LiSSA: Toward Generic Traceability Link Recovery through Retrieval-Augmented Generation

Dominik Fuchß, Tobias Hey, Jan Keim, Haoyu Liu, Niklas Ewald, Tobias Thirolf, Anne Koziolek
KASTEL - Institute of Information Security and Dependability
Karlsruhe Institute of Technology (KIT),
Karlsruhe, Germany
{dominik.fuchss, hey, jan.keim, haoyu.liu, koziolek}@kit.edu
niklas.ewald@alumni.kit.edu, tobias.thirolf@student.kit.edu

***Abstract*—There are a multitude of software artifacts which need to be handled during the development and maintenance of a software system. These artifacts interrelate in multiple, complex ways. Therefore, many software engineering tasks are enabled — and even empowered — by a clear understanding of artifact interrelationships and also by the continued advancement of techniques for automated artifact linking.**

However, current approaches in automatic Traceability Link Recovery (TLR) target mostly the links between specific sets of artifacts, such as those between requirements and code. Fortunately, recent advancements in Large Language Models (LLMs) can enable TLR approaches to achieve broad applicability. Still, it is a nontrivial problem how to provide the LLMs with the specific information needed to perform TLR.

In this paper, we present LiSSA, a framework that harnesses LLM performance and enhances them through Retrieval-Augmented Generation (RAG). We empirically evaluate LiSSA on three different TLR tasks, requirements to code, documentation to code, and architecture documentation to architecture models, and we compare our approach to state-of-the-art approaches.

Our results show that the RAG-based approach can significantly outperform the state-of-the-art on the code-related tasks. However, further research is required to improve the performance of RAG-based approaches to be applicable in practice.

***Index Terms*—Traceability Link Recovery, Large Language Models, Retrieval-Augmented Generation**

I. Introduction

In the complex task of software development, developers and other stakeholders handle numerous artifacts, such as requirements, code, documentation, and models. Consequently, success in development depends, in part, on understanding how software artifacts interrelate. To deal with the interrelations of these artifacts, researchers and practitioners actively investigate the creation and recovery of so-called trace links between these artifacts. Traceability Link Recovery (TLR) identifies correspondences between elements in artifacts and makes them explicit. Thus, TLR helps to reduce the complexity of many development tasks, such as change impact analysis [1], [2], bug localization [3], and maintenance [4], [5]. Moreover, trace links can also be used to identify inconsistencies between artifacts [6], [7]. Typically, automated approaches are task-specialized, i.e., they are designed to recover specific types of trace links, such as requirements to code [8]–[14], documentation to code [15], [16], issues to commit [17]–[21], requirements to requirements [1], [22]–[24], or architecture documentation to architecture models [7]. However, the approaches' performances vary depending on the tasks. Often, approaches do not achieve the necessary performance to be used in practice [22], [25], [26].

Large Language Models (LLMs) offer promising capabilities regarding natural language understanding. Furthermore, LLMs have already proven to be effective in a variety of software engineering tasks, including code generation [27]–[30], code summarization [31]–[33], or API recommendation [34]. In particular, LLMs could lead to a generic approach for TLR. Nevertheless, LLMs face the challenge that they do not know the project and its context. In particular, an LLM cannot handle the complete content of a larger software project due to the project's size and limited input lengths. Fortunately, the processing can be augmented by retrieval techniques, called retrieval-augmented generation (RAG) [35]. Here, the first step is retrieving relevant artifacts based on information retrieval (IR) techniques. The LLM then generates the answer using the retrieved artifacts.

In this paper, we propose using RAG with LLMs to perform TLR between various artifacts. Our RAG-based approach is called *LiSSA (Linking Software System Artifacts)*. We focus on three different TLR tasks that cover diverse settings to generate insights about the performance of RAG-based approaches for TLR: tracing requirements to code, tracing architecture documentation to code, and tracing architecture documentation to architecture models.

The example project in Figure 1 illustrates these tasks. The project contains requirements, some classes from the source code, the component-based architecture, and a part of the documentation of a system that provides and processes images. We consider the four artifacts as different sources of information about the system. The requirements contain the need for a REST API that is implemented in the code as `RestFacade`. Thus, there is this trace link between the requirement and the code file. The component diagram shows the database component, realized in code as `package db`. This results in a trace link between the architecture and the code. Lastly, the documentation describes using a REST interface to process the images. This defines trace links from the documentation to the code and to the `Image Processing` component.

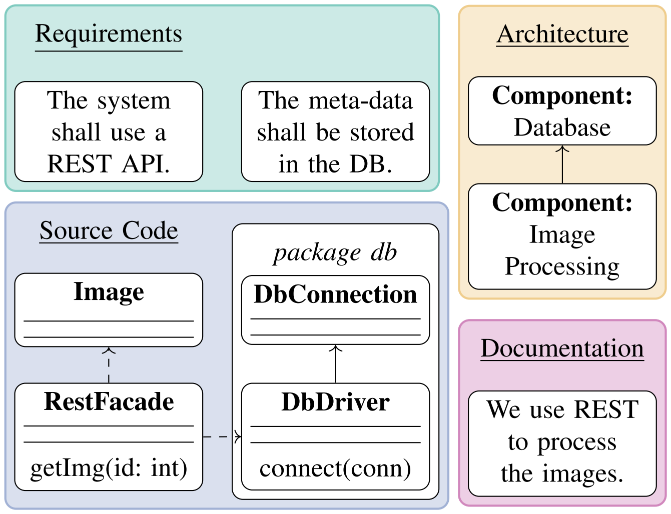

Fig. 1. Example Project with Requirements, an Architecture Diagram, Natural Language Documentation, and Source Code

We select these three different TLR tasks to empirically evaluate LiSSA on a variety of artifact types and trace links. Moreover, these tasks have been actively researched in the past and datasets are available to evaluate TLR performance.

This motivates us to ask **RQ1**: Is a generic RAG-based TLR approach better than task-specific, state-of-the-art approaches?

Moreover, as recent research indicates that Chain-of-thought (CoT) prompting can be more effective than other zero-shot prompts in certain scenarios [36], we want to further explore how the performance is affected by these LLM prompting techniques. Thus, we ask **RQ2**: Is CoT prompting more effective than a simple classification prompt?

RAG heavily depends on the ability to retrieve relevant information, i.e., (parts of) artifacts. Consequently, we investigate the impact of different preprocessing techniques for this retrieval; and since related research shows that fine-grained mappings can improve the performance of TLR tasks [13], we ask **RQ3**: Does retrieval and mapping on a fine-grained, sub-artifact level improve the TLR performance compared to mapping on an artifact level? Furthermore, we investigate **RQ4**: Does RAG improve the TLR performance compared to embedding-based IR TLR?

We make the following contributions:

C1: We provide LiSSA, an RAG-based approach for TLR between different types of software artifacts. The approach, datasets, and results are publicly available in our replication package [37].

C2: We evaluate the performance of LiSSA on three different TLR tasks and compare it to state-of-the-art approaches.

C3: We investigate the influence of prompt techniques and preprocessing on the performance of LiSSA.

The remainder of this work is structured as follows: In Section II, we present related work and discuss the state-of-the-art for different TLR tasks. Afterward, in Section III, we present our RAG-based approach LiSSA. In Section IV, we discuss the evaluation of retrieval-augmented LLMs for TLR and their performance on three established TLR tasks. Afterward, we discuss our findings and their implications for practice and research in Section V. Lastly, we conclude the paper in Section VI.

II. Related Work

In this section, we focus on the state-of-the-art in the different TLR tasks and discuss the various existing approaches. We also cover recent advances in using LLMs for TLR.

A. *Traceability Link Recovery*

In the following, we discuss the state-of-the-art approaches for the three types of TLR tasks we consider: requirements to code, documentation to code, and architecture documentation to architecture models.

1) Requirements to Code: A significant amount of research in TLR focuses on the relationship between requirements and code. Within this context, methods from natural language processing serve as fundamental components of various approaches. Notably, vector space models, latent semantic indexing and latent dirichlet allocation are frequently employed techniques [8]–[12]. These models consider semantic similarity between requirements and code to bridge the semantic gap and recover trace links. To measure similarity, important features such as common terms between artifacts and semantic vectors of artifacts are primarily considered.

However, developers might use different terms to refer to the same concept across different artifacts [38]. This might result in mismatched vocabulary, making trace links hard to recover. To address this problem, Gao et al. [14], [39] use co-occurring term pairs (biterms) from both requirements and code to enrich the artifact texts from both sides. The idea is that biterms indicate the consensus between the artifacts.

Hey et al. [13] show that a more fine-grained interpretation of artifacts could improve recovery results. Their approach FTLR uses sentences from requirements and methods from code files as units for the similarity mapping, rather than the whole requirements and source code files. FTLR represents the semantics of the fine-grained elements through word embeddings. Moreover, they discovered that selecting the appropriate information is crucial for enhancing the performance of the approach. Hey et al. also show that the relevant information can be derived automatically with an LLM-based classifier [40].

The consensus of these approaches is the ambition to semantically understand the artifacts to bridge the semantic gap. LLMs promise deep language understanding of the input text, making them a plausible alternative for TLR.

Another option to bridge the semantic gap is using transitive links between artifacts. The underlying idea is that intermediate artifacts can find implicit tracing relationships [41].

Nishikawa et al. [42] generate trace links between two artifact types, *A* and *B*, by utilizing an intermediate artifact type *C*. They use a vector space model to create trace links between artifacts of types *A* and *C* and between artifacts of types *B* and *C*. Artifacts from *A* and *B* that both trace to the same artifact of type *C* are considered connected.

Moran et al. [41] introduced COMET, a Bayesian inference-based method for TLR. The method involves three stages: initially combining multiple text similarity metrics to generate preliminary trace links, then incorporating developer feedback to refine these links, and finally utilizing information from transitive links to enhance the results.

However, when there are multiple transitive paths between artifact pairs, it remains a challenge how to aggregate them. This challenge is explored by Rodriguez et al. [43]. Their approach aggregates multiple paths using one of three methods: the maximum score, the sum, or a weighted sum. Rodriguez et al. found that an optimal transitive technique requires specific tuning for each project and possibly for different trace paths within a project.

Transitive links can also help within a single artifact type like code. Panichella et al. [44] showed the benefits of including call and inheritance dependencies for TLR. Kuang et al. [45] then demonstrated that data dependencies provide complementary information to call dependencies that are beneficial for linking requirements to code. The idea is that "close" methods are probably linked to similar requirements.

2) Documentation to Code: The second type of TLR task is the recovery of trace links between documentation and code. In their study on recovering trace links between code and manual pages, Antoniol et al. [8] found that programmers often use meaningful names for program items. As such, they use a unigram language model and a Bayesian classifier and compared it to a vector space model. Marcus and Maletic [9] apply latent semantic indexing, achieving higher precision and recall.

The approach TransArC by Keim et al. [15] uses a component-based architecture model as an intermediate artifact to recover trace links between architecture documentation and code. This transitive approach yields improved results by combining TLR approaches that need to bridge smaller semantic gaps. Because architecture models are not always present, they also provide a baseline for their performance without using the intermediate artifact, namely ArDoCode.

3) Documentation to Models: The third type of TLR task is the recovery of trace links between documentation and architecture models. For this task, Cleland-Huang et al. [46] propose enhancement strategies to maximize recall and precision, while maintaining a high recall. To achieve this, they discover that links that have a medium probability (a.k.a. low confidence links) are hard to distinguish between true and false links, resulting in bad performance. To mitigate this problem, they propose the usage of hierarchical information and logical groupings of artifacts to enhance precision.

For tracing documentation to Business Process Model and Notation (BPMN), Lapena et al. found that both the linguistic particularities [47] of BPMN models and their execution traces [48] can help recovering lost links.

Keim et al. [7] propose ArDoCo for TLR of architecture documentation to component-based architecture models. They use heuristics to identify word clusters that could define a model element. ArDoCo then traces these clusters to the model elements. Their dataset for this task is publicly available [49].

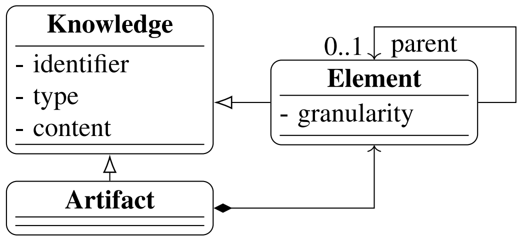

Fig. 2. Data Model: Knowledge, Elements, and Artifacts

B. Using LLMs for TLR: Prompting and RAG

Until now, LLMs have only rarely been employed for TLR. Lin et al. [17] fine-tune a CodeBERT LLM for the task of recovering trace links between issues and commits. However, their approach requires initial trace links from a project for fine-tuning. Therefore, they tackle a different kind of TLR problem than our approach.

Recently, Rodriguez et al. [36] studied the application of prompt engineering on TLR. They showed the LLM's ability to comprehend domain-specific knowledge such as acronyms. Furthermore, they showed that small modifications of prompts can result in differences in TLR results.

In contrast to our approach, neither the work by Rodriguez et al. nor other existing approaches explore the application of RAG for different TLR tasks. Since TLR handles large amounts of data, it can be expensive or even impossible due to size restrictions to process all data naively. Therefore, retrieval-augmented generation [35] is one possible approach to make TLR applicable to large datasets.

Moreover, we focus on the application of prompting techniques and different preprocessing instead of prompt engineering. Finally, we also apply our approach to different TLR tasks to evaluate its applicability on various TLR tasks.

III. The LiSSA Approach

In this section, we present the LiSSA approach and discuss its different components. The approach uses multiple steps to recover trace links between different artifacts. As input, LiSSA takes a set of artifacts, e.g., requirements, documentation, or architecture models. The output of LiSSA is a set of trace links between the elements within the artifacts or between artifacts, e.g., sentences to source code files or text files to code files.

A. Preprocessing & Embedding

LiSSA distinguishes *artifacts*, *elements*, and *knowledge*. Their relationship is shown in Figure 2. We define *artifacts* as the inputs that can be processed by LiSSA. Artifacts consist of the original content of files and have a specific artifact type. *Elements* are the traceable units of the artifacts and, thus, are mainly the parts that are used to create the trace links. Besides their identifier, type, and content, they also have a granularity. This granularity is used to define the level of abstraction of the trace links. Furthermore, elements

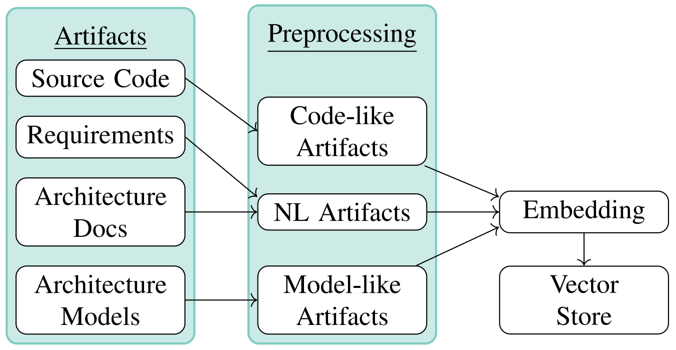

Fig. 3. Preprocessing & Embedding of Artifacts (NL for Natural Language)

can have a parent. The child-parent relationship is directly reflected by the granularity of the elements. For example, a sentence (child) is part of a text file (parent). *Knowledge* defines all data that can be used by LiSSA to create trace links. It is defined by an identifier, a type, and its content. In this work, we regard both the contents of the artifacts and their elements as knowledge. We have kept this base class knowledge intentionally general because we designed LiSSA as an extensible framework. Most important, knowledge does not need to be artifacts or elements. This makes it possible to enrich prompts with information that is not (directly) part of them. It also enables more sophisticated retrieval techniques and ensures the extensibility of the framework.

After loading the relevant artifacts, the approach first preprocesses the artifacts. As shown in Figure 3, the approach currently distinguishes three types of artifacts: Natural language artifacts, code artifacts, and model artifacts.

Natural language artifacts describe text-based artifacts that mainly consist of written natural language text. Examples include requirements and documentation. These artifacts are typically preprocessed by splitting them into lines, sentences, paragraphs, or arbitrary chunks of text.

Code artifacts define artifacts that are written in a programming language. This could either be source code, test code, or shell scripts. Code artifacts are preprocessed based on their programming language. For this paper, we focus on Java code since the obtained datasets from related work mostly contain Java code. The approach extracts elements of the code artifacts by splitting the code based on methods, classes, or files.

Model artifacts like architecture models are instances of a certain metamodel. In our case, we focus on UML component diagrams that are used to describe the architecture of a software system. We select these models because they are used by one of the state-of-the-art approaches, allowing us to compare our approach to them. Model artifacts are preprocessed by extracting the elements like components and interfaces.

In general, the approach preprocesses elements from the different artifact types by transforming them into a textual representation that can be used for retrieval. As shown in Figure 3, after transformation, LiSSA uses embedding models to create a vector representation of each element. The embeddings of elements from artifacts are stored in a vector store.

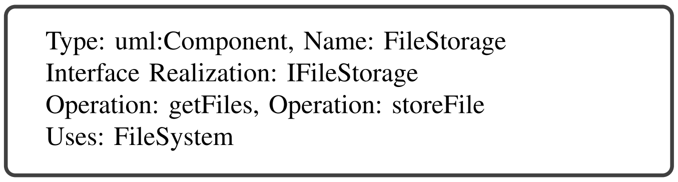
Type: uml:Component, Name: FileStorage
Interface Realization: IFileStorage
Operation: getFiles, Operation: storeFile
Uses: FileSystem

Fig. 4. Example textual representation of a model element

Preprocessors in Detail: In the following, we present the preprocessors of LiSSA that we identified as relevant for the artifacts of the given TLR tasks. The decision is informed by the datasets from related work (cf. Section II).

a) None: The first preprocessor does not change the artifact and only creates the embedding. This means that no splitting is done, and the whole artifact is used as one element. One issue in this preprocessing is that artifacts might be longer than the maximum input size of the embedding model or the LLM. To deal with this, we cap the artifacts w.r.t. the input size of the models. In the future, this issue might dissolve with models that support increased maximum input sizes.

b) Sentence: The second way of preprocessing is tailored to text-based natural language artifacts. The artifact can be split into sentences and each sentence is then defined as an element. This way, the approach supports fine-grained trace links between sentences and other elements.

c) Chunk(N): When preprocessing text-based artifacts like source code, one method to split is chunking. This means the text is divided into chunks of a given size N. For source code, this chunking can also be tailored to the programming language by splitting the text at language-dependent keywords to create chunks of approximately the desired size. In our evaluation, we also assess the impact of different chunk sizes.

d) Method: A second way to split source code is to split it based on method declarations. Methods including their body, signature, and documentation define a unit that shall fulfill a certain task. Consequently, the preprocessor splits the code before each method definition, including its method documentation like Javadoc. The underlying idea is to treat methods as meaningful, atomic units and not possibly split in the middle of a method. The resulting elements could also be used to generate trace links on the method level.

e) Models (Feature Extraction): Compared to the other kinds of artifacts, we preprocess models differently. Models are instances of a certain metamodel. In our case, we focus on UML component models based on the Eclipse Modeling Framework (EMF). The model preprocessor is tailored to this specific type of model and extracts features from a given model instance. The extractor identifies components, interfaces, their operations (methods), and their dependencies (realizations and usages). These extracted features are used to create a textual representation of a model element (component or interface) that can be used for the retrieval process. The creation of the textual representation is based on templates that are filled with the information of extracted features. Figure 4 shows an example of the textual representation of a model element.

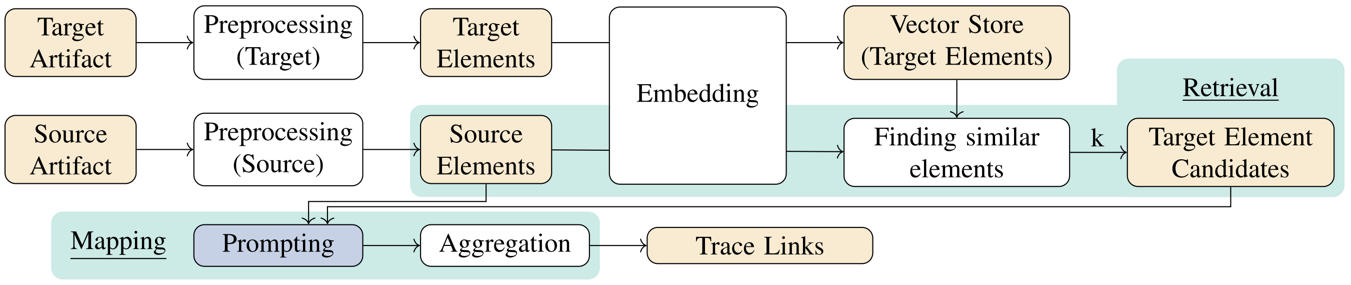

Fig. 5. Overview of the Retrieval & Mapping. Data is in orange, prompting is in blue, and other processing is in white.

B. Retrieval

As shown in Figure 5, the next step after preprocessing the artifacts is the retrieval. The retrieval step is responsible for finding the Top-k most similar elements for a given source element in the vector store containing the target elements. For the comparison of embeddings, the approach uses the cosine similarity of vectors. These most similar elements are then used as target element candidates. Only these candidates can be potential targets of a trace link for the given source element. Because the source elements are used to retrieve the target elements, the retrieval step defines the *direction* of the TLR task. We call *A to B TLR* (e.g., Requirements to Code TLR) when A is the source artifact and B is the target artifact.

In summary, the retrieval step takes the elements from a preprocessed source artifact, creates the embeddings, and identifies for each source element the Top-k most similar target elements as candidates. The preprocessing steps strongly influence the retrieval because the retrieval defines the possible mappings between elements. For example, in requirements to code TLR with the sentence preprocessor for requirements and the method preprocessor for code, the candidate mappings are between single sentences and single methods.

C. Mapping

After the retrieval comes the mapping process. The mapping takes a source element and its respective target element candidates and uses an LLM to classify which of the target element candidates belong to the source element. The approach then aggregates the identified mappings to create trace links on the desired level of abstraction. Consequently, the whole mapping process consists of two steps: prompting and aggregation.

1) Prompting: In the prompting step, the source element and the target element candidates are combined to create prompts for the LLM. The LLM's responses are then used to decide, for each candidate, if the source element belongs to the target element, and, therefore, if there should be a trace link. The approach supports different prompting techniques.

The prompting relies on the granularity of the source and target elements. For example, if there is requirements to code TLR and the approach uses the sentence preprocessor for requirements and the method preprocessor for code, then the LLM classifies if a sentence belongs to a method.

We base our prompts on the work of Rodriguez et al. [36]. Based on their work, we derive two prompts: the KISS prompt and the CoT prompt. The KISS prompt is a simple *classification prompt* that gives the LLM the freedom to interpret its task. The CoT prompt is a zero-shot prompt that is more complex and requires the LLM to reason about the elements. We detail these prompts in the following.

Prompt 1: KISS
Question: Here are two parts of software development artifacts.
{source_type}: '''{source_content}'''
{target_type}: '''{target_content}'''
Are they related?
Answer with 'yes' or 'no'.

Prompt 2: Chain-of-thought
Below are two artifacts from the same software system. Is there a traceability link between (1) and (2)? Give your reasoning and then answer with 'yes' or 'no' enclosed in <trace> </trace>.
(1) {source_type}: '''{source_content}'''
(2) {target_type}: '''{target_content}'''

a) KISS Prompt: The KISS prompt is a simple zero-shot prompt (see Prompt 1). The prompt first defines the software development domain. Afterward, the prompt states the types of elements and their contents. Lastly, the prompt contains a yes-or-no question if the source element belongs to the target element candidate. In the LLM's response, the approach looks for the term *yes*. If found, the approach traces the source element to the target element.

b) Chain-of-thought (CoT): The second prompt is the CoT prompt (see Prompt 2). This prompt also is a zero-shot prompt. It starts with the setting of the context by defining that the LLM gets two artifacts from a software system. Afterward, the prompt contains the same yes-or-no question as the KISS prompt. In contrast to the KISS prompt, the prompt asks the LLM to provide reasoning. Lastly, the prompt defines the output format and provides the source and target elements. Again, the approach looks at the output of the LLM and if the LLM approves, the approach again traces the source element to the target element. The reasoning itself is disregarded.

TABLE I
OVERVIEW OF DATASET FOR REQUIREMENTS TO CODE TLR. IT MARKS ITALIAN AND EN ENGLISH NATURAL LANGUAGE DESCRIPTIONS, AND TL STANDS FOR TRACE LINK

		Language		# of Artifacts		
Dataset	Domain	NL	Programming	Req	Code	TL
SMOS	Education	IT	Java	67	100	1044
eTour	Tourism	EN	Java	58	116	308
iTrust	Healthcare	EN	Java	131	226	286
Dronology (RE)	Aerospace	EN	Java, Python	99	423	602
Dronology (DD)	Aerospace	EN	Java, Python	211	423	740

TABLE II
NUMBER OF ARTIFACTS PER TYPE AND NUMBER OF TRACE LINKS IN THE GOLD STANDARD FOR EACH ARCHITECTURE TRACEABILITY PROJECT. AD STANDS FOR ARCHITECTURE DOCUMENTATION.

Artifact Type	MS	TS	TM	BBB	JR
AD Sentences	37	43	198	85	13
Architecture Model Elements	23	19	16	24	6
Source Code Files	97	205	832	547	1,979
AD to Model Trace Links	29	27	51	52	18
AD to Code Trace Links	50	707	7,610	1,295	8,240

2) Aggregation: The final step of the mapping is the aggregation. In this step, the traced pairs of source and target elements are combined to trace links that match the defined level of abstraction. The following example illustrates this: Based on the configured preprocessors, the prompting step has traced sentences of requirements and methods of code classes. The desired granularity for trace links is requirement file to code file. The aggregator creates the trace links if there is at least one correspondence between a sentence of a requirement file and a method of a certain class. By doing so, the aggregator can output the trace links on the desired level of abstraction.

IV. EVALUATION

In this section, we present the evaluation of our approach. We evaluate the performance of our approach on the three different TLR tasks and compare it to state-of-the-art approaches. Therefore, we discuss the used datasets, metrics, used models, baselines, and the performance of our RAG-based approach. We also perform significance tests to provide a statistical analysis of the results. Finally, we address the potential threats to validity. We provide all our results and datasets in our replication package [37].

A. Datasets

To reduce the bias of the creation of a new dataset, we use existing datasets from the literature. As already discussed, this is one factor in our selection of the respective TLR tasks.

1) Tracing Requirements to Code: For tracing requirements to code, we use a dataset from the replication package by Hey [50]. We use the projects SMOS, eTour, and iTrust as these projects are used for the evaluation of the state-of-the-art approaches FTLR and COMET. These projects were gathered by the *Center of Excellence for Software & Systems Traceability* (CoEST) [51] and cover different domains (education, tourism, and healthcare). Additionally, we use the Dronology dataset [52] (aerospace domain), and apply our approach and FTLR on it. Table I shows the characteristics of these projects, such as the used natural (NL) and programming languages, and the number of requirements, source code files, and trace links in the datasets. Note that all projects except for SMOS provide natural language descriptions in English. The SMOS project has requirements and source code identifiers in Italian. For SMOS, eTour, and iTrust, we can use the information on traceability links between requirements and code directly. However, the Dronology dataset contains different types of natural language artifacts, such as requirements (RE) and design definitions (DD). We use their summary and description, as well as their relationship to tasks and code, to derive two ground truths: A mapping from requirements to code and a mapping from design definition, similar to low-level requirements, to code.

2) Tracing Architecture Documentation to Architecture Models: For tracing architecture documentation to architecture models, we use the ArDoCo dataset [7], [49]. The dataset contains component-based architecture models, their documentation, and the gold standard for documentation to model trace links for five projects. The projects are MediaStore (MS), TeaStore (TS), TEAMMATES (TM), BigBlueButton (BBB), and JabRef (JR). Again, the projects cover a variety of domains (media, micro-services, education, web conferencing, and reference management). Table II gives an overview of the artifacts in the dataset. The artifacts of all projects are written in English. We use these projects to evaluate our approach against the state-of-the-art.

3) Tracing Documentation to Code: For tracing documentation to code, we use the dataset provided in TransArC's replication package [53]. The dataset expands the ArDoCo dataset by the corresponding code for the five projects and a gold standard for architecture documentation to code trace links. We use the projects BigBlueButton, MediaStore, and TeaStore to compare our approach to the state-of-the-art approaches. Furthermore, we discuss challenges regarding large projects like the two remaining projects of the dataset, JabRef and TEAMMATES, separately. These projects are characterized by the fact that they have many architecture documentation to code trace links. While the projects BigBlueButton (1295 trace links), MediaStore (50 trace links), and TeaStore (707 trace links) contain on average up to 16.4 trace links per sentence of documentation, TEAMMATES contains 38.4 and JabRef 633.8 trace links per sentence on average. The code artifacts of all projects are also written in English, and the gold standard covers only the Java and Shell script parts of their code.

B. Metrics

To evaluate the performance of our approach, we use commonly used metrics for TLR [22], [54]: precision, recall, and F_β-score. We use the F_1-score, as both precision and recall are essential for fully automated TLR. For semi-automatic

TLR, where recall is more critical to avoid missing links, we also report the F_2-score. By doing so, we can compare our approach's performance to the reported results of the state-of-the-art approaches. The information also gives us insights into future applications of an RAG-based approach for TLR.

C. Significance Tests

To provide a statistical analysis of the results, we perform significance tests. Due to non-negligible costs of the experiments, we test only the on-average best configuration. This also makes sense with a view to the practical application of LiSSA. In practice, a ready-to-use solution would be required. We use the two-sided Wilcoxon signed-rank test with a significance level of $\alpha = 0.05$. We calculate the Wilcoxon effect size r and interpret it using common thresholds: small $[0.1, 0.3)$, moderate $[0.3, 0.5)$, and large $[0.5, 1.0]$. We run the best configuration 5 additional times with other random seeds, resulting in a sample size of still only 6. Thus, the power of the conclusions might be limited due to a risk of errors.

D. Models

For the evaluation, we use OpenAI's recent models. We use *text-embedding-3-large* as the embedding model. We opted for that model based on the datasets we used for the evaluation. The datasets do not only contain English language artifacts but also artifacts in other languages. Furthermore, the vendor of the embedding model states that this model is their newest and best-performing embedding model.

For the Large Language Model for classification, we use *GPT-4o (gpt-4o-2024-05-13)* and *GPT-4o mini (gpt-4o-mini-2024-07-18)*. OpenAI claims that GPT-4o is their most advanced model. GPT-4o mini is OpenAI's most recent model at the time of writing that is potent, cheap, and fast.

E. Baselines

In order to evaluate the performance of our approach, we compare it to state-of-the-art approaches.

1) Retrieval-Only: A baseline approach for all our TLR tasks is the retrieval-only approach. This approach does not use the classifying LLM, but only the retrieval step of our approach. We then mock the classifier by always stating that a source element belongs to all found target element candidates. By doing so, we maximize the overall recall of the approach for a given amount of k target elements that should be retrieved. This allows the analysis of what effects the classifier step has and whether the classifier is beneficial.

2) Tracing Requirements to Code: For tracing requirements to code, we compare our approach to the state-of-the-art approaches FTLR and COMET. Both approaches have already been discussed in Section II. Additionally, we compare our approach to a VSM and LSI baseline. For FTLR [13], [40], we provide the results of the best configuration with the originally defined thresholds. It uses method comments, call dependencies, and filter based on use case templates and functional aspects. Additionally, we take a look at the results of the best per-project optimized threshold configuration (FTLR_{OPT}), illustrating the upper boundary of FTLR's performance. This configuration only uses requirements filter based on use case templates and functional aspects, and no method comments or call dependencies. For the baseline COMET [41], we also provide the results for the best per-project optimized configuration ($\text{COMET}_{\text{OPT}}$). The results for SMOS, eTour, and iTrust are already part of the replication package by Hey [50]. Thus, we report these results for comparison. Dronology has not been evaluated with FTLR so far, so we apply FTLR to it. Unfortunately, we could not run and evaluate COMET on the Dronology dataset. For the VSM and LSI baselines we make use of the code provided by Gao et al. [14] in their replication package. The code only produces ranked lists of target artifacts per source artifact without a fixed threshold for determining the final trace links. Therefore, we again calculated the per-project optimized F_1-scores (VSM_{OPT} & LSI_{OPT}) by varying the cutoff threshold between $[0, 1]$ in 0.01 steps. The results, thus, again show the upper boundary of their performance.

3) Tracing Documentation to Code: For this task, we compare our approach with the work of Keim et al. [15] (see also Section II-A2). Since we focus on the direct linking of documentation to code, we do not compare to TransArC. TransArC uses intermediate artifacts and, thus, extra knowledge that is not available for the direct linking approach. Consequently, we use the best-known approach for direct linking from Keim et al. — ArDoCode. ArDoCode is based on heuristics that have been originally optimized for the TLR from documentation to architecture models. Nevertheless, ArDoCode achieved the best results for the direct TLR from documentation to code, i.e., without using architecture models.

4) Tracing Architecture Documentation to Architecture Models: For tracing between architecture documentation and architecture models, we compare our approach with the state-of-the-art approach ArDoCo [7]. ArDoCo is based on heuristics and tailored to this TLR task (cf. Section II-A3). We again compare our results directly with the results reported by the authors of ArDoCo without any modifications.

F. Results

This section presents and discusses the evaluation. Our replication package [37] contains all detailed results.

1) Tracing Requirements to Code: Table III shows the detailed results using GPT-4o. For the TLR from Requirements to Code, we consider eleven different configurations. In the first group, there are three configurations without preprocessing the artifacts. These three configurations compare the effects of the different classifiers introduced in Section III-C, i.e., using no LLM at all, using Prompt 1 (KISS prompt), and using Prompt 2 (CoT prompt).

The second group of results shows the effects of preprocessing the input by reducing the requirements to sentences and splitting the code into chunks. As chunk size, we define a value of 200 characters. This value should ensure a fine-grained splitting, where also parts of the methods are transformed into elements. In this setting, we evaluate four classifiers. Again, we evaluate the use of no LLM, the KISS prompt, and the CoT

TABLE III
RESULTS FOR REQUIREMENT-TO-CODE TLR WITH GPT-4O & TOP-20 (PREP. FOR PREPROCESSOR, CLS. FOR CLASSIFIER, ART. FOR ARTIFACTS)

		SMOS				eTour				iTrust				Dronology (RE)				Dronology (DD)			
Approach		P	R	F_1	F_2	P	R	F_1	F_2	P	R	F_1	F_2	P	R	F_1	F_2	P	R	F_1	F_2
VSM_{OPT}		.430	.414	.422	.417	.557	.427	.483	.448	.208	.227	.217	.223	.844	.087	.158	.106	.846	.071	.131	.087
LSI_{OPT}		.415	.430	.422	.427	.452	.453	.453	.453	.251	.255	.253	.254	.333	.107	.162	.124	.757	.074	.135	.090
$COMET_{OPT}$		.195	.572	.291	.413	.410	.468	.437	.455	**.361**	.231	.282	.249	—	—	—	—	—	—	—	—
FTLR		.444	.331	.380	.349	.379	.633	.474	.558	.165	.339	.222	.280	.183	.161	.172	.165	.129	.154	.140	.148
$FTLR_{OPT}$		.314	**.588**	.409	**.501**	**.505**	.597	**.548**	.576	.234	.241	.238	.240	.184	.170	.177	.173	.140	.147	.144	.146
Prep.	**Cls.**																				
None / None	No	.325	.418	.366	.395	.216	**.815**	.342	.525	.058	.531	.105	.202	.128	**.420**	.196	.288	.085	**.482**	.144	.249
	P1	**.632**	.184	.285	.214	.378	.711	.493	.604	.206	.493	**.290**	**.385**	.200	.372	.260	**.317**	.155	.442	.229	**.322**
	P2	.590	.195	.294	.226	.409	.734	.526	**.633**	.199	.451	.276	.360	**.226**	.344	**.273**	.311	.177	.380	.241	.309
Sentence / Chunk(200)	No	.247	.546	.340	.439	.091	**.815**	.164	.315	.066	.563	.119	.225	.150	.331	.206	.266	.119	.424	.186	.281
	P1	.327	.297	.311	.302	.136	.581	.220	.351	.152	.451	.227	.324	.198	.233	.214	.225	.193	.315	.239	.280
	P2	.344	.289	.314	.299	.140	.617	.228	.366	.143	.465	.219	.321	.216	.194	.205	.198	**.217**	.281	.245	.265
	+Art.	.333	.336	.335	.336	.138	.662	.229	.377	.153	.517	.236	.350	.225	.281	.250	.267	.199	.351	**.254**	.305
Sentence / Method	No	.327	.541	.408	.479	.073	.597	.130	.245	.063	**.598**	.114	.221	.132	.282	.180	.230	.101	.369	.159	.241
	P1	.475	.351	.403	.370	.107	.412	.169	.262	.184	.503	.269	.373	.215	.208	.211	.209	.203	.276	.234	.257
	P2	.479	.379	**.423**	.396	.107	.435	.172	.270	.171	.517	.257	.368	.209	.211	.210	.211	.197	.258	.224	.243
	+Art.	.486	.343	.402	.364	.109	.484	.178	.287	.168	.535	.256	.372	.203	.249	.224	.238	.190	.323	.239	.283

prompt. Additionally, we evaluate how the provided elements affect the performance of the approach. Therefore, we defined a variant where the LLM is provided with the original artifact instead of the elements from the retrieval. This means that we use the elements, i.e., parts of artifacts, to retrieve relevant artifacts. The LLM then uses the whole artifact to decide.

The final group shows the results where the preprocessor for code is replaced with the method preprocessor tailored to Java source code. Similar to the previous group of results, we evaluate four classifiers and their performance.

In the first group, as expected, the configuration using no LLM at all achieves the highest recall. This configuration defines the upper bound for recall of a retrieval-augmented TLR approach for a defined preprocessing. The next two rows in the table show the results for the two prompt types. Except for SMOS, the configurations using the two prompt types outperform the no LLM configuration, both in F_1-score and F_2-score. In many cases, for GPT-4o, Prompt 2, the CoT prompt, outperforms the KISS prompt regarding F_1-score. On average, this is our overall best configuration w.r.t. F_1-score.

In the second group with chunks consisting of 200 characters, CoT prompts outperform the KISS prompts. Furthermore, the table shows that the use of target artifacts instead of target elements increases the performance of the approach in F_1-score and F_2-score. Nevertheless, this preprocessing performs slightly worse compared to the variant without preprocessing.

In the third group with method-chunking, the performance of the prompts varies between the projects. Our overall best results in F_1-score and F_2-score for SMOS are achieved using this preprocessing. For eTour, this preprocessing achieved the worst results. The other projects are comparable to the results in the previous groups. This indicates that there might be some characteristics in the projects that affect the performance of the approach. SMOS is the only project written in Italian and, therefore, another configuration might be more suitable for projects in English. eTour's method comments are often inconsistent with the code they describe [55]. This may negatively impact the performance of the embedding, and therefore, the retrieval. On eTour, we observe a decrease in recall from 0.815 to 0.597 in the retrieval-only configurations, if considering methods as elements to embed.

TABLE IV
COMPARISON OF THE AVERAGE AND WEIGHTED AVERAGE F_β-SCORES USING GPT-4O AND GPT-4O MINI FOR REQUIREMENT-TO-CODE TLR USING TOP-20 RETRIEVAL

		Avg.				w. Avg.			
Approach		GPT-4o		GPT-4o mini		GPT-4o		GPT-4o mini	
Prep.	**Cls.**	F_1	F_2	F_1	F_2	F_1	F_2	F_1	F_2
None / None	No	.230	.332	.230	.332	.249	**.332**	.249	.332
	P1	.312	**.369**	.247	.345	.288	.319	.260	**.334**
	P2	**.322**	.368	**.295**	**.356**	**.299**	.319	**.282**	.315
Sentence / Chunk(200)	No	.203	.305	.203	.305	.235	**.332**	.235	.332
	P1	.242	.296	.210	.309	.256	.288	.241	.331
	P2	.242	.290	.236	.317	.257	.279	.261	.325
	+Art.	.261	.327	.243	.318	.277	.320	.263	.314
Sentence / Method	No	.198	.283	.198	.283	.243	.321	.243	.321
	P1	.257	.294	.211	.292	.285	.299	.256	.325
	P2	.257	.297	.232	.295	.289	.305	.268	.307
	+Art.	.260	.309	.238	.296	.288	.312	.267	.298
Baselines		F_1		F_2		F_1		F_2	
VSM_{OPT}		.282		.256		.283		.257	
LSI_{OPT}		.285		.270		.285		.268	
FTLR		.278		.300		.273		.277	
$FTLR_{OPT}$		.303		.327		.294		.329	

Table IV provides an overview of the results for the two models, *GPT-4o* and *GPT-4o mini*, in comparison to FTLR, VSM, and LSI. We report the average and weighted average

TABLE V
P-VALUES OF THE WILCOXON SIGNED-RANK TEST (TWO-SIDED) FOR F_1/F_2-SCORES OF REQUIREMENT-TO-CODE TLR WITH GPT-4O, NONE/NONE/P2, AND TOP-20. (↑ OURS BETTER, ↓ BASELINE BETTER).

Approach	SMOS	eTour	iTrust	DR RE	DR DD	Avg.	w. Avg.
VSM_{OPT}	**.03↓**	**.03↑**	**.03↑**	**.03↑**	**.03↑**	**.03↑**	**.03↑**
LSI_{OPT}	**.03↓**	**.03↑**	**.03↑**	**.03↑**	**.03↑**	**.03↑**	**.03↑**
$COMET_{OPT}$	.09/**.03↓**	**.03↑**	.84/**.03↑**	—	—	—	—
FTLR	**.03↓**	**.03↑**	**.03↑**	**.03↑**	**.03↑**	**.03↑**	**.03↑**
$FTLR_{OPT}$	**.03↓**	.03↓/**.03↑**	**.03↑**	**.03↑**	**.03↑**	**.03↑**	.44/**.03↓**

TABLE VI
COMPARISON OF THE AVERAGE AND WEIGHTED AVERAGE F_β-SCORES FOR SOFTWARE ARCHITECTURE DOCUMENTATION TO CODE TLR ON THE SMALLER PROJECTS BIGBLUEBUTTON, TEASTORE, MEDIASTORE USING GPT-4O MINI AND TOP-20 RETRIEVAL

Approach		Avg.		w. Avg.	
Preprocessor	**Classifier**	F_1	F_2	F_1	F_2
Sentence / None	No LLM	.209	.257	.240	.260
	Prompt 1	.212	.259	.243	.262
	Prompt 2	**.217**	.255	**.249**	.253
Sentence / Chunk(200)	No LLM	.206	.238	.214	.204
	Prompt 1	.206	.238	.214	.204
	Prompt 2	.213	.241	.221	.203
Sentence / Chunk(1000)	No LLM	.196	.230	.213	.206
	Prompt 1	.196	.228	.213	.205
	Prompt 2	.205	.237	.217	.203
Baselines		F_1	F_2	F_1	F_2
ArDoCode		.178	**.302**	.189	**.318**

for the F_1-score and F_2-score. The overall best results regarding both scores are achieved by our approach without splitting the artifacts. The larger model outperforms both the smaller model and the state-of-the-art approach on average and on weighted average F_1-scores. Regarding, F_2-scores, the results show that, on average, the best configuration is achieved by the GPT-4o model using Prompt 1 without preprocessing. Except for the weighted average F_2-score using GPT-4o, applying classifiers improve our performance in both scores. There, the best performance is achieved by using no LLM. However, this result is mainly influenced by the SMOS project.

Interestingly, for *SMOS* GPT-4o mini achieves an F_1-score of 0.425 using the KISS prompt for Sentence-Method-Preprocessing. Thereby, GPT-4o mini outperforms GPT-4o for this project. This means that for some projects, GPT-4o mini can outperform the state-of-the-art and the larger GPT-4o. Overall, GPT-4o mini is comparable to the state-of-the-art.

We perform significance tests for the best configuration of the GPT-4o model (No preprocessing and Prompt 2). The p-values of the significance test are shown in Table V. The results with a significant difference between our results and the baseline are marked in bold font. If our result is better than a baseline, we mark it with ↑ — otherwise, we mark it with ↓. On average, our approach is significantly better than all state-of-the-art approaches. However, the results also show that our on-average best prompt is not significantly better for the SMOS dataset. Here, we emphasize that this is the only project of the dataset that contains Italian language artifacts. The only non-significant differences occur for per-project optimized baselines. It may be expected that their actual performance in practice is lower, exemplified by FTLR vs. $FTLR_{OPT}$. The results for F_1-score and F_2-score are similar. The only differences are for $COMET_{OPT}$ on SMOS and iTrust as well as $FTLR_{OPT}$ on eTour and in weighted average. In F_2-score LiSSA is significantly better than the baselines on eTour and iTrust. Regarding effect size, we observed that if the difference is significant, the effect size is large ($r \geqslant 0.5$). For confidence intervals, means, and standard deviations, we refer to our replication package [37].

In summary, the approach LiSSA outperforms the state-of-the-art approaches FTLR and COMET using GPT-4o. In contrast to the GPT-4o mini model, this comes with the disadvantage of high costs. Considering the results for GPT-4o mini, the approach performs similarly to FTLR and performs comparable to GPT-4o. In the end, we've spent approx. \$850 for the evaluation of this task with GPT-4o (excluding the repetitions for the significance tests). GPT-4o mini only cost us approx. \$30 for worse, but comparable results.

2) Tracing Documentation to Code: In the second task, we evaluate TLR from documentation to code. As discussed in Section IV-A3, we use the TransArC dataset for this evaluation. Since this task is defined as tracing single sentences to code files, we always use the sentence preprocessor for the documentation. Regarding code, we evaluate the configurations for no preprocessing, small chunks with 200 characters, and larger chunks with 1000 characters.

We first evaluate the performance of the smaller projects (BigBlueButton, MediaStore, and TeaStore). The results for those using Top-20 retrieval are shown in Table VI. Similar to the evaluation of requirements to code TLR, we report the results for the two different prompt techniques and the retrieval-only (No LLM) for each preprocessing configuration. On average, we can see again that the combination of CoT and no further preprocessing achieves the best results regarding F_1. For this configuration, the Wilcoxon signed-rank test shows that the F_1-scores of the averages are significantly better than the baseline ArDoCode and the No LLM configuration (both $p = 0.031$). Moreover, the effect size is large ($r \geqslant 0.5$) in both cases. The baseline ArDoCode is better in F_2-score due to high recall. However, the average precision is only 0.107. For the smaller projects, the approach already achieves similar or even better results than the baseline.

Nevertheless, we also uncover challenges for larger projects. With large projects, the retrieval is problematic: On the two largest projects JabRef and TEAMMATES, the best retrieval-only configuration for Top-40 elements achieves a recall of 0.027 for JabRef and 0.061 for TEAMMATES. Additionally, the bad retrieval for the large projects affects the weighted average heavily: the weighted average F_1-score of the best configuration including the large projects is 0.081 (without:

TABLE VII
Comparison Of The Average And Weighted Average F_β-scores for Architecture Model to Architecture Documentation (M2D) and Architecture Documentation to Architecture Model (D2M) TLR Using Top-10 Retrieval And GPT-4o mini

Approach		Avg.				w. Avg.			
		D2M		M2D		D2M		M2D	
Features	**Cls.**	F_1	F_2	F_1	F_2	F_1	F_2	F_1	F_2
Name	No	.162	.305	.350	.505	.131	.259	.334	.466
	P1	.173	.323	.357	.512	.140	.275	.340	.472
	P2	**.286**	**.467**	**.458**	**.589**	**.234**	**.404**	**.424**	**.534**
Name, Interfaces	No	.162	.305	.349	.504	.131	.259	.334	.466
	P1	.169	.317	.355	.509	.138	.271	.339	.471
	P2	.274	.447	.450	.586	.223	.383	.419	**.534**
Name, Interfaces, Usages	No	.161	.303	.341	.494	.130	.257	.322	.452
	P1	.169	.315	.348	.501	.137	.269	.329	.458
	P2	.265	.435	.417	.554	.215	.373	.384	.499
Baselines		F_1		F_2		F_1		F_2	
ArDoCo		**.822**		**.814**		**.802**		**.806**	

0.249). Again, the best configuration in this case is the same as for the smaller projects: CoT prompt without preprocessing.

In summary, our approach LiSSA achieves significantly better results in F_1-score than state-of-the-art approaches on smaller projects. For larger projects, the performance of the retrieval needs to improve.

3) Tracing Architecture Documentation to Architecture Models: The final TLR task is recovering trace links between architecture documentation and architecture models. Because there are comparably few model elements and few sentences, we additionally analyzed the effect of the direction for TLR. This means, we evaluate both, TLR for Documentation to Model (D2M) and TLR for Model to Documentation (M2D). For both cases, we use a retriever that retrieves the 10 most similar target elements for each source element. Since the TLR task requires a trace link between a sentence and a model element (i.e., for this TLR task, a component), we used the sentence preprocessor to split the documentation.

We evaluate the effect of the prompt type and the effect of the features that are extracted from the model preprocessor. The results of the evaluation are shown in Table VII. On average, the CoT prompt outperforms the KISS prompt and using retrieval-only in both scores.

For the D2M TLR task, the displayed F_1-score comes from a very high recall of almost 1.0 but a very low precision.

When examining the flipped task, the recall decreases, while the precision increases substantially. Nevertheless, the results vary on the projects. We assume that this is mainly caused by the retrieval. The retrieval always tries to select the Top-k most similar target elements. Considering M2D TLR, this means that for each component in the model, the approach retrieves the Top-k most similar sentences. Nevertheless, the value k might depend on the project and also the direction of the TLR task. Depending on the project, the assumption that a component is described in at most k sentences can be fulfilled.

Considering the different features, on average, the approach's performance worsens with the number of used features. We assume that this is due to the kind of trace links. The LLM has to decide if a component belongs to a sentence. Too much information that might not be directly related or that is unique to the component can negatively influence ("distract") the LLM. This then leads to decreased precision.

Overall, our approach does not outperform the state-of-the-art approach ArDoCo.

G. Threats to Validity

In this part, we discuss threats to validity and base this discussion on the guidelines by Runeson et al. [56]. We also consider the work by Sallou et al. [57] that focuses on threats for experiments with LLMs in software engineering.

Regarding *construct validity*, there is the threat that the datasets are biased. To mitigate this threat, we use the same datasets as the state-of-the-art approaches. The datasets cover different domains and project sizes. Nevertheless, the datasets mostly contain open-source projects that may have other characteristics compared to closed-source projects. In this work, we do not focus on prompt engineering, i.e., we do not extensively modify prompts and compare their performance. Consequently, a potential threat to the validity of our results regarding **RQ1-4** is the selection of the prompts. However, we closely follow the suggestions of Rodriguez et al. [36]. Another threat is a biased selection of metrics. To diminish this risk, we use commonly used metrics in TLR research. Overall, we try to reduce potential confounding factors that could prevent us from effectively addressing our RQs.

Considering *internal validity*, there is the threat that other factors might influence our experiments. These could influence our interpretations and lead to wrong conclusions. To mitigate this threat, we follow established practices. We ensure that we use datasets and projects from state-of-the-art approaches. Furthermore, we clearly state the origins of the projects and ground truths. This also reduces the risk of selection bias. Since the datasets mostly consist of open-source projects, they still might vary in quality. The quality and consistency within one project can affect the performance of the approach.

Regarding *external validity*, there are various threats. First, we only evaluate the approach on a limited number of projects and a limited number of TLR tasks. The results can vary for other projects and other TLR tasks. To reduce this threat, we use established datasets and projects from different domains and sizes. Second, we evaluate the approach with only one embedding model and two LLMs. To mitigate this threat, we use state-of-the-art models. The third threat is the use of *Closed Source Models* for evaluation. Since we do not know the training data of the models, we cannot ensure that there is no *data leakage*. In order to mitigate this threat, we provide our code, the prompts, embeddings, and the responses of the LLMs in our replication package [37] to ensure transparency. The fourth threat is the non-determinism of the LLMs. To mitigate this threat, we set a random seed for the LLMs and set their temperature to 0.

With the aforementioned evaluation on established datasets using established metrics, we address the *reliability* of our research. Therefore, we do not introduce a threat regarding the manual creation of the datasets. Nevertheless, the gold standard might be imperfect, as they are manually created. To improve reliability, we also base our prompts on the literature.

V. Discussion & Future Work

In this section, we analyze our results on RAG-based TLR concerning challenges, impediments, and implications in both research and practice. We also provide a brief overview of planned future improvements.

One challenge is defining trace links so that LLMs can effectively interpret them. Terms such as "related" or "traceability link" can be too ambiguous for LLMs and might need further refinement. Another challenge involves optimizing the retrieval process, which sets the upper limit for recall. A significant issue arises when a single sentence maps to hundreds of code files, as observed in larger projects like JabRef. In such cases, a high number of elements must be retrieved to ensure all relevant code files are included, pushing the retrieval process to its limits. To address this, we suggest focusing on improved retrieval techniques. Possible solutions to enhance the process include dynamic thresholds, alternative data structures for retrieval, or artifact summarization.

Our results show that the performance in F_1-score and F_2-score of the proposed RAG-based TLR does not yet enable fully automated TLR in practical applications; even for semi-automated settings (F_2-score), the performance remains insufficient for practice [22]. Nevertheless, state-of-the-art approaches perform worse on code-related TLR tasks.

RAG presents several opportunities to improve performance: Future research could focus on prompt engineering, on incorporating more context, and on exploring advanced RAG techniques. Our framework, LiSSA, serves as a foundation here. LiSSA already outperforms code-related TLR approaches, though it does not yet address these future directions. Therefore, we see significant potential for RAG in these tasks.

In the future, we aim to expand our LiSSA approach in several ways. Our findings show that our current fine-grained mappings do not enhance performance, motivating us to explore more advanced aggregation techniques. For example, we plan to adapt methods like majority voting from FTLR [13] to improve the aggregation step. Moreover, there is a clear need for more sophisticated preprocessing methods for code. Sentences in architecture documentation can map to hundreds of code files, because they refer to higher-level structures such as packages. To address this, we intend to investigate summarization techniques and clustering methods for code preprocessing. Lastly, we aim to improve the classification process during the prompting step. Currently, the decision to create a trace link is based solely on the LLM's classification. In the future, we will explore incorporating the LLM's reasoning into this decision-making process. We also plan to enhance prompts by integrating additional contextual information, such as project-specific or domain-specific knowledge.

VI. Conclusion

This paper presents a novel approach for Traceability Link Recovery (TLR) using Large Language Models (LLMs) with Retrieval-Augmented Generation (RAG). The benefit of our RAG-based approach Linking Software System Artifacts (LiSSA) is its applicability to various TLR tasks. LiSSA uses preprocessors to prepare artifacts according to their type to create traceable elements of the source and target artifacts. The approach then uses the embeddings of these elements to retrieve similar target elements for each source element. LiSSA then generates prompts for the LLM to classify if source and target elements are related. If classified as related, the approach aggregates the elements to finally create the trace links.

To answer our research questions, we evaluate LiSSA on established datasets and compare it to state-of-the-art approaches for three different TLR tasks.

Regarding **RQ1**, which examines the performance of RAG-based TLR compared to state-of-the-art, our evaluation shows that our RAG-based approach significantly outperforms state-of-the-art approaches for requirements to code TLR. For smaller projects, the RAG approach achieves significantly better F_1-scores than the state-of-the-art approach for documentation to code TLR. However, the approach underperforms for larger projects. For TLR from documentation to architecture models, LiSSA does not outperform the state-of-the-art.

For **RQ2** about the effectiveness of CoT prompting, we find that CoT prompting outperforms simple classification prompts in F_1-score. This is consistent with the findings by Rodriguez et al. [36].

Looking at the different preprocessing techniques and the impact of fine-grained mappings (**RQ3**), we show that the preprocessing of artifacts is, on average, not beneficial. In particular, the preprocessing does not improve the performance of the TLR from requirements to code. However, the best-performing approach is project-dependent.

Regarding **RQ4** that investigates the effects of the classification step, we showed that, on average, a LLM-based classification improves the performance over retrieval-only in all considered TLR tasks for both scores.

In summary, this research contributes to the field of traceability by providing a novel RAG-based approach for TLR tasks. We provide insights into how different prompt types and preprocessing techniques influence the performance. Our findings open up new research directions for the application of LLMs to TLR tasks. To ensure replicability, transparency, and extensibility, we provide the source code of LiSSA, the used datasets, and our results as part of our replication package [37]. We aim to facilitate further research in this field and want to enable other researchers to build upon our work.

Acknowledgements

This work was funded by Core Informatics at KIT (KiKIT) of the Helmholtz Assoc. (HGF) and the German Research Foundation (DFG) - SFB 1608 - 501798263 and supported by KASTEL Security Research Labs. Thanks to our textician.

References

[1] D. Falessi, J. Roll, J. L. Guo, and J. Cleland-Huang, "Leveraging historical associations between requirements and source code to identify impacted classes," *IEEE Transactions on Software Engineering*, vol. 46, no. 4, pp. 420–441, 2020.

[2] J. L. C. Guo, J.-P. Steghöfer, A. Vogelsang, and J. Cleland-Huang, "Natural Language Processing for Requirements Traceability," May 2024, arXiv:2405.10845 [cs]. [Online]. Available: http://arxiv.org/abs/2405.10845

[3] M. Rath, D. Lo, and P. Mäder, "Analyzing requirements and traceability information to improve bug localization," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 442–453. [Online]. Available: https://doi.org/10.1145/3196398.3196415

[4] P. Mäder and A. Egyed, "Assessing the effect of requirements traceability for software maintenance," in *2012 28th IEEE International Conference on Software Maintenance*, 2012.

[5] ——, "Do developers benefit from requirements traceability when evolving and maintaining a software system?" *Empirical Software Engineering*, vol. 20, pp. 413–441, 2015.

[6] J. Grundy, J. Hosking, and W. Mugridge, "Inconsistency management for multiple-view software development environments," *IEEE Transactions on Software Engineering*, vol. 24, no. 11, pp. 960–981, Nov. 1998, conference Name: IEEE Transactions on Software Engineering. [Online]. Available: https://ieeexplore.ieee.org/document/730545

[7] J. Keim, S. Corallo, D. Fuchß, and A. Koziolek, "Detecting inconsistencies in software architecture documentation using traceability link recovery," in *2023 IEEE 20th International Conference on Software Architecture (ICSA)*, Mar. 2023, pp. 141–152.

[8] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970–983, Oct. 2002. [Online]. Available: http://ieeexplore.ieee.org/document/1041053/

[9] A. Marcus and J. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *25th International Conference on Software Engineering, 2003. Proceedings.*, May 2003, pp. 125–135, iSSN: 0270-5257. [Online]. Available: https://ieeexplore.ieee.org/document/1201194

[10] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1, May 2010, pp. 95–104.

[11] M. Gethers, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "On integrating orthogonal information retrieval methods to improve traceability recovery," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, Sep. 2011, pp. 133–142.

[12] A. Mahmoud and N. Niu, "On the role of semantics in automated requirements tracing," *Requirements Eng*, vol. 20, no. 3, pp. 281–300, Sep. 2015.

[13] T. Hey, F. Chen, S. Weigelt, and W. F. Tichy, "Improving Traceability Link Recovery Using Fine-grained Requirements-to-Code Relations," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 12–22.

[14] H. Gao, H. Kuang, K. Sun, X. Ma, A. Egyed, P. Mäder, G. Rong, D. Shao, and H. Zhang, "Using Consensual Biterms from Text Structures of Requirements and Code to Improve IR-Based Traceability Recovery," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, Oct. 2022, pp. 1–1.

[15] J. Keim, S. Corallo, D. Fuchß, T. Hey, T. Telge, and A. Koziolek, "Recovering Trace Links Between Software Documentation And Code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639130

[16] D. Fuchß, H. Liu, T. Hey, J. Keim, and A. Koziolek, "Enabling Architecture Traceability by LLM-based Architecture Component Name Extraction," in *22nd International Conference on Software Architecture (ICSA)*. Institute of Electrical and Electronics Engineers (IEEE), 2025.

[17] J. Lin, Y. Liu, Q. Zeng, M. Jiang, and J. Cleland-Huang, "Traceability Transformed: Generating more Accurate Links with Pre-Trained BERT Models," in *Proceedings of the 43rd International Conference on Software Engineering*, ser. ICSE '21. Madrid, Spain: IEEE Press, Nov. 2021, pp. 324–335. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00040

[18] M. Rath, J. Rendall, J. L. C. Guo, J. Cleland-Huang, and P. Mäder, "Traceability in the wild: automatically augmenting incomplete trace links," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, May 2018, pp. 834–845. [Online]. Available: https://doi.org/10.1145/3180155.3180207

[19] L. Dong, H. Zhang, W. Liu, Z. Weng, and H. Kuang, "Semi-supervised pre-processing for learning-based traceability framework on real-world software projects," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 570–582. [Online]. Available: https://doi.org/10.1145/3540250.3549151

[20] M. Izadi, P. R. Mazrae, T. Mens, and A. van Deursen, "An empirical study on data leakage and generalizability of link prediction models for issues and commits," 2023. [Online]. Available: https://arxiv.org/abs/2211.00381

[21] P. R. Mazrae, M. Izadi, and A. Heydarnoori, "Automated recovery of issue-commit links leveraging both textual and non-textual data," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 263–273.

[22] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods," *IEEE Trans. Softw. Eng.*, vol. 32, no. 1, pp. 4–19, Jan. 2006.

[23] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically Enhanced Software Traceability Using Deep Learning Techniques," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, May 2017, pp. 3–14, iSSN: 1558-1225. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7985645

[24] A. Schlutter and A. Vogelsang, "Improving Trace Link Recovery Using Semantic Relation Graphs and Spreading Activation," in *Requirements Engineering: Foundation for Software Quality*, F. Dalpiaz and P. Spoletini, Eds. Cham: Springer International Publishing, 2021, pp. 37–53.

[25] Y. Shin, J. H. Hayes, and J. Cleland-Huang, "Guidelines for Benchmarking Automated Software Traceability Techniques," in *2015 IEEE/ACM 8th International Symposium on Software and Systems Traceability*, May 2015, pp. 61–67.

[26] G. Antoniol, J. Cleland-Huang, J. H. Hayes, and M. Vierhauser, "Grand Challenges of Traceability: The Next Ten Years," *arXiv:1710.03129 [cs]*, Oct. 2017. [Online]. Available: http://arxiv.org/abs/1710.03129

[27] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Evaluating large language models in class-level code generation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639219

[28] S. Gao, X.-C. Wen, C. Gao, W. Wang, H. Zhang, and M. R. Lyu, "What makes good in-context demonstrations for code intelligence tasks with llms?" in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 761–773.

[29] F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, and Q. Wang, "Clarifygpt: A framework for enhancing llm-based code generation via requirements clarification," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, jul 2024. [Online]. Available: https://doi.org/10.1145/3660810

[30] A. Singha, B. Chopra, A. Khatry, S. Gulwani, A. Z. Henley, V. Le, C. Parnin, M. Singh, and G. Verbruggen, "Semantically aligned question and code generation for automated insight generation," 2024. [Online]. Available: https://arxiv.org/abs/2405.01556

[31] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an llm to help with code understanding," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639187

[32] T. Ahmed, K. S. Pai, P. Devanbu, and E. Barr, "Automatic semantic augmentation of language model prompts (for code summarization)," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639183

[33] M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, and X. Liao, "Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning," in *Proceedings of the*

IEEE/ACM 46th International Conference on Software Engineering, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3608134

[34] Q. Huang, Z. Wan, Z. Xing, C. Wang, J. Chen, X. Xu, and Q. Lu, "Let's chat to find the apis: Connecting human, llm and knowledge graph through ai chain," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 471–483.

[35] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 9459–9474. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf

[36] A. D. Rodriguez, K. R. Dearstyne, and J. 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, pp. 455–464.

[37] D. Fuchß, T. Hey, J. Keim, H. Liu, N. Ewald, T. Thirolf, and A. Koziolek, "Replication Package: LiSSA: Toward Generic Traceability Link Recovery through Retrieval-Augmented Generation," Jan. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.14714706

[38] T. Dasgupta, M. Grechanik, E. Moritz, B. Dit, and D. Poshyvanyk, "Enhancing Software Traceability by Automatically Expanding Corpora with Relevant Documentation," in *2013 IEEE International Conference on Software Maintenance*, Sep. 2013, pp. 320–329, iSSN: 1063-6773. [Online]. Available: https://ieeexplore.ieee.org/document/6676903

[39] H. Gao, H. Kuang, W. K. G. Assunção, C. Mayr-Dorn, G. Rong, H. Zhang, X. Ma, and A. Egyed, "Triad: Automated traceability recovery based on biterm-enhanced deduction of transitive links among artifacts," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639164

[40] T. Hey, J. Keim, and S. Corallo, "Requirements Classification for Traceability Link Recovery," in *2024 IEEE 32nd International Requirements Engineering Conference (RE)*, Jun. 2024, pp. 155–167.

[41] K. Moran, D. N. Palacio, C. Bernal-Cárdenas, D. McCrystal, D. Poshyvanyk, C. Shenefiel, and J. Johnson, "Improving the effectiveness of traceability link recovery using hierarchical bayesian networks," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 873–885. [Online]. Available: https://doi.org/10.1145/3377811.3380418

[42] K. Nishikawa, H. Washizaki, Y. Fukazawa, K. Oshima, and R. Mibe, "Recovering transitive traceability links among software artifacts," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2015, pp. 576–580. [Online]. Available: https://ieeexplore.ieee.org/document/7332517

[43] A. D. Rodriguez, J. Cleland-Huang, and D. Falessi, "Leveraging intermediate artifacts to improve automated trace link retrieval," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 81–92.

[44] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "When and How Using Structural Information to Improve IR-Based Traceability Recovery," in *2013 17th European Conference on Software Maintenance and Reengineering*, Mar. 2013, pp. 199–208.

[45] H. Kuang, P. Mäder, H. Hu, A. Ghabi, L. Huang, J. Lü, and A. Egyed, "Can method data dependencies support the assessment of traceability between requirements and source code?" *Journal of Software: Evolution and Process*, vol. 27, no. 11, pp. 838–866, 2015, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.1736. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1736

[46] J. Cleland-Huang, R. Settimi, C. Duan, and X. Zou, "Utilizing supporting evidence to improve dynamic requirements traceability," in *13th IEEE International Conference on Requirements Engineering (RE'05)*, Aug. 2005, pp. 135–144, iSSN: 2332-6441. [Online]. Available: https://ieeexplore.ieee.org/document/1531035

[47] R. Lapena, F. Perez, C. Cetina, and O. Pastor, "Leveraging BPMN particularities to improve traceability links recovery among requirements and BPMN models," *Requirements Engineering*, vol. 27, no. 1, pp. 135–160, Mar. 2022. [Online]. Available: https://doi.org/10.1007/s00766-021-00365-1

[48] R. Lapena, F. Perez, O. Pastor, and C. Cetina, "Leveraging execution traces to enhance traceability links recovery in BPMN models," *Information and Software Technology*, vol. 146, p. 106873, Jun. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584922000398

[49] D. Fuchß, S. Corallo, J. Keim, J. Speit, and A. Koziolek, "Establishing a benchmark dataset for traceability link recovery between software architecture documentation and models," in *Software Architecture. ECSA 2022 Tracks and Workshops*, T. Batista, T. Bureš, C. Raibulet, and H. Muccini, Eds. Cham: Springer International Publishing, 2023, pp. 455–464.

[50] T. Hey, "Fine-grained Traceability Link Recovery (FTLR)," Sep. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8367392

[51] Center of Excellence for Software & Systems Traceability (CoEST) Datasets. [Online]. Available: http://sarec.nd.edu/coest/datasets.html

[52] J. Cleland-Huang, M. Vierhauser, and S. Bayley, "Dronology: an incubator for cyber-physical systems research," in *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 109–112. [Online]. Available: https://doi.org/10.1145/3183399.3183408

[53] J. Keim, S. Corallo, D. Fuchß, T. Hey, T. Telge, and A. Koziolek, "Replication Package for ICSE24 paper "Recovering Trace Links Between Software Documentation And Code"," Jan. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.10598371

[54] J. Cleland-Huang, O. Gotel, A. Zisman *et al.*, *Software and systems traceability*. Springer, 2012, vol. 2, no. 3.

[55] T. Hey, "Automatische Wiederherstellung von Nachverfolgbarkeit zwischen Anforderungen und Quelltext," Ph.D. dissertation, Karlsruher Institut für Technologie (KIT), 2023.

[56] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, p. 131, 2008.

[57] J. Sallou, T. Durieux, and A. Panichella, "Breaking the silence: the threats of using llms in software engineering," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER'24. New York, NY, USA: Association for Computing Machinery, 2024, p. 102–106. [Online]. Available: https://doi.org/10.1145/3639476.3639764