Karlsruhe Institute of Technology

Automated Prompt Engineering for Traceability Link Recovery

Bachelor's Thesis of

Daniel Schwab

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

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

23. June 2025 – 23. October 2025

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Automated Prompt Engineering for Traceability Link Recovery (Bachelor's Thesis)

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

Karlsruhe, 23. October 2025

. . . . . .

(Daniel Schwab)

Abstract

Traceability link recovery (TLR) is an important task in software engineering that helps to establish and maintain trace links between different software artifacts. Traditional TLR methods often rely on information retrieval techniques to identify candidate links. More novel approaches use large language models (LLMs) to improve the accuracy and retrieval rates of TLR. However, to utilize LLMs effectively, it is crucial to design appropriate prompts. This process of prompt engineering is often done manually and often time-consuming, requiring significant expertise.

In this work, an automatic prompt engineering (APE) approach is proposed and realized to automate this task for TLR in the Linking Software System Artifacts (LiSSA) [10] framework. Using feedback form previous iterations, LLMs are used to refine prompts. The approach is evaluated on five datasets from the requiremens-to-requirements task using three different LLMs. As a baseline, current classification prompts from the LiSSA framework are used.

The results show that the realized approach can improve the performance of TLR tasks. Furthermore, it is demonstrated that the best-performing optimized prompt for a dataset, in terms of F_1-Score, also performs well on other datasets of the same task. As the optimized prompts yielded by this work can be used as fixed classification prompts, the APE algorithm is not required to be applied for every evaluation. Overall, this work contributes to the field of TLR by realizing an approach to APE using LLMs.

Further research can be conducted in order to explore more degrees of freedom for the APE algorithms. The APE approach can also be investigated in other TLR domains, with different models and new prompting strategies.

Zusammenfassung

Traceability link recovery (TLR) ist eine wichtige Aufgabe im Software Engineering, die dabei hilft, trace links zwischen verschiedenen Softwareartefakten zu etablieren und zu pflegen. Traditionelle TLR-Methoden basieren häufig auf information retrieval-Techniken, um Kandidatenlinks zu identifizieren. Neuere Ansätze verwenden large language models (LLMs), um die Genauigkeit und Rückgewinnungsraten von TLR zu verbessern. Um LLMs jedoch effektiv zu nutzen, ist es entscheidend, geeignete Prompts zu entwickeln. Dieser Prozess des Prompt Engineering wird oft manuell durchgeführt und ist häufig zeitaufwändig, da er erhebliche Expertise erfordert.

In dieser Arbeit wird ein automatic prompt engineering (APE)-Ansatz vorgeschlagen und realisiert, um diese Aufgabe für TLR im Linking Software System Artifacts (LiSSA) [10] framework zu automatisieren. Unter Verwendung von Feedback aus vorherigen Iterationen werden LLMs eingesetzt, um Prompts zu verfeinern. Der Ansatz wird auf fünf Datensätzen aus der requiremens-to-requirements-Aufgabe unter Verwendung von drei verschiedenen LLMs evaluiert. Als Baseline werden aktuelle Klassifikationsprompts aus dem LiSSA framework verwendet.

Die Ergebnisse zeigen, dass der realisierte Ansatz die Leistung von TLR-Aufgaben verbessern kann. Darüber hinaus wird demonstriert, dass der leistungsstärkste optimierte Prompt für einen Datensatz, gemessen am F_1-Score, auch auf anderen Datensätzen derselben Aufgabe gut funktioniert. Da die durch diese Arbeit erzielten optimierten Prompts als feste Klassifikationsprompts verwendet werden können, muss der APE-Algorithmus nicht für jede Evaluierung angewendet werden. Insgesamt trägt diese Arbeit zum Feld von TLR bei, indem ein Ansatz für APE unter Verwendung von LLMs realisiert wird.

Weitere Forschung kann durchgeführt werden, um mehr Freiheitsgrade für die APE-Algorithmen zu erforschen. Der APE-Ansatz kann auch in anderen TLR-Domänen, mit verschiedenen Modellen und neuen Prompting-Strategien untersucht werden.

Contents

List of Figures

List of Tables

1. Introduction

During software development, numerous artifacts are created, ranging from the actual project code to documentation and a multitude of formal and informal diagrams. traceability link recovery (TLR) aims to link correlating artifacts across different domains or versions. For instance, a requirement might be implemented in a specific class. These links are called trace links (TLs). However, establishing and maintaining these links manually is often done incompletely and inconsistently [4]. Further complications, such as inconsistent naming, are frequent issues in software projects [37]. TLR approaches need to be able to deal with these challenges.

Figure 1.1 provides an overview of what these artifacts might look like. We can see various snippets of a larger project, including requirements, source code, architectural diagrams, and documentation. For instance, we have the requirement that a Representational State Transfer (REST) application programming interface should be used. In the documentation, we can read that REST is used to process images. The source code reveals that the `Image` class is utilized by the `REST Facade`. A TL exists between the REST documentation and the REST requirement. Furthermore, the documentation snippet describes the relation

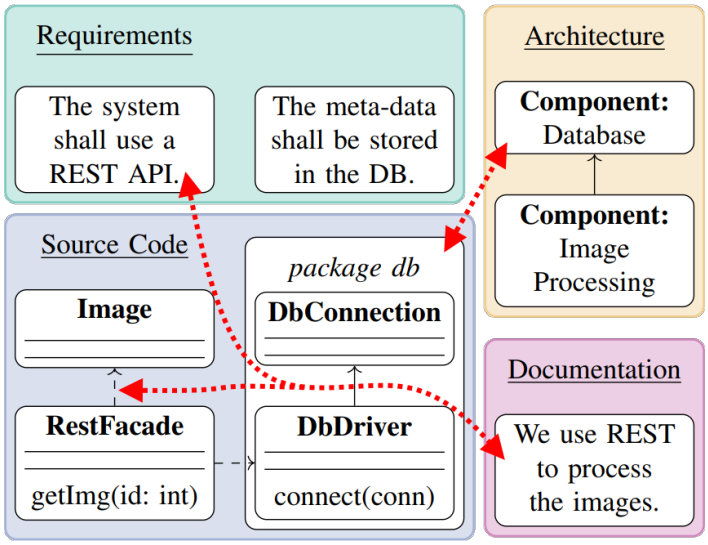

Figure 1.1.: Overview of different artifacts during software development, modified from Fuchß et al. [10].

Artifacts are presented in white, rounded boxes. Existing TLs between artifacts are highlighted with red dashed arrows. The background colors indicate different domains of artifacts.

between the `REST Facade` and `Image`. So a TL exists between them as well. We can also note inconsistencies in naming for the database. In the architectural diagram, it is referred to as `Database`. At the same time, the source code uses the abbreviated `Db` identifier. Technically, with just the knowledge displayed in Figure 1.1, we can not even be certain that `Db` is used to abbreviate database. The documentation does not mention a database at all. However, we can infer that `Db` is likely a database because it is used in the `REST Facade` class, which is getting images from somewhere. Thus, the full name database is a reasonable assumption.

This example illustrates that a multitude of factors need to be considered for TLR. Aside from or rather especially with naming inconsistencies, we need to consider more information to establish TLs correctly. As large language models (LLMs) have made rapid advancements in recent years [42], they have been applied to various tasks in software engineering, including TLR [10, 30]. Especially »LLMs have demonstrated significant potential in automating and enhancing tasks such as requirement elicitation, classification, generation, specification generation, and quality assessment« [19]. Yet they are still unable to reason and think on their own [33], prompt engineering is required to extract good results [30]. However, manually determining prompts suited for each specific problem is quite a tedious, time- and labor-intensive task. Our goal is to determine whether current classification prompts can be refined further to improve TLR rates. As this proves to be promising, we also want to reduce the overall effort required for TLR instead of just shifting the task from manual classification to prompt engineering. To address this shortcoming, automatic prompt engineering (APE) tools can be used. They typically use the LLM itself to determine suitable adjustments or generate new prompts based on a description or training data of the problem [28].

The Linking Software System Artifacts (LiSSA) [10] framework proposes an environment for TLR tasks, including multiple pipeline steps for the preprocessing of input data. They harness the power of LLMs to extract traceability links from a variety of different artifacts, also across multiple tasks. My work contributes an APE component to this framework. It can be used to automatically refine a prompt on a dataset. The refined prompt can improve TLR rates, for projects in the requiremens-to-requirements (Req2Req) task, compared to the current classification prompts in use by LiSSA.

This thesis aims to answer the following research questions:

RQ1: Can APE be used to improve TLR performance in the LiSSA framework?

RQ2: How does the performance of optimized prompts compare to the current TLR performance in the Req2Req task?

RQ3: Can optimized prompts improve TLR performance for untrained datasets?

The remainder of my work is structured as follows: Chapter 2 introduces the necessary fundamentals. Chapter 3 addresses related work in the fields of TLR and APE. Chapter 4 describes the concept of the proposed APE component. Chapter 5 details the implementation of the concept. Chapter 6 evaluates the implementation and answers the research questions. Finally, Chapter 7 concludes the thesis and gives an outlook on possible future work. A replication package for this thesis is also provided [32].

2. Foundations

To understand the application of automated prompt engineering for traceability link recovery (TLR) in the Linking Software System Artifacts (LiSSA) [10] framework, some key concepts are quite important. They are explained in the following sections. First an overview of TLR is given in Section 2.1. Next, the LiSSA framework is introduced in Section 2.2. Finally, automatic prompt engineering (APE) is explained in Section 2.3, followed by a more detailed explanation of the Prompt Optimization with Textual Gradients (ProTeGi) [27] algorithm in Section 2.4. The ProTeGi algorithm is the basis for the most promising implementation of the APE component in this work.

2.1. Traceability Link Recovery

According to the Cambridge Dictionary, traceability is »the ability to find or follow something« [34]. Meaning, there is evidence of some past occurrence. In software engineering, trace links (TLs) are used to link different software artifacts, which are all varying instances of the same element. What these elements might look like is visualized in Figure 1.1. Recovery is defined by the Cambridge Dictionary as »the process of getting something back« [29]. The combination of these two terms, traceability link recovery (TLR), is therefore the process of finding and establishing TLs between existing software artifacts. This is necessary as they are often not established or maintained properly [4]. These traceability links help track relationships between, for example, code, requirements, diagrams, documentation, and other elements. This information can be used to ensure consistency between different artifacts and versions.

Finding all or most TLs in a larger project can be quite a time-consuming task if done manually. This becomes especially apparent when inconsistencies are introduced into the project's artifacts. As shown by Wohlrab et al. [37], inconsistencies in wording and language are quite common during different stages of development. For example, naming conventions for architectural components might not be followed during implementation. They find the impact of these naming inconsistencies to be quite insignificant. In this case, typically, there will be different naming conventions on each layer, which are followed correspondingly. This might thus not be much of a hurdle when a human is manually determining TLs. However, for automated trace link recovery, this means that simple string comparisons by name are insufficient. On the other hand the absence of TLs can also be used as an indicator of inconsistencies. Keim et al. [20] focus on unmentioned model elements and missing model element for software architecture documentation. They have shown that these inconsistencies can be detected reasonable with TLR techniques.

2.2. Linking Software System Artifacts

The Linking Software System Artifacts (LiSSA) [10] framework by Fuchß et al. proposes a generic framework for traceability link recovery tasks. The framework is organized in a pipeline structure with several different modules used to process trace links (TLs) in a modular manner. The current LiSSA evaluation pipeline first retrieves artifacts from the configured providers for the respective source and target domains. They will be preprocessed with the configured preprocessor into generalized elements. Importantly elements contain the content of the artifact as text, which will be used in later steps. Next, embeddings for these elements are created using an embedding model. »Text embeddings are vector representations of natural language that encode its semantic information« [35] By design semantic similar elements will also have similar embeddings. Utilizing the elements and their embeddings, `Element Store` 's are initialized. They provide functionality to search for similar elements based for instance on their embedding vectors. In the LiSSA evaluation pipeline, source elements are combined with similar target element candidates in the `Classifier` module. They will be prompted to an large language model to determine whether a TL exists between the source element and each candidate. This is done using a classification prompt. The classification results are then aggregated and post-processed into a set of TLs.

Prompt 1: keep it short and simple (KISS) by Ewald [7] based on Rodriguez, Dearstyne, and Cleland-Huang [30]
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 1 shows one of the current default classification prompt used in the LiSSA framework. It is a zero-shot prompt based on the KISS principle. This is the base classification prompt which is optimized during the course of this thesis.

2.3. Automated Prompt Engineering

Prompt engineering is the process of refining a prompt for a specific use case. This is typically done in a non-systematic, manual manner. Automatic prompt engineering enables large language models (LLMs) to refine the initial prompt to optimize their performance [40]. Usually, APE processes are iterated to improve previous results further [27, 40, 43, 25]. The performance of prompts can be estimated using a set of training data, which consists of

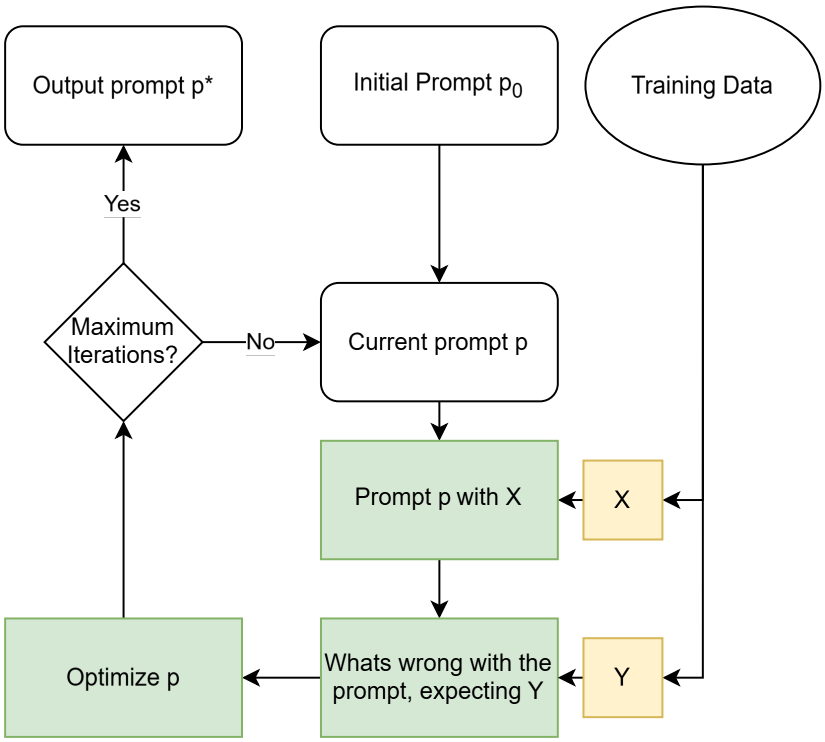

Figure 2.1.: Visualization of a simple iterative optimization algorithm
The green boxes represent prompting calls to the LLM. The yellow boxes are individual data points consisting of some input (x) and the expected classification (y). The diamond represents options for branching. The white boxes are values for or during processing.

textual input/output pairs for the LLM. The exact content of these pairs depends on the context of the problem for which the prompts are optimized. For example, sentiment analysis training data might provide pairs consisting of sentences and their correctly identified sentiment. A prompt performs better the more inputs are correctly mapped onto their expected outputs.

As seen in Figure 2.1, the refinement process will start with some initial prompt. It needs to be selected first before any optimization can occur. This prompt can be chosen manually, which is the most straightforward approach. However, human interaction is still required to choose this first prompt. Zhou et al. [43] propose using LLMs to automate this process and generate the initial prompt. They prompt the model to generate a likely set of instructions, also called candidates. The instructions will achieve good results when prompted with a batch of input/output pairs. These pairs of input/output data will be processed as text by the LLM, pre- and or post-processing may be required.

The initial prompt $p_0$can be optimized using an LLM in the iterative loop illustrated in Figure 2.1. The current prompt p_i is adjusted to a new prompt p_{i+1} using an optimization prompt. The optimization prompt is prompted to the LLM together with the current prompt p_i. The LLM will then return a new prompt p_{i+1}, which is expected to perform better than p_i. This cycle can be repeated until a fixed number of iterations is reached or the performance

of the current prompt p_i exceeds some threshold value. The final prompt p_n will then be returned as the output of the APE algorithm.

While this is a very simple and general attempt to APE, algorithms can of course be more complex as well.

2.4. Prompt Optimization with Textual Gradients

Pryzant et al. [27] propose the Prompt Optimization with Textual Gradients (ProTeGi) [27] algorithm. This entire section is based on their work.

In contrast to the general iterative automatic prompt engineering in Section 2.3, they create multiple improved prompts after each step of the optimization during a single iteration. This is also illustrated in Figure 2.2. The set of candidates for optimization, the set of improved candidates that corrected previous shortcomings, as well as a set of paraphrased improved candidates, are taken into consideration when selecting the candidates that are carried to the next iteration. Once any candidate performs better than some threshold value or a maximum number of iterations has been reached, the best performing candidate is returned as the output of the ProTeGi algorithm.

The following subsections will explain each step of the ProTeGi optimization in more detail. The ProTeGi algorithm also takes an initial prompt p_0 and training data $\{(x_1, y_1), \ldots, (x_n, y_n)\}$ consisting of inputs and outputs. They »assume access to a black box LLM API [...]which returns a likely text continuation y of the prompt formed by concatenating p and x« [27, sec. 2]. They then iteratively optimize the initial prompt p_0 to produce an approximation of the most optimized prompt for the given task. In order to optimize the prompt, a function is required that computes deviance between the actual output y and the expected output y_i as a numeric value.

2.4.1. Evaluation and Expansion

To evaluate the output of the current prompt p_i, Pryzant et al. use a loss signal prompt ∇, the gradient prompt. In addition to the prompt, inputs that were not correctly classified when testing p_i are also provided as context for ∇. The result summarizes the flaws of p_i in natural language. This summary is called the textual gradient g. The second prompt δ, the optimization prompt, is required to adjust p_i in the opposite direction of g to minimize the loss signal.

They differ from other iterative implementations by generating multiple gradients and refinements that might improve the classification rate of p_i. In addition, they broaden their candidate prompts further by paraphrasing them into semantically similar prompts, which are worded differently. Generating new prompts with g and broadening them is considered candidate expansion.

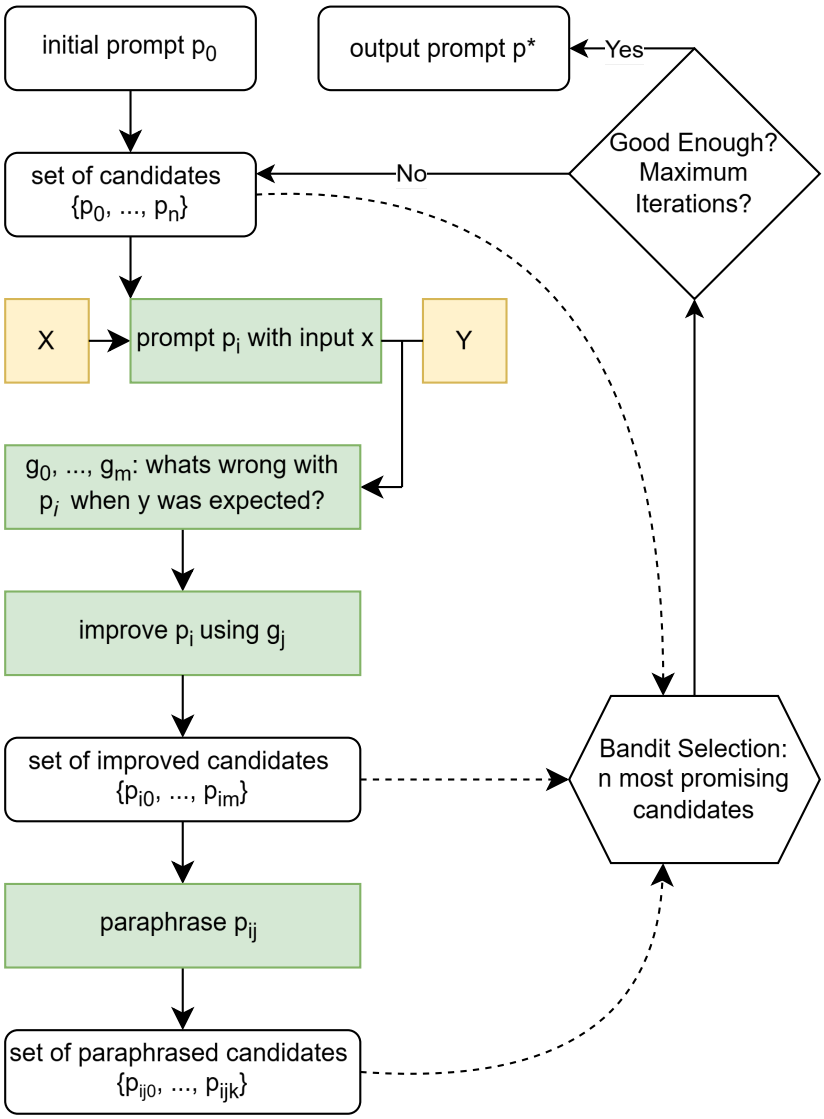

Figure 2.2.: Overview of the iterative optimization loop in [27]
The green boxes represent prompting calls to the large language model. The yellow boxes are individual data points consisting of some input (x) and the expected classification (y). The diamond represents options for branching. The white boxes are values for or during processing.

2.4.2. Candidate Selection with Multi-Armed Bandit

To select candidates for the next iteration, Pryzant et al. apply a beam search algorithm. Beam search is a heuristic best-first graph search algorithm to select a fixed number of promising paths. The remaining paths will be discarded to allocate resources to more promising paths instead [2]. A heuristic algorithm finds a good result efficiently, with the trade-off that it is not guaranteed to be optimal. The selection of the most promising candidates is quite resource-intensive for the entire data set, since many calls to preferably larger large language models are required. As the problem is quite similar to the best arm

identification in multi-armed bandit optimization by Audibert and Bubeck [1], they rely on this well-studied problem instead.

The multi-armed bandit problem consists of N stochastically independent probability distributions $\{D_1, \ldots, D_N\}$ with corresponding expected values and variances. These distributions are unknown to the user initially. The goal is to maximize the sum of rewards for each pull from the set of probability distributions, utilizing the knowledge gained through previous pulls [23]. It is an analogy to multiple one-armed bandit slot machines. Pulling a lever on one of these slot machines does not affect the others but costs noticeable resources. The optimization aims to find the best-performing arms with as few pulls as possible.

Pryzant et al. have used the successive rejects algorithm by Audibert and Bubeck [1] to select the best prompt candidates for the next iteration step. The set of current candidates S is therefore initialized with all k candidate prompts in the iteration step. Depending on the budget for sampling, candidates and remaining candidates in the set S will be prompted with $n(k)$ pairs of the training data. The value $n(k)$ depends on the number of remaining candidate prompts. Other factors, such as a budget to account for pull costs, can also be included. The results are evaluated with a metric function m to quantify the performance. For example, m can be a simple binary function as seen in Equation 2.1. If the prompt p provides the expected output y for a given input x, one is returned. Otherwise, zero will be returned.

$$m(llm(p, x), y) = \begin{cases} 0, \text{if } llm(p, x) \neq y \\ 1, \text{if } llm(p, x) = y \end{cases} \tag{2.1}$$

The lowest scoring prompt is then discarded from S, and sampling is repeated with the remainder until the set S only contains the desired amount of candidates. The remaining candidates will be used as initial prompts for the next expansion step. This process is called successive rejects. A pseudocode implementation of this candidate selection algorithm can be seen in Algorithm 1.

Algorithm 1 »$Select(\cdot)$ with Successive Rejects« [27, Modified from Algorithm 4, p. 4]

1: Initialize: $S_0 \leftarrow \{p_1, \ldots, p_n\}$
2: **for** $k = 1, \ldots, n-1$ **do**
3: Sample $D_{sample} \subset D_{tr}, |D_{sample}| = n(k)$
4: Evaluate $p_i \in S_{k-1}$ with $m(p_i, D_{sample})$
5: $S_k \leftarrow S_{k-1}$, excluding the prompt with the lowest score from the previous step
6: **end for**
7: **return** Best prompt $p^* \in S_{n-1}$

3. Related Work

Traceability link recovery (TLR) through large language models (LLMs) is an active field of work to apply rapid advancements in LLM performance. This work rests at the intersection of two different fields of study. On the one hand side the field of TLR does not always rely on LLMs as shown in Section 3.1. On the other hand, automatic prompt engineering has a much broader application and research than simply being focused on TLR as expanded in Section 3.2. The tasks that TLR can be applied to are discussed in Section 3.3.

3.1. Traceability Link Recovery

Before the emergence of large language models (LLMs) information retrieval (IR) methods are widely used to recover trace links (TLs). De Lucia et al. [6] provide an overview about different IR methods and their performance.

More recently, in work by Hey et al. [14], they attempt to overcome the semantic gap, which occurs when two instances of the same artifact are described using differing semantics, by exploring word embeddings.

In previous work by Fuchß et al. [10], simple prompts of Ewald [7] are used to recover TLs between software documentation and architectural diagrams. The work of Ewald adapts prompts studied by Rodriguez, Dearstyne, and Cleland-Huang [30]. These prompts are manually designed and formulated. They have proven that they can already outperform other state-of-the-art approaches for source code related traceability link recovery (TLR) tasks.

Hey et al. [16] have used these same prompts for TLR on the task of requiremens-to-requirements. While they are also able to outperform state-of-the-art approaches, the recall rate, especially for larger datasets, still has room for improvement.

Rodriguez, Dearstyne, and Cleland-Huang [30] have also evaluated LLM usage for TLR. Their main takeaway is that even minor adjustments, »such as pluralizing words, interchanging prepositions, or reordering phrases« [30, sec. VI] lead to major differences in the outcome. They are unable to find a singular generalized optimal prompt to cover all TLR tasks. They manually adjusted prompts for each task to greatly improve recovery rates.

Zhang et al. [41] use LLMs to augment TLR training data through synthetic TLs. They are able to achieve superior results compared to encoders trained on the original data only.

Fuchß et al. [9] use small LLMs instead of IR based TL candidate selection. They are able to outperform the traditional IR approaches vector space model and latent semantic indexing. The embedding based retrieval of Linking Software System Artifacts [10] TLR performance is however stil greater, especially in regard to precision.

My work will build on these previous developments. I aim to improve F_1-Score further by using automatically optimized prompts.

3.2. Automatic Prompt Engineering

Many prompt optimization algorithms require an initial prompt to start the refinement process by using training data consisting of input/output pairs [28]. Data sets of these pairs, describing the problem to be solved using large language models (LLMs), vary greatly depending on the actual task. For example, sentiment analysis training data might provide pairs consisting of sentences and their correctly identified sentiment. The Automatic Prompt Engineer by Zhou et al. [43] can generate prompts for tasks that are specified only by input/output pairs. This eliminates the need for the initial prompt to seed the optimization process.

Zadenoori et al. [40] propose an iterative feedback based prompt optimization algorithm. They apply it to requirement classification tasks, increasing F_1-Score scores by 5 percentage points.

Self-Refine by Madaan et al. [25] takes feedback from the same LLM that generated the prompt, to improve it further. This imitates human behavior when initial drafts are adjusted rapidly.

Prompt Optimization with Textual Gradients [27] by Pryzant et al. [27] utilizes a gradient descent algorithm to find the minimal deviance between an optimized prompt and the expected outputs. More details about their work are already introduced in Section 2.4.

Yang et al. [38] have taken a slightly different approach in their Optimization by Prompting [38] optimizer. Instead of adjusting the current iteration of prompts, they generate fully new, independent prompts instead. This aims to reduce the bias of modifying existing prompts and encourages further exploration.

In order to reduce uncertainty and improve reproducibility, the Declarative Self-Improving Python [21] Framework by Khattab et al. [21] proposes a composite like structure in Python to program prompts. They are also generating and refining the prompts in a pipeline-like structure, not unlike other LLM-based prompt optimization algorithms.

My work focuses on the optimization algorithm by Pryzant et al. While my work does not expand the field of automatic prompt engineering directly, I am adapting it into the domain of traceability link recovery. Existing training sets can be used to optimize the prompts.

3.3. Traceability Application

Many different applications exist for traceability link recovery (TLR). Often, authors focus on one or a few tasks instead of attempting to cover everything.

The Linking Software System Artifacts [10] framework focuses on code related artifacts. Several datasets can be used for this. Software architecture documentation to software architecture models for BigBlueButton, MediaStore, Teammates, and Teastore are publicly available by Fuchß et al. [8] on GitHub. For the task of requiremens-to-requirements (Req2Req), which my work focuses on, a compilation of training data can be found in the replication package [15] for recent work of Hey et al. This includes the five datasets used in my work. • CM-1NASA [13] • Dronology [3] • GANNT [17] • MODIS Dataset [12] • WARC [22] They are collected from various sources over the years and have been used in multiple preceding works on Req2Req TLR. Aside from Dron. they are provided by the Center of Excellence for Software & Systems Traceability community[5].

Santos et al. [31] apply TLR techniques in a different domain, to determine requirements satisfiability. They examine various smartphone applications for satisfaction of consent requirements of the european general data protection regulation

4. Approach

This chapter outlines how the research questions of the thesis are addressed. The produced source code can seamlessly integrate into the Linking Software System Artifacts [10] project. In this thesis, four different methods are used to test their traceability link recovery performance. First, a simple tree-of-thought style prompt is used for classification instead of the existing zero-shotor chain-of-thought style prompts. The usage of this prompting style is presented in Section 4.1. Next, naive iterative prompt optimization approaches with and without feedback are elaborated in Section 4.2.1. They provide basic functionality upon which the more sophisticated Prompt Optimization with Textual Gradients [27] algorithm in Section 4.2.2 is based.

4.1. Tree-of-Thought Prompting

Tree-of-thought (ToT) is a prompting technique exploring more different branches of thoughts than typical chain-of-thought (CoT) prompts. This can be achieved by chaining multiple requests to an large language model (LLM) [24].

In Figure 4.1, we can see the three different prompting techniques highlighted in green. Among them are, aside from the ToT prompt, also a zero-shotand CoT prompt, to visualize the differences. The input data, highlighted in yellow, is queried with one of the prompting techniques to the LLM. In our case, this can be a pair of requirements, for which the output should state whether a trace link exists between them. Examining the three different prompting techniques, we note a varying number of thoughts before the output is reached. Thoughts are represented by white and orange rounded boxes, with orange ones being discarded during the thought process. As for the zero-shot prompt, the LLM reaches its conclusion without a thought process. The CoT style prompt results in a linear thought path. Thus, if any thought deviates, the LLM might be led to a false conclusion [36]. Prompting with ToT addresses this, as we can see multiple thought explorers per level independently. This concrete number is chosen just for illustrative purposes. If and only if the LLM identifies them as wrong, they will be discarded. The output is then reached as a consensus of these multiple paths.

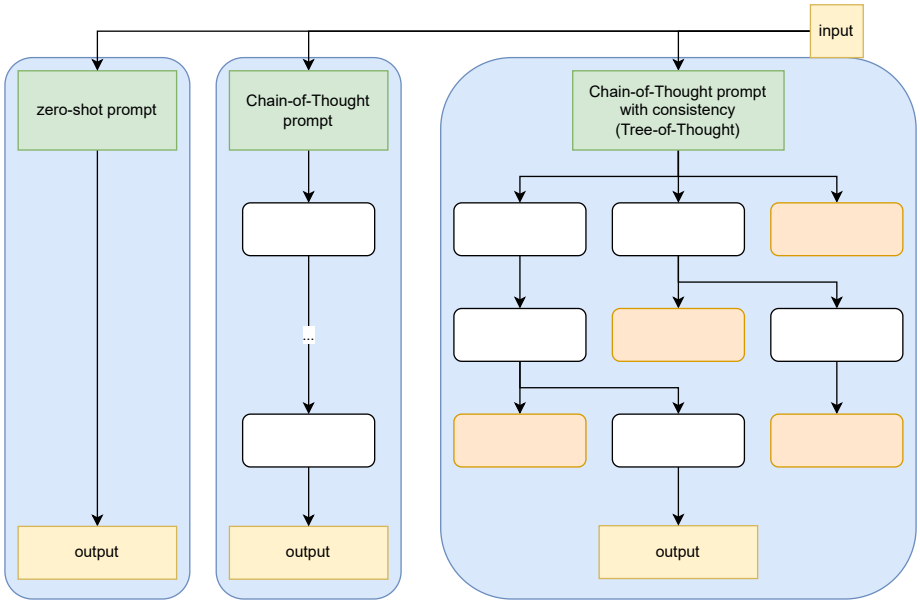

Figure 4.1.: ToT visualization inspired by Yao et al. [39]
Highlighted in green are different prompting techniques. Their related paths are enclosed in the light blue box. Highlighted in yellow is data, with the input, which is queried using one of the prompting techniques to the LLM, and output representing the returned result of this query. The rounded boxes represent thoughts of the LLM. Orange thoughts are considered inconsistent and will be disregarded in the next step. Arrows represent the processing flow of data.

Prompt 2: Excerpt of a ToT prompt by Hulbert [18]
Identify and behave as three different experts that are appropriate to answering this question.
[...]
If any expert is judged to be wrong at any point then they leave.
[...]

The prompts by Hulbert [18] aim to simplify ToT prompting into a singular request. They provide three different prompts to achieve this. The complete best performing ToT Prompt A.1 for traceability link recovery (TLR) in the Linking Software System Artifacts (LiSSA) [10] framework can be found in the appendix. As illustrated in Figure 4.1, many branches are explored concurrently instead of following a single strain of thoughts like CoT prompting. We can see this instruction in Prompt A.1. The LLM is instructed to act as multiple experts to initiate multiple thought processes concurrently. Discarding misleading thoughts, symbolised by orange boxes in Figure 4.1 is instructed by removing the experts from the simulated discussion. The benchmark data to compare ToT prompting with the existing zero-shot and CoT classifiers in the LiSSA framework for requiremens-to-requirements (Req2Req) also exists in public repositories [8, 15]. Adding this prompting style to the LiSSA framework will broaden the baseline to compare later results and possible improvements against.

The LiSSA framework is publicly accessible in a repository using the MIT license. Various classifiers are already included there. To apply the ToT prompting technique, existing classifiers can be modified in their classification prompt to utilize Prompt A.1 instead. Of course the relevant question for Req2Req TLR will be included in the actual prompt as well. This ToT classification has two major goals. Foremost, to get confident and set up with the general project structure and benchmark data. This is a prerequisite for my future work in the LiSSA framework. Second, this broadens the baseline to compare later results and possible improvements against.

4.2. Automatic Prompt Engineering in LiSSA

To add automatic prompt engineering (APE) capabilities to the Linking Software System Artifacts (LiSSA) [10] framework, a new pipeline step is introduced. Figure 4.2 provides an overview of the proposed approach. The new optimization step, highlighted in light blue, is an iterative pipeline step to alter the classification prompt. It is placed after the initial preprocessing of the input data. The `Prompt Optimizer` component uses the preprocessed data to optimize a given prompt. The optimized prompt is then used in the regular large language model (LLM)-based prompting step to classify trace links.

The different prompt optimization algorithms to be used are be introduced in the following subsections. First a naive iterative approach is presented in Section 4.2.1. This implementa-

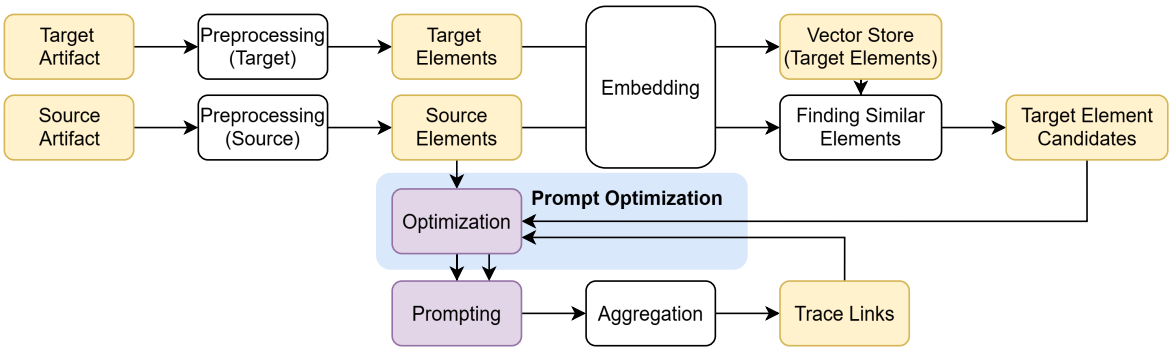

Figure 4.2.: Overview of the proposed prompt optimization approach for traceability link recovery (TLR) tasks in the Linking Software System Artifacts (LiSSA) framework. Modified from Fuchß et al. [10]

The new prompt optimization component is highlighted in light blue. Highlighted in yellow are the actual artifacts. Highlighted in purple are steps prompting large language models (LLMs) for processing. White nodes represent processing inside the framework.

tion is the basis for the more sophisticated Prompt Optimization with Textual Gradients (ProTeGi) [27] algorithm in Section 4.2.2.

4.2.1. Naive Iterative Automatic Prompt Engineering

To add APE capabilities to the LiSSA framework, a naive iterative optimizer is added. As in Section 4.1 the existing classifiers of the LiSSA framework can be used to evaluate the traceability link recovery (TLR) performance of the optimized prompts.

Prompt 3: Simple Optimization Prompt
Optimize the following prompt to achieve better classification results for traceability link recovery. Traceability links are to be found in the domain of {source_type} to {target_type}. Do not modify the input and output formats specified by the original prompt.
Enclose your optimized prompt with <prompt></prompt> brackets.

The original prompt is provided below:
'"{original_prompt}"'

To improve TLR performance, Prompt 4.2.1 is used as the optimization prompt. As visualized in Figure 2.1 the expected outputs for training data may also be used to improve the prompt. The most naive approach is to simply ask the LLM to improve the prompt without providing further direction or information. Context about why the prompt did not meet the requirements, also referred to as feedback, is not yet provided with Prompt 4.2.1. The optimized prompt is taken into the next iteration. The loop of evaluating and optimizing is repeated until a threshold value is passed, meaning the prompt is considered good enough.

To limit resource usage, a hard limit is also included to put an upper bound on the number of optimization attempts.

This naive `Prompt Optimizer` implementation is used as a basis for more sophisticated prompt optimization algorithms. Furthermore, we are able to compare whether optimization results differ between naive and sophisticated approaches.

4.2.2. Automatic Prompt Engineering with ProTeGi

In this subsection, feedback is introduced to the optimization process. The naive iterative optimization implementation created for Section 4.2.1 is used as a foundation for the implementation. The core concepts of the ProTeGi optimization algorithm remain the same. Details of the ProTeGi algorithm are introduced in Section 2.4.

First the Naive Iterative Automatic Prompt Engineering is extended to include simple feedback strings about misclassifications. The feedback is generated by comparing the actual output of the LLM classification with the expected output for the training data. This feedback is then provided to the LLM in the optimization prompt. However, this does not include any processing logic to analyze or select the optimized prompts. This is addressed by implementing the ProTeGi algorithm.

$$f : llm(p, X) \rightarrow [0, 1] \tag{4.1}$$

$$t \in [0, 1] \tag{4.2}$$

We can compare the performance of prompts using a metric function. More formally as written in Equation 4.1, a function f will be required to quantify the result, also referred to as performance. The current prompt p iteration, called to a LLM llm using a set of training data X, returns this result. The metrics provided by an LiSSA evaluation run: precision, recall and F_1-Score, can be used as this performance function. The function f needs to have the same range for any set of training data to ensure the threshold value t will always be viable.

Pryzant et al. [27] have provided their Python source code in a publicly accessible repository[1] under the MIT license. The project code for their algorithm, is the basis for the adaption to the LiSSA Java framework. The prompt candidates are compared with each other using the performance function f. The best performing prompts are then taken into the next iteration of the optimization loop. Feeding more information and processing logic into the optimization system is expected to present better results than the naive approach. Especially, suitable results may be reached faster and or cheaper by more processing on the local machine instead of expensive application programming interface calls to the LLM.

[1] https://github.com/microsoft/LMOps/tree/main/prompt_optimization

5. Implementation

This chapter describes the implementation of the proposed automatic prompt engineering features to the Linking Software System Artifacts (LiSSA) [10] framework to optimize classification prompts. As outlined in Chapter 4, the goal is to optimize prompts for a given set of traceability link recovery (TLR) tasks on a fixed dataset. To achieve this goal, a new `Prompt Optimizer` component is added to the LiSSA pipeline. This is elaborated in Section 5.3. This component can be combined with further optional `Metric` and `Evaluator` components introduced in Section 5.1 and Section 5.2 to enable a flexible and reusable architecture. The integration into the existing LiSSA pipeline is explained in Section 5.4.

Figure 5.1 provides an overview of the components involved in this implementation. The `Prompt Optimizer` component is the central part of the implementation. The `optimize` method is used to optimize the prompt for a given set of TLR tasks. The source and target `Element Store` 's as in the existing LiSSA pipeline are provided as arguments to the method for training data. The `Metric` component is used to score the quality of a prompt on a given set of tasks. The `getMetric` method is used to retrieve the metric implementation for all prompts on set of classification tasks. The `Evaluator` component is used to select a subset of tasks to evaluate the prompt on using the `Metric` . Its `sampleAndEvaluate` method is used to selectively evaluate the prompts on a subset of tasks. Ideally more evaluation budget is spent on prompts that are expected to perform better.

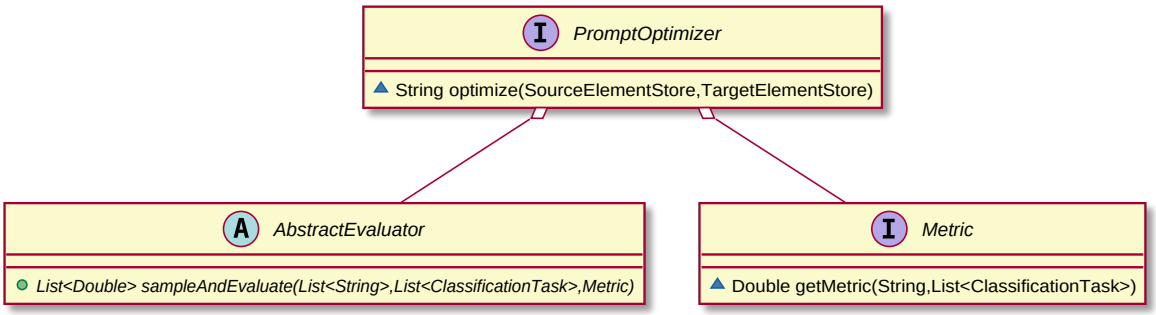

Figure 5.1.: Overview of the major new components involved in the implementation

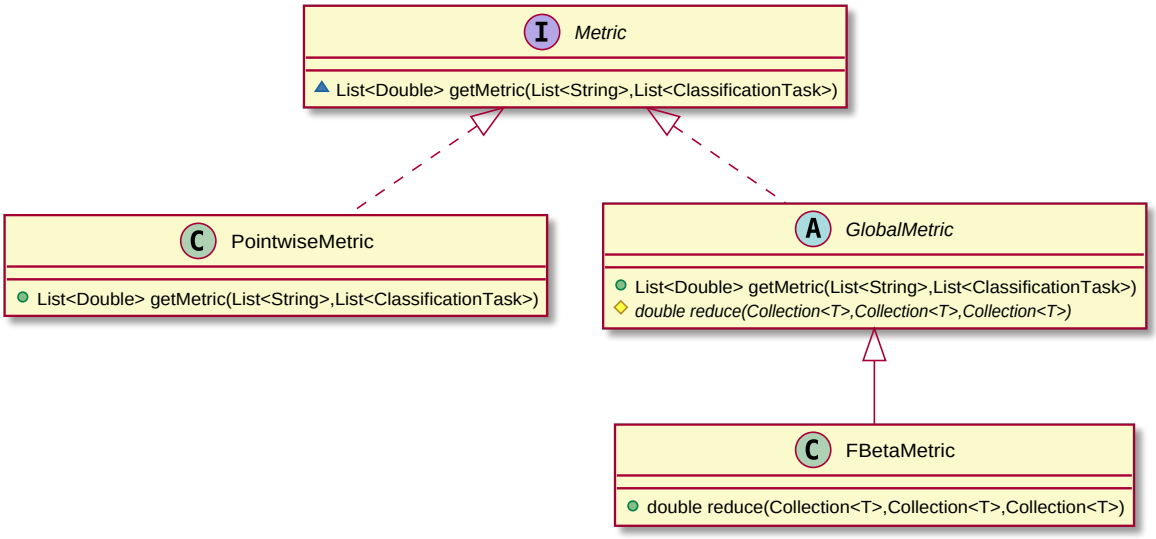

Figure 5.2.: The new `Metric` component and its implementations

5.1. Metric component

The `Metric` component provides implementations for different metrics to score the quality of a prompt on a given set of tasks. Metrics can be divided into two major categories. They are visualized in Figure 5.2. Pointwise metrics will evaluate each trace link (TL) prediction and return a separate score for each. Scoring strategies for pointwise metrics are provided by the `Scorer` subcomponent. The overall score can then be calculated by reducing the individual scores using a `Reductor`. The `Reductor` subcomponent provides different strategies to reduce a collection of scores to a single value. In contrast, global metrics will evaluate the entire set of TL predictions at once and return a single score directly. This is because they factor in relationships between different predictions, like whether the correct prediction is a true positive or a true negative. The scores are cached using Linking Software System Artifactss [10] built-in `Cache` component.

Currently, only the F_β-Score metric is implemented as a global metric. The beta parameter can be altered in the component configuration. The score is computed utilizing a `reduce` method, that is provided the ground truth as well as the accepted and rejected values.

The pointwise metric implementation relies on the `Binary Scorer` and the `Mean Reductor`. This is also portrayed in Figure 5.3. A `Scorer` is required to implement the `score` logic for a list of tasks and their classified results. The `Binary Scorer` will return a score of 1.0 for a correct prediction and 0.0 otherwise. A correct prediction is defined as predicting a TL for a pair of artifacts that actually has a TL in the ground truth or not predicting a TL for a pair of artifacts that does not have a TL in the ground truth. An `Reductor` is required to implement the `reduce` logic for a collection of individual scores. The `Mean Reductor` will then compute the mean of all individual scores to return a single value.

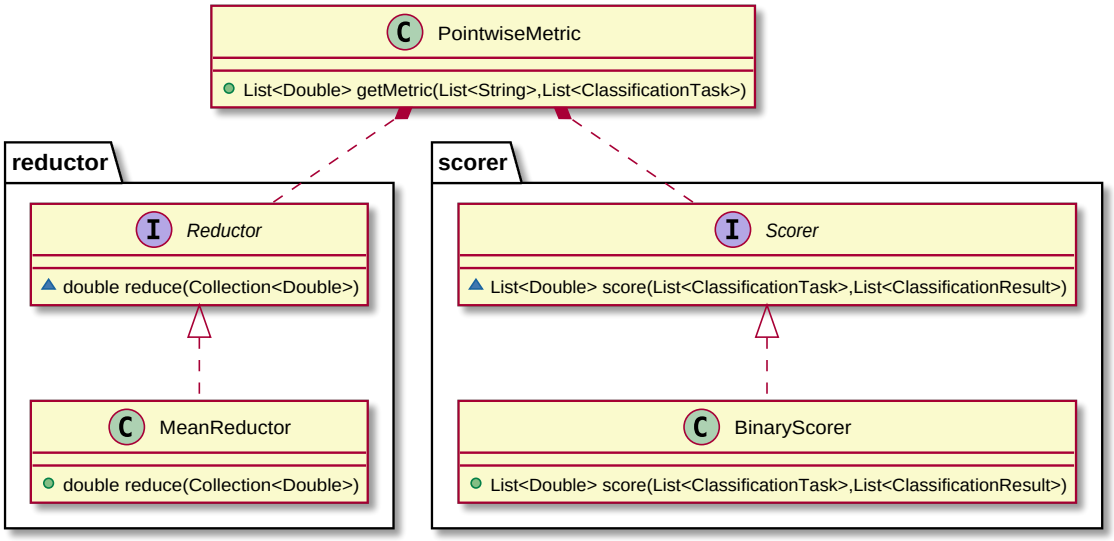

Figure 5.3.: The subcomponents utilized by the pointwise metric implementation

5.2. Evaluator component

The `Evaluator` component provides implementations to evaluate the quality of a prompt on a given set of tasks. In contrast to the `Metric` component, the `Evaluator` will not use all tasks to compute the prompt score. Instead, more complex sampling strategies can be used to select a subset of tasks. This is especially useful for larger datasets, where evaluating all tasks would be too costly. The upper confidence bound (UCB) bandit evaluator is the most noteworthy implementation. It will use the UCB algorithm to select a subset of tasks that are expected to provide the most information gain. This is achieved by balancing exploration and exploitation. The size of the subset can be configured in the component configuration. The UCB evaluator is supplied with a random seed to ensure reproducibility of results.

5.3. Prompt Optimizer component

The prompt optimizer component provides implementations for the actual prompt optimization. Concrete objects will be instantiated with a set of configuration parameters, again following the format of the Linking Software System Artifacts (LiSSA) [10] framework. A factory class is used to create the object based on the configuration. Furthermore, the `Metric` and `Evaluator` components are provided to the respective constructors if required. The `Metric` component is used to score the quality of a prompt on a given set of tasks. The `Evaluator` component is used to select a subset of tasks to evaluate the prompt on. This component is designed to be used in the LiSSA evaluation pipeline after the preprocessing of artifacts into `Element Store` 's is completed. These steps are already part of the existing LiSSA framework for evaluation. They are explained in more detail in Section 2.2. The only

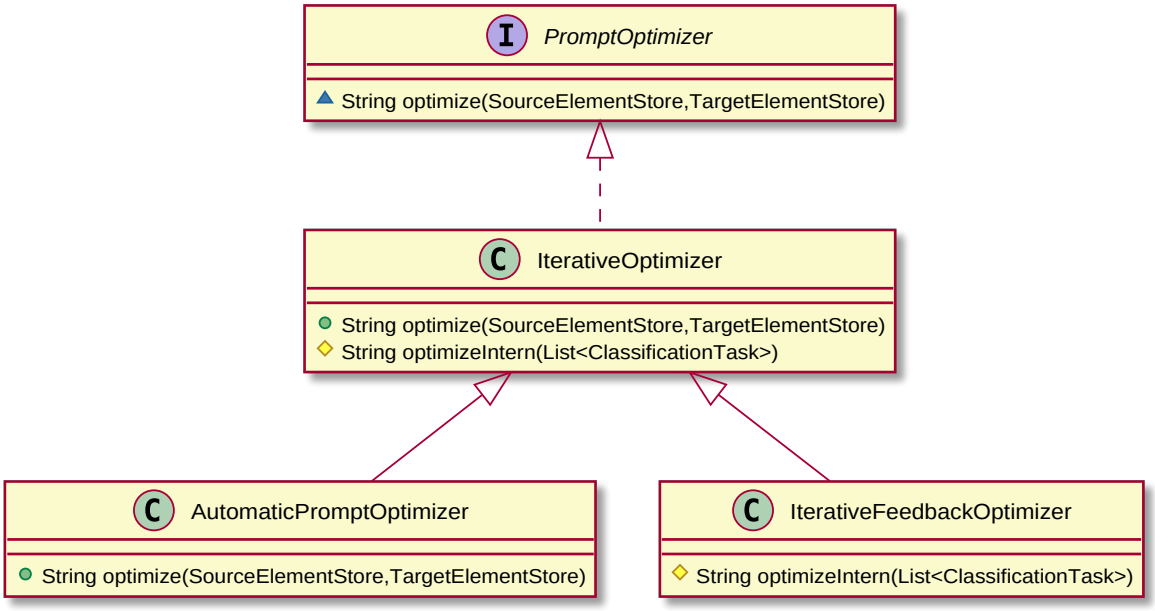

Figure 5.4.: The new `Prompt Optimizer` component and its implementations

functionality required in the `Prompt Optimizer` interface is the `optimize` method. This method can also be seen in Figure 5.4. The `optimize` requires the source and target `Element Store` 's created in the previous pipeline steps as arguments. Using the retrieval strategy of the target `Element Store` a set of candidate pairs of elements is created. This set will also be referred to as the training data, on which the prompt will be optimized.

The prompt to be optimized is provided in the component configuration during the optimizers' instantiation. The method will return the optimized prompt as a string. When `Prompt Optimizer` implementations utilize `Classifier` 's, the training data or subsets will be passed on to the classification method, integrating into the existing LiSSA classification pipeline. Thus, utilizing LiSSAs caching for the large language model (LLM) queries, the optimization is deterministic and reproducible. All further implementations should also be deterministic and reproducible. Configuration parameters such as random seed are used to ensure this.

The `Prompt Optimizer` component will typically use the LLM to evaluate different prompts. Three different implementations are provided, as shown in Figure 5.4. The `optimizeIntern` is overwritten in the `IterativeFeedbackOptimizer` and to add the feedback of previously misclassified trace links to the optimization prompt. Combined with the `IterativeOptimizer` this implements the naive automatic prompt engineering algorithms from Section 4.2.1. Many implementations of the `Prompt Optimizer` component will require sampling strategies to reduce the number of calls to the LLM. This keeps the optimization cost manageable, as not every possible classification task is queried with the optimized prompt candidates. This is achieved by introducing the new `Sample Strategy` subcomponent.

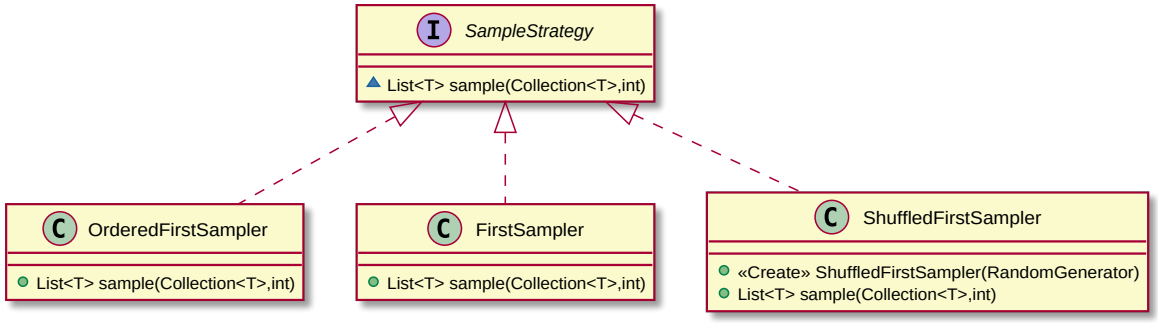

Figure 5.5.: The new `Sample Strategy` subcomponent and its implementations

The `Sample Strategy` provides different strategies to select a sublist of comparable elements from a collection. The strategies can be instantiated with configuration parameters and are used by the `Prompt Optimizer` component. This way, sampling strategies can be swapped easily without changing the actual optimization logic. The random sampler uses a fixed random seed to ensure reproducibility of results.

In Figure 5.5, the sampling strategy subcomponent is depicted. The `sample` method is required to be implemented by all sampling strategies. It will receive a collection of comparable elements and the desired sample size as arguments. Currently, three different strategies are implemented.

- The `ShuffledFirstSampler` will shuffle the input list using the `RandomGenerator` and return the first elements.
- The `OrderedFirstSampler` will sort the input list and return the first elements.
- The `FirstSampler` will return the first elements of the input list.

5.4. LiSSA Pipeline Integration

The new components are integrated into the Linking Software System Artifacts (LiSSA) [10] pipeline. A new command was added to the LiSSA command-line interface to run the prompt optimization. The command will read the configuration file and create the required components using their constructors or factories, respectively. To do so an evaluation pipeline object is created and relevant components will be retrieved from it. This includes the classifier, aggregator, trace link id postprocessor as well as the source and target `Element Store` 's. The classifier component is used to classify a pair of elements having a trace link or not during the optimization process.

The features integrate seamlessly into the existing LiSSA architecture, espcially existing evaluation configurations can still be used. The current classifiers can be utilized by the `Metric` implementations to classify pairs of elements during the optimization process. This enables optimization of both zero-shot and chain-of-thought classification prompts.

The tree-of-thought classification utilizes the existing `Classifier` 's in LiSSA as well. To integrate sampling of optimization candidates, the `Classifier` 's are extended to modify their classification prompt. This functionality is only used by the `Metric` component.

While the optimization pipeline, as a standalone, is independent of the regular evaluation pipeline, they can be combined. This is achieved by feeding the optimized prompt from the optimization pipeline into the evaluation pipeline, indicated by a separate configuration. End-to-end tests are provided to ensure the correct integration of the new features into the existing LiSSA framework. The implementation is open source and available in the replication package [32].

6. Evaluation

The three different automatic prompt engineering approaches and tree-of-thought (ToT) promting proposed in Chapter 4 are evaluated in this chapter. To make the results comparable, the same datasets as in Hey et al. [16] are used. In Section 6.1, the evaluation approach is described. Section 6.2 describes the general evaluation setup, including datasets and large language models used. First, Section 6.3 presents the results of ToT prompting for traceability link recovery in the Linking Software System Artifacts [10] framework. Then, Section 6.4, Section 6.5, and Section 6.6 present the naive prompt optimization approaches. The findings of gradient-based prompt optimization are presented in Section 6.7. They are also tested for their generalizability to new datasets in Section 6.8.

6.1. Evaluation Approach

$$Precision = \frac{TP}{TP + FP} \tag{6.1}$$

$$Recall = \frac{TP}{TP + FN} \tag{6.2}$$

$$f_\beta = (1 + \beta^2) \cdot \frac{Precision \cdot Recall}{(\beta^2 \cdot Precision) + Recall} \tag{6.3}$$

Evaluating the experimental results of prompt optimization is central to making them accessible and comparable. Precision (Equation 6.1) and recall (Equation 6.2) are key measures for information retrieval tasks [13]. They are commonly used by many authors to compare different approaches to traceability link recovery. Further the F_1-Score using a fixed β in Equation 6.3, are used commonly [10, 16]. » 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 F_1-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. « [7, sec. 7.1] These three measures are also used in my thesis. Hey et al. [16] and Fuchß et al. [10] have used these when evaluating their results on the same data sets I intend to use. This way comparison is very simple.

	Artifact Type		Number of Artifacts		
Dataset	Source	Target	Source	Target	Traceability Links
CM-1NASA	HLR	LLR	22 (86%)	53 (57%)	45
Dronology	HLR	LLR	99 (93%)	211 (99%)	220
GANNT	HLR	LLR	17 (100%)	69 (99%)	68
MODIS Dataset	HLR	LLR	19 (63%)	49 (63%)	41
WARC	HLR	LLR	63 (95%)	89 (89%)	136

Table 6.1.: Overview of the datasets adjusted from Hey et al. [16, Table 1]

6.2. Setup

To evaluate the automatic prompt engineering (APE) algorithms proposed in Chapter 4, a multitude of different datasets will be used. They are taken from the replication package of Hey et al. [15]. Some of the gold standard files are modified to provide consistency between artifact naming and the gold standard reference. The actual contents are not affected by this. The CCHIT dataset is omitted, as it differs from the others in that it does not link high-level artifacts with low-level artifacts. An overview of the sets used can be seen in Table 6.1. »Datasets comprise [...]high-level requirements (HLR), low-level requirements (LLR) [...]. Percentage of linked source and target artifacts in the gold standard is given in brackets.« [16]

Several large language models (LLMs) are be used. The focus is on OpenAI's `GPT-4o-2024-08-06` [26] and `GPT-4o-mini-2024-07-18` [26] models. Compared to the locally hosted `codellama:13b` and `llama3.1:8b` models by Meta AI, they enable faster evaluation with parallel requests instead of limitations through the host-hardware. application programming interface access is already implemented in the Linking Software System Artifacts (LiSSA) [10] framework. In addition to the four LLMs used by Hey et al. [16] in their preceding work. `GPT-5-mini-2025-08-07` [11] is also briefly addressed, as a newer LLM. As of this publication, it is the most up-to-date text embedding model by OpenAI, thus possibly improving future comparability.

The LiSSA framework enables the usage of different embedding models for evaluation. However, previous work [10, 16] exclusively uses `text-embedding-3-large` by OpenAI. Thus, to ensure comparability of results, this embedding model will also be used in this evaluation. The similarity retriever in the LiSSA pipeline will also not be modified for this evaluation. The default cosine-similarity-retriever will be utilized. Future work may address the current work of Fuchß et al. [9].

Depending on the APE algorithm, different variables, such as the optimization prompt, are introduced. Unless specified, each evaluation will assume that the model used to optimize the prompt will be used to classify the trace links later to derive the quantitative metrics for evaluation. The datasets and exact configurations used for each evaluation are also available in the replication package [32] for this thesis.

6.3. Tree-of-Thought Prompting Evaluation

Before evaluating the proposed automatic prompt engineering (APE) approaches, tree-of-thought (ToT) prompting by Hulbert [18] is evaluated for the task of traceability link recovery (TLR) in the Linking Software System Artifacts (LiSSA) [10] framework. Table 6.2 shows the systematic evaluation results for varying datasets using ToT prompting compared to the simple keep it short and simple (KISS) Prompt 1. The average (Avg.) and weighted average (W. Avg.) are also included. The Avg. value is computed across all datasets, while the W. Avg. factors the dataset size in as well. The amount of trace links in the gold standard reference for the dataset, seen in Table 6.1, is used as the metric for dataset size.

We can see in table Table 6.2, that the ToT prompts by Hulbert [18] fail to outperform the simple KISS Prompt 1 for all tested datasets on the `GPT-4o-mini-2024-07-18` [26] large language model. Precision is slightly higher for several datasets, but recall and F_1-Score are lower across the board. The ToT prompts are not promising for TLR in the LiSSA framework. Instead, this thesis focuses on the APE approaches described in Chapter 4.

6.4. Naive Prompt Optimization

The initial thought people may have, when thinking about prompt optimization, is to ask the large language model (LLM) to optimize the prompt before usage. To utilize this very simple approach, just two prerequisites are required. Firstly, we need our prompt, which is to be optimized. Prompt 1 is chosen here as the initial prompt. It is a keep it short and simple (KISS) zero-shot binary classification prompt. As the Linking Software System Artifacts (LiSSA) [10] framework has already employed this prompt, the automatic prompt engineering algorithm is initialized with it as well. Secondly, an optimization prompt is needed. Therefore, Prompt 3 has been arbitrarily selected. Unlike Prompt 1, this prompt is designed by me, while implementing the naive prompt optimization approach. Sentence completion through GitHub Copilot based on the `GPT-4.1 Copilot` model has also been utilized.

With these two prompts, the naive prompt optimization can be evaluated. The results are quite consistent across different tested LLMs. The optimized prompt by OpenAI's `GPT-4o-2024-08-06` (GPT-4o) [26] model is presented as an example of these. The full results of this evaluation are included in the replication package [32].

6.4.1. Systematic Evaluation Results

The systematic evaluation results for varying models and datasets are presented in Table 6.3. As in Table 6.2 the average and weighted average are also included.

We can see in Table 6.3, that the GPT-4o and `GPT-4o-mini-2024-07-18` [26] LLMs managed to improve the F_1-Score scores for most datasets with both the simple and iterative optimization

Dataset	Metric	KISS-Original	ToT-Original-1	ToT-Original-2	ToT-Original-3
		GPT-4o-mini	GPT-4o-mini	GPT-4o-mini	GPT-4o-mini
CM1	P.	**0.341**	0.333	0.263	0.337
	R.	**0.644**	0.6	0.333	**0.644**
	F_1	**0.446**	0.429	0.294	0.443
Dron.	P.	0.386	0.351	0.385	**0.39**
	R.	**0.695**	0.427	0.682	0.691
	F_1	0.497	0.385	0.492	**0.498**
GANNT	P.	0.544	**0.547**	0.544	0.545
	R.	**0.544**	0.515	**0.544**	0.529
	F_1	**0.544**	0.53	**0.544**	0.537
MODIS	P.	0.212	**0.256**	0.157	0.18
	R.	**0.268**	**0.268**	**0.268**	**0.268**
	F_1	0.237	**0.262**	0.198	0.216
WARC	P.	0.38	**0.399**	0.381	0.381
	R.	**0.699**	0.596	0.691	0.684
	F_1	**0.492**	0.478	0.491	0.489
Avg.	P.	0.373	**0.377**	0.346	0.367
	R.	**0.57**	0.481	0.504	0.563
	F_1	**0.443**	0.417	0.404	0.437
W. Avg.	P.	**0.388**	0.381	0.376	0.387
	R.	**0.637**	0.486	0.602	0.629
	F_1	**0.477**	0.423	0.458	0.473

Table 6.2.: ToT promting to compare performance with the KISS Prompt 1.

Dataset	Metric	KISS-Original			simple (i =1)			iterative (i =5)		
		GPT-4o	GPT-4o-mini	Llama 3.1	GPT-4o	GPT-4o-mini	Llama 3.1	GPT-4o	GPT-4o-mini	Llama 3.1
CM1	P.	0.349	0.341	0.33	0.358	0.341	0.33	0.367	0.349	**1.0**
	R.	**0.644**	**0.644**	**0.644**	**0.644**	**0.644**	**0.644**	**0.644**	**0.644**	0.0
	F_1	0.453	0.446	0.436	0.46	0.446	0.436	**0.468**	0.453	0.0
Dron.	P.	0.394	0.386	0.382	0.413	0.389	0.386	0.414	0.405	**1.0**
	R.	**0.695**	**0.695**	0.682	0.686	0.691	**0.695**	0.686	0.682	0.0
	F_1	0.503	0.497	0.489	0.515	0.498	0.497	**0.516**	0.508	0.0
GANNT	P.	0.552	0.544	0.561	0.578	0.544	0.544	0.578	0.569	**1.0**
	R.	**0.544**	**0.544**	**0.544**	**0.544**	**0.544**	**0.544**	**0.544**	**0.544**	0.0
	F_1	0.548	0.544	0.552	**0.561**	0.544	0.544	**0.561**	0.556	0.0
MODIS	P.	0.212	0.212	0.224	0.385	0.22	0.145	0.455	0.435	**1.0**
	R.	**0.268**	**0.268**	**0.268**	0.244	**0.268**	**0.268**	0.244	0.244	0.0
	F_1	0.237	0.237	0.244	0.299	0.242	0.188	**0.317**	0.312	0.0
WARC	P.	0.394	0.38	0.377	0.436	0.382	0.377	0.444	0.408	**1.0**
	R.	**0.699**	**0.699**	0.684	**0.699**	**0.699**	**0.699**	0.669	0.669	0.0
	F_1	0.504	0.492	0.486	**0.537**	0.494	0.49	0.534	0.507	0.0
Avg.	P.	0.38	0.373	0.375	0.434	0.375	0.356	0.452	0.433	**1.0**
	R.	**0.57**	**0.57**	0.564	0.563	0.569	**0.57**	0.557	0.557	0.0
	F_1	0.449	0.443	0.441	0.474	0.445	0.431	**0.479**	0.467	0.0
W. Avg.	P.	0.396	0.388	0.387	0.434	0.39	0.38	0.443	0.425	**1.0**
	R.	**0.637**	**0.637**	0.627	0.631	0.635	**0.637**	0.623	0.622	0.0
	F_1	0.483	0.477	0.472	0.505	0.478	0.471	**0.507**	0.494	0.0

Table 6.3.: Naive prompt optimization approach prompting Prompt 3 to the LLM to optimize the classification prompt Prompt 1.

Results of the naive prompt optimization approach from Section 4.2.1 without feedback. The KISS zero-shot prompt Prompt 1 is optimized using Prompt 3. The results are presented for both a single ($i = 1$) optimization step (»simple« []) and five ($i = 5$) optimization steps (»iterative« []).

approaches. Precision and recall are mostly consistent. The `llama3.1:8b` (llama) LLM failed to classify any trace links (TLs) with the five times optimized prompt, resulting in recall and F_1-Score of 0.0 across all datasets. This is because the output format was altered during optimization, thus failing in the LiSSA framework. The prompt is included in the appendix as Prompt 18.

6.4.2. Optimized Prompt Analysis

Prompt 4: KISS Single Naive Optimization Step
Question: You are given two parts of software development artifacts that need to be analyzed for traceability link recovery in the domain of requirement to requirement.

{source_type}: '''{source_content}'''

{target_type}: '''{target_content}'''

Carefully examine the content and context of both artifacts. Are they related in terms of requirements?

Answer with 'yes' or 'no'.

To examine the optimized prompts, Prompt 4 shows the result of a single ($maximumiterationsteps = 1$) iterative optimization run with GPT-4o. The first application of the optimization prompt yielded an inclusion of the source and target types into the text of the prompt. This is possible as a set of training data was also provided for the following, more sophisticated, prompt optimization approaches. Further, the very simple classification question »Are they related?« [Prompt 1] was also expanded to be more specific for the traceability link recovery problem. The optimized prompt only depends on the input prompt and the domain of the source and target elements. This very prompt is thus used for all simple GPT-4o evaluations in Table 6.3 for the task of requiremens-to-requirements.

As seen in Table 6.3, application of the optimization prompt in the column »simple« [] seems to generally improve the F_1-Score. By repeatedly applying the optimization prompt to the optimized prompt of the previous iteration, we can push this optimization further. This yields Prompt 5 This particular prompt is generated by GPT-4o using five ($maximumiterationsteps = 5$) iteration steps. We can see that mostly longer and more detailed instructions are included on how to find TLs.

Figure 6.1.: Configurable parameters used in the automatic prompt engineering process.

$$\begin{aligned} i &:= \text{maximum iteration steps} \\ n &:= \text{feedback sample size} \end{aligned} \tag{6.4}$$

Prompt 5: KISS Iterative Naive Optimization GPT-4o
Question: You are given two parts of software development artifacts that need to be analyzed for traceability link recovery in the domain of requirement to requirement.

{source_type}: '''{source_content}'''

{target_type}: '''{target_content}'''

Carefully examine the content and context of both artifacts. Consider factors such as terminology, objectives, dependencies, and any implicit or explicit connections. Assess whether they share common goals, constraints, or functionalities. Pay close attention to shared terminology, aligned objectives, or mutual dependencies. Determine if they are related in terms of requirements by evaluating the presence of these elements.

Answer with 'yes' or 'no'.

However, unfortunately, this prompt does not necessarily perform better for our task. On the WARC [22] dataset, F_1-Score performance degraded slightly over Prompt 4 as we have seen in Table 6.3. What is especially noteworthy is that for llama, the classification fails with Prompt 18 As this shortcoming is expected, it is addressed by Section 4.2.2

6.5. Simple Feedback Optimization

The naive automatic prompt engineering process with feedback, as described in Section 4.2.1, utilizes more degrees of freedom than the naive approach in Section 6.4. To describe the different configurations used in this evaluation section, the parameters in Equation 6.4 are used. The prompt is optimized i times. n examples of misclassified trace links (TLs) are provided.

In addition to the optimization prompt Prompt 4.2.1 used in **??**, an extended version is used here. Prompt 6 shows the additional details included in this prompt, which is added to the optimization prompt. This provides the large language model (LLM) with more context on where the prompt failed to classify correctly. The misclassified TLs are be provided as additional context. The LLM can then use this information to improve the prompt. Up to feedback sample size misclassified TLs are provided. The optimization process only utilizes a subset of the dataset, as explained in Section 4.2.1. This training subset includes n source

artifacts with four target artifact candidates each, resulting in up to twelve possible TLs. So fewer than feedback sample size misclassified TLs may be available. The parameter maximum iteration steps will be used to limit the number of optimization iterations. In this evaluation section, maximum iteration steps was always fully utilized, as the required F_1-Score threshold of 1.0 for the training subset was never reached.

Prompt 6: Feedback Prompt
The current prompt is not performing well in classifying the following trace links. To help you improve the prompt, I will provide examples of misclassified trace links. Please analyze these examples and adjust the prompt accordingly. The examples are as follows:
{identifier}
{source_type}: '''{source_content}'''
{target_type}: '''{target_content}'''
Classification result: {classification}

6.5.1. Systematic Evaluation Results

The systematic evaluation results for varying models, datasets, and parameters are presented in Table 6.4. Like in Section 6.4.1, the average and weighted average are also included. The full optimization results are included in the replication package [32] for this thesis.

Dataset	Metric	KISS-Original			feedback (i =1, n =3)			feedback (i =5, n =3)			feedback (i =10, n =5)		
		GPT-4o	GPT-4o-mini	Llama 3.1	GPT-4o	GPT-4o-mini	Llama 3.1	GPT-4o	GPT-4o-mini	Llama 3.1	GPT-4o	GPT-4o-mini	Llama 3.1
CM1	P.	0.349	0.341	0.33	0.408	0.382	0.33	0.394	0.364	0.33	0.349	**0.5**	0.333
	R.	**0.644**	**0.644**	**0.644**	**0.644**	0.578	**0.644**	0.578	0.267	**0.644**	**0.644**	0.244	**0.644**
	F_1	0.453	0.446	0.436	**0.5**	0.46	0.436	0.468	0.308	0.436	0.453	0.328	0.439
Dron.	P.	0.394	0.386	0.382	**0.549**	0.386	0.342	0.45	0.387	-	0.405	0.413	0.385
	R.	**0.695**	**0.695**	0.682	0.668	**0.695**	0.3	0.691	0.691	-	0.691	0.659	0.691
	F_1	0.503	0.497	0.489	**0.602**	0.497	0.32	0.545	0.496	-	0.511	0.508	0.494
GANNT	P.	0.552	0.544	0.561	0.544	0.578	0.544	0.544	**0.617**	0.544	0.544	0.541	0.544
	R.	**0.544**	**0.544**	**0.544**	**0.544**	**0.544**	**0.544**	**0.544**	**0.544**	**0.544**	**0.544**	0.485	**0.544**
	F_1	0.548	0.544	0.552	0.544	0.561	0.544	0.544	**0.578**	0.544	0.544	0.512	0.544
MODIS	P.	0.212	0.212	0.224	0.391	0.289	0.312	**0.667**	0.346	0.455	0.5	0.234	0.244
	R.	**0.268**	**0.268**	**0.268**	0.22	**0.268**	0.244	0.146	0.22	0.122	**0.268**	**0.268**	0.244
	F_1	0.237	0.237	0.244	0.281	0.278	0.274	0.24	0.269	0.192	**0.349**	0.25	0.244
WARC	P.	0.394	0.38	0.377	0.388	0.382	0.379	0.397	0.397	0.383	0.399	**0.462**	0.378
	R.	**0.699**	**0.699**	0.684	**0.699**	0.691	0.691	**0.699**	0.691	**0.699**	**0.699**	0.676	0.669
	F_1	0.504	0.492	0.486	0.499	0.492	0.49	0.507	0.504	0.495	0.508	**0.549**	0.483
Avg.	P.	0.38	0.373	0.375	0.456	0.403	0.381	**0.49**	0.422	0.428	0.439	0.43	0.377
	R.	**0.57**	**0.57**	0.564	0.555	0.555	0.485	0.532	0.483	0.502	0.569	0.466	0.558
	F_1	0.449	0.443	0.441	**0.485**	0.458	0.413	0.461	0.431	0.417	0.473	0.429	0.441
W. Avg.	P.	0.396	0.388	0.387	**0.48**	0.402	0.375	0.461	0.415	0.423	0.425	0.436	0.388
	R.	**0.637**	**0.637**	0.627	0.622	0.629	0.463	0.62	0.596	0.573	0.635	0.572	0.625
	F_1	0.483	0.477	0.472	**0.532**	0.483	0.402	0.503	0.474	0.454	0.496	0.483	0.473

Table 6.4.: Naive iterative prompt optimization approach considering previous misclassified TLs

They are visualized in Table 6.4

6.5.2. Optimized Prompt Analysis

As the model now possesses additional knowledge of the dataset through the misclassified TLs, we can see that the optimized prompts for each dataset start to diverge. Compared to the naive optimization approach in Section 6.4, the optimized prompts are now more tailored to the specific dataset. To ease comparison, the results of `GPT-4o-2024-08-06` [26] for the WARC [22] dataset are presented here again. Prompt 7 shows the result of one optimization run with $feedbacksamplesize = 3$ and $maximumiterationsteps = 1$. Detailed instructions on how to classify TLs are generated by the LLM. This is quite common across different models and parameterization for the early iterations. This behavior can be observed with the full results in the replication package [32]. The prompt is also adapted to the requiremens-to-requirements (Req2Req) task as indirectly instructed through the optimization prompt.

Prompt 7: Optimized Prompt feedback sample size = 3, maximum iteration steps = 1
Question: Here are two parts of software development artifacts.

{source_type}: '''{source_content}'''

{target_type}: '''{target_content}'''

Consider the following when determining if they are related:
1. Look for common themes or concepts, such as specific file names, structures, or functionalities.
2. Identify if one requirement directly supports or implements the other.
3. Check if both requirements refer to the same component or feature, even if described differently.

Are they related?

Answer with 'yes' or 'no'.

However, later iteration steps start to include more overfitted instructions to the provided misclassified TLs. This is likely contributing to the overfitting of the prompt to the provided misclassified TLs. Prompt 14 in the appendix shows the full result of one optimization run with $n = 5$ and $maximumiterationsteps = 10$. The full optimized prompt can be found in the appendix. The LLM provided 30 indicators for classification. One of these is obviously fitted to the WARC dataset, explicitly mentioning it in the first indicator. A fitting to the Req2Req task is also present with indicators pointing to requirements directly. This can be seen in Prompt 8

Prompt 8: Shortend Optimized Prompt feedback sample size = 5, maximum iteration steps = 10
[...]
1. Look for shared terminology or concepts, such as specific file names, structures, or functionalities. Pay attention to terms like "single header file," "universal header," and specific file names like "warc.h."
[...]
8. Look for implicit relationships where one requirement might imply the necessity of the other, even if not explicitly stated.
[...]

6.6. Varying the Optimization Prompt

In a recent paper by Zadenoori et al. [40] the authors discussed the application of automatic prompt engineering (APE) for requirement classification. They took varying initial prompts to run the optimization process with Meta's open source `llama3.1:8b` model. They were able to improve F_1-Score and F_2-Score scores by 5 % and 9 % respectively to around 80 %. Their APE process is generally aligned with the general iterative optimization loop visualized in Figure 2.1.

Their prompt is designed to optimize classification prompts by enhancing the explanations of categories within the prompt. As it also utilizes feedback from misclassified trace links, it can be used as an alternative optimization prompt for the naive feedback optimization approach. They used the optimization prompt in Prompt 15. The prompt is tailored to their chain-of-thought classification prompt, comprising a five-step pipeline for requirement classification.

Prompt 9: Excerpt of Zadenoori et al.'s Optimization Prompt
[...]
the steps in the optimized prompt must remain exactly the same
[...]
ensure all content remains within the existing steps and does not extend beyond them
[...]

We can see that Zadenoori et al. used some similar elements to my optimization prompt. Just as I instructed the model, that the in and output format are not to be modified, to ensure correct classification with the used classifiers, they also included this in Prompt 9 Further, the second instruction Prompt 9 is intended to negate behavior such as in Prompt 14. However, they also gave the large language model (LLM) more specific instructions in how to optimize the prompt. Their classification tasks include more different outputs than just the simple yes and no of Prompt 1. The LLM is instructed to add details to these

classes. Prompt 13 illustrates what this can look like for our simple traceability link recovery requiremens-to-requirements (Req2Req) task. This full prompt is kept in the appendix. Here we will focus on the most important details shown in Prompt 10

As their optimization prompt is more detailed than the one used in Section 6.5, it is used as an alternative optimization prompt for the naive optimization approach. The initial prompt Prompt 1 in my work does not incorporate this multistep process. It needs to be adjusted to be used for Req2Req prompt optimization. The modified version of this prompt can be found in the appendix under Prompt 16 As seen in Table 6.5, the results of this optimization approach are quite similar to the ones of Section 6.4.

6.6.1. Optimized Prompt Analysis

An excerpt of the optimized prompt generated with the `GPT-4o-2024-08-06` [26] model for the WARC [22] dataset can be seen in Prompt 10. We can see that the LLM focused on adding explanations to the possible classification results. While this does include general instructions, overfitting is becoming quite obvious as well. Some sections from the prompt are specified for the WARC dataset, but might also find appliance elsewhere too. As such, F_1-Score scores during the APE process on the reduced training set started to diverge more from the actual performance in Table 6.5 than the previous prompt in Simple Feedback Optimization.

Prompt 10: Opimized WARC prompt feedback sample size = 3, maximum iteration steps = 5
[...]Class Explanations:

1. Yes: [...]
if both artifacts mention a single header file required for a software tool or application based on libwarc, they are related.
[...]
they all revolve around the concept of a single header file, "warc.h", which serves as a universal entry point
[...]
2. No: [...]
Use these examples to guide your classification decisions, ensuring that implicit connections are recognized and appropriately classified as 'yes'.

6.6.2. Systematic Evaluation Results

The systematic evaluation results for varying models, datasets, and parameters are presented in Table 6.5. Once again, the average and weighted average are also included. The results are quite similar to the naive feedback optimization approach in Table 6.4. The F_1-Score

scores are slightly higher on average. A prevalent issue remains that more iteration steps often do not yield better results.

Dataset	Metric	KISS-Original		feedback (i =1, n =3)		feedback (i =3, n =1)		feedback (i =5, n =3)	
		GPT-4o	GPT-4o-mini	GPT-4o	GPT-4o-mini	GPT-4o	GPT-4o-mini	GPT-4o	GPT-4o-mini
CM1	P.	0.349	0.341	**0.583**	0.389	0.392	0.341	0.571	0.386
	R.	**0.644**	**0.644**	0.467	0.622	**0.644**	**0.644**	0.444	0.6
	F_1	0.453	0.446	**0.519**	0.479	0.487	0.446	0.5	0.47
Dron.	P.	0.394	0.386	0.431	0.409	0.406	0.392	**0.449**	0.417
	R.	**0.695**	**0.695**	0.686	0.691	0.691	0.691	0.682	0.686
	F_1	0.503	0.497	0.53	0.514	0.512	0.5	**0.542**	0.519
GANNT	P.	0.552	0.544	**0.578**	0.554	0.561	0.561	**0.578**	0.565
	R.	**0.544**	**0.544**	**0.544**	0.529	**0.544**	**0.544**	**0.544**	0.515
	F_1	0.548	0.544	**0.561**	0.541	0.552	0.552	**0.561**	0.538
MODIS	P.	0.212	0.212	**0.37**	0.256	0.355	0.367	**0.37**	0.314
	R.	**0.268**	**0.268**	0.244	**0.268**	**0.268**	**0.268**	0.244	**0.268**
	F_1	0.237	0.237	0.294	0.262	0.306	**0.31**	0.294	0.289
WARC	P.	0.394	0.38	0.45	0.427	0.403	0.386	**0.485**	0.425
	R.	**0.699**	**0.699**	**0.699**	0.662	**0.699**	0.684	**0.699**	0.662
	F_1	0.504	0.492	0.548	0.519	0.511	0.493	**0.572**	0.517
Avg.	P.	0.38	0.373	0.482	0.407	0.423	0.409	**0.491**	0.421
	R.	**0.57**	**0.57**	0.528	0.554	0.569	0.566	0.523	0.546
	F_1	0.449	0.443	0.49	0.463	0.474	0.46	**0.494**	0.467
W. Avg.	P.	0.396	0.388	0.464	0.419	0.421	0.406	**0.48**	0.428
	R.	**0.637**	**0.637**	0.616	0.622	0.635	0.631	0.612	0.616
	F_1	0.483	0.477	0.519	0.496	0.498	0.485	**0.529**	0.498

Table 6.5.: Naive prompt optimization approach using the optimization prompt by Zadenoori et al. [40]

6.7. Gradient Descent Based Automatic Prompt Optimization

The last prompt optimization algorithm implemented during this thesis is based on work by Pryzant et al. [27]. They have implemented this algorithm in Python. I adapted it to the Linking Software System Artifacts (LiSSA) [10] framework for use in the optimization module. As this algorithm is the most sophisticated one, it is expected to yield the best results. The implementation has many configurable parameters. In this evaluation, not every parameter will be altered. Instead, the default values as by Pryzant et al.'s implementation will widely be used.

Like with previous evaluations, we will alter the number of iteration steps performed in the algorithm. We can see for several entries of Table 6.6, that the precision, recall and F_1-Score remains identical between two adjacent iteration steps. This indicates in most cases, that the identical optimized prompt was found. Since large language model requests are cached in the LiSSA framework this suggests, that the automatic prompt engineering (APE) algorithm was not able to improve the prompt in this iteration step. Table 6.6 does not visualize other information that is passed on between iteration steps though. Thus the APE algorithm was still able to alter the optimized output prompt further in later steps, instead of stalling with this candidate.

Prompt 11: Opimized WARC prompt maximum iteration steps = 3
Question: We have two software development artifacts that may or may not be related in terms of their purpose, functionality, or requirements.

Source Artifact {*source_type*}: '"{source_content}"'

Target Artifact {*target_type*}: '"{target_content}"'

Please analyze the content of both artifacts and determine if they are related. For the purpose of this analysis, consider the following criteria:
1. Purpose: Do both artifacts serve a similar goal or objective within the software development process?
2. Functionality: Do both artifacts provide similar features or capabilities?
3. Requirements: Do both artifacts address the same requirements or constraints?

Based on these criteria, answer with 'yes' if they are related, or 'no' if they are not related.

To examine one of the optimized prompts closer we can take a look at Prompt 11. It is also the prompt with the single highest F_1-Score in Table 6.6. We can note, that compared to optimized prompts of the previous Simple Feedback Optimization with Prompt 14 for example, the optimized prompt is significantly shorter. Furthermore, the prompt does not show signs of overfitting for features unique to the WARC [22] dataset. The input and output formats of the prompt are left intact, as is required for prompts to perform well within the constraints of the LiSSA framework. The major changes over Prompt 1 are simply the

Dataset	Metric	KISS-Original	gradient (i =2)	gradient (i =3)	gradient (i =4)	gradient (i =5)
		GPT-4o-mini	GPT-4o-mini	GPT-4o-mini	GPT-4o-mini	GPT-4o-mini
CM1	P.	**0.341**	0.333	0.333	0.33	0.333
	R.	**0.644**	**0.644**	**0.644**	**0.644**	**0.644**
	F_1	**0.446**	0.439	0.439	0.436	0.439
Dron.	P.	0.386	0.404	0.4	**0.476**	0.368
	R.	**0.695**	0.691	0.691	0.445	0.255
	F_1	0.497	**0.51**	0.507	0.46	0.301
GANNT	P.	0.544	0.561	0.561	**0.667**	**0.667**
	R.	**0.544**	**0.544**	**0.544**	0.5	0.5
	F_1	0.544	0.552	0.552	**0.571**	**0.571**
MODIS	P.	0.193	0.204	0.204	**0.368**	0.257
	R.	**0.268**	**0.268**	**0.268**	0.171	0.22
	F_1	0.224	0.232	0.232	0.233	**0.237**
WARC	P.	0.38	0.392	0.528	0.528	**0.541**
	R.	**0.699**	0.691	0.632	0.632	0.588
	F_1	0.492	0.5	**0.575**	**0.575**	0.563
Avg.	P.	0.369	0.379	0.405	**0.474**	0.433
	R.	**0.57**	0.568	0.556	0.478	0.441
	F_1	0.441	0.447	**0.461**	0.455	0.422
W. Avg.	P.	0.386	0.399	0.434	**0.494**	0.442
	R.	**0.637**	0.633	0.618	0.498	0.408
	F_1	0.475	0.484	**0.503**	0.485	0.414

Table 6.6.: Gradient Descent based Automatic Prompt Optimization

explanatory text describing when artifacts should have a trace link (TL) existing between them. Indications of this prompt being fitted to requiremens-to-requirements datasets are present. The optimized prompt lists three criteria for linked artifacts.

1. purpose
2. functionality
3. requirements

While the purpose is very general applicable, the functionality already requires domains illustrating features or capabilities. They might not be present in high level artifacts speci-

fying the project outline, such as architectural diagrams. Last but not least requirements require specific knowledge of the project. Evaluating TLs between documentation and code might hide many requirements or only implicitly mention them.

6.8. Application of an Optimized Prompt to Unfamiliar Datasets

automatic prompt engineering with the proposed algorithms in my work is always connected to additional costs in execution time and possibly budget for large language model tokens. Thus, the optimized prompts should ideally be generally applicable, eliminating the need for repeated optimization. To test this, the single best-performing prompt in terms of F_1-Score will be used to classify trace links in the other datasets.

Table 6.7 illustrates the classification results with the previous most promising Prompt 11. In addition to the `GPT-4o-mini-2024-07-18` [26] model, used to optimize the prompt previously, it is also tested for `GPT-4o-2024-08-06` [26] and `GPT-5-mini-2025-08-07` [11]. We can see that the F_1-Score score is improved across all datasets and models. This suggests that Prompt 11 performs better in this regard than the default KISS Prompt 1 of Linking Software System Artifacts [10]. Furthermore, the precision is generally improved as well. On the other hand, recall rates are usually lower than for the unoptimized counterpart.

Dataset	Metric	KISS-Original			WARC-optimized		
		GPT-4o	GPT-4o-mini	GPT-5-mini	GPT-4o	GPT-4o-mini	GPT-5-mini
CM1	P.	0.349	0.341	0.358	0.414	**0.415**	0.384
	R.	**0.644**	**0.644**	**0.644**	**0.644**	0.6	0.622
	F_1	0.453	0.446	0.46	**0.504**	0.491	0.475
Dron.	P.	0.394	0.386	0.393	0.426	**0.48**	0.416
	R.	**0.695**	**0.695**	**0.695**	0.691	0.668	**0.695**
	F_1	0.503	0.497	0.502	0.527	**0.559**	0.52
GANNT	P.	0.552	0.544	0.544	0.578	**0.627**	0.561
	R.	**0.544**	**0.544**	**0.544**	**0.544**	**0.544**	**0.544**
	F_1	0.548	0.544	0.544	0.561	**0.583**	0.552
MODIS	P.	0.212	0.212	0.312	0.476	**0.625**	0.455
	R.	**0.268**	**0.268**	0.244	0.244	0.244	0.244
	F_1	0.237	0.237	0.274	0.323	**0.351**	0.317
WARC	P.	0.394	0.38	0.394	0.464	**0.518**	0.442
	R.	**0.699**	**0.699**	**0.699**	0.669	0.647	**0.699**
	F_1	0.504	0.492	0.504	0.548	**0.575**	0.541
Avg.	P.	0.38	0.373	0.4	0.472	**0.533**	0.452
	R.	**0.57**	**0.57**	0.565	0.558	0.541	0.561
	F_1	0.449	0.443	0.457	0.493	**0.512**	0.481
W. Avg.	P.	0.396	0.388	0.404	0.459	**0.516**	0.443
	R.	**0.637**	**0.637**	0.635	0.625	0.606	0.633
	F_1	0.483	0.477	0.486	0.519	**0.544**	0.51

Table 6.7.: Classification performance of an Prompt Optimization with Textual Gradients optimized prompt for WARC by `GPT-4o-mini-2024-07-18` across other datasets

7. Conclusion

This thesis presents an automatic prompt engineering (APE) approach using large language models (LLMs) to optimize prompts for requiremens-to-requirements (Req2Req) traceability link recovery (TLR) tasks in the Linking Software System Artifacts (LiSSA) [10] framework. To realize this, a new prompt optimization component is realized for the LiSSA framework. Among other strategies for APE, the gradient descent-based algorithm Prompt Optimization with Textual Gradients (ProTeGi) [27] is implemented. Utilizing the added `Prompt Optimizer` component, prompts can be refined iteratively to provide better classification results on a training set. First tree-of-thought prompts were evaluated for the Req2Req task. They did not indicate better performance than the existing zero-shot or chain-of-thought prompts. In addition, the naive strategies of prompting the LLM to optimize the prompt with or without examples of misclassified trace links are also presented. The approach is evaluated on five datasets from the Req2Req task using four LLMs. The results show that the realized approach can improve the performance of TLR tasks compared to the baseline classification prompts currently used in the LiSSA framework. At the same time, the absolute difference between the different APE algorithms TLR performances is quite small. Importantly, it is demonstrated that the best-performing optimized prompt in terms of F_1-Score, Prompt 6.7 for the WARC [22] dataset also performed well on other datasets of the same task. This indicates that the optimized prompts may be used as fixed classification prompts in further work or projects, even if the APE pipeline is not applied. Prompt 6.7 was optimized with the ProTeGi implementation of the APE component with the zero-shot keep it short and simple Prompt 1 as initial prompt.

Further work might include applying the APE component to different domains and tasks. Establishing generalized optimized prompts with improved performance on multiple domains is desirable, as LiSSA is not restricted to the Req2Req task. The gradient descent-based APE algorithm still has many more parameters that have been introduced during development, but have not been systematically altered during evaluation. Exploring configuration variations, for instance, utilizing greater optimization budgets, is another possibility to build on this work. Stepping up from TLR, the optimization prompts used for optimization in the `Prompt Optimizer` component might also be optimized themselves.

As the LLMs output is not deterministic, the cached requests and responses are included in the replication package [32] for this thesis. Alongside the cached requests, the used datasets and code are also publicly available in this replication package as well. Variances in performance are expected when rerunning the requests instead of using the cached values.

Bibliography

[1] Jean-Yves Audibert and Sébastien Bubeck. "Best Arm Identification in Multi-Armed Bandits". In: COLT - 23th Conference on Learning Theory - 2010. June 27, 2010, 13 p. URL: https://enpc.hal.science/hal-00654404 (visited on 05/20/2025).

[2] *Beam Search from FOLDOC*. URL: https://foldoc.org/beam+search (visited on 06/12/2025).

[3] Jane Cleland-Huang, Michael Vierhauser, and Sean Bayley. "Dronology: An Incubator for Cyber-Physical Systems Research". In: *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*. ICSE '18: 40th International Conference on Software Engineering. Gothenburg Sweden: ACM, May 27, 2018, pp. 109–112. ISBN: 978-1-4503-5662-6. DOI: 10.1145/3183399.3183408. URL: https://dl.acm.org/doi/10.1145/3183399.3183408 (visited on 10/14/2025).

[4] Jane Cleland-Huang et al. "Best Practices for Automated Traceability". In: *Computer* 40.6 (June 2007), pp. 27–35. ISSN: 1558-0814. DOI: 10.1109/MC.2007.195. URL: https://ieeexplore.ieee.org/document/4249808/ (visited on 10/15/2025).

[5] *Datasets*. URL: https://web.archive.org/web/20250119010949/http://sarec.nd.edu/coest/datasets.html (visited on 10/17/2025).

[6] Andrea De Lucia et al. "Information Retrieval Methods for Automated Traceability Recovery". In: *Software and Systems Traceability*. Ed. by Jane Cleland-Huang, Orlena Gotel, and Andrea Zisman. London: Springer, 2012, pp. 71–98. ISBN: 978-1-4471-2239-5. DOI: 10.1007/978-1-4471-2239-5_4. URL: https://doi.org/10.1007/978-1-4471-2239-5_4 (visited on 06/21/2025).

[7] Niklas Ewald. *Retrieval-Augmented Large Language Models for Traceability Link Recovery*. 2024. DOI: 10.5445/IR/1000178218. URL: https://publikationen.bibliothek.kit.edu/1000178218 (visited on 05/20/2025).

[8] Dominik Fuchß et al. *ArDoCo/Benchmark*. Version ecsa22-msr4sa. Zenodo, Aug. 5, 2022. DOI: 10.5281/zenodo.6966832. URL: https://zenodo.org/records/6966832 (visited on 05/20/2025).

[9] Dominik Fuchß et al. "Beyond Retrieval: A Study of Using LLM Ensembles for Candidate Filtering in Requirements Traceability". In: *2025 IEEE 33rd International Requirements Engineering Conference Workshops (RE)*. 33rd IEEE International Requirements Engineering conference (RE 2025), Valencia, Spanien, 01.09.2025 – 05.09.2025. Sept. 6, 2025. URL: https://publikationen.bibliothek.kit.edu/1000183058 (visited on 10/13/2025).

[10] Dominik Fuchß et al. “LiSSA: Toward Generic Traceability Link Recovery through Retrieval-Augmented Generation”. In: *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. International Conference on Software Engineering (ICSE 2025), Ottawa, Kanada, 27.04.2025 – 03.05.2025. 2025, p. 723. DOI: 10.1109/ICSE55347.2025.00186. URL: https://publikationen.bibliothek.kit.edu/1000179816 (visited on 05/20/2025).

[11] *GPT-5 System Card*. Oct. 9, 2025. URL: https://openai.com/index/gpt-5-system-card/ (visited on 10/14/2025).

[12] J.H. Hayes, A. Dekhtyar, and J. Osborne. “Improving Requirements Tracing via Information Retrieval”. In: *Proceedings. 11th IEEE International Requirements Engineering Conference, 2003.* . 11th IEEE International Requirements Engineering Conference, 2003. Sept. 2003, pp. 138–147. DOI: 10.1109/ICRE.2003.1232745. URL: https://ieeexplore.ieee.org/abstract/document/1232745 (visited on 10/14/2025).

[13] J.H. Hayes, A. Dekhtyar, and S.K. Sundaram. “Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods”. In: *IEEE Transactions on Software Engineering* 32.1 (Jan. 2006), pp. 4–19. ISSN: 1939-3520. DOI: 10.1109/TSE.2006.3. URL: https://ieeexplore.ieee.org/abstract/document/1583599 (visited on 05/26/2025).

[14] Tobias Hey et al. “Improving Traceability Link Recovery Using Fine-grained Requirements-to-Code Relations”. In: *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME). Sept. 2021, pp. 12–22. DOI: 10.1109/ICSME52107.2021.00008. URL: https://ieeexplore.ieee.org/abstract/document/9609109 (visited on 06/14/2025).

[15] Tobias Hey et al. *Replication Package for "Requirements Traceability Link Recovery via Retrieval-Augmented Generation"*. Zenodo, Jan. 31, 2025. DOI: 10.5281/zenodo.14779458. URL: https://zenodo.org/records/14779458 (visited on 06/15/2025).

[16] Tobias Hey et al. “Requirements Traceability Link Recovery via Retrieval-Augmented Generation”. In: *Requirements Engineering: Foundation for Software Quality*. Ed. by Anne Hess and Angelo Susi. Cham: Springer Nature Switzerland, 2025, pp. 381–397. ISBN: 978-3-031-88531-0. DOI: 10.1007/978-3-031-88531-0_27.

[17] E. Ashlee Holbrook, Jane Huffman Hayes, and Alex Dekhtyar. “Toward Automating Requirements Satisfaction Assessment”. In: *2009 17th IEEE International Requirements Engineering Conference*. 2009 17th IEEE International Requirements Engineering Conference. Aug. 2009, pp. 149–158. DOI: 10.1109/RE.2009.10. URL: https://ieeexplore.ieee.org/abstract/document/5328545 (visited on 10/14/2025).

[18] Dave Hulbert. *Using Tree-of-Thought Prompting to Boost ChatGPT’s Reasoning*. Version 0.1. May 2023. URL: https://github.com/dave1010/tree-of-thought-prompting (visited on 06/02/2025).

[19] Haolin Jin et al. *From LLMs to LLM-based Agents for Software Engineering: A Survey of Current, Challenges and Future*. Apr. 13, 2025. DOI: 10.48550/arXiv.2408.02479. arXiv: 2408.02479 [cs]. URL: http://arxiv.org/abs/2408.02479 (visited on 10/14/2025). Pre-published.

[20] Jan Keim et al. "Detecting Inconsistencies in Software Architecture Documentation Using Traceability Link Recovery". In: *2023 IEEE 20th International Conference on Software Architecture (ICSA)*. 2023 IEEE 20th International Conference on Software Architecture (ICSA). Mar. 2023, pp. 141–152. DOI: 10.1109/ICSA56044.2023.00021. URL: https://ieeexplore.ieee.org/abstract/document/10092702 (visited on 10/13/2025).

[21] Omar Khattab et al. "DSPy: Compiling Declarative Language Model Calls into State-of-the-Art Pipelines". In: The Twelfth International Conference on Learning Representations. Oct. 13, 2023. URL: https://openreview.net/forum?id=sY5N0zY5Od (visited on 06/15/2025).

[22] Wei-Keat Kong et al. "Process Improvement for Traceability: A Study of Human Fallibility". In: *2012 20th IEEE International Requirements Engineering Conference (RE)*. 2012 20th IEEE International Requirements Engineering Conference (RE). Sept. 2012, pp. 31–40. DOI: 10.1109/RE.2012.6345824. URL: https://ieeexplore.ieee.org/abstract/document/6345824 (visited on 10/14/2025).

[23] Volodymyr Kuleshov and Doina Precup. *Algorithms for Multi-Armed Bandit Problems*. Feb. 25, 2014. DOI: 10.48550/arXiv.1402.6028. arXiv: 1402.6028 [cs]. URL: http://arxiv.org/abs/1402.6028 (visited on 06/12/2025). Pre-published.

[24] Jieyi Long. *Large Language Model Guided Tree-of-Thought*. May 15, 2023. DOI: 10.48550/arXiv.2305.08291. arXiv: 2305.08291 [cs]. URL: http://arxiv.org/abs/2305.08291 (visited on 06/02/2025). Pre-published.

[25] Aman Madaan et al. "Self-Refine: Iterative Refinement with Self-Feedback". In: *Advances in Neural Information Processing Systems* 36 (Dec. 15, 2023), pp. 46534–46594. URL: https://proceedings.neurips.cc/paper_files/paper/2023/hash/91edff07232fb1b55a505a9e9f6c0ff3-Abstract-Conference.html (visited on 05/20/2025).

[26] OpenAI et al. *GPT-4o System Card*. Oct. 25, 2024. DOI: 10.48550/arXiv.2410.21276. arXiv: 2410.21276 [cs]. URL: http://arxiv.org/abs/2410.21276 (visited on 06/21/2025). Pre-published.

[27] Reid Pryzant et al. "Automatic Prompt Optimization with "Gradient Descent" and Beam Search". In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. EMNLP 2023. Ed. by Houda Bouamor, Juan Pino, and Kalika Bali. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 7957–7968. DOI: 10.18653/v1/2023.emnlp-main.494. URL: https://aclanthology.org/2023.emnlp-main.494/ (visited on 06/02/2025).

[28] Kiran Ramnath et al. *A Systematic Survey of Automatic Prompt Optimization Techniques*. Apr. 2, 2025. DOI: 10.48550/arXiv.2502.16923. arXiv: 2502.16923 [cs]. URL: http://arxiv.org/abs/2502.16923 (visited on 06/14/2025). Pre-published.

[29] *Recovery*. Oct. 8, 2025. URL: https://dictionary.cambridge.org/dictionary/english/recovery (visited on 10/15/2025).

[30] 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 IEEE 31st International Requirements Engineering Conference Workshops (REW). Sept. 2023, pp. 455–464. DOI: 10.1109/REW57809.2023.00087. URL: https://ieeexplore.ieee.org/abstract/document/10260721 (visited on 05/13/2025).

[31] Sarah Santos et al. "Requirements Satisfiability with In-Context Learning". In: *2024 IEEE 32nd International Requirements Engineering Conference (RE)*. 2024 IEEE 32nd International Requirements Engineering Conference (RE). June 2024, pp. 168–179. DOI: 10.1109/RE59067.2024.00025. URL: https://ieeexplore.ieee.org/abstract/document/10628468 (visited on 10/17/2025).

[32] Daniel Schwab. *DanielDango/Replication-Package-Automated-Prompt-Engineering-for-Traceability-Link-Recovery: Replication Package*. Zenodo, Nov. 7, 2025. DOI: 10.5281/zenodo.17552402. URL: https://zenodo.org/records/17552402 (visited on 11/08/2025).

[33] Parshin Shojaee et al. *The Illusion of Thinking: Understanding the Strengths and Limitations of Reasoning Models via the Lens of Problem Complexity*. June 1, 2025. DOI: 10.48550/arXiv.2506.06941. URL: https://ui.adsabs.harvard.edu/abs/2025arXiv250606941S (visited on 06/15/2025). Pre-published.

[34] *Traceability*. June 18, 2025. URL: https://dictionary.cambridge.org/dictionary/english/traceability (visited on 06/18/2025).

[35] Liang Wang et al. *Improving Text Embeddings with Large Language Models*. May 31, 2024. DOI: 10.48550/arXiv.2401.00368. arXiv: 2401.00368 [cs]. URL: http://arxiv.org/abs/2401.00368 (visited on 10/15/2025). Pre-published.

[36] Xuezhi Wang et al. *Self-Consistency Improves Chain of Thought Reasoning in Language Models*. Mar. 7, 2023. DOI: 10.48550/arXiv.2203.11171. arXiv: 2203.11171 [cs]. URL: http://arxiv.org/abs/2203.11171 (visited on 05/20/2025). Pre-published.

[37] Rebekka Wohlrab et al. "Improving the Consistency and Usefulness of Architecture Descriptions: Guidelines for Architects". In: *2019 IEEE International Conference on Software Architecture (ICSA)*. 2019 IEEE International Conference on Software Architecture (ICSA). Mar. 2019, pp. 151–160. DOI: 10.1109/ICSA.2019.00024. URL: https://ieeexplore.ieee.org/abstract/document/8703919 (visited on 05/26/2025).

[38] Chengrun Yang et al. *Large Language Models as Optimizers*. Apr. 15, 2024. DOI: 10.48550/arXiv.2309.03409. arXiv: 2309.03409 [cs]. URL: http://arxiv.org/abs/2309.03409 (visited on 05/20/2025). Pre-published.

[39] Shunyu Yao et al. *Tree of Thoughts: Deliberate Problem Solving with Large Language Models*. Dec. 3, 2023. DOI: 10.48550/arXiv.2305.10601. arXiv: 2305.10601 [cs]. URL: http://arxiv.org/abs/2305.10601 (visited on 06/17/2025). Pre-published.

[40] Mohammad Amin Zadenoori et al. “Automatic Prompt Engineering: The Case of Requirements Classification”. In: *Requirements Engineering: Foundation for Software Quality*. Ed. by Anne Hess and Angelo Susi. Cham: Springer Nature Switzerland, Apr. 1, 2025, pp. 217–225. ISBN: 978-3-031-88531-0. DOI: 10.1007/978-3-031-88531-0_15.

[41] Jianzhang Zhang et al. *Enhancing Requirement Traceability through Data Augmentation Using Large Language Models*. Sept. 24, 2025. DOI: 10.48550/arXiv.2509.20149. arXiv: 2509.20149 [cs]. URL: http://arxiv.org/abs/2509.20149 (visited on 10/14/2025). Pre-published.

[42] Wayne Xin Zhao et al. *A Survey of Large Language Models*. Version 16. 2023. DOI: 10.48550/ARXIV.2303.18223. URL: https://arxiv.org/abs/2303.18223 (visited on 10/15/2025). Pre-published.

[43] Yongchao Zhou et al. *Large Language Models Are Human-Level Prompt Engineers*. Mar. 10, 2023. DOI: 10.48550/arXiv.2211.01910. arXiv: 2211.01910 [cs]. URL: http://arxiv.org/abs/2211.01910 (visited on 05/13/2025). Pre-published.

A. Appendix

A.1. Prompts

Prompt 12: Original Chain-of-thought
Below are two artifacts from the same software system. Is there a traceability link between (1) and (2)?
Give your reasoning and then answer with 'yes' or 'no' enclosed in <trace> </trace>.
(1) {source_type}: '"{source_content}"'
(2) {target_type}: '"{target_content}"'

Prompt 13: Opimized WARC prompt feedback sample size = 3, maximum iteration steps = 5
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'.

Class Explanations:

1. Yes: The two artifacts are related if they refer to the same concept, functionality, or requirement. For example, if both artifacts mention a single header file required for a software tool or application based on libwarc, they are related. This includes scenarios where one artifact describes the necessity of a single entry point, and the other explains the structure or functionality enabled by this single header file. For instance, if a requirement states that a single header file is needed, and another requirement specifies that this header file includes all necessary components, they are related. Similarly, if a requirement mentions a single entry point and another requirement discusses the hierarchical structure of headers that the entry point encompasses, they are related. Additionally, if a requirement specifies the ability to manipulate aspects of a file format through a single header file, and another requirement confirms this capability, they are related. For example, if one requirement states that a single header file allows access to all aspects of a file format, and another requirement confirms that this header file includes all necessary components, they are related. In the provided examples, the artifacts are related because they all revolve around the concept of a single header file, "warc.h", which serves as a universal entry point and includes all necessary components for accessing and manipulating the WARC file format.

2. No: The two artifacts are not related if they discuss different concepts, functionalities, or requirements. For instance, if one artifact talks about a single header file requirement and the other discusses a different aspect of the software that does not directly relate to the single header file, they are not related. However, if the second artifact implicitly supports or expands on the concept of a single header file, such as by detailing its structure or capabilities, they should be considered related. For example, if a requirement discusses a single header file and another requirement talks about a completely unrelated feature or component of the software, they are not related. If a requirement specifies a single header file and another requirement discusses a different feature that does not enhance or relate to the header file's purpose, they are not related. In the provided examples, the artifacts were misclassified as 'No' because they actually support the concept of a single header file by detailing its structure and capabilities, thus they should be classified as 'Yes'.

Use these examples to guide your classification decisions, ensuring that implicit connections are recognized and appropriately classified as 'yes'.

Prompt 14: Optimized Prompt feedback sample size = 5, maximum iteration steps = 10

Question: Here are two parts of software development artifacts.

{source_type}: '''{source_content}'''

{target_type}: '''{target_content}'''

Consider the following when determining if they are related:
1. Look for shared terminology or concepts, such as specific file names, structures, or functionalities. Pay attention to terms like "single header file," "universal header," and specific file names like "warc.h."
2. Identify if one requirement directly supports, implements, or complements the functionality described in the other. Consider if one requirement describes a feature that is necessary for the implementation of the other.
3. Determine if there is a logical connection, dependency, or enhancement between the two requirements. Consider if one requirement ensures compatibility, access, or encapsulation that is relevant to the other.
4. Evaluate if the requirements address the same aspect of the system, such as compatibility, accessibility, or structure, even if they use different wording.
5. Consider the intent and purpose behind each requirement. Determine if they share a common goal or outcome, such as simplifying integration, ensuring compatibility, or providing comprehensive access.
6. Assess if the requirements are part of a sequence or workflow that contributes to a larger functionality or system behavior, indicating a traceability link.
7. Pay attention to the context and implications of each requirement. Consider if they are part of a broader architectural or design principle that connects them.
8. Look for implicit relationships where one requirement might imply the necessity of the other, even if not explicitly stated.
9. Consider the broader impact of each requirement on the system and whether they collectively contribute to a cohesive design or functionality.
10. Examine if the requirements are addressing the same problem or solution space, even if they approach it from different angles or levels of detail.
11. Analyze if the requirements are interdependent, where the fulfillment of one is contingent upon the other, indicating a strong traceability link.
12. Consider if the requirements are describing different aspects of the same feature or functionality, which might indicate a traceability link.
13. Look for any mention of ensuring or maintaining a specific feature or functionality that is described in the other requirement.
14. Evaluate if the requirements are part of a common theme or objective, such as improving usability, performance, or maintainability, which might suggest a relationship.
15. Consider if the requirements are addressing the same technical constraints or design considerations, which could indicate a relationship.
16. Look for any shared assumptions or preconditions that might link the requirements together.
17. Determine if the requirements are part of a modular design where one module's functionality is dependent on or enhanced by the other.
18. Consider if the requirements are part of a layered architecture where one layer's requirements are inherently linked to another layer's requirements.
19. Pay special attention to requirements that mention compatibility, encapsulation, or abstraction, as these often indicate a relationship with other requirements

Prompt 15: Zadenoori et al.'s Optimization Prompt
You are required to enhance and clarify the explanations of the categories in the prompt by integrating illustrative examples and information implicitly referenced in the initial context.
The optimized prompt must follow these strict guidelines:
Maintain the Original Steps: The steps in the optimized prompt must remain exactly the same as in the sample prompt; no changes should be made to the steps' structure or order.
Expand Explanations: Enrich and expand the explanations of each category within the steps, incorporating examples provided. Use these examples to enhance understanding and provide clarity, but ensure all content remains within the existing steps and does not extend beyond them.
Incorporate Class Explanations: Specifically, integrate the detailed "Class Explanations" of categories from the first prompt into the optimized prompt. For each category, introduce implicit clarifications based on relevant data extracted from the context, keeping all additions within the boundaries of the original steps.
End Strictly After Step 5: The optimized prompt must strictly end after step 5. Do not add any additional steps, conclusions, or content beyond this point.

Prompt 16: Optimization Prompt modified from Zadenoori et al.
You are required to enhance and clarify the explanations of the categories in the prompt by integrating illustrative examples and information implicitly referenced in the initial context.
The optimized prompt must follow these strict guidelines:
Maintain the Original Format: The formating in the optimized prompt must remain exactly the same as in the sample prompt; no changes should be made to the formatings' structure or order.
Expand Explanations: Enrich and expand the explanations of each category within the steps, incorporating examples provided. Use these examples to enhance understanding and provide clarity
Incorporate Class Explanations: Specifically, integrate the detailed 'Class Explanations' of categories from the first prompt into the optimized prompt. For each category, introduce implicit clarifications based on relevant data extracted from the context
Enclose your optimized prompt with<prompt></prompt>brackets.
The original prompt is provided below:
'"{original_prompt}

Prompt 17: tree-of-thought prompt by Hulbert [18]
Identify and behave as three different experts that are appropriate to answering this question. All experts will write down the step and their thinking about the step, then share it with the group. Then, all experts will go on to the next step, etc. At each step all experts will score their peers response between 1 and 5, 1 meaning it is highly unlikely, and 5 meaning it is highly likely. If any expert is judged to be wrong at any point then they leave. After all experts have provided their analysis, you then analyze all 3 analyses and provide either the consensus solution or your best guess solution. Give your reasoning enclosed in <think> </think>. The question is…

Prompt 18: Naive Optimization Failure by `llama3.1:8b`
Question: Determine the direct or indirect link between two software development artifacts from the domain of requirements, focusing on connections within the scope of requirementtorequirement traceability.

{source_type}: '''{source_content}'''

{target_type}: '''{target_content}'''

Classify the relationship as:
Direct: a clear and explicit connection between the two artifacts
Indirect via functional requirement: an indirect connection through one or more functional requirements
Indirect via nonfunctional requirement: an indirect connection through one or more nonfunctional requirements
No connection: no direct or indirect link exists between the two artifacts