Retrieval-Augmented Large Language Models for Traceability Link Recovery

Master's Thesis of

Niklas Ewald

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

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

08. January 2024 – 08. July 2024

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

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

Karlsruhe, 08. July 2024

. . . .

(Niklas Ewald)

Abstract

During the development of software many kinds of artifacts are created. Traceability information between them is important for development tasks, such as compliance with requirements, change impact analysis, and finding documentation inconsistencies. Manually establishing trace links is error-prone and costly. One challenge is the different abstraction levels artifacts can have. This thesis uses retrieval-augmented large language models to bridge this semantic gap and evaluates this approach on three different traceability link recovery tasks. Cost and processing time are kept low by only using the language model to classify artifact pairs containing similar artifacts. For requirement to source code traceability link recovery, the approach achieves results comparable to state-of-the-art approaches with F1 scores of 0.388 on SMOS, 0.478 on eTour, and 0.313 on iTrust. The approach does not reach the state-of-the-art for the software architecture documentation to software architecture model and software architecture documentation to source code tasks.

Zusammenfassung

Während der Entwicklung von Software werden viele Artefakte erstellt. Rückvervolgbarkeitsinformation zwischen ihnen sind wichtig für Aufgaben wie die Einhaltung von Anforderungen, Auswirkungsanalysen von Änderungen und das Auffinden von Dokumentationsinkonsistenzen. Das manuelle Erstellen von Rückverfolgbarkeitsverbindungen ist fehleranfällig und kostspielig. Eine Herausforderung besteht in den unterschiedlichen Abstraktionsebenen von Artefakten. Diese Arbeit verwendet Retrieval-Augmented Large Language Models, um diese semantische Lücke zu überbrücken, und bewertet diesen Ansatz in drei verschiedenen Aufgaben zur Wiederherstellung von Rückverfolgbarkeitsverbindungen. Kosten und Verarbeitungszeit werden niedrig gehalten, indem das Sprachmodell nur zur Klassifizierung von Artefaktpaaren verwendet wird, die Artefakte enthalten, die ähnlich zueinander sind. Für die Wiederherstellung von Rückverfolgbarkeitsverbindungen zwischen Anforderungen und Quellcode erzielt der Ansatz vergleichbare Ergebnisse zu dem Stand der Technik mit F1-Werten von 0.388 auf SMOS, 0.478 auf eTour und 0.313 auf iTrust. Für die Aufgaben der Rückverfolgbarkeitsverbindungen von Softwarearchitekturdokumentation zu Softwarearchitekturmodellen und von Softwarearchitekturdokumentation zu Quellcode erreicht der Ansatz nicht den Stand der Technik.

Contents

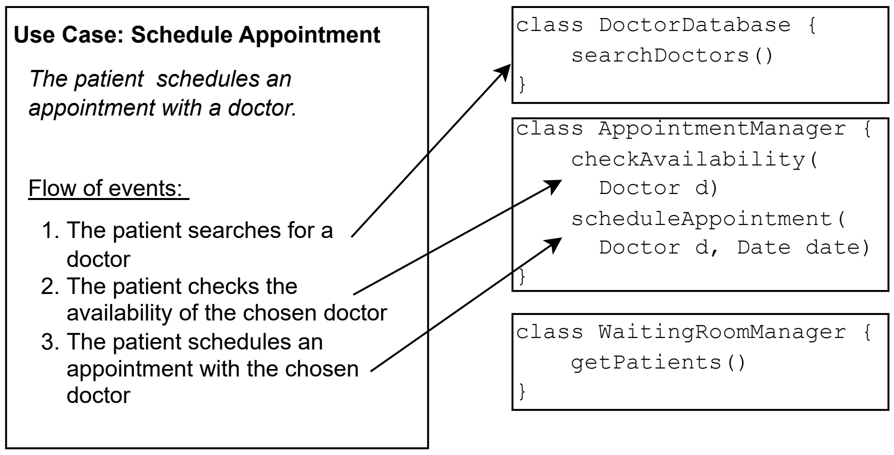
List of Figures

List of Tables

1 Introduction

There are several artifacts created during the development of a software project. Among others, artifacts include requirements, design documents, source code, and test cases. Usually, one artifact is the refinement of another artifact on a different level of abstraction. Understanding the relationships between artifacts is an important part of developing and maintaining software. Trace links help with tasks such as change impact analyses and software reusability analyses [5]. For safety-critical software, traceability might even be required [31]. Establishing trace links by hand is a costly and error-prone task. Therefore, trace links often are not documented. Different approaches to automatically recover trace links have been proposed. They often use information retrieval or machine learning methods. Methods to automatically recover trace links between artifacts do not reach high enough precision for acceptable values of recall, and therefore cannot be used in practice. As different kinds of artifacts use different levels of abstraction to describe the same information, there exists a semantic gap. It has been shown that large language models can achieve good results in several tasks like translation, summarizing text, and some logic tasks. They have shown to possess reasoning abilities and an understanding of language, which might help in bridging the semantic gap. There is some research applying large language models to trace link recovery: Lin et al. created Trace BERT, showing promising results for commit-to-code traceability link recovery [24]. Rodriguez et al. show their iterative approach for creating a prompt suitable for trace link recovery [36]. Large language models are typically pre-trained on large amounts of data. Even though, it is reasonable to assume any particular software project is not contained within the training data. Context about the project has to be included when using the language model. Large language models have a maximum amount of tokens for their input, and it is unreasonable to use all artifacts in a single prompt. Also, a single artifact often only relates to a small subset of other artifacts. Therefore, the amount of non-related artifact pairs is quite high.

This thesis continues the trend to leverage large language models, but in the form of retrieval-augmented large language models. Instead of comparing all possible combinations of source and target artifacts, only a subset of those pairs is processed by the language model. Target artifacts are stored in a way in which it is possible to quickly find the most similar ones to a query artifact. Figure 1.1 shows an example containing a use case involving three steps and two classes containing methods related to the use case. Even though the classes are not written in natural language it is obvious that they are related to the use case. A third class `WaitingRoom` on the other hand is not related. Using the large language model to classify only a subset of all possible pairs of query and target artifacts might lead to lower costs and less time spent processing artifacts. The approach tries to be as generic as possible. This means it is designed in a way where it is not specific to the


Use Case: Schedule Appointment
The patient schedules an appointment with a doctor.
Flow of events:
1. The patient searches for a doctor
2. The patient checks the availability of the chosen doctor
3. The patient schedules an appointment with the chosen doctor

```
class DoctorDatabase {
    searchDoctors()
}
```

```
class AppointmentManager {
    checkAvailability(
      Doctor d)
    scheduleAppointment(
      Doctor d, Date date)
}
```

```
class WaitingRoomManager {
    getPatients()
}
```

Figure 1.1: Example of a use case and related classes.

type of artifact that is under consideration. At the same time, it is easy to extend to allow such functionality if desired. The thesis shall answer how well the retrieval-augmented large language model approach performs and whether using a large language model has any benefit over an embedding based information retrieval approach. Another aspect to explore is, whether this approach is works for all types of artifacts or if it performs worse for some than for others.

The thesis is structured in the following way: In Chapter 2 the fundamentals needed to understand this thesis are explained. After, in Chapter 3 datasets commonly used in traceability research are presented. Chapter 4 discusses the idea of the retrieval-augmented large language model traceability link recovery approach, while Chapter 5 presents how it is implemented. Following, other approaches to the same and related problems are shown in Chapter 6. Then the evaluation of this work follows in Chapter 7. This thesis concludes with a summary and an outlook to future work in Chapter 8.

2 Fundamentals

This chapter will provide an overview of relevant topics that are required to understand this thesis proposal. It is split into two main parts. section 2.1 will contain the information needed to understand the problem we are trying to solve and section 2.2 discusses the foundations of the tools we plan to use.

2.1 Traceability Link Recovery

During the software development process, several documents are created. These documents, for example, requirements, software architecture descriptions and models, source code, and test cases, are called artifacts. Between two artifacts, there can be a trace link. Gotel et al. define trace links as "a specified association between a pair of artifacts, one comprising the source artifact and one comprising the target artifact." [5]. Where the source artifact is from where the trace originates and the target artifact is the trace link leads. The target artifact, for example, might "implement", "test", or "refine" the source artifact [5].
Another characteristic of trace links is the granularity which is decided by the granularity of the source and target artifacts. For example, a trace link between a sentence of a requirement and a method from a source code document has a finer granularity than a whole requirement and a whole class [10].
Traceability can be established manually by a person. It can be established automatically, where deciding between which artifacts to create trace links is automated. The process can also be semi-automated, where an automated tool may suggest trace links that have to be verified by a human [5].
Gotel et al. differentiate between trace capture and trace recovery. Trace capture refers to when the trace link is created at the same time as the artifacts to which it relates. Trace recovery describes the approach when the creation of the trace link happens after the creation of the artifacts it relates to [5].

2.2 Large Language Model

Large Language Models are a machine-learning approach that trains on large amounts of data. They are statistical language models that are used to generate text, for machine translation, question answering, summarization, and other natural language tasks [20].The

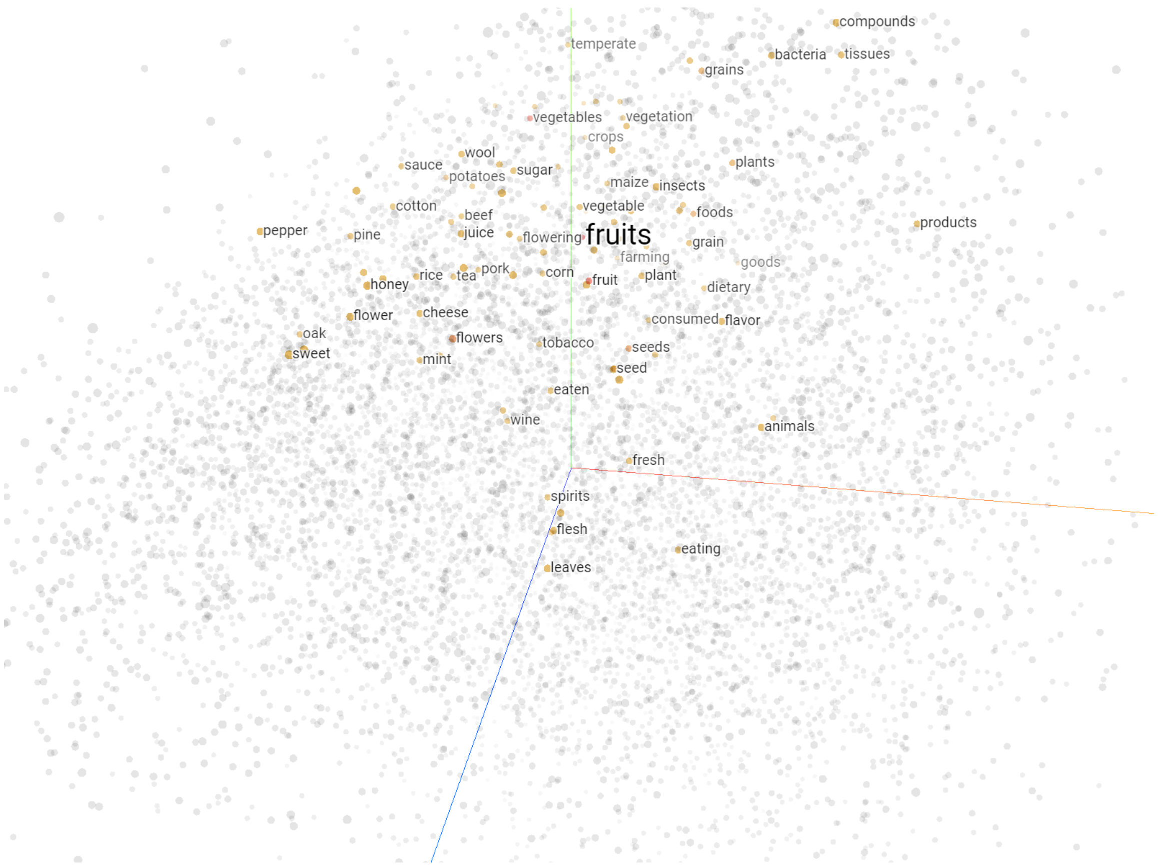


Figure 2.1: Visualization of word2vec embeddings using TensorFlow's projector tool [7].

input of a large language model usually is tokenized into tokens which are then processed by the model. A token represents a common sequence of characters, often a word or a part of a word. For example, the tokenizer of the model GPT-4 splits the sentence "The patient checks the availability of the chosen doctor." into "The", " patient", " checks", " the", " availability", " of", " the", " chosen", " doctor", "." [43]. The first "The" in the sentence is mapped to a different token than the following two "the". Large language models that are offered as a service often have costs calculated depending on the number of tokens used [33] [28].

2.2.1 Embeddings

An embedding is a vector representation of some kind of information without losing its original meaning [37]. Artificial Intelligence models, like large language models, are used to create them [38]. There are several kinds of embeddings to create, such as words, sentences, or whole documents. For example for word embeddings, embeddings of words with a similar meaning such as "fruits" and "vegetables" might be close to each other. Word2vec is a frequently used embedding model used to generate word embeddings. Figure 2.1 shows a three-dimensional representation of several word embeddings generated with it. The highlighted words are the nearest words in the original high-dimensional vector space.

Given a collection of embeddings that are stored in a vector database, to find relevant data we need some kind of query. An embedding is calculated for this query and the database is queried for similar embeddings. The are different similarity measures that can be used [38]. For cosine similarity, the cosine of the angle between two vectors is measured which leads to a result between -1 and 1. The Euclidean distance measures the length of a line between them. 0 means the same vector and higher values represent more different vectors.

2.2.2 Prompting

When transformer-based large language models were first introduced, the main way to adapt them to downstream tasks was fine-tuning. A language model was trained on a smaller dataset created specifically for the chosen task.
In recent years prompting became a popular method to solve tasks [40] [25]. The user provides a template describing the task in natural language. "Does this text contain information about [topic]? [text]" is a simple prompt that could be used in a classification task. The quality of results can vary quite a lot based on the choice of words and type of prompt. To increase the likelihood of generating better results one can use several techniques, some of which will be presented here.

Zero-shot prompting
With zero-shot prompting the prompt just describes the task itself.
Example: "Add 3 and 4: "

Few-shot prompting
A few-shot prompt includes examples of how to answer the questions.
Example (2-shot):
"Add 3 and 4: 7
Add 1 and 3: 4
Add 5 and 3: "

Chain-of-Thought prompting [44]
The language model is prompted to add reasoning to its prediction. This can happen in the form of 1-shot prompting where the example answer contains reasoning or in the form of zero-shot prompting by adding text along the lines of "let's think step by step." [19] to the prompt.

Prompt Chaining [45]
The idea of chaining is to break down the task into smaller but well-scoped sub-tasks. Sub-tasks can be completed by independent language model calls. Answers will be used in another prompt.

3 Datasets

To evaluate how well the approach works a number of publicly available datasets is used. First, the datasets used in the use case to source code traceability are presented in section 3.1. section 3.2 covers the datasets containing software architecture documentation, architecture models and source code.

3.1 Use Cases and Source Code

For the use case to source code traceability task, Table 3.1 shows some general information about the used datasets. SMOS, eTour and iTrust are commonly used to compare the effectiveness of different traceability link recovery approaches [30, 10]. They are part of a dataset collection of the Center of Excellence for Software & Systems Traceability (CoEST)[2]. The chosen datasets cover different domains. The gold standards map use cases to source code. All three datasets contain java source files. ITrust additionally contains Jakarta Server Page files. Since some preprocessing steps target Java files specifically only part of the iTrust dataset containing java files is used. For a similar reason, a translated version of the eTour dataset is used. The original version contains syntax errors which make it difficult to extract features like class signatures, method signatures, and method bodies. For the English version the Italian identifiers in the source code and the Italian use case names were translated into English[11]. In the process the syntax errors within the source files were fixed. The SMOS dataset used contains Italian use case description and comments within the source code, but identifiers are English. Since the language models used have multilingual capabilities, there was no need to use a translated version.

Section 3.1 contains information about the size of the datasets and about the gold standards. The coverage shows what proportion of the artifacts is mentioned at least once in the gold standards. A low coverage might hint to incomplete gold standards. For the used datasets, iTrust has the lowest coverage of target artifacts with just 38.5 %.

Project	Domain	Language	Programming Language
SMOS	Education	IT/EN	Java
eTour	Tourism	EN	Java
iTrust	Healthcare	EN	Java

Table 3.1: Overview of the used datasets for use case to source code TLR.

Project	Source Artifacts	Target Artifacts	Links	Source Cov.	Target Cov.
SMOS	67	100	1044	1.000	0.680
eTour	58	116	308	0.983	0.767
iTrust	131	226	286	0.802	0.385

Table 3.2: Information About the Artifacts and the Gold Standards Contained Within the Datasets Used for Use Case to Source Code TLR.

Project	SAD Sentences	SAM Elements	Code Files
BigBlueButton	85	24	547
JabRef	13	6	1987
MediaStore	37	23	97
TEAMMATES	198	16	834
TeaStore	43	19	205

Table 3.3: Number of Artifacts for Each Project Used in Tasks Involving Software Architecture Documentation.

3.2 Software Architecture Documentation, Software Architecture Models and Source Code

For traceability link recovery task concerning software architecture documentation, architecture models, and code five datasets are used, that were originally provided by Fuchß et al. [8] and later extended by Keim et al. [17] to include source code. They cover open source projects from several domains.

- BigBlueButton is a web conference system.
- JabRef is a tool to manage citations.

	SAD-SAM			SAD-Code		
Project	Links	SAD Cov.	SAM Cov.	Links	SAD Cov.	Code Cov.
BigBlueButton	52	0.482	0.458	1295	0.459	0.446
JabRef	18	0.769	0.833	8240	0.769	0.980
MediaStore	29	0.757	0.435	50	0.676	0.155
TEAMMATES	51	0.202	0.5	7610	0.439	0.993
TeaStore	27	0.535	0.316	707	0.535	0.707

Table 3.4: Number of Trace Links in the Gold Standard and Coverage Used in Task Involving Software Architecture Documentation.

- MediaStore is a model application modeled after the iTunes store.
- TEAMMATES is an application to manage feedback in education.
- TeaStore is a microservice reference application emulating an online store.

Each dataset includes software architecture documentation in natural language and software architecture models as an XML containing a UML model. For some projects multiple documentation files and models are included. In this case architecture documentation and architecture models from the same year are used. All datasets contain documentation written in English and use English identifiers in the source code. The datasets also contain gold standards mapping sentences of architecture documentation to an architecture model element and gold standards mapping the same sentences to source code files. Sometimes, a sentence is mapped to a folder or package in which case all contained source files are considered to have a link.

Table 3.3 lists the number of artifacts for each project while Table 3.4 contains the number of trace links and the coverage they achieve.

4 Approach

In this chapter the approach to recover traceability links using retrieval-augmented large language models is presented. The approach relies on large language models to find traceability links between software artifacts. Due to the nature of the traceability link recovery task, it is not always feasible to compare every possible artifact combination. Usually, an artifact is only related to a relatively small number of other artifacts. This number is dependent on the specific project and the established practices for the project. Instead of having the language model compare all possible source and target artifact combinations, only the query artifacts and potential target artifacts are compared. For this information retrieval techniques are combined with the large language model. Retrieval augmented generation is a method to use external knowledge without relying on the implicit knowledge stored in the parameters of a language model. If there is new information, the model does not have to be retrained and instead a relevant document can be included as context when prompting the language model. The retrieval of relevant artifacts can be seen as a filtering step. In general, there are two main methods to use retrieval-augmented generation. The first approach usually lets the language model decide if and how retrieval should be used. For example, a search engine that searches on Wikipedia could be used in case the language model does not have enough information to answer a question. This approach requires multiple calls to the language model and therefore is slower and more expensive. The advantages, however, are the language model decides if the external information is necessary, and it can write its own query for additional information. As neither the source nor the target artifacts are assumed to be an explicitly trained part of the language model, additional context in the form of potential target artifacts is always needed and the main benefit of the advanced approach is lost. Therefore, the more simple approach is used.

The approach can be separated into three steps, that will be explained in the following sections.

4.1 Preprocessing

Gotel et al. [9] define a trace artifact as "A traceable unit of data (e.g., a single requirement, a cluster of requirements, a UML class, a UML class operation, a Java class or even a person). A trace artifact is one of the trace elements and is qualified as either soruce or as a target artifact when it participates in a trace. The size of the traceable unit of data defines the granularity of the related trace." In this thesis, the term *artifact* generally is used to refer to unprocessed pieces of data between which a trace link may or may not

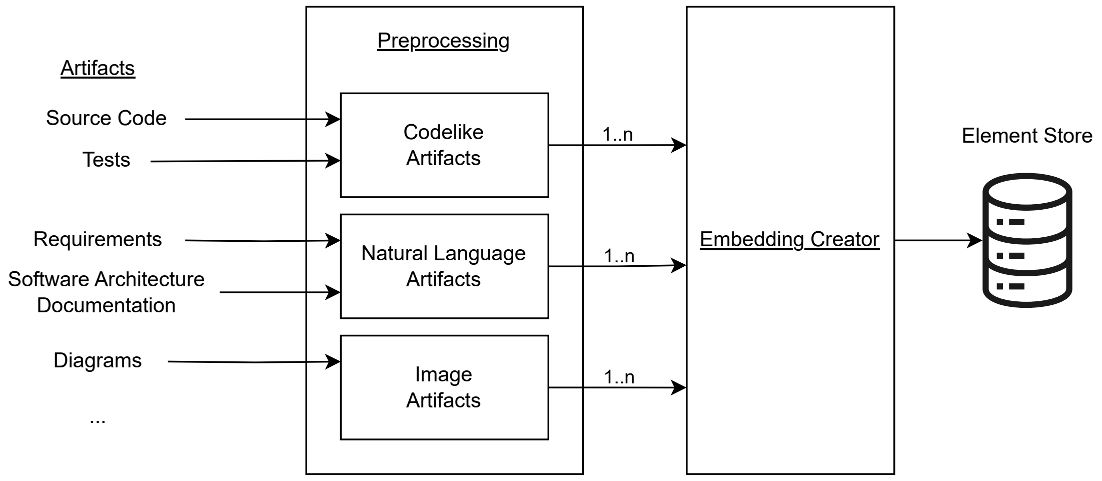


Figure 4.1: The preprocessing step. Each artifact is transformed into one or more elements. For each element an embedding is generated.

be classified, such as the files contained within a project or dataset. The term *element* is used to refer to an artifact or a part of an artifact that has been transformed in some way to be used with the retrieval augmented large language model traceability link recovery approach. One or multiple elements represent a transformed, or preprocessed artifact. The first step of the retrieval-augmented large language model traceability link recovery approach, is show in Figure 4.1. Artifacts are transformed into elements, and for each element an embedding is calculated.

The way in which an element is created depends on the type of the artifact. The simplest method is to not do anything at all. Most textual artifacts can be kept as is. The text contained within the artifact will be the text used in the element. This way all information contained within the artifact can be used by the language model at the same time. In some cases the resulting input will be too large for the used language model, in which case the input needs to be shortened. Another preprocessing method, used for textual artifacts, is to split them into smaller pieces. Depending on the type of the artifact, different ways to split it into elements can be used. Natural language text, commonly used in requirements or software architecture documentation, can be split into sentences. Artifacts containing source code, can be split into methods. In addition to the method signature and body, the lines between methods can be included in order to not lose surrounding context. This includes comments such as method documentation, but can also be class variables. Textual artifacts can also be split into chunks of a specified maximum size. When used with large language models, this chunk size is usually defined in tokens and the result, therefore, is dependent on the tokenization method used for a language model.

For each element, an embedding is calculated. To have comparable embeddings between different elements the embedding model is fixed for each run. It does not differ between source and target artifacts. The embedding and the associated element are stored for later retrieval.

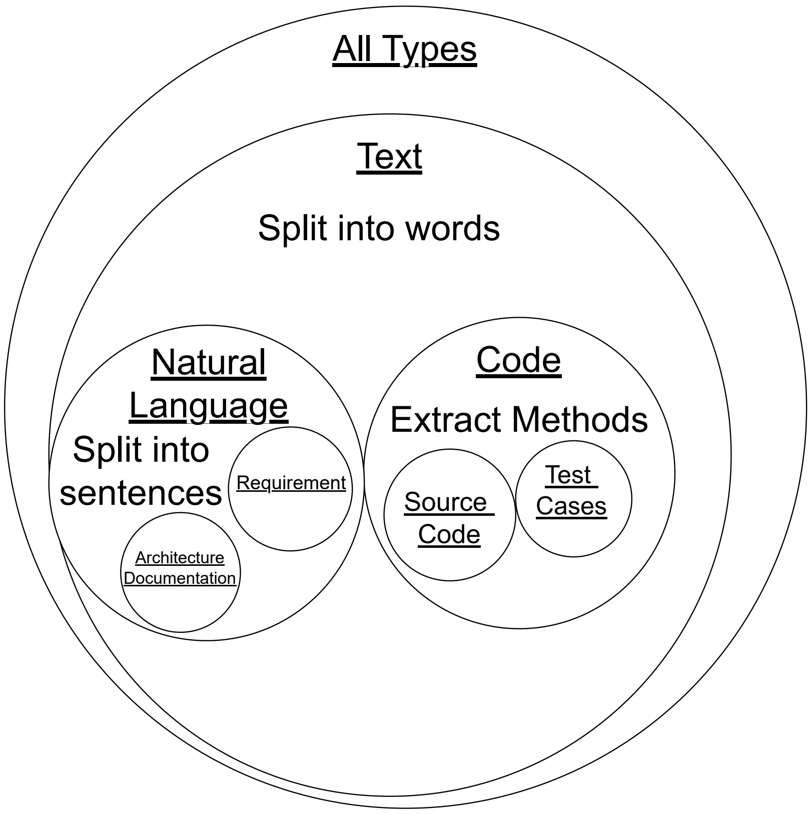


Figure 4.2: Example of possible preprocessing steps shared by different artifact types.

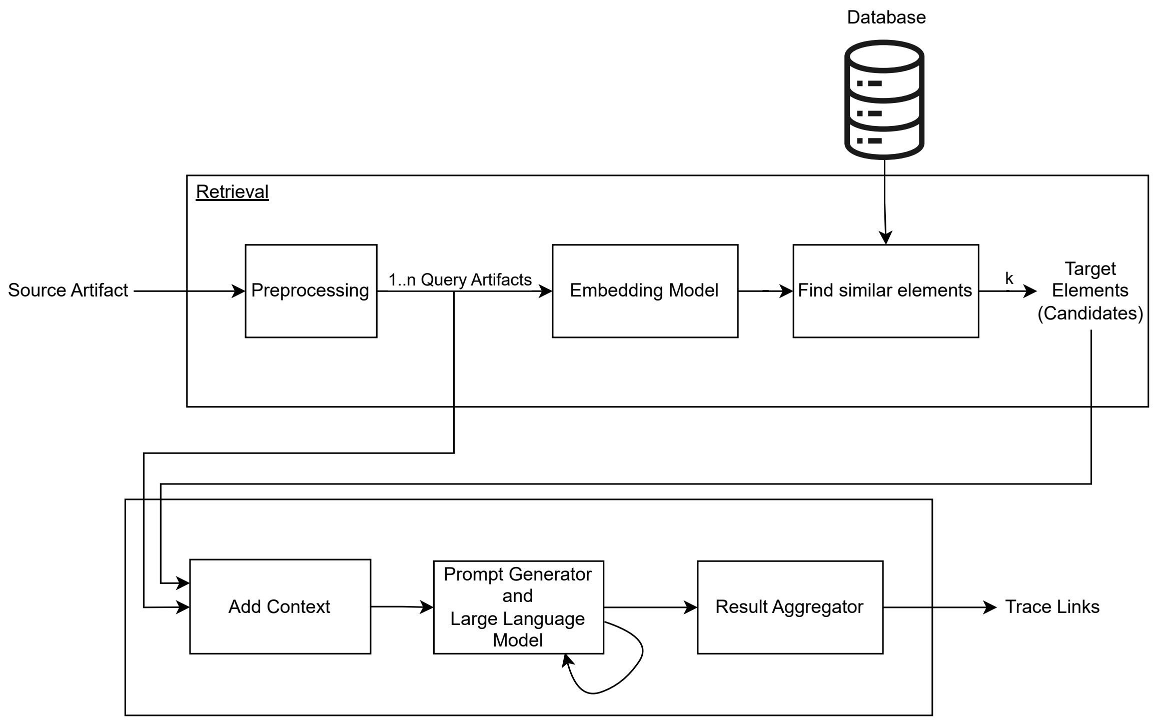


Figure 4.3: The process for retrieving target candidates and for recovering trace links.

4.2 Candidate Retrieval

The second step is the retrieval step. During this step candidate elements are retrieved. Candidate elements are the most similar target elements to a source element. The source element and the target elements are the elements, that will be processed by the language model. As figure Figure 4.3 shows, only one source element is considered at once. The source element is used as a query to find target elements that are similar. Originally, elements were supposed to be stored in a vector store to leverage fast retrieval of elements. However, fast comparisons and retrieval are often achieved through approximate algorithms. To improve reproducibility, the source elements are compared to all target elements. This might lead to slow retrieval with very large projects or preprocessing steps which generate a large number of elements. For the datasets used in this thesis comparing the embeddings of every source element to the embedding of each target element is acceptable since speed is not a main concern.

4.3 Traceability Link Recovery

The final step is the classification step. A large language model is prompted to decide whether a source element and a target element are related. For this prompt templates are used in which the elements are filled.

Just like with the embedding model, this approach is not designed with a specific large language model in mind. However, to keep the amount of configurations manageable only OpenAI's GPT-3.5-turbo model, specifically gpt-3.5-turbo-0125 [29], was used. GPT-4-turbo was considered. In a small scale test on a subset of the used datasets, it performed slightly worse than GPT-3.5-turbo. Regarding the higher costs compared to GPT-3.5-turbo, only the latter is used in experiments.

There is an optional context fetching phase before prompting the language model. This can include fetching additional information about the project, such as a description. For elements created by splitting artifacts, it can also be elements which surrounded the source or target element in the artifact.

4.4 Traceability Link Recovery Tasks

Traceability link recovery is the task to find pairs of artifacts created during a development process that belong together. They can have relationships such as refinements, implementations, verifications or represent a different point of view to each other. One challenge of traceability link recovery is different levels of abstraction between artifacts, which does not just manifest in different words used for the same things, but also in different forms an artifact can appear in. While the approach discussed in this thesis is not limited to specific traceability link recovery tasks, only the following artifact types are discussed.

4.4.1 Requirements to Source Code

A task commonly studied is the recovery of trace links between requirements and source code. It involves mapping a high-level requirement, often written in natural language, to low-level source code, written in a programming language. Requirements describe "A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents" [13]. A common way to express requirements is through natural language [46]. There is no fixed form a requirement has to be in. One possible form used are use cases, that describe a scenario of how the software system is to be used. But simple descriptions of how a system shall behave exist as well. In contrast to requirements, source code is not written in natural language but a programming language. This results in a semantic gap between those artifact types. Additionally, a single requirement might be implemented in different parts of the source code. In a Java project for example, multiple classes might be related to a requirement.

4.4.2 Software Architecture Documentation to Software Architecture Model

The task of software architecture documentation to software architecture model traceability link recovery, involves finding trace links between different documents outlining the architecture of a software project. Software architecture documentation describes the system's architecture in natural language. It provides an overview of components and their interactions. Software architecture models on the other hand, are a formal representation of the system's architecture using a modeling language such as UML. An architecture model often only contains basic information about software components, such as names and realized interfaces and what other components they interact with. Compared to other artifact types, such as source code, the amount of immediately available information is low.

4.4.3 Software Architecture Documentation to Source Code

Like the requirements to source code traceability link recovery, the software architecture to source code traceability link recovery task involves finding trace links between high-level natural language text and low-level source code. The challenge of this task is bridging the gap between the high-level design descriptions and the source code containing the actual low-level implementation.

5 Framework

This chapter will explain the framework created for this thesis. It will show which concepts presented in Chapter 4 are supported and how they are implemented.

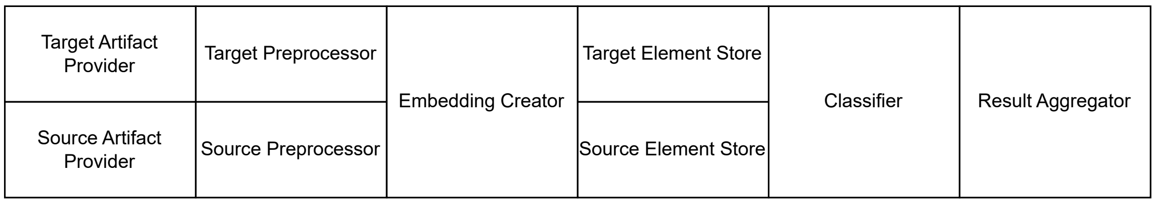


Figure 5.1: Structure of the Pipeline.

The framework uses a fixed pipeline design. Each step has a fixed function which is realized using different modules that follow the interface of the step. The general structure of the pipeline is illustrated in Figure 5.1. For any given module, the step to the previous needs to be concluded, while some steps are independent of each other.

A controller manages the data transfer between different modules. Figure 5.2 shows the controller is connected to all pipeline modules, while the modules themselves are mostly independent of other modules to increase maintainability and reusability.

5.1 Data Classes

Different modules within the framework use several data classes. The most important ones are `Knowledge` and `Element`. `Knowledge` represents any kind of information that might be used to determine the existence of trace links. `Element` is a subclass of `Knowledge`. As the name suggests it represents an Element. As explained in Chapter 4 it is created by transforming an Artifact. Section 5.3 explains this in more detail. As Figure 5.3 shows, a `Knowledge` object possesses a `content` attribute. In most cases this is some kind of text. However, it is intended to be extendable to include other kinds of data, such as images or perhaps sound. Besides the original artifact content or a part of it, as it is mostly used in `Element` objects, it could contain a description of the software project. `Knowledge` also possesses an `identifier` and a `type` attribute. `type` stores information about what kind of information is represented. The `identifier` attribute is a unique text used to identify a specific piece of information while the program is running.

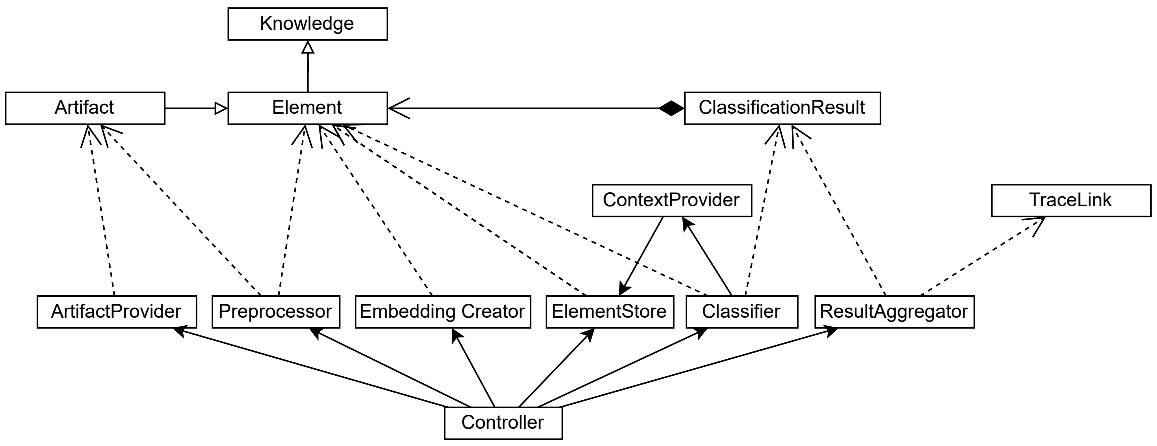


Figure 5.2: The coarse structure of the framework. Implementations of pipeline modules are not shown.

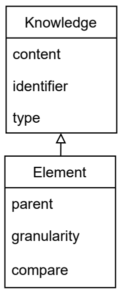


Figure 5.3: Data classes Knowledge and Element.

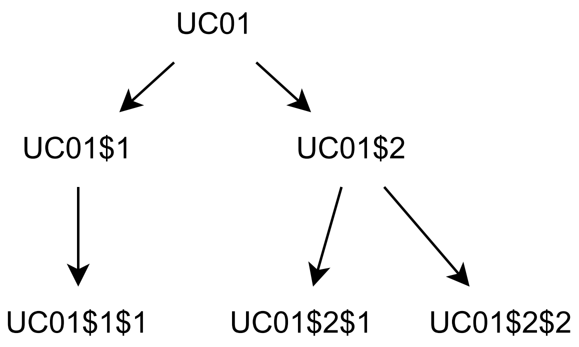


Figure 5.4: An example fo how a tree of preprocessed elements might look like.

In addition to the attributes inherited by `Knowledge`, `Element` adds three additional ones. For a given `Element` `parent` points to the Element that was used to create it. The value of `granularity` shows how fine the `Element` is. This allows for pipeline modules to work on `Elements` of specific granularity. The `granularity` of an `Element` with the content of an original artifact is `0`. A higher `granularity` value represents a finer granularity. As it is not supported to work on a more coarse granularity than the original artifacts, the `granularity` of an `Element` is always greater than or equal to `0`. Therefore, child elements will be given a higher `granularity`. With the attribute `compare` a `Preprocessor` decides whether an `Element` should be considered when making similarity comparisons and whether they will be sent to a `Classifier`. `Artifact` inherits from `Element`.It represents the original inputs, such as the content of files containing source code, requirements, diagrams or other software artifacts. When creating an `Artifact` object, `compare` is set to `false`. They are never used for direct comparisons and only exist to be preprocessed or to be used as context.

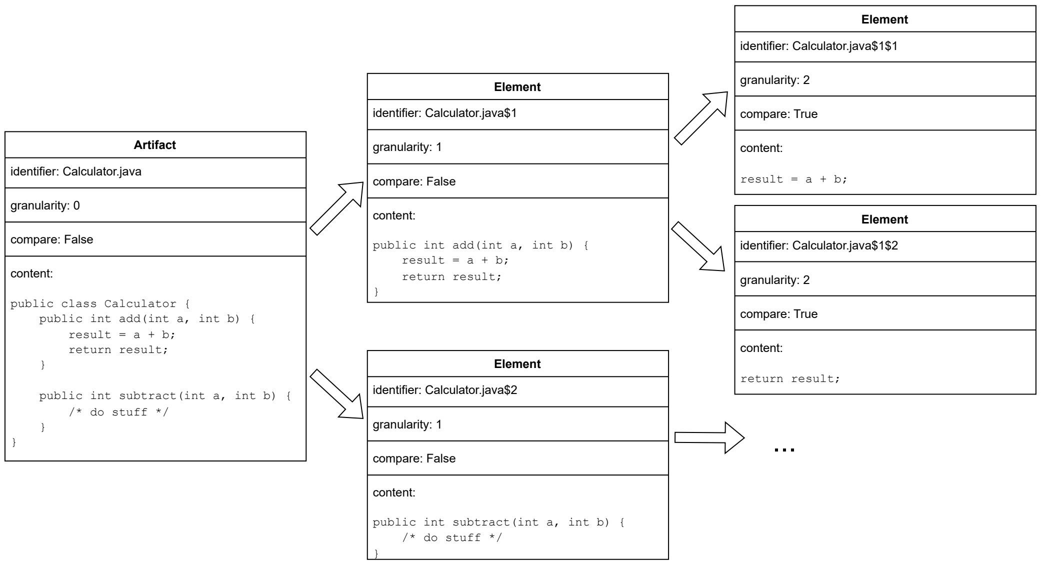
5.2 Artifact Provider

The artifact provider module is responsible to load files from the file system and create artifact objects. In its simplest form an artifact provider reads a text contained within one or more files and creates an artifact containing the same text. The choice what kind of artifact provider to use is dependent on how the dataset shaped. There are three artifact providers implemented for commonly used structures.

- The `TextArtifactProvider` simply treats each file in a given folder as a single artifact.
- The `DeepTextArtifactProvider` searches for files with specified extensions within subfolders of a given folder and treats them the same way the `TextArtifactProvider` does.
- The `SingleFileArtifactProvider` treats a single file as a single artifact. A corresponding preprocessor needs to split it into seperate elements.

5.3 Preprocessor

A preprocessor converts an artifact into one or more elements. It also carries out additional preprocessing actions, such as lemmatization or stop word removal. The main focus for this thesis lies in splitting artifacts into smaller chunks. The idea is, if embedding models and the large language model are trained on natural text and existing source code, it might be a hindrance to change the artifact content into an unnatural form.Different types of artifacts might need different preprocessors. For example a preprocessor which is meant to separate natural language text into sentences is not suitable for model descriptions in xml notation. A preprocessor splitting source code into methods can't work with natural text.



```
public class Calculator {
    public int add(int a, int b) {
        result = a + b;
        return result;
    }

    public int subtract(int a, int b) {
        /* do stuff */
    }
}
```

```
public int add(int a, int b) {
    result = a + b;
    return result;
}
```

```
public int subtract(int a, int b) {
    /* do stuff */
}
```

```
result = a + b;
```

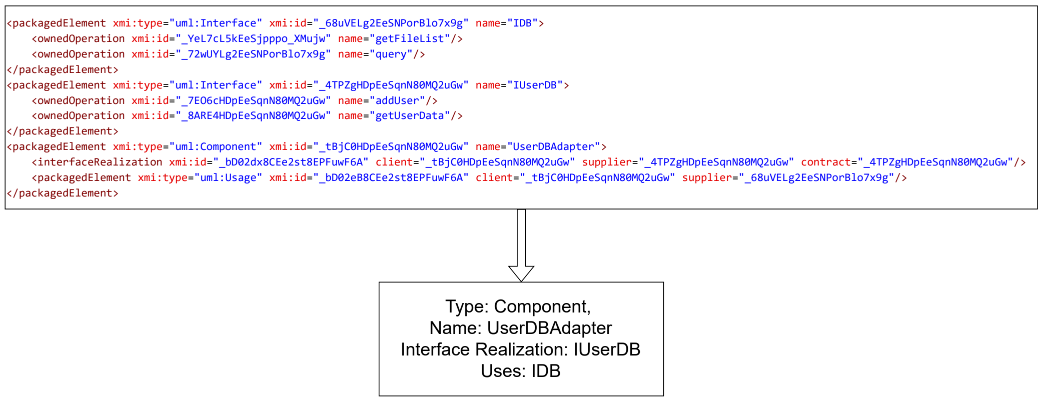
```
return result;
```

Figure 5.5: Example of a source code class artifact split into methods and then into single lines of code.

Figure 5.5 shows an example of how a Java class might be seperated into elements. First the `Calculator` class contained in the artifact `Calculator.java` is seperated into methods. Then the methods are split into single lines. The preprocessor can split the artifact into elements of different granularities. Usually, the granularity of a child element is the parent's granularity + 1, which can also be seen in the example. `Calculator.java$1` and `Calculator.java$2` have a granularity of 1. They are finer than the original artifact. Additionally, `Calculator.java$1$1` and `Calculator.java$1$2` have a granularity of 2. Often it is enough to have two levels of granularities: 0 for the original artifact and 1 used for the elements it was split into. If there is no preprocessing done, i.e. the content of the original artifact is used, only one granularity level is used.

As mentioned in Section 5.1, the preprocessor decides whether the `compare` attribute should be set. This differentiates an `Element` that is completely preprocessed and therefore meant to be part of the retrieval process form an `Element` that is only returned as possible context. In the example from Figure 5.5, the preprocessor returns {`Calculator.java`, `Calculator.java$1`}, `Calculator.java$2`, `Calculator.java$1$1`, `Calculator.java$1$2` and all `Elements` created from `Calculator.java$2`. Only the `Elements` containing a single code line will be used during the retrieval step. By returning and storing all `Elements`, it is ensured modules located at a later point in the pipeline can traverse the `Artifact`/`Element` tree, either as additional context or to aggregate the results correctly.

Each element has a unique identifier. Identifiers of children of the same parent should be in ascending order when natural sorting. This way later modules can use the surrounding context of an `Element`. In the implementation for this thesis, a child is given its parent's identifier concatenated with "$k" where k is increasing by 1 for each child of an `Element`. The '$' symbol was chosen due to being not used in artifact identifiers in any of the used datasets.



```
<packagedElement xmi:type="uml:Interface" xmi:id="_68uVELg2EeSNPorBlo7x9g" name="IDB">
    <ownedOperation xmi:id="_YeL7cL5kEeSjpppo_XMujw" name="getFileList"/>
    <ownedOperation xmi:id="_72wUYLg2EeSNPorBlo7x9g" name="query"/>
</packagedElement>
<packagedElement xmi:type="uml:Interface" xmi:id="_4TPZgHDpEeSqnN80MQ2uGw" name="IUserDB">
    <ownedOperation xmi:id="_7EO6cHDpEeSqnN80MQ2uGw" name="addUser"/>
    <ownedOperation xmi:id="_8ARE4HDpEeSqnN80MQ2uGw" name="getUserData"/>
</packagedElement>
<packagedElement xmi:type="uml:Component" xmi:id="_tBjC0HDpEeSqnN80MQ2uGw" name="UserDBAdapter">
    <interfaceRealization xmi:id="_bD02dx8CEe2st8EPFuwF6A" client="_tBjC0HDpEeSqnN80MQ2uGw" supplier="_4TPZgHDpEeSqnN80MQ2uGw" contract="_4TPZgHDpEeSqnN80MQ2uGw"/>
    <packagedElement xmi:type="uml:Usage" xmi:id="_bD02eB8CEe2st8EPFuwF6A" client="_tBjC0HDpEeSqnN80MQ2uGw" supplier="_68uVELg2EeSNPorBlo7x9g"/>
</packagedElement>
```

Type: Component,
Name: UserDBAdapter
Interface Realization: IUserDB
Uses: IDB

Figure 5.6: Excerpt of the software architecture model from the MediaStore dataset and a resulting Element.

The pipeline contains two preprocessor stages. One is used for source artifacts and the other for target artifacts, as usually they belong to different artifact types and therefore may require different preprocessing.

Out of the implemented preprocessors the simplest one does not do any preprocessing to the content of the artifact. It takes the text contained within the artifact and creates an element with the same content. Since there are preprocessing steps dependent on the artifact type, this preprocessor is usable for all text based artifacts.

For artifacts containing source code two preprocessors exist. The first one searches for methods within a Java class and creates a separate element for each of them. Often, the method signature and method body do not contain all the relevant information. Documentation, such as Javadoc comments, are usually directly above the method signature. To include this information the preprocessor also include the lines since the end of the last method in each element. The second code preprocessor splits the artifact into chunks. Chunking is a technique often used in conjunction with retrieval-augmented large language models to only fetch relevant parts of a document or to at least reduce the amount of unnecessary information retrieved. For this langchain's `RecursiveCharacterTextSplitter` is used. This text splitter tries to split on characters of a given list sequentially until all chunks are smaller than a specified size. As each programming language uses its own syntax the implemented preprocessors are not universally applicable. Only Java is supported, which is the language used in the datasets, that the approach is evaluated with.

Artifacts containing natural language text, are preprocessed by either splitting the text into sentences or into lines. Which one is used depends on the structure of the artifacts and the wanted granularity for the resulting elements.

A preprocessor for generating elements from UML models saved as XML files is implemented. This preprocessor parses the XML file and extracts information about the individual UML elements, such as type, name and which interfaces are realized by components. In addition to its name, the resulting `Element` objects can be configured to contain usages, operations and/or interface realizations. Figure 5.6 shows original interfaces and a component from the MediaStore dataset and the information that is extracted from them.

5.4 Embedding Creator

The embedding creator module calculates an embedding of the content of each element. Since comparable embeddings, i.e. embeddings in the same vector space, for source and target elements are needed, only one embedding creator exists in the pipeline. For this thesis only the `OpenAIEmbeddingCreator` was implemented. It uses langchain's OpenAIEmbedding [3] to calculate an embedding based on the content of an `Element`. While the embedding model can be freely chosen from a number of existing models, or even be one trained from scratch, only OpenAI's `text-embedding-3-large` [32] is used in this thesis to cut down the amount of possible configurations. It caches the calculated embeddings, so they aren't recalculated every time the program is run with the same inputs.

5.5 Element Store

The element store module stores and retrieves elements. During creation, the element store receives all elements from the preprocessor together with their embedding. It is assumed once the store is created, no new elements will be created and added. Since the evaluation is done on fixed datasets, this is a reasonable assumption for this thesis but might impact its reusability in future projects. The pipeline uses two element stores: a source element store and a target element store. They store source elements and target elements respectively.

After its creation, the element store processes queries. A query contains the embedding of an `Element`'s `content`. The amount of returned elements is dependent on the implementation. For the current implementation however, it is configurable to a specific amount and/or a threshold. Originally the element stores were intended to only store `Elements` with `compare` set to True. However, since the underlying database only stores the `identifier` for parents of `Elements` instead of the python object, the element store needs access to every element. To avoid having multiple places providing the same information the element store is also responsible for elements which are not part of the query answer. Therefore, besides querying the element store for similar elements, it also provides methods to search an element with a specific identifier and for elements whose parent has a specific identifier.

The framework uses ChromaDB [4] as its vector store implementation. Alternatively, faiss [6] was considered. Both are commonly used vector store implementations used in conjungtion with langchain. However, a Python 3.12 compatible version of faiss only became available during the development. Therefore, ChromaDB was used. Chroma uses an approximate nearest neighbour algorithm to find entries similar to a given query. As a

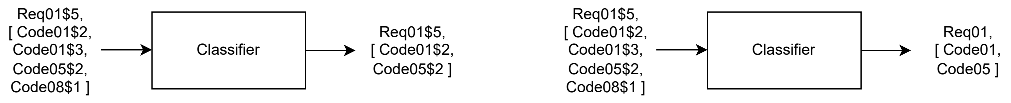

Req01$5, [Code01$2, Code01$3, Code05$2, Code08$1]
Classifier
Req01$5, [Code01$2, Code05$2]
Req01$5, [Code01$2, Code01$3, Code05$2, Code08$1]
Classifier
Req01, [Code01, Code05]

Figure 5.7: Examples of how the classifier can return its results. Left: the output is a subset of the input. Right: the output is a subset of the parent elements of the input.

result, it is non-deterministic. To remedy this problem and have reproducibility, whenever chroma is queried with a specific amount of entries `n` in mind, it is instead queried for all entries. They are then sorted based on their distance to the query embedding and the first `n` elements of the list are returned. This approach is contrary to the speed benefit of vector stores when compared to conventional databases. For this work it is deemed an acceptable trade-off. The used datasets at most contain a couple of hundred artifacts and even when preprocessed into elements, the element store only needs to handle a few thousand of entries.

5.6 Classifier

The classifier module decides whether a trace link will be classified. For this thesis, only classifiers using large language models are used, however, other implementations are possible. Per call, the classifier takes a single source element and a list of target elements. This results in a source element and a list of target elements, where the `Classifer` predicts a traceability link between the source element and all target elements.

Figure 5.7 illustrates how returned elements do not need to be the same as the ones used to call the classifier. While the classifier implementation in the left example compared the input elements, the implementation on the right compared the parents of the input elements and therefore only makes assumptions about the parents. Future implementations might return a sorted list instead of a binary classification. In this case, it is important to choose a result aggregator that expects such an input. If needed, a classifier can use context through a context provider to gain information about the siblings of elements, i.e. the elements with the same parent.

The classifiers were implemented iteratively. The first classifier is the Yes/No classifier. It is a classifier which expects either 'yes' or 'no' as an answer from the large language model. Its main purpose is to get to know how a large language model behaves and how the system works without accidentally spending a lot of money. The second classifier is more open in the prompts used with it. It is meant to for prompts asking for a reason why the language model answers in the way it does. The last classifier is capable of performing more than one prompt per element pair. It uses a list of prompts and termination conditions, that will be used sequentially. After each prompt the classifier checks whether the language model reply fulfills the current termination condition. If not it continues with the next prompt.

The termination condition can lead either to a predicted trace link or no predicted trace link.

In some cases, it is important to assess the quality of the retrieved elements without filtering them further. However, the framework uses a fixed pipeline layout and each step must be filled with a module. For this the `MockClassifierIt` does not query the large language model and instead considers the source element to have a trace link to each similar target element.

5.7 Result Aggregator

The result aggregator performs two functions. First it aggregates the results of the classifier to the correct granularity. For example, if the desired granularity of trace links is between classes and whole requirements, but the classifier returned potential trace links between methods and sentences, then, the result aggregator will look up parents until the desired granularity for source and target elements is found. The result aggregator can only traverse the element tree towards the original artifact. Even if the result aggregator could traverse it to find finer elements, the resulting trace link predictions would stay as course as the output of the classifier. The second function is to whether a trace link should be classified. The implemented result aggregator, `AnyResultAggregator`, will classify a trace link if any of the children, or the element itself, is considered to be related. It therefore puts the responsibility of classifying trace links completely on the `Classifier`.

5.8 Controller

The controller is responsible to instantiate the pipeline modules and control the flow of data between them. To use the controller it needs a pipeline configuration. This configuration is saved in form of a JSON file. The content of a possible configuration file is shown in Listing 5.1. Each pipeline module has a name. This name specifies which implementation to use. In addition, each module has a field for arguments. Which arguments are possible and whether they are optional or not depends on the implementation of the module in question.

```
{
    "source_artifact_provider": {
        "name": "text",
        "args": {"artifact_type": "requirement", "path": "./datasets/eTour_en/UC"}
    },
    "target_artifact_provider": {
        "name": "text",
        "args": {"artifact_type": "source code", "path": "./datasets/eTour_en/CC"}
    },
    "source_preprocessor": {
        "name": "sentence",
        "args": {"language": "en"}
    },
    "target_preprocessor": {
        "name": "code_chunking",
        "args": {"language": "java"}
    },
    "embedding_creator": {
        "name": "open_ai",
        "args": {"path": "./storage/eTour_en/embeddings/", "model": "text-embedding-3-large"}
    },
    "source_store": {
        "name": "chroma",
        "args": {"direction": "source", "path": "./storage/eTour_en/"}
    },
    "target_store": {
        "name": "chroma",
        "args": {"direction": "target", "path": "./storage/eTour_en/", "similarity_function": "cosine", "n_results": 20}
    },
    "classifier": {
        "name": "chain_of_thought",
        "args": {"model": "gpt-3.5-turbo-0125"}
    },
    "result_aggregator": {
        "name": "any_connection",
        "args": {
            "source_granularity": 0
        }
    }
}
```

Listing 5.1: Example pipeline configuration

6 Related Works

This chapter gives an overview of related work. In section 6.1 different approaches to traceability link recovery are presented. section 6.2 covers approaches to related problems, while in section 6.3 discusses the work using retrieval augmented language models.

6.1 Traceability Link Recovery

To bridge the semantic gap problem between natural language requirements and code, Hey et al. propose to use fine-grained similarities in the paper "Improving Traceability Link Recovery Using Fine-grained Requirements-to-Code Relations" [12]. Their approach Fine-Grained Traceability Link Recovery (FTLR) automatically recovers trace links in an unsupervised setting. It does not need existing trace links. Instead of whole documents, the smallest elements FTLR operates on are requirement sentences and source code methods. The process consists of three steps: representing artifact elements as word embeddings, calculating similarities between the elements, and aggregating the fine-grained relations. FTLR defines the relation between a (course-grained) requirement and a (fine-grained) code element as the most similar relation between a requirement element and a code element. To create trace link candidates a majority vote is used. Each method votes for the requirements it is related to after a threshold is applied. Final trace links are retrieved by applying another threshold. Hey et al. consider additional information contained in the artifact documents. Different variants of FTLR are created which use method comments, method call dependencies, and structural information of use cases in the form of use case templates. The datasets used for evaluation are the eTour, iTrust, SMOS, and eAnci datasets provided by the CoEST [2]. Hey et al. report precision, recall, the F_1-score, and mean average precision for the different FTLR variants. The variant incorporating all three additional information sources achieves the best average F_1-score of 0.327. Using use case templates had the greatest impact.

Moran et al. present their Hierarchical Probabilistic Model for Software Traceability (COMET) in their paper "Improving the Effectiveness of Traceability Link Recovery using Hierarchical Bayesian Networks" [30]. It combines multiple measures of textual similarities to model relationships between artifacts. The authors name three shortcomings of prior approaches: only using a single textual similarity metric, no ability to use developer feedback, and approaches only considering pairs of artifacts. Moran et al. define the existence of trace links as a probabilistic problem, and they model the traceability link recovery task as a bayesian inference problem. COMET uses three stages. The first stage combines a number of different textual similarity metrics. The second stage uses feedback of developers. And the third stage makes use of transitive traceability links. To decide whether there is a trace link between two artifacts the posterior probability has to be calculated. Since it cannot be calculated analytically, different approximations are used: Maximum a Posteriori estimation, A Markov Chain Monte Carlo technique using the No-U-Turn sampling, and Variational Inference, a machine learning-based technique. Moran et al. showed, that only using stage 1 already comes close to optimally configured baseline combination.

In their paper "Leveraging Intermediate Artifacts to Improve Automated Trace Link Retrieval" [35], Rodriguez et al. explore the use of intermediate artifacts to increase accuracy. For example, to find trace links between requirements and code, trace links between requirements and design, and trace links between design and code are incorporated. To find trace links, both, the Vector Space Model (VSM) and the Latent Semantic Indexing (LSI) approach are included. The authors compare three technique families: direct (e.g. requirement to code), transitive (e.g. requirement to design, and design to code), and hybrid approaches, which combine the results of the direct and transitive approaches. For the transitive approach, there are multiple paths from source to target artifact possible. Therefore, the individual path scores have to be aggregated using one of three methods: the maximum score, the sum of all scores, or a weighted sum using principal component analysis (PCA). Similarly, the hybrid approach needs to aggregate the results of the direct and transitive approaches. The best hybrid approach achieved better results than the best direct approach for all five datasets used, for all reported metrics (MAP, AUC, LAG). However, the best transitive approach did outperform the best hybrid approach on one dataset for all metrics and on two other datasets for MAP. No single technique performed best on all datasets. Rodriguez et al. assume an optimal technique needs to be tuned specifically for a project and maybe for different trace paths within a project.

Besides requirements and code, there are other artifacts to be looked at. In the paper "Detecting Inconsistencies in Software Architecture Documentation Using Traceability Link Recovery", Keim et al. focused on software architecture documentation in the form of natural language texts and formal architecture models [16]. While the main goal is to find inconsistencies in the documentation, they present their approach ArDoCo, which uses traceability link recovery to detect inconsistencies. The approach is based on SWATTR [18], which uses four major steps: model extraction, text extraction, element identification, and element connection. ArDoCo improves on SWATTR's handling of compound nouns by using phrases and adapting the heuristics if possible names of recommended instances belong to the same phrase. Keim et al. evaluated their approach on a benchmark dataset [8], containing the three open source projects MediaStore, TeaStore, and TEAMMATES to which they added JabRef and BigBlueButton. For traceability link recovery their approach achieves an average F_1-score of 0.81 with a precision of 0.83 and a recall of 0.82.

In "Recovering Trace Links Between Software Documentation and Code", Keim et al. present their approach TransArC for traceability link recovery between architecture documentation and code [17]. Their approach generates transitive links using architecture models as intermediate artifacts by combining ArDoCo [16] for finding trace links between architecture documentation and models, and a new approach ARchitecture-to-COde Trace Linking (ArCoTL) for finding links between architecture models and code. ArCoTL uses intermediate representations of the artifacts on a more abstract level. Using heuristics, each pair of architecture and code items is evaluated, resulting in a confidence value of each heuristic for each pair. Keim et al. use standalone, independent, heuristics, which compare package and component names, paths of compilation units and component names, method names and signature names, and architecture element names and names of compilation units and datatypes. They also used dependent heuristics, which depend on the results of other heuristics, such as a common word heuristic which increases the score of the name comparison heuristic if the difference of the compared names is a common prefix or suffix like "Test" or "Exception". Aggregators combine the results and filter out unlikely pairs. After, trace links are generated for the remaining pairs. To find trace links between architecture documentation and code the results of ArDoCo and ArCoTL are aggregated. Keim et al. evaluate their approaches on the same projects used to evaluate ArDoCo [16], for which the dataset [8] contains documentation, structural architecture models. In addition, gold standards for architecture documentation to code and architecture model to code were created. For TLR between architecture models and code, the authors report an average F_1-score of 0.98, which they explain as the result of software architecture models and code being very closely related with a small semantic gap. They note, that some naming-related issues, causing false positives and false negatives in some projects, still exist. For TLR between software documentation and code, an average F_1-score of 0.82 is reported, significantly outperforming the baselines.

Rodriguez et al. discuss prompt engineering for using large language models for traceability link recovery in their paper "Prompts Matter: Insights and Strategies for Prompt Engineering in Automated Software Traceability" [36]. They present two approaches: classification and ranking. They start the classification approach with a simple prompt and iteratively refine it. To learn why a prompt might lead to mispredictions they prompted the model to explain its decision. While the authors say whether the explanations are accurate to the actual reasoning is not part of their paper they mention that it was a useful tool to improve the prompts. Chain-of-thought reasoning was used to first prompt the language model for possible reasons why requirement pairs might be related and why they might be unrelated before asking it to classify based on these reasons.
In their second approach, Rodriguez et al. prompted the language model to rank several artifacts from most to least related. Since random order for target artifacts resulted in a barely above-random performance, they decided to present the artifacts in a sorted order based on similarity to the query artifact, on which the language model improved. The authors list several key takeaways. Small changes in the prompt can lead to big differences in the language model output. Even subtle changes such as pluralization can change outcomes. The performance of a prompt varies between datasets and models, however, techniques such as chain-of-thought tend to be more consistent.

Another approach toward the traceability link recovery problem includes leveraging a project's known trace links to improve results. Since often only a small amount of trace links is known for projects, Mills et al. incorporate active learning to reduce the needed training data with their approach ALCATRAL [27]. ALCATRAL first trains a classifier on a small initial training set. The classifier then labels the unlabeled pairs and assigns a confidence score. Uncertain classifications are inspected by an expert to provide the true label, which then is used to update the training set.

6.2 Related Problems

A problem related to traceability link recovery, especially if it involves code artifacts, is bug localization. Given a textual bug report, the goal is to find the corresponding location in the source code. In recent years machine learning approaches and large language models became popular to solve this problem. [23] Liang et al. use a fine-tuned language model in their paper "Modeling function-level interactions for file-level bug localization" in their approach FLIM. Large files can cause issues during the training of machine learning approaches. Additionally, functions within a file might not be related, which might lead to incorrect results if adjacent information is used. To solve this, Liang et al. split the source files at the function level. As the base model for their approach, CodeBERT was chosen. It is first fine-tuned on a code search task. In the second stage, the model captures semantic features, which then are used by a learning-to-rank model to predict the relevance of a given code section. Liang et al. compare their approach to three IR-based and three machine-learning approaches. For five out of six projects, FLIM achieves the best MAP score, increasing between 6.0% and 10.5% compared to the best baseline approach. As FLIM project-dependent training data often is not available, the authors conduct an additional experiment, in which FLIM is fine-tuned on the projects not used for testing. It achieves an average MAP of 0.402.

In their paper "Improving Bug Localization With Effective Contrastive Learning Representation" Luo et al. present their approach CoLoc [26]. For their language model, they carry out two pre-training phases: using masked language model pre-training and contrastive learning. Then, it is fine-tuned for the bug localization task. To evaluate CoLoc, a dataset containing bug reports and files of AspectJ, Eclipse UI, JDT, SWT, and Tomcat are used. CoLoc is compared to several baseline approaches, achieving a higher MAP score on all projects. In a second experiment, CoLoc is compared to other pre-trained language models: BERT, RoBERTa, and CodeBERT. The authors note the pre-trained approaches all either outperform or have close results to the baseline approaches of the first experiment.

6.3 Retrieval Augmented Generation

Inside large language models, knowledge is saved implicitly inside their parameters. Storing more information therefore requires larger models and updating their knowledge usually requires retraining. In "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks", Lewis et al. tackle this problem by augmenting a language model with a vector index of Wikipedia as a non-parametric memory [22]. This memory is accessed using a neural retriever. The retriever and the generator model are trained jointly. During training the encoder model parameters are not changed not to have to recalculate the vector index. Lewis et al. evaluate their approach using several NLP tasks: Open-domain Question Answering, Abstractive Question Answering, Jeopardy Question Generation, and Fact Verification. As a classification task, the fact verification task, FEVER [42] is of special interest to us. The authors report their approach comes within 4.3% of the baseline state-of-the-art models.

One hindrance to adopting retrieval-augmented language models is the needed changes to the model and the pre-training required for incorporating a knowledge retriever. This is not always feasible and might be impossible in some cases when the pre-trained language model has to be used with no changes. In "In-Context Retrieval-Augmented Language Models" Ram et al. present an approach for using off-the-shelf language models which works by adding a retrieved document to the beginning of the language model input [34]. During generation, a retriever is called every couple of generated tokens to provide a retrieved document. Ram et al. report, that calling the retriever often leads to better results, but also calling it more rarely improves performance compared to a language model without a retriever. They claim it is "a tradeoff between runtime and performance".

Other work focuses on a single use of the retriever before the language model generates an answer [14] [21] [15] [39]. Usually, one or more relevant documents are concatenated to the original query.

7 Evaluation

The answer of OpenAI's large language models are not deterministic. A parameter to set a seed is now available as a beta feature, but was not when the experiments were performed. To minimize this problem the temperature of the used language model is set to 0 which should lead to more consistent outputs according to OpenAI's API documentation [41]. Each experiment also could have been performed multiple times and the results of each run averaged. Due to the rapidly increasing cost this was dismissed, and each experiment is only performed once.

The language model used in all experiments is OpenAI's GPT-3.5-turbo-0125. GPT-4.0-turbo was considered, but performed similar to GPT-3.5-turbo on a subset of the used datasets. Due to this and the increased cost of GPT-4.0-turbo, it was decided to perform the experiments on GPT-3.5-turbo only.

7.1 GQM Plan

This section discusses the planned evaluation of the proposed approach. It follows the Goal Question Metric method [1].

The GQM-Plan is the following:

- Goal: Automatically recover traceability links between artifacts of a project.
- Question 1: How well does the retrieval-augmented large language model approach perform?
- Metric 1: Precision, Recall, F1-scores on benchmark datasets for the retrieval-augmented large language model approach
- Question 2: Does the large language model improve upon the embedding-based retrieval method?
- Metric 2: Precision, Recall, F1-scores on benchmark datasets for the retrieval system with and without the language model
- Question 3: How does the approach compare to state-of-the-art approaches?
- Metric 3: Precision, Recall, F1-scores on benchmark datasets for the retrieval-augmented large language model approach and state-of-the-art approaches

To evaluate the approach, Precision, Recall, and the F1 score are used. These metrics are commonly used in classification tasks. They are also used in prior traceability link recovery research, and therefore allows a comparison of the retrieval-augmented large language model approach and state-of-the-art approaches. Precision measures the ratio of correctly predicted links and all predicted links. Recall measures the ratio of correctly predicted links and all relevant documents. It shows an approach's ability to provide correct trace links. The F1 score is the harmonic mean of precision and recall. The Metrics are calculated in the following way, where TP is true positives, FP is false positives, and FN is false negatives.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{7.1}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{7.2}$$

$$F_1 \text{ Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{7.3}$$

Another commonly used metric in traceability link recovery research is the Mean Average Precision. It requires for all, according to the gold standard, positive trace links to be in the results, sorted by how certain the approach is. Since the retrieval-augmented large language model approach only compares a small part of all possible artifact combinations, and it does not generate a sorted result set.

7.2 Baselines

To assess the qualityThe first one is the use of only retrieved from the Element Store. For each source element a number of most similar target elements is returned. For this each element that is returned by the Element Store is classified having a trace link with the source element used to query it. The language model is not prompted. This allows to see what kind of effect the classifier step has or whether a using just the results of the retrieval step might be beneficial.

Besides this Retrieval-Baseline, the approach is compared to state-of-the-art approaches. For requirement to source code traceability link recovery those approaches are COMET [30] and FTLR [10]. For both projects the results presented are as Hey reported them [10] or recreated using the replication package he provided [11]. COMET is an approach which model the probability of two artifacts having a trace link. It uses a combination of several textual similarity metrics. FTLR considers fine-grained trace links. Instead of comparing artifact to artifact it extracts smaller features from them. It uses word embeddings to compare requirement and source code artifacts. FTLR has several variants, it can use method comments, filtering of templates and call dependencies. For the software architecture documentation to software architecture model traceability link recovery task, ArDoCo [16] is used as a baseline. ArDoCo is an approach to detect inconsistencies in software architecture documentation. It makes use or trace link recovery to do so. ArDoCo uses four major steps: model extraction, text extraction, element identification, and element connection. The results presented in their paper were achieved using a combination of normalized Levenshtein distance and Jaro-Winkler as word similarity metric. For the software architecture documentation to source code traceability link recovery task, TransArC [17] is used as a baseline. TransArC assumes a transitive feature of trace links. First ArDoCo is used to find traceability links between architecture documentation and architecture models. Then, a new approach, ArCoTL, is used to recover traceability links between architecture models and source code. Finally, found trace links from both approaches are linked based on whether they share a architecture model component. The presented result of ArDoCo and TransArC are taken form their respective papers.

7.3 General Prompts

Multiple prompts are used to evaluate the retrieval-augmented large language model traceability link recovery approach. Two of them will be used for all artifact type pairs. The goal of this thesis is not to do extensive prompt engineering, as different large language models react differently to the same prompt and therefore a very good prompt might perform worse with a future language model. Nevertheless, I believe by basing the used prompts on prompts found by prior work, especially on the prompts by Rodriguez et al. [36], the results will be valid. Rodriguez et al. did not pursue a retrieval-augmented approach and only presented their results for requirement to requirement tracing. However, since this thesis discusses an approach that tries to be universal when it comes to artifact types, a generalized prompt is wanted.

Prompt 7.1: Yes/No

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 7.2: Reasoning

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}”’

The No/Yes classifier uses Prompt 7.1, a prompt template based on the initial prompt Rodriguez et al. used [36]. This prompt is simple enough to get an idea how well the approach might work and to figure out how the large language model behaves. While the language model mostly follows the command, it often answers with a variation of "yes" or "no", such as adding punctuation or using uppercase (e.g. "yes." or "Yes" instead of "yes"). Therefore, when evaluating the reply of the language model, it is only checked whether a lowercase version of the reply contains "yes". In each prompt template *{source_type}* and *{target_type}* are placeholders that will be filled in with the respective artifact types, such as software *architecture documentation* or *source code*. The same is true for *{source_content}* and *{target_content}* which are placeholders for the actual content of an element. The second prompt template used, Prompt 7.2, improves on the first one by prompting the language model to give a reason for its answer. This invokes a from of zero-shot chain-of-thought prompting. Prompt 7.1 and Prompt 7.2 are used to evaluate all examined artifact types. Variations of Prompt 7.2 are used in more specific experiments.

7.4 Requirement to Source Code Traceability Link Recovery

To evaluate the retrieval-augmented large language model approach on the requirement to source code traceability task, a subset of all possible configurations was chosen. The first experiment included no specific preprocessing, the content of the artifacts is taken as is. The datasets, SMOS, eTour, and iTrust, use one file for each requirement and each source code Java file. The goldstandard maps between files.

Table 7.1 shows the results of the experiments. They are grouped by the preprocessing techniques used on the artifacts. In the first group of experiments the artifacts are not preprocessed, and whole artifacts are embedded. For the second and third group requirements are split into sentences. The source code in the second experiment group is split into chunks. By default, langchain sets the chunk size to 1000This means, the source code would be split into chunks containing 1000 characters. For the experiment a chunk size of 200 is used, resulting in pieces of a consistent size, that usually are smaller than methods. For the source code artifacts in the third group, the methods within a class are extracted. Comments and class attributes before a method are concatenated to it.

For the classifier, "No LLM" refers to the values calculated based on the similar elements without prompting the language model. "Yes/No" uses Prompt 7.1, simply asking whether two artifact parts are related expecting a just ’yes’ or ’no’ as an answer. "Reasoning" refers to the usage of Prompt 7.2. In addition to answering with "yes" or "no", the language model is supposed to give a reason, inducing a form of Zero-shot-Chain-of-Thought reasoning. Single-shot and Few-shot approaches were not considered as they require a known trace links of a project. A universal constructed example or known trace links of another project were not used since projects differ in style and form of artifacts.

In all experiments, the 20 most similar elements are compared. This value was chosen, so it is theoretically possible all trace links are found without comparing an unreasonably

		SMOS			eTour			iTrust		
Approach		P	R	F1	P	R	F1	P	R	F1
$\text{COMET}_{\text{best}}$		0.195	0.572	0.291	**0.410**	0.468	0.437			0.282
$\text{FTLR}_{\text{best}}$		0.393	0.386	**0.390**	0.376	0.643	0.475	0.165	0.339	0.222
Preprocessor	Classifier									
None/ None	No LLM	0.334	0.428	0.375	0.216	0.815	0.342	0.058	0.535	0.105
	Yes/No	**0.616**	0.129	0.214	0.309	0.669	0.423	0.075	0.528	0.132
	Reasoning	0.564	0.250	0.346	**0.410**	0.575	**0.478**	0.155	0.371	0.219
Sentence/ Chunk (200)	No LLM	0.195	0.807	0.314	0.086	**0.834**	0.157	0.063	0.577	0.113
	Yes/No	0.275	0.404	0.328	0.158	0.653	0.255	0.089	0.441	0.148
	Reasoning	0.230	0.583	0.330	0.135	0.617	0.221	0.094	0.444	0.155
	Reasoning, Artifacts	0.596	0.175	0.271	0.333	0.403	0.365	**0.259**	0.395	**0.313**
Sentence/ Method	No LLM	0.212	**0.838**	0.338	0.071	0.594	0.126	0.060	**0.619**	0.110
	Yes/No	0.294	0.572	0.388	0.097	0.484	0.162	0.088	0.591	0.154
	Reasoning	0.275	0.593	0.375	0.109	0.513	0.180	0.102	0.510	0.170
	Reasoning, Artifacts	0.549	0.272	0.364	0.261	0.354	0.301	0.168	0.451	0.245

Table 7.1: Results for Requirement-to-Code Traceability Link Recovery Using the 20 Most Similar Elements for Each Source Element

large amount of element pairs. Ideally, this value is chosen individually for each project, however it is difficult to find a value that works well. Depending on the conventions of a project a single source artifact might be related to many or only a few target artifacts. At the same time this can vary within the same project. Some projects might not be complete, and therefore some source artifacts miss target artifacts altogether. While working on this thesis, comparing a varying number of target elements based on the similarity value to the source element was theorized, but due to time constraints a fixed value is used.

The rows in Table 7.1 marked with "Artifacts" use the preprocessed elements for similarity comparisons, but use the original artifact when prompting the large language model. The experiment in the group without preprocessing also use the original artifacts but are not marked since finding similar elements is done on the artifacts as well. For both state-of-the-art approaches COMET and FTLR a number of different configurations exist. The values used in this table are of the best configuration per dataset as reported by Hey [10, 11].

In the following paragraph the results of the retrieval baseline are analyzed. Since those results are taken after the retrieval of similar elements, the recall value serves as an upper bound for the other experiments in the same preprocessing group. Due to the limit of

Preprocessor	Prompt	SMOS			eTour			iTrust		
		P	R	F1	P	R	F1	P	R	F1
Sentence/ Method	Prompt 7.2	.549	.272	.364	.261	.354	.301	.168	.451	.245
	Prompt 7.3	.345	.606	.44	.125	.539	.203	.099	.57	.169

Table 7.2: Comparison Between Original and Slightly Altered Prompt.

retrieving 20 elements a perfect precision can only be reached if every element had exactly 20 trace links. Therefore, putting an upper bound on the precision of the retrieval baseline. For the SMOS dataset no preprocessing achieves the lowest recall value with 0.428, while both Sentence/Chunk and Sentence/Method preprocessing achieve a much higher recall with 0.807 and 0.838 respectively. The results of the iTrust dataset also show an improvement through preprocessing, where splitting by methods achieves the highest recall value of 0.619. This would suggest splitting source code artifacts into elements containing whole methods is a beneficial preprocessing technique when using embedding based similarity comparisons. However, eTour resulted in a high recall value of 0.815 without any preprocessing, that increased to 0.834 for Sentence/Chunk preprocessing, while the Sentence/Method preprocessing only achieved a recall value of 0.594, 12 percentage points lower than no preprocessing. The size of the element seems to have no or little effect. Elements containing eTour's methods have an average size of roughly 860 characters, while SMOS's elements contain 885 characters on average. When compared to COMET and FTLR, the retrieval-augmented large language model approach achieves comparable and for eTour and iTrust even slightly better results for some configurations. For SMOS the optimal result is achieved with Sentence/Method preprocessing and the Yes/No classifier using Prompt 7.1 with a F_1 score of 0.388, only slightly less than FTLR's 0.390. The configurations using Prompt 7.2 achieve a F_1 score of 0.375 and 0.364. All F_1 scores in the Sentence/Method preprocessing group are higher than their Sentence/Chunk counterpart. One notable value is the F_1 score of 0.375 achieved with no preprocessing and no language model. For the eTour dataset, the approach achieves a F_1 score of 0.478 using Prompt 7.2 and no preprocessing. This value is slightly higher than FTLR's 0.475 F_1 score. Due to randomness of the replies of the large language model, this difference might be negligible. For iTrust, the best result of 0.313 was achieved within the Sentence/Chunk preprocessing group using Prompt 7.2 with the original aritfacts. This is higher than FTLR's F_1 value of 0.222 and COMET's 0.282.

For each tested dataset, the highest F_1 score was achieved in a different preprocessing group. Therefore, it is reasonable to assume the effectiveness of a specific preprocessing technique heavily depends on characteristics of the examined project. Using the original artifacts when prompting the language model led to a strong increase in precision, independent of the dataset and preprocessing technique, while at the same time the recall was decreased compared to using the preprocessed elements.

Prompt 7.3: Conceivable Trace Link

Below are two artifacts from the same software system. Is there a **conceivable** 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}"'

To get an idea what effect a small change to the prompt has, Prompt 7.3 was created. Instead of asking whether there is "a traceability link" between two artifacts the language model was asked whether there is "a **conceivable** traceability link". As Table 7.2 shows, the recall for all three datasets increased, while the precision decreased. This is in line with the expectation, as "conceivable" implies more broad ways to find a connection between two artifacts. However, only the F1 value for SMOS increased, while it decreased for eTour and iTrust. This shows different prompts are more beneficial for some but not all projects.

7.5 Software Architecture Documentation to Software Architecture Model Traceability Link Recovery

The second pair of artifact types on which the retrieval-augmented large language model traceability link recovery approach is evaluated, are software architecture documentation and software architecture models. Other than the requirement and code in the previous chapter, the datasets used share the same file for all artifacts of a type. For the experiments, software architecture documentation is split into single sentences, as the used gold standard contains a mapping of sentence to software architecture model component. No further preprocessing is done on the resulting elements. As described in Section 5.3, different features can be extracted from the software architecture models. For the evaluation, there are three groups. The first contains only the component name. The second contains the name and the names of interfaces the component realizes. The third contains name, interfaces and the names of components which the model component uses. The same classifiers as in Section 7.4 are used. However, no original artifacts are used, as those would be the original file containing all documentation sentences or model components. Instead, three different variations of the large language model and prompt are tested.

Table 7.3 contains the results of the software architecture documentation to software architecture model traceability link recovery task. Only the five most similar elements are taken into account. This is due to the datasets being way smaller and having less traceability links between documentation sentences and model components. JabRef is the most extreme example. It only contains six model components in total. Therefore, it is almost expected to see a perfect recall value. Another anomaly of the JabRef datasets is that it only contains UML components but no interfaces. This results in all three feature

		BigBlueButton			JabRef			MediaStore			TeamMates			TeaStore		
Approach		P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
ArDoCo		**0.88**	0.83	**0.85**	0.90	**1.00**	**0.95**	**1.00**	0.62	**0.77**	0.56	0.90	**0.69**	**1.00**	0.74	**0.85**
Features	Classifier															
Name	No LLM	0.12	0.94	0.21	0.28	**1.00**	0.43	0.15	**0.97**	0.26	0.05	**0.94**	0.09	0.11	0.89	0.20
	Yes/No	0.25	0.77	0.38	0.38	**1.00**	0.55	0.22	**0.97**	0.36	0.14	0.92	0.25	0.22	0.89	0.36
	Reasoning	0.33	0.75	0.46	0.69	**1.00**	0.82	0.40	0.86	0.54	0.36	0.88	0.51	0.42	0.78	0.55
Name, Interfaces	No LLM	0.12	0.94	0.21	0.28	**1.00**	0.43	0.15	0.93	0.25	0.05	0.92	0.09	0.12	0.93	0.21
	Yes/No	0.26	0.69	0.37	0.38	**1.00**	0.55	0.22	0.90	0.35	0.18	0.84	0.29	0.34	0.85	0.49
	Reasoning	0.27	0.67	0.39	0.69	**1.00**	0.82	0.34	0.90	0.50	0.19	0.75	0.31	0.40	0.78	0.53
Name, Interfaces, Usages	No LLM	0.12	**1.00**	0.22	0.28	**1.00**	0.43	0.15	**0.97**	0.26	0.05	0.90	0.09	0.12	**0.96**	0.22
	Yes/No	0.22	0.69	0.33	0.38	**1.00**	0.55	0.22	**0.97**	0.35	0.12	0.84	0.21	0.28	0.85	0.42
	Reasoning	0.22	0.75	0.34	0.69	**1.00**	0.82	0.24	0.90	0.38	0.13	0.82	0.22	0.32	0.82	0.46
Name	Yes if certain	0.54	0.62	0.58	**0.93**	0.78	0.85	0.55	0.79	0.65	0.50	0.77	0.61	0.64	0.67	0.66
	Two Steps	0.28	0.19	0.23	0.80	0.22	0.35	0.46	0.66	0.54	**0.60**	0.65	0.62	0.67	0.30	0.41
	Neighbours	0.18	0.92	0.29	0.44	**1.00**	0.61	0.19	**0.97**	0.32	0.10	0.92	0.18	0.20	0.89	0.33

Table 7.3: [SAD-SAM] Results for Software Architecture Documentation to Software Architecture Model Traceability Link Recovery using the 5 Most Similar Elements for Each Source Element.

groups being identical since only component names can be extracted. In general, the retrieval-augmented large language model approach reaches a good recall with a poor precision.

For all datasets, both the Yes/No and the Reasoning Classifier increase the precision and F1 score compared to just using the most similar elements. For some datasets and feature groups, they perform very similarly, such as with BigBlueButton in the all features group with an F1 score of 0.33 and 0.34.

When comparing the different feature groups, the best performing group is the group only containing component names with an average F1 score of 0.51 for the Reasoning Classifier, excluding JabRef. The other two groups have an average F1 score of 0.43 and 0.35. The more information the language model gets the worse the results become. This seems to be in contrast to the results in Section 7.4 where the original artifacts, containing more information than the elements, achieved better results than elements containing only a part of the information. However, there are differences between source code artifacts and model components. By comparison, source code files are large while the only unique part of a model component is its name.

Reply 7.1

The first artifact mentions a specific requirement related to downloaded audio files being uniquely connected to a user ID for association purposes. The second artifact describes a component named "Reencoding" that interfaces with "IDownload" and uses "IDownload." While the second artifact does not explicitly mention the user ID association requirement, it does relate to the downloading functionality, which could potentially be linked to the requirement mentioned in the first artifact. Therefore, there is a possibility of a traceability link between these two artifacts.
<trace>yes</trace>

By adding the name of a single interface half of the total information is not just about the element under consideration. The attention of the language model might not be on just the component name but on the other features. Reply 7.1 is a reply involving an element with an interface realization and a usage of the same. According to the gold standard, there should not be a traceability link between the considered artifacts. However, the reason explains there might be a traceability link due to both artifacts being connected to downloading.

Reply 7.2

The first artifact describes the security measure taken by the UserManagement component when a user logs into the system. The second artifact describes the components and interfaces used in the software architecture, including the MediaAccess component.
Based on the information provided, there is a traceability link between the artifacts. The UserManagement component mentioned in the first artifact is likely related to the MediaAccess component described in the second artifact, as the UserManagement component is responsible for handling user login security, which could be related to media access control.
<trace>yes</trace>"

Reply 7.2 contains a reply of the language model for a query using elements from the MediaStore dataset. The sentence from the software architecture documentation is "When a user logs into the system, Media Store does not store the password in plain text, but rather the UserManagement component implements further functions to hash and salt the passwords.". Instead of figuring out whether the sentence describes the component "MediaAccess", the explicitly named component "UserManagement" is linked to "MediaAccess".

Reply 7.3

The first artifact mentions a specific functionality related to downloaded audio files being uniquely connected to a user ID for association purposes. The second artifact describes a software component named MediaAccess that interfaces with IMediaAccess, IDownload, IFileStorage, and IDB. While there is no direct mention of downloaded audio files in the second artifact, the MediaAccess component could potentially be responsible for handling the functionality described in the first artifact.
<trace>yes</trace>

Reply 7.3 contains a reply in which the language model first describes the artifacts, gives a reason why there might not be a trace link and then says there might be one without elaborating.

To get an idea what kind of effect other techniques have on the results, three modified classifiers are used. The first is a small modification of Prompt 7.2.

Prompt 7.4: Yes If Certain

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>. Only answer yes if you are absolutely certain.
(1) {source_type}: "'{source_content}"'
(2) {target_type}: "'{target_content}"'

Prompt 7.4 is the same as the prior prompt with the small addition of a command to only answer yes if the language model is "absolutely certain". This increased the precision and F1 score on all datasets. The F1 score increased by 0.10 to 0.12, except on JabRef which only changed from 0.82 to 0.85 due to lower recall of 0.78. At the same time the recall decreased on all projects.

Prompt 7.5: Is Component?

You are given a part of a {source_type}. Does it refer specifically to a component? Give your reasoning and then answer with '<component>yes</component>' or '<component>no</component>'.
{source_type}:
"'{source_content}"'

The second alternative technique queries the large language model two times. Since the gold standard only references components from the architecture model and no interfaces, another prompt before the usual reasoning propmt, Prompt 7.2, was added. Prompt 7.5 asks whether the software architecture documentation sentence refers to a component. Unlike the other used prompts, this prompt is somewhat specific to the software architecture documentation to software architecture model traceability link recovery task. It also explicitly states the two available answers: *<component>yes</component>* and *<component>no</component>*. During earlier tests the language model's replies contained text such as *<yes></yes>*. If the language model determines it does refer to a component the language model is queried as usual. If it does not determine the sentence refers to a component it is immediately predicted there is no traceability link between the elements. This technique functions as two language model filters instead of one which can be seen by the low recall values compared to the other classifiers. Especially for BigBlueButton, JabRef and TeaStore that have a recall of 0.19, 0.22 and 0.30 respectively. However, TeamMates reaches the highest precision of 0.60 that is even slightly higher than ArDoCo's 0.56. Due to the recall of 0.65 the two-step technique still only reaches an F1 score of 0.62, with ArDoCo reaching 0.69.

Reply 7.4

The statement provided in the software architecture documentation mentions the storage of salted hashes of passwords in the Database component. This description is more related to the data storage aspect of the system rather than a specific software component. It does not provide details about the functionality or behavior of a software component itself.
<component>no</component>

Reply 7.5

The provided artifact describes a specific behavior or action related to the system, which is the storage of a file in the DataStorage component without any modification. This does not explicitly refer to a component itself but rather to a specific action that occurs within a component.
<component>no</component>

Reply 7.4 shows an example of the language model missing a component in a documentation sentence from MediaStore. This example is especially confusing since the *Database component* was identified and listed in the reason. Reply 7.5 contains another example of a missed component. One problem which occurred consistently was the language model identifying behaviors of the software and then ignoring the related components. This problem might be alleviated by carefully creating a prompt which tells the language model to connect behaviors to the involved components if they are mentioned.

Prompt 7.6: Include Neighbours

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"'
(1) is surrounded by this:
source_context_pre
source_content
source_context_post

The third alternative technique was to include the surrounding sentences for each source element. The prompt itself, Prompt 7.6 is another variation of Prompt 7.2. If possible in addition to the source element the sentences that surrounded it in the software architecture documentation. Up to two sentences before and two sentences after the element are added. The idea is to increase the available information just like using the original artifacts was

		BigBlueButton			JabRef			MediaStore			TeamMates			TeaStore		
Approach		P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
ArDoCo		**0.88**	**0.83**	**0.85**	**0.90**	**1.00**	**0.95**	**1.00**	0.62	**0.77**	0.56	**0.90**	**0.69**	**1.00**	0.74	**0.85**
Features	Classifier															
Name	No LLM	0.48	0.56	0.52	0.53	0.89	0.67	0.31	**0.76**	0.44	0.70	0.55	0.62	0.38	0.78	0.51
	Yes/No	0.58	0.56	0.57	0.71	0.83	0.77	0.37	**0.76**	0.50	0.72	0.55	0.62	0.48	0.78	0.59
	Reasoning	0.64	0.48	0.55	0.79	0.83	0.81	0.46	0.69	0.55	0.79	0.53	0.64	0.58	0.67	0.62
Name, Interfaces	No LLM	0.48	0.56	0.52	0.53	0.89	0.67	0.31	**0.76**	0.44	0.65	0.51	0.57	0.40	**0.82**	0.54
	Yes/No	0.69	0.48	0.57	0.71	0.83	0.77	0.42	0.72	0.53	0.74	0.49	0.59	0.59	0.70	0.64
	Reasoning	0.69	0.42	0.52	0.79	0.83	0.81	0.53	0.66	0.59	**0.81**	0.49	0.61	0.45	0.70	0.55
Name, Interfaces, Usages	No LLM	0.45	0.52	0.48	0.53	0.89	0.67	0.31	**0.76**	0.44	0.60	0.47	0.53	0.40	**0.82**	0.54
	Yes/No	0.54	0.44	0.48	0.71	0.83	0.77	0.54	0.72	0.62	0.64	0.45	0.53	0.61	0.70	0.66
	Reasoning	0.46	0.44	0.45	0.79	0.83	0.81	0.44	0.69	0.53	0.69	0.47	0.56	0.50	0.78	0.61
Name	Yes if certain	0.72	0.44	0.55	0.88	0.78	0.82	0.55	0.62	0.58	0.79	0.53	0.64	0.68	0.63	0.65

Table 7.4: [SAM-SAD-TLR] Results for Software Architecture Model to Software Architecture Documentation Traceability Link Recovery using the 5 Most Similar Elements for Each Source Element.

beneficial for the requirements to source code traceability link recovery task. In addition, some sentences within an architecture documentation give an explanation or additional information about the component or behavior described in the prior sentence. By putting multiple While the recall is the highest for all datasets when using the large language model, the precision is rather low with values of 0.10 to 0.20, except for JabRef where the technique reaches a precision of 0.44, which is still lower than the classifier using Prompt 7.2 with a precision of 0.69. The large language model might be confusing the unrelated parts with the actual element and make false connections. However, when checking the reasons the large language model generated this does not seem to be a big problem. Instead, it seems to be make more broad almost hallucinated connections between the elements. For example, it did connect a sentence about administrator keys and a special login procedure to a component *Common*. While such behavior is also noticeable when using Prompt 7.2, it is not clear why the language model behaved like this in a more pronounced way for Prompt 7.6.

No matter which Features are extracted or which classifier was used, the retrieval-augmented large language model traceability link recovery approach did not reach the performance of ArDoCo when using documentation sentences as source elements and model components as target elements. Using Prompt 7.4 leads to the best or close to best F1 scores for this approach, but still falls short of ArDoCo.

In another experiment the trace direction is switched. Model components are used as source elements and documentation sentences as target elements. As large language models can generate replies in very different ways with only small changes to the prompt

		BigBlueButton			JabRef			MediaStore			TeamMates			TeaStore		
Approach		P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
ArDoCo		**0.88**	**0.83**	**0.85**	**0.90**	**1.00**	**0.95**	**1.00**	0.62	**0.77**	0.56	**0.90**	0.69	**1.00**	0.74	**0.85**
Features	Classifier															
Name	No LLM	0.31	0.71	0.43	0.30	**1.00**	0.46	0.17	**0.86**	0.30	0.45	0.71	0.55	0.23	0.93	0.37
	Yes/No	0.46	0.67	0.54	0.47	0.89	0.62	0.23	**0.86**	0.36	0.48	0.71	0.57	0.39	0.93	0.54
	Reasoning	0.56	0.58	0.57	0.61	0.94	0.74	0.33	0.79	0.47	0.57	0.69	0.63	0.50	0.78	0.61
Name, Interfaces	No LLM	0.31	0.71	0.43	0.30	**1.00**	0.46	0.18	**0.86**	0.30	0.49	0.77	0.60	0.23	0.93	0.37
	Yes/No	0.55	0.50	0.53	0.47	0.89	0.62	0.31	0.83	0.45	0.58	0.73	0.64	0.58	0.78	0.67
	Reasoning	0.58	0.48	0.53	0.61	0.94	0.74	0.44	0.72	0.55	**0.69**	0.75	**0.72**	0.41	0.78	0.54
Name, Interfaces, Usages	No LLM	0.28	0.65	0.40	0.30	**1.00**	0.46	0.18	**0.86**	0.30	0.39	0.61	0.47	0.23	**0.93**	0.37
	Yes/No	0.39	0.52	0.45	0.47	0.89	0.62	0.39	0.83	0.53	0.46	0.55	0.50	0.57	0.78	0.66
	Reasoning	0.32	0.50	0.39	0.61	0.94	0.74	0.31	0.76	0.44	0.46	0.61	0.53	0.43	0.85	0.57
Name	Yes if certain	0.66	0.52	0.58	0.75	0.83	0.79	0.47	0.69	0.56	0.60	0.67	0.63	0.62	0.67	0.64

Table 7.5: [SAM-SAD-TLR] Results for Software Architecture Model to Software Architecture Documentation Traceability Link Recovery using the 10 Most Similar Elements for Each Source Element.

[36] they most likely are sensitive to the order in which the elements are added to the prompt. Table 7.4 contains the results of the first experiment. The number of the most similar elements under consideration is still five. The first thing to notice is the rather low recall values, even for experiments not using the large language model. This indicates the number of most similar elements is too low. Since the datasets typically contain way less software architecture model components than software architecture documentation sentences this is plausible. Table 7.5 contains the results when the ten most similar elements are under consideration, that will be discussed shortly. For the experiments using the five most similar elements, it can be seen that additional features generally decrease the F1 value. However, this effect is less pronounced than it is when tracing the other way around. When looking only at the Name feature group and the Reasoning classifier, as those performed best in the previous experiments when ignoring the alternative techniques, the model to documentation tracing reaches higher F1 values for most datasets with 0.55, 0.81, 0.55, 0.64 and 0.62 compared to 0.46, 0.82, 0.54, 0.51 and 0.55. The performance on JabRef was slightly worse with 0.81 compared to 0.82. The best performing technique for documentation to model tracing was using Prompt 7.4. Model to documentation tracing achieves similar results. On BigBlueButton, JabRef and TeaStore it performs slightly worse with F1 scores of 0.55, 0.82 and 0.65 compared to 0.58, 0.85 and 0.66. On MediaStore it achieves an F1 score of only 0.58 compared to the previous 0.65, while on TeamMates the F1 score increased from 0.61 to 0.64.

Next the results for using the ten most similar elements, as shown in Table 7.5 are discussed. As expected the recall values increased again, while the precision decreased. The Reasoning

Approach		BigBlueButton P	BigBlueButton R	BigBlueButton F1	MediaStore P	MediaStore R	MediaStore F1	TeaStore P	TeaStore R	TeaStore F1
TransArC		**0.94**	**0.96**	**0.95**	**0.98**	**1.00**	**0.99**	**0.98**	**0.98**	**0.98**
Preprocessor	Classifier									
Sentence/ None	No LLM	0.143	0.376	0.207	0.03	0.9	0.059	0.242	0.59	0.344
	Yes/No	0.152	0.305	0.203	0.033	0.84	0.064	0.255	0.537	0.345
	Reasoning	0.194	0.213	0.203	0.078	0.52	0.135	0.403	0.276	0.327
Sentence/ Chunk (200)	No LLM	0.145	0.253	0.184	0.056	0.72	0.104	0.285	0.322	0.303
	Yes/No	0.156	0.177	0.166	0.068	0.7	0.123	0.316	0.277	0.295
	Reasoning	0.234	0.161	0.191	0.09	0.64	0.157	0.318	0.218	0.259
Sentence/ Chunk (1000)	No LLM	0.169	0.303	0.217	0.051	0.82	0.096	0.28	0.332	0.304
	Yes/No	0.184	0.249	0.212	0.056	0.82	0.105	0.3	0.286	0.293
	Reasoning	0.195	0.176	0.185	0.077	0.64	0.137	0.338	0.24	0.281

Table 7.6: [SAD-Code-TLR] Results for Software Architecture Documentation to Code Traceability Link Recovery using the 40 Most Similar Elements for Each Source Element. Note for the reader: Please check the original TransArC publication to find the correct values of the TransArC approach.

classifier in the Name and Interface Realization group achieves the best result for the TeamMates dataset. It even achieves a slightly higher F1 score of 0.72 than ArDoCo's 0.69. For MediaStore the Reasoning classifer in the Name and Interface Realization group performed better than the other Feature groups. For BigBlueButton and TeaStore the results follow the previous results where Interface Realizations tended to decrease the F1 score. The classifier using Prompt 7.4 performs best except for TeamMates and TeaStore.

There is no clear best approach when it comes to the trace direction. In contrast to the requirements to source code traceability link recovery task, the best preprocessing technique appears to be less dependent on the dataset for the software architecture documentation to software architecture model traceability link recovery task. This might be due to the datasets being a lot more similar since the gold standard maps single sentences to components, that consist of a name, which often is just a single word, and interface realizations and usages, both also often being only one or two words.

7.6 Software Architecture Documentation to Source Code Traceability Link Recovery

The final pair of artifact types discussed in this thesis are software architecture documentation and source code. Both artifact types have already been discussed in previous chapters.

Unlike the datasets used in the requirement to code traceability link recovery task, BigBlue-Button, MediaStore, and TeaStore contain shell script files. As these do not contain Java methods, the technique to extract Java methods cannot be used for these datasets. Instead, an additional Chunk splitting preprocessing is used, that splits the source code artifacts into elements containing at most 1000 characters. This compliments the rather small number of 200 characters. Just like the previous pairs of artifact types, documentation and source code are classified using no large language model, Prompt 7.1, and Prompt 7.2. Table 7.6 contains the results of the retrieval-augmented large language model traceability link recovery approach using the 40 most similar elements. JabRef and TeamMates are not considered as their dataset contain software architecture documentation sentences that, according to the gold standard, have trace links to a very high proportion of the source code files. JabRef's first sentence for example, is mapped to 1922 out of 1987 source files. TeamMate's first sentence is mapped to 808 out of 834 source files. Since the approach discussed in this thesis is meant to reduce unnecessary language model queries, such trace link mappings cannot work well, as long as a fixed amount of most similar elements is used.

The results of the retrieval-augmented large language model approach do not come close to TransArC's. TransArC achieves more than 0.94 for precision, recall and F1 score for all datasets. The retrieval-augmented large language model approach on the other hand reaches its highest F1 score of 0.345 on TeaStore. Partly responsible for the poor performance is the choice of using 40 similar elements. Ignoring elements that do not appear in the gold standard, sentences from BigBlueButton have 33 trace links on average, and sentences form TeaStore 30. Both contain at least some sentences with more than 40 trace links. Only MediaStore's sentences appear less than 40 times in the gold standard with the most appearing sentence only appearing four times. The precision of not using the language model is therefore poor. Both language model prompts improve the precision on all datasets across all preprocessing groups. Like the results of the requirement to source code traceability link recovery task, there is no preprocessing group which always performs the best. For MediaStore, the Yes/No classifier achieves better F1 scores than No LLM, and the Reasoning classifier achieves better F1 scores than the No/Yes classifier. On the other datasets, No LLM achieved the best F1 score. For BigBlueButton this is the Chunk(1000) preprocessing with an F1 score of 0.217. For TeaStore this is in the group without code preprocessing with an F1 score of 0.344.

7.7 Element Similarity

As mentioned in an earlier section, the amount of most similar elements to consider was thought about, but moved to future work due to time constraints. This section includes some findings. The Figures in this section contain the distance of single elements over the number of elements sorted by their distance. A red dot symbolizes an element which, according to the corresponding gold standard, has a trace link with the element the Figure is about.

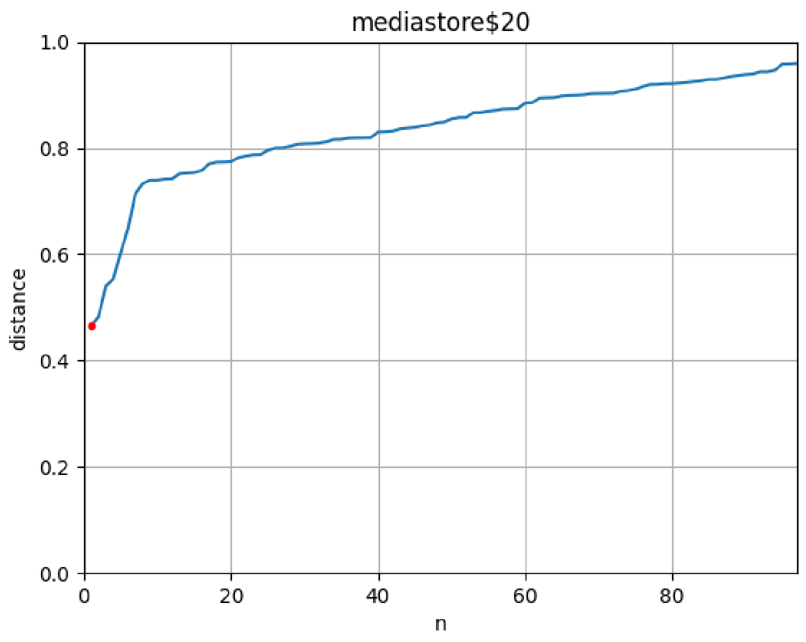


Figure 7.1: [SAD-Code-TLR] Similarity Distance of the n Most Similar Elements For MediaStore Sentence 20, Sorted By Distance

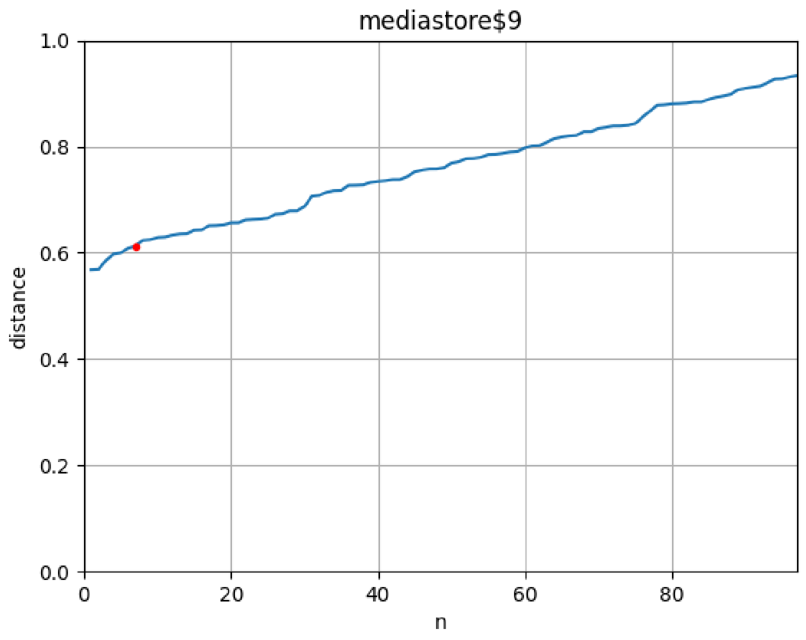


Figure 7.2: [SAD-Code-TLR] Similarity Distance of the n Most Similar Elements For MediaStore Sentence 9, Sorted By Distance

Figure 7.1 shows a very good embedding similarity. The only element with a trace link is at the very beginning and the distance increases quickly. However, the distances for other elements lead to several kinds of problems.

Figure 7.2 also has the only true positive element as one of the lowest elements. However, the distances follow an almost linear curve. There is no clear change at which a threshold could be decided.

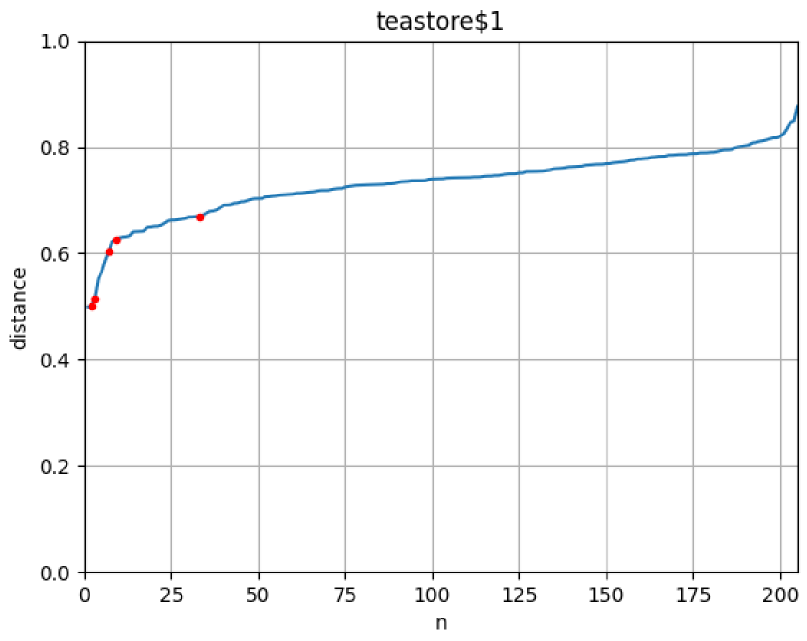


Figure 7.3: [SAD-Code-TLR] Similarity Distance of the n Most Similar Elements For TeaStore Sentence 1, Sorted By Distance

Figure 7.3 shows, that if there is a rapid increase, even after a change to the gradient there might an element that should be retrieved.

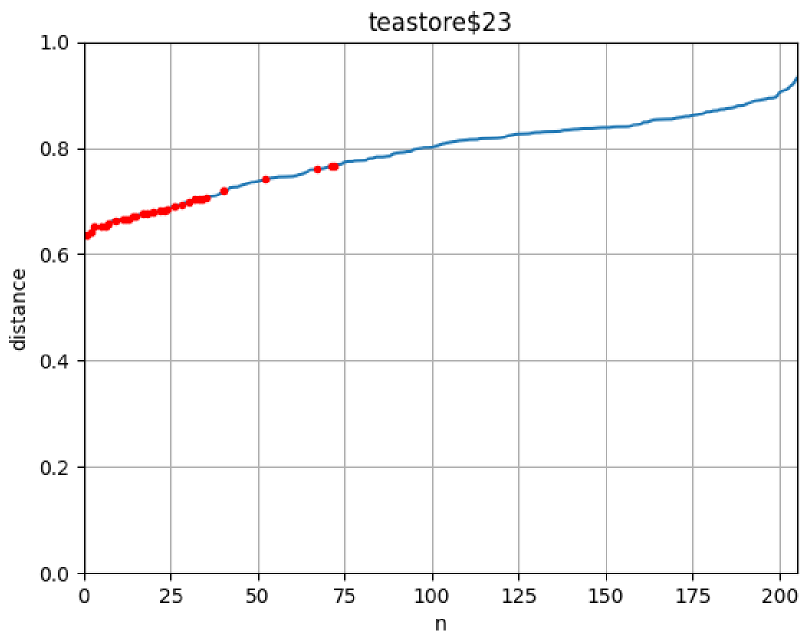


Figure 7.4: [SAD-Code-TLR] Similarity Distance of the n Most Similar Elements For TeaStore Sentence 23, Sorted By Distance

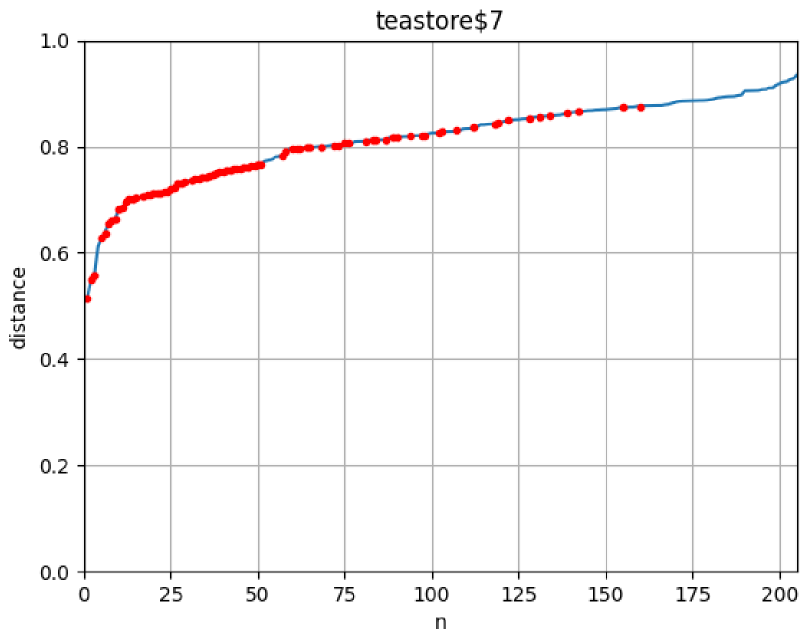


Figure 7.5: [SAD-Code-TLR] Similarity Distance of the n Most Similar Elements For TeaStore Sentence 7, Sorted By Distance

Figure 7.4 and especially Figure 7.5 illustrate a problem of the approach when a single element has a lot of trace links. The wanted elements generally clump in the low distance area in Figure 7.4 and most of them should be considered to be the most similar elements to TeaStore sentence 23, if the number of elements is set high enough. This is somewhat problematic for elements with little or no trace links since far too many element pairs will be compared by the large language model. Figure 7.5 is a lot worse. Sentence 7 has even more trace links, and they do not just clump to the low distances.

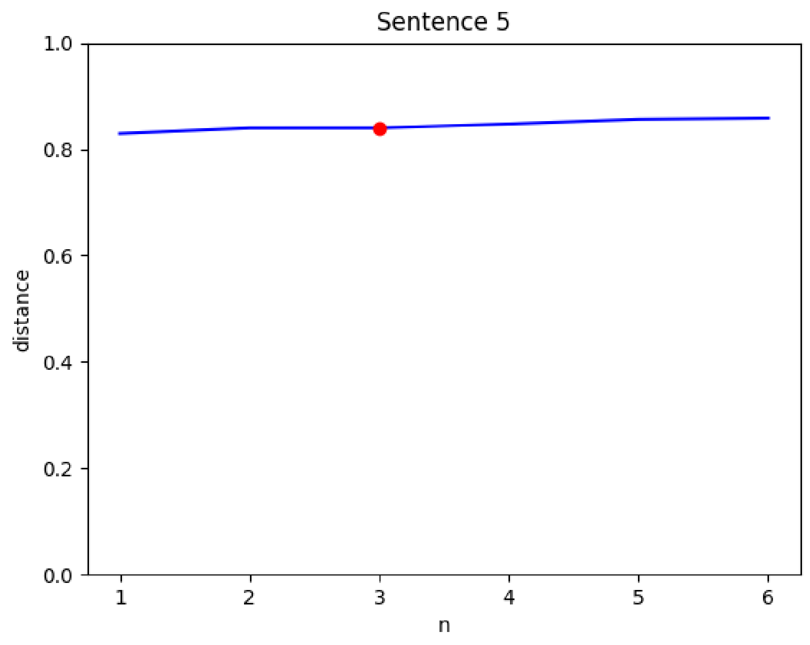


Figure 7.6: [SAD-SAM-TLR] Similarity Distance of the n Most Similar Elements For JabRef Sentence 7, Sorted By Distance

Figure 7.6 shows a curve which is almost completely flat. It barely contains any information on where wanted elements might be. Part of the problem is JabRef's very low count of only six model component artifacts.

Some of those problems might be alleviated by improving the preprocessing. Better preprocessing would move the wanted elements towards lower distances and therefore increase the precision of the retrieval-augmentation. However, problems such as varying numbers of trace links each element within a dataset has, will not be solved by this. Unless wanted elements achieve consistently low values and unwanted elements high values. In this case a change to use thresholds would be preferable.

			eTour				
			Input		Output		Total
			Tokens	Cost	Tokens	Cost	Cost
Preprocessor	Classifier	Elements					
None/None	Yes/No	20	1945348	$9.73	2319	$0.03	$9.76
	Reasoning	20	1990588	$9.95	152143	$2.28	$12.24
	Yes/No	All	11283018	$56.42	13450	$0.20	$56.62
	Reasoning	All	11545410	$57.73	882429	$13.24	$70.96

Table 7.7: Actual Cost of Using the Approach on eTour for the 20 Most Similar Elements and approximate Cost if All Elements Would Have Been Compared

7.8 Cost

Table 7.7 shows the costs of the approach using the eTour dataset as an example. Costs for calculating element embeddings are negligible and are not listed. The prices used are the current prices OpenAI lists for GPT-3.5-turbo on their pricing page [33]: 0.50$ / 1M input tokens and 1.50$ / 1M output tokens. As of the time of writing, GPT-4-turbo costs 20 times, and GPT-4o 10 times as much. Cost for "All Elements" are approximations if each source artifact was paired with each target artifact. The total cost shows how comparing all source and target elements would lead to much higher costs than when only a small subset is compared. ETour is also a relatively small project. The amount of needed comparison only grow linearly for each added source element if a fixed number of most similar elements is used. But they grow exponentially for each element added if all comparisons are done. For large projects the retrieval-augmented approach might be worth using.

7.9 Threads to Validity

In this section the threads to the validity of the evaluation will be discussed.

In terms of internal validity, the main factor is the use of a non-deterministic generating large language model. Due to the randomness, the results might sway when repeating the same experiments. To decrease the random nature of language model replies the temperature was set to 0. To further decrease the effect, I could have run the experiments multiple times while averaging the individual results. This was not done. The number of most similar target elements to consider for each source element was an estimate done based on the collection of datasets.

In terms of external validity, due to the limited amount of projects on which the approach was evaluated, the results might not generalize to other projects. The datasets used have been used in prior work and cover a variety of domains and vary in size. It is possible some datasets or parts of the datasets were used in the training of GPT3.5-turbo. When asked, GPT3.5-turbo claims to know of the iTrust project and correctly assigns it to the healthcare domain. It also gives a correct description of BigBlueButton, TeamMates, TeaStore, and JabRef. Another threat is rapid development of large language models. This thesis only used a very limited number of prompts and a single language model. As a prompt's performance varies between different large language models. It is unclear how the approach generalizes to different language models. I tried to reduce the influence of a potential bias towards the used datasets when creating the prompts used by basing them on prompts used in prior work.

8 Conclusion and Future Works

This thesis set out to develop a system to automatically recover traceability links between different types of software development artifacts. To bridge the semantic gap a retrieval-augmented large language models are used. The system uses three steps, preprocessing of artifacts, retrieval of similar elements to each source element, and classification of source and target elements using the large language model. On the requirements to source code traceability link recovery task, the approach achieved results comparable to the state-of-the-art approaches FTLR and COMET. On the tasks involving software architecture documentation, the approach did not reach a comparable result to ArDoCo and TransArC on most datasets. Also, the approach cannot be used if (single) artifacts are expected to have a very large number of trace links. Based on the few variations of prompts, the result can change quite a bit even if the prompt does not change drastically. However, in most cases the direction of change, that is whether precision and recall increase or decrease, is predictable. Prompts asking for a reasoning tend to perform better than simple yes/no prompts. One reoccurring problem with those prompts, is the freedom the language model has in finding connections between software development artifacts. While evaluating the system it became clear, the best representation of an artifact to find similar artifacts based on similarity of their embeddings is not necessarily the same as the best representation of those artifacts when prompting a large language model. For the requirement to source code traceability link recovery task, configurations using the original artifacts instead of preprocessed elements tended to achieve a better performance. Whether some combination of preprocessed element and original artifact might perform even better is left as future work. How many of the most similar elements to consider when recovering trace links is a non-trivial problem which needs more work. This became very clear when evaluating the software architecture model to software architecture documentation traceability task. Neither a fixed number nor a fixed threshold seem applicable. Not just because of differences between datasets but also differences between artifacts within a project. Future work might focus on dynamically finding a threshold or improving the retrieval part of this approach. It is possible to replace the embedding approach with another traceability link recovery approach and use the large language model as a filter for its results. In general, using the language model lead to an improved performance compared to just the embedding-based retrieval method. It was also shown, that more information can either be beneficial or be a hindrance. If possible it should relate directly to the element or artifact under consideration.

During the thesis, several more future works and ways to enhance the system became clear. Some of them could not be implemented due to time constraints. An additional classifier was planned, that receives multiple target elements at once and the large language model either picks related elements or sorts them by their likelihood to have a trace link to a given source element. Instead of extracting features from the architecture model artifacts it might be better to do no preprocessing and instead prompt the language model using the XML code the UML model was stored as. A problem that was noticed while working with source code files was, that only one specific form of source code, that is Java files, is supported. However, projects can contain several kinds of source code artifacts. The system can be extended with a way to use different preprocessors for different kinds of artifacts withing the source or target group. Lastly, as the landscape of large language models is changing rapidly, the approach can be tested with several, newer language models. Additionally, during the creation of this thesis, OpenAI released a functionality to set a seed value when querying one of their large language models. Future research might want to set a seed to further decrease the non-random nature of language models.

Bibliography

[1] Victor R. Basili and David M. Weiss. “A Methodology for Collecting Valid Software Engineering Data”. In: *IEEE Transactions on Software Engineering* SE-10.6 (1984), pp. 728–738. DOI: `10.1109/TSE.1984.5010301`.

[2] *Center of Excellence for Software & Systems Traceability (CoEST) Datasets*. URL: `http://sarec.nd.edu/coest/datasets.html` (visited on 05/16/2024).

[3] Harrison Chase. *LangChain*. Oct. 2022. URL: `https://github.com/langchain-ai/langchain`.

[4] *ChromaDB*. URL: `https://www.trychroma.com/`.

[5] Jane Cleland-Huang, Orlena Gotel, and Andrea Zisman, eds. *Software and Systems Traceability*. London: Springer London, 2012. ISBN: 978-1-4471-2238-8 978-1-4471-2239-5. DOI: `10.1007/978-1-4471-2239-5`. URL: `http://link.springer.com/10.1007/978-1-4471-2239-5` (visited on 02/04/2021).

[6] Matthijs Douze et al. “The Faiss library”. In: (2024). arXiv: `2401.08281 [cs.LG]`.

[7] *Embedding projector - visualization of high-dimensional data*. URL: `http://projector.tensorflow.org` (visited on 12/11/2023).

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

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

[10] Tobias Hey. “Automatische Wiederherstellung von Nachverfolgbarkeit zwischen Anforderungen und Quelltext”. PhD thesis. Karlsruher Institut für Technologie (KIT) / Karlsruher Institut für Technologie (KIT), 2023. 314 pp. DOI: `10.5445/IR/1000162446`.

[11] Tobias Hey. *Fine-grained Traceability Link Recovery (FTLR)*. Sept. 2023. DOI: `10.5281/zenodo.8367392`. URL: `https://doi.org/10.5281/zenodo.8367392`.

[12] Tobias Hey et al. “Improving Traceability Link Recovery Using Fine-grained Requirements-to-Code Relations”. In: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. ISSN: 2576-3148. Sept. 2021, pp. 12–22. DOI: `10.1109/ICSME52107.2021.00008`.

[13] "IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Std 610.12-1990* (1990), pp. 1–84. DOI: 10.1109/IEEESTD.1990.101064.

[14] Gautier Izacard and Edouard Grave. "Leveraging Passage Retrieval with Generative Models for Open Domain Question Answering". In: *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*. EACL 2021. Ed. by Paola Merlo, Jorg Tiedemann, and Reut Tsarfaty. Online: Association for Computational Linguistics, Apr. 2021, pp. 874–880. DOI: 10.18653/v1/2021.eacl-main.74. URL: https://aclanthology.org/2021.eacl-main.74 (visited on 01/03/2024).

[15] Gautier Izacard et al. *Atlas: Few-shot Learning with Retrieval Augmented Language Models*. Nov. 16, 2022. arXiv: 2208.03299[cs]. URL: http://arxiv.org/abs/2208.03299 (visited on 01/03/2024).

[16] Jan Keim et al. "Detecting Inconsistencies in Software Architecture Documentation Using Traceability Link Recovery". en. In: *2023 IEEE 20th International Conference on Software Architecture (ICSA)*. L'Aquila, Italy: IEEE, Mar. 2023, pp. 141–152. ISBN: 9798350397499. DOI: 10.1109/ICSA56044.2023.00021. URL: https://ieeexplore.ieee.org/document/10092702/ (visited on 12/19/2023).

[17] Jan Keim et al. "Recovering Trace Links Between Software Documentation And Code". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. <conf-loc>, <city>Lisbon</city>, <country>Portugal</country>, </conf-loc>: Association for Computing Machinery, 2024. ISBN: 9798400702174. DOI: 10.1145/3597503.3639130. URL: https://doi.org/10.1145/3597503.3639130.

[18] Jan Keim et al. "Trace Link Recovery for Software Architecture Documentation". In: *Software Architecture*. Ed. by Stefan Biffl et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 101–116. ISBN: 978-3-030-86044-8. DOI: 10.1007/978-3-030-86044-8_7.

[19] Takeshi Kojima et al. "Large Language Models are Zero-Shot Reasoners". In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo et al. Vol. 35. Curran Associates, Inc., 2022, pp. 22199–22213. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/8bb0d291acd4acf06ef112099c16f326-Paper-Conference.pdf.

[20] *Large Language Models (LLMs) with Google AI*. Google Cloud. URL: https://cloud.google.com/ai/llms (visited on 12/10/2023).

[21] Haejun Lee et al. "You Only Need One Model for Open-domain Question Answering". In: *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. EMNLP 2022. Ed. by Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang. Abu Dhabi, United Arab Emirates: Association for Computational Linguistics, Dec. 2022, pp. 3047–3060. DOI: 10.18653/v1/2022.emnlp-main.198. URL: https://aclanthology.org/2022.emnlp-main.198 (visited on 01/03/2024).

[22] Patrick Lewis et al. “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks”. In: *Advances in Neural Information Processing Systems.* Vol. 33. Curran Associates, Inc., 2020, pp. 9459–9474. URL: `https://proceedings.neurips.cc/paper_files/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html` (visited on 10/26/2023).

[23] Hongliang Liang, Dengji Hang, and Xiangyu Li. “Modeling function-level interactions for file-level bug localization”. en. In: *Empirical Software Engineering* 27.7 (Oct. 2022). ISSN: 1573-7616. DOI: `10.1007/s10664-022-10237-z`. URL: `https://doi.org/10.1007/s10664-022-10237-z` (visited on 12/14/2022).

[24] Jinfeng Lin et al. “Traceability Transformed: Generating More Accurate Links with Pre-Trained BERT Models”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE).* ISSN: 1558-1225. May 2021, pp. 324–335. DOI: `10.1109/ICSE43902.2021.00040`.

[25] Pengfei Liu et al. “Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing”. In: *ACM Computing Surveys* 55.9 (Sept. 30, 2023), pp. 1–35. ISSN: 0360-0300, 1557-7341. DOI: `10.1145/3560815`. URL: `https://dl.acm.org/doi/10.1145/3560815` (visited on 11/07/2023).

[26] Zhengmao Luo, Wenyao Wang, and CaiChun Cen. “Improving Bug Localization with Effective Contrastive Learning Representation”. In: *IEEE Access* (2022). Conference Name: IEEE Access, pp. 32523–32533. ISSN: 2169-3536. DOI: `10.1109/ACCESS.2022.3228802`.

[27] Chris Mills et al. “Tracing with Less Data: Active Learning for Classification-Based Traceability Link Recovery”. en. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME).* Cleveland, OH, USA: IEEE, Sept. 2019, pp. 103–113. ISBN: 978-1-72813-094-1. DOI: `10.1109/ICSME.2019.00020`. URL: `https://ieeexplore.ieee.org/document/8919048/` (visited on 12/19/2023).

[28] *Model Pricing.* Anthropic. URL: `https://www-files.anthropic.com/production/images/model_pricing_dec2023.pdf` (visited on 12/10/2023).

[29] *Models | OpenAI.* URL: `https://platform.openai.com/docs/models/gpt-3-5-turbo` (visited on 05/12/2024).

[30] Kevin Moran et al. “Improving the effectiveness of traceability link recovery using hierarchical bayesian networks”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering.* ICSE ’20. New York, NY, USA: Association for Computing Machinery, June 2020, pp. 873–885. ISBN: 978-1-4503-7121-6. DOI: `10.1145/3377811.3380418`. URL: `https://doi.org/10.1145/3377811.3380418` (visited on 01/05/2021).

[31] Shiva Nejati et al. “A SysML-based approach to traceability management and design slicing in support of safety certification: Framework, tool support, and case studies”. In: *Information and Software Technology.* Special Section: Engineering Complex Software Systems through Multi-Agent Systems and Simulation 54.6 (June 1, 2012), pp. 569–590. ISSN: 0950-5849. DOI: `10.1016/j.infsof.2012.01.005`. URL: `https:`

//www.sciencedirect.com/science/article/pii/S095058491200016X (visited on 11/13/2023).

[32] *New embedding models and API updates | OpenAI.* URL: https://openai.com/index/new-embedding-models-and-api-updates/ (visited on 05/26/2024).

[33] *Pricing.* OpenAI. URL: https://openai.com/pricing (visited on 07/06/2024).

[34] Ori Ram et al. "In-Context Retrieval-Augmented Language Models". In: *Transactions of the Association for Computational Linguistics* 11 (Nov. 2023), pp. 1316–1331. ISSN: 2307-387X. DOI: 10.1162/tacl_a_00605. URL: https://doi.org/10.1162/tacl_a_00605 (visited on 12/22/2023).

[35] Alberto D. Rodriguez, Jane Cleland-Huang, and Davide Falessi. "Leveraging Intermediate Artifacts to Improve Automated Trace Link Retrieval". In: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME).* ISSN: 2576-3148. Sept. 2021, pp. 81–92. DOI: 10.1109/ICSME52107.2021.00014.

[36] Alberto D. Rodriguez, Katherine R. Dearstyne, and Jane Cleland-Huang. "Prompts Matter: Insights and Strategies for Prompt Engineering in Automated Software Traceability". In: *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW).* 2023, pp. 455–464. DOI: 10.1109/REW57809.2023.00087.

[37] Roie Schwaber-Cohen. *Vector Embeddings for Developers: The Basics | Pinecone.* URL: https://www.pinecone.io/learn/vector-embeddings-for-developers/ (visited on 12/11/2023).

[38] Roie Schwaber-Cohen. *What is a Vector Database & How Does it Work? Use Cases + Examples | Pinecone.* URL: https://www.pinecone.io/learn/vector-database/ (visited on 12/11/2023).

[39] Weijia Shi et al. *REPLUG: Retrieval-Augmented Black-Box Language Models.* May 24, 2023. DOI: 10.48550/arXiv.2301.12652. arXiv: 2301.12652[cs]. URL: http://arxiv.org/abs/2301.12652 (visited on 01/02/2024).

[40] Hendrik Strobelt et al. "Interactive and Visual Prompt Engineering for Ad-hoc Task Adaptation With Large Language Models". In: *IEEE Transactions on Visualization and Computer Graphics* (2022), pp. 1–11. ISSN: 1077-2626, 1941-0506, 2160-9306. DOI: 10.1109/TVCG.2022.3209479. URL: https://ieeexplore.ieee.org/document/9908590/ (visited on 12/11/2023).

[41] *Text generation - OpenAI API.* URL: https://platform.openai.com/docs/guides/text-generation (visited on 06/18/2024).

[42] James Thorne et al. "FEVER: a Large-scale Dataset for Fact Extraction and VERification". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers).* NAACL-HLT 2018. Ed. by Marilyn Walker, Heng Ji, and Amanda Stent. New Orleans, Louisiana: Association for Computational Linguistics, June 2018, pp. 809–819. DOI: 10.18653/v1/N18-1074. URL: https://aclanthology.org/N18-1074 (visited on 01/03/2024).

[43] *Tokenizer.* OpenAI Platform. URL: https://openai.com/pricing (visited on 12/10/2023).

[44] Jason Wei et al. “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models”. In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo et al. Vol. 35. Curran Associates, Inc., 2022, pp. 24824–24837. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf.

[45] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. “AI Chains: Transparent and Controllable Human-AI Interaction by Chaining Large Language Model Prompts”. In: *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. CHI ’22. New York, NY, USA: Association for Computing Machinery, Apr. 29, 2022, pp. 1–22. ISBN: 978-1-4503-9157-3. DOI: 10.1145/3491102.3517582. URL: https://dl.acm.org/doi/10.1145/3491102.3517582 (visited on 12/11/2023).

[46] Liping Zhao et al. “Natural Language Processing for Requirements Engineering: A Systematic Mapping Study”. In: *ACM Comput. Surv.* 54.3 (Apr. 2021). ISSN: 0360-0300. DOI: 10.1145/3444689. URL: https://doi.org/10.1145/3444689.