Karlsruhe Institute of Technology

Automated Prompt Engineering for Traceability Link Recovery

Proposal for 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

Contents

List of Figures

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 aims to link these artifacts across different domains or versions. Challenges such as inconsistent naming are frequent issues in software projects [22]. TLR approaches need to be able to deal with such hardships. Figure 1.1 provides an overview of what these artifacts might look like.

Large language models have made rapid advancements in recent years. They are becoming increasingly popular for dealing with TLR tasks and have shown auspicious results so far. As they are still unable to reason and think on their own [20], prompt engineering is required to extract good results. However, manually determining prompts suited for each specific problem is quite a tedious, time- and labor-intensive task.

To address this shortcoming, automated prompt engineering tools can be used. They will typically use the LLM itself to determine suitable adjustments or generate new prompts based on a description or training data of the problem [18].

The LiSSA framework [6] proposes an environment for TLR tasks, including multiple pipeline steps for preprocessing of input data. They rely on the power of LLMs to extract traceability links from a variety of different artifacts, also across multiple domains. My work will contribute an automatic prompt refinement algorithm to this framework. I am

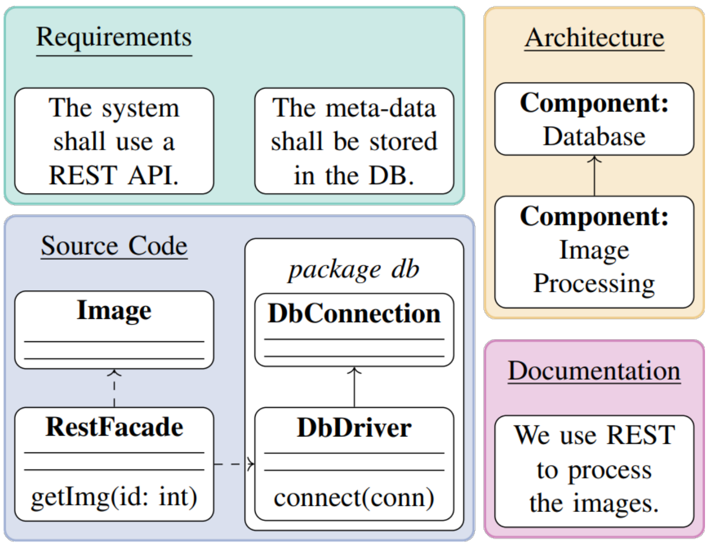

Figure 1.1: Overview of different artifacts during software development by Fuchß et al. [6]

expecting to improve TLR, especially for larger projects in the requirements-to-requirements domain.

2 Foundations

To understand the application of automated prompt engineering for traceability link recovery, some key concepts are quite important. They will be explained in the following subsections.

2.1 Definition of Traceability Link Recovery

Traceability is «the ability to find or follow something» [21], according to the Cambridge Dictionary. Meaning, there is evidence of some past occurrence. The task of traceability link recovery in software engineering is to find instances of the same element across different software artifacts and link them for further processing. What these elements might look like was visualized in Figure 1.1. 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. To find all or most traceability links (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. [22], 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 looking for TLs. However, for automated trace link recovery, this means that simple string comparisons by name are insufficient. While large language models can overcome these inconsistencies, finding suitable prompts might be challenging or include tedious manual work in prompt engineering.

2.2 Definition of 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 [25].

Typically, APE processes are iterated to improve previous results further [17, 25, 26, 16]. An illustration of the core concept and how it can be implemented will be shown in section 4.2 and Figure 4.2.

In general, 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. This approach is very intuitive. However, human interaction is still required to choose this first prompt. Zhou et al. [26] 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 performance of prompts can be estimated using a set of training data, which consists of 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.

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

2.3 Automatic Prompt Optimization Using Gradient Descent

Pryzant et al. [17] propose the Prompt Optimization with Textual Gradients (ProTeGi) algorithm. This entire section is based on their work.

In contrast to the general iterative automatic prompt engineering in section 2.2, they create multiple improved prompts after each step of the optimization during a single iteration. This is also illustrated in Figure 2.1. 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 will be 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 $\{(x_1, y_1), \dots, (x_n, y_n)\}$ 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» [17, 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.

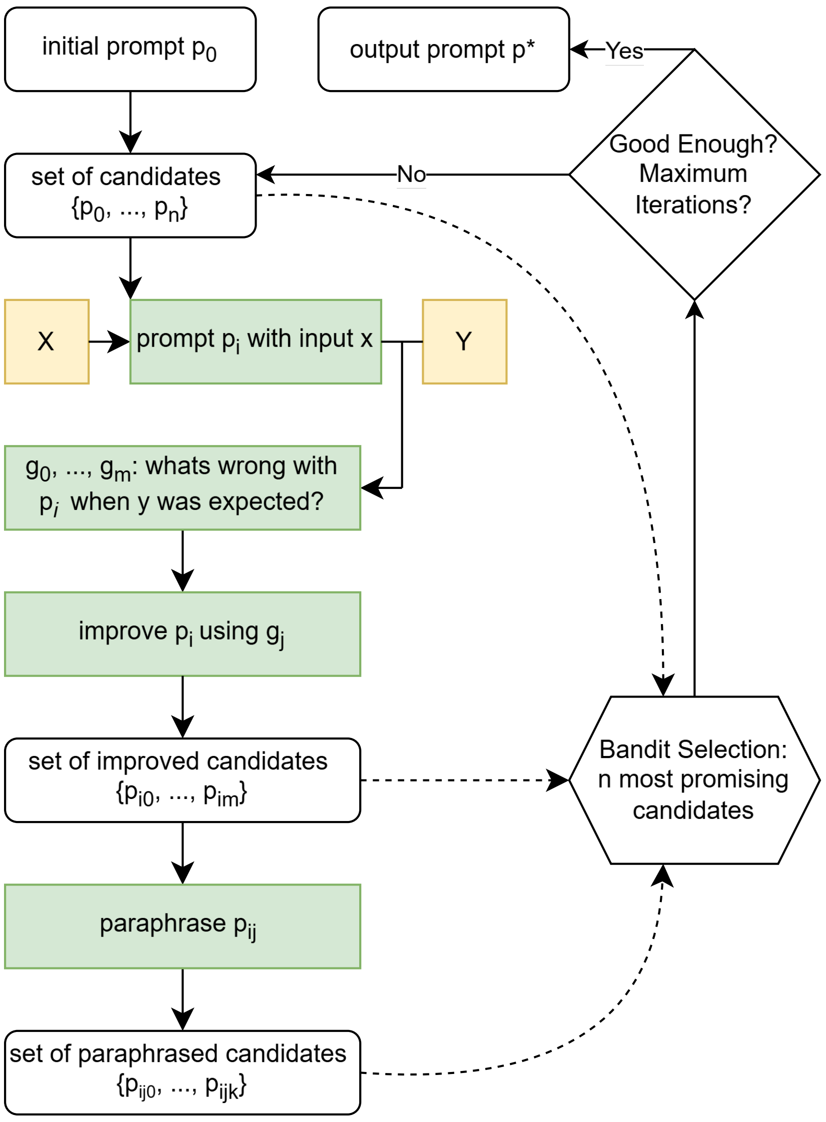
initial prompt p_0
output prompt p*
Yes
Good Enough? Maximum Iterations?
No
set of candidates $\{p_0, ..., p_n\}$
X
prompt p_i with input x
Y
$g_0, ..., g_m$: whats wrong with p_i when y was expected?
improve p_i using g_j
set of improved candidates $\{p_{i0}, ..., p_{im}\}$
Bandit Selection: n most promising candidates
paraphrase p_{ij}
set of paraphrased candidates $\{p_{ij0}, ..., p_{ijk}\}$

Figure 2.1: Overview of the iterative optimization loop in [17]

2.3.1 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.

2.3.2 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 [14]. 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» [17, 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 through large language models is an active field of work to apply rapid advancements in large language model (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.

3.1 Trace Link Recovery

Before the emergence of large language models information retrieval (IR) methods were used to recover traceability links. De Lucia et al. [3] have provided an overview about different IR methods and their performance.

More recently, in work by Hey et al. [8] they tried to overcome the semantic gap, when two instances of the same artifact are described using differing semantics, by exploring word embedding. Keim et al. [12] have proposed cross-domain trace link recovery using agent-like machine learning algorithms.

In previous work by Fuchß et al. [6], simple prompts by Ewald [4] were used to recover trace links between software documentation and architectural diagrams. The work of Ewald adapts prompts studied by Rodriguez, Dearstyne, and Cleland-Huang [19]. These prompts were 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) domains.

Hey et al. [10] have used these same prompts for TLR in the domain of requirements to requirements. While they were also able to outperform state-of-the-art approaches, the recall rate, especially for larger data sets, still has room for improvement.

Rodriguez, Dearstyne, and Cleland-Huang [19] have also evaluated LLM usage for trace link recovery. Their main takeaway was that even minor adjustments, «such as pluralizing words, interchanging prepositions, or reordering phrases» [19, sec. VI] lead to major differences in the outcome. They were 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.

My work will build on these previous developments. I aim to improve recovery rates 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 [18]. Data sets of these pairs, describing the problem to be solved using large language models, 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. [26] 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.

Self-Refine by Madaan et al. [16] takes feedback from the same large language model that generated the prompt, to improve it further. This imitates human behavior when initial drafts are adjusted rapidly.

ProTeGi by Pryzant et al. [17] utilizes a gradient descent algorithm to find the minimal deviance between an optimized prompt and the expected outputs. More details about their work can be found in 2.3.

Yang et al. [23] have taken a slightly different approach in their OPRO 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 (DSPy) Framework by Khattab et al. [13] 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 will focus on the optimization algorithm by Pryzant et al. While my work will not expand the field of automatic prompt engineering directly, I will adapt it into the domain of traceability link recovery. Existing training sets can be used to optimize the prompts.

3.3 Training Data for different Domains

Trace Link Recovery tasks can be applied across many domains. Often, authors will focus on one or a few domains instead of attempting to cover everything. Domain specific training data includes the artifacts and, ideally, also a gold standard to compare results to.

Software architecture documentation to software architecture models for BigBlueButton, MediaStore, Teammates, and Teastore are publicly available by Fuchß et al. [5] GitHub.

For the domain of requirements to requirements, which my work will also focus on, a compilation of training data can be found in the replication package [9] for recent work of Hey et al.

4 Approach

In this chapter, I will outline how my thesis will be realized and provide source code that can be merged into the LiSSA project. My work aims to implement three different classifiers into the framework. First, I plan to implement a simple tree-of-thought classifier in section 4.1 to broaden the baseline to compare later optimized prompts against. Next, I expect to work on a reduced iterative optimizer in section 4.2 using a naive approach to optimization. After this already provides basic components and data to evaluate the more sophisticated algorithm by Pryzant et al. [17] will be implemented in section 4.3. Last but not least, I will explain how I plan to evaluate and compare my findings in section 4.4.

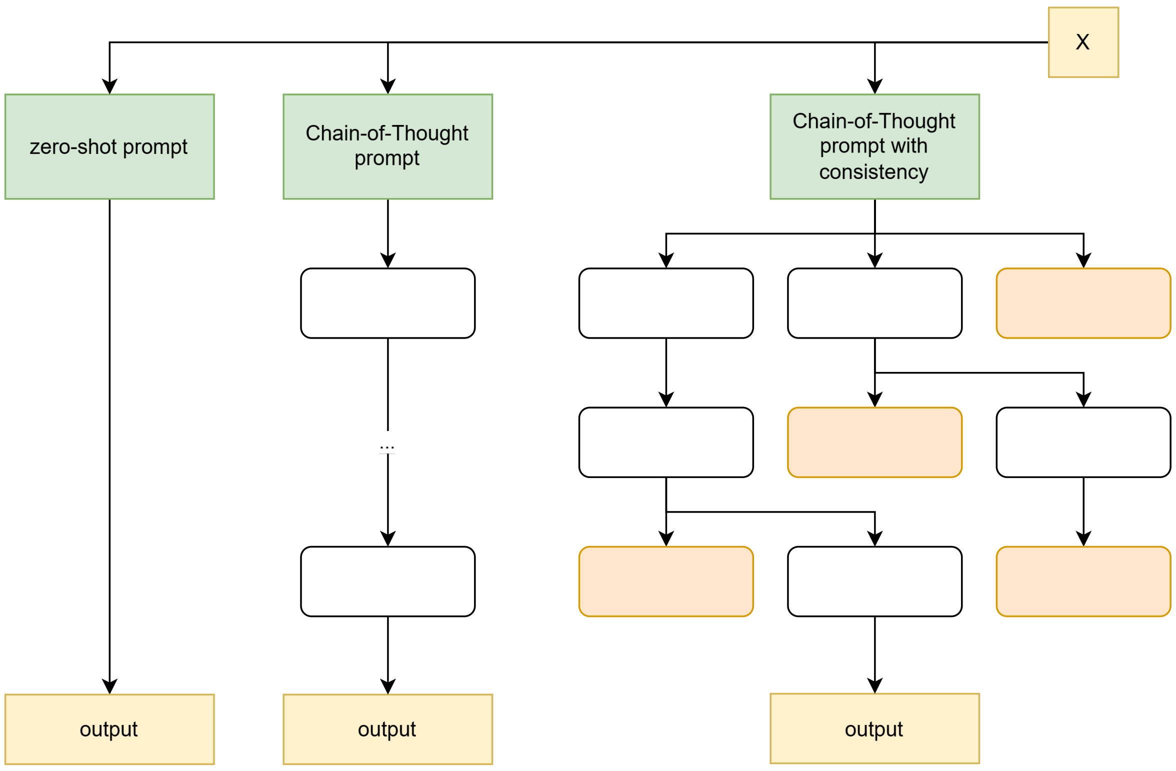

Figure 4.1: Tree-of-thought (ToT) visualization inspired by Yao et al. [24]

4.1 Simple Tree-of-Thought Classifier

Tree-of-thought (ToT) is a prompting technique exploring more different branches of thoughts than typical chain-of-thought (CoT) prompts. As illustrated in Figure 4.1, many branches are explored concurrently instead of following a single strain of thoughts like CoT prompting. The prompts by Hulbert [11] aim to simplify ToT prompting into a singular request, instead of chaining multiple calls. I will use one of their prompts. The benchmark data to compare ToT prompting with the existing zero-shot and CoT classifiers also exists in public repositories [5, 9].

The LiSSA framework is publicly accessible in a repository using the MIT license. Various classifiers are already included there. They can be used as an inspiration for development. The class `classifier` is an abstraction of all classifiers. It provides basic functionality to include further classifiers. Integration for different large language model (LLM) providers, such as Chat-GPT, Ollama, and Blablador, already exists in the framework. The new classifier can be plugged in directly with little more effort than writing the request prompt and implementing the `classify` function to extract the classification result out of the LLM output.

This work package has two major goals. Foremost, to get confident and set up with the general project structure and benchmark data. This will be a prerequisite for all future work in the LiSSA framework. Second, this will broaden the baseline to compare later results and possible improvements against.

4.2 Naive Iterative Optimizing Classifier

Another classifier implementation will be the simplest automatic iterative prompt optimization algorithm. The same existing abstract `classifier` class can be used as in subsection 5.1.1 to provide basic functionality for the iteratively optimized classifier.

Next, a function f will be required to quantify the result, also referred to as performance, of the current prompt p iteration, called to a large language model llm using a set of training data X. The simple approach here is to use the successful retrieval rate by dividing the number of correctly classified trace links by the total number of recoverable trace links. The function f will need to have the same range for any set of training data to ensure the threshold value t will always be viable.

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

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

The benchmark data by Hey et al. [9] also provides the gold standard for each given problem. The total amount of recoverable trace links is thus already available.

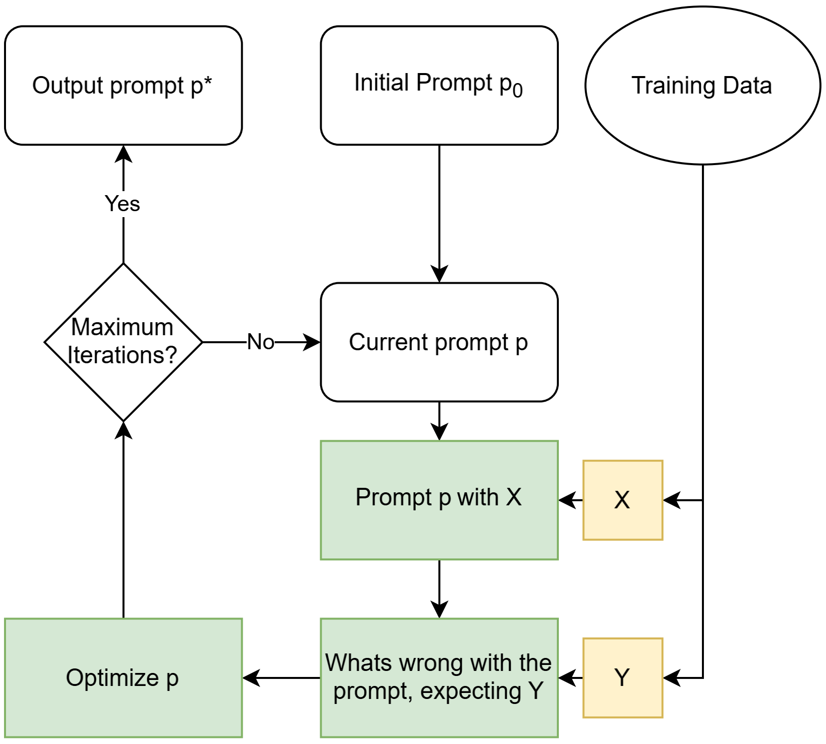

Figure 4.2: Visualization of a simple iterative optimization algorithm

To improve the performance, an optimization prompt is used. As visualized in Figure 4.2 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 about why the prompt did not meet the requirements. The hopefully improved prompt will be taken into the next iteration. The act of evaluation and improving will be repeated until a threshold value is passed, meaning the prompt is considered good enough. To limit resource usage, a hard limit will also be included to put an upper bound on the optimization attempts.

Implementing this naive, iteratively optimized classifier will yield fast results to evaluate later, even if more complicated implementations might fail. The core concepts also apply to subsection 5.1.3 and can be reused.

4.3 Automatic Prompt Optimization with Gradient Descent

The existing iterative optimization implementation from subsection 5.1.2 can be used as a foundation for the implementation, as the core concepts remain the same. Refer to section 2.3 for details of the gradient-descent-based optimization algorithm.

Pryzant et al. [17] have provided their Python source code in a publicly accessible repository[1] under the MIT license. The project code can be of great help when adapting their algorithm to the LiSSA Java framework. The actual implementation of gradient descent and the bandit-selection algorithm will be the main focus, in order to steer optimized prompts in the right direction.

The primary idea of this thesis is to explore automated optimization. The naive approach may yield good results already. Feeding more information and processing logic into the optimization system is expected to present better results. Especially suitable results may be reached faster by doing more processing on the machine instead of expensive API calls to the large language model.

4.4 Evaluation

Evaluating the experimental results of prompt optimization is central to making them accessible and comparable. Precision (Equation 4.3) and recall (Equation 4.4) are key measures for information retrieval tasks [7]. They are commonly used by many authors to compare different approaches to traceability link recovery. Further the $f1 - score$ and $f2 - score$, using a fixed β in Equation 4.5, are used often. I will use these four measures. Hey et al. [10] and Fuchß et al. [6] have used these when evaluating their results on the same data sets I intend to use. This way comparrision will be very simple.

«Precision measures the ratio of correctly predicted links and all predicted links. Recall measures the ratio of correctly predicted links and all relevant documents. It shows an approach's ability to provide correct trace links. The F1 score is the harmonic mean of precision and recall. The Metrics are calculated in the following way, where TP is true positives, FP is false positives, and FN is false negatives.» [4, sec. 7.1]

$$Precision = \frac{TP}{TP + FP} \tag{4.3}$$

$$Recall = \frac{TP}{TP + FN} \tag{4.4}$$

$$f_\beta = (1 + \beta^2) \cdot \frac{Precision \cdot Recall}{(\beta^2 \cdot Precision) + Recall} \tag{4.5}$$

Depending on how long each implementation phase will take, it is possible to generate a bunch of different data and performance comparisons. A simple yet interesting approach is to compare different LLMs for the task. Many variations can be achieved by comparing, for example, how a prompt optimized by one system performs on the others. We can also compare how well each system manages to optimize the initial prompt. Another interesting thought is to take optimized prompts from a different system as the initial prompt.

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

Varying testing data is also plausible. Including artifacts and their links from different projects into a singular data set might have the potential to yield more general but less optimized prompts. The exact scope of the evaluation remains to be seen. It will be dynamically adjusted when actual implementations exist and are ready to test.

5 Work Plan

In this chapter, I will elaborate on my plan to realize the thesis topic and mitigate risks. This includes the individual work packages outlined in section 5.1. Different artifacts that will be produced through the thesis are mentioned in section 5.2. Some expected risks during the project, along with their mitigation, will be addressed in section 5.3. Lastly, a Gantt chart will be provided to visualize the temporal planning for my thesis project in section 5.4.

5.1 Phases

My work will be split into four distinct phases, with only a few dependencies in between. Each of the first three implementation packages can be evaluated separately. However, each phase provides building blocks used by the next with increasing complexity. Even though Evaluation and Buffer is placed as the last phase, some progress will already be made after the first implementation in subsection 5.1.1 is finished.

5.1.1 Initial Overview

To become familiar with the LiSSA framework [6], I will implement another basic classifier with a singular request variation of the simple prompting technique tree-of-thought (ToT) [15] explained in the section Simple Tree-of-Thought Classifier.

Currently, a zero-shot and chain-of-thought (CoT)-style reasoning classifier is implemented. However, as Long has shown, ToT-style prompts can improve performance compared to CoT approaches when solving logic puzzles. They have tested their approach to ToT prompting with different-sized Sudoku puzzles and compared their results against zero-shot prompting as well as CoT styled one- and few-shot prompting. This will provide an additional baseline on the possibilities without automatic prompt optimization.

5.1.2 Naive iterative Optimization

Next, I plan to implement a naive iterative approach to prompt optimization. Many automatic prompt optimization algorithms [17, 25, 26, 16] depend on an iterative core loop (see Figure 4.2). This loop will be repeated until the optimized prompt performs sufficiently good or a maximum number of iterations has been reached. My plan to realize this is explained in the section Naive Iterative Optimizing Classifier.

5.1.3 Automatic Prompt Optimization Based on Gradient Descent

The major implementation of this thesis will be an adaptation of the gradient-descent-based automatic prompt optimization (APO) algorithm by Pryzant et al. [17]. Refer to section 4.3 for details about my implementation or section 2.3 for a more general overview of their algorithm. The APO algorithm will be trained and tested on the requirements-to-requirements benchmark data by Hey et al. It is available in their replication package [9]. The optimized prompt will be compared against previous prompts used by Fuchß et al. and Hey et al. in the greater LiSSA framework.

5.1.4 Evaluation and Buffer

As this work can be seen as an expansion on the recent work of Hey et al. [10], the same metrics will be used to evaluate the different prompts used and optimized in this work. These are the precision, recall, $f_1 - score$, and $f_2 - score$. This enables an easy comparison, especially with the manual prompts designed by Ewald [4] included in the work of Hey et al. For further evaluation, aside from the performance of the optimized prompts, different large language models will be used, as explained in section 4.4.

5.2 Artifacts

This work will yield code to be merged into the LiSSA repository[1]. The implementation during the work packages Naive iterative Optimization and Automatic Prompt Optimization Based on Gradient Descent in section 5.1 will provide a base to implement further automatic prompt optimization techniques with similar building blocks.

Also, a written report, my bachelor's thesis, will be created to document, summarize, and evaluate the results of this automatic prompt optimization work to retrieve trace links.

5.3 Risk Management

The major risk of my work would be failing to implement the Automatic Prompt Optimization Based on Gradient Descent in subsection 5.1.3. By choosing an optimization algorithm that has Python source code publicly available, this risk is already greatly reduced. Furthermore, all three work packages with implementation content will yield a classifier that can be tested and evaluated independently against the existing classifiers in the LiSSA Framework. Thus, early working examples will be available throughout the thesis.

It is hard to estimate the required time for each phase accurately. Frequent buffer slots are planned to allocate more time dynamically as needed.

[1]https://github.com/ArDoCo/LiSSA-RATLR/

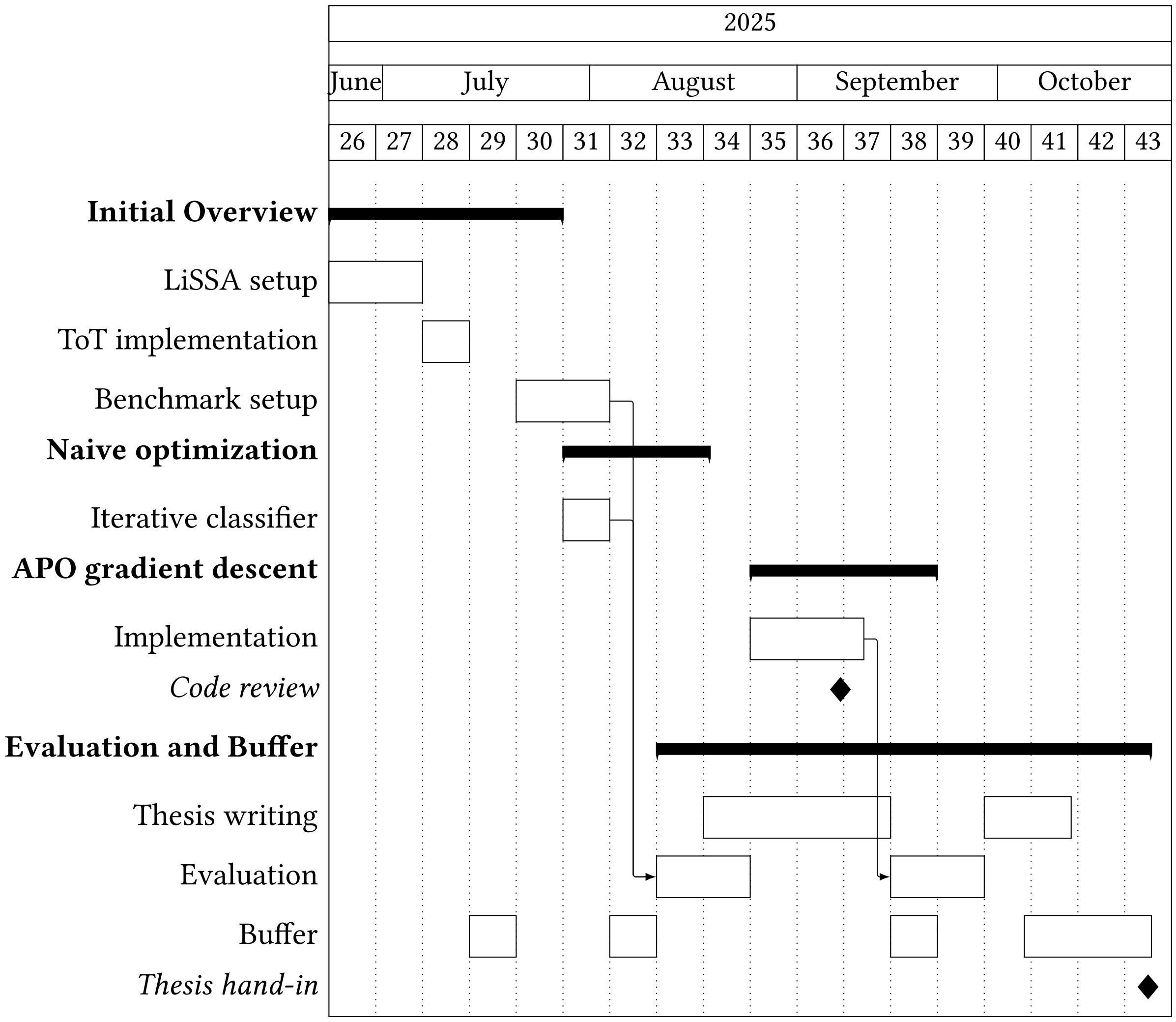

Figure 5.1: Gantt chart for thesis schedule

5.4 Schedule

The Gantt chart in Figure 5.1 illustrates how I plan to allocate my time. The early months will probably take up less time than planned, which can be used to reduce congestion in later phases. As mentioned in section 5.3, the work packages are designed with early working prototypes in mind. The major dependencies, as visualized, are that evaluation requires implementation. Even though there are multiple packages with a planned implementation, they can all be evaluated separately and independently. There is also a code review scheduled during the likely longest implementation section in the Automatic Prompt Optimization Based on Gradient Descent phase.

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] 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).

[4] 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).

[5] 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).

[6] 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).

[7] 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).

[8] 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).

[9] 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).

[10] 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`.

[11] 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).

[12] Jan Keim et al. “Trace Link Recovery for Software Architecture Documentation”. In: *Software Architecture*. Ed. by Stefan Biffl et al. Cham: Springer International Publishing, 2021, pp. 101–116. ISBN: 978-3-030-86044-8. DOI: `10.1007/978-3-030-86044-8_7`.

[13] 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).

[14] 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.

[15] 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.

[16] 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/91edff07232fb1b55a505a9e9f` `Abstract-Conference.html` (visited on 05/20/2025).

[17] 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).

[18] 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.

[19] 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).

[20] 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.

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

[22] 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).

[23] 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.

[24] 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.

[25] 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, 2025, pp. 217–225. ISBN: 978-3-031-88531-0. DOI: 10.1007/978-3-031-88531-0_15.

[26] 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.