explored different avenues to improve the quality of the TLs recovered using the LiSSA framework. We have investigated how both LiSSA's retrieval stage and the classification of the retrieved TLs can be enhanced by incorporating the context of source code elements. Regarding retrieval, in Section 5.1 we introduced new preprocessors designed to narrow the similarity gap between natural language requirements and source code elements. We also proposed a retrieval strategy that considers the neighborhood of code elements. Our evaluation of these new LiSSA modules has shown that, while contextual information can

be beneficial, its effect is relatively minor in our implementation and depends heavily on the appropriate choice of preprocessor and retrieval strategy parameters.
Overall, we did not succeed in substantially improving LiSSA's retrieval performance. Since retrieval recall serves as an upper bound on the recall achievable by the complete execution of the framework, we decided to forgo our planned improvements to LiSSA's classifier modules. Instead, based on this finding, we proposed an agentic approach to determining classes related to an input requirement and thus indirectly recovering requirement-to-class TLs. To enable the use of contextual information in this approach, we implemented tools that allow the agent to access the project's code base and requirements.
However, our evaluation has shown that, in its current implementation, this approach performs worse than both traditional IR-based TLR techniques and SOTA methods employing LLMs such as LiSSA.

7.3. Future Work

Due to time constraints, we were not able to explore additional retrieval strategies. Different types of source code element relationships may prove to be more beneficial than the caller–callee relationships we investigated. We also did not explore combining different transformations into a single preprocessor module, i.e., generating both class-level elements and elements representing a class's methods. For all of these approaches, however, the overall low textual similarity between related requirements and source code—resulting from the semantic gap—remains a fundamental issue.
Using an approach similar to USERTRACE [17], which aggregates source code elements into representations at a higher level of abstraction, may bridge this gap more effectively than our preprocessors did. However, given the inherent limitations of textual similarity, we consider our agentic approach to be the most promising direction for further research.
A key issue in our implementation, uncovered during evaluation, can be attributed to the agreeable nature of conversational LLMs. Even when tools are available to the agent, it often errs on the side of considering a requirement–class pair as related, even when the relationship is weak—for example, when they merely share a domain context—which is insufficient to consider them linked by a TL. Different or more focused prompting may alleviate this issue by shifting from the broad concept of "relatedness" toward establishing a more precise and explicit relationship—for instance, determining whether a class represents an entity mentioned in the requirement or whether one of its functions contributes to a described action flow.
In our evaluation, using GPT-5-mini as the LLM controlling the agent did not meaningfully improve the results. Nevertheless, models capable of higher-level reasoning or specifically fine-tuned for factual accuracy or software engineering tasks may be able to more accurately assess which classes are likely candidates or which tools should be used. Combining the two individual approaches to improving TLR—by tasking an LLM-based agent to recover related classes partially based on the results of previous retrieval—may also be worthwhile.

Bibliography

[1] Arshia Akhavan et al. *LinkAnchor: An Autonomous LLM-Based Agent for Issue-to-Commit Link Recovery*. 2025. arXiv: 2508.12232 [cs.SE]. URL: https://arxiv.org/abs/2508.12232.

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

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

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

[5] Ajay Bandi et al. "The Rise of Agentic AI: A Review of Definitions, Frameworks, Architectures, Applications, Evaluation Metrics, and Challenges". In: *Future Internet* 17.9 (2025). ISSN: 1999-5903. DOI: 10.3390/fi17090404. URL: https://www.mdpi.com/1999-5903/17/9/404.

[6] Yoshua Bengio, Yann Lecun, and Geoffrey Hinton. "Deep learning for AI". In: *Commun. ACM* 64.7 (June 2021), pp. 58–65. ISSN: 0001-0782. DOI: 10.1145/3448250. URL: https://doi.org/10.1145/3448250.

[7] Elke Bouillon, Patrick Mäder, and Ilka Philippow. "A Survey on Usage Scenarios for Requirements Traceability in Practice". In: *Requirements Engineering: Foundation for Software Quality*. Ed. by Joerg Doerr and Andreas L. Opdahl. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 158–173. ISBN: 978-3-642-37422-7.

[8] Giannis Chatziveroglou, Richard Yun, and Maura Kelleher. *Exploring LLM Reasoning Through Controlled Prompt Variations*. 2025. arXiv: 2504.02111 [cs.AI].

[9] Dominik Fuchß et al. "Beyond Retrieval: A Study of Using LLM Ensembles for Candidate Filtering in Requirements Traceability". In: *2025 IEEE 33rd International Requirements Engineering Conference Workshops (REW)*. 2025, pp. 5–12. DOI: 10.1109/REW66121.2025.00006.

[10] Dominik Fuchß et al. "LiSSA: Toward Generic Traceability Link Recovery through Retrieval-Augmented Generation". In: *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*. ICSE '25. Ottawa, Ontario, Canada: IEEE Press, 2025, pp. 1396–1408. ISBN: 9798331505691. DOI: 10.1109/ICSE55347.2025.00186. URL: https://doi.org/10.1109/ICSE55347.2025.00186.

[11] Hui Gao et al. “Using Consensual Biterms from Text Structures of Requirements and Code to Improve IR-Based Traceability Recovery”. In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’22. Rochester, MI, USA: Association for Computing Machinery, 2023. ISBN: 9781450394758. DOI: 10.1145/3551349.3556948. URL: https://doi.org/10.1145/3551349.3556948.

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

[13] Jin L. C. Guo et al. “Natural Language Processing for Requirements Traceability”. In: *Handbook on Natural Language Processing for Requirements Engineering*. Springer, 2025. DOI: 10.1007/978-3-031-73143-3_4.

[14] Tobias Hey et al. “Improving Traceability Link Recovery Using Fine-grained Requirements-to-Code Relations”. In: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2021, pp. 12–22. DOI: 10.1109/ICSME52107.2021.00008.

[15] Inar W. Høst and Bjarte M. Østvold. “Debugging Method Names”. In: *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009, Proceedings*. Ed. by Sophia Drossopoulou. Vol. 5653. Lecture Notes in Computer Science. Springer, 2009, pp. 294–317. DOI: 10.1007/978-3-642-03013-0_14. URL: https://doi.org/10.1007/978-3-642-03013-0_14.

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

[17] Dongming Jin et al. *UserTrace: User-Level Requirements Generation and Traceability Recovery from Software Project Repositories*. 2025. arXiv: 2509.11238 [cs.SE]. URL: https://arxiv.org/abs/2509.11238.

[18] Hongyu Kuang et al. “Can method data dependencies support the assessment of traceability between requirements and source code?” In: *J. Softw. Evol. Process* 27.11 (Nov. 2015), pp. 838–866. ISSN: 2047-7473. DOI: 10.1002/smr.1736. URL: https://doi.org/10.1002/smr.1736.

[19] Matt Kusner et al. “From Word Embeddings To Document Distances”. In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. 7–9 July. Lille, France: PMLR, 2015, pp. 957–966. URL: https://proceedings.mlr.press/v37/kusnerb15.html.

[20] Mosh Levy, Alon Jacoby, and Yoav Goldberg. “Same Task, More Tokens: the Impact of Input Length on the Reasoning Performance of Large Language Models”. In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2024, pp. 15339–15353. DOI: 10.18653/v1/2024.acl-long.818.

[21] Patrick Lewis et al. “Retrieval-augmented generation for knowledge-intensive NLP tasks”. In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. NIPS ’20. Vancouver, BC, Canada: Curran Associates Inc., 2020. ISBN: 9781713829546.

[22] A. Marcus and J.I. Maletic. “Recovering documentation-to-source-code traceability links using latent semantic indexing”. In: *25th International Conference on Software Engineering, 2003. Proceedings.* 2003, pp. 125–135. DOI: 10.1109/ICSE.2003.1201194.

[23] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. 1st. See p. 61 for discussion of the Single Responsibility Principle and the idea that each function or module should do one thing well. USA: Prentice Hall Press, 2017, p. 61. ISBN: 0134494164.

[24] Lachlan McGinness et al. "Highlighting Case Studies in LLM Literature Review of Interdisciplinary System Science". In: *AI 2024: Advances in Artificial Intelligence: 37th Australasian Joint Conference on Artificial Intelligence, AI 2024, Melbourne, VIC, Australia, November 25–29, 2024, Proceedings, Part I*. Melbourne, VIC, Australia: Springer-Verlag, 2024, pp. 29–43. DOI: 10.1007/978-981-96-0348-0_3. URL: https://doi.org/10.1007/978-981-96-0348-0_3.

[25] Tomas Mikolov et al. "Efficient Estimation of Word Representations in Vector Space". In: *Proceedings of Workshop at ICLR* 2013 (Jan. 2013).

[26] Chris Mills, Javier Escobar-Avila, and Sonia Haiduc. "Automatic Traceability Maintenance via Machine Learning Classification". In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018, pp. 369–380. DOI: 10.1109/ICSME.2018.00045.

[27] Shervin Minaee et al. *Large Language Models: A Survey*. 2024. arXiv: 2402.06196 [cs.CL].

[28] 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. Seoul, South Korea: Association for Computing Machinery, 2020, pp. 873–885. ISBN: 9781450371216. DOI: 10.1145/3377811.3380418. URL: https://doi.org/10.1145/3377811.3380418.

[29] Daye Nam et al. "Using an LLM to Help With Code Understanding". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. Lisbon, Portugal: Association for Computing Machinery, 2024. ISBN: 9798400702174. DOI: 10.1145/3597503.3639187. URL: https://doi.org/10.1145/3597503.3639187.

[30] Rocco Oliveto et al. "On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery". In: *2010 IEEE 18th International Conference on Program Comprehension*. 2010, pp. 68–71. DOI: 10.1109/ICPC.2010.20.

[31] Annibale Panichella et al. "When and How Using Structural Information to Improve IR-based Traceability Recovery". In: *2013 17th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 2013, pp. 199–208. DOI: 10.1109/CSMR.2013.29.

[32] Michael Rath et al. "Traceability in the wild: automatically augmenting incomplete trace links". In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 834–845. ISBN: 9781450356381. DOI: 10.1145/3180155.3180207. URL: https://doi.org/10.1145/3180155.3180207.

[33] Patrick Rempel and Parick Mäder. "Preventing defects: The impact of requirements traceability completeness on software quality". In: *IEEE Transactions on Software Engineering* 43.8 (2016), pp. 777–797.

[34] Alberto D. Rodriguez, Katherine R. Dearstyne, and Jane Cleland-Huang. "Prompts Matter: Insights and Strategies for Prompt Engineering in Automated Software Traceability". In: *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*. 2023. DOI: 10.1109/REW57809.2023.00087.

[35] Per Runeson and Martin Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical Software Engineering* 14.2 (2009), pp. 131–164. ISSN: 1573-7616. DOI: 10.1007/s10664-008-9102-8. URL: https://doi.org/10.1007/s10664-008-9102-8.

[36] Bangchao Wang et al. "An empirical study on the state-of-the-art methods for requirement-to-code traceability link recovery". In: *Journal of King Saud University - Computer and Information Sciences* 36.6 (2024), p. 102118. ISSN: 1319-1578. DOI: https://doi.org/10.1016/j.jksuci.2024.102118. URL: https://www.sciencedirect.com/science/article/pii/S1319157824002076.

[37] Lei Wang et al. "A survey on large language model based autonomous agents". In: *Frontiers of Computer Science* 18.6 (Mar. 2024), p. 186345. ISSN: 2095-2236. DOI: 10.1007/s11704-024-40231-1. URL: https://doi.org/10.1007/s11704-024-40231-1.

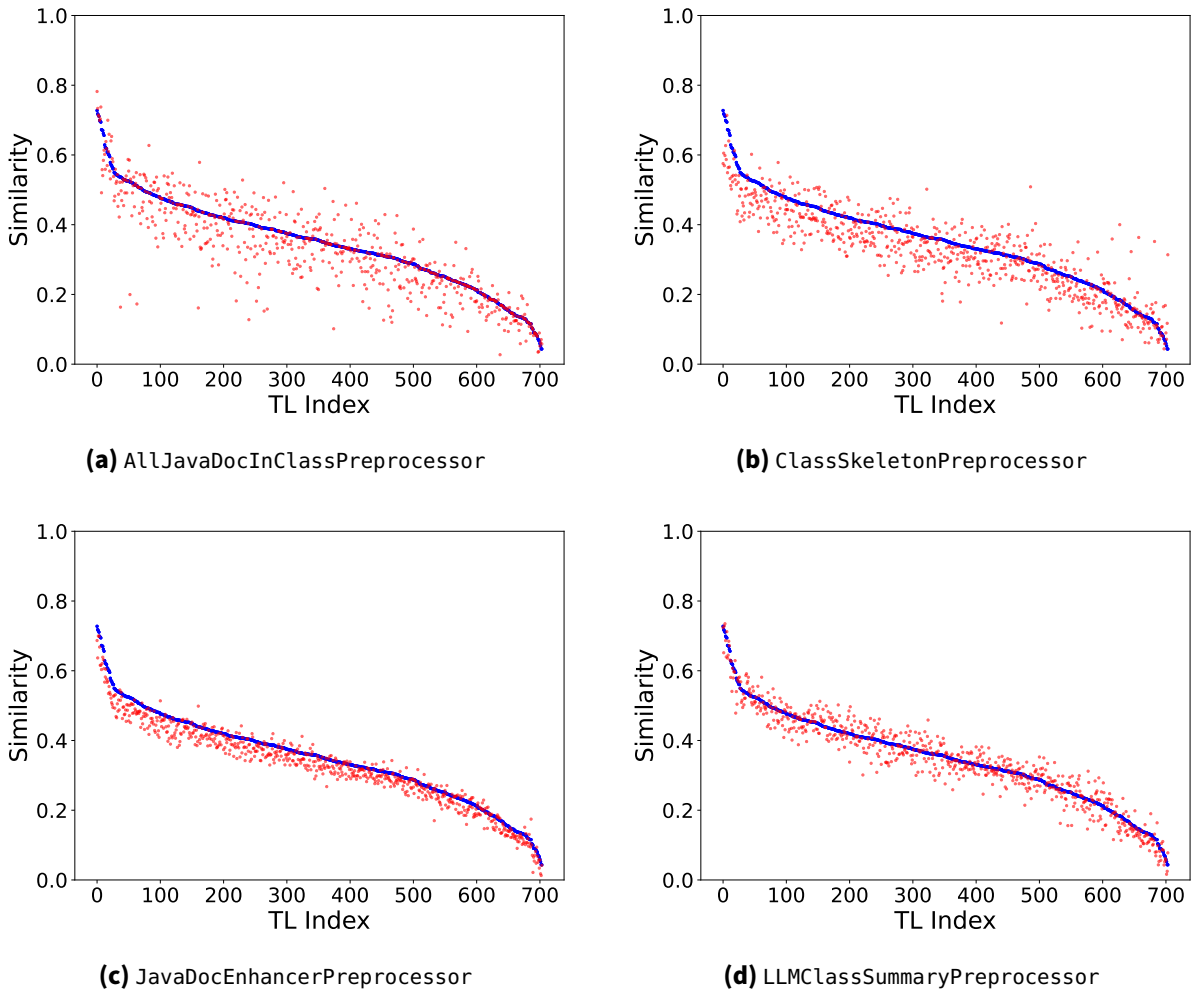
[38] Zhiyuan Zou et al. *Natural Language-Programming Language Software Traceability Link Recovery Needs More than Textual Similarity*. 2025. arXiv: 2509.05585 [cs.SE]. URL: https://arxiv.org/abs/2509.05585.

A. Appendix

A.1. Gold-Standard Link Element Similarities and Ranks

B. Preprocessors

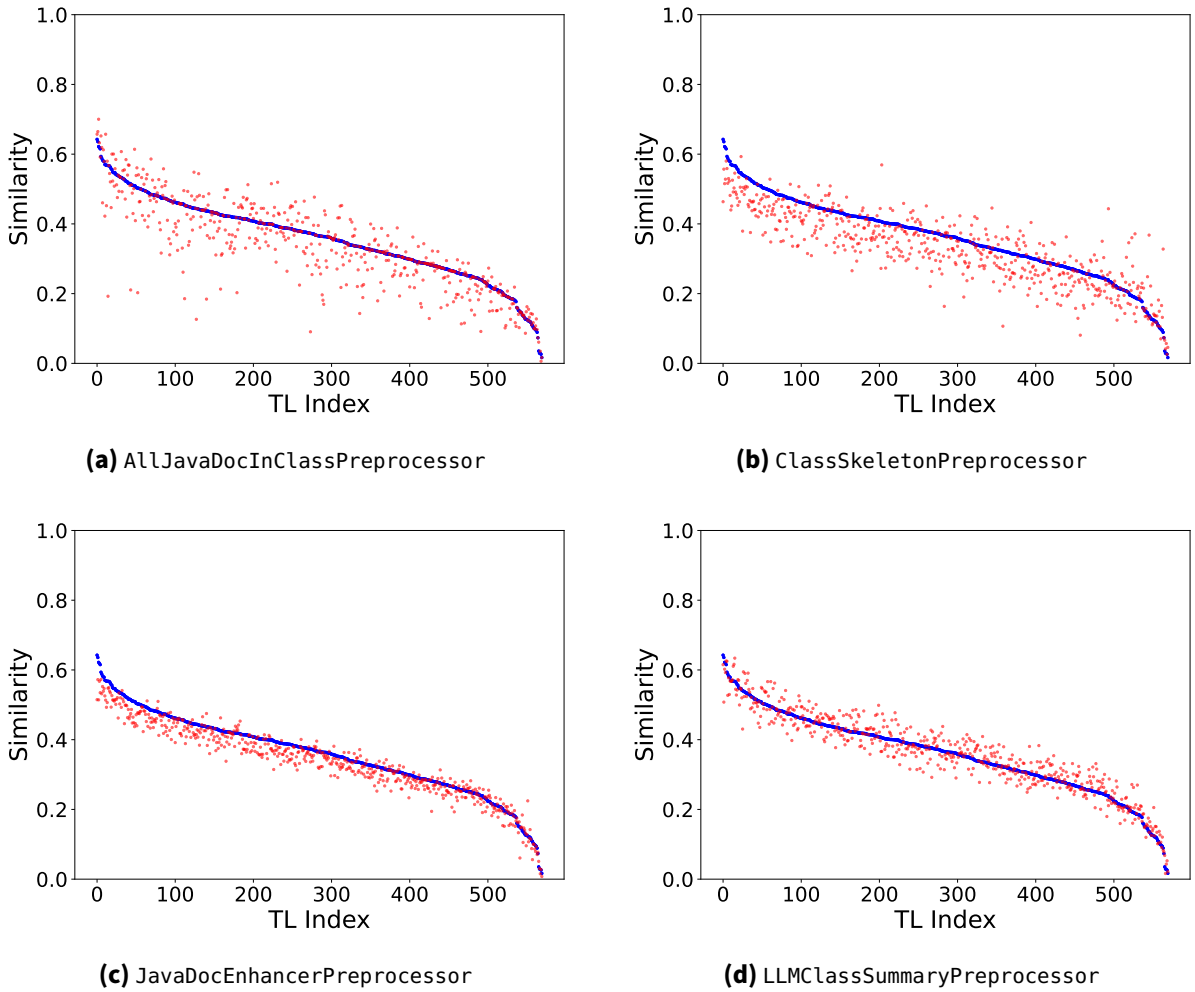
B.1. Gold-Standard Artifact Similarity and Retrieval Rank Modifications by Preprocessors



(a) `AllJavaDocInClassPreprocessor` **(b)** `ClassSkeletonPreprocessor`

(c) `JavaDocEnhancerPreprocessor` **(d)** `LLMClassSummaryPreprocessor`

Figure B.1.: Pairwise similarity of elements linked by gold-standard TLs using class-level preprocessors. Blue markers indicate baseline similarity between unprocessed source and target elements, while red markers indicate similarity after preprocessing. Gold-standard TLs are ordered by baseline similarity for comparability across preprocessors. on Dronology-DD dataset



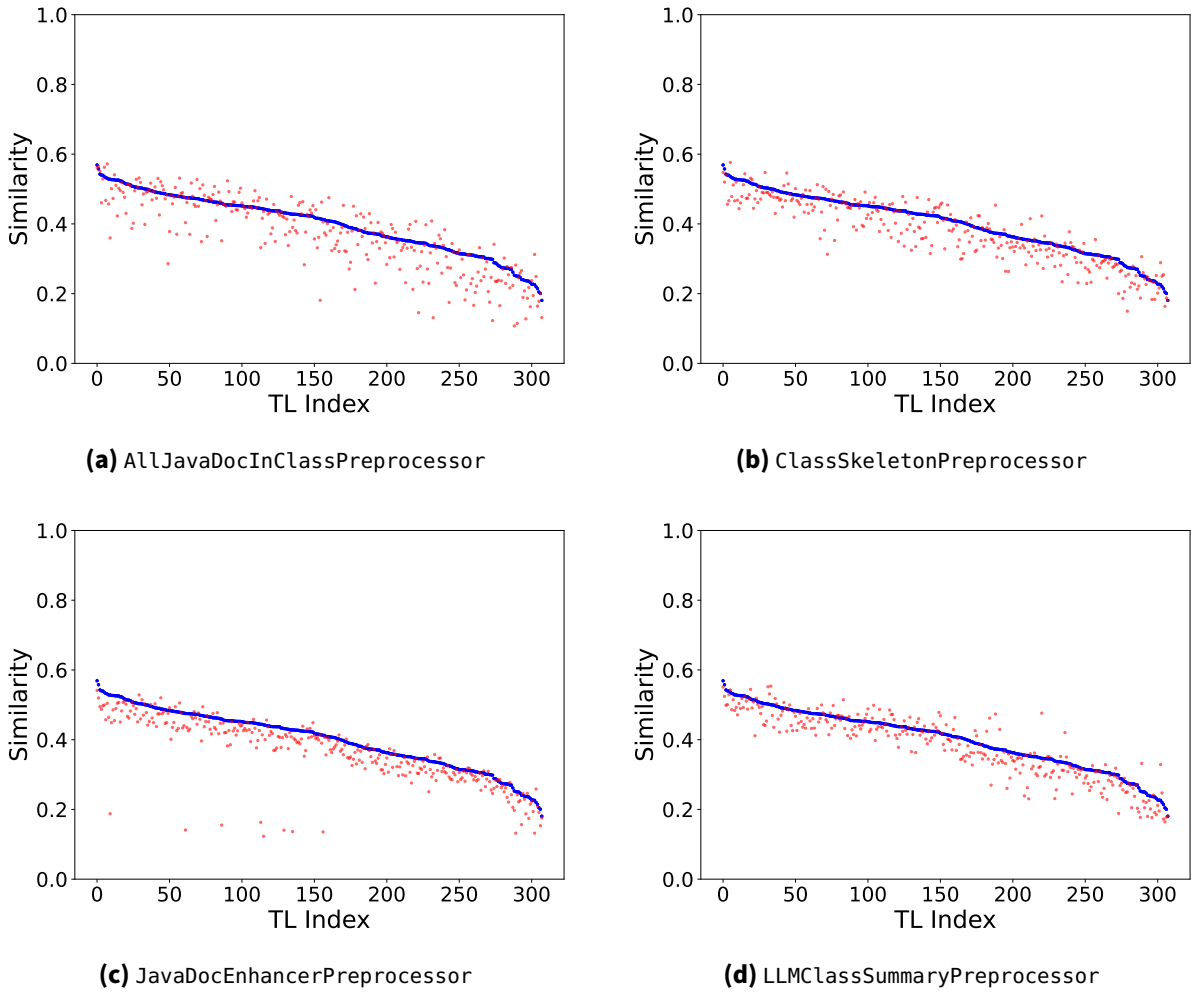
(a) `AllJavaDocInClassPreprocessor`

(b) `ClassSkeletonPreprocessor`

(c) `JavaDocEnhancerPreprocessor`

(d) `LLMClassSummaryPreprocessor`

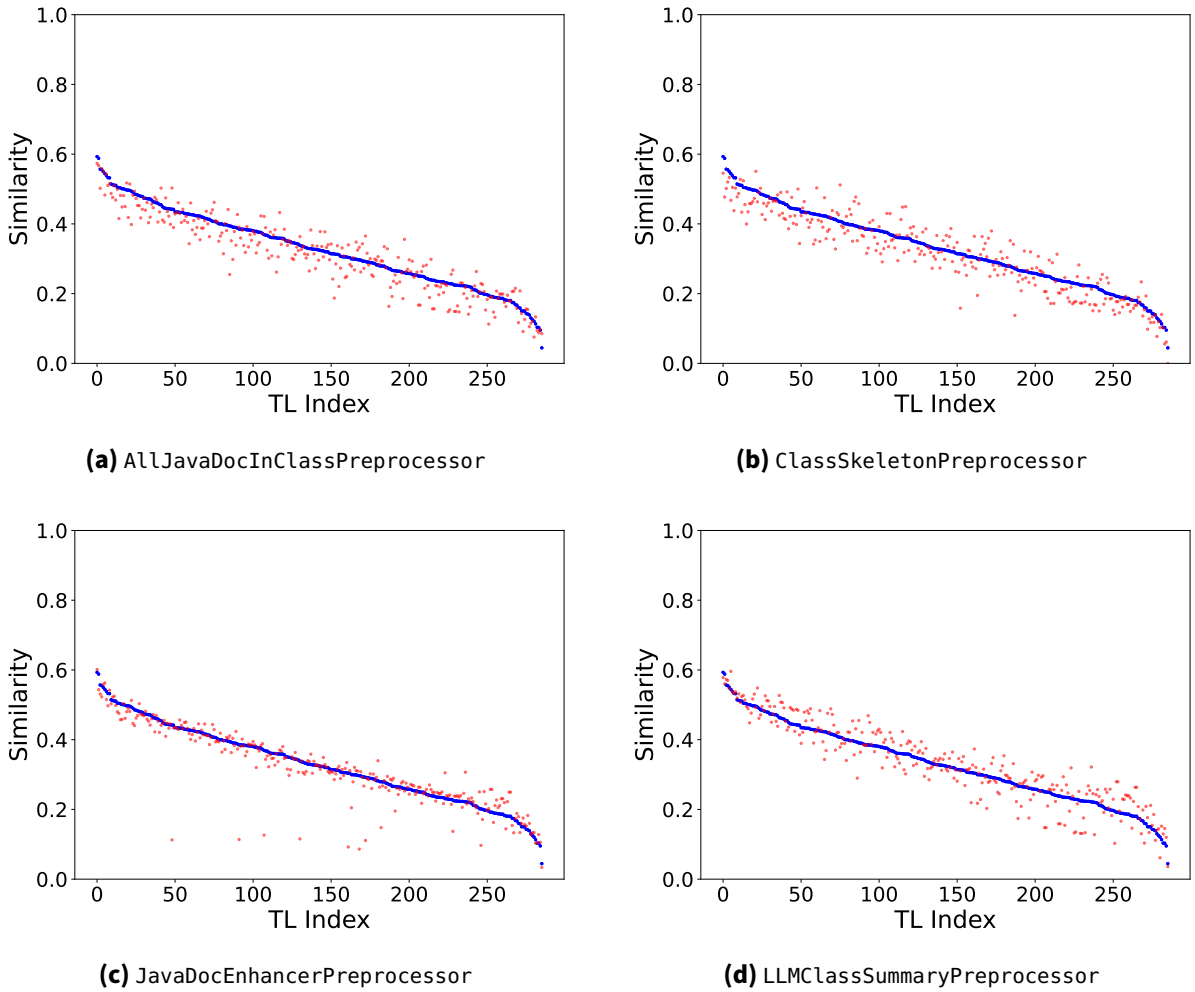
Figure B.2.: Pairwise similarity of elements linked by gold-standard TLs using class-level preprocessors. Blue markers indicate baseline similarity between unprocessed source and target elements, while red markers indicate similarity after preprocessing. Gold-standard TLs are ordered by baseline similarity for comparability across preprocessors. on Dronology-RE dataset



(a) `AllJavaDocInClassPreprocessor` **(b)** `ClassSkeletonPreprocessor`

(c) `JavaDocEnhancerPreprocessor` **(d)** `LLMClassSummaryPreprocessor`

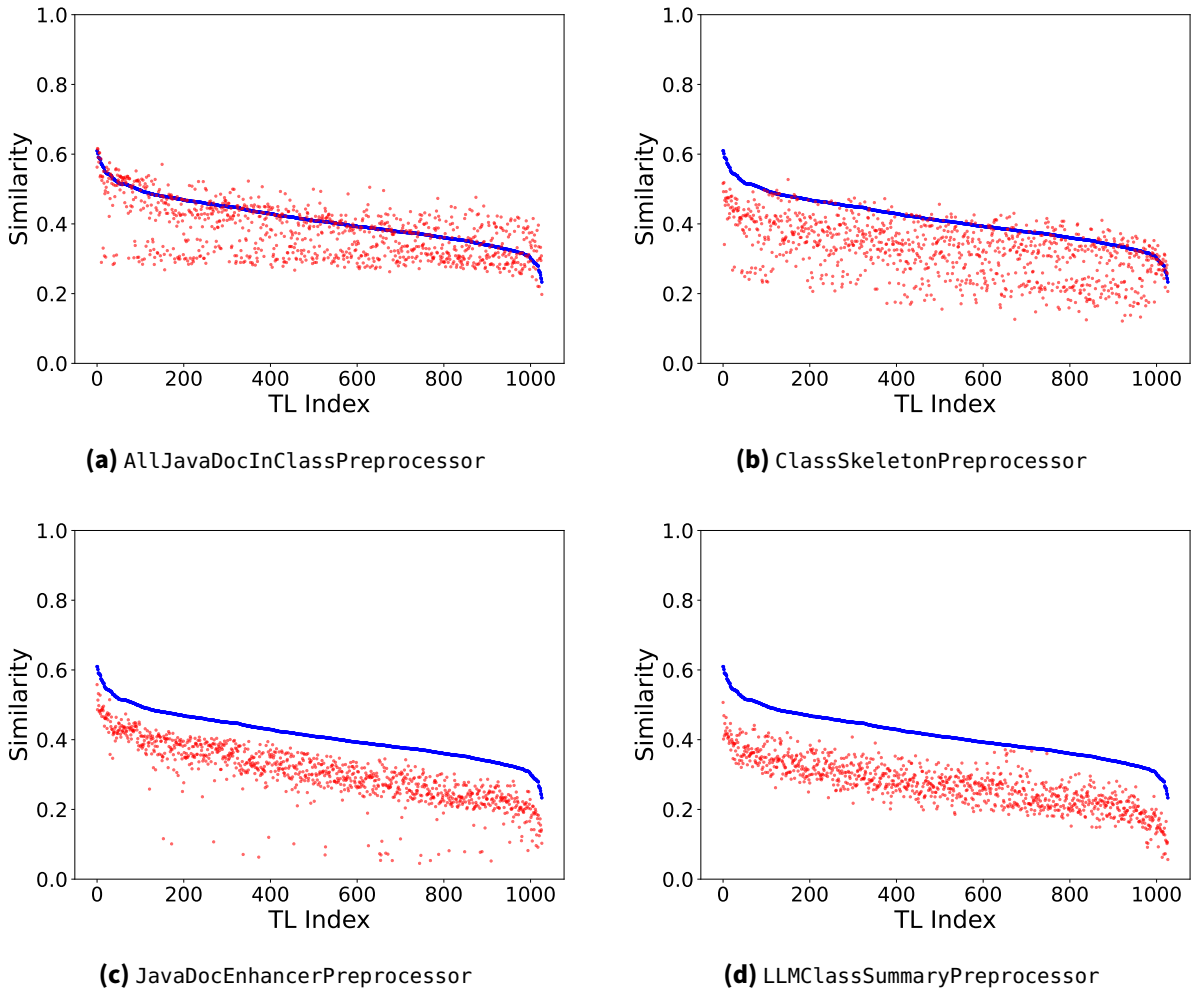
Figure B.3.: Pairwise similarity of elements linked by gold-standard TLs using class-level preprocessors. Blue markers indicate baseline similarity between unprocessed source and target elements, while red markers indicate similarity after preprocessing. Gold-standard TLs are ordered by baseline similarity for comparability across preprocessors. on ETour dataset



(a) `AllJavaDocInClassPreprocessor` **(b)** `ClassSkeletonPreprocessor`

(c) `JavaDocEnhancerPreprocessor` **(d)** `LLMClassSummaryPreprocessor`

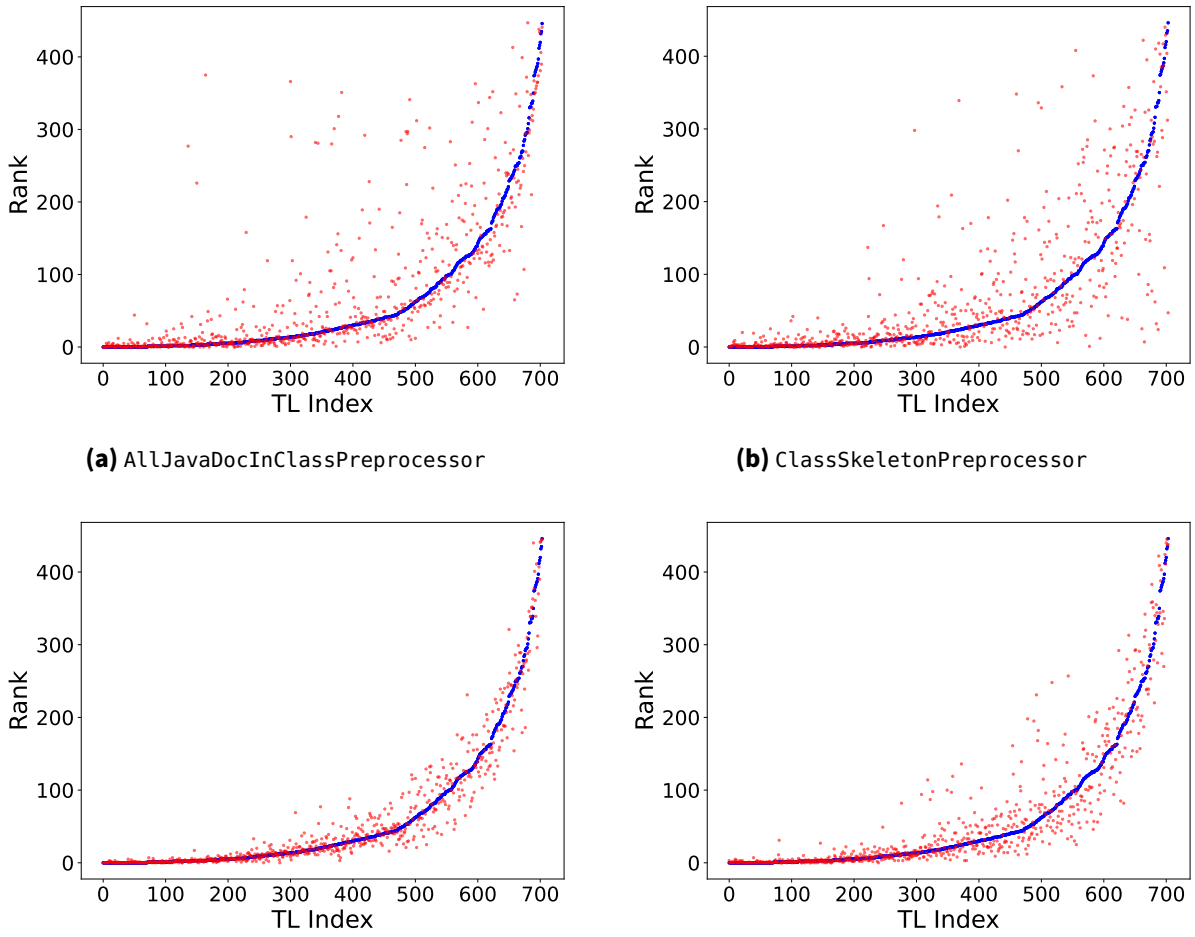
Figure B.4.: Pairwise similarity of elements linked by gold-standard TLs using class-level preprocessors. Blue markers indicate baseline similarity between unprocessed source and target elements, while red markers indicate similarity after preprocessing. Gold-standard TLs are ordered by baseline similarity for comparability across preprocessors. on iTrust dataset



(a) `AllJavaDocInClassPreprocessor` **(b)** `ClassSkeletonPreprocessor`

(c) `JavaDocEnhancerPreprocessor` **(d)** `LLMClassSummaryPreprocessor`

Figure B.5.: Pairwise similarity of elements linked by gold-standard TLs using class-level preprocessors. Blue markers indicate baseline similarity between unprocessed source and target elements, while red markers indicate similarity after preprocessing. Gold-standard TLs are ordered by baseline similarity for comparability across preprocessors. on SMOS dataset



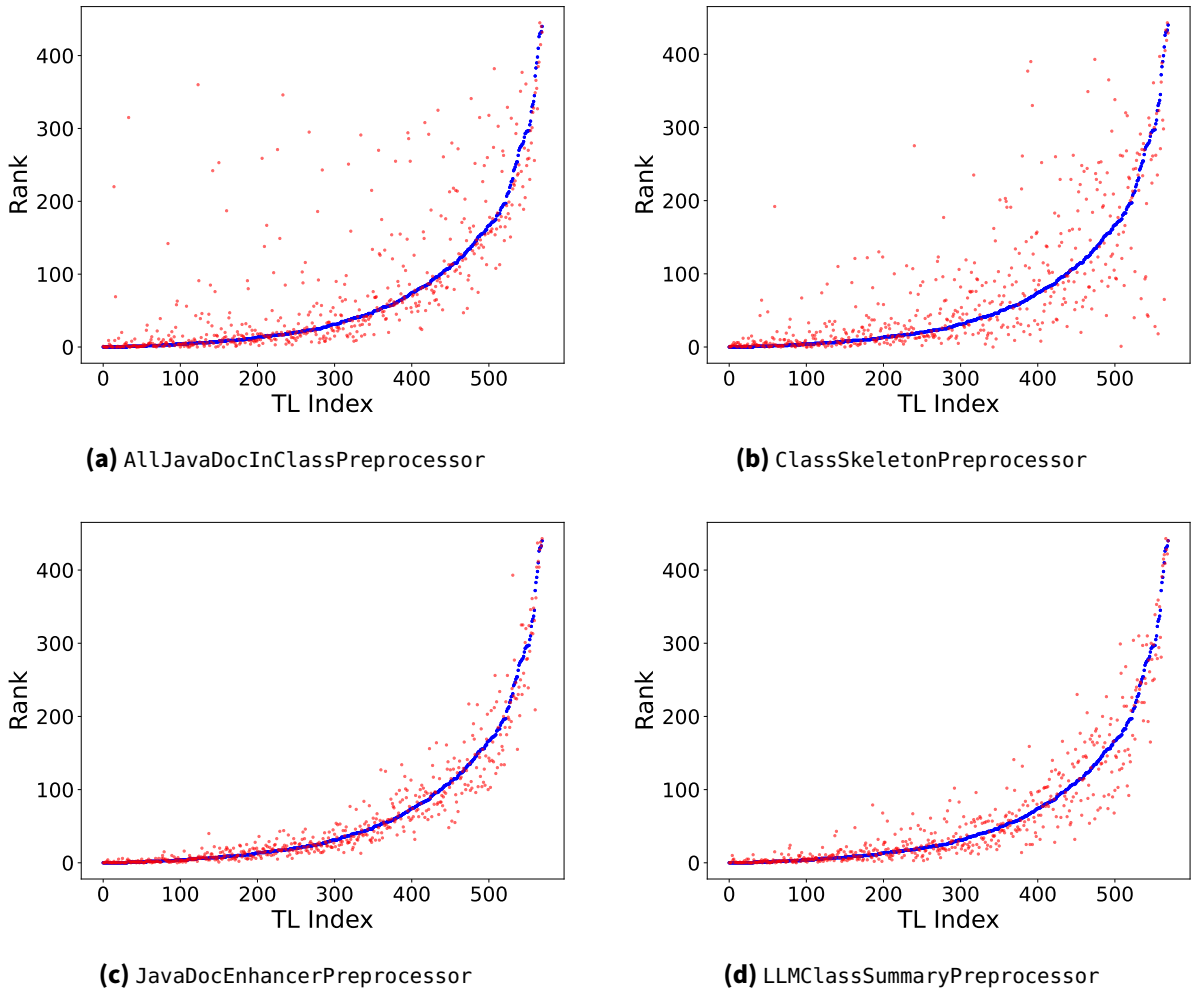
(a) `AllJavaDocInClassPreprocessor`

(b) `ClassSkeletonPreprocessor`

(c) `JavaDocEnhancerPreprocessor`

(d) `LLMClassSummaryPreprocessor`

Figure B.6.: Ranking, i.e. index of the correct target element when using the sorce element for retrieval for each pair of elements linked by gold-standard TLs using class-level preprocessors. Blue markers indicate baseline rank of unprocessed source and target elements, while red markers indicate ranks after preprocessing all target elements. Lower rank values implicate higher retrieval performance, with all pairs of rank smaller than k being retrieved by configurations using a configuration with this value of k, the respective preprocessor and LiSSA's default retrieval strategy. on Dronology-DD dataset



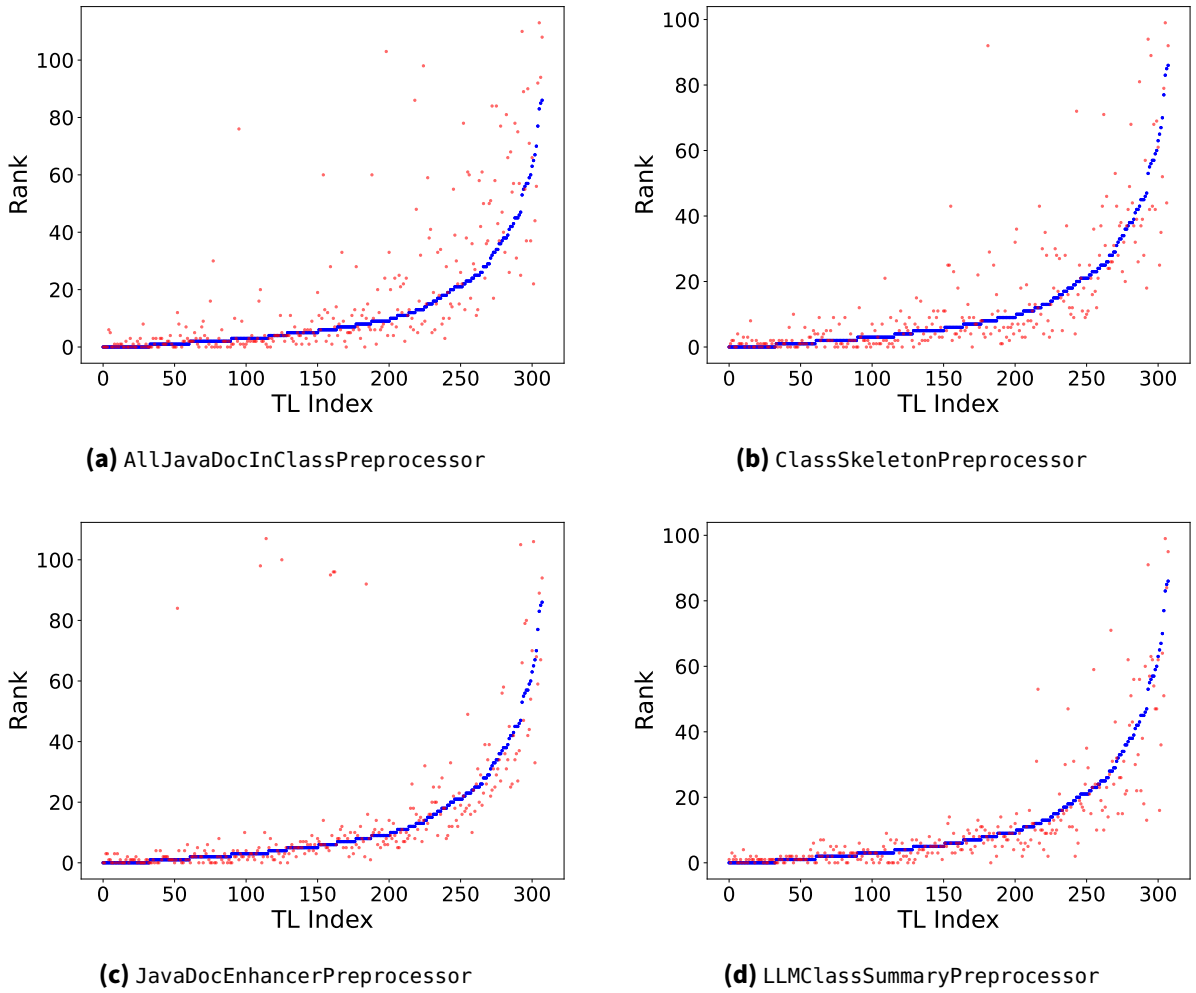
(a) `AllJavaDocInClassPreprocessor`

(b) `ClassSkeletonPreprocessor`

(c) `JavaDocEnhancerPreprocessor`

(d) `LLMClassSummaryPreprocessor`

Figure B.7.: Ranking, i.e. index of the correct target element when using the sorce element for retrieval for each pair of elements linked by gold-standard TLs using class-level preprocessors. Blue markers indicate baseline rank of unprocessed source and target elements, while red markers indicate ranks after preprocessing all target elements. Lower rank values implicate higher retrieval performance, with all pairs of rank smaller than k being retrieved by configurations using a configuration with this value of k, the respective preprocessor and LiSSA's default retrieval strategy. on Dronology-RE dataset



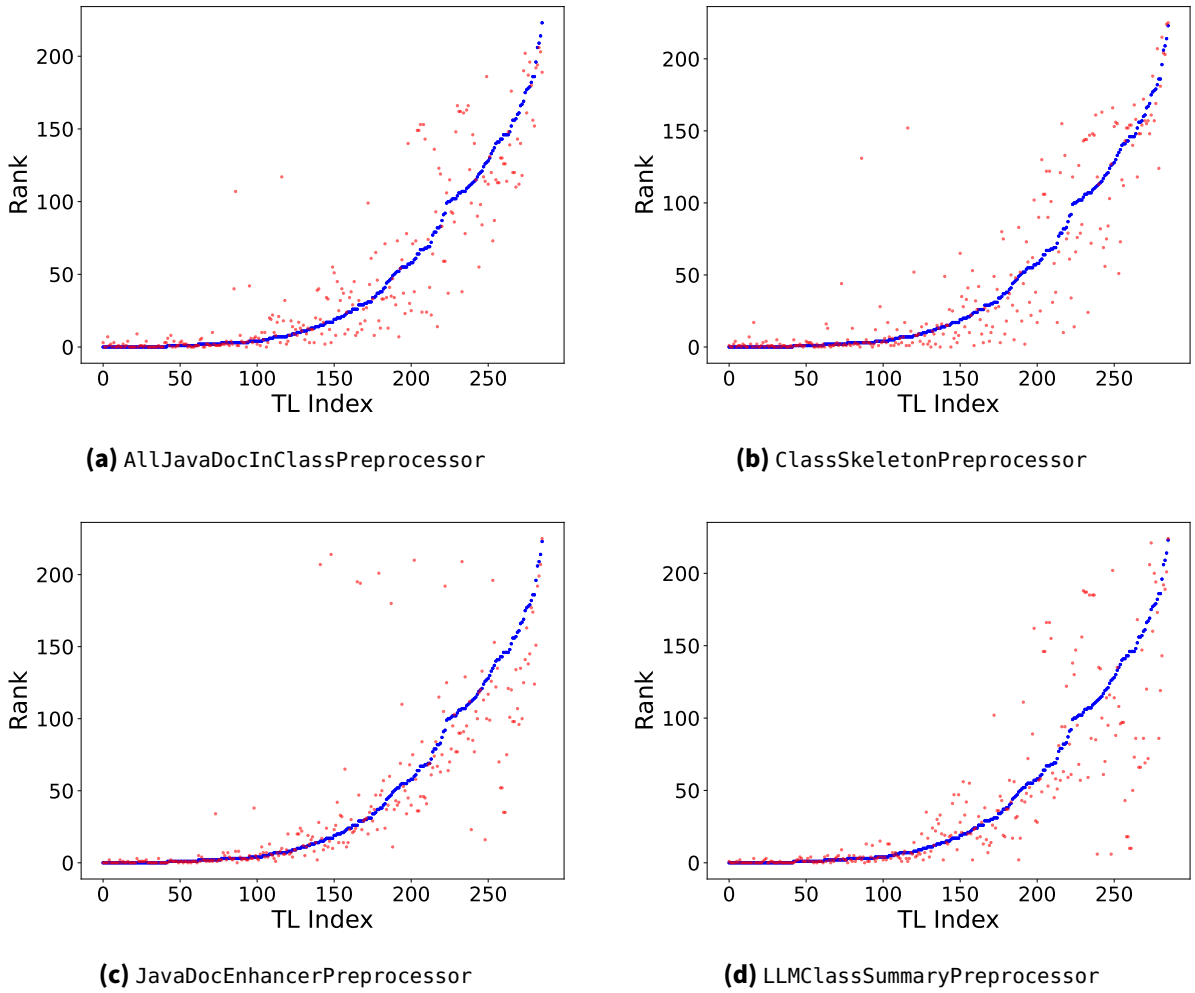
(a) `AllJavaDocInClassPreprocessor`

(b) `ClassSkeletonPreprocessor`

(c) `JavaDocEnhancerPreprocessor`

(d) `LLMClassSummaryPreprocessor`

Figure B.8.: Ranking, i.e. index of the correct target element when using the sorce element for retrieval for each pair of elements linked by gold-standard TLs using class-level preprocessors. Blue markers indicate baseline rank of unprocessed source and target elements, while red markers indicate ranks after preprocessing all target elements. Lower rank values implicate higher retrieval performance, with all pairs of rank smaller than k being retrieved by configurations using a configuration with this value of k, the respective preprocessor and LiSSA's default retrieval strategy. on ETour dataset



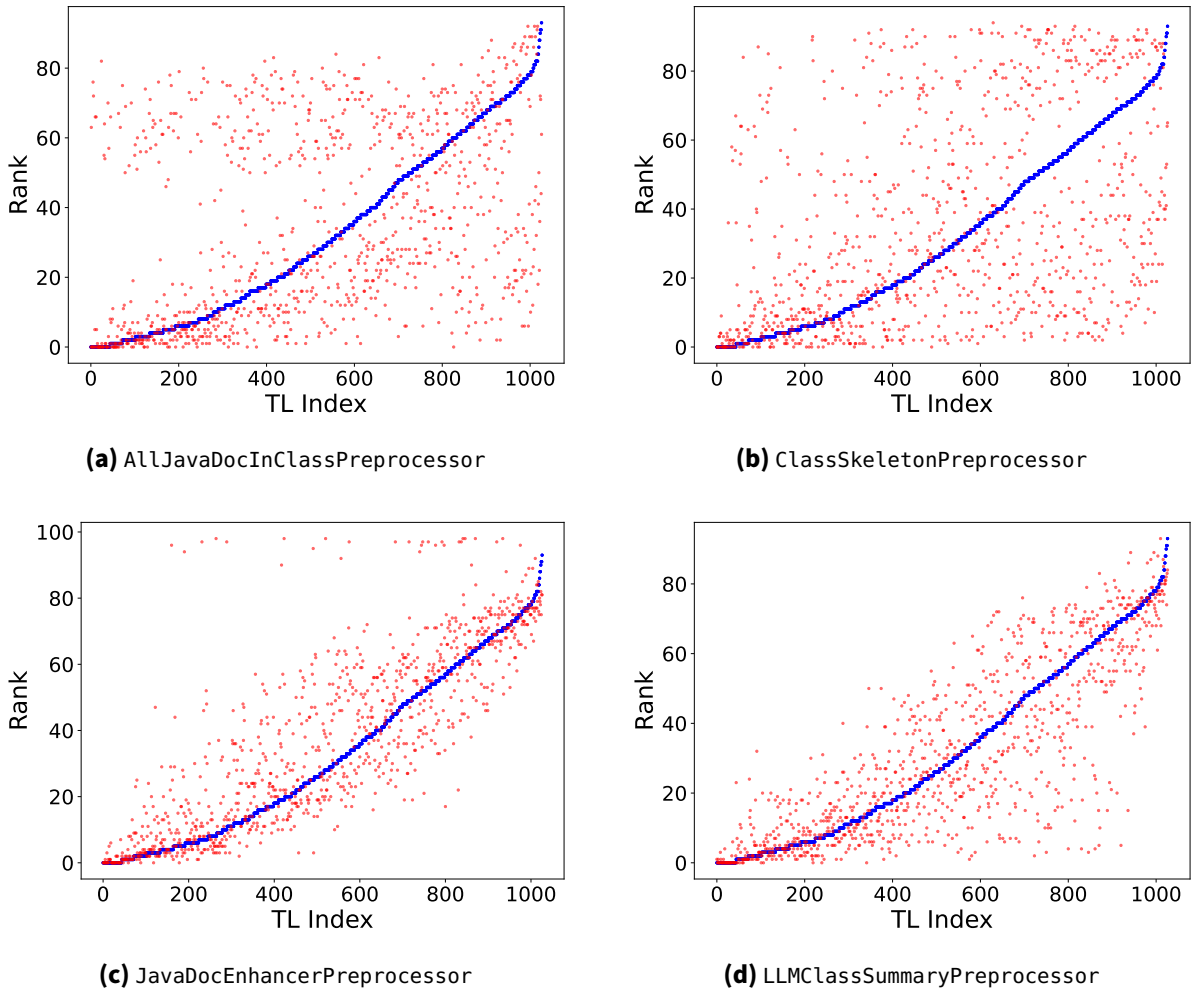
(a) `AllJavaDocInClassPreprocessor`

(b) `ClassSkeletonPreprocessor`

(c) `JavaDocEnhancerPreprocessor`

(d) `LLMClassSummaryPreprocessor`

Figure B.9.: Ranking, i.e. index of the correct target element when using the sorce element for retrieval for each pair of elements linked by gold-standard TLs using class-level preprocessors. Blue markers indicate baseline rank of unprocessed source and target elements, while red markers indicate ranks after preprocessing all target elements. Lower rank values implicate higher retrieval performance, with all pairs of rank smaller than k being retrieved by configurations using a configuration with this value of k, the respective preprocessor and LiSSA's default retrieval strategy. on iTrust dataset



(a) `AllJavaDocInClassPreprocessor`

(b) `ClassSkeletonPreprocessor`

(c) `JavaDocEnhancerPreprocessor`

(d) `LLMClassSummaryPreprocessor`

Figure B.10.: Ranking, i.e. index of the correct target element when using the sorce element for retrieval for each pair of elements linked by gold-standard TLs using class-level preprocessors. Blue markers indicate baseline rank of unprocessed source and target elements, while red markers indicate ranks after preprocessing all target elements. Lower rank values implicate higher retrieval performance, with all pairs of rank smaller than k being retrieved by configurations using a configuration with this value of k, the respective preprocessor and LiSSA's default retrieval strategy. on SMOS dataset

C. Agentic Classification

C.1. Agent Configuration Tool Uses

C.2. Jaccard Indices of Retrieved TLs by preprocessor and dataset

Table C.1.: Aggregated Jaccard similarity intervals between true-positive sets for SMOS (k20, matching strategies only).

Preprocessor	AllJavaDocInClass	MethodJavadoc	ClassSkeleton	ToMethods	ToMethodContext	LLMClassSummary	SingleArtifact	JavaDocEnhancer
MethodJavadoc	[0.39, 0.40]							
ClassSkeleton	[0.47, 0.47]	[0.37, 0.37]						
ToMethods	[0.42, 0.42]	[0.54, 0.55]	[0.49, 0.50]					
ToMethodContext	[0.41, 0.41]	[0.50, 0.50]	[0.49, 0.49]	[0.89, 0.90]				
LLMClassSummary	[0.42, 0.42]	[0.59, 0.60]	[0.49, 0.49]	[0.51, 0.51]	[0.48, 0.48]			
SingleArtifact	[0.49, 0.49]	[0.62, 0.63]	[0.50, 0.50]	[0.56, 0.57]	[0.53, 0.53]	[0.70, 0.70]		
JavaDocEnhancer	[0.47, 0.47]	[0.56, 0.57]	[0.42, 0.42]	[0.44, 0.45]	[0.41, 0.41]	[0.57, 0.57]	[0.67, 0.67]	

Table C.2.: Aggregated Jaccard similarity intervals between true-positive sets for iTrust (k20, matching strategies only).

Preprocessor	MethodJavadoc	SingleArtifact	ToMethods	ClassSkeleton	ToMethodContext	AllJavaDocInClass	JavaDocEnhancer	LLMClassSummary
SingleArtifact	[0.73, 0.75]							
ToMethods	[0.85, 0.87]	[0.77, 0.79]						
ClassSkeleton	[0.70, 0.72]	[0.83, 0.83]	[0.77, 0.78]					
ToMethodContext	[0.84, 0.87]	[0.78, 0.79]	[0.98, 0.99]	[0.77, 0.78]				
AllJavaDocInClass	[0.70, 0.72]	[0.83, 0.83]	[0.75, 0.77]	[0.85, 0.85]	[0.75, 0.76]			
JavaDocEnhancer	[0.72, 0.74]	[0.88, 0.88]	[0.74, 0.76]	[0.82, 0.82]	[0.75, 0.76]	[0.77, 0.77]		
LLMClassSummary	[0.69, 0.71]	[0.81, 0.81]	[0.72, 0.74]	[0.75, 0.75]	[0.72, 0.73]	[0.76, 0.76]	[0.78, 0.78]	

Table C.3.: Aggregated Jaccard similarity intervals between true-positive sets for dronology-dd (k20, matching strategies only).

Preprocessor	LLMClassSummary	SingleArtifact	ToMethods	MethodJavadoc	ToMethodContext	JavaDocEnhancer	AllJavaDocInClass	ClassSkeleton
SingleArtifact	[0.84, 0.88]							
ToMethods	[0.58, 0.64]	[0.62, 0.66]						
MethodJavadoc	[0.61, 0.66]	[0.64, 0.67]	[0.73, 0.81]					
ToMethodContext	[0.60, 0.64]	[0.64, 0.65]	[0.77, 0.83]	[0.73, 0.78]				
JavaDocEnhancer	[0.83, 0.88]	[0.88, 0.93]	[0.62, 0.68]	[0.62, 0.68]	[0.63, 0.67]			
AllJavaDocInClass	[0.56, 0.65]	[0.59, 0.68]	[0.45, 0.55]	[0.46, 0.53]	[0.46, 0.53]	[0.58, 0.65]		
ClassSkeleton	[0.74, 0.79]	[0.73, 0.81]	[0.62, 0.68]	[0.64, 0.68]	[0.62, 0.67]	[0.74, 0.83]	[0.52, 0.62]	

Table C.4.: Aggregated Jaccard similarity intervals between true-positive sets for dronology-re (k20, matching strategies only).

Preprocessor	SingleArtifact	AllJavaDocInClass	LLMClassSummary	ToMethods	ClassSkeleton	MethodJavadoc	ToMethodContext	JavaDocEnhancer
AllJavaDocInClass	[0.54, 0.62]							
LLMClassSummary	[0.78, 0.84]	[0.52, 0.61]						
ToMethods	[0.53, 0.59]	[0.41, 0.48]	[0.50, 0.56]					
ClassSkeleton	[0.62, 0.74]	[0.42, 0.53]	[0.62, 0.74]	[0.58, 0.63]				
MethodJavadoc	[0.55, 0.60]	[0.41, 0.45]	[0.55, 0.60]	[0.70, 0.76]	[0.59, 0.64]			
ToMethodContext	[0.56, 0.64]	[0.45, 0.51]	[0.55, 0.60]	[0.69, 0.77]	[0.53, 0.62]	[0.64, 0.68]		
JavaDocEnhancer	[0.82, 0.91]	[0.49, 0.59]	[0.77, 0.84]	[0.55, 0.60]	[0.66, 0.76]	[0.56, 0.61]	[0.56, 0.65]	

Table C.5.: Aggregated Jaccard similarity intervals between all retrieved links for SMOS (k20, all strategies).

Preprocessor	AllJavaDocInClass	MethodJavadoc	ClassSkeleton	ToMethods	ToMethodContext	LLMClassSummary	SingleArtifact	JavaDocEnhancer
MethodJavadoc	[0.35, 0.35]							
ClassSkeleton	[0.27, 0.27]	[0.28, 0.28]						
ToMethods	[0.24, 0.25]	[0.42, 0.43]	[0.28, 0.28]					
ToMethodContext	[0.23, 0.23]	[0.39, 0.40]	[0.28, 0.28]	[0.85, 0.86]				
LLMClassSummary	[0.34, 0.34]	[0.47, 0.48]	[0.34, 0.34]	[0.33, 0.33]	[0.31, 0.31]			
SingleArtifact	[0.46, 0.46]	[0.51, 0.52]	[0.35, 0.35]	[0.36, 0.36]	[0.33, 0.33]	[0.56, 0.56]		
JavaDocEnhancer	[0.44, 0.44]	[0.42, 0.42]	[0.28, 0.28]	[0.27, 0.27]	[0.24, 0.24]	[0.44, 0.44]	[0.59, 0.59]	

Table C.6.: Aggregated Jaccard similarity intervals between all retrieved links for iTrust (k20, all strategies).

Preprocessor	MethodJavadoc	SingleArtifact	ToMethods	ClassSkeleton	ToMethodContext	AllJavaDocInClass	JavaDocEnhancer	LLMClassSummary
SingleArtifact	[0.30, 0.31]							
ToMethods	[0.53, 0.55]	[0.30, 0.30]						
ClassSkeleton	[0.31, 0.31]	[0.48, 0.48]	[0.31, 0.32]					
ToMethodContext	[0.53, 0.54]	[0.30, 0.30]	[0.82, 0.85]	[0.31, 0.31]				
AllJavaDocInClass	[0.28, 0.28]	[0.47, 0.47]	[0.28, 0.29]	[0.45, 0.45]	[0.28, 0.28]			
JavaDocEnhancer	[0.31, 0.32]	[0.69, 0.69]	[0.30, 0.30]	[0.48, 0.48]	[0.30, 0.30]	[0.42, 0.42]		
LLMClassSummary	[0.30, 0.30]	[0.60, 0.60]	[0.30, 0.30]	[0.47, 0.47]	[0.29, 0.29]	[0.41, 0.41]	[0.61, 0.61]	

Table C.7.: Aggregated Jaccard similarity intervals between all retrieved links for Dronology-DD (k20, all strategies).

Preprocessor	LLMClassSummary	SingleArtifact	ToMethods	MethodJavadoc	ToMethodContext	JavaDocEnhancer	AllJavaDocInClass	ClassSkeleton
SingleArtifact	[0.47, 0.65]							
ToMethods	[0.25, 0.29]	[0.28, 0.31]						
MethodJavadoc	[0.26, 0.30]	[0.29, 0.32]	[0.44, 0.57]					
ToMethodContext	[0.26, 0.29]	[0.28, 0.32]	[0.56, 0.68]	[0.44, 0.52]				
JavaDocEnhancer	[0.47, 0.66]	[0.50, 0.72]	[0.27, 0.31]	[0.29, 0.32]	[0.28, 0.31]			
AllJavaDocInClass	[0.18, 0.28]	[0.19, 0.31]	[0.16, 0.19]	[0.17, 0.19]	[0.16, 0.18]	[0.18, 0.29]		
ClassSkeleton	[0.32, 0.45]	[0.33, 0.47]	[0.26, 0.33]	[0.27, 0.33]	[0.27, 0.32]	[0.34, 0.47]	[0.13, 0.22]	

Table C.8.: Aggregated Jaccard similarity intervals between all retrieved links for Dronology-RE (k20, all strategies).

Preprocessor	SingleArtifact	AllJavaDocInClass	LLMClassSummary	ToMethods	ClassSkeleton	MethodJavadoc	ToMethodContext	JavaDocEnhancer
AllJavaDocInClass	[0.19, 0.32]							
LLMClassSummary	[0.47, 0.67]	[0.19, 0.31]						
ToMethods	[0.26, 0.32]	[0.16, 0.20]	[0.25, 0.29]					
ClassSkeleton	[0.33, 0.48]	[0.14, 0.23]	[0.32, 0.47]	[0.28, 0.33]				
MethodJavadoc	[0.28, 0.35]	[0.16, 0.20]	[0.27, 0.33]	[0.40, 0.55]	[0.28, 0.34]			
ToMethodContext	[0.28, 0.33]	[0.16, 0.19]	[0.26, 0.30]	[0.52, 0.65]	[0.26, 0.33]	[0.41, 0.52]		
JavaDocEnhancer	[0.49, 0.73]	[0.18, 0.30]	[0.45, 0.66]	[0.27, 0.32]	[0.35, 0.49]	[0.29, 0.35]	[0.27, 0.32]	