Establishing Traceability between Natural Language Requirements and Software Artifacts by Combining RAG and LLMs

Syed Juned Ali[1], Varun Naganathan[2], and Dominik Bork[1]

[1] TU Wien, Business Informatics Group, Vienna, Austria
{syed.juned.ali, dominik.bork}@tuwien.ac.at
[2] Microsoft, India, varun.naganathan@microsoft.com

Abstract. Software Engineering aims to effectively translate stakeholders' requirements into executable code to fulfill their needs. Traceability from natural language use case requirements to classes in a UML class diagram, subsequently translated into code implementation, is essential in systems development and maintenance. Tasks such as assessing the impact of changes and enhancing software reusability require a clear link between these requirements and their software implementation. However, establishing such links manually across extensive codebases is prohibitively challenging. Requirements, typically articulated in natural language, embody semantics that clarify the purpose of the codebase. Conventional traceability methods, relying on textual similarities between requirements and code, often suffer from low precision due to the semantic gap between high-level natural language requirements and the syntactic nature of code. The advent of Large Language Models (LLMs) provides new methods to address this challenge through their advanced capability to interpret both natural language and code syntax. Furthermore, representing code as a knowledge graph facilitates the use of graph structural information to enhance traceability links. This paper introduces an LLM-supported retrieval augmented generation approach for enhancing requirements traceability to the class diagram of the code, incorporating keyword, vector, and graph indexing techniques, and their integrated application. We present a comparative analysis against conventional methods and among different indexing strategies and parameterizations on the performance. Our results demonstrate how this methodology significantly improves the efficiency and accuracy of establishing traceability links in software development processes.

Keywords: Large Language Models · LLM · Requirements Traceability · Retrieval Augmented Generation · Requirements Engineering

1 Introduction

Traceability information is a fundamental prerequisite for many essential software maintenance and evolution tasks, such as change impact and software

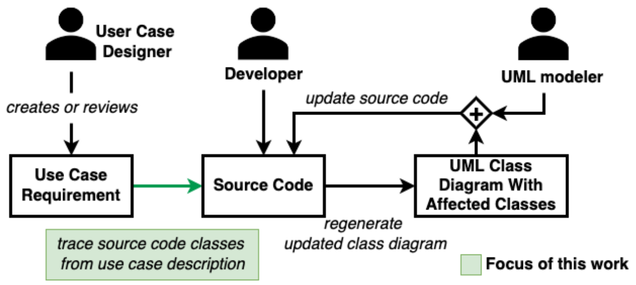


Fig. 1: Requirements to Code Traceability-based UML Classes Evolution

reusability analyses. Traceability can also help validate that the right system is being built, effectively meeting user and business needs as specified in the use case. A traceable relationship between requirements, conceptual design, and implementation helps effective communication about the software across stakeholders. However, manually maintaining traceability is costly and error-prone [14].

Unified Modeling Language (UML) class diagrams are a de-facto standard for the static structural representation of a system's design [3]. However, there can be several reasons for employing traceability from use case requirements to the classes in source code rather than UML class diagrams and then subsequently update the UML class diagram such as — *i*) Readily available and regularly updated source code particularly those involving legacy systems or open-source projects whereas UML class diagrams may not be regularly updated or might not exist at all, especially in agile development settings where documentation might lag behind code changes, *ii*) Richness of information in source code especially the availability of the business logic that the UML class diagram might lack, *iii*) Automatic generation of UML diagrams from source code using modern tools and IDEs (Integrated Development Environments) [3] and *iv*) Dynamic adaptation to system changes, thereby keeping the code to UML class diagrams synchronous with the latest system state.

Given that source code changes more frequently than most other artifacts during software development, traceability from source code ensures that the derived UML diagrams are kept synchronous with the latest system state and thereby helps maintain the relevance and usefulness of the UML diagrams as conceptual models of the system. Figure 1 illustrates the support that traceability from requirements to classes in source code provides for the evolution of UML class diagrams. Therefore, in this work, our approach focuses on improving the state-of-the-art of use case requirements to source code classes traceability, as a prerequisite for the approach in Fig. 1.

Automated approaches that utilize textual similarities between artifacts to establish trace links exist, however, these approaches tend to achieve low precision at reasonable recall levels, as they are not able to bridge the semantic gap between high-level NL requirements and code [14]. Recent advancements in Large Language Models (LLM) have marked a substantial progression in soft-

[3] `https://staruml.io/`, `https://pypi.org/project/pyreverse/`

ware engineering. The utilization of LLMs in software engineering can be effectively reframed into data, code, or text analysis tasks. The applicability of LLMs is particularly pronounced in tasks such as code summarization [35], which involves yielding an abstract NL depiction of a code's functionality, as well as the generation of well-structured code [40] and code artifacts like annotations [22].

Despite their ability to interpret the syntax and logic of programming languages, LLMs face huge challenges in applying this technical understanding to non-technical, use case-driven code descriptions. Documentation like code comments, while informative, often remains incomplete, overly technical, and detached from the actual code structure, and fails to consider the broader use case and context of the code's application. A recent survey on the use of LLMs in software engineering [15] highlights that LLMs often produce syntactically correct but functionally inadequate code, compromising the reliability and efficacy of LLM-based code generation. The survey reveals that LLMs are employed in software development in 58.37% of cases but only in 4.72% of requirements engineering scenarios. This discrepancy underscores LLMs' efficiency in generating syntactically accurate solutions when provided with clear requirements, yet indicates a gap in addressing ambiguities within requirements engineering.

LLMs are trained on extensive data, limiting their efficacy on tasks requiring specialized domain knowledge. Retrieval Augmented Generation (RAG) techniques enhance LLM's performance by utilizing external knowledge sources to augment the LLM's inherent data representation. This approach offers notable advantages over purely generative models: 1) knowledge is not merely encoded within model parameters but is dynamically incorporated in a scalable, plug-and-play fashion; and 2) it leverages human-authored texts for response generation, which simplifies the generation process and may improve output quality. As highlighted in the survey by Hou (2023) [15], the increasing complexity of tasks in this field necessitates more advanced, tailored computational strategies.

To address these crucial challenges, we introduce a novel **R**etrieval **A**ugmented **G**eneration (RAG)-based approach for code repository-specific **R**equirements **T**raceability (**RARTG**). Our approach is designed to bridge the gap between abstract, high-level use-case requirements and their corresponding source code. By leveraging code comments and the class functions dependency graph as contextual anchors for the RAG model, our solution provides an alignment of code with use-case requirements. The performance of an LLM with RAG depends on the quality of the RAG index. The RAG index acts as a contextual bridge between the query and the response from an LLM. The better the context provided by a RAG index for given requirements in natural language as a query is, the better an LLM responds by disambiguating the requirements. Therefore, we aim to study the impact of the quality of RAG index on RT by evaluating the impact of several parameters influencing the quality of RAG indexes.

In summary, we present our RARTG approach and provide a comprehensive evaluation of it on four requirements to code alignment datasets from [14] and provide a comparison of our approach with the existing work [14] (cf. Section 5) for the natural language requirements for the traceability task. The evaluation reveals our method's capability to outperform the state-of-the-art on the given datasets. Further our results show the impact of different data sources and the constructed RAG indexes for traceability, thereby showing that an improvement in the semantic quality of the RAG index leads to an improvement in requirements to code traceability. Finally, we provide the entire source code for the implementation of our approach.

In the remainder of this paper, Section 2 introduces relevant foundations to our approach. We present the related work in Section 3. In Section 4, we present the architecture and details of our RAG-based approach. In Section 5, we provide the evaluation of our approach. In Section 6 we discuss our results and findings and provide some insights and finally, we conclude in Section 7.

2 Background

Traceability in software engineering refers to the ability to link and trace the life of a requirement, forward and backward, throughout the stages of the Software Development Life Cycle. By establishing a clear and coherent connection between requirements and their corresponding implementation in the code, traceability not only supports effective project management but also provides a roadmap for future maintenance and upgrades. This end-to-end visibility is crucial in complex software projects, where understanding how each piece of code reflects specific requirements can significantly improve the efficiency and effectiveness of the software development process, e.g., connecting requirements to source code elements provides insight into what has been implemented and where [14].

Large Language Models - In the field of language processing, traditional Language Models (LMs) have been foundational elements, establishing a basis for text generation and understanding [28]. Increased computational power, advanced machine learning techniques, and access to very large-scale data have led to a significant transition into the emergence of LLMs. Equipped with expansive and diverse training data, these models have demonstrated an impressive ability to simulate human linguistic capabilities [15]. The training of LLMs involves learning patterns and structures in language by analyzing and predicting text sequences, which enable LLMs to generate coherent and contextually relevant responses. Their applications are diverse and impactful, ranging from natural language processing tasks like translation, summarization, and question-answering, to more creative uses such as content generation and dialogue systems [13,15]. LLMs have become integral in enhancing user experiences in virtual assistants, providing support in customer service through chatbots, and even assisting in writing and educational tools [19].

Retrieval Augmented Generation (RAG) is an approach in natural language processing that combines the strengths of information retrieval and language generation. In this methodology, a system first retrieves relevant information from a large dataset, like a database or the internet, and then uses this information to generate a response or output. RAG leverages a combination of information retrieval and neural network-based natural language generation to vectorize documents. The process involves two main components i.e., a *retriever* and a *generator*. The retriever is responsible for fetching relevant context from a large corpus of documents. The retrieval can be based on classical information retrieval techniques such as TF-IDF best matching (BM) algorithm [32], which is a ranking function used by search engines to estimate the relevance of documents to a given search query, or advanced vector embeddings-based similarity techniques. Vector-embedding-based retrieval techniques vectorize both the query and the documents as points in a shared continuous high-dimensional vector space. The embeddings are typically produced by models such as BERT [7] (Bidirectional Encoder Representations from Transformers) or its variants. These embeddings aim to capture deep semantic meaning of words and phrases in the context of the entire document, rather than inisolation. The generator, often an LLM, then takes the retrieved documents and the original query to generate a coherent response. The generator has been trained on a vast amount of text and thus has learned to predict the next word in a sequence, generating human-like text based on the context it is given. RAG allows the model to access a wide range of information beyond its training data, enabling it to provide more accurate, detailed, and contextually relevant responses.

3 Related Work

There has been significant work conducted in the area of applying natural language processing (NLP) and machine learning (ML) techniques for software requirements traceability. Several existing systematic literature reviews and mapping studies elaborate on the intersection of the RT and NLP or ML [38,31]. Hou et al. [15] provide a systematic mapping study at the intersection of LLMs and Software Engineering (LLM4SE). In the following, we first discuss the works involving traceability in UML models and source code, and then the role of LLMs in the context of RT. Finally, we discuss the role of context enrichment in improving RT.

3.1 Requirements Traceability in Models

Mills et al.[26] present an approach that supports the maintenance of traceability relations between requirements, analysis and design models of a software systems expressed in UML. Eyl et al. [9] present a metamodel expressing relationships between requirements and the UML model at the meta-level. For each meta-requirement, the author adds a 'REQTYPE' attribute to decide which UML diagram shall be used for the traceability. Netaji et al. [30] present a graph-based information retrieval approach to identify the requirement change impact on design models. Yazawa et al. [39] present an approach to derive a functional model from a use case diagram, a structure diagram, and a transition diagram. By decomposing the existing functional model into model components, traceability links are recovered based on guidelines that allow a mapping of model components to requirements. Divya et al. [8] present an approach to calculate the semantic traceability between the use case documentation and the sequence diagram. Khlif et al. [20] present an approach to support traceability between design, requirements, and code. Their approach extracts an expanded textual description from a natural language text available in UML models in order to trace between related elements belonging to requirements, design, and code while using an information retrieval technique.

3.2 Requirements Traceability in Source Code

Hey et al. [14] propose an approach for traceability link recovery by leveraging fine-grained, method-, and sentence-level similarities between the artifacts using word embeddings. Guerrouj et al. [12] present a solution that uses deep learning to incorporate requirements semantics and domain knowledge into the tracing solution. Tian et al. [34] adapt the word embeddings for traceability recovery tasks, and handle the out-of-vocabulary words involved in tracing words the artifacts that might not be in the vocabulary of an embedding model.

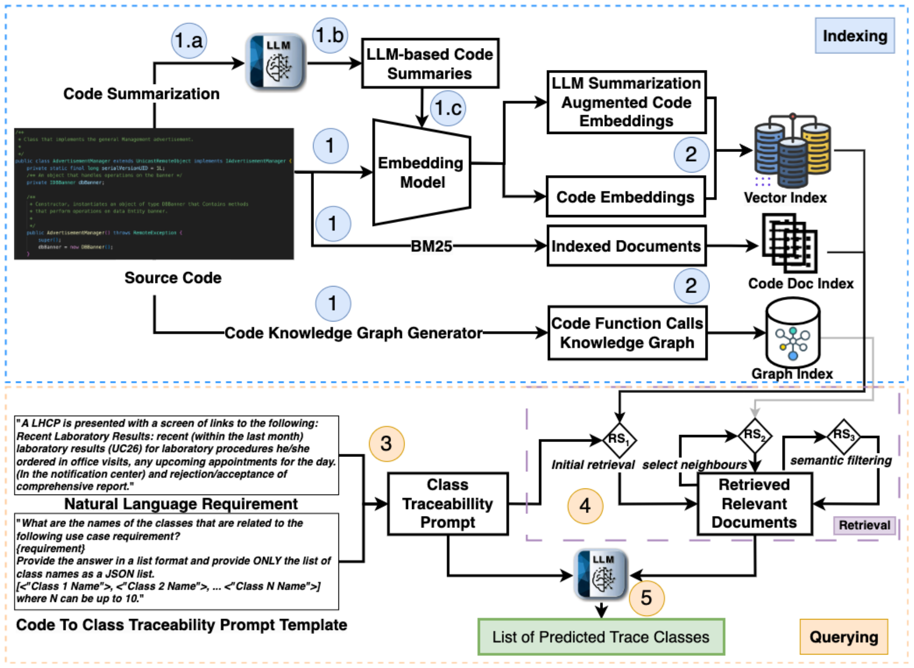
3.3 LLM-based Requirements Traceability in Source Code

Anaphoric ambiguity in software requirements arises when a single reader can interpret a natural language requirement in multiple ways, or different readers have varying understandings of the same requirement. Unclear and ambiguous software requirements can lead to suboptimal or even invalid software artifacts during later development stages. Moharil et al. [27] and Ezzini et al. [10] have empirically demonstrated the significant role of LLMs such as BERT [7] and SpanBERT [18] in effectively addressing anaphoric ambiguity. Lin et al. [23] found that T-BERT can effectively migrate knowledge from code search to NL artifacts to programming language artifacts traceability, even with limited training instances. It outperforms existing techniques in accuracy and can be adapted to different domains without intermediate training for each project, offering a promising step toward practical, trustworthy traceability. Sridhara et al. [33] revealed that ChatGPT excels in addressing anaphoric ambiguity in software requirements. ChatGPT consistently demonstrated its capability to accurately identify antecedents. These studies assert the valuable role of LLMs like GPT-4 [2] or the most capable open source LLMs like Llama 3 [4] can play in enhancing the clarity and precision of software requirements, thereby contributing to more effective software development with minimal interpretational uncertainties.

3.4 Context Enrichment for Requirements Traceability

Chen et al. [5] propose a self-enhanced automatic traceability link recovery approach based on structure knowledge mining for small-scale labeled data. This work enhances the semantic representations of artifacts by mining context information from the code structure. Lin et al. [24] exploit the idea that software source code contains a large amount of software-specific conceptual knowledge and semantic relatedness between queries and documents could be measured according to software-specific concepts involved in them. Iyer et al. [17] present a data-driven approach for generating high-level summaries of source code which ~~can later be~~ used as a semantic representation of the code for RT.

[4] `https://ai.meta.com/blog/meta-llama-3/`


Indexing
1.a
LLM
1.b
LLM-based Code Summaries
Code Summarization
1.c
LLM Summarization Augmented Code Embeddings
1
Embedding Model
2
Vector Index
Code Embeddings
1
BM25
Indexed Documents
Code Doc Index
Source Code
2
1
Code Knowledge Graph Generator
Code Function Calls Knowledge Graph
Graph Index
Natural Language Requirement
3
Class Traceabillity Prompt
RS_1
Initial retrieval
RS_2
select neighbours
RS_3
semantic filtering
4
Retrieved Relevant Documents
Retrieval
LLM
5
List of Predicted Trace Classes
Code To Class Traceability Prompt Template
Querying

```
/**
 * Class that implements the general Management advertisement.
 *
 */
public class AdvertisementManager extends UnicastRemoteObject implements IAdvertisementManager {
    private static final long serialVersionUID = 1L;
    /** An object that handles operations on the banner */
    private IDBBanner dbBanner;

    /**
     * Constructor, instantiates an object of type DBBanner that Contains methods
     * that perform operations on data Entity banner.
     *
     */
    public AdvertisementManager() throws RemoteException {
        super();
        dbBanner = new DBBanner();
    }
```

"A LHCP is presented with a screen of links to the following: Recent Laboratory Results: recent (within the last month) laboratory results (UC26) for laboratory procedures he/she ordered in office visits, any upcoming appointments for the day. (In the notification center) and rejection/acceptance of comprehensive report."

"What are the names of the classes that are related to the following use case requirement?
{requirement}
Provide the answer in a list format and provide ONLY the list of class names as a JSON list.
[<"Class 1 Name">, <"Class 2 Name">, ... <"Class N Name">]
where N can be up to 10."

Fig. 2: Retrieval Augmented Requirements Traceability Generation

3.5 Synopsis

A limitation of existing works in RT for models and source code include the fact that such works employ syntactical methods and guidelines for RT. Even the semantic approaches involve *shallow* semantics, i.e., semantics derived from textual similarity and therefore miss out on capturing the 'purpose' of the UML model or the code of the class and thereby cannot bridge the semantic gap between the UML model/code and the corresponding NL requirement.

In summary, we believe that none of the presented works use the benefits of RAG-based approaches for software RT. To the best of our knowledge, ours is the first work that combines RAG with LLM-based code summarization and combined (keyword, vector, and graph) indexing for RT. Our research contributes to the growing body of work seeking to bridge the gap between high-level requirements and executable code. By integrating the advances in LLMs with RAG techniques and exploiting the latent knowledge within codebases and the knowledge graph-based structural representation of software code, we provide a novel solution that tackles the challenges outlined by previous research while paving the way for more improvements in requirement-to-code traceability.

4 Retrieval Augmented Requirements Traceability Generation

Next, we present the details of our RAG-based approach for code repository-specific traceability. We present a novel architecture for enhancing traceability from natural language requirements to code through the utilization of Llama 3 and a suite of indexing and retrieval mechanisms. Our methodology is structured into several interconnected stages, as depicted in Fig. 2. The approach involves an *indexing stage* (steps 1 and 2) and a *querying stage* (steps 3, 4, and 5).

4.1 Indexing Stage

The indexing stage involves LLM-based code summary generation and the subsequent storage indexes creation from the the available codebase for efficient storage and retrieval of code documents, thereby facilitating quick and relevant matches, given a specific use-case requirement.

Code Summary Generation - There are cases where the classes *i*) do not have any docstrings; *ii*) the docstrings are incomplete; or *iii*) the docstrings are generic and do not capture the high-level purpose in the codebase and the relationship to other classes. To circumvent these issues, we use an LLM to generate the class docstring using the source code of the whole class. Note that all the subsequent prompts used in this paper are a result of rigorous prompt engineering, i.e., constructing different prompts and evaluating which prompt provides the best results. This task was entirely manual, and we chose the prompt that provided the best results. Below we show the code summarization prompt. The ⟨CODE_EXAMPLE⟩ and ⟨DOC_TEMPLATE⟩ are examples of code and structure of the document that are used to provide some contextual information for code summarization.

> You are an expert in summarizing code. Given a class in a Java code file, generate a class summary that can be used to map the code to a given use case requirement of the provided java code. The summary should capture the purpose of the class and its attributes and methods. Given a doc with the following structure that will be useful for use case requirements with code elements traceability: ⟨DOC_TEMPLATE⟩. Code example: ⟨CODE_EXAMPLE⟩ Provide the summary in simple abstract use case scenario language in the following format: Class Name: <Class Summary> Method Name: <Method Summary> . . Method Name N: <Method Summary N> Here is the code - ⟨code⟩

After LLM-based code summary generation, we create several code indexes.

Code Documents Keyword Index - We create a keyword-based index (KI) that treats the code documents as natural language documents and applies standard text preprocessing steps of stemming and stopwords, i.e., common frequent words removal from the code. We use the PorterStemmer [36] to stem the words. We store the generated index in a vector database. We create separate keyword indexes with and without code summaries in the code files.

Vector Index In this step, we use a multi-lingual embedding model [4]which is well suited for its versatility in Multi-Linguality to generate the text embeddings of the code documents and subsequently construct a vector index (VI). It can support more than 100 working languages, leading to new state-of-the-art performances on multi-lingual and cross-lingual retrieval tasks. We chose a multi-lingual embedding model because our dataset contains source code in two languages i.e., English and Italian. We store the generated embeddings of the code documents in a vector database. We create separate vector indexes with and without code summaries in the code files.

Knowledge Graph Index A knowledge graph index (KGI) is constructed from the dependency graph of the modules in the codebase. We refer to this knowledge graph as *Function Dependency Graph* (FDG). Each node of the FDG stores class names and the existing code documentation of the class as the class content. Next, we generate triples of subject, object, and a predicate where the subject is the calling method's class and the object is the called method's class. Finally, we store the created FDG in a Neo4j database to support querying.

Index Creation Subsequently, in step two, the system generates three specialized indexes as shown in Fig. 2. The first index is constructed using only the code comments and the names of the classes, class attributes and method names in the codebase. The second index is constructed using docstrings generated from an LLM as a code summarization response in the data embedding step and finally, the last index is for indexing the node embeddings from the FDG. The indexes generated using only the code docstrings do not consider any external sources of information or structural semantics from the code via interconnections from the code and only focus on the available semantics of the code from the code's textual content. The LLM-based indexes aim to improve on the first indexes by augmenting the documentation by an LLM-generated docstring, thereby including the LLM's knowledge as an external source of knowledge. However, this index still does not involve the structural semantics of the codebase. Therefore, the knowledge graph index aims to capture the structural semantics from the code. Furthermore, as performed in [14], we can optionally add the method call triples of a given class in the knowledge graph during the creation of the VIs and KIs. This step adds the structural connections explicitly to the indexes. However, as the authors in [14] showed, the effect of this addition on the traceability depends on the semantic quality of the connections with respect to the purpose of the class, which can result in an improved or worsened index quality. We investigate the impact of this step in our evaluation.

4.2 Querying Stage

In this stage, we use the constructed indexes of the indexing stage to first retrieve a set of documents relevant to the use case requirements and then subsequently using the contents of these documents as context to the LLM for predicting the classes relevant to the given requirement.

Requirement Traceability Prompt Generation. Upon receiving a NL requirement, in step three the system first uses a prompt template to construct a prompt that will be used to fetch the documents relevant to the query. The generated prompt is subsequently provided, along with the contents of the retrieved documents as context to the LLM to generate the final answer. For the purpose of requirements to code traceability, we constructed the following template where the use-case requirements is filled in the place of the 'requirements' placeholder.

> What are the names of the classes that are related to the following use case requirement? ⟨requirement⟩
>
> Provide the answer in a list format and provide ONLY the list of class names as a JSON list. [<"Class 1 Name">, <"Class 2 Name">, ... <"Class N Name">] where N can be up to 10.

Furthermore, a requirement that is too brief and lacks sufficient detail can result in poor recall of initial document retrieval. Therefore, to enhance the semantic quality of the initial requirements, we employ query expansion by generating semantically similar requirements. We use LLM to generate these similar requirements, which are then added as context to the prompt. However, this step does not guarantee improved traceability, as its effectiveness depends on the quality of the generated requirements. We investigate the impact of this step in our evaluation. After formulating the requirement-specific prompt, our system proceeds to fetch the relevant documents using various indexes from the indexing stage.

Relevant Documents Retrieval. The retrieval step takes place in step four in three stages denoted by retrieval stages RS_i in Fig. 2. In RS_1, the keyword and vector indexes are queries to give a combined set (union) of the relevant documents. In case of the VI, the retrieval is achieved by querying the indexes for vectors closest to the vector representation of the requirement. Note that the requirement is transformed into a vector using the same embedding model which is used to generate the code structural and semantic embeddings in step 1 so that the requirement shares the same vector space as the vector representation of the existing docstrings and LLM-generated docstrings. This initial selection using the keyword and vector indexes can retrieve semantically similar documents, however this approach may miss out on the structurally connected relevant documents, thereby reducing the recall of the approach. Therefore, in RS_2, the structurally connected neighbouring documents of the retrieved relevant documents in the knowledge graph are added to the list of relevant documents. This forms the content of all candidate documents that can be given as a context to the LLM. However, this further has few limitations. Firstly, retrieving a higher number of nodes (all the neighbours in this case) can affect the precision of our approach by retrieving false positives. Secondly, a context created from a large number of documents does not fit the maximum prompt size allowed for the LLM. To mitigate these limitations, in RS_3, we add a semantic filter that selects only the documents with a semantic similarity higher than a predefined threshold. Note, that this threshold is more relaxed than the threshold for document retrieval in RS_1 to allow keeping the documents that were not selected in RS_1 but not entirely semantically unrelated. In summary, our retrieval mechanism combines multiple indexes to construct a unified context and provide that as input to the LLM for preparing the response.

Response Generation. In the final step, the LLM receives the contextualized prompt, which includes the query, i.e., the natural language requirement and the relevant code elements and documentation. The LLM utilizes this enriched context to formulate a response that aims to predict the classes relevant to the requirement.

5 Evaluation

Now, we elaborate on an extensive evaluation of our approach. First, we provide the description of the experimental setup involving the investigated research questions, the datasets description and evalution metrics, and finally the results.

5.1 Research Questions

We aim to investigate the following research question with our approach:

- **[RQ.1]** How does RARTG perform compared to the state-of-the-art?
- **[RQ.2]** How do various parameterizations influence the traceability performance?
  - **[RQ2.1]** How does incorporating the LLM generated summaries influence traceability?
  - **[RQ2.2]** How does incorporating the structural knowledge graph index influence traceability?
  - **[RQ2.3]** How does query expansion by adding similar requirements to the initial requirements influence traceability?
  - **[RQ2.4]** How does incorporating method calls explicitly to the VI and KI influence traceability?
  - **[RQ2.5]** How does incorporating combining all the three indexes influence traceability compared to a single index (VI or KI)?

5.2 Experimental Setup

Datasets: We use the datasets with the descriptions given in Table. 1. The table shows the language of the use-case descriptions and the code artifacts. EN and IT indicate if the language of the artifact is English or Italian. The programming language for all the datasets is Java, however, the identifiers, such as class and method names can be in Italian as well. We see that for two datasets, eTour and iTrust, the artifacts are in English and for the remaining two datasets, the artifacts are in Italian. These datasets are provided by the Center of Excellence for Software & Systems Traceability (CoEST) [1] and are commonly used in automated traceability link recovery [14,25,29]. These datasets are suitable for our study as they provide trace links between NL requirements and source code. Note that iTrust is composed of Java and JSP target artifacts. However, we only consider the Java target artifacts and links here.

Table 1: Datasets

Project	**Domain**	**Language**			**Codebase description**		
		Use-case	**Code**	**Docstrings**	**# Usecases**	**#Classes**	**#Links**
eTour	Tourism	EN	Java-EN	EN	58	116	308
iTrust	Healthcare	EN	Java-EN	EN	131	226	286
SMOS	Education	IT	Java-IT	IT	67	100	1044
eAnci	Governance	IT	Java-IT	IT	139	55	567

RARTG Configuration Parameters: There are five controlling parameters that can affect the performance of RARTG: *i*) Use Generated Summary (GS) - which controls if the indexes use the LLM-generated summaries during indexing, *ii*) Use KG Index (KGI) - which controls if document retrieval from KGI using RS_2 and RS_3 is performed or not during RAG context generation, *iii*) Use Method Calls (MC) - which controls if method calls are explicitly added as strings in the documents during indexing stage, *iv*) Use Query Expansion (QE) - which controls if extra semantically similar requirements are generated and added to the context before response generation using LLM and *v*) Combine Indexes (CI) - which controls if all the three indexes are used together or not. These five parameters are used to respond to the five sub-RQs of RQ2. We execute the experiments on all the combinations of the values of these five parameters to evaluate the impact of these parameters. We term a single combination of values of all the parameters a single *configuration*.

Evaluation Metrics: In all our experiments we use the precision, recall and F1-score to evaluate our approach. The F1-score is the preferred metric for classification tasks and is commonly used for traceability link recovery [14]. It is defined as the harmonic mean of precision and recall. In case of traceability link recovery, precision reveals how accurate an approach proposes correct trace links. It measures the ratio of correctly proposed links to all proposed links. Recall shows the ability of an approach to propose all correct links. It measures the share of expected trace links that were actually found by an approach. High F1-scores should be the goal for all automated traceability link recovery approaches [11], as they indicate the approaches' ability to produce the expected results without missing links and producing many false positives. In our work, we not only focus on the F1-score but also on precision individually due to the research gap that indicates that existing approaches perform poorly, specifically in precision (cf. [14]).

5.3 Results

In the following, we provide the results of our experimental evaluation and respond to the RQs.

Response to RQ1: In order to respond to RQ1, we show the performance of our approach regarding the selected evaluation metrics in Table. 2. Out of all the different configuratins, we selected the *configuration* that gives the best F1-scores. We consider the results from [14] on the same four datasets as the baseline. The results in Table. 2 show, that the top two performing configurations of

Table 2: Traceability comparison across evaluation datasets

Dataset	**iTrust**			**eANCI**			**SMOS**			**eTour**		
Approach	**Prc**	**Rec**	**F1**	**Prc**	**Rec**	**F1**	**Prc**	**Rec**	**F1**	**Prc**	**Rec**	**F1**
Baseline	0.176	0.353	0.235	0.294	0.220	0.252	0.443	0.297	**0.356**	0.411	0.623	**0.495**
RARTG-C2	0.284	0.286	0.285	0.737	0.178	0.287	0.526	0.118	0.192	0.488	0.239	0.321
RARTG-C1	**0.289**	0.292	**0.290**	**0.779**	0.199	**0.317**	**0.608**	0.126	0.209	**0.543**	0.242	0.334

Prc: Precision, Rec: Recall

RARTG outperform the baseline for iTrust and eANCI datasets for F1-scores. Our approach improved the state-of-the-art F1-scores by almost 6% for both datasets. In case of the SMOS and eTour dataset, our approach underperforms with respect to the F1-score due to insufficient recall scores. Moreover, given that our approach focused on improving the precision scores for traceability based on the research gap, it is important to note that our approach consistently sufficiently outperforms the baseline with respect to precision. Furthermore, given that we used a single multi-lingual embedding model, it is important to note that our approach is robust to the language of the artifacts. This makes our approach independent of any translation step to English. In summary, these results clearly show the feasibility and value of our approach to support precise natural language to code traceability.

Response to RQ2: We performed a statistical evaluation of the F1-scores for different configurations. We had five controlling RARTG configuration parameters i.e., GS, MC, QE, KGI, and CI. To evaluate the impact of each of these parameters, we evaluate the change in F1-score by changing the value of the parameter in question and keeping the remaining parameters fixed. We do this exercise for all the combinations of the fixed values for each parameter. For e.g., to evaluate the impact of using LLM generated summaries, we calculated the change in F1-scores with and without using GS by keeping a particular combination of the values of other parameters fixed. We calculate the change in F1-scores for all possible combinations and calculate the T-statistic and P-values [21] to determine the significance of change in F1-scores. A p-value of less than 0.05 indicates a significant impact on the F1-score. The t-statistic indicates a significant positive or negative impact depending upon the sign of the t-statistic. Table. 3 shows the impact of all five parameters on the F1-scores. The KGI and CI parameters have two rows because we evaluate the impact of using KGI on top of an already existing VI in one case, and an already existing KI in another. Similarly, we evaluate the impact of using CI compared to configurations using single VI or single KI.

***Impact of GS*:** The p-values for GS in Table. 3 show a significant positive impact of adding LLM generated summaries to the F1-scores for iTrust and eANCI dataset. Note that our approach also outperforms for these two datasets. This indicates using LLM-generated summaries that capture the high-level purpose can be a very useful contrary to using the technical code docstrings. Adding LLM summaries do not have a significant impact on SMOS and eTour datasets,

Table 3: Impact of parameter change on the F1-score

Parameter	**iTrust**			**eANCI**			**SMOS**			**eTour**		
	TS	**P**	**Effect**	**TS**	**P**	**Effect**	**TS**	**P**	**Effect**	**TS**	**P**	**Effect**
GS	2.60	0.02	**SP**	5.48	<0.05	**SP**	0.40	0.70	-	1.60	0.13	-
MC	-2.36	0.03	**SN**	2.24	0.04	**SP**	3.22	<0.05	**SP**	1.78	0.09	Pos.
QE	-4.77	<0.05	**SN**	1.72	0.10	-	0.29	0.77	-	-3.14	0.01	**SN**
KGI to VI	25.84	0.02	**SP**	-0.31	0.77	-	0.36	0.73	-	15.75	<0.05	**SP**
KGI to KI	0.18	0.87	-	0.40	0.70	-	12.24	<0.05	**SP**	3.08	0.02	**SP**
CI to VI	6.34	0.00	**SP**	0.58	0.58	-	1.74	0.12	-	7.96	<0.05	**SP**
CI to KI	-0.19	0.86	-	3.42	0.01	**SP**	16.32	<0.05	**SP**	3.06	0.02	**SP**

SP: Significant Positive, SN: Significant Negative, -: Inconclusive, Pos. - Positive
TS: T-statistic, P: p-value

however the t-statistic still shows a slight positive impact which indicates an overall positive potential of adding LLM generated summaries.

***Impact of MC*:** The p-values for the impact of adding method calls directly to indexes show a mixed result, i.e., for iTrust we see a significantly negative impact and for eANCI and SMOS, we see a significantly positive result. This indicates that using method calls information directly in the document indexes may or may not be beneficial depending on the dataset and the relatedness of the semantic quality of the function calls with the purpose or the associated use-cases of the classes.

***Impact of QE*:** The p-values for the impact of QE are significantly negative on two out four datasets. This is an interesting result because intuitively we would expect that adding semantically related information to an already existing document would improve the retrieval of semantically related documents, however, our results indicate a negative impact on the performance. This indicates that query expansion by adding more questions may add more noise to the data that reduces the semantic similarity of a NL use-case requirement with code. Furthermore, the impact of QE may be dataset dependent as well. A dataset may require use-case language to be even syntactically close to the code. For e.g., a use-case requirement extract from the eTour dataset - "Use case name: DeleteCulturalHeritage. View the list of CulturalHeritage as a result of the use case SearchCulturalHeritage" directly hints towards a match with code with class names based on the keywords. In such cases keyword-based retrieval methods may outperform retrieval methods based on vector-based methods, therefore, adding extra information information may act as noise.

***Impact of KGI*:** The p-values for the impact of using KGI with VI and KI show, that KGI can have a positive impact as can be seen for iTrust, SMOS and eTour datasets. This result is quite expected because our RAG-based approach gave lower recall scores because of its strict semantic or even keyword-based similarity threshold, whereas a knowledge graph index adds structurally connected documents to the list of relevant documents that may not necessarily be related based on the semantic or keyword similarity and thereby missed by VI or KI.

***Impact of CI*:** The p-values for the impact of using a combined index comprising of all three indices shows an overall positive impact. Using a CI with

VI shows a significant improvement in iTrust and eTour datasets and using CI with KI shows a significant improvement for three out of four datasets. This result underpins the value of our approach that combines the different indexes for relevant documents retrieval and subsequent querying the LLM for extracting the requirement-specific classes from the retrieved documents.

6 Discussion

In the following, we discuss our results regarding both of our research questions and provide some meta-insights that we learned from our experiments.

Our results showed, that LLMs can be used to enrich the existing documentations which lack the contextual semantics to fill the abstraction gaps as pointed out in [16]. In our work, we exploited LLM's code summarization capabilities to associate high-level descriptions and tasks related to the abstract description of the source code thereby creating a contextual summary of the code which enabled improved traceability. This is particularly useful in cases where the code is not well documented and hence the semantic gap between the technical and business layers is high. We note that LLMs have shown a remarkable expertise in code generation from natural language descriptions [41]. However, semantically ambiguous descriptions can lead to incorrect code generation. Using our approach to incorporate contextual, semantically enriched summaries with a high-level description can enable an unambiguous natural language description to code generation. Finally, we note that combining a structural and a semantic index is beneficial and we learnt that a knowledge graph representation of the code provides significant value toward requirements to code traceability.

While investigating the reason for low recall scores, we analysed the RAG pipeline. The effectiveness of RAG-based applications heavily relies on the quality of RAG index such that for a given query, all the information relevant for answering the query should not only be retrieved but also fit into the maximum allowed prompt length for an LLM (e.g., 4096 tokens for Llama 3). This limits the number of documents that can be retrieved and hence it becomes imperative for the RAG-based approach to have a strict relevance check on the indexed documents. Due to this, a number of moderately relevant documents are missed. However, this also means that the document that are retrieved have a higher relevance accuracy. This can be preferable over higher recall in a setting like RT which ensures that the links established between requirements and code are accurate and reliable, reducing the likelihood of irrelevant connections that can mislead developers and complicate maintenance and verification processes [6].

Nevertheless, the lower recall underscores potential areas for future research and refinement of our model, indicating a clear path for progressive enhancements. As part of our future work, we aim to improve the recall scores by relaxing the initial documents retrieval constraints to retrieve more documents and then employing domain-specific criteria to perform a relevant information retrieval from the retrieved documents itself. This approach involves a layered documents retrieval that incrementally builds a more relevant, requirement-specific context from a larger set of documents, thereby not compromising with recall. We aim to make our approach accessible by making the source code openly available on GitHub [5]. Our tool provides support for creating indexes, code summarization, text embddings, setting up multiple retrieval mechanism including vector, keyword and knowledge graph index so that users can simply extend our implementation for their solution.

Lastly, assessing threats to validity is essential to ensure the reliability and generalizability of your findings. We use the four different types of validity threats as defined by Wohlin et al. [37]. If the generated code summaries or indexed documents miss information critical to traceability, then the construct validity is threatened. We mitigated this threat by manually refining prompts and examining the generated code summaries and their alignment with the code purpose. We mitigated the construct validity threats on the indexes by evaluating the different indexes on four datasets for the traceability quality. The dataset quality of the initial traceability requirements dataset threatens the internal validity of our work. We mitigated this threat by choosing quality evaluated datasets used in literature [1]. If the datasets are not diverse enough (e.g., all from similar types of software or similar domains), the results might not be widely applicable. Our work mitigates this threat by choosing all the datasets from different domains. The precision, recall, and F1 metrics need to be reliably measured to ensure that the observed effects are real and not due to measurement errors or randomness. We mitigated this error by repeating the experiments by making the index creation reproducible and fixing a seed and temperature value for an LLM to make the results reproducible.

7 Conclusion

In this work, we presented a novel RAG-based approach that enhances traceability from natural language use-case requirements to code repositories. We leveraged the synergy between code comments and the code dependency tree within the RAG framework, coupled with a technique to generate improved code summarizations. After an empirical evaluation of our approach, we have demonstrated an effective method for bridging the divide between high-level business requirements and the corresponding technical code constructs. The empirical evaluation reveals that a configuration that integrates class names and LLM-generated code summaries with the KGI created from class function calls improves tracing code from natural language requirements. Further, we evaluated the impact of using different indexes and their combination thereof and provided a statistical evaluation to analyze the impact of each parameter on RT. In the future, we aim to extend our solution to develop a tool for tracking a UML class diagrams evolution and impact analysis with changing requirements, thereby providing an feedback during design time about the history and impact of the requirement on the software artifacts.

[5] `https://github.com/junaidiiith/nl2codeTrace`

References

1. Center of excellence for software & systems traceability (coest). `http://sarec.nd.edu/coest/datasets.html` (2024), accessed: 3rd June 2024
2. Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F.L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al.: Gpt-4 technical report. arXiv preprint arXiv:2303.08774 (2023)
3. Booch, G., Rumbaugh, J.E., Jacobson, I.: The unified modeling language user guide - covers UML 2.0, Second Edition. Addison Wesley object technology series, Addison-Wesley (2005)
4. Chen, J., Xiao, S., Zhang, P., Luo, K., Lian, D., Liu, Z.: M3-embedding: Multi-linguality, multi-functionality, multi-granularity text embeddings through self-knowledge distillation. In: Findings of the Association for Computational Linguistics ACL 2024. pp. 2318–2335 (2024)
5. Chen, L., Wang, D., Shi, L., Wang, Q.: A self-enhanced automatic traceability link recovery via structure knowledge mining for small-scale labeled data. In: 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC). pp. 904–913. IEEE (2021)
6. De La Vara, J.L., Wnuk, K., Berntsson-Svensson, R., Sánchez, J., Regnell, B.: An empirical study on the importance of quality requirements in industry. In: SEKE. pp. 438–443 (2011)
7. Devlin, J., Chang, M., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis. pp. 4171–4186. Association for Computational Linguistics (2019). `https://doi.org/10.18653/V1/N19-1423`
8. Divya, K., Subha, R., Palaniswami, S.: Similar words identification using naive and tf-idf method. International Journal of Information Technology and Computer Science (IJITCS) **6**(11), 42 (2014)
9. Eyl, M., Reichmann, C., Müller-Glaser, K.: Traceability in a fine grained software configuration management system. In: Software Quality. Complexity and Challenges of Software Engineering in Emerging Technologies: 9th International Conference, SWQD 2017, Vienna, Austria, January 17-20, 2017, Proceedings 9. pp. 15–29. Springer (2017)
10. Ezzini, S., Abualhaija, S., Arora, C., Sabetzadeh, M.: Automated handling of anaphoric ambiguity in requirements: a multi-solution study. In: Proceedings of the 44th International Conference on Software Engineering. pp. 187–199 (2022)
11. Gotel, O., Cleland-Huang, J., Hayes, J.H., Zisman, A., Egyed, A., Grünbacher, P., Dekhtyar, A., Antoniol, G., Maletic, J.: The grand challenge of traceability (v1. 0). Software and systems traceability pp. 343–409 (2012)
12. Guerrouj, L., Bourque, D., Rigby, P.C.: Leveraging informal documentation to summarize classes and methods in context. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. vol. 2, pp. 639–642. IEEE (2015)
13. Hadi, M.U., Qureshi, R., Shah, A., Irfan, M., Zafar, A., Shaikh, M.B., Akhtar, N., Wu, J., Mirjalili, S., et al.: A survey on large language models: Applications, challenges, limitations, and practical usage. Authorea Preprints (2023)
14. Hey, T., Chen, F., Weigelt, S., Tichy, W.F.: Improving traceability link recovery using fine-grained requirements-to-code relations. In: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 12–22. IEEE (2021)

15. Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J.C., Wang, H.: Large language models for software engineering: A systematic literature review. CoRR **abs/2308.10620** (2023). `https://doi.org/10.48550/ARXIV.2308.10620`
16. Huang, Y., Liu, Z., Chen, X., Luo, X.: Automatic matching release notes and source code by generating summary for software change. In: 2016 6th International Conference on Digital Home (ICDH). pp. 104–109. IEEE (2016)
17. Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L.: Summarizing source code using a neural attention model. In: 54th Annual Meeting of the Association for Computational Linguistics 2016. pp. 2073–2083. Association for Computational Linguistics (2016)
18. Joshi, M., Chen, D., Liu, Y., Weld, D.S., Zettlemoyer, L., Levy, O.: Spanbert: Improving pre-training by representing and predicting spans. Transactions of the association for computational linguistics **8**, 64–77 (2020)
19. Kasneci, E., Seßler, K., Küchemann, S., Bannert, M., Dementieva, D., Fischer, F., Gasser, U., Groh, G., Günnemann, S., Hüllermeier, E., et al.: Chatgpt for good? on opportunities and challenges of large language models for education. Learning and individual differences **103**, 102274 (2023)
20. Khlif, W., Kchaou, D., Bouassida, N.: A complete traceability methodology between uml diagrams and source code based on enriched use case textual description. Informatica **46**(1) (2022)
21. Kim, T.K.: T test as a parametric statistic. Korean journal of anesthesiology **68**(6), 540 (2015)
22. Liang, Y., Zhu, K.: Automatic generation of text descriptive comments for code blocks. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 32 (2018)
23. Lin, J., Liu, Y., Zeng, Q., Jiang, M., Cleland-Huang, J.: Traceability transformed: Generating more accurate links with pre-trained bert models. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 324–335. IEEE (2021)
24. Lin, Z., Zou, Y., Zhao, J., Xie, B.: Improving software text retrieval using conceptual knowledge in source code. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 123–134. IEEE (2017)
25. Lohar, S., Amornborvornwong, S., Zisman, A., Cleland-Huang, J.: Improving trace accuracy through data-driven configuration and composition of tracing features. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. pp. 378–388 (2013)
26. Mills, C., Escobar-Avila, J., Haiduc, S.: Automatic traceability maintenance via machine learning classification. In: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 369–380. IEEE (2018)
27. Moharil, A., Sharma, A.: Tabasco: A transformer based contextualization toolkit. Science of Computer Programming **230**, 102994 (2023)
28. Moore, R.C., Lewis, W.: Intelligent selection of language model training data. In: Proceedings of the ACL 2010 conference short papers. pp. 220–224 (2010)
29. Moran, K., Palacio, D.N., Bernal-Cárdenas, C., McCrystal, D., Poshyvanyk, D., Shenefiel, C., Johnson, J.: Improving the effectiveness of traceability link recovery using hierarchical bayesian networks. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 873–885 (2020)
30. Nejati, S., Sabetzadeh, M., Arora, C., Briand, L.C., Mandoux, F.: Automated change impact analysis between sysml models of requirements and design. In: Pro-

ceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 242–253 (2016)

31. Pauzi, Z., Capiluppi, A.: Applications of natural language processing in software traceability: A systematic mapping study. Journal of Systems and Software **198**, 111616 (2023)
32. Robertson, S., Zaragoza, H., et al.: The probabilistic relevance framework: Bm25 and beyond. Foundations and Trends® in Information Retrieval **3**(4), 333–389 (2009)
33. Sridhara, G., Mazumdar, S., et al.: Chatgpt: A study on its utility for ubiquitous software engineering tasks. arXiv preprint arXiv:2305.16837 (2023)
34. Tian, Q., Cao, Q., Sun, Q.: Adapting word embeddings to traceability recovery. In: 2018 International conference on information systems and computer aided Education (ICISCAE). pp. 255–261. IEEE (2018)
35. Wan, Y., Zhao, Z., Yang, M., Xu, G., Ying, H., Wu, J., Yu, P.S.: Improving automatic source code summarization via deep reinforcement learning. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering. pp. 397–407 (2018)
36. Willett, P.: The porter stemming algorithm: then and now. Program **40**(3), 219–223 (2006)
37. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in software engineering. Springer Science & Business Media (2012)
38. Xu, C., Li, Y., Wang, B., Dong, S.: A systematic mapping study on machine learning methodologies for requirements management. IET Software **17**(4), 405–423 (2023)
39. Yazawa, Y., Ogata, S., Okano, K., Kaiya, H., Washizaki, H.: Traceability link mining - focusing on usability. In: 41st IEEE Annual Computer Software and Applications Conference, COMPSAC 2017,. Volume 2. pp. 286–287. IEEE Computer Society (2017). `https://doi.org/10.1109/COMPSAC.2017.254`
40. Yin, P., Neubig, G.: A syntactic neural model for general-purpose code generation. arXiv preprint arXiv:1704.01696 (2017)
41. Zan, D., Chen, B., Zhang, F., Lu, D., Wu, B., Guan, B., Yongji, W., Lou, J.G.: Large language models meet nl2code: A survey. In: Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 7443–7464 (2023)